Interaction Design in Service Compositions

Teduh Dirgahayu

Interaction Design in Service Compositions

Teduh Dirgahayu

Enschede, The Netherlands, 2010

CTIT Ph.D. Thesis Series, No. 10-172
SIKS Dissertation Series, No. 2010-34

CTIT Ph.D.-Thesis Series, No. 10-172
Centre for Telematics and Information Technology, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

SIKS Dissertation Series, No. 2010-34
The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN 978-90-8891-188-0
ISSN 1381-3617 (CTIT Ph.D.-Thesis Series, No. 10-172)

Published by Uitgeverij BOXPress, Oisterwijk
Printed by Proefschriftmaken.nl || Printyourthesis.com

Cover photo "Conversation between Kresna and Arjuna" by Teduh Dirgahayu

INTERACTION DESIGN
IN SERVICE COMPOSITIONS

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 vrijdag 10 september 2010 om 13.15 uur

door
Teduh Dirgahayu
geboren op 22 juni 1974
te Yogyakarta, Indonesië

Dit proefschrift is goedgekeurd door:
prof.dr.ir. C.A. Vissers (promotor), dr.ir. M.J. van Sinderen (assistent-promotor) en
dr.ir. D.A.C. Quartel (assistent-promotor)

Abstract

This thesis proposes a concept and transformations for designing interactions in a service composition at related abstraction levels. The concept and transformations are aimed at helping designers to bridge the conceptual gap between the business and software domains. In this way, the complexity of an interaction design can be managed adequately.

A service composition is specified as one or more interactions between application components. Interaction design is therefore the central activity in the design of a service composition. Interaction design at related abstraction level requires an interaction concept that can model interactions at a higher abstraction level (called *abstract interactions*) and interactions at a lower abstraction level (called *concrete interactions*), in order to avoid any conceptual gap between abstraction levels.

An *interaction* is defined as a unit of activity that is performed by multiple entities or participants in cooperation to establish a common result. Different participants can have different views on the established result. The possible results of an interaction are specified using contribution constraints and distribution constraints. Contribution constraints model the responsibility of the participants in the establishment of the interaction result. Distribution constraints model the relation between the participants' views. An interaction provides mutual synchronisation or time dependency between the participants. This interaction concept can model abstract and concrete interactions. A designer can hence use a single interaction design concept during a design process.

Two design transformations are defined, namely *interaction refinement* and *interaction abstraction*. Interaction refinement replaces an abstract interaction with a concrete interaction structure. Interaction abstraction replaces a concrete interaction structure with an abstract interaction. A set of conformance requirements and a conformance assessment method are defined to check the conformance between an abstract interaction and concrete interaction structure.

In an interaction design process, a designer first represents a service composition as an abstract interaction that specifies the desired result. This abstract interaction is then refined into a concrete interaction structure that specifies how to establish that result. Interaction refinement can be done recursively until it results in a concrete interaction structure that can be mapped onto available interaction mechanisms. Every refinement is followed by conformance assessment.

To facilitate the development process of a service composition, this thesis provides

- patterns for interaction refinement, which serve as guidelines on possible refinements of an abstract interaction;
- abstract representations of interaction mechanisms, which allow interaction mechanisms to be included in an interaction design at a higher abstraction level; and
- a transformation tool to transform an interaction design at an implementation level to an executable implementation.

The use of the interaction concept, design transformations, patterns for interaction refinement, abstract representations of interaction mechanisms, and transformation tool are illustrated with two case studies. In the first case study, we design a travel reservation application as a service composition using a top-down design approach. The services and application components that are involved in the service composition have to be developed. In the second case study, we design enterprise application integration for an order management that composes existing services and applications. We follow an integration approach and use our interaction concept during the design process. The obtained integration solution is then transformed to an executable implementation using our transformation tool.

Acknowledgements

During the development of this thesis, many people supported and contributed to my work. For them, I would like to express my gratitude here.

First of all, I would like to thank my promotor Chris Vissers and my assistant promotors Marten van Sinderen and Dick Quartel for their supervision. Also, I would like to thank Remco Dijkman, who formulated the initial ideas of my research.

I would like to thank the members of my graduation committee: prof.dr. Colin Atkinson, prof.dr. Peter Linington, prof.dr.ir. Roel Wieringa, and prof.dr. Jos van Hillegersberg. It is an honour to have you in this committee.

I would like to thank the people of the A-MUSE project for the collaboration during my research. Also, I would like to thank my colleagues in the ASNA and IS groups for providing very pleasant working environment. Special thanks should go to the secretaries of these groups: Annelies and Suse for making administrative work much easier.

It was not easy to have a baby while writing a PhD thesis. Luckily, I have so many good friends that helped me and my family during that period: David and Vince, Arie and Meli, Pablo and Flavia, Stanislav and Vania, Meti, Lizda, Oma Dewi, Oma Wil, all the PPI- and IMEA-members, and all other friends that I cannot mention their names here one by one.

I would like to thank my parents: almarhumah Ibu, Bapak, Ibuk, and Ayah; and my brothers and sisters for their constant support, love, and pray.

Finally, I would like to thank my wife Emma for her company, support outside of the office, and belief in my success. To my little angel Muzhda, thank you for being the reason for moving forward in my life.

Teduh Dirgahayu
Yogyakarta, July 2010

Contents

Abstract v

Acknowledgements vii

Chapter 1. Introduction 1

1.1 Background 1
1.2 Motivation 4
1.3 Objectives 9
1.4 Scope 10
1.5 Approach 11
1.6 Outline of the thesis 12

Chapter 2. Analysis of interaction design concepts and methods 15

2.1 Design concepts and design language 15
2.2 Design concepts and design methods 17
2.3 Framework for suitability analysis 18
2.4 Suitability analysis 23
2.5 Examples of interaction designs 38
2.6 Concluding remarks 43

Chapter 3. Design concepts for interaction modelling 45

3.1 Basic design concepts 46
3.2 System perspectives 50
3.3 Concepts for behaviour modelling 53
3.4 Abstract interaction modelling 67
3.5 Enhanced interaction concept 70
3.6 Relationships between behavioural concepts 76
3.7 Shorthand notations 77
3.8 Concluding remarks 80

Chapter 4. Interaction design transformations 83

4.1 Behaviour refinement 84
4.2 Behaviour abstraction 88
4.3 Refinement of an action into an interaction........ 99
4.4 Strategy for interaction refinement........ 100
4.5 Interaction refinement........ 101
4.6 Conformance assessment 107
4.7 Patterns for interaction refinement 114
4.8 Related work 124
4.9 Concluding remarks........ 125

Chapter 5. Abstract representations of interaction mechanisms 127

5.1 Motivation........ 127
5.2 Approach........ 129
5.3 Abstractions of interaction mechanisms........ 133
5.4 Example of use 150
5.5 Related work 152
5.6 Concluding remarks........ 153

Chapter 6. Transformation to executable implementations 155

6.1 Service compositions in Web Services 155
6.2 Approach........ 160
6.3 Modelling restrictions 163
6.4 Pattern recognition........ 169
6.5 Constraint transformation........ 172
6.6 Model realisation 182
6.7 Related work 190
6.8 Concluding remarks........ 191

Chapter 7. Case study: travel reservation application 193

7.1 Case description 193
7.2 Design process 1........ 195
7.3 Design process 2........ 207
7.4 Discussion 214
7.5 Evaluation........ 216

Chapter 8. Case study: enterprise application integration 219

8.1 EAI approach........ 219
8.2 Case description 221
8.3 Integration solution 223
8.4 Discussion 227
8.5 Evaluation........ 228

Chapter 9. **Conclusions** **231**
9.1 General conclusions ... 231
9.2 Research contributions ... 233
9.3 Directions for further research ... 235

Appendix A. **Conformance assessments in case study 1** **237**
A.1 Design process 1 ... 237
A.2 Design process 2 ... 249

References **253**

Publications by the author **265**

SIKS Dissertation series **267**

Chapter 1

Introduction

This thesis proposes a concept and transformations for designing interactions in the development of a service composition. To facilitate the development of a service composition, this thesis also provides abstract representations of interaction mechanisms and a transformation tool to transform an interaction design into an executable implementation. The concept, transformations, abstract representations, and tool aim at enabling and encouraging designers to design interactions at related abstraction levels. In this way, the business requirements of a service composition can be transformed correctly to a software application. Also, the complexity of an interaction design can be managed adequately.

This chapter presents the motivation and objectives of this thesis as well as the approach followed. It is organised as follows: Section 1.1 presents background information, Section 1.2 provides the motivation, Section 1.3 defines the objectives, Section 1.4 defines the scope, and Section 1.5 presents the approach followed in the research. Section 1.6, finally, presents the structure of the remainder of this thesis.

1.1 Background

A distributed application is an application that is composed from a number of application components that interact with each other. Typically, these application components are distributed over different computing machines at different locations, connected with each other via a communication network. Distributed applications range from enterprise applications, e.g., e-commerce, enterprise application integration, and computer-supported cooperative work, to personal applications, e.g., e-mail and instant messengers. Distributed applications facilitate many activities of modern life.

In the development of a distributed application, a paradigm called *service-oriented computing* [55, 100] has been widely accepted and is now gaining popularity. In this paradigm, an application component exposes its external functionality without revealing its internal functions and structures. Application components interact with each other to deliver a service.

A service is the establishment of some valuable effect through the interaction between two or more application components [113].

In a service, one application component plays the role of service user while the other application components play the role of service providers. A service user requests a service from one or more service providers. A service provider offers a service to a service user. These partial definitions of the service are called the *requested service* and the *offered service*, respectively.

Services can be composed into a service composition [55, 100, 127].

A service composition is a composition of services to deliver a new service.

In service oriented computing, a distributed application is developed as a service composition by reusing existing services. Development of distributed applications by reuse promises less development cost and shorter time-to-market [50, 122].

A service composition is specified as one or more related interactions between application components. Therefore, designing those interactions and their relations is the central activity in the design of a service composition. It deals only with the external functionality of the application components. This activity results in an interaction design.

An interaction design is a design that describes interactions between application components.

A service composition can be a choreography or orchestration [21, 103]. A choreography defines a set of related interactions between application components to achieve a common goal. The business logic of the choreography is distributed over the application components. Figure 1-1 illustrates a choreography between inventory and manufacture services. The common goal of this choreography can be the completion of a production order. This figure uses an intuitive graphical notation. A rounded rectangle represents an application component. A bidirectional arrow represents an interaction in terms of message exchanges.

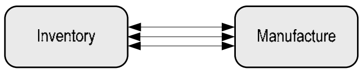


Figure 1-1
A choreography between an inventory and manufacture services

An orchestration defines the offered service of a service provider as interactions between a coordinator and other service providers. The business logic of the orchestration is centralised in the coordinator. The coordinator coordinates interactions between the service user of the service provider for which the orchestration is defined and the service providers that are parts of the orchestration. Figure 1-2 illustrates a travel agent as an orchestration. The travel agent is composed of a coordinator, hotel service provider, and airline service provider.

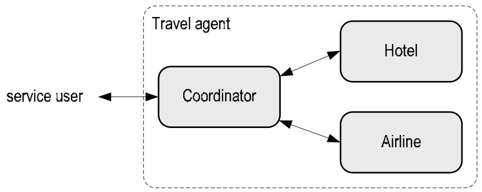


Figure 1-2
A travel agent as an orchestration

Problem description

A service composition can support an organisation's business. In the design process of such a service composition, a designer plays the role of *business analyst* or *application designer*. A business analyst analyses and elicits requirements for a business process and recommends a business process that satisfies those requirements [56]. This business process is specified as an interaction design. An application designer designs a software application that implements the interaction design specified by the business analyst.

The different sets of concepts in the business and software domains create a conceptual gap between the domains. This gap can mean that a business process is not correctly implemented as a software application [17, 49]. To bridge this gap, the business analyst and application designer should collaborate, to some extent, with each other [49, 62]. Such collaboration can use related abstraction levels as illustrated in Figure 1-3. At a certain abstraction level, e.g., $n+1$, the business analyst and application designer work together to develop interaction design $D1$.

A proper design method is necessary to guide the development process of a service composition through these related abstraction levels. In this development process, an interaction design at an abstraction level is refined or abstracted into another interaction design at another abstraction level. The design method should have a correctness mechanism to ensure that a refinement or abstraction results in a correct interaction design.

To avoid any conceptual gap between abstraction levels, the design method should use the same set of design concepts at all abstraction levels. This would also facilitate the development of a correctness mechanism.

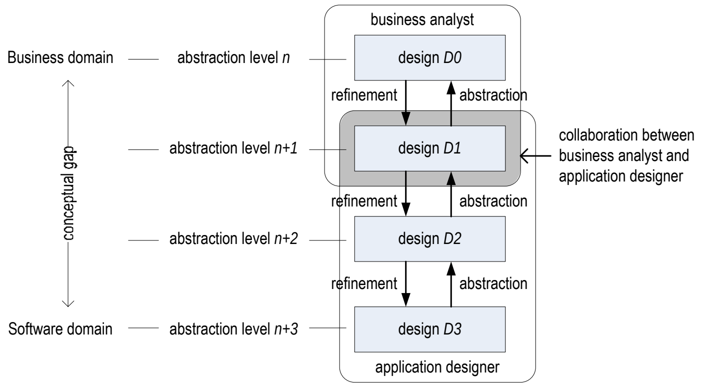


Figure 1-3
Collaboration between a business analyst and application designer to bridge the conceptual gap

Several methods for designing service compositions have been proposed, e.g., in [15, 32, 33, 41, 51, 65, 76, 81, 108, 113, 117, 125, 143, 145]. Our analysis [37], later presented in Chapter 2, shows that these design methods use design languages whose interaction design concepts are not suitable for modelling interactions at higher abstraction levels. Most of the interaction design concepts represent interaction mechanisms that are provided by communication middleware. Such an interaction design concept forces a designer to develop interaction designs at an implementation level. All interactions have to be represented in terms of middleware interaction mechanisms.

Designing a complex service composition at an implementation level, though, results in an interaction design that reveals the complexity of its intended implementation. This has several disadvantages as follows.

– The interaction design is difficult to create because the designer has to define a service composition that satisfies business and implementation requirements at the same time. A complex interaction design is prone to design errors.
– The interaction design is difficult to modify when some implementation requirements change. It offers no implementation alternative.

1.2 Motivation

Designing a service composition at higher abstraction levels can bridge the conceptual gap between the business and software domains. It also helps the designer to manage the complexity of a service composition and to overcome the disadvantages mentioned in Section 1.1. We adopt two design approaches: *related abstraction levels* and *the MDA approach*. We identify research questions associated with these approaches, which need to be answered to allow the design of service compositions at higher abstraction levels. Those questions motivate us to do the research.

1.2.1 Related abstraction levels

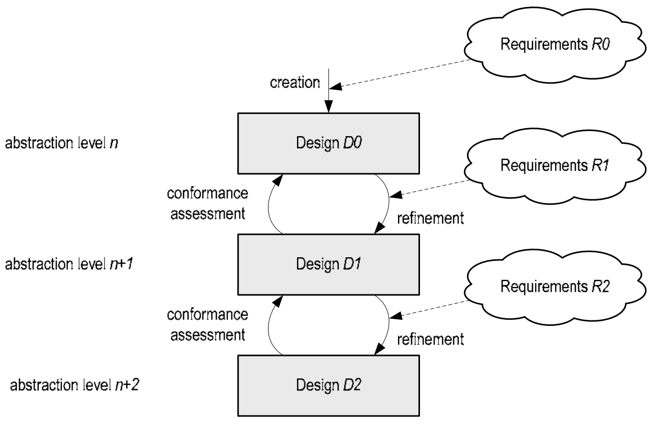
In a design process that uses related abstraction levels, a design at a certain abstraction level is transformed into a design at a lower or higher abstraction level. The transformation is called *refinement* or *abstraction*, respectively.

In this thesis, the terms *abstract* and *concrete* are used to denote a higher and lower abstraction level, respectively, without referring to particular abstraction levels. The notion of higher and lower abstraction levels is relative, i.e., an abstraction level *n* is lower than an abstraction level *n–1* and higher than an abstraction level *n+1*.

A step-wise refinement is a design process in which an abstract design is successively refined into more concrete designs. During refinement, the designer gradually includes solutions that satisfy business or implementation requirements, or defines these solutions in more detail. This approach reveals design complexity in a controlled way, i.e., the design complexity gradually increases from an abstract design to concrete designs.

Figure 1-4 illustrates a step-wise refinement in an interaction design process. At a higher abstraction level *n*, an interaction design *D0* is created to satisfy initial requirements *R0*. This interaction design is refined into another interaction design *D1* at a lower abstraction *n+1* to satisfy requirements *R1*. Interaction design *D1* can be further refined into another interaction design *D2* at another lower abstraction level *n+2* to satisfy requirements *R2*.

Figure 1-4
Step-wise refinement



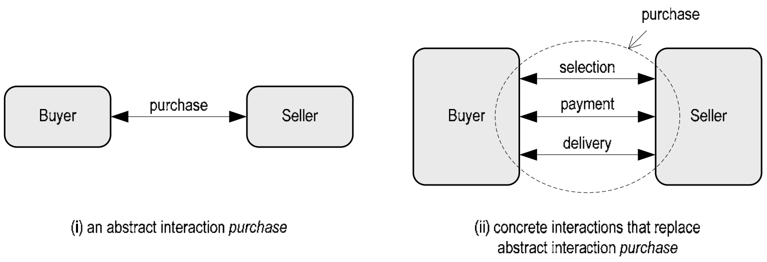
A concrete design is a correct refinement of an abstract design if it preserves the design information defined in the abstract design, while it defines additional design details that does not conflict with the abstract design. Such a concrete design *conforms* to an abstract design. There are two alternatives to obtain a conforming concrete design. In the first alternative, a concrete design is defined by applying well-considered refinement rules. These rules guarantee that the concrete design conforms to its abstract design. This alternative, however, limits the designer's freedom in defining a concrete design. In the second alternative, a concrete design is defined without applying refinement rules and then is checked whether it conforms to an abstract design. This "trial-and-error" alternative gives the designer more freedom in defining a concrete design. In this alternative, every refinement must be followed by a conformance assessment [44, 107, 110].

This design process can be further continued until it results in an interaction design at an implementation level that can be mapped onto available interaction mechanisms.

In our design approach, every interaction design is developed as a complete design. An abstract design is complete when it addresses and satisfies the requirements that are essential at the abstraction level considered. A concrete design is complete by preserving the design information defined in an abstract design and by satisfying requirements that result from specific implementation choices.

Figure 1-5 depicts an example of abstract and conforming concrete interaction designs. Figure 1-5(i) represents the purchase of a product between a buyer and seller as a single abstract interaction *purchase*. This interaction should specify the essential properties of the interaction, which we define as follows: when the interaction is completed, a product must have been selected, delivered, and payed for. Since this interaction cannot be directly mapped onto available interaction mechanisms, e.g., message-passing communication or request-response operation, this interaction should be replaced with conforming concrete interactions.

Figure 1-5
Examples of abstract and concrete interactions


Buyer
purchase
Seller
(i) an abstract interaction *purchase*
purchase
Buyer
selection
payment
delivery
Seller
(ii) concrete interactions that replace abstract interaction *purchase*

In Figure 1-5(ii), three related concrete interactions, namely *selection*, *payment*, and *delivery*, replace the abstract interaction *purchase*. The relations between these concrete interactions define the order in which the interactions should be performed. (The relations are not shown in the figure.) Further refinement is required since these interactions cannot be directly mapped onto available interaction mechanisms either.

Designing interactions at related abstraction levels produces a sequence of interaction designs of the same service composition; each of these has a different degree of complexity. Different interaction designs serve different purposes. For example, an abstract interaction design can be used in an analysis in the business domain. A concrete interaction design with full implementation details is used as a reference for an executable implementation.

Designing interactions at related abstraction levels gives the following benefits.

– An abstract interaction design is easier to understand and to analyse in its business domain. Implementation details are decided and elaborated in concrete interaction designs.
– When some implementation requirements change, it affects only concrete interaction designs. Abstract interaction designs remain intact and can be refined again into concrete interaction designs that satisfy the changes.
– Alternative implementations can be developed based on the same abstract interaction design to satisfy alternative implementation requirements.

1.2.2 Model-Driven Architecture approach

The Model Driven Architecture (MDA) approach [90, 91] has been widely accepted for designing distributed applications and has also been applied in the development of service compositions [14, 23, 35, 48, 74, 98].

The MDA approach distinguishes three types of models: a computational-independent model (CIM), a platform-independent model (PIM), and a platform-specific model (PSM). This distinction allows separation of concerns by specifying models at different abstraction levels. Figure 1-6 depicts the relationships between those types of models.

A CIM defines the goal and requirements of a distributed application. It abstracts from the structure and functionality of the application. A CIM accommodates different designs to achieve the goal and to satisfy the requirements.

A PIM specifies the structure and functionality of a distributed application defined in a CIM. It abstracts from the details of the technological platform to which the application is targeted. In this way, a PIM can be implemented with a number of different platforms. This reduces the development cost of the same application functionality on different platforms.

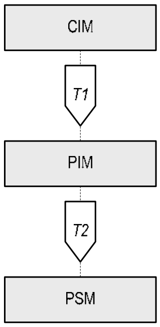


Figure 1-6
Relationships between CIM, PIM, and PSM

In the MDA approach, a PIM and PSM are obtained from the application of model transformations on a CIM and PIM, respectively. A model transformation defines a specification for transforming a model to another model of the same application. In Figure 1-6, a model transformation *T1* transforms a CIM to a PIM. Another model transformation *T2* transforms that PIM to a PSM. A CIM can be transformed to a number of PIMs. A PIM can be transformed to a number of PSMs. Each transformation requires a different transformation specification. A model transformation can also be defined to transform a PSM to an executable implementation. A model transformation can be done manually or (semi-)automatically.

Interaction designs at successive abstraction levels can be aligned with a CIM, PIMs, and PSMs, as illustrated in Figure 1-7. An interaction design *D0* that consists of an abstract interaction specifying the goal and requirements of a service composition is a CIM. This interaction design is recursively refined into interaction designs *D1* and *D2* that abstract from the details of the technological platform to which the service composition is targeted. These interaction designs are PIMs. Further refinement into an interaction design *D3* is done to facilitate an implementation with a specific target platform. This interaction design is a PSM. Refinement is a model transformation in the MDA approach.

The separation of concerns between a PIM and PSM proposed by the MDA approach can reduce the development cost of the same distributed application on different platforms. When a tool support for model transformation is available, the MDA approach can improve implementation quality, speeds up the development process, and further reduces the development cost.

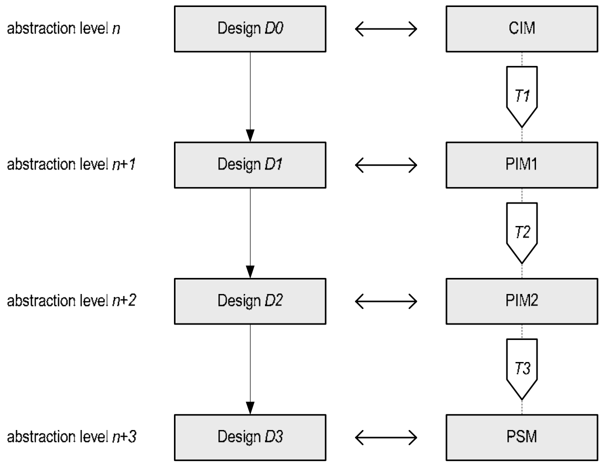


Figure 1-7
Interaction designs at successive abstraction levels aligns with a CIM, PIMs, and PSM

1.2.3 Research questions

To obtain benefits of the use of related abstraction levels and the MDA approach in the design process of service compositions, we identify the following research questions.

- *RQ1*: What interaction design concept is suitable for modelling interactions at related abstraction levels? Are available interaction design concepts suitable for this purpose?
- *RQ2*: How to transform interaction designs between related abstraction levels? How to assess the conformance between interaction designs at different abstraction levels?
- *RQ3*: How to facilitate the development process of a service composition? How can the MDA approach contribute to that process?

1.3 Objectives

The objectives of our research directly correspond to the research questions identified in Section 1.2.3. The objectives are as follows.

The first objective is *to propose an interaction design concept that is suitable for modelling interactions at related abstraction levels*. The interaction design concept should be independent from any interaction mechanism, in order to prevent the interaction design concept from forcing a designer to design service compositions at an implementation level. The interaction design concept should be generic with regard to abstraction levels and application domains. This is to allow a designer to model interactions at any abstraction level in any application domain.

The second objective is *to provide interaction design transformations between related abstraction levels*. Interaction design transformations between related abstraction levels should preserve the conformance between them. Thus, the design transformation should allow a designer to assess the conformance between interaction designs at different abstraction levels. This includes the definition of conformance requirements and a conformance assessment method.

The third object is *to facilitate the design and implementation process of a service composition*. We aim to provide

- guidelines on the possible refinements of an interaction design;
- abstract representations of interaction mechanisms, which allow interaction mechanisms to be included in an interaction design at a higher abstraction level; and
- a transformation tool to transform an interaction design at an implementation level to an executable implementation on a Web Services platform.

1.4 Scope

We define an interaction design concept and transformations for ISDL (Interaction System Design Language) reported in [31, 44, 107, 110, 111, 130, 131]. Reasons for choosing ISDL are as follows.

- The ISDL design concepts are basic design concepts that can be used at any abstraction level in a design process. These design concepts are not specific to some application domain. ISDL, hence, allows us to develop an interaction design concept that is generic with regard to abstraction levels and application domains.
- The ISDL design concepts have clear semantics that are necessary to assess whether a concrete design conforms to an abstract design. For assessment, ISDL is supported with general behaviour transformations. We can reuse and extend those transformations in order to develop interaction design transformations.
- The ISDL interaction concept satisfies some of the requirements that we define for an interaction design concept that is suitable for modelling abstract interactions (presented later in Chapter 2). We can enhance the interaction concept such that it satisfies all the requirements.
- ISDL is supported with a modelling and simulation tool [57]. Availability of such tools encourages designers to use ISDL.

To facilitate the implementation process of a service composition, this thesis provides abstract representations of interaction mechanisms provided by communication middleware. We focus on interaction mechanisms that are provided in CORBA [89] and Web Services [133] platforms because CORBA and Web Services specifications are available in the public domain and, therefore, allow us to study the behaviour of their interaction mechanisms. Web Services, in particular, is a popular platform to implement services and service compositions.

To facilitate the implementation process of a service composition, this thesis provides a transformation tool to transform an interaction design at an implementation level to an executable implementation in a Web Service platform. We focus on a transformation tool that transforms an interaction design that describes an orchestration to an executable implementation in BPEL (Business Process Execution Language) 1.1 [20].

1.5 Approach

In order to achieve the objectives of our research, we use the following approach.

1. We analyse interaction design concepts and methods for designing service compositions. In the analysis, we focus on the suitability of the design concept to model interactions at higher abstraction levels; and the way the interaction design methods manage the complexity of interaction designs. Our first observation is that the interaction design concepts are less suitable for that purpose and, consequently, the interaction design methods that are based on those concepts cannot manage the complexity of interaction designs adequately.
2. We define an interaction design concept that is suitable for modelling interactions at related abstraction levels. Since we use behaviour concepts of ISDL, we first analyse the suitability of the current ISDL interaction concept to model interactions at higher abstraction levels. We then propose the necessary enhancement for that interaction concept.
3. We define interaction design transformations between successive abstraction levels. Since we use ISDL, we first analyse design transformations that are available in ISDL for designing interactions at successive abstraction levels. We then propose extensions of the existing design transformations. We also provide guidelines on interaction refinements.
4. Using our interaction design concept and transformations, we provide abstract representations of common interaction mechanisms provided by CORBA and Web Services platforms. We first represent interaction mechanisms as interaction patterns that are independent of the details of specific middleware. We then abstract each interaction pattern into a

single interaction. These abstract representations could serve as target refinements at an implementation level.

5. We develop an automatic model transformation tool to transform an interaction design at an implementation level to an executable implementation in a Web Services platform.
6. We apply our interaction design concept and transformations in the development processes of service compositions. We carry out two case studies: travel reservation application [134] and enterprise application integration [123]. In the case studies, we evaluate our interaction design concept, transformations, abstract representations of interaction mechanisms, and transformation tool to assess whether they serve their purposes well and can be used in practice.

1.6 Outline of the thesis

The remainder of this thesis is structured as follows.

Chapter 2 (*Analysis of interaction design concepts and methods*) analyses available interaction design concepts and methods for designing service compositions. This chapter concludes that available interaction design concepts are not suitable for modelling interactions at higher abstraction levels. The design methods are forced to produce interaction designs at an implementation level.

Chapter 3 (*Design concept for interaction modelling*) defines an interaction design concept that is suitable for modelling interactions at related abstraction levels. This chapter first introduces ISDL design concepts, including its current interaction concept. It then discusses the limitation of the interaction concept for modelling interactions at higher abstraction levels and proposes the necessary enhancement.

Chapter 4 (*Interaction design transformations*) defines interaction design transformations between successive abstraction levels. This chapter first introduces the design transformations provided by ISDL. It then discusses their limitations for transforming interaction designs between successive abstraction levels and proposes the necessary extensions. This chapter includes guidelines on possible refinements of an interaction design.

Chapter 5 (*Abstract representations of interaction mechanisms*) presents abstract representations of common interaction mechanisms supported by communication middleware, i.e., CORBA and Web Services. When an abstract representation of an interaction mechanism cannot be obtained, a shorthand notation is introduced.

Chapter 6 (*Transformation to executable implementations*) presents an automatic transformation tool to transform an interaction design at an implementation level to an executable implementation in BPEL.

Chapter 7 (*Case study: travel reservation application*) and Chapter 8 (*Case study: enterprise application integration*) present applications and evaluations of our interaction design concept and transformations in the development of service compositions.

Chapter 9 (*Conclusions*) concludes this thesis by outlining our main contributions and some directions for further research.

Chapter 2

Analysis of interaction design concepts and methods

This chapter analyses interaction design concepts and methods for service compositions. Specifically, we analyse the suitability of the interaction design concepts to model interactions at higher abstraction levels. We analyse the way the design methods use the interaction design concepts to bridge the conceptual gap between the business and software domains and to manage the complexity of interaction designs.

This chapter is organised as follows. Section 2.1 describes the relation between design concepts and a design language. Section 2.2 describes the relation between design concepts and design methods. Section 2.3 defines a framework for suitability analysis. Section 2.4 presents our analysis. Section 2.5 shows an example of a top-down design process of a service composition using an unsuitable interaction design concept. Section 2.6, finally, presents some concluding remarks.

2.1 Design concepts and design language

The purpose of a design process is to produce a *design* prescribing a system that should be built. A design addresses a system's characteristics or properties that are relevant to a certain purpose while ignoring properties that are considered irrelevant to that purpose.

A design is created as a composition of *design concepts*. A design concept models aspects of objects or phenomena in a given domain. Design concepts exist only in the mind of a designer. Since a design is a composition of design concepts, a design also exists only in the mind of a designer [44, 131]. When a design process starts, the system that should be built does not exist yet. A designer creates a design as a mental image that represents the system.

For the purposes of documentation, communication, and analysis, a design, and thus the design concepts used, must be represented in some tangible form. A *design notation* is therefore necessary to represent a design concept in a concise, complete, and unambiguous way. Such notations can be graphical or textual. A *design language* is a collection of design notations and rules to compose them. In a design language, the design concepts define the semantics, the design notations define the syntax, and the composition rules define the grammar.

Using a design language, a designer can express a design as a *specification*. A specification is created as a composition of design notations that specifies a system. By interpreting a specification, a designer can create a mental image of the corresponding design and refer to it.

Figure 2-1 depicts the relations between design concepts, design, design notation, and specification [44]. Design concepts and design exist in the conceptual world in the designer's mind. Design notations and specification exist in the symbolic world.

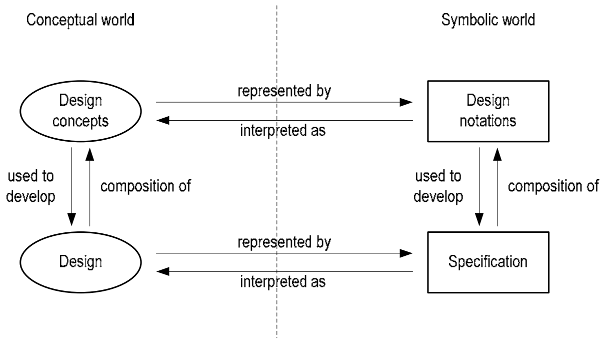


Figure 2-1
Relations between the conceptual world and the symbolic world

A specification represents a design of a system. In this thesis, the term *design* is used to denote *design* and *specification*.

A design language may have multiple different notations for the same design concept. For example, UML (Unified Modeling Language [96]) provides different interaction notations that represent the same interaction design concept in different types of diagrams. Our analysis focuses on interaction design concepts, not on interaction design notations. However, examples of interactions in a design notation remain necessary to communicate the interaction design concept to the reader.

Some design languages may have no interaction design concept. In such a design language, an interaction is typically represented by a composition of other design concepts. In this case, we analyse what kind of interaction is represented by that composition.

2.2 Design concepts and design methods

A *design method* provides guidelines to perform design steps in a design process. A design method can distinguish a number of abstraction levels. An abstraction level marks an intermediate design as the result of a step in a design process. Refinement, the transformation of an abstract design into a more concrete design, is a creative process of composing design concepts [44]. A set of design patterns [11, 16, 46, 54, 70, 129] can be provided to give a designer hints in composing design concepts in order to satisfy certain generic requirements.

A design method can refer to the design concepts of a design language. For examples, the design methods in [32, 41, 143] are specific to BPMN (Business Process Modeling Notation [87]). Those design methods provide guidelines for the development of a design by referring to the BPMN design concepts. Such a design method is language dependent, i.e., depends on the referenced design language. It, therefore, cannot be used with other design concepts. Figure 2-2 depicts the relations between a design method, design concepts, and design steps in the development of designs.

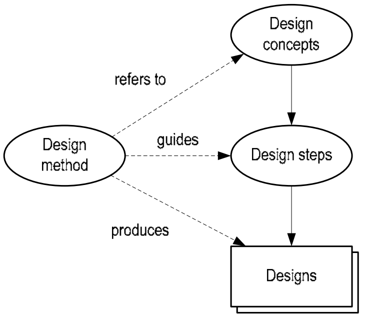


Figure 2-2
A design method refers to design concepts to guide design steps to produce designs

A design method can restrict the use of design concepts, for example, by using specific annotations or stereotypes. The semantics of a design concept, however, cannot be violated by a design method. A design method can also define a subset of design concepts in a design language, which are allowed to be used in a design. In this way, a design method creates a *profile* of that design language [95]. A profile is targeted to a specific use, e.g., a specific application domain or implementation platform [9, 71, 92, 93].

We analyse interaction design concepts and methods for service compositions. Since a number of design methods may use the same design language, we first analyse the interaction design concepts of the design languages that are used by the design methods and then analyse the design methods.

2.3 Framework for suitability analysis

In this section, we define a framework for analysing the suitability of interaction design concepts to model abstract interactions.

2.3.1 Abstract interactions

In general, an abstract design reflects only design properties that are essential at the considered abstraction level, while ignoring properties that are irrelevant at the considered abstraction level. The ignored properties may be essential at a lower abstraction levels. At any abstraction level, one can choose which properties are considered essential and thus which properties one abstracts from. In interaction design, we want to abstract a structure of interactions for achieving a specific goal into a single interaction that only specifies that goal. This allows us to separate concerns of "*what is the desired goal*", the higher abstraction level, and "*how to achieve that goal*", the lower abstraction level [60]. In this thesis, a goal is represented by a desired result. A goal is achieved when this result is established.

This separation of concerns leads to the definition of an interaction at two related abstraction levels as follows.

- At a higher abstraction level, an abstract interaction specifies a desired result.
- At a lower abstraction level, a structure of more concrete interactions specifies how to establish that result.

These related abstraction levels can be considered as a relative notion. An abstraction level n is higher than an abstraction level $n+1$, but is lower than an abstraction level $n-1$.

An entity that is involved in an interaction has its responsibility in the establishment of the interaction result. This responsibility can be modelled as requirements or constraints that have to be satisfied by the result. In a design process at related abstraction levels, an abstract interaction specifies the requirements that the involved entities have for the result; and a structure of more concrete interactions specifies how the involved entities satisfy those requirements.

We argue that a designer should be able to represent a structure of interactions that establish a certain result by an abstract interaction that yields the same result. This allows better understanding of the involved entities, the responsibilities of those entities, and the desired result, while abstracting from detailed interactions between those entities.

Of course, a designer could represent a structure of interactions by a generic activity, i.e. an action [131], that abstracts also from the participation of individual entities. With this representation, the designer only knows the desired result. This, however, is not sufficient in case of a service composition where a designer wants to distinguish the different entities. In the beginning of a design process of a service composition, most likely, the designer has already some knowledge of existing or future services, and the participating entities, to be composed and of the distribution of responsibilities between the entities in the establishment of a desired result. This knowledge would be best expressed as an interaction, not as an action.

Abstract interactions allow a business analyst to participate in the design of a service composition. A business analyst understands very well the business domain, but they are typically not knowledgeable or interested in system or implementation details. The participation of a business analyst is important to increase the possibility that a service composition indeed meets the business needs.

2.3.2 Motivating example

We use a service composition in Figure 2-3 to motivate the definition of our requirements for an interaction design concept that is suitable for modelling abstract interactions. This figure uses intuitive graphical notation. A rounded rectangle represents an entity. A bidirectional arrow represents an interaction. Interactions are numbered to indicate the order in which they should be performed. Entities that are involved in an interaction are called *participants* of that interaction.

Figure 2-3 illustrates the following interactions between a buyer, seller, bank, and courier for purchasing an article.

1. The buyer browses a product catalogue of the seller.
2. The buyer orders an article in that product catalogue.
3. The seller sends the invoice of the ordered article to the buyer.
4. The buyer orders the bank to transfer some amount of money as indicated in the invoice, from the buyer's bank account to the sellers' bank account.
5. The buyer notifies the seller that the requested amount of money has been transferred to the seller's bank account as the payment of the invoice.
6. The seller checks with the bank whether the money has been received.
7. The seller confirms the payment to the buyer.
8. The seller orders the courier to deliver the purchased article.
9. The courier delivers the article to the buyer.
10. The courier confirms the delivery of the article to the seller.

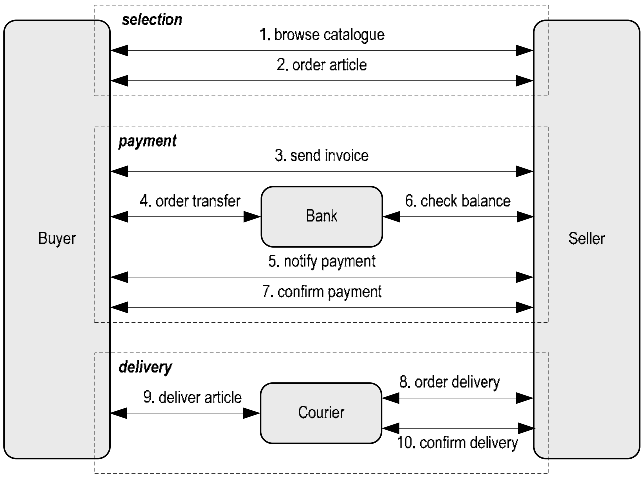


Figure 2-3
A service composition for purchasing an article

The designer may want to represent this example by a single abstract interaction for purchasing an article. However, the complexity of this example hinders the designer to derive an abstraction in a single step. To overcome the complexity, the designer can group those interactions into three smaller compositions of interactions: *selection*, *payment*, and *delivery* (as indicated with dashed rectangles in the figure); each of which is for achieving a sub-goal. Interaction *selection* is for selecting an article from the seller's catalog. Interaction *payment* is for paying a selected article. Interaction *delivery* is for delivering a purchased article from the seller to the buyer. Their abstractions can be derived and then further abstracted into a single interaction.

2.3.3 Abstraction patterns

The motivating example consists of four generic abstraction patterns. A pattern is characterised by a generic structure of interactions and its desired abstraction.

Pattern 1: multiple interactions to a single interaction

A structure of interactions may consist of multiple interactions between participants, in which all participants are engaged in all interactions. This pattern abstracts such a structure of interactions into an interaction between those participants.

Figure 2-4 illustrates multiple interactions between a buyer and seller for selecting an article (i.e., interactions 1 and 2 of the motivating example in Figure 2-3). We want to be able to abstract those interactions into an interaction between the buyer and seller.

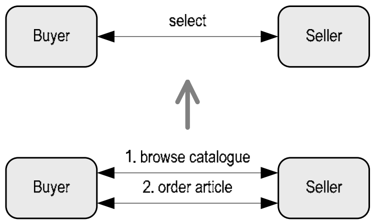


Figure 2-4
Pattern 1: multiple interactions to a single interaction

Pattern 2: intermediary elimination

A structure of interactions may consist of indirect interactions between participants through an intermediary. This pattern abstracts such a structure of interactions into a direct interaction between those participants.

Figure 2-5 illustrates indirect interactions between a buyer and seller through a courier for delivering an article (i.e., interactions 8, 9, and 10 in Figure 2-3). We want to be able to abstract those interactions into an interaction between the buyer and seller.

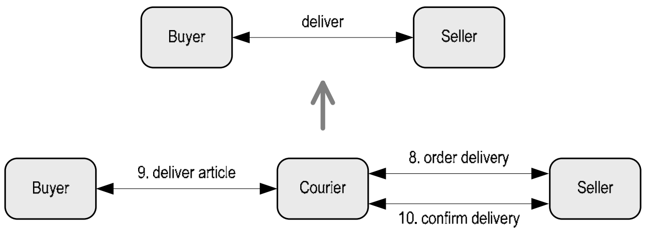


Figure 2-5
Pattern 2: intermediary elimination

Pattern 3: bilateral interactions to a multilateral interaction

A structure of interactions may involve three or more participants that interact with each other, in which every interaction is a bilateral interaction, i.e., performed by two participants only. A participant does not have to interact with all other participants. This pattern abstracts such a structure of interactions into a multilateral interaction between the participants.

Figure 2-6 illustrates interactions between a buyer, seller, and bank for paying an invoice (i.e., interactions 3, 4, 5, 6, and 7 in Figure 2-3). We want to be able to abstract those interactions into a multilateral interaction between the buyer, seller, and courier.

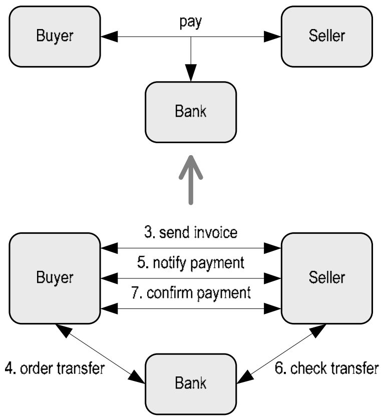


Figure 2-6
Pattern 3: bilateral interaction to a multilateral interaction

Pattern 4: participant elimination

A structure of interaction may consist of an interaction between three or more participants, in which some of the participants facilitate the implementation of that interaction. This pattern abstracts such an interaction into an interaction that abstracts from the facilitating participant.

Figure 2-7 illustrates an interaction between a buyer, seller, and bank for paying an invoice (i.e., the abstraction of interactions 3, 4, 5, 6, and 7 in pattern 3 in Figure 2-6). The bank acts as a facilitating participant in this interaction. We want to be able to abstract this interaction into an interaction between the buyer and seller only.

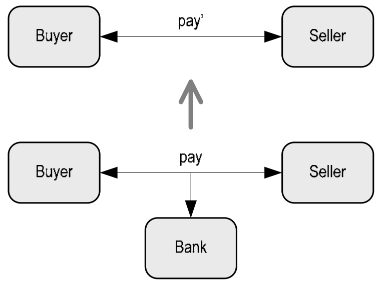


Figure 2-7
Pattern 4: participant elimination

2.3.4 Suitability requirements

To assess whether an interaction design concept is suitable for modelling abstract interactions, we define the following suitability requirements. An interaction design concept should allow a designer

– *SR1: to model an interaction between two or more participants.*

None of the abstraction patterns limits the maximum number of participants of an interaction. The abstract interaction in pattern 3 and the structure of interaction in pattern 4 require an interaction design concept that can model an interaction between three or more participants.

- *SR2: to define different views of different participants on the established result.*
  Different participants may have different views on the result that is established by an abstract interaction. A view is modelled by a set of values that represents a (partial) result. In pattern 2, the different interactions between an intermediary and different participants may establish different views on a desired result. In patten 4, a facilitating participant may have a partial interest in and, hence, a different view on the established result.
- *SR3: to specify the relation between different views of different participants.*
  Since different views represent the same established result, they must be related to each other.
- *SR4: to specify participants' requirements.*
  Participants are interested in the interaction result and use it for their own activities. They need to be able to impose their own requirements on the result.

2.4 Suitability analysis

In this section, we analyse the interaction design concepts and methods for service compositions in [15, 32, 33, 41, 51, 65, 76, 81, 108, 113, 117, 125, 143, 145]. These methods are selected based on the following criteria.

- *Supported by graphical notations*. A business analyst prefers to use graphical notations to model (interacting) business processes because this way of modelling is more intuitive and comfortable for them [87]. This criterion excludes design methods that use only textual or mathematical specifications for modelling service compositions.
- *No technical details of implementation platforms*. Typically, a business analyst has no knowledge of, or interest in, the technical details of implementation platforms. This criterion excludes design methods that specific for implementation platforms.

The design languages used by those design methods are UML [96], BPMN [87, 88], Petri Net [104], Let's Dance [146], and ISDL [44, 107, 110]. We focus on the behaviour modelling of service compositions, not on the structural modelling.

2.4.1 UML

UML provides different types of diagrams to serve the modelling of different aspects of a system. UML offers a large number of packages. A package consists of a set of design concepts.

The *CommonBehaviors* package provides a communication infrastructure for interactions between objects, regardless of the diagram that is used to model the interactions, i.e., activity, sequence, communication, or interaction overview diagram. This package defines two kinds of communication: *signal passing* and *operation call*. A signal passing is an asynchronous communication. An operation call can be either an asynchronous or synchronous communication. An asynchronous communication between a sender and receiver allows the sender to continue its execution without having to wait any reply from the receiver. A synchronous communication makes the sender wait for a reply from the receiver before it can continue its execution.

The *Actions* package provides action types for behaviour modelling. It includes actions for communication: *SendSignalAction*, *AcceptEventAction*, *CallOperationAction*, *AcceptCallAction*, and *ReplyAction*.

In a signal passing, a sender sends a send request (called a *signal*) to a receiver by executing a SendSignalAction. After sending the signal, this action completes immediately. To receive a signal, a receiver executes an AcceptEventAction. A signal triggers a reaction in the receiver and is without a reply. Figure 2-8 depicts signal-passing communication in a sequence diagram.

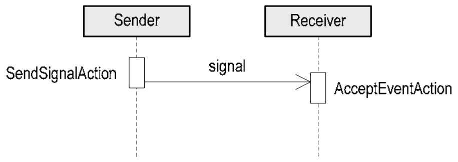


Figure 2-8
Signal passing in a sequence diagram

To make an asynchronous operation call, a sender sends a call request to a receiver by executing a CallOperationAction with attribute 'isSynchronous' set to 'false'. After sending the call request, this action completes immediately. To receive a call request, a receiver executes an AcceptEventAction. This request invokes an operation in the receiver. Figure 2-9 depicts an asynchronous operation call in a sequence diagram.

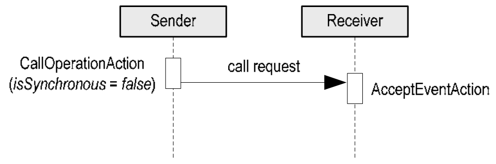


Figure 2-9
Asynchronous operation call in a sequence diagram

To make a synchronous operation call, a sender sends a call request to a receiver by executing a CallOperationAction with attribute 'isSynchronous' set to 'true'. This attribute setting makes the action wait for a reply. To receive a call request, a receiver executes an AcceptCallAction. This request invokes an operation in the receiver. To send a reply, the receiver executes a ReplyAction. When the sender receives the reply, its CallOperationAction completes and produces outputs describing the reply. Figure 2-10 depicts a synchronous operation call in a sequence diagram.

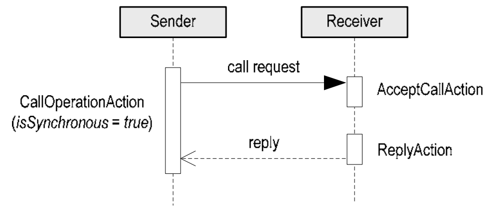


Figure 2-10
Synchronous operation call in a sequence diagrams

A request (i.e., a send request or a call request) is sent by exactly one sender and is received by exactly one receiver. A sender, however, may generate a number of requests; each of which is sent to a different receiver.

Signal passing and operation call represent message-passing and request-response interaction mechanisms, respectively, that are commonly provided by communication middleware.

The *UseCases* package includes the concept of *UseCase*. A use-case defines a behaviour that a systems offers to its users, abstracting from the internal structure or functions of the behaviour. Hence, a use-case can be considered as an abstract interaction between a system and its users. The behaviour of a use case can be described using interactions, activities, or state machines. A system may offer a set of use-cases, but their execution order cannot be specified.

The *CompositeStructure* package includes the concept of *Collaboration*. A collaboration defines an abstraction of a structure of participants to accomplish some functionality. A collaboration models the structure, not the behaviour, of a distributed application.

Suitability

The suitability analysis of the UML interaction design concepts to model abstract interactions is as follows.

- *SR1:* A signal passing and operation call is performed by two participants only, i.e., a sender and receiver. A designer cannot model an interaction between three or more participants.

  With a use-case, a designer can model an interaction between two or more participants, i.e., a system and one or more users.
- *SR2:* In a signal passing, the participants see the same signal between them. In an operation call, the participants see the same call request and the same reply, if any. The participants have the same view on the established result. A designer cannot define different views for different participants.

  A designer cannot define the result established in a use-case and thus the views on the result.
- *SR3:* In a signal passing and operation call, the relation between the participants' views is pre-defined, i.e., all participants have the same view. A designer cannot specify the relation between the participant's views.

  Since the result and the views on the result cannot be defined in a use-case, a designer cannot specify the relation between the views.
- *SR4:* A signal passing, operation call, and use-case have no property that allows a designer to specify the participants' requirements.

The UML interaction design concepts do not satisfy all the suitability requirements.

Design methods

UML is used in the design methods in [15, 65, 81, 117, 125].

[15] distinguishes between a static model and a dynamic model. The static model specifies the structure of a service composition. The dynamic model specifies the behaviour of the service composition. In the static model, participants are represented by components that have *uses* relationships with each other. In the dynamic model, a sequence diagram is used to model interactions between those participants.

This design method does not distinguish any abstraction level. It cannot bridge the conceptual gap between the business and software domains. It cannot help a designer to manage the complexity of an interaction design.

[65] distinguishes three abstraction levels: *collaboration level*, *transaction level*, and *interaction level*. At the collaboration level, a service composition is modelled as a collaboration between objects that represent participants. A collaboration may consist of sub-collaborations. At the transaction level, a collaboration is refined into an activity diagram that specifies the behaviour of the collaboration. An action in an activity diagram represents a transaction between two or more participants. A transaction can be refined into sub-transactions in another activity diagram. At the interaction level, an

activity diagram is refined into a number of sequence diagrams; each of which refines an action of that activity diagram.

The abstraction levels can bridge the conceptual gap between the business and software domains. This design method can help a designer to manage the complexity of an interaction design. However, the use of different concepts to represent interactions at different abstraction levels can create conceptual gaps between the abstraction levels. Furthermore, the use of collaborations and activity diagrams at higher abstraction levels shows that the UML interaction design concepts cannot model abstract interactions.

[81] distinguishes two abstraction levels. At the higher abstraction level, a collaboration between participants is represented by a number of use-cases in a use-case diagram. A participant is represented as an actor. At the lower abstraction level, the behaviour of each use-case is specified in a sequence diagram.

The higher and lower abstraction levels represent the business and software domains, respectively, but do not bridge them. The use of use-cases at the higher abstraction level allows the complexity of an interaction design to be managed at the lower abstraction level. A use-case represents a sub-goal of a service composition. A corresponding sequence diagram specifies how to achieve a sub-goal. Since a use-case diagram does not specify the execution order of use-cases, such an ordering has to be specified in a sequence diagram. This makes the use-cases less helpful in managing the complexity of an interaction design.

In [117], interactions between participants are modelled in an activity diagram. An activity is annotated with a stereotype that indicates the activity in an implementation. Stereotypes «*WebServiceCall*» and «*ImmediateStep*» indicate a service call and internal activity, respectively. A participant is modelled as a class that is stereotyped with «*BusinessService*» in a class diagram. This class lists operations provided by that participant.

This design method does not distinguish any abstraction level. It cannot bridge the conceptual gap between the business and software domains. It cannot help a designer to manage the complexity of an interaction design.

[125] distinguishes between a static model and a dynamic model, but does not distinguish any abstraction level (similar to [15]). In the static model, a participant is modelled as a class that is stereotyped with «*serviceComponent*». This class lists the operations that are provided by that participant. In the dynamic model, the behaviour of a service composition is modelled as an activity diagram. In this diagram, an activity represents a service invocation.

This design method cannot bridge the conceptual gap between the business and software domains. It cannot help a designer to manage the complexity of an interaction design.

2.4.2 BPMN

BPMN is a design language for business process modelling. Interactions between business processes can be defined as a collaboration, choreography, or conversation.

A collaboration defines interactions between business processes in terms of *message flows*. A message flow defines the flow of a message between two participants, in which one participant sends the message and another participant receives the message. A message flow can only be specified across business processes, i.e., a message flow cannot be specified between tasks, activities, or sub-processes of the same business process.

A message flow represents a message-passing interaction mechanism. Figure 2-11 depicts an example of an interaction between a sender and receiver for sending a message.

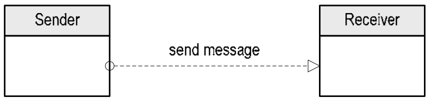


Figure 2-11
Sending a message from a sender to a receiver

A choreography defines the coordination of interactions between participants in terms of *choreography activities* and their ordering relations. A choreography activity represents an interaction or message exchanges between two or more participants. A choreography activity can be decomposed into sub-activities.

Figure 2-12 depicts the interactions between a distributor, retailer, and shipper as a choreography that consists of two choreography activities *stock order* and *plan shipment*. Activity *stock order* involves two participants, i.e., the distributor and retailer. The retailer initiates this activity. The initating participant is indicated by the white band on which the participant name is specified. Activity *plan shipment* involves three participants, i.e., the distributor, retailer, and shipper.

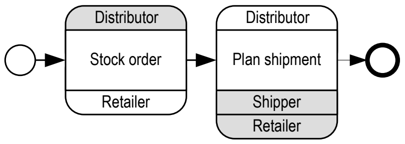
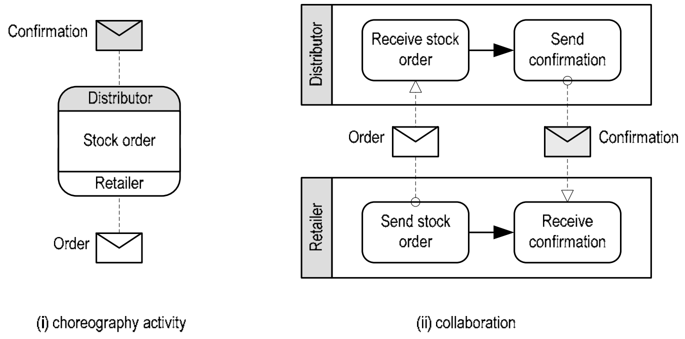


Figure 2-12
A choreography between a distributor, retailer, and shipper

The messages that are exchanged in a choreography activity can be specified, as depicted in Figure 2-13(i). This choreography activity represents the message flows in the collaboration in Figure 2-13(ii).

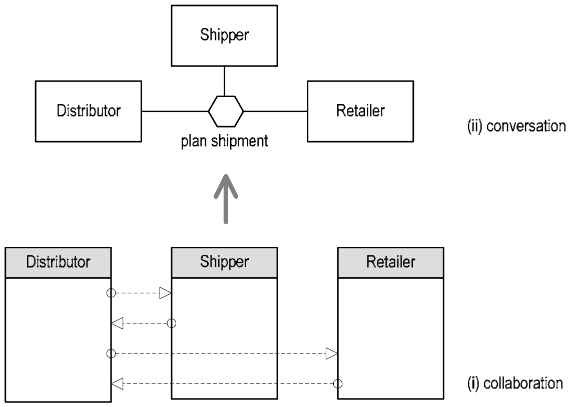
Figure 2-13
A choreography activity and a collaboration that is represented by that choreography activity



A conversation represents a group of related message exchanges between two or more participants. A conversation can be decomposed into sub-conversations.

Figure 2-14(i) depicts a conversation between a distributor, retailer, and shipper to plan the shipment of ordered products. This conversation represents a set of message flows in the collaboration in Figure 2-14(ii).

Figure 2-14
A collaboration is represented by a conversation



Suitability

The suitability analysis of the BPMN interaction design concepts to model abstract interactions is as follows.

- *SR1:* A message flow is performed by two participants only, i.e., a sender and receiver. A designer cannot model an interaction between three or more participants.

  With a choreography activity or conversation, a designer can model an interaction between two or more participants.
- *SR2:* In a message flow, the participants see the same message between them. They have the same view on the established result. A designer cannot define different views for different participants.

In a choreography activity or conversation between two participants, the participants see the same messages between them. The participants have the same view on the established result. A designer cannot define different views for different participants.

In a choreography activity or conversation between three or more participants, different participants see different messages. However, a choreography activity or conversation has no property that allows a designer to define the different messages for different participants.

- *SR3:* In a message flow, the relation between the participants' views is pre-defined. In a choreography activity or conversation between two participants, the relation between the participants' views is also pre-defined. A designer cannot specify the relation between the participants' views.

  In a choreography activity or conversation between three or more participants, there is no interaction property that allows a designer to specify the relation between the different messages for different participants.
- *SR4*: A message flow, choreography activity, and conversation have no property that allows a designer to specify the participants' requirements.

The BPMN interaction design concepts does not satisfy all the suitability requirements.

Design methods

BPMN is used in the design methods in [32, 41, 143]. These design methods use message flows only. To our knowledge, no design method for service compositions uses the BPMN choreography activity and conversation yet.

[32] defines four deliverables in different types of models: *choreography milestone*, *choreography scenario*, *choreography*, and *provider behaviour*. A choreography milestone model specifies the milestones in a collaboration. No interaction is defined in this model. A choreography scenario model specifies a possible conversation scenario between participants from one milestone to another milestone. A choreography model represents all interactions between participants. A provider behaviour model of a participant specifies all interactions in which that participant is involved. This model can also show internal activities of that participant.

Different deliverables represent different concerns on a service composition. They do not represent abstraction levels, but we can derive the abstraction-refinement relationships between the deliverables. A choreography scenario model is a refinement of a choreography milestone model. A provider behaviour model is a refinement of the internal behaviour of a participant in a choreography scenario model. A choreography model is an abstraction of a choreography scenario model.

However, since all interactions are specified as message flows, those abstraction levels cannot bridge the conceptual gap between the business and software domains. This design method cannot help a designer to manage the complexity of an interaction design.

[41] uses the BPMN process types to represent abstraction levels: *collaboration (global) process*, *abstract (public) process*, and *private (internal) process*. A collaboration process specifies interactions between participants, abstracting from the internal activities of those participants. An abstract process specifies the participants and their interaction activities. A private process specifies internal behaviour of a participant. It can contain sub-processes; each of which is to be refined into activities or tasks.

All interactions are specified as message flows. Similar to [32], the abstraction levels cannot bridge the conceptual gap between the business and software domains. This design method cannot help a designer to manage the complexity of an interaction design.

[143] models a service composition as the internal behaviour of the coordinator of an orchestration. This design method does not distinguish any abstraction level. Hence, it cannot bridge the conceptual gap between the business and software domains. It cannot help a designer to manage the complexity of an interaction design.

2.4.3 Petri Nets

Petri Nets is a formal/mathematical modelling language for analysing distributed systems. It consists of two basic concepts: *places* and *transitions*, which represent a *state* and an *activity* of a system, respectively. Control flows between activities can be specified by directional relations between transitions and places.

Petri Nets do not have an interaction design concept. To model an interaction, a design method typically uses a pair of transitions connected via a place as depicted in Figure 2-15. One transition represents an activity for sending a message and another transition represents an activity for receiving the message. An interaction that is modelled in this way represents a message-passing interaction mechanism.

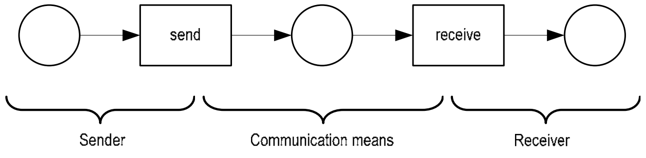


Figure 2-15 Modelling message passing in Petri Net

Suitability

A transition represents an activity in general, not an interaction. Petri Nets do not have an interaction design concept, thus we cannot analyse their suitability.

A designer may use hierarchical Petri Nets [45] for modelling abstract interactions. Figure 2-16 depicts the abstraction of the interaction in Figure 2-15. In this example, a transition is used to represent an interaction. This representation allows a designer to model an interaction between two or more participants and, therefore, satisifies requirement *SR1*. However, it does not satisfy requirements *SR2*, *SR3*, and *SR4* because Petri Nets cannot specify the result that should be established in an interaction.

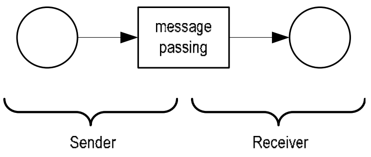


Figure 2-16
Message passing as a transition

Design methods

Petri Nets are used in the design methods in [33, 51, 76].

[33] defines four viewpoints from which a service composition can be described: *choreography viewpoint*, *interface behaviour viewpoint*, *provider behaviour viewpoint*, and *orchestration viewpoint*. A choreography viewpoint describes a collaboration between participants. It shows only the activities that are used in the collaboration, i.e., the send and receive activities as depicted in Figure 2-15. An interface behaviour viewpoint describes the observable behaviour of a role played by a participant to interact with another participant. A provider behaviour viewpoint describes the observable behaviour of a participant in a collaboration. If a participant plays multiple roles, its provider behaviour is a composition of the interface behaviour viewpoints that describe those roles. An orchestration viewpoint describes the internal behaviour of the coordinator of an orchestration. The relations between viewpoints are defined as follows. A participant in a choreography can be refined into a provider behaviour. A provider behaviour of a participant can be refined into an orchestration.

The relations between viewpoints lead us to associate the viewpoints with the following successive abstraction levels: choreography, provider behaviour, and orchestration. However, since all interactions are specified as message passings, those abstraction levels cannot bridge the conceptual gap between the business and software domains. This design method cannot help a designer to manage the complexity of an interaction design.

[51, 76] do not distinguish any abstraction level. A service composition is represented in terms of message exchanges between participants. The internal activities of the participants are specified in the same model. These design methods cannot bridge the conceptual gap between the business and software domains. They cannot help a designer to manage the complexity of an interaction design.

2.4.4 Let's Dance

Let's Dance is a design language for modelling service behaviours. An interaction between participants or actors is described in terms of a message exchange. One actor performs a communication action called a *message sending action*; another actor performs a communication action called a *message receipt action*. Two types of message exchanges are distinguished: *send without acknowledgement* and *send with acknowledgement*, as depicted in Figure 2-17 and Figure 2-18, respectively. In these figures, actor *Sender* performs a message sending action and actor *Receiver* performs a message receipt action. An interaction is modelled by two complementary communication actions that are connected to each other.

Message passing without acknowledgement represents an unconfirmed message-passing interaction mechanism. Message passing with acknowledgement represents a provider-confirmed message-passing interaction mechanism.

Figure 2-17
Send without acknowledgement

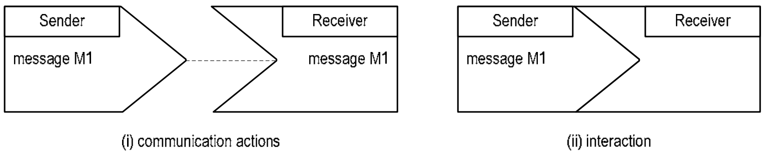
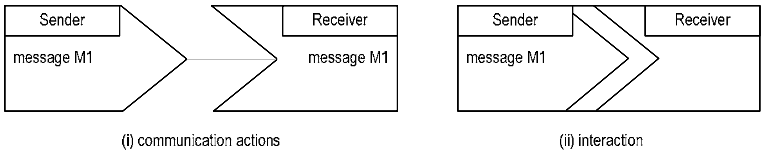


Figure 2-18
Send with acknowledgement



Suitability analysis

The suitability analysis of the Let's Dance interaction design concepts to model abstract interactions is as follows.

– *SR1:* A message passing is performed by two participants only, i.e., a sender and a receiver. A designer cannot model an interaction between three or more participants.

- *SR2:* The participants see the same message between them. They have the same view on the established result. A designer cannot define different views for different participants.
- *SR3:* The relation between the participants' views is pre-defined, i.e., all participants have the same view. A designer cannot specify the relation between the participants' views.
- *SR4*: A communication action has no property that allows a designer to specify the participants' requirements.

The Let's Dance interaction design concept does not satisfy all the suitability requirements.

Design method

Let's Dance is used in the design method in [145]. The design method defines two views: *global view* and *local view*. The global view of a service composition shows a choreography model between participants. The local view of a participant is obtained by taking the communication actions of that participant from the choreography model in the global view. This method does not distinguish any abstraction level. It cannot bridge the conceptual gap between the business and software domains. It cannot help a designer to manage the complexity of an interaction design.

2.4.5 ISDL

ISDL (Interaction Systems Design Language) is a design language for modelling distributed systems. It was developed based on the experiences with the use of LOTOS (Language Of Temporal Ordering Specification [24, 59]) [44], while the development of LOTOS itself was based on earlier research on the interaction concept. The dynamic part of the behavioural model in LOTOS is derived from the process algebras of CCS (Calculus of Communicating Systems [82]) and CSP (Communicating Sequential Processes [53]). Since then, several ideas for improving ISDL have been suggested in, and beyond, our research group, including some ideas elaborated in this thesis. Most of these ideas, though, have not been formalised yet. For these reasons, our analysis in this section is based on the formulation of the ISDL interaction concept as available in the literature.

In ISDL, an *interaction* is defined as a unit of activity that is performed by two or more participants in cooperation to establish a common result. A result is represented by a set of values. Participants' requirements are specified as constraints that are imposed on that result.

An interaction is considered atomic in the sense that it either occurs or does not occur at all, when used in a specification at a certain abstraction level. If an interaction occurs, it establishes the same set of values representing the result. These values are available from the same time moment and at the same location for all participants. The ISDL interaction design concept adopts a synchronous interaction model, requiring participants to be involved in an interaction simultaneously.

Figure 2-19 depicts the purchase interaction of a car between two participants: a buyer and seller. When this interaction occurs, it establishes the same value for the car and the same value for the price in the buyer and seller that are available from the same time moment and at the same location.

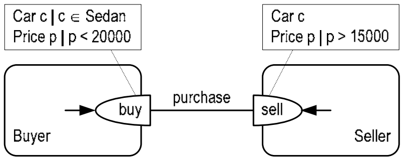


Figure 2-19
A purchase interaction between a buyer and seller

Suitability

The suitability analysis of the ISDL interaction design concept to model abstract interactions is as follows.

– *SR1:* A designer can model an interaction between two, three, or more participants.
– *SR2:* The participants have the same view on the established result. A designer cannot define different views for different participants.
– *SR3:* The relation between the participants' views is pre-defined, i.e., all participants have the same view. A designer cannot specify the relation between the participants' views.
– *SR4:* A designer can specify the participant's requirements as constraints that are imposed on the interaction result.

The ISDL interaction design concept does not satisfy all the suitability requirements.

Design methods

ISDL is used in the design methods in [108, 113].

[108] defines four abstraction levels: *business process specification*, *application service specification*, *application service design*, and *application service implementation*. A business process specification defines activities in a service composition, abstracting from possible assignments of the activities to individual entities. An application service specification defines entities that are involved in a service composition. It identifies which activities should be performed by which individual entities and which activities should be performed by the entities in cooperation. The business process specification is decomposed accordingly. An activity that is to be performed in cooperation becomes an interaction. Figure 2-20 illustrates the

decomposition of a business process into application services. An application service design specifies a participant in terms of a composition of sub-entities. An application service implementation specifies the implementation of an application service design with a specific service technology or platform.

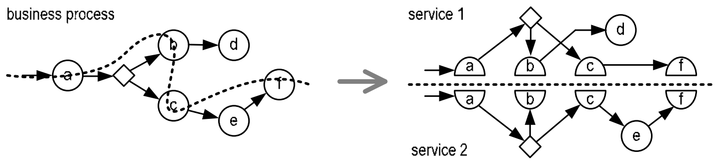


Figure 2-20 Decomposition of a business process into application service designs

An interaction is defined as a refinement of an activity in a business process specification. Once an interaction between participants is defined (at the application service specification level), the interaction is not further refined. The interaction design between participants is defined only at that abstraction level. The abstraction levels cannot bridge the conceptual gap between the business and software domains. This design method cannot help a designer to manage the complexity of an interaction design.

[113] defines three generic abstraction levels: *single interaction*, *choreography*, and *orchestration*. At the abstraction level of a single interaction, a service composition is modelled as a single interaction between participants. This interaction specifies the goal of the service composition. It is refined into a structure of interactions between the participants at the choreography level. An interaction at the choreography level can be further refined into a structure of interactions. A participant is refined into a composition of sub-entities at the orchestration level.

Interactions are designed at related abstraction levels. The design method can bridge the conceptual gap between the business and software domains. It can help a designer to manage the complexity of an interaction design. However, the synchronous interaction model that is adopted by the ISDL interaction concept limits possible interaction refinements. A designer can only define direct interactions between participants because participants have to be involved in an interaction simultaneously. Since all participants have the same view on the established result, a designer cannot include a participant that has partial interest on the result, e.g., a facilitating participant.

2.4.6 Summary

We summarise our analysis as follows.

Interaction design concepts

Table 2-1 summarises the results of our suitability analysis. We consider the suitablity of the interaction design concept only based on our specific requirements and not based on suitability for other purposes.

Table 2-1
Results of our suitability analysis

Language and interaction design concept	Suitability requirements			
	SR1	SR2	SR3	SR4
UML: signal passing	–	–	–	–
UML: operation call	–	–	–	–
UML: use-case	+	–	–	–
BPMN: message flow	–	–	–	–
BPMN: choreography activity	+	–	–	–
BPMN: conversation	+	–	–	–
Petri Nets	N/A	N/A	N/A	N/A
Let's Dance	–	–	–	–
ISDL	+	–	–	+

Legend:

+	: satisfied
–	: not satisfied
N/A	: not applicable (Petri Nets does not have any interaction design concept)

A designer can use comments or other textual notation to add design properties that are not provided by an interaction design concept. Comments or other textual notation is an informal way to specify a design. We do not include them in the analysis.

An interaction design concept that is suitable for modelling abstract interactions would satisfy all the suitability requirements. From Table 2-1, we observe that the ISDL interaction concept is the closest one to a suitable interaction design concept. Therefore, we take the ISDL interaction concept as a basis for our interaction design concept and enhance it in order to make it satisfy all the suitability requirements.

Design methods

The design methods in [32, 33, 41, 65, 81, 108] distinguish related abstraction levels. However, the interaction design concepts that they use force a designer to define interactions at an implementation level only.

Except for [108, 113], the design methods that distinguish related abstraction levels do not provide a mechanism to assess the conformance between designs at different abstraction levels. Without such conformance assessment, the correctness relation between interaction designs cannot be established.

2.5 Examples of interaction designs

In this section, we design the service composition in the motivating example in Section 2.3.2 in two interaction designs. The first interaction design is developed by using an interaction design concept that can only model concrete interactions. It aims at showing that such an interaction design concept cannot help a designer to manage the complexity of the interaction design at related abstraction levels. The second interaction design is developed by using an interaction design concept that can model abstract interactions. It aims at giving an outlook for a design method that allows a designer to model interactions at related abstraction levels.

In the service composition, we identify the buyer and seller as the essential participants; and the bank and courier as supporting participants. An essential participant is a participant without which a service composition cannot occur. A supporting participant is a participant that facilitates interactions between the esential participants. It can be an intermediary or facilitating participant. A supporting participant may be removed if it is not used; or substituted with other supporting participant(s).

2.5.1 Interaction design using concrete interactions

We use the BPMN message flow to design a service composition for the following reasons.

- The BPMN message flow represents a message-passing mechanism.
- BPMN supports abstraction levels by providing the concepts of abstract processes and sub-processes. An abstract process represents a business process abstracting from its internal behaviour. A sub-process represents a composition of sub-actitivies of a business process as a single activity. Hence, we can show the use of the BPMN message flow at related abstraction levels.

We follow the design method in [41] for designing our motivating example in Section 2.3.2. Firstly, we design the service composition as a collaboration process, as depicted in Figure 2-21. The design shows the message flows between the participants. All participants, i.e., the essential and the supporting participants, and all message flows have to be specified in the design.

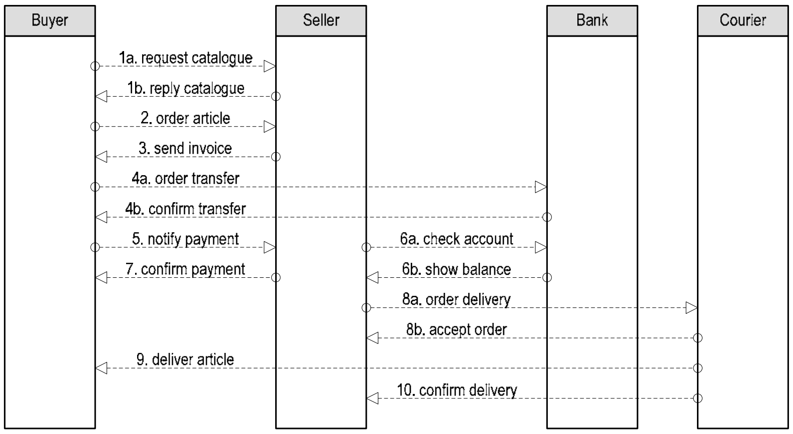


Figure 2-21
The purchase scenario as interacting abstract processs

A message flow cannot represent multiple related message flows in different directions. A request-response interaction mechanism must be modelled as two message flows. One message flow is for sending the request and another message flow in the opposite direction is for sending the response. Interactions 1, 4, 6, and 8 in Figure 2-3 are to be implemented using a request-response interaction mechanism. Each of those interactions is therefore modelled as two message flows in opposite directions, e.g., message flows 1a and 1b.

Abstracting from the bank as a facilitating participant and the courier as an intemediary, will remove message flows numbered with 4, 5, 6, 7, 8, 9, and 10 as depicted in Figure 2-22. This would leave the design incomplete and unclear. Questions may arise. For example, does the buyer have to pay the invoice before sending a payment notification to the seller (message flow no. 5)? Does the buyer get the purchased article?

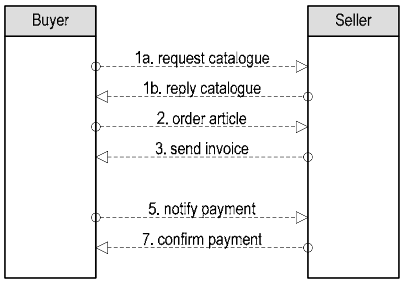


Figure 2-22
Abstracting from the bank and courier

Secondly, we refine the collaboration process by modelling the phases of the purchasing. We model those phases as collapsed sub-processes within the participants' processes. The collapsed sub-processes are *selection*, *payment* and *delivery*, as depicted in Figure 2-23. The numbers of message flows correspond to the numbers of message flows in Figure 2-21.

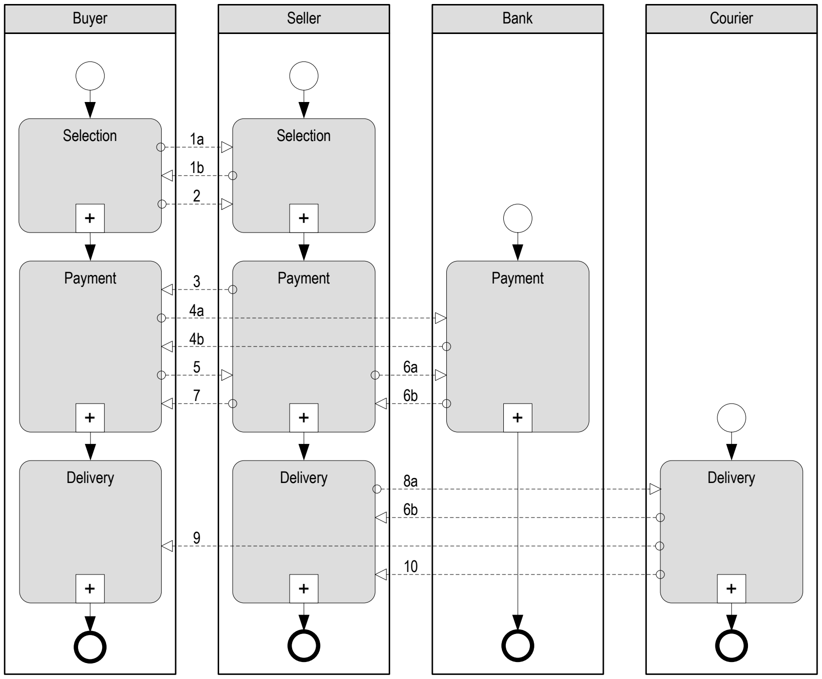


Figure 2-23
Phases of the purchasing are represented as collapsed sub-processes

Finally, we refine the design to model the internal behaviour of the phases in the participants. We expand the sub-processes with activities as depicted in Figure 2-24. We consider this design as a design at an implementation level.

This example shows that, although we can model the participants' behaviours at a higher abstraction level (Figure 2-21 and Figure 2-23), the BPMN message flow forces us to specify the interactions between the participants at an implementation level. Abstractions are applicable only for modelling the internal behaviour of the participants. The BPMN message flow does not allow us to abstract a structure of related interactions into a single abstract interaction.

The BPMN message flow does not allow us to abstract from supporting participants. The introduction of the behaviour of supporting participants increases the complexity of an interaction design at the early phases of a design process.

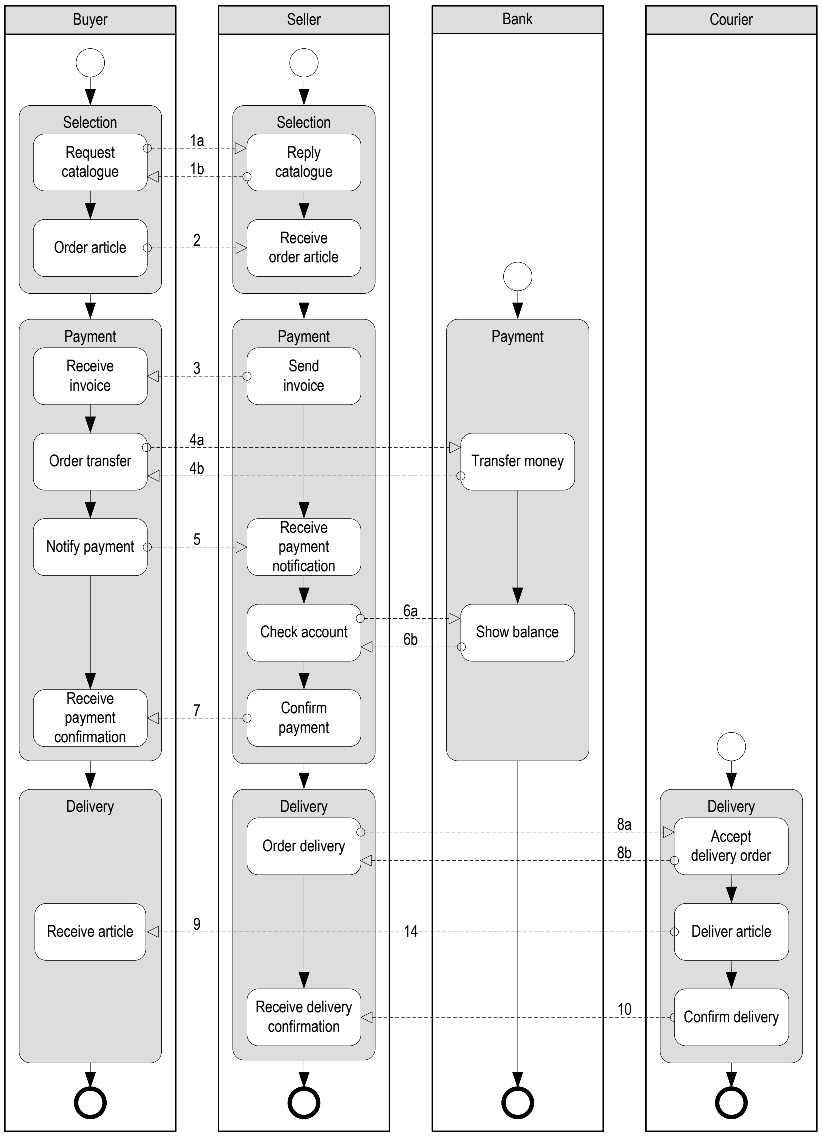


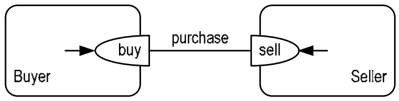
Figure 2-24
The service composition for purchasing at an implementation level

2.5.2 Interaction design using abstract interactions

We use ISDL to design a service composition because we will use the ISDL interaction concept as a basis for our interaction design concept (see Section 2.4.6).

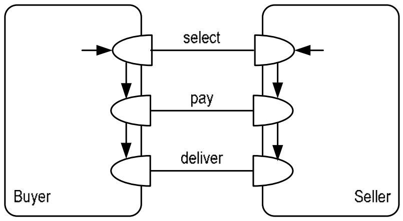
At a higher abstraction level, the service composition is represented as an abstract interaction between the essential participants, i.e., the buyer and seller, as depicted in Figure 2-25. The design should specify the result to be established, the participants' views on the result, the relations between the participants' views, and the participants' requirements on the establishment of the result.

Figure 2-25
A service composition as an abstract interaction



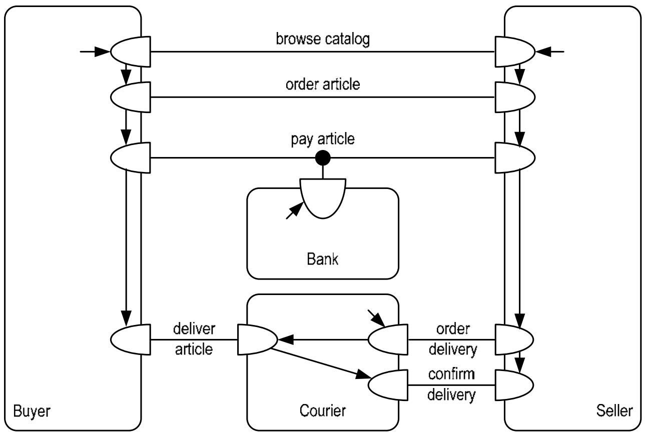
We then refine this abstract interaction into a structure of interactions that specifies how the service composition establishes the result. The structure of interactions consists of interactions *select*, *pay*, and *deliver* that are performed consecutively, as depicted in Figure 2-26.

Figure 2-26
A structure of interactions (1)



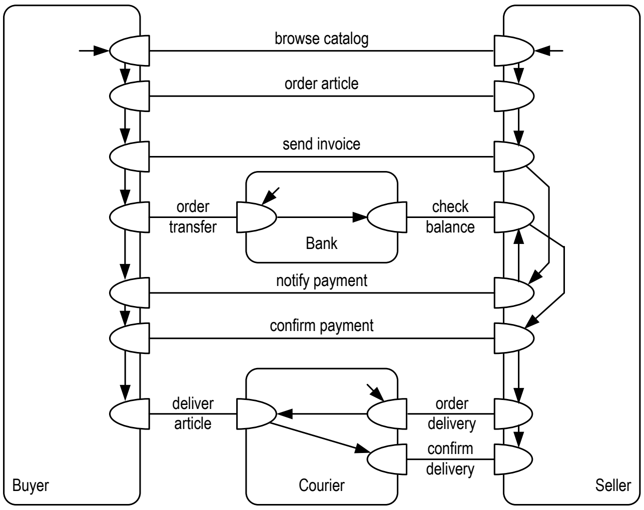
We recursively refine each interaction in Figure 2-26 into a structure of interactions, as depicted in Figure 2-27. Abstract interaction *select* is refined into two interactions: *browse catalog* and *order article*. Abstract interaction *pay* is refined into an interaction *pay article* that introduces a bank as a facilitating participant. Abstract interaction *deliver* is refined into a structure of interactions that introduces a courier as an intermediary.

Figure 2-27
A structure of interactions (2)



This recursive refinement can be done until it results in interactions that can be mapped onto available interaction mechanisms. Figure 2-28 is obtained from refining abstract multilateral interaction *pay article* in Figure 2-27 into a structure of bilateral interactions.

Figure 2-28
A structure of interactions (3)



In order to obtain a concrete interaction design that conforms to an abstract interaction design, refinement should be followed by conformance assessment.

2.6 Concluding remarks

In this chapter, we have analysed interaction design concepts and methods for service compositions. The interaction design concepts are from UML, BPMN, Let's Dance, and ISDL. Most of the interaction design concepts represent interaction mechanisms that are provided by communication middleware. Such an interaction design concept forces a designer to develop interaction designs at an implementation level.

We have also analysed how Petri Nets can be used to represent an interaction, i.e., a message-passing mechanism. Such an interaction represents a concrete interaction that cannot model abstract interactions.

Some of the analysed design methods define related abstraction levels. However, they fail to manage the complexity of an interaction design, because they are not supported by interaction design concepts that are suitable for modelling abstract interactions.

Requirements of an interaction design concept

We aim to define an interaction design concept that is suitable for modelling interactions at related abstraction levels. We define the following requirements for such an interaction design concept.

- *An interaction design concept should be suitable for modelling abstract interactions.* The suitability requirements in Section 2.3.4 should be satisified. The interaction design concept should allow a designer
  - to model an interaction between two or more participants,
  - to define different views of different participants on the established result,
  - to specify the relation between different views of different participants, and
  - to specify participants' requirements directly.
- *An interaction design concept should be suitable for modelling concrete interactions.* Such an interaction design concept should allow a designer to model interaction mechanisms precisely, because an interaction design is eventually realised by an application developer. A precise interaction model avoids misinterpretation of an interaction design by the application developers.

In Chapter 3, we discuss further the limitations of the ISDL interaction design concept and propose the necessary enchancements. In Chapter 4, we define transformations for designing interactions at related abstraction levels. In Chapter 5, we show that the enhanced interaction concept is suitable for modelling concrete interactions.

Chapter 3

Design concepts for interaction modelling

Any artificial system, including a distributed application, is developed to deliver a specific functionality. In order to deliver that functionality, a system performs a certain behaviour. An external entity that wants to access that functionality must interact with the system while itself exhibiting a certain behaviour. Behaviour modelling, therefore, is necessary in system design. As a distributed application is characterised by interactions between application components, interaction modelling is essential in the behaviour modelling of a distributed application.

This chapter presents design concepts in ISDL (Interaction System Design Language [31, 44, 107, 110, 111, 130, 131]), especially for behaviour modelling, and enhances the ISDL interaction concept. This chapter is organised as follows: Section 3.1 gives an overview of basic design concepts for modelling distributed systems. Section 3.2 presents perspectives on distributed systems. Section 3.3 presents in more detail design concepts for behaviour modelling. These three sections summarise the current state-of-the-art of ISDL. The following sections present new contributions. Section 3.4 shows the limitations of the ISDL interaction concept for modelling abstract interactions. Section 3.5 enhances the ISDL interaction concept to make it satisfy the suitability requirements defined in Chapter 2. Section 3.6 describes the relationship between the action concept and the enhanced interaction concept. Section 3.7 presents shorthand notations for interaction specifications that appear frequently in designs. Finally, Section 3.8 presents some concluding remarks.

3.1 Basic design concepts

A distributed system can be represented in two conceptual domains: *the entity domain* and *the behaviour domain*.

3.1.1 Entity domain

In *the entity domain*, a distributed system is represented as a structure of interconnected entities. Three basic design concepts are identified: *entity*, *interaction point*, and *action point*.

Entity

An *entity* represents a logical or physical mechanism as a carrier of the behaviour of a part of a distributed system or of the system as a whole. It does not represent the behaviour itself. This behaviour has to be defined by a separate behaviour specification. For example, a travel reservation application can be represented by an entity whose behaviour specifies how to help a customer in making a flight or hotel reservation. Alternative terms to denote an entity are 'component', 'object', 'module', or 'resource'. An entity is uniquely identified by an *entity identifier* (entity name).

Interaction point

An *interaction point* represents a logical or physical mechanism through which entities can interact. It does not represent the interactions themselves. These interactions are defined by the behaviour specification of the entity. An interaction point is shared between entities. This implies that interacting entities can establish results to which these entities can refer. An entity can access the functionality of another entity only through interactions at one or more interaction points. For example, a series of web pages displaying the steps to make a flight reservation is an interaction point through which a customer can interact with a travel reservation application. An entity can access the functionality of another entity only through their interaction points. An alternative term to denote an interaction point is 'connector'. An interaction point is uniquely identified by an *interaction point identifier* (interaction point name).

Figure 3-1 depicts a model of a distributed system consisting of two entities: a *customer* and *travel reservation application*, that interact with each other through an interaction point *ip*. An entity is graphically expressed as a box with cut-off corners. Its identifier is placed inside that box. An interaction point is graphically expressed as an ellipse connecting entities that share this interaction point. Its identifier is placed inside that ellipse.

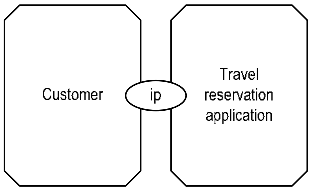


Figure 3-1
Entities and interaction points in a distributed system

Action point

During refinement, an entity can be decomposed into multiple internal entities that are interconnected via internal interaction points. When considering that entity as a whole, one may abstract from the internal entities but keep the internal interaction points. In this case, internal interaction points are called *action points*. An *action point* represents an internal mechanism of an entity at which a result is established.

In Figure 3-2, a travel reservation application is refined by decomposing it into four internal entities: a *presentation component*, *coordinator*, *flight reservation system*, and *hotel reservation system*. These entities are interconnected via internal interaction points p_1, p_2, and p_3. Interaction point *ip* of the original entity is maintained. One may consider this decomposed application by abstracting from its internal entities. This leaves the application with action points p_1 and p_2.

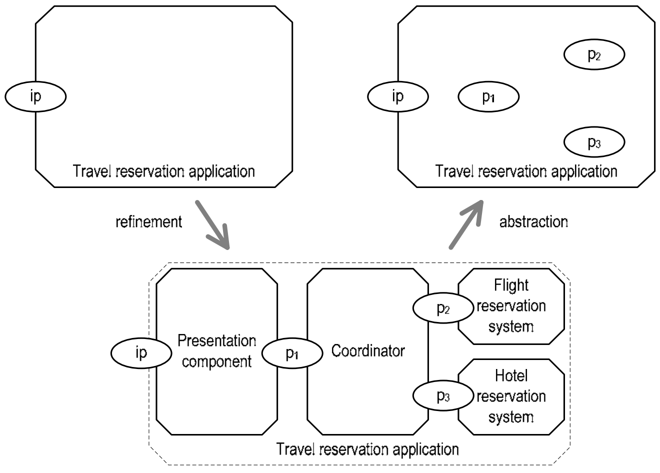

ip
Travel reservation application
ip
p1
p2
p3
Travel reservation application
refinement
abstraction
ip
Presentation component
p1
Coordinator
p2
Flight reservation system
p3
Hotel reservation system
Travel reservation application

Figure 3-2
Refinement and abstraction of a travel reservation application

Of course, one can also define these internal action points directly, avoiding the detour of defining the internal entities. However, a design process is generally intended to define the internal entities.

An action point is uniquely identified by an *action point identifier* (action point name). An action point is graphically expressed as an ellipse. Its identifier is placed inside that ellipse.

A model that represents a distributed system in the entity domain is called an *entity model*.

3.1.2 Behaviour domain

In *the behaviour domain*, a distributed system is represented as a composition of interacting behaviours. The system as a whole is represented as a single behaviour that delivers the system's functionality.

To define and specify a behaviour, three basic design concepts are defined: *action*, *interaction*, and *causality relation*. These design concepts are briefly introduced here and further elaborated in Section 3.3.

Action

A distributed system delivers its functionality by performing one or more activities. An *action* represents a unit of activity that is performed by a single entity to establish a result. A *result* is represented by the availability of a set of information values at a certain moment in time and at a certain location. In the example of a travel reservation application above, parsing a user request into an operation request that is understandable to the coordinator is an action of the presentation component. Formatting an operation result into a response that is understandable to a user is another action. Alternative terms to denote actions are 'tasks' or 'internal activities'. An action is uniquely identified by an *action identifier* (or action name).

Interaction

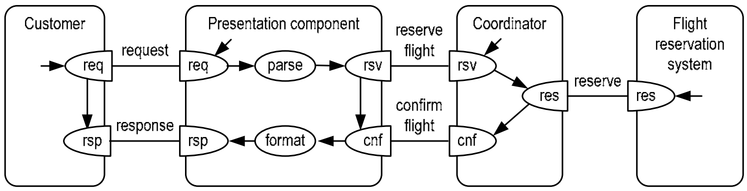
An *interaction* represents an action that is performed by multiple entities in cooperation to establish a common result. For example, a customer interacts with a travel reservation application to make a flight reservation. The contribution of an entity in an interaction is called an *interaction contribution*. For example, in an interaction for making a flight reservation, a customer contributes by providing the departure place, destination place, and preferred travel date; while the travel reservation application contributes by providing a reserved flight to the customer. Alternative terms to denote an interaction are 'joint task', 'shared activity', or 'collaboration'. An interaction is uniquely identified by an *interaction identifier* (interaction name).

An action can also be seen as an integrated interaction where the individual contributions of the entities are abstracted away.

Related actions and interaction contributions of an entity can be grouped within a *behaviour*. A behaviour is uniquely identified by a *behaviour identifier* (behaviour name).

Figure 3-3 depicts a model of a distributed application that consists of activities for making a flight reservation. The activities are two actions, i.e., *parse* and *format*, and five interactions, i.e., *request*, *response*, *reserve flight*, *confirm flight*, and *reserve*. An action is graphically expressed as an ellipse. Its identifier is placed inside that ellipse. An interaction is graphically expressed as segmented ellipses linked with a line. Its identifier is placed near that line. A segmented ellipse represents an interaction contribution of a participating entity. It has an interaction contribution identifier that is unique within a behaviour.

Figure 3-3
Actions and interactions for making a flight reservation



Activities in Figure 3-3 are grouped in four sub-behaviours: *customer*, *presentation component*, *coordinator*, and *flight reservation system*. A behaviour is graphically expressed as a rounded rectangle. Its identifier is placed inside that rounded rectangle. Actions are placed inside a behaviour in which they are grouped. An interaction is shared between behaviours, such that one interaction contribution is in one behaviour and other interaction contribution(s) is in other behaviour(s).

Causality relation

A *causality relation* defines the condition for the occurrence of an individual action and for the value of its result. For example, in Figure 3-3, a causality relation between interaction *request* and action *parse* (which is graphically expressed as an arrow) defines that action *parse* may occur only after interaction *request* has occurred.

A model that represents a distributed system in the behaviour domain is called a *behaviour model*. A design process of developing behaviour models is called *behaviour modelling*.

3.1.3 Assignment relation

An *assignment relation* relates an entity and behaviour model of the same system. This relation assigns a behaviour to an entity in order to define the behaviour of that entity. This relation therefore indicates which entity executes which behaviour.

In an assignment relation, the following consistency rules must be complied with:

- An action of a behaviour happens at an action point of an entity to which that behaviour is assigned.

– An interaction between behaviours happens at an interaction point that is shared by entities to which those behaviours are assigned.
– Related actions and interaction contributions in a behaviour are assigned to the same entity.

Figure 3-4 depicts an assignment relation between a behaviour and entity model. Behaviours *customer*, *presentation component*, *coordinator*, and *flight reservation system* are assigned to entities with the same names. Actions *parse* and *format* are assigned to action points ap_1 and ap_2, respectively. Interactions *request* and *response* are assigned to interaction point ip_1. Interaction *reserve flight* and *confirm flight* are assigned to interaction point p_1. Interaction *reserve* is assigned to interaction point p_2.

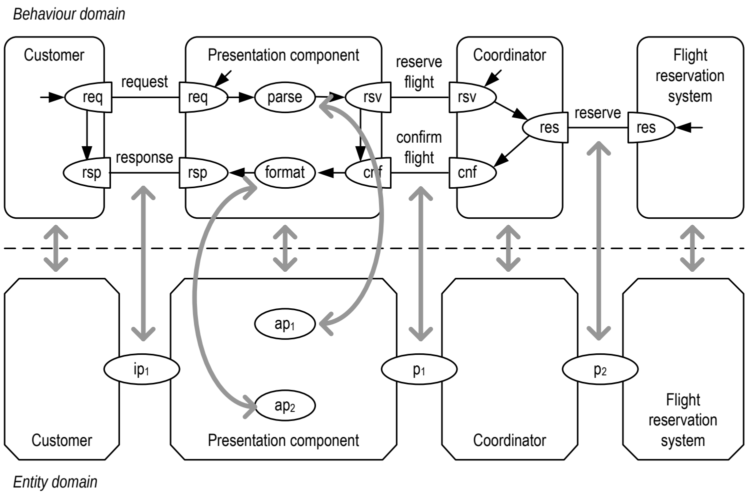


Figure 3-4
Assignment relation

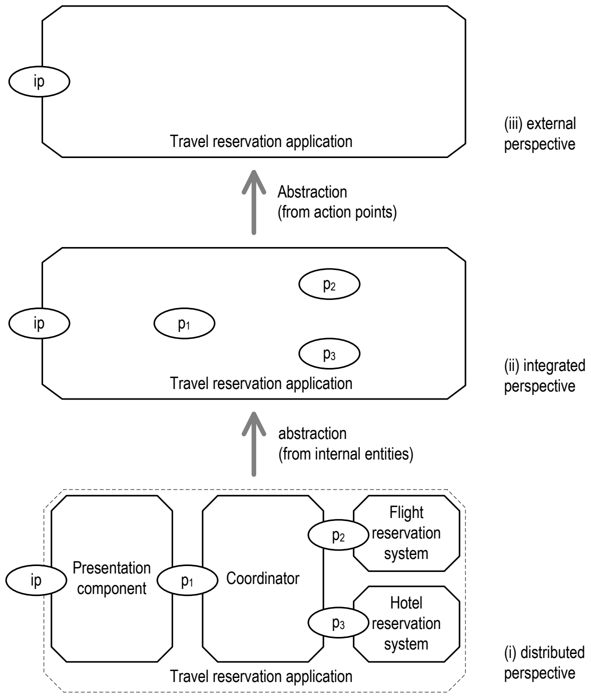
3.2 System perspectives

A distributed system can be viewed from different perspectives. Three perspectives are distinguished: *distributed perspective*, *integrated perspective*, and *external perspective*.

Distributed perspective

A distributed system can be represented as a structure of interconnected entities, and correspondingly, as a composition of interacting behaviours. This representation is called *the distributed perspective*. Figure 3-5(i) depicts the distributed perspective of a travel reservation application in the entity domain. It shows the internal entities of the application and how the entities are structured.

Figure 3-5
System perspectives


ip
Travel reservation application
(iii) external perspective
Abstraction (from action points)
ip
p1
p2
p3
Travel reservation application
(ii) integrated perspective
abstraction (from internal entities)
ip
Presentation component
p1
Coordinator
p2
Flight reservation system
p3
Hotel reservation system
Travel reservation application
(i) distributed perspective

Integrated perspective

A distributed system can also be represented as a single entity that performs a specific behaviour, abstracting from its internal structure of entities or its composition of behaviours. This representation is called *the integrated perspective*. Figure 3-5(ii) depicts the integrated perspective of a travel reservation application in the entity domain. It shows only the action points in which activities of the application are performed.

The integrated perspective of a distributed system is at a higher abstraction level than the distributed perspective since it abstracts from internal entities and the distribution of system functionality into the behaviours of internal entities.

External perspective

A distributed system can also be represented as an entity providing specific functionality to its users, considering only the possible interactions between the system and its users. This representation is called *the external perspective*. Figure 3-5(iii) depicts the external perspective of a travel reservation application in the entity domain. It shows only the interaction points of the application. It does not show any internal detail of the application.

The external perspective of a distributed system is at a yet higher abstraction level than the integrated perspective since it abstracts from the any internal detail of the system. The external perspective defines *what* should be provided by the system. The integrated and distributed perspectives define in increasingly more detail *how* the system is constructed.

Figure 3-5 depicts the relations between the distributed, integrated, and external perspectives of a travel reservation application in the entity domain. These relations are also applicable in the behaviour domain.

These perspectives, in a reversed order, are used as a basis for a top-down design approach. Firstly, a distributed system is defined from the external perspective. Secondly, the external perspective is refined into the integrated perspective by making the system internally explicit. Finally, the integrated perspective is refined into the distributed perspective by distributing the system over its internal entities. Action points are then transformed into interaction points.

In the behaviour domain, the term *observable behaviour* refers to the external perspective of a system behaviour, because this perspective shows only the behaviour that can be observed by the users. The term *internal behaviour* refers to the integrated or distributed perspective of a system behaviour, because these perspectives shows the internal actions and/or interactions of that system.

Figure 3-6(i) depicts the observable behaviour of a travel reservation application. A user can interact with this application using interaction contributions *req* and *rsp*. These interaction contributions allow the user to send a request to, and receive a response from, this application, respectively. This behaviour only shows a direct mapping between interactions and does not show the internal actions of the application to produce a response.

Figure 3-6(ii) depicts the internal behaviour of the travel reservation application. This internal behaviour shows that the application is composed of two interacting behaviours *coordinator* and *flight reservation system*. This behaviour shows the internal activities of the application to produce a response.

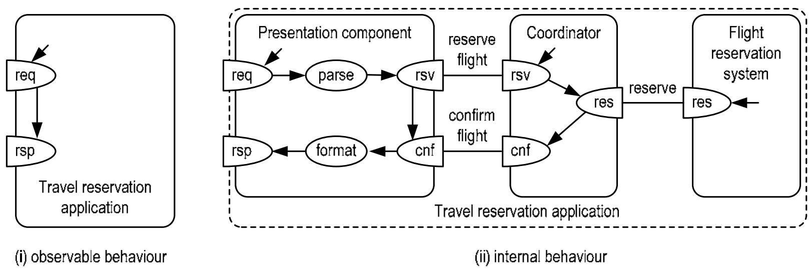


Figure 3-6
Observable and internal behaviours

3.3 Concepts for behaviour modelling

This section elaborates basic design concepts for behaviour modelling.

3.3.1 Action

An *action* represents a unit of activity that is performed by a single entity to establish a result. A result is represented by three attributes:

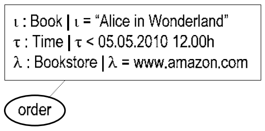
– *information attribute*, which models a set of information values established by the action;
– *time attribute*, which models the time moment from which the information attribute is available;
– *location attribute*, which models the location at which the information attribute is available.

An action is an abstraction of an activity that is considered as a unit of behaviour at a certain abstraction level and cannot be split at this abstraction level. An action, at this level, is hence defined as *atomic*. However, an action can be replaced with a composition of multiple actions at a lower abstraction level.

Since an action, at the considered abstraction level, is atomic, it implies that its result either occurs in full or does not occur at all. The occurrence of the result in full implies that the defined action occurs, while the non-occurrence of the result implies that the defined action does not occur. If an action occurs, other actions can refer to the action's full result. If an action does not occur, other actions cannot refer to any result.

Figure 3-7 depicts an action *order* for ordering a book in an online bookstore. Its attributes are specified in a text box attached to the action. Information, time, and location attribute values are expressed by the symbols ι, τ, and λ, respectively. An attribute has a type (value domain) and may have a constraint specifying the possible values that may be established if the action occurs. An attribute type declaration and its constraint is separated by the symbol '|'.

Figure 3-7
Action *send*


ι : Book | ι = "Alice in Wonderland"
τ : Time | τ < 05.05.2010 12.00h
λ : Bookstore | λ = www.amazon.com
order

Information attribute value ι in Figure 3-7 has an information type *Book* and its constraint allows only one specific value "Alice in Wonderland". Time attribute value τ is of type *Time* and its constraint allows any time moment before 5th May 2010 12.00 hours. Location attribute value λ is of type *Bookstore* and its constraint allows only one specific value 'www.amazon.com'. If this action occurs, it results in the ordering of a book "Alice in Wonderland" at a time moment before 5th May 2010 12.00 hours at an online bookstore 'www.amazon.com'.

Figure 3-8 textually expresses the order action depicted in Figure 3-7. An action is expressed by an action identifier followed by attribute type declarations and constraints. Attribute type declarations are expressed between the symbols '(' and ')'. Constraints are expressed between the symbols '[' and ']'.

Figure 3-8
Textual expression of Figure 3-7

```
order (ι : Book, τ : Time, λ : Bookstore)
    [ι = "Alice in Wonderland",
    τ < 05.05.2010 12.00h,
    λ = www.amazon.com]
```

The information, time, and location attribute values of an action *a* can be expressed as a.ι, a.τ, and a.λ, respectively.

3.3.2 Interaction

An *interaction* represents an action that is performed by multiple entities in cooperation to establish a common result. Entities that are involved in an interaction are called *participants*. Like an action result, an interaction result is represented by information, time, and location attributes.

The contribution of a participant in an interaction is called an *interaction contribution*. An interaction contribution defines constraints that a participant has on the interaction result. An interaction can only occur if the constraints of all participants can be satisfied. The result is established through some form of cooperation or common activity of the participants. Since all constraints have to be satisfied for the interaction to occur, we also call this cooperation a "negotiation" between the participants. The interaction itself abstracts from how the negotiation is performed. When the interaction occurs, a participant can refer to the established result only through its interaction contribution.

An interaction, like an action, is considered and defined as *atomic* at a certain abstraction level. This property imposes that an interaction's result either occurs in full for all participants or does not occur at all. If an interaction occurs, all participants can refer to the interaction result. If an interaction does not occur, none of the participants can refer to any result However, an interaction can be replaced with multiple interactions at a lower abstraction level.

Figure 3-9 depicts an interaction *purchase* between a buyer and seller for purchasing a computer. The interaction contributions of the buyer and seller are 'to buy' and 'to sell', respectively. The buyer has constraints that the price should be between 400 and 700 euro; and the purchase should be done before 22nd January 2010. The buyer has no constraint on the shop at which he should buy the computer. The seller has a constraint that the price should be higher than 500 euro; and that the purchase should occur at his shop 'www.mystore.nl'. The seller has no constraint on the purchase date. If this interaction occurs, it results in the purchase of a computer with a price between 500 and 700 euro before 22nd January 2010 at 'www.mystore.nl'.

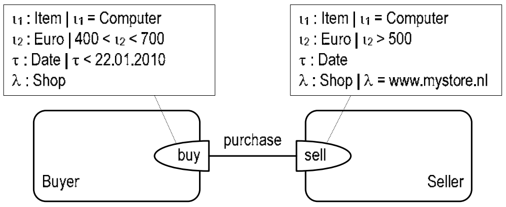

ι_1 : Item | ι_1 = Computer
ι_2 : Euro | 400 < ι_2 < 700
τ : Date | τ < 22.01.2010
λ : Shop
ι_1 : Item | ι_1 = Computer
ι_2 : Euro | ι_2 > 500
τ : Date
λ : Shop | λ = www.mystore.nl
purchase
buy
sell
Buyer
Seller

Figure 3-9
Interaction *purchase* between a buyer and seller

Figure 3-10 textually expresses a purchase interaction depicted in Figure 3-9. The interaction contribution buy is considered as part of the behaviour *Buyer*. The behaviour *Buyer* is expressed by its behaviour identifier *Buyer* followed by the symbol '=' and its behaviour definition. A behaviour definition is delimited by the symbols '{' and '}'. Interaction contributions (contribution *buy* in behaviour *Buyer*) are expressed in the same way as expressing actions, i.e., interaction contribution identifier, attribute type declarations, and constraints. An interaction is expressed by its identifier followed by a list of interaction contributions involved in the interaction, delimited by the symbols '(' and ')'. An interaction definition is only completed when all its interaction contributions are defined. Since an information attribute consists of a set of values, indexes (in subscripts) are used to enumerate the values, e.g., ι_1 and ι_2. This index can be omitted in case an information attribute consists of a single value.

Figure 3-10
Textual expression of Figure 3-9

```
Buyer = {
    buy ($\iota_1$ : Item, $\iota_2$ : Euro, $\tau$ : Date, $\lambda$ : Shop)
        [$\iota_1$ = Computer,
        $400 < \iota_2 < 700$,
        $\tau$ < 22.01.2010]
}

Seller = {
    sell ($\iota_1$ : Item, $\iota_2$ : Euro, $\tau$ : Date, $\lambda$ : Shop)
        [$\iota_1$ = Computer,
        $\iota_2 > 700$,
        $\lambda$ = www.mystore.nl]
}

purchase (buy : Buyer.buy, sell : Seller.sell)
```

Attribute value establishment

Three frequently occurring forms of attribute value establishment in an interaction between two participants are listed in Table 3-1 [128].

Table 3-1
Frequently occurring forms of attribute value establishment

Form	**Description**	**Condition for occurrence**
Value checking	One participant requires a specific value *x* to be established and the other participant requires a specific value *y* to be established.	$x = y$
Value passing	One participant requires a specific value *x* to be established and the other participant allows any value from a set of values *Y* to be established.	$x \in Y$
Value generation	One participant allows any value from a set of values *X* to be established and the other participant allows any value from a set of values *Y* to be established.	$X \cap Y \neq \varnothing$

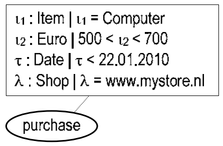
Other forms of attribute value establishment are possible. For example, an auction is an interaction in which all participants bid for an offered item. A bid represents a participant's constraint on the acceptable value of the item. When the auction occurs, it establishes a value that is equal to the highest bid.

Integrated interaction

When moving from the distributed perspective to the integrated perspective, one abstracts from the individual interaction contributions of the participants. This results in an *integrated interaction* that is modelled as an action.

Figure 3-11 depicts an action *purchase* as an abstraction of an interaction *purchase* in Figure 3-9. The attribute constraints of action *purchase* are the intersections of the domains defined by the attribute constraints of the interaction contributions of interaction *purchase*. Interaction contributions *buy* and *sell* in Figure 3-9 define constraints [$400 < \iota_2 < 700$] and [$\iota_2 > 500$], respectively. The intersection of these two constraints is a constraint [$500 < \iota_2 < 700$].

Figure 3-11
Action *purchase* as an integrated interaction of an interaction *purchase*

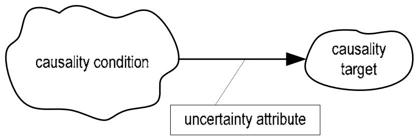

ι1 : Item | ι1 = Computer
ι2 : Euro | 500 < ι2 < 700
τ : Date | τ < 22.01.2010
λ : Shop | λ = www.mystore.nl
purchase

An interaction is considered as a refinement of an action, i.e. it defines an action at a lower abstraction level. An interaction defines the distribution of responsibilities between multiple entities in the establishment of a result. Rules that are applicable to actions, e.g., as presented later in Section 3.3.3, are also applicable to interactions.

3.3.3 Causality relation

A *causality relation* defines the condition for the occurrence of an individual action and the value of its attributes. A causality relation consists of a *causality target*, *causality condition*, and *uncertainty attribute* as depicted in Figure 3-12.

Figure 3-12
Causality relation



Causality target

A *causality target* is an action or interaction contribution for which a causality condition is specified. For brevity, we use in the sequel the term *action* to denote an action as well as an interaction contribution.

Causality condition

A *causality condition* defines the dependency of the occurrence of a causality target on the occurrence or non-occurrence of other actions. Four basic causality conditions are identified (as depicted in Figure 3-13 and textually in Table 3-2):

- *start condition*: action *a* is enabled to occur from the beginning (start). This *start condition* for action *a* is actually no more than a placeholder for another, yet undefined causality condition in another behaviour that enables *a*.
- *enabling condition*: the occurrence of action *b* enables the occurrence of action *a*. When actions *a* and *b* both occur, $a.\tau > b.\tau$. Action *a* can refer to the result of action *b*.
- *disabling condition*: the occurrence of action *b* disables the occurrence of action *a* if action *a* has not yet occurred before action *b*. When actions *a* and *b* both occur, then $a.\tau < b.\tau$. Action *a* can never refer to the result of action *b* because if action a occurs, either action *b* does not occur or action *b* occurs after action *a* occurs.
- *synchronisation condition*: the occurrence of action *b* enables the occurrence of action *a*, such that action *a* occurs at the same time as action *b*. When actions *a* and *b* both occur, $a.\tau = b.\tau$. Action *a* can refer to the result of action *b*.

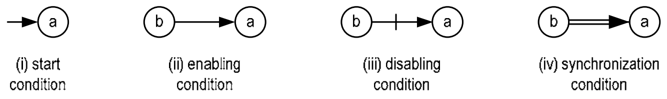


Figure 3-13
Basic causality conditions

Table 3-2
Textual expressions of basic causality conditions

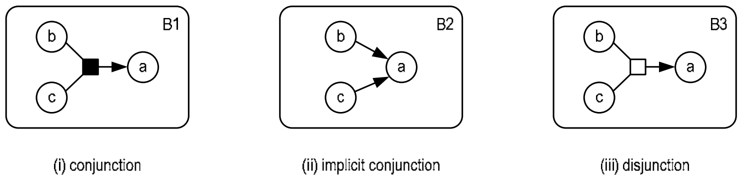
Causality condition	Textual expression
Start condition	$\surd \rightarrow a$
Enabling condition	$b \rightarrow a$
Disabling condition	$\neg b \rightarrow a$
Synchronisation condition	$^{=}b \rightarrow a$

More complex causality conditions can be modelled by combining these basic causality conditions into

- a *conjunction*: all causality conditions must be satisfied to enable the occurrence of a target action. A conjunction is defined using the *and*-operator ($\wedge$);
- a *disjunction*: at least one causality condition must be satisfied to enable the occurrence of a target action. A disjunction is defined using the *or*-operator ($\vee$). The target action can refer to the attribute values of only one of the causality conditions;
- a combination of conjunctions and/or disjunctions.

Figure 3-14 depicts examples of conjunction and disjunction.

Figure 3-14
Conjunction and disjunction



In Figure 3-14(i), the occurrence of action *a* depends on the occurrences of actions *b* and *c*. The *and*-operator is graphically expressed by a small black box (the symbol ■). This symbol can be omitted as in Figure 3-14(ii) without changing the causality condition of action *a*. In Figure 3-14(iii), the occurrence of action *a* depends on the occurrence of action *b* or the occurrence of action *c*. The *or*-operator is graphically expressed by a small white box (the symbol ☐). These behaviours are textually expressed in Figure 3-15.

Figure 3-15
Textual expressions of Figure 3-14

$B1 = \{b \wedge c \rightarrow a\}$
$B2 = \{b \wedge c \rightarrow a\}$
$B3 = \{b \vee c \rightarrow a\}$

The *and*-operator and *or*-operator have properties as listed in Table 3-3. Both operators have commutativity and associativity properties. The *and*-operator can be distributed over the *or*-operator, while the inverse does not hold.

Table 3-3
Properties of *and*-operator and *or*-operator

Property	***and*-operator**	***or*-operator**
Commutativity	$a \wedge b = b \wedge a$	$a \vee b = b \vee a$
Associativity	$(a \wedge b) \wedge c = a \wedge (b \wedge c)$	$(a \vee b) \vee c = a \vee (b \vee c)$
Distributivity	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	-

A causality condition of a causality target can be constructed as a disjunctive normal form of alternative causality conditions using those properties. An *alternative causality condition* defines a necessary and sufficient condition for the causality target to occur. An alternative causality condition can be a basic causality condition or a conjunction of basic causality conditions. For example, the causality condition of an action *a* can be $C = C_1 \vee C_2 \vee C_3$, where C is the causality condition of action *a* and C_i ($i = 1, 2, 3$) are basic causality conditions or conjunctions of basic causality conditions. C_i is an alternative causality condition of action *a*.

Uncertainty attribute

An *uncertainty attribute* defines the probability of the occurrence of a causality target when its causality condition is satisfied. It can be

- a *must*: the causality target must occur when the associated condition is satisfied;
- a *may*: the causality target may occur when the associated condition is satisfied.

Figure 3-16 depicts examples of uncertainty attributes. Action *b* may occur from the start. If action *b* occurs, then action *a* must occur. Uncertainty attributes *may* and *must* are respectively expressed by symbols '?' and '!' attached to the associated causality condition. These symbols are textually expressed as subscripts of the associated causality conditions as depicted in Figure 3-17. When a causality condition is not explicitly associated with an uncertainty attribute, it is assumed to have a *must* uncertainty attribute.

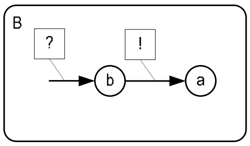


Figure 3-16
Uncertainty attributes

Figure 3-17
Textual expression of Figure 3-16

```
B = {
    √? → b,
    b! → a
}
```

Constraints

Constraints may be added to a causality relation to specify extra conditions on the occurrence of a causality target or on the established values. A constraint can be a causality constraint or attribute constraints.

A *causality constraint* defines the dependency of the occurrence of a causality target on the attribute values that are established by actions in the causality condition. In Figure 3-18, the occurrence of action *a* does not only depends on the occurrences of actions *b* and *c*, but also on the attribute values b.ι and c.ι. Action *a* can only occur if actions *b* and *c* occur, and b.ι > c.ι. The constraint [b.ι > c.ι] is a causality constraint. Causality constraints are textually expressed before the arrow symbols as shown in Figure 3-19. Information type ***N*** represents natural numbers.

An *attribute constraint* defines a restriction on the attribute values that can be established in a causality target. If this constraint cannot be satisfied, the causality target cannot occur. In Figure 3-18, the constraint [b.ι + c.ι = a.ι < 100] of action *a* is an attribute constraint. An attribute constraint may refer to attribute values that have been established by actions in the causality condition, e.g., [a.ι = b.ι + c.ι], or to specific independent values, e.g., [a.ι < 100]. Constraints mentioned in Sections 3.3.1 and 3.3.2 are attribute constraints.

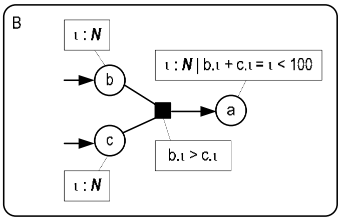


Figure 3-18
Causality constraint and attribute constraint

Figure 3-19
Textual expression of Figure 3-18

```
B = {
    √ → b (ι : N),
    √ → c (ι : N),
    b ∧ c [b.ι > c.ι] → a (ι : N) [b.ι + c.ι = ι < 100]
}
```

3.3.4 Behaviour and sub-behaviours

A behaviour can be structured into sub-behaviours. In a structured behaviour, a causality target in one sub-behaviour may have a causality condition in another sub-behaviour(s). The notations of entry and exit points are used to allow causality relations with causality conditions and causality targets in different behaviours.

An *entry point* in a behaviour represents a causality condition involving actions from other behaviours. An entry point in a behaviour can be used to define (part of) causality condition of a causality target in that behaviour.

An *exit point* in a behaviour represents a causality condition involving actions of that behaviour. An exit point in a behaviour can be used to define (part of) causality condition of a causality target in other behaviours.

Causality relations between causality conditions and causality targets in different behaviours can be defined by connecting entry and exit points of those behaviours.

Figure 3-20 illustrates a behaviour *B* that is structured into two sub-behaviours *B1* and *B2*. Entry and exit points are graphically expressed by small triangles pointing inside and outside of a behaviour, respectively. An entry or exit point has an identifier that is unique within a behaviour. We use natural numbers as identifiers for entry and exit points. Behaviour *B* is called the *super behaviour* of behaviours *B1* and *B2*.

Figure 3-21 textually expresses a structured behaviour *B* in Figure 3-20. An entry and exit points are expressed with keyword 'entry' and 'exit', respectively, appended with their identifiers.

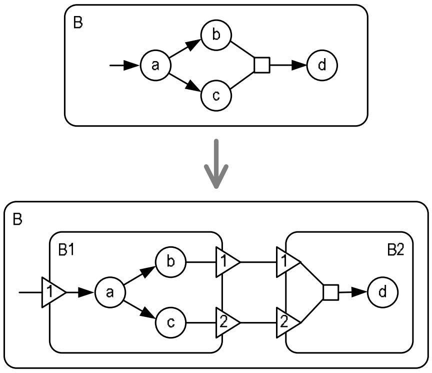


Figure 3-20
Behaviour *B* is structured into sub-behaviours *B1* and *B2*

Figure 3-21
Textual expression of Figure 3-20

```
B = {
    √ → B1.entry1,
    B1.exit1 → B2.entry1,
    B1.exit2 → B2.entry2
}

B1= {
    entry1 → a,
    a → b,
    b → exit1,
    a → c,
    c → exit2
}

B2= {
    entry1 ∨ entry2 → d
}
```

Parameterised entry and exit points

An entry or exit point can be parameterised to pass information, time, or location attribute values from one behaviour to another behaviour. Figure 3-22 depicts parameterised entry and exit points and their value assignments. Parameter *v* of entry point *B1.entry1* is assigned with information attribute values of action *a*. It allows action *b* to indirectly refer to information attribute value of action *a*. Parameters *v1* and *v2* of exit point *B1.exit1* are assigned with information and time attribute values of action *b*, respectively. Parameters *v1* and *v2* of entry point *B2.entry1* are assigned with parameter values *v1* and *v2* of exit point *B1.exit1*, respectively. Figure 3-23 textually expresses these parameterised entry and exit points and their value assignments. Parameter value assignments of an entry point are specified in the super behaviour. Parameter value assignments of an exit point are specified in the behaviour that defines the associated entry point.

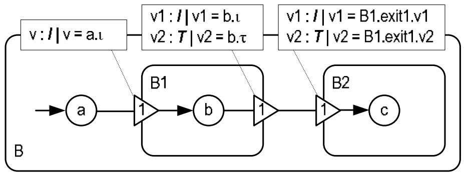


Figure 3-22 Parameters and parameter value assignments

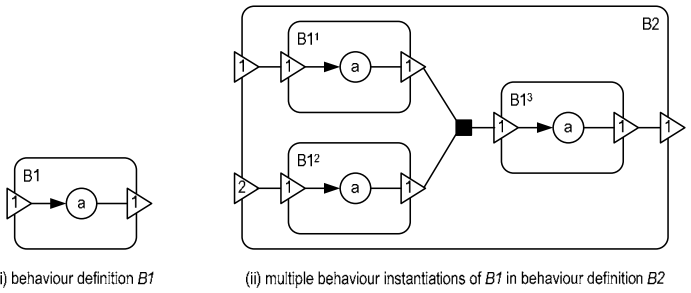
Figure 3-23 Textual expression of Figure 3-22

```
B = {
    √ → a,
    a → B1.entry1
        [v = a.ι],
    B1.exit1 → B2.entry1
        [v1 = B1.exit1.v1,
        v2 = B1.exit.v2]
}

B1= {
    entry1 (v : I) → b,
    b → exit1 (v1 : I, v2 : T)
        [v1 = b.ι,
        v2 = b.τ]
}

B2= {
    entry1 (v1: I, v2 : T) → c
}
```

Behaviour instantiations

A behaviour definition can be used to provide a template to create behaviour instances. A structured behaviour, thus, can possibly define the instantiation of sub-behaviours, i.e., the creation of sub-behaviour instances. Figure 3-24(i) depicts a behaviour definition *B1*. In Figure 3-24(ii), behaviour *B1* is instantiated multiple times in a structured behaviour *B2*. A unique identifier is necessary to distinguish between behaviour instances. A natural number in superscript is used as a behaviour instance identifier, e.g., $B1^1$, $B1^2$, and $B1^3$. This identifier can be omitted if a behaviour definition is instantiated only once in a super behaviour, as in Figure 3-22.

Figure 3-24
Behaviour instantiation


B1
1
a
1
B2
B1¹
1
1
a
1
2
B1²
1
a
1
B1³
1
a
1
1
(i) behaviour definition *B1*
(ii) multiple behaviour instantiations of *B1* in behaviour definition *B2*

A behaviour may define instantiation(s) of itself. This instantiation is called *recursive behaviour instantiation*. Figure 3-25 and Figure 3-26 depicts graphical and textual expressions, respectively, of behaviour *B* that contains recursive behaviour instantiation.

Figure 3-25
Recursive behaviour instantiation

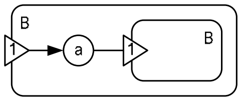


Figure 3-26
Textual expression of Figure 3-25

```
B = {
    entry1 → a,
    a → B.entry1
}
```

Delegated interaction contributions

The contribution of an entity in an interaction may be delegated to a sub-entity. The interaction contribution for that interaction is hence defined in the sub-entity's behaviour; not in the entity's behaviour. For this purpose, the entity's behaviour should contain a *delegated interaction contribution*. This notation allows one to define an interaction between the entity and other entitie(s).

Figure 3-27 depicts an interaction *p* between behaviours *B1* and *B2*. Behaviour *B1* delegates its contribution to interaction *p* to its sub-behaviour *B1a*. A delegated interaction contribution is graphically expressed as a grey segmented ellipse that is connected to an interaction contribution of a sub-behaviour to which the contribution is delegated. Textually, a keyword 'delegated' indicates that constrains between symbols '[' and ']' following that keyword is a delegated interaction contribution, as depicted in Figure 3-28.

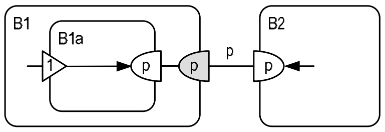


Figure 3-27
Delegated interaction contribution

Figure 3-28
Textual expression of Figure 3-27

```
B1 = {
    delegated [p = B1a.p],
    √ → B1a.entry1
}

B1a = {
    entry1 → p
}

B2 = {
    √ → p
}

p (p : B1.p, p : B2.p)
```

3.3.5 Shorthand notations

Several shorthand notations are introduced to facilitate the modelling of frequently occurring specifications. A shorthand notation is a graphical expression of a certain composition of concepts without abstracting from any design information in that composition of concepts. Shorthand notations used in this thesis are *disabling relation*, *choice relation*, *concurrency relation*, and *repetitive behaviour instantiation*.

Disabling relation

In a disabling condition $\{\neg b \rightarrow a\}$ as depicted in Figure 3-29(i), action *a* may occur if action *b* has not occurred *and* actions *a* and *b* do not occur simultaneously. This non-simultaneous condition can be explicitly defined as $\{a \vee \neg a \rightarrow b\}$, i.e. action *b* can only occur after or before action *a*, as depicted in Figure 3-29(ii). A causality relation which consists of a disabling condition and its corresponding non-simultaneous condition is called *disabling relation*. A shorthand notation for a disabling relation is depicted in Figure 3-29(iii).

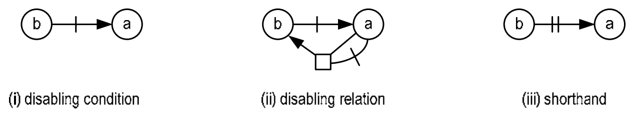


Figure 3-29
Disabling relation

Choice relation

A *choice relation* between two actions *b* and *c* defines that only one of those actions may occur. This relation is depicted in Figure 3-30(i). This relation is modelled as a mutual disabling {¬b → c, ¬c → b}, i.e., the occurrence of action *b* disables the occurrence of action *c*, and vice versa. Shorthand notations for a choice relation are depicted in Figure 3-30(ii) and (iii). The shorthand notation in Figure 3-30(iii) is called *or-split* shorthand.

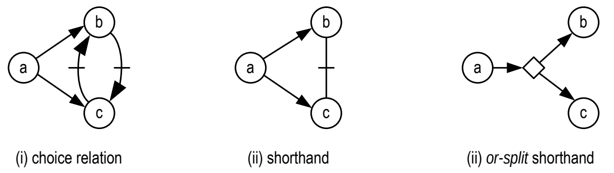


Figure 3-30
Choice relation

Concurrency relation

A *concurrency relation* between two actions *b* and *c* defines the independence of those actions from each other, as depicted in Figure 3-31(i). It is implicitly modelled by the absence of a causality relation between those actions. Alternatively, concurrency can be explicitly modelled using a shorthand notation depicted in Figure 3-31(ii). This shorthand notation is called *and-split* shorthand.

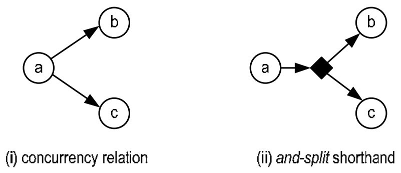


Figure 3-31
Concurrency relation

Repetitive behaviour instantiation

A *repetitive behaviour instantiation* is a behaviour instantiation that repeateadly creates instances of a behaviour as long as a condition holds. This condition is called a *repetition constraint*. It is repetitive because it creates a behaviour instance only after the execution of a previous behaviour instance completes, i.e., all actions have occurred or cannot occur anymore. A repetitive behaviour instantiation can be modelled using a recursive behaviour instantiation. Figure 3-32(i) depicts a repetitive behaviour instantiation to execute action *a* repeatedly as long as repetition constraint *x* holds. It should have one entry point and one exit point only. These entry and exit points should have the same list of parameters, because the parameter values of the exit point of a behaviour instance will be assigned to the parameters of the entry point of the next behaviour instance. A shorthand notation for a repetitive behaviour instantiation is depicted in Figure 3-32(ii). This shorthand includes assignments of parameter values of the exit point of a behaviour instance to parameters of the entry point of the next behaviour instance. Figure 3-33 textually expresses references to the entry and exit point of that repetitive behaviour instantiation. A repetitive behaviour instantiation is expressed with keyword 'Repeat' followed by the name of the behaviour definition to be repeated and the repetition constraint between the symbols '(' and ')'. The repetition constraint is delimited by the symbols '[' and ']'.

Figure 3-32
Repetitive behaviour instantiation

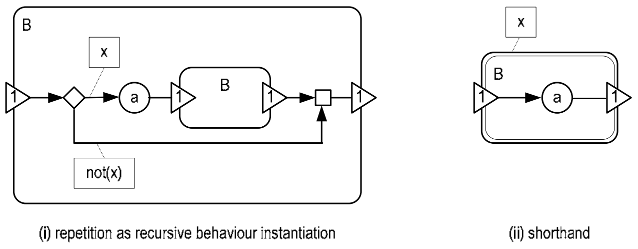


Figure 3-33
Textual expression of Figure 3-32(ii)

Repeat(B[x]).entry1	; entry point *entry1* of repetitive behaviour instantiation *B*
Repeat(B[x]).exit1	; exit point *exit1* of repetitive behaviour instantiation *B*

3.4 Abstract interaction modelling

This section shows the limitations of the ISDL interaction concept for modelling abstract interactions with respect to the criteria of the suitability analysis given in Chapter 2.

An interaction establishes a common result. This result is available from the same time moment and at the same location for all participants. This uniform representation of a result is sufficient for the modelling of an abstraction of direct interactions between participants, in which all participants have the same interest in the result (a case of pattern 1 in Section 2.3.2). However, this uniform representation cannot be used in the modelling of an abstraction of indirect interactions (pattern 2); an abstraction of interactions between three or more participants, in which every interaction is performed by two participants only (pattern 3); and an interaction whose participants have different interests on the result (pattern 4).

Example 1

Figure 3-34 depicts an ISDL model of interactions between a buyer, seller, and bank for a payment in our motivating example (see Section 2.3.1). The payment is done using a money transfer from the buyer's bank account in the Netherlands to the seller's bank account in Switzerland. An invoice indicates that the seller demands a transfer of some amount of money, e.g., CHF 1500, to her bank account 56.002.876. The buyer orders the bank to transfer that amount of money from his bank account 93.123.992. The bank calculates the currency conversion to euro and charges a transfer fee. The buyer hence has to transfer EUR 1100 [$ot_B.\iota_3 = f_1(si_B.\iota_2) + fee$]. Both the buyer and the bank see a uniform representation of the result that EUR 1100 is deducted from the buyer's bank account. After receiving a notification about the payment, the seller checks the balance of her bank account. Both the seller and the bank see a uniform representation of the result that CHF 1500 [$cb_K.\iota_4 = f_2(ot_K.\iota_3) - fee$] has been credited to the seller's bank account.

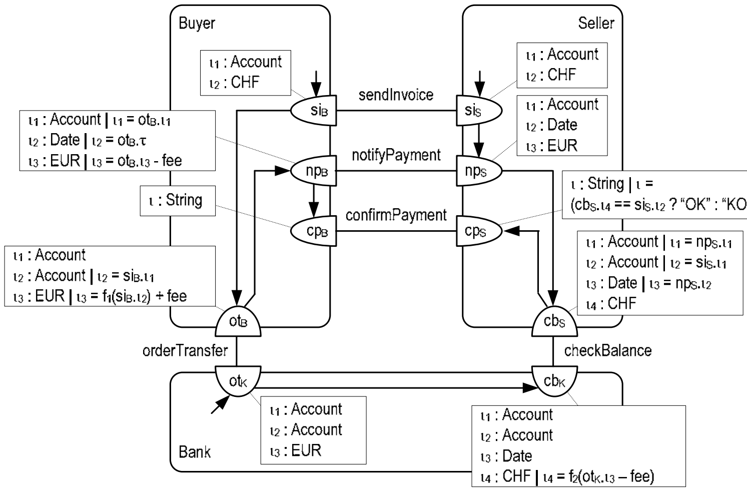


Figure 3-34
Interactions between a buyer, seller, and bank for a payment

At a higher abstraction level, one may want to model this example as a single interaction between the buyer, seller, and bank, as depicted in Figure 3-35. The buyer sees the payment of EUR 1100 and the seller sees the payment of CHF 1500. The bank is interested in transferring the money and collecting the fee for that money transfer. These different views on the result cannot be represented using a uniform representation of the result.

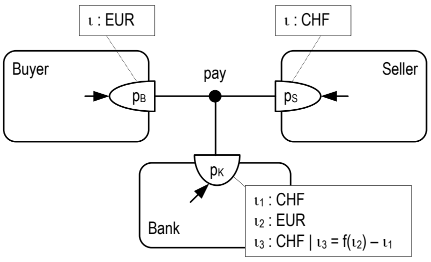


Figure 3-35
An abstract interaction between the buyer, seller, and bank

Example 2
In indirect interactions through an intermediary, participants are typically at different locations. This implies that the result available from different time moments for different participants. Figure 3-36 depicts an ISDL model of indirect interactions between a seller and buyer through a courier for an article delivery in our motivating example (see Section 2.3.1). The seller is in Switzerland and the buyer is in the Netherlands. This article delivery occurs for the buyer when he receives the article. It occurs for the seller when she gets a confirmation from the courier that the buyer has received the article.

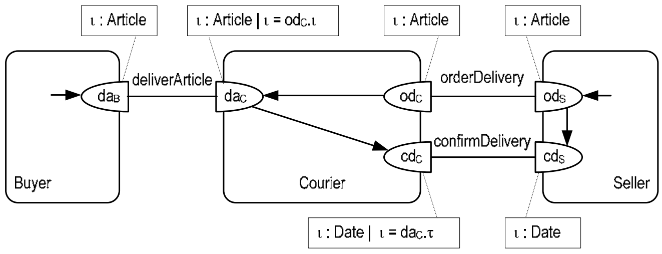


Figure 3-36
Indirect interactions between a seller and buyer through a courier

At a higher abstraction level, one may want to model this example as a single interaction between the buyer and the seller, abstracting from the courier as an intermediary. The buyer and the seller see that the delivery of the article completes at different time moments and locations. For the buyer, the delivery completes when he receives the article. For the seller, the delivery completes when she gets a confirmation indicating that the buyer has received the article. These different views on the result cannot be represented using a uniform representation of the time moment in the result.

Example 3
Modelling the sending of an invoice from the seller to the buyer as an interaction *sendInvoice*, as depicted in Figure 3-34, requires that the invoice should be available from the same time and at the same location for the

seller and buyer. This limits its implementation to a direct interaction between the buyer and seller, i.e., no intermediary may exist between them. An implementation in which the buyer and seller are distributed in different locations is not possible, because such an implementation needs communication middleware between the buyer and seller. Any intermediary makes the implementation also distributed in time.

Conclusion

While a uniform representation of a result is sufficient for modelling concrete interactions, this definition is too restrictive for modelling abstract interactions. The distributed nature of an interaction makes the interaction result available from different time moments and at different locations. At a higher abstraction level, different participants may want to have their own views on the interaction result. Therefore, an enhancement of the ISDL interaction concept is necessary in order to make it fully suitable for modelling abstract interactions.

3.5 Enhanced interaction concept

As mentioned in Section 2.4.5, several ideas for improving ISDL have been suggested in, and beyond, our research group. Most of these ideas, though, have not been formalised yet. In this section, we formalize the improvement on the interaction concept in order to make it satisfy the requirements defined in Section 2.6.

We introduce the concept of *view* that enables us to enhance the interaction concept. The concept of view is not specific to the interaction concept, as it can also be used in the action concept. We then introduce concepts that are specific to the interaction concept: *contribution constraints*, *distribution constraints*, and *interaction synchronisation*.

Contribution constraints and distribution constraints specify the possible results of an interaction. An interaction can only occur if all contribution constraints and distribution constraints can be satisfied.

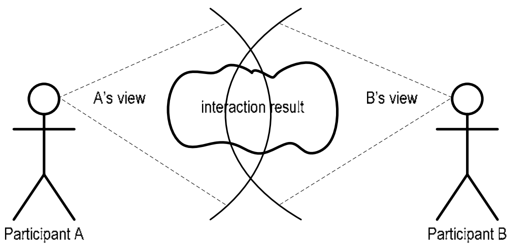
3.5.1 Views

An interaction establishes a common result between participants. A participant has its own view on the established result. A *view* represents a result in terms of information, time, and location attribute values.

Figure 3-37 illustrates this idea. Two participants *A* and *B* may have different views on an interaction result. A participant may want to see the result in the way that is convenient for the participant to refer to that result.

A participant may be interested only in some part of the result. Participants may have overlapped interests in the interaction result.

Figure 3-37
Each participant has its own view on the interaction result



Since different participants may have different views on the established result, different participants may see the result as different sets of information values that are available from different time moments and at different locations. This not only matches the distributed nature of an interaction, it also matches the primary role of abstract specification. Since the abstract specification acts as a prescription for implementation, results of interactions that are not of interest to a participant are not only superfluous and confusing to the specifier, they also lead to superfluous implementation. Therefore, results of an interaction that are not of interest to a participant should be avoided.

For example, in a money transfer interaction as in Section 3.4, a sender sends EUR 1000 from her bank account in the Netherlands on Tuesday; and the receiver receives CHF 1500 (because of currency conversion rate and a charged transfer fee) in his bank account in Switzerland two days later.

In the original interaction concept, we can consider that all participants have the same view on the result, i.e., all participants see the same set of information values that are available from the same time moment and at the same location. In the action concept, an entity that performs an action sees the action result through a view. Since no other entity is involved, there is only one view on the action result.

3.5.2 Contribution constraints

A participant has its conditions for the acceptability of the results of an interaction, and thus must provide the information on the basis of which these results can be calculated. We formulate this by saying that the participant has a responsibility in the establishment of an interaction result. *Contribution constraints* of a participant model the responsibility of that participant in the establishment of an interaction result. They also model the view that the participant has on the result.

For example, in a purchase interaction, the buyer's contribution constraints are that the item to buy should be a bicycle XYZ whose maximum price is EUR 500 (including fees, if any); and that the purchase should occur before 16th April 2010. The seller's contribution constraints are that the item should be a bicycle; that the bicycle should be sold at a price that is higher than the minimum price tagged to that bicycle; that the purchase should occur at any day except Sunday; and that the purchase should occur at his bicycle shop.

The concept of contribution constraint is identical to the concept of attribute constraint that is attached to an interaction in the original interaction concept.

3.5.3 Distribution constraints

A participant's view represents an interaction result in terms of information, time, and location attribute values. Since the interaction result is common to all participants, it makes the views of different participants be related to each other. *Distribution constraints* of an interaction model the relations between participants' views.

For example, in the purchase interaction above, the distribution constraints are that the bicycle bought by the buyer should be the bicycle sold by the seller; that the buying price excluding some fee, if any, should be equal to the selling price; and that the purchase should occur at the same day both for the buyer and the seller. In a money transfer interaction, the distribution constraint is that the money received by the receiver should be equal to the money sent by the sender minus a transfer fee.

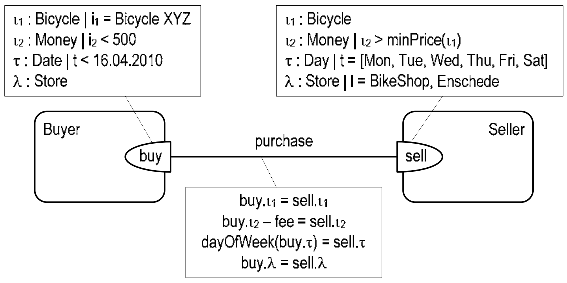
At least one distribution constraint must be specified in an interaction, regardless of which attributes are related by that distribution constraint. For example, two entities need to synchronise their execution at some point of time, without establishing any information attribute value. This can be modelled as an interaction that has one distribution constraint relating the time attributes of the interaction contributions of those entities.

When information attributes are unconstrained, they are not related to each other. Those attributes hence should not be specified in an interaction. When time attributes are unconstrained, no assumption can be made whether information attribute values should be available from the same time moment or from different time moments. Similarly, when location attributes are unconstrained, no assumption can be made whether information attribute values should be available at the same location or at different locations.

The concept of distribution constraint is a new concept that is added to the enhanced interaction concept. In the original interaction concept, distribution constraints are pre-defined, i.e., all participants have the same views on the interaction result.

Figure 3-38 depicts the graphical expression of a purchase interaction between a buyer and seller. Contribution constraints are specified in a text box attached to an interaction contribution. Distribution constraints are specified in a text box attached to the line linking the segmented ellipses, e.g., ‘buy.ι_1 = sell.ι_1’.

Figure 3-38
Interaction *purchase*


ι1 : Bicycle | i1 = Bicycle XYZ
ι2 : Money | i2 < 500
τ : Date | t < 16.04.2010
λ : Store
ι1 : Bicycle
ι2 : Money | ι2 > minPrice(ι1)
τ : Day | t = [Mon, Tue, Wed, Thu, Fri, Sat]
λ : Store | l = BikeShop, Enschede
Buyer
buy
purchase
sell
Seller
buy.ι1 = sell.ι1
buy.ι2 – fee = sell.ι2
dayOfWeek(buy.τ) = sell.τ
buy.λ = sell.λ

The distribution constraints of an interaction are textually expressed between the symbols ‘[’ and ‘]’ after the list of interaction contributions of that interaction as depicted in Figure 3-38.

Figure 3-39
Textual expression of Figure 3-38

```
Buyer = {
    buy (ι₁ : Bicycle, ι₂ : Money, τ : Date, λ : Store)
        [ι₁ = Bicycle XYZ,
        ι₂ < 500,
        τ < 16.04.2010]
}

Seller = {
    sell (ι₁ : Bicycle, ι₂ : Money, τ : Day, λ : Store)
        [ι₂ > minPrice(ι₁),
        τ = [Mon, Tue, Wed, Thu, Fri, Sat]
        λ= BikeShop, Enschede]
}

purchase (buy: Buyer.buy, sell: Seller.sell)
    [buy.ι₁ = sell.ι₁,
    buy.ι₂ – fee = sell.ι₂,
    dayOfWeek(buy.τ ) = sell.τ
    buy.λ = sell.λ]
```

3.5.4 Interaction synchronisation

Interaction synchronisation models the time dependencies between participants on each other via the interaction. Synchronisation implies that the future behaviour of all participants depend on the occurrence of the actions that lead to the interaction. In Figure 3-40, interaction *q* depends on actions *a* and *c*. Consequently, the future behaviours of participants *B1* and *B2* depends indirectly on those actions, i.e., action *b* of participant *B1* not only depends on action *a*, but also depends on action *c* of participant *B2*. Hence, it is possible that $b.\tau < q2.\tau$, but it is always true that $b.\tau < c.\tau$. Figure 3-41 calculates these dependencies.

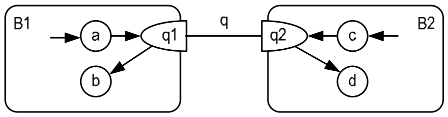


Figure 3-40
Dependencies between interaction *q* and actions of participants

We introduce a notation ‘:→’ to denote an indirect causality relation. If ‘a → b’ and ‘b → c’ then ‘a :→ c’. This notation is useful to show the dependency of our interest without having to show the complete causality relations. This notation is typically used when we draw a conclusion from a dependency calculation.

Figure 3-41
Calculation of indirect dependency between actions of different participants

B1.a → B1.q1	
q (B1.q1, B2.q2)	; interaction q
B2.q2 → B2.d	
B1.a :→ B2.d	; B2.d indirectly depends on B1.a
B2.c → B2.q2	
q (B1.q1, B2.q2)	; interaction q
B1.q1 → B1.b	
B2.c :→ B1.b	; B1.b indirectly depends on B2.c

It should be noted that an action of a participant cannot refer to the result of an action of another participant, unless that result is made available in the interaction. In Figure 3-40, action *d* of participant *B2* cannot directly refer to the result of action *a* of participant *B1*.

3.5.5 Multilateral interactions

The interaction concept allows us to specify an interaction between more than two participants. We call such interactions *multilateral interactions*. For example, an auction is an interaction between an auctioneer and multiple bidders to determine the price of an item. Figure 3-42 depicts an auction interaction between an auctioneer and two bidders. This interaction is graphically expressed by connecting each interaction contribution to a thick black dot. This thick black dot can be omitted for brevity in case only two participants are involved (as in Figure 3-38). This auction interaction is textually expressed in Figure 3-43.

Figure 3-42
Interaction *auction*

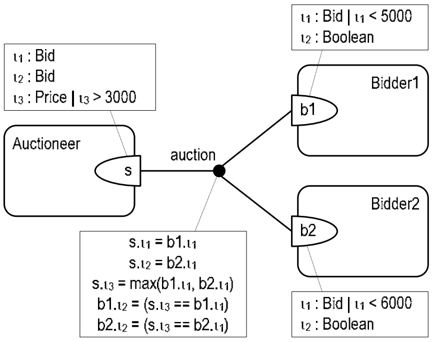


Figure 3-43
Textual expression of Figure 3-42

```
Auctioneer = {
    s (ι1 : Bid, ι2 : Bid, ι3 : Price)
    [ι3 > 3000]
}

Bidder1 = {
    b1 (ι1 : Bid, ι2 : Boolean)
    [ι1 < 5000]
}

Bidder2 = {
    b2 (ι1 : Bid, ι2 : Boolean)
    [ι1 < 5000]
}

auction (s : Auctioneer.s, b1 : Bidder1.b1, b2 : Bidder2.b2)
    [s.ι1 =b1.ι1,
    s.ι2 = b2.ι1,
    s.ι3 = max(b1.ι1, b2.ι1),
    b1.ι2 = (s.ι3 == b1.ι1),
    b1.ι2 = (s.ι3 == b2.ι1)]
```

The auction interaction in Figure 3-42 models a first-price sealed-bid auction [75]. Since a bidder should not be allowed to observe the other bidder's bid, no distribution constraint is defined to relate the bidders' attribute values. Only the auctioneer can see the bids. This is modelled as distribution constraints [$s.\iota_1 = b1.\iota_1$] and [$s.\iota_2 = b2.\iota_1$]. The price of an item is determined by the highest bid: [$s.\iota_3 = \max(b1.\iota_1, b2.\iota_1)$]. Each bidder is notified whether he is the winner or not: [$b1.\iota_2 = (s.\iota_3 == b1.\iota_1)$] and [$b2.\iota_2 = (s.\iota_3 == b2.\iota_2)$]. The contribution constraints specify that the auctioneer demands a price higher than a certain price [$s.\iota_3 > 3000$] and each bidder allows a certain maximum bid: [$b1.\iota_1 < 5000$] and [$b2.\iota_1 < 6000$]. It should be noted that, at this abstraction level, there is no ordering in the establishment of information attribute values.

3.6 Relationships between behavioural concepts

The relationships between behavioural concepts presented in the previous sections can be explained using the conceptual model in Figure 3-44. Concepts in grey are specific to interaction.

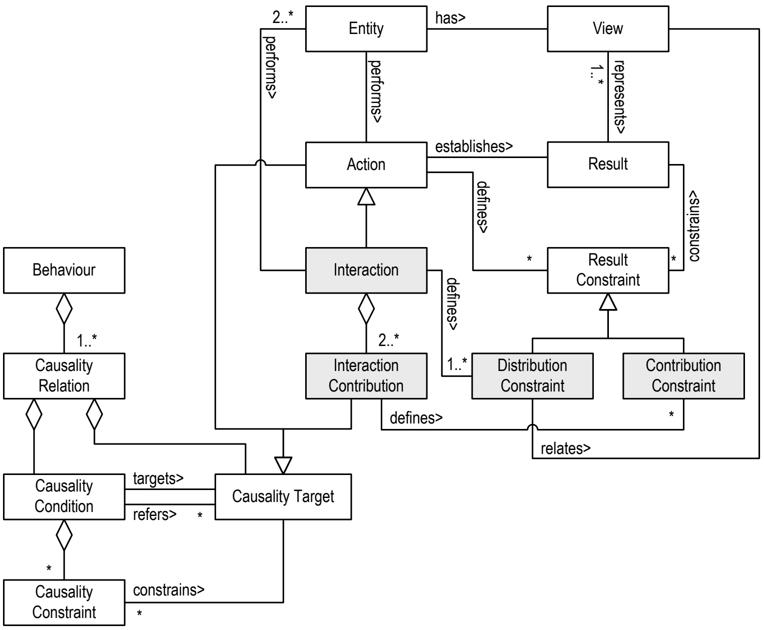


Figure 3-44
Conceptual model of concepts for behaviour modelling

An *action* is a unit of activity that is performed by an *entity* to establish a *result*. An action may define a number of *result constraints* to constrain the result that can be established. In Section 3.3, a *result constraint* is called an *attribute constraint*. An entity has a *view* that represents the established result, in terms of information, time, and location attribute values.

An *interaction* is an action that is performed by two or more entities to establish a common result. An interaction is composed of two or more *interaction contributions*; each of which models the contribution of a participating entity to the interaction. Each participating entity has its own view on the established result. Different participants may have different views on the result.

The result established by an interaction is constrained by zero or more contribution constraints and one or more distribution constraints. A *contribution constraint* is a result constraint that is defined by an interaction contribution. A *distribution constraint* is a result constraint that is defined by an interaction to relate the views of participating entities.

A *behaviour* consists of one or more causality relations. A *causality relation* consists of a causality condition and causality target. A *causality target* can be an action or an interaction contribution.

A *causality condition* defines the condition for the occurrence of a causality target. A causality condition may refer to other causality targets. A causality condition may include causality constraints. A *causality constraint* defines the dependency of the occurrence of a causality target on the result established by other causality targets.

We enhance the ISDL interaction concept by introducing the concept of distribution constraint. We reuse the concepts of interaction, interaction contribution, and contribution constraint that have been previously defined. The concept of contribution constraint was previously called *attribute constraint*.

Another enhancement is the introduction of the concept of view. This concept is not specific to interaction, but it enables us to enhance the interaction concept.

3.7 Shorthand notations

To facilitate interaction modelling, three shorthand notations are introduced: *local interaction*, *remote interaction*, and *message-passing interaction*.

3.7.1 Local interaction

A *local interaction* is an interaction whose distribution constraints specify that the same set of information values should be available for all participants from the same time moment and at the same location. This implies that all participants should have attributes of the same types. An interaction in Figure 3-45(i) has interaction contributions *q1* and *q2* that specify the same attributes of the same types. This interaction has distribution constraints specifying that all participants have the same view on the interaction result.

I, ***T***, and ***L*** are information, time, and location types, respectively. Figure 3-45 (ii) depicts a shorthand notation for this interaction specification. Distribution constraints between information, time, and location attribute values are graphically expressed as double solid lines between segmented ellipses. Textually, this shorthand notation can be indicated with keyword 'local' as depicted in Figure 3-46.

Figure 3-45
Local interaction

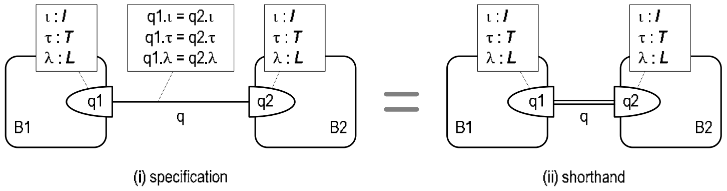


Figure 3-46
Textual expression of a local interaction

```
B1 = {
    q1 (ι : I, τ : T, λ : L)
}

B2 = {
    q2 (ι : I, τ : T, λ : L)
}

q(q1: B1.q1, q2: B2.q2) [local]
```

The original interaction concept of ISDL presented in Section 3.3.2 is a local interaction.

3.7.2 Remote interaction

A *remote interaction* is an interaction whose distribution constraints specify that the same set of information values should be available for all participants, but it can be available from different time moments and at different locations. This implies that all participants should have information attributes of the same types. An interaction in Figure 3-47(i) has interaction contributions *q1* and *q2* that specify the same information attributes of the same types. This interaction has a distribution constraint that specifies that all participants should see the same set of information values representing the interaction result. Time and location attributes are omitted because they are not of interest in this interaction. Figure 3-47 (ii) depicts a shorthand notation for this interaction specification. Distribution constraints between information attribute values are graphically expressed as double lines (one solid line and one dash line) between segmented ellipses. Textually, this shorthand notation can be indicated with keyword 'remote' as depicted in Figure 3-48.

Figure 3-47
Remote interaction

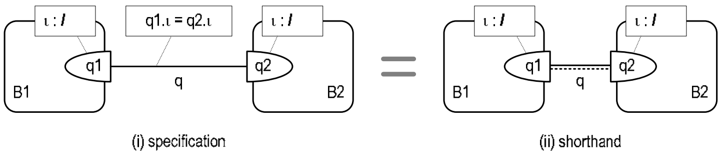


Figure 3-48
Textual expression of a remote interaction

```
B1 = {
    q1 (ι : I)
}

B2 = {
    q2 (ι : I)
}

q(q1: B1.q1, q2: B2.q2) [remote]
```

Distribution constraints that relate between time attribute values or between location attribute values should be defined when those attribute values are of interest in an interaction. Figure 3-49 depicts a remote interaction with a distribution constraint on time attributes, i.e., that the difference between time attribute values should be less than an acceptable delay δ. This remote interaction is textually expressed in Figure 3-50.

Figure 3-49
Remote interaction with a distribution constraint for time attributes

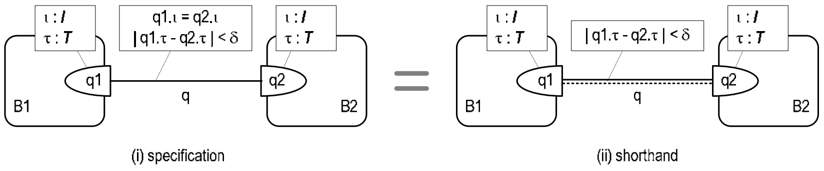


Figure 3-50
Textual expression of a remote interaction with distribution constraint

```
q(q1: B1.q1, q2: B2.q2)
    [remote,
    | q1.τ – q2.τ | <δ]
```

3.7.3 Message-passing communication

A *message-passing communication* represents the exchange of a message between two entities: a *sender* and *receiver*, in which the sender sends a message to the receiver via some communication means and the receiver receives the message. The sender continues its execution immediately after sending the message. Figure 3-51 depicts this behaviour. Behaviours *S*, *R*, and *M* represent the behaviours of the sender, receiver, and communication means, respectively. Since interaction *rcv* depends on interaction *snd*, the receiver depends on action *a* of the sender. The sender does not depend on action *c* of the receiver.

Figure 3-51
Interactions for message-passing communication

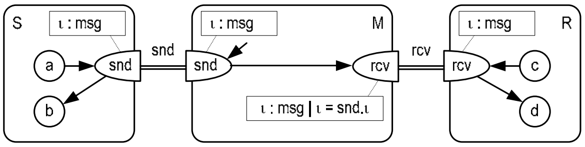
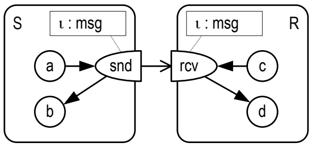


Figure 3-52 depicts a shorthand notation for this behaviour by hiding the behaviour of the communication means. This allows one to model partial or one-way synchronisation. Causal dependency between participants is indicated by an arrow between segmented ellipses. Information attributes of interaction contributions *snd* and *rcv* should be of the same type.

Figure 3-52
Shorthand notation for message-passing communication



A message-passing communication cannot be represented using our interaction concept, because it does not provide full synchronisation between the sender and receiver, i.e., the receiver depends on the sender, while the sender does not depend on the receiver.

3.8 Concluding remarks

In this chapter, we have presented the ISDL design concepts for behaviour modelling of distributed systems. Section 3.4 shows the limitations of the original ISDL interaction concept for modelling abstract interactions. Subsequently, we have enhanced that interaction concept in order to make it fully suitable for that purpose.

Our main enhancement on the interaction concept is that different participants may have different views on the interaction result, i.e., different participants may see different sets of information values that are available from different time moments and at different location. We introduce an interaction property, called *distribution constraints*, to relate those different views.

The enchanced interaction concept satisfies the suitability requirements for modelling abstract interaction, as defined in Section 2.6. It allows one

- to model an interaction between two or more participants,
- to define different views of different participants on the established result,
- to specify the relation between different views of different participants, and
- to express participants' requirements directly.

Therefore, we consider that the enchanced interaction concept is suitable for modelling abstract interactions. The suitability requirement for modelling concrete interactions is addressed later in Chapter 5.

The original ISDL interaction concept is based on synchronous interaction model [44], which we found too limited to use at higher abstraction levels. Synchronous interactions can be easily modelled using our enhanced interaction concept. A shorthand notation called *local interaction* is provided to represent synchronous interactions.

Chapter 4

Interaction design transformations

During a design process, interaction designs are transformed from one abstraction level to another abstraction level. An interaction design transformation can be a refinement or an abstraction. In interaction refinement, an abstract interaction is replaced with a structure of more concrete interactions. Conversely, in interaction abstraction, a structure of interactions is replaced with a more abstract interaction. Each transformation should result in a correct interaction design.

Three basic concepts for behavioural modelling in ISDL are action, interaction, and causality relation, as presented in Chapter 3. ISDL supports two basic types of behaviour refinement: action refinement and causality refinement. Interaction refinement can be done indirectly by using these basic types of refinements, abstraction of an interaction into an action, and refinement of an action into an interaction. We find out that indirect interaction refinement is not sufficient because it loses information about the distribution of responsibility between participants. Direct interaction refinement that maintains that information is therefore necessary.

This chapter presents design transformations for behaviour models of distributed systems in ISDL [44, 107, 110, 111] and extends them with interaction design transformations. This chapter is organised as follows: Section 4.1 presents behaviour refinement. Section 4.2 presents behaviour abstraction. Section 4.3 presents refinement of actions into interactions. These three sections summarise the current state-of-the-art of behaviour transformation in ISDL. The following sections are new contributions. Section 4.4 presents a strategy for indirect interaction refinement and shows that the strategy is not sufficient for transforming interaction designs. Section 4.5 presents direct interaction refinement. Section 4.6 provides a method for assessing the conformance between an abstract and concrete interaction design. Section 4.7 gives guidelines on possible interaction refinement. Section 4.8 discusses related work. Finally, Section 4.9 presents some concluding remarks.

4.1 Behaviour refinement

In a top-down design process, an abstract design is replaced with a more concrete design, i.e. a design that is closer to the real system to be built. We call *behaviour refinement* a design transformation that replaces an abstract behaviour with a more concrete behaviour. The choice of a particular concrete behaviour is determined by specific design objectives. Behaviour refinement allows one to add design details, also called design information, to an abstract behaviour such that the abstract behaviour can be better implemented, preferably with available building blocks.

Behaviour refinement is a creative process in which one creates a concrete behaviour to replace an abstract behaviour. Behaviour refinement can be guided, e.g., by design patterns to satisfy generic requirements as in [16, 46, 54, 70, 129], but in general it cannot be automated.

4.1.1 Conformance

A concrete behaviour should be "correct" with regard to an abstract behaviour. Such correct concrete behaviour preserves the design information defined in the abstract behaviour, while it defines additional design details that do not conflict with the abstract behaviour. When this is the case, the concrete behaviour is said to *conform* to the abstract behaviour.

It is assumed that the occurrence of an abstract action corresponds to the occurrence(s) of one or more concrete actions whose results are equivalent to the desired result of the abstract action. This assumption allows one to compare the abstract behaviour to the concrete behaviour in order to assess the conformance of the concrete behaviour.

Concrete actions whose occurrences determine the occurrence of an abstract action are called *reference actions*, because they are used as reference points in the concrete behaviour for assessing conformance. The occurrence of an abstract action is determined by the occurrence(s) of one or more reference actions, depending on the type of behaviour refinement.

To determine the conformance between abstract and concrete behaviours, the following conformance requirements are identified [110].

- *BR1: Preservation of causality relations*. The causality relations between abstract actions should be preserved by the causality relations between their corresponding concrete actions.

– *BR2: Preservation of attribute values*. The possible resulting attribute values of abstract actions should be preserved by their corresponding concrete actions, i.e., the concrete actions should be able to establish the same results as specified by the abstract actions.

4.1.2 Basic types of behaviour refinement

Two basic types of behaviour refinement are identified: *causality refinement* and *action refinement* [110]. Behaviour refinement may consist of one of these basic types of refinement or a combination of both. Figure 4-1 depicts these basic types of behaviour refinement. These basic types of behaviour refinement are explained below.

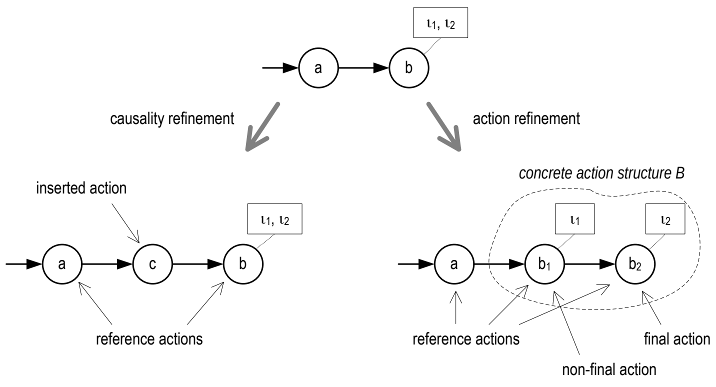


Figure 4-1
Causality refinement and action refinement

Causality refinement

Causality refinement is a behaviour refinement that replaces causality relations between abstract actions with causality relations that involve their corresponding concrete actions and some inserted actions, to model in more detail the relations between those abstract actions. *Inserted actions* are concrete actions that are not reference actions. They are inserted during causality refinement to model additional activities in the concrete behaviour that are not considered in the abstract behaviour.

Each abstract action corresponds to a single reference action in the concrete behaviour. The possible results of an abstract action are preserved by its corresponding reference action.

In Figure 4-1, an abstract behaviour consists of related abstract actions *a* and *b*. Abstract action *b* establishes a result that is represented by two attribute values ι_1 and ι_2. Causality refinement of the enabling relation between abstract actions *a* and *b* results in two enabling relations through an inserted action *c*. Concrete actions *a* and *b* are reference actions that correspond to abstract action *a* and *b*, respectively. In the concrete behaviour, attribute values ι_1 and ι_2 are established by concrete action *b*.

Conformance requirement *BR1* (preservation of causality relations) is addressed in Section 4.2.1. Conformance requirement *BR2* (preservation of attribute values) is interpreted as follows.

– The *information values* of an abstract action should be preserved in the information attribute of the corresponding reference action.
– The *time moment* of an abstract action should be preserved by the time moment of the corresponding reference action.
– The *location* of an abstract action should be preserved by the location of the corresponding reference action.

Action refinement

Action refinement is a behaviour refinement that replaces an abstract action by a composition of more concrete actions and their causality relations, to model in more detail the activity that is represented by that abstract action. The concrete actions and their causality relations are represented in a structure that is called a *concrete action structure*. The attributes of the abstract action are distributed, and allocated to one, more or all of the attributes of the concrete actions.

In Figure 4-1, action refinement of abstract action *b* results in a concrete action structure *B* that consists of concrete actions b_1 and b_2 and their causality relations. Attribute values ι_1 and ι_2 are established by concrete actions b_1 and b_2, respectively. Actions b_1 and b_2 are reference actions that correspond to abstract action *b*.

A concrete action structure establishes the attribute values as specified by the abstract action through the occurrence of one or more reference actions. These reference actions are called the *final actions* of the concrete action structure. Reference actions that are not final actions are called *non-final actions*. In Figure 4-1, concrete action b_2 is the final action of concrete action structure *B*. Concrete action b_1 is a non-final action.

With regard to the actions that depend on the concrete action structure, three basic configurations of final actions are identified:

– *single final action*: a concrete action structure makes all its attribute values available when this final action occurs;
– *conjunction of final actions*: a concrete action structure makes all its attribute values available when all these final actions occur;
– *disjunction of final actions*: a concrete action structure makes all its attribute values available when one of these final actions occurs. Other final actions do not occur.

These basic configurations can be combined into a more complex configuration of final actions.

The concrete action structure in Figure 4-1 has a single final action b_2.

Conformance requirement *BR1* is addressed in Section 4.2.2. Conformance requirement *BR2* is interpreted as follows [110]. The abbreviations (sf), (cf) and (df) indicate single final action, conjunction of final actions, and disjunction of final actions, respectively.

- The *information values* of an abstract action should be preserved in
  - (sf) the information attributes of the final action, or
  - (cf) the union of the information attributes of the final actions, or
  - (df) the information attributes of the actual final action that occurs; and
  - the information attributes of non-final actions that can be referred to via the final actions.

  Information values of the abstract action are established in the final action(s) and/or in the non-final action(s) that can be referred to via the final action(s). As mentioned, the attributes of the abstract action are distributed to one, more, or all of the attributes of the concrete actions. An action that refer to the information attributes of that abstract action should be able to refer to all information attributes of the reference actions. This reference can only be done if the reference actions occur and enable the final actions.
- The *time moment* of an abstract action should be preserved by
  - (sf) the time moment of the final action, or
  - (cf) the time moment of the latest final action, or
  - (df) the time moment of the actual final action that occurs.

  The abstract action occurs when all information values of the concrete action structure are available.
- The *location* of an abstract action should be preserved by
  - (sf) the location of the final action, or
  - (cf) the collection of the locations of the final actions, or
  - (df) the location of the actual final action that occur.

  The location of an abstract action represents the location(s) of the final action(s).

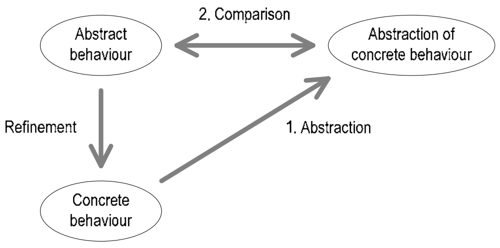
4.1.3 Conformance assessment

Conformance assessment checks whether a concrete behaviour conforms to an abstract behaviour. It consists of the following steps [110] as illustrated in Figure 4-2.

1. Determine the abstraction of the concrete behaviour. This activity is done by applying abstraction rules in order to obtain the abstraction of the concrete behaviour that has the same design details as the original abstract behaviour.
2. Compare the abstraction of the concrete behaviour to the original abstract behaviour. This activity checks whether both abstract

behaviours comply with a certain correctness relation. If this is the case, the concrete behaviour conforms to the abstract behaviour. Otherwise, the concrete behaviour does not conform to the abstract behaviour.

Figure 4-2
Conformance assessment



A correctness relation can be
- an *equivalence relation*: the concrete behaviour preserves all design information of the abstract behaviour; or
- a *partial ordering relation*: the concrete behaviour preserves a subset of design information of the abstract behaviour.

In general, a concrete behaviour is a correct refinement of an abstract behaviour if an equivalence relation holds. However, one may allow a partial ordering relation between an abstract and concrete behaviour. This can be done, for example, to satisfy implementation requirements.

4.2 Behaviour abstraction

Behaviour abstraction is a design transformation that replaces a concrete behaviour with an abstract behaviour. It is the reverse transformation of behaviour refinement in which some design information is abstracted from, we say "removed", from the concrete behaviour. An abstraction of a concrete behaviour is hence determined by the remaining design information. Abstraction rules can be defined to obtain abstractions of concrete behaviours. Thus, in principle, behaviour abstraction can be automated if one knows which design information should be preserved.

In general, the choice of particular design information to be removed is determined by specific objectives. In conformance assessment, behaviour abstraction is used to obtain an abstraction of a concrete behaviour such that the obtained abstraction can be compared to the original abstract behaviour (as depicted in Figure 4-2). Design information that is added during behaviour refinement should be removed when transforming back to the original abstract behaviour.

A method to determine an abstraction of a concrete behaviour consists of the following steps [110].

1. Determine the concrete actions that are considered as reference actions, inserted actions, and final actions in the concrete behaviour.
2. Abstract from inserted actions. This step can be done by using the abstraction method presented in Section 4.2.1.
3. Replace each group of reference actions with an abstract action. This step can be done by using the abstraction method presented in Section 4.2.2.

Two abstraction methods are defined: *abstraction from inserted actions* and *abstraction from final actions*, which correspond to the basic types of behaviour refinement, i.e., causality refinement and action refinement, respectively.

4.2.1 Abstraction from inserted actions

This abstraction method allows one to obtain an abstraction of a concrete behaviour, abstracting from a *single* inserted action. By consecutively abstracting from each single inserted action in any order, one can abstract from multiple inserted actions.

Approach

An inserted action is inserted in the causality relations between reference actions. To abstract from the inserted action, one should determine the concrete actions that are considered as reference actions. These reference actions are called the *causality context* of the inserted action.

A causality relation between actions represents the causal dependencies between the occurrences of those actions in execution. These causal dependencies in execution are called *execution relations*. When an action is inserted in the causality relation between reference actions, the execution relation between those reference actions is defined indirectly via the occurrence of the inserted action. Conversely, an indirect execution relation between two actions via an inserted action can be abstracted from the occurrence of the inserted action.

Multiple indirect execution relations between the same referene actions via the same inserted action can be combined into an *execution structure*. By abstracting from the occurrence of the inserted action in each indirect execution relation, the execution structure can abstract from the occurrence of the inserted action. Based on that, a method for abstracting a behaviour definition from an inserted action is defined.

The following sub-sections explain this approach in more detail.

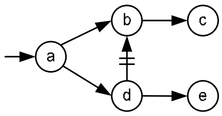
Causality context

The *causality context* of an inserted action only considers actions that are directly related to the inserted action, i.e.,

– actions whose causality conditions contain the inserted action; and
– actions that are parts of the causality condition of the inserted action.

These actions are called *context actions*. The causality context of inserted action *a* is denoted by *Con(a)*. In Figure 4-3, *Con(a)* = {*b*, *d*}, *Con(b)* = {*a*, *c*, *d*}, *Con(c)* = {*b*}, *Con(d)* = {*a*, *b*, *e*}, and *Con(e)* = {*d*}.

Figure 4-3
Example behaviour



An inserted action allows a context action to refer indirectly to the attribute values of another context action. In Figure 4-3, action *b* can be considered as an inserted action that relates actions *a* and *c* indirectly.

Execution relations

A behaviour definition allows multiple possible executions. An *execution* represents the outcome of a possible run of a behaviour definition. The outcome consists of

– action occurrences, which include actions that have occurred and attribute values that are established in those actions; and
– the relations between action occurrences, which are called *execution relations*.

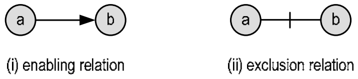
A behaviour definition can be specified as the disjunction of all possible executions.

Two distinct execution relations between actions *a* and *b* are identified (as depicted in Figure 4-4):

– *enabling relation*, which defines the ordering between the occurrences of actions *a* and *b*, such that the occurrence of action *b* depends on the occurrence of action *a*. The occurrence of action *a* can be either independent of the occurrence of action *b*, i.e. $\{\surd \to a, a \to b\}$; or be dependent on the non-occurrence of action *b*, i.e. $\{\neg b \to a, a \to b\}$.
– *exclusion relation*, which defines the choice between the occurrences of actions *a* and *b*, such that the occurrence of action *b* depends on the non-occurrence of action *a*, and vice versa $\{\neg a \to b, \neg b \to a\}$.

Action occurrences are graphically expressed as grey ellipses.

Figure 4-4
Execution relations



An *indirect execution relation* between two actions can be defined via a third action. Figure 4-5 illustrates that the conjunction of two execution relations, i.e., an enabling relation between actions *a* and *c* and another enabling relation between actions *c* and *b*, defines an indirect execution relation between actions *a* and *b* via action *c*.

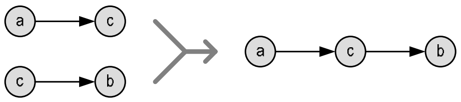


Figure 4-5
Indirect execution relation between actions a and b

Abstraction from indirect execution relations

Two basic rules for abstracting indirect execution relations from an inserted action are defined: *transitivity of enabling* and *inheritance of exclusion* [110].

Transitivity of enabling:

> If a concrete action *x* is an enabling condition of an inserted action *z* and the inserted action *z* is an enabling condition of a concrete action *y*, then an abstract action *x* is an enabling condition of an abstract action *y*.

Figure 4-6 illustrates this rule.

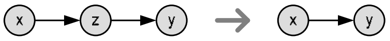


Figure 4-6
Transitivity of enabling

If the condition of inserted action *z* is a start condition √, then the condition of abstract action *y* is a start condition √. Figure 4-7 illustrates this rule.

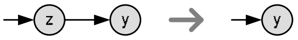


Figure 4-7
Transitivity of enabling: the condition of action z is a start condition

Inheritance of exclusion:

> If an inserted action *z* is an enabling condition of a concrete action *y*, then the exclusion between the inserted action *z* and another concrete action *x* is inherited by abstract actions *y* and *x*.

Figure 4-8 illustrates this rule.

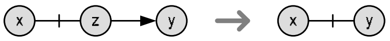


Figure 4-8
Inheritance of exclusion

If none of the above rules is applicable to an indirect execution relation, concrete actions *x* and *y* occur independently. Consequently, abstract actions *x* and *y* also occur independently.

Table 4-1 shows the results of the applications of the abstraction rules for every possible indirect execution relations between concrete actions *x* and *y* when abstracting from an inserted action *z* [110].

Table 4-1
Abstraction rules for indirect execution relations

	$z \rightarrow y$	$z \leftarrow y$	$z -\!\!+\!\!- y$
$x \rightarrow z$	$x \rightarrow y$	$x \quad y$	$x \quad y$
$x \leftarrow z$	$x \quad y$	$x \leftarrow y$	$x -\!\!+\!\!- y$
$x -\!\!+\!\!- z$	$x -\!\!+\!\!- y$	$x \quad y$	$x \quad y$

Legend:

$x \rightarrow z$	represents enabling relation:	$\{\surd \rightarrow x, x \rightarrow z\}$ or $\{\neg z \rightarrow x, x \rightarrow z\}$
$x -\!\!+\!\!- z$	represents exclusion relation:	$\{\neg x \rightarrow z, \neg z \rightarrow x\}$
$x \quad y$	represents independence:	$\{\surd \rightarrow x, \surd \rightarrow y\}$

Execution structures

An *execution structure* is a conjunction of multiple indirect execution relations in which an inserted action relates different pairs of context actions. Conversely, one can determine all possible indirect execution relations from an execution structure. Figure 4-9(i) depicts an execution structure with action *d* as an inserted action. Figure 4-9(ii) depicts all possible indirect execution relations from that execution structure.

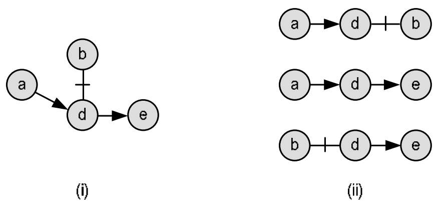


Figure 4-9
Execution structures and its indirect execution relations

A method for abstracting an execution structure from an inserted action consists of the following steps [110].

1. Determine all possible indirect execution relations.
2. Determine the abstraction of each indirect execution relation using the rules defined in Table 4-1.
3. Compose the condition of each abstract action as the conjunction of its condition.
4. If possible, simplify the condition of each abstract action using the following rules: $C \wedge C = C$, $C \vee C = C$, and $\surd \wedge C = C$, where C is an arbitrary causality condition.

Figure 4-10 illustrates the application of this method to abstract an execution structure in Figure 4-10(i) from an inserted action *d*. Figure 4-10(ii) depicts all possible indirect execution relations of that execution structure. Figure 4-10(iii) depicts abstractions of these indirect execution relations. Figure 4-10(iv) depicts the conjunctions of those abstractions.

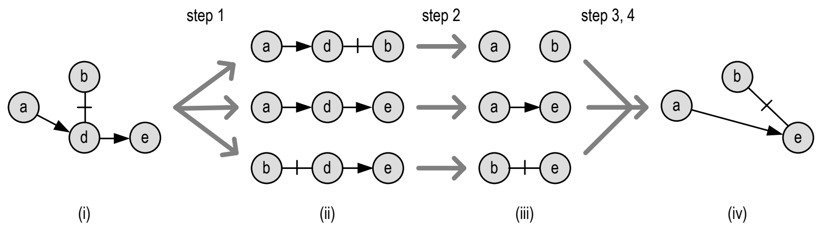


Figure 4-10
Abstraction of an execution structure

An *alternative execution structure* represents a possible execution structure when the causality conditions of an inserted action or its context actions are defined using disjunctions (*or*-operator). For example, Figure 4-11 depicts a disabling relations between actions *a* and *b*. As mentioned in Section 3.3.5, a disabling relation is composed of a disjunction of an enabling relation {¬b → a, a → b} and an exclusion relation {¬a → b, ¬b → a}. Its execution results in either an enabling relation or an exclusion relation. A disabling relation hence has two alternative execution structures.

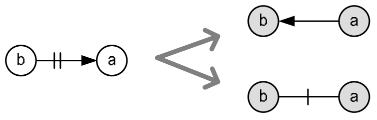


Figure 4-11
Alternative execution structures of a disabling relation

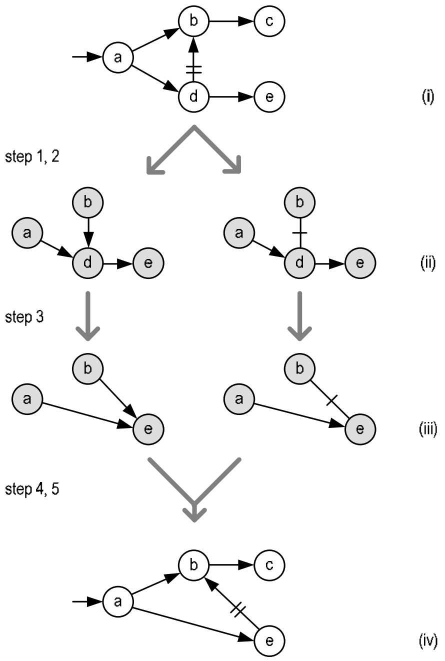
Abstraction method

A method for abstracting a behaviour definition from an inserted action consists of the following steps [110].

1. Determine the causality context of the inserted action.
2. Determine all possible alternative execution structures between the inserted action and its context actions.
3. Determine the abstraction of each execution structure from the inserted action.
4. Compose the condition of each abstract action from the condition of its corresponding concrete action by replacing the conditions of this concrete action in the alternative execution structures that are obtained in Step 2 with the disjunction of the conditions of that abstract action in the execution structures that are obtained in Step 3.
5. If possible, simplify the condition of each abstract action.

Figure 4-12 illustrates an application of this method to abstract the behaviour in Figure 4-12(i) from an inserted action *d*. The causality context of inserted action *d* is *Con(d)* = {a, b, e}. Figure 4-12(ii) depicts all possible alternative execution structures between inserted action *d* and its context actions. A disabling relation between actions *d* and *b* is composed of a disjunction of an enabling relation {¬d → b, b → d} and an exclusion relation {¬d → b, ¬b → d}. Figure 4-12(iii) depicts abstractions of these alternative execution structures. Figure 4-12(iv) depicts the resulted abstract behaviour.

Figure 4-12
Abstraction from an inserted action *d*



An action can refer to other action occurrences. The rule of *transitivity of enabling* enables a context action to refer indirectly to the attributes of another context action that has occurred via an inserted action. This abstraction rule should be applied in combination with the following rule [110].

References to the attributes of an inserted action should be replaced with their possible values or constraints.

When referring to the time attribute of an inserted action, implicit time constraint should be taken into account.

Figure 4-13 illustrates an application of this rule. A concrete action *b* refers to the attributes of an inserted action *c*. Action *c* refers to the attributes of another concrete action *a*. Action *b* hence refers indirectly to the attributes of action *a*. The reference to the information attribute of inserted action *c* is replaced by substituting information attribute constraint of inserted action *c* in the information attribute constraint of action *b*. When replacing the reference to the time attribute of inserted action *c*, the implicit time constraint [c.τ > a.τ] imposed by the causality relation {a → c} should be taken into account.

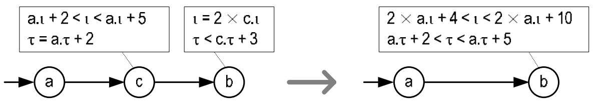


Figure 4-13
Abstraction of indirect attribute references

4.2.2 Abstraction from final actions

This abstraction method allows one to obtain an abstraction of a concrete behaviour, abstracting from a configuration of final actions. A method for replacing final actions of a concrete action structure *A* by an abstract action *a* consists of the following steps. We refer to Figure 4-14 for illustration. Actions a_1 and a_2 are final actions of concrete action structure *A*.

1. Determine the causality relation of abstract action *a*:
    a. Determine the causality condition of abstract action *a* by integrating the causality condition of the final actions.
    b. Determine possible values or constraints of the attributes of abstract action *a*, in terms of the possible values or constraints of the attributes of the final actions by considering conformance requirement *BR2* (preservation of attribute values) for action refinement (as presented in Section 4.1.2).
2. Determine the causality relations of abstract action *b* outside concrete action structure *A* which depends on the abstract action *a*:
    a. Replace the causality condition of abstract actions *b* by the completion condition of concrete action structure *A*, in terms of abstract action *a*.
    b. Replace references to the attributes of the final actions of concrete action structure *A* in abstract actions *b* by references to the corresponding attributes of abstract action *a* as obtained in Step 1b.

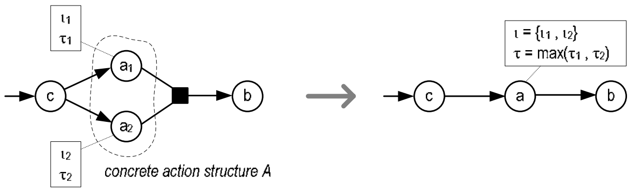


Figure 4-14
Abstraction from final actions

A *completion condition* of a concrete action structure represents the condition for the successful occurrence of the concrete action structure. The completion conditions of the generic cases identified in Section 4.1.2 are defined as follows. The abbreviations (sf), (cf) and (df) indicate single final action, conjunction of final actions, and disjunction of final actions, respectively.

- (sf) The successful occurrence of the concrete action structure corresponds to the occurrence of the single final action;
- (cf) The successful occurrence of the concrete action structure corresponds to the occurrences of all final actions;
- (df) The successful occurrence of the concrete action structure corresponds to the occurrence of a final action.

An application of this method to a concrete action structure *A* in Figure 4-14 is as follows.

- Step 1: the causality condition of abstract action *a* corresponds to the conjunction of the conditions of final actions a_1 and a_2, i.e., $c \wedge c$ which is equal to *c*. The information attribute value of abstract action *a* is the union of the information attribute values of final actions a_1 and a_2 [$\iota = \{\iota_1, \iota_2\}$]. The time attribute value of abstract action *a* corresponds to the time moment of the latest final actions [$\tau = \max(\tau_1, \tau_2)$].
- Step 2: the completion condition of concrete action structure *A* corresponds to the occurrence of all final actions, i.e., $a_1 \wedge a_2$. This completion condition is the causality condition of concrete action *b*, i.e., $\{a_1 \wedge a_2 \rightarrow \text{b}\}$. In the causality relation of abstract action *b*, that causality condition is replaced by an enabling condition *a*, i.e., $\{a \rightarrow b\}$.

It is however not always possible to abstract a concrete action structure into an abstract action using the defined abstraction rules. This might be because of

- incorrect refinement of an abstract action; or
- incorrect determination of the reference actions in the concrete action structure.

4.2.3 Abstraction from repetitive behaviour instantiation

A repetitive behaviour instantiation (see Section 3.3.5) can create a finite or infinite number of behaviour instances, depending on the condition for its repetition. A repetitive behaviour instantiation that repeats an action for a finite number of times can be considered as a result of causality refinement or action refinement of an abstract action. Hence, this repetititive behaviour instantiation can be abstracted back into that abstract action. The abstraction is explained in the following sub-sections.

Abstraction from behaviour instantiation

A super behaviour may abstract from a behaviour instantiation. Indirect attribute references via entry and exit points are replaced with direct attribute references. Figure 4-15 illustrates an abstraction from behaviour instantiation *B*.

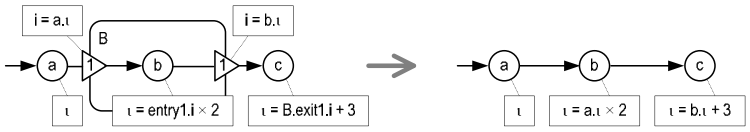


Figure 4-15
Abstraction from behaviour instantiation

Repetitive behaviour instantiation as causality refinement

If a repetitive behaviour instantiation establishes a result as an aggregation of the results of individual behaviour instantiations, it can be considered as causality refinement. Such a repetitive behaviour instantiation that repeats an action *a* for *n* times can be abstracted into an abstract action with attributes:

- $\iota = a^n.\iota = \sum a^k.\iota$; where k = 1..n
- $\tau = a^n.\tau$
- $\lambda = a^n.\lambda$

Action a^n is action *a* in the *n*-th behaviour instance. Here the symbol '$\sum$' represents aggregation in general.

Figure 4-16 depicts an example of such a repetitive behaviour instantiation that repeats behaviour *B* twice. An action *c* refers to parameter values of the exit point of the last behaviour instance B^n.

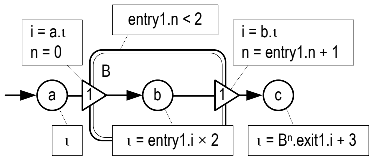


Figure 4-16
Repetitive behaviour instantiation

Figure 4-17 depicts a behaviour definifiton that includes two behaviour instantiations B^0 and B^1 that are equivalent to repetitive behaviour instantiation *B* in Figure 4-16. In behaviour instantiation B^0, action *b* refers to the information attribute of action *a*, i.e., [b.ι = entry1.i × 2] and [entry1.i = a.ι], thus [b.ι = a.ι × 2]. In behaviour instantiation B^1, action *b* refers to the information attribute of action *b* of behaviour instantiation B^0, i.e., [b.ι = entry1.i × 2] and [entry1.i = B^0.i], thus [b.ι = B^0.i × 2].

Figure 4-17
Equivalent behaviour instantiations

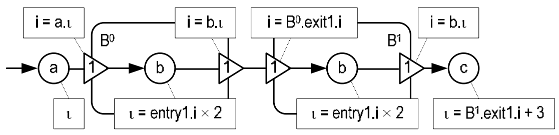
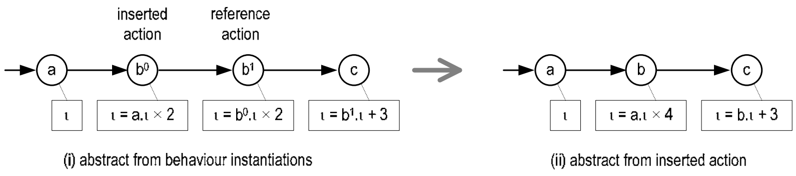


Figure 4-17 can be abstracted from behaviour instantiations B^0 and B^1. This results in Figure 4-18(i). Actions b^0 and b^1 correspond to action *b* in behaviour instantiations B^0 and B^1, respectively. Considering actions b^0 as an inserted action and b^1 as a reference action, one can abstract from action b^0 and obtain the behaviour in Figure 4-18(ii).

Figure 4-18
Abstraction from inserted action



Repetitive behaviour instantiation as action refinement

If a repetitive behaviour instantiation establishes a set of results as a union of the results of individual behaviour instantiations, it can be considered as action refinement. Such a repetitive behaviour instantiation that repeats an action *a* for *n* times can be abstracted into an abstract action with attributes:

- $\iota = \cup\, a^k.\iota$; where k = 1..n
- $\tau = a^n.\tau$
- $\lambda = \cup\, a^k.\lambda$; where k = 1..n

Figure 4-19 depicts an example of such a repetitive behaviour instantiation that repeats behaviour *B* twice. An action *c* refers to parameter values of the exit point of the last behaviour instance B^n.

Figure 4-19
Repetitive behaviour instantiation

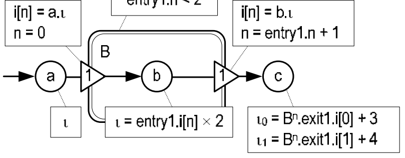


Figure 4-20 depicts a behaviour definifiton that includes two behaviour instantiations B^0 and B^1 that are equivalent to repetitive behaviour instantiation *B* in Figure 4-19. In behaviour instantiation B^0, action *b* refers to the information attribute of action *a*, i.e., [b.ι = entry1.i[0] × 2] and [entry1.i[0] = a.ι], thus [b.ι = a.ι × 2]. In the behaviour instantiation B^1, action *b* again refers to the information attribute of action *a* but via parameter i[1] of the entry point. Behaviour instantiation B^1 maintains parameter values of the exit point of behaviour instantiation B^0.

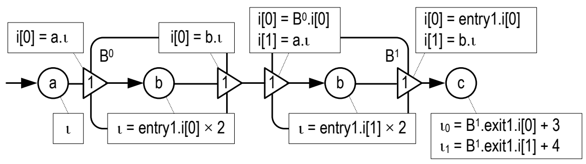
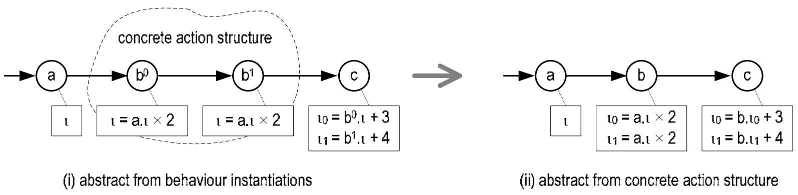


Figure 4-20
Equivalent behaviour instantiations

Figure 4-20 can be abstracted from behaviour instantiations B^0 and B^1. This results in Figure 4-21(i). Considering actions b^0 and b^1 as actions of a concrete action structure with a single final action b^1, one can abstract from this concrete action structure and obtain the behaviour in Figure 4-21(ii).



concrete action structure
a
b⁰
b¹
c
ι
ι = a.ι × 2
ι = a.ι × 2
ι₀ = b⁰.ι + 3
ι₁ = b¹.ι + 4
a
b
c
ι
ι₀ = a.ι × 2
ι₁ = a.ι × 2
ι₀ = b.ι₀ + 3
ι₁ = b.ι₁ + 4
(i) abstract from behaviour instantiations
(ii) abstract from concrete action structure

Figure 4-21
Abstraction from concrete action structure

4.3 Refinement of an action into an interaction

Section 3.3.2 explains that an interaction can be abstracted into an integrated interaction and modelled as an action. Conversely, an action can be refined into an interaction. The conjunction of the causality conditions of interaction contributions of the interaction should be equivalent to the causality condition of the action. Also, the conjunction of attribute constraints of the interaction should be equivalent to attribute constraints of the action. This refinement provides a basis for structuring a behaviour into several interacting behaviours.

Figure 4-22 illustrates the structuring of behaviour *B* into interacting behaviours *B1* and *B2*. Action *a* is refined into interaction *a* whose interaction contributions are a_1 and a_2. Action *a* can occur only after both actions *b* and *c* occur. Similarly, interaction *a* can only occur after both action *b* and *c* occur. The conjunction of the causality conditions of interaction contributions a_1 and a_2 is equal to the causality condition of action *a*.

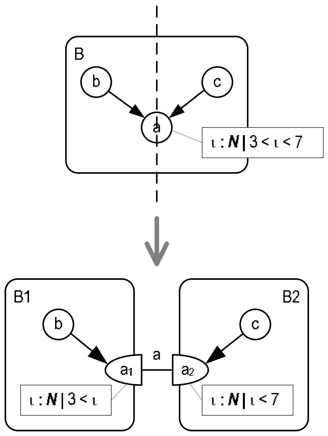


Figure 4-22
Refinement of action into interaction

When action *a* occurs, an information attribute value between 3 and 7 is established [$3 < a.\iota < 7$]. When interaction *a* occurs, a value larger than 3 (contribution constraint of a_1 [$3 < a_1.\iota$]) and lower than 7 (contribution constraint of a_2 [$a_2.\iota < 7$]) is established. The conjunction of the attribute constraints of the interaction contributions a_1 and a_2 is equal to the attribute constraint of action *a*.

4.4 Strategy for interaction refinement

In Section 3.1.2, three basic concepts for behaviour modelling are identified, i.e., *action*, *interaction*, and *causality relation*. However, design transformations presented in the previous sections support only two basic types of behaviour refinement, i.e., *action refinement* and *causality refinement*. Interaction refinement is not considered as a basic type of behaviour refinement.

Interaction refinement is a type of behaviour refinement in which an abstract interaction is replaced with multiple concrete interactions and their causality relations. Using existing design transformations, it can be done using the strategy that is depicted in Figure 4-23 [107]. In Step 1, interacting behaviours *BA1* and *BA2* are integrated into abstract behaviour *BA*. Interaction *b* is abstracted into an integrated interaction and modelled as an abstract action *b*. In Step 2, abstract behaviour *BA* is refined into a concrete behaviour *BC*. In this refinement, abstract action *b* is refined into a concrete action structure consisting of actions b_1 and b_2. Finally, in Step 3, concrete behaviour *BC* is decomposed using constraint-oriented behaviour structuring into two interacting sub-behaviours *BC1* and *BC2*. Actions b_1 and b_2 are refined into interactions b_1 and b_2, respectively.

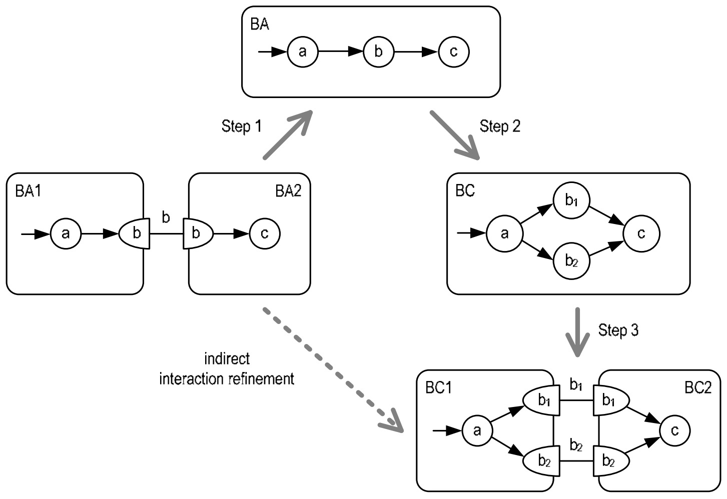


Figure 4-23
Strategy for interaction refinement

This strategy performs interaction refinement indirectly. An abstract interaction *b* between abstract behaviours *BA1* and *BA2* is refined into concrete interactions b_1 and b_2 between concrete behaviours *BC1* and *BC2*. The benefit of this strategy is that the design transformations presented in the previous sections can be reused in interaction refinement.

This strategy, however, cannot preserve *the information about the distribution of responsibility*. The attribute constraints of interaction contributions model the responsibility of participants in the establishment of an interaction result. In Step 1, when one abstracts interaction *b* into an integrated interaction and models it as action *b*, the information about the distribution of responsibility between behaviours *BA1* and *BA2* dissapears.

Consequently, in Step 3, the decomposition of concrete behaviour *BC* into interacting sub-behaviours *BC1* and *BC2* can result in interactions b_1 and b_2 that do not preserve the distribution of responsibility as specified in abstract interaction *b*. Support for direct interaction refinement that preserves the distribution of responsibility between participants is therefore necessary.

4.5 Interaction refinement

Interaction refinement allows one to model in more detail the activity that is represented by an abstract interaction. Concrete interactions and their causality relations are represented in a structure that is called a *concrete interaction structure*.

An abstract interaction specifies *what* result should be established. A corresponding concrete interaction structure specifies *how* to establish that result. To model abstract and concrete interactions, we use the ISDL enhanced interaction concept as defined in Section 3.5.

For example, a purchase of a bicycle between a buyer and a seller is modelled as an abstract interaction *purchase* as depicted in Figure 4-24 and textually in Figure 4-25. Contribution and distribution constraints define the result that should be established.

Figure 4-24
A purchase interaction and its causality relations with other actions

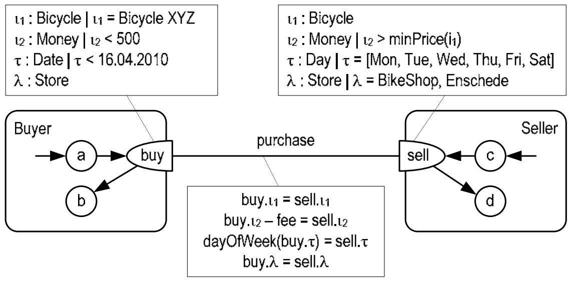

ι_1 : Bicycle | ι_1 = Bicycle XYZ
ι_2 : Money | ι_2 < 500
τ : Date | τ < 16.04.2010
λ : Store
ι_1 : Bicycle
ι_2 : Money | ι_2 > minPrice(ι_1)
τ : Day | τ = [Mon, Tue, Wed, Thu, Fri, Sat]
λ : Store | λ = BikeShop, Enschede
Buyer
a
buy
b
purchase
Seller
sell
c
d
buy.ι_1 = sell.ι_1
buy.ι_2 – fee = sell.ι_2
dayOfWeek(buy.τ) = sell.τ
buy.λ = sell.λ

Figure 4-25
Textual expression of interacting behaviours in Figure 4-24

```
Buyer = {
    √ → a,
    a → buy ($\iota_1$ : Bicycle, $\iota_2$ : Money, $\tau$ : Date, $\lambda$ : Store)
        [$\iota_1$ = Bicycle XYZ,
        $\iota_2$ < 500,
        $\tau$ < 16.04.2010],
    buy → b
}

Seller = {
    √ → c,
    c → sell ($\iota_1$ : Bicycle, $\iota_2$ : Money, $\tau$ : Day, $\lambda$ : Store)
        [$\iota_2$ > minPrice($\iota_1$),
        $\tau$ = [Mon, Tue, Wed, Thu, Fri, Sat]
        $\lambda$= BikeShop, Enschede],
    sell → d
}

purchase (buy: Buyer.buy, sell: Seller.sell)
    [buy.$\iota_1$ = sell.$\iota_1$,
    buy.$\iota_2$ – fee = sell.$\iota_2$,
    dayOfWeek(buy.$\tau$) = sell.$\tau$,
    buy.$\lambda$ = sell.$\lambda$]
```

Figure 4-26 depicts a refinement of abstract interaction *purchase* into a concrete interaction structure *Purchase* that consists of interactions *select*, *pay*, and *deliver*. Interaction *select* models the selection of a bicycle by the buyer from bicycles available in the seller's shop. Interaction *pay* models the payment of a selected bicycle. Interaction *deliver* models the delivery of a purchased bicycle from the seller to the buyer. Concrete actions *a*, *b*, *c*, and *d* correspond to abstract actions *a*, *b*, *c*, and *d*, respectively. Textual expression of these interacting behaviours is depicted in Figure 4-27.

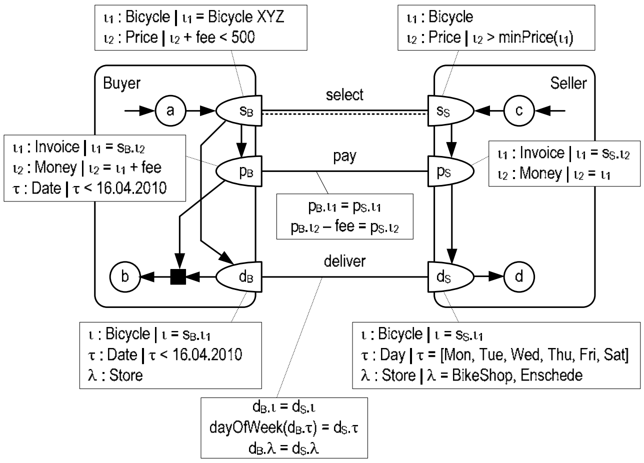

ι1 : Bicycle | ι1 = Bicycle XYZ
ι2 : Price | ι2 + fee < 500
ι1 : Bicycle
ι2 : Price | ι2 > minPrice(ι1)
Buyer
Seller
select
a
sB
sS
c
ι1 : Invoice | ι1 = sB.ι2
ι2 : Money | ι2 = ι1 + fee
τ : Date | τ < 16.04.2010
pay
pB
pS
ι1 : Invoice | ι1 = sS.ι2
ι2 : Money | ι2 = ι1
pB.ι1 = pS.ι1
pB.ι2 – fee = pS.ι2
deliver
b
dB
dS
d
ι : Bicycle | ι = sB.ι1
τ : Date | τ < 16.04.2010
λ : Store
ι : Bicycle | ι = sS.ι1
τ : Day | τ = [Mon, Tue, Wed, Thu, Fri, Sat]
λ : Store | λ = BikeShop, Enschede
dB.ι = dS.ι
dayOfWeek(dB.τ) = dS.τ
dB.λ = dS.λ

Figure 4-26
Refinement of interaction *purchase*

Figure 4-27
Textual expression of interacting behaviours in Figure 4-26

Buyer = {
    $\surd \rightarrow$ a,
    a $\rightarrow s_B$ (ι_1 : Bicycle, ι_2 : Price)
        [ι_1 = Bicycle XYZ,
        ι_2 + fee < 500],
    $s_B \rightarrow p_B$ (ι_1: Invoice, ι_2 : Money, τ : Date)
        [$\iota_1 = s_B.\iota_2$,
        $\iota_2 = \iota_1$ + fee,
        τ < 16.04.2010],
    $s_B \rightarrow d_B$ (ι : Bicycle, τ : Date, λ : Store)
        [$\iota = s_B.\iota_1$,
        τ < 16.04.2010],
    $p_B \wedge d_B \rightarrow$ b
}

Seller = {
    $\surd \rightarrow$ c,
    c $\rightarrow s_S$ (ι_1 : Bicycle, ι_2 : Price)
        [ι_2 > minPrice(ι_1)],

```
    $s_S \rightarrow p_S$ ($\iota_1$ : Invoice, $\iota_2$ : Money)
        [$\iota_1 = s_S.\iota_2$,
        $\iota_2 = \iota_1$],
    $p_S \rightarrow d_S$ ($\iota$ : Bicycle, $\tau$ : Day, $\lambda$ : Store)
        [$\iota = s_S.\iota_1$,
        $\tau$ = [Mon, Tue, Wed, Thu, Fri, Sat],
        $\lambda$ =BikeShop, Enschede],
    $d_S \rightarrow d$
}

select ($s_B$: Buyer.$s_B$, $s_S$: Seller.$s_S$) [remote]

pay ($p_B$: Buyer.$p_B$, $p_S$: Seller.$p_S$)
    [$p_B.\iota_1 = p_S.\iota_1$,
    $p_B.\iota_2$ – fee = $p_S.\iota_2$]

deliver ($d_B$: Buyer.$d_B$, $d_S$: Seller.$d_S$)
    [$d_B.\iota = d_S.\iota$,
    dayOfWeek($d_B.\tau$) = $d_S.\tau$,
    $d_B.\lambda = d_S.\lambda$]
```

For each participant, an abstract interaction contribution is replaced with multiple concrete interaction contributions and their causality relations. These concrete interaction contributions and causality relations are represented in a structure that is called a *concrete interaction contribution structure*. After selecting the bicycle to buy, the buyer is ready to pay for the bicycle and to receive the delivery of the bicycle. The payment and delivery can be done in arbitrary order. The seller requires that the payment should be done before the delivery.

For example, abstract interaction contribution *buy* in Figure 4-24 is replaced with a concrete interaction contribution structure consisting of interaction contributions s_B, p_B, and d_B in Figure 4-26. Attributes of the abstract interaction contribution are distributed over the concrete interaction contribution structure.

A concrete interaction structure may establish attribute values that are not specified in an abstract interaction. These attribute values represent intermediate results. For example, concrete interaction structure *Purchase* establishes attribute values representing an invoice, i.e., $p_B.\iota_1$ and $p_S.\iota_1$, which are not specified in abstract interaction *purchase*.

4.5.1 Completion of a concrete interaction structure

The complete result of a concrete interaction structure is the union of the results that are established by the concrete interactions in that concrete interaction structure. For a participant, the results of the concrete interactions in which the participant is involved or interested represent the complete result from the participant's view (see Section 3.5.1). For that participant, the concrete interaction structure completes when the results of those concrete interactions are available. The interaction contributions of those concrete interactions that allow other actions of that participant to refer to the results are called *final interaction contributions* of that participant.

In Figure 4-26, the complete result of concrete interaction structure *Purchase* from the buyer's view is $\{s_B.\iota_1, s_B.\iota_2, s_B.\tau, s_B.\lambda, p_B.\iota_1, p_B.\iota_2, p_B.\tau, p_B.\lambda, d_B.\iota, d_B.\tau, d_B.\lambda\}$. These attribute values can be referred to by other actions of the buyer, e.g., action *b*, via two final interaction contributions p_B and d_B. The complete result from the seller's view is $\{s_S.\iota_1, s_S.\iota_2, s_S.\tau, s_S.\lambda, p_S.\iota_1, p_S.\iota_2, p_S.\tau, p_S.\lambda, d_S.\iota, d_S.\tau, d_S.\lambda\}$. These attribute values can be referred to by other actions of the seller, e.g., action *d*, via a final interaction contribution d_S.

The interactions in which final interaction contributions are involved are called *final interactions*. Final interactions of concrete interaction structure *Purchase* are interaction *pay* (in which final interaction contributions p_B of the buyer is involved) and interaction *deliver* (in which final interaction contributions d_B of the buyer and d_S of the seller are involved).

4.5.2 Configuration of final interaction contributions

Like in action refinement (see Section 4.1.2), three basic configurations of final interaction contributions are identified:

- *single final interaction contribution*: a concrete interaction structure makes its complete result available from a participant's view when this final interaction contribution occurs;
- *conjunction of final interaction contributions*: a concrete interaction structure makes its complete result available from a participant's view when all these final interaction contributions occur;
- *disjunction of final interaction contributions*: a concrete interaction structure makes its complete result available from a participant's view when one of these final interaction contributions occurs. Other final interaction contributions do not occur.

An interaction contribution occurs when an interaction in which the interaction contribution is involved occurs.

These basic configurations can be combined into a more complex configuration of final interaction contributions. Different participants may have different configurations of final interaction contributions.

The possible attribute values of an abstract interaction contributions should be preserved in the attributes of concrete interaction contribution structure. The interpretation of preservation of attribute values is similar to the cases for action refinement, i.e., when interaction contributions are considered as actions.

4.5.3 Correspondence relation

A *correspondence relation* between an abstract interaction and concrete interaction structure shows which design information in a concrete interaction structure that preserves which design information in an abstract interaction. Specifically, it maps between

- the abstract participants and the concrete participants;
- the attributes of abstract interaction contributions and the attributes of concrete interaction contributions; and
- the occurrences of the abstract interaction and the occurrences of one or more concrete interactions.

Table 4-2 depicts the correspondence relation between abstract interaction *purchase* and concrete interaction structure *Purchase*.

Table 4-2 Correspondence relation between abstract interaction *purchase* and concrete interaction structure *Purchase*

	Abstract interaction	**Concrete interaction structure**
Participants	Buyer	Buyer
	Seller	Seller
Attributes	buy.ι_1	$d_B.\iota$
	buy.ι_2	$p_B.\iota_2$
	buy.τ	max($p_B.\tau$, $d_B.\tau$) [a]
	buy.λ	$d_B.\lambda$
	sell.ι_1	$d_S.\iota$
	sell.ι_2	$p_S.\iota_2$
	sell.τ	$d_S.\tau$
	sell.λ	$d_S.\lambda$
Occurrences	purchase	payment $\wedge$ delivery [b]

[a] see Section 4.1.2 for explanation.

[b] The occurrence of an abstract interaction corresponds to the occurrence of one or more final interactions of a concrete interaction structure. In a correspondence relation, *and*-operator '$\wedge$' indicates that all final interactions must occur and *or*-operator '$\vee$' indicates that one of the final interactions must occur. They should not be confused with a conjunction and disjunction in causality relations.

4.6 Conformance assessment

A concrete interaction structure must conform to the abstract interaction it replaces. Such a concrete interaction structure has more detailed design information, while preserving what has been prescribed by the abstract interaction. The conformance of a concrete interaction structure is assessed by checking whether a set of conformance requirements are satisfied.

4.6.1 Causality context

Our conformance requirements make use of the notion of *causality context* (as presented in Section 4.2.1). The causality context of an interaction consists of actions and interaction contributions that are directly related to the interaction, i.e.,

- actions and interaction contributions whose causality conditions contain the interaction; and
- actions and interaction contributions that are defined in the causality condition of the interaction.

As in Section 4.2.1, those actions are called *context actions*. The notion of causality context can also be applied to an interaction contribution. The causality context of an interaction *q* or an interaction contribution *q* is denoted as *Con(q)*.

In Figure 4-28, *Con(q)* = {*B1.a*, *B1.b*, *B2.c*, *B2.d*}, *Con(B1.q)* = {*B1.a*, *B1.b*}, and *Con(B2.q)* = {*B2.c*, *B2.d*}.

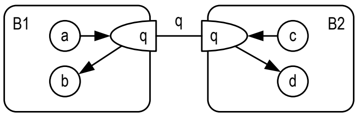


Figure 4-28
Interaction q and its context actions

Similarly, the causality context of an interaction structure consists of actions or interaction contributions that are directly related to the interaction structure, i.e.,

- actions and interaction contributions whose causality conditions contain interaction(s) of the interaction structure; and
- action and interaction contributions that are defined in the causality condition of interactions of the interaction structure, excluding interactions that are parts of the interaction structure.

The notion of causality context can also be applied to an interaction contribution structure. The causality context of an interaction structure consisting of interactions q_i (i = 1, 2, …, n) is denoted as *Con*(q_1, q_2, …, q_n).

In Figure 4-29, *Con(q1, q2)* = {*B1.a*, *B1.b*, *B2.c*, *B2.d*}, *Con(B1.q1, B1.q2)* = {*B1.a*, *B1.b*}, and *Con(B2.q1, B2.q2)* = {*B2.c*, *B2.d*}.

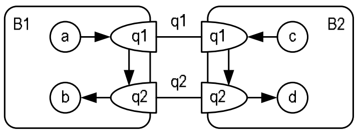
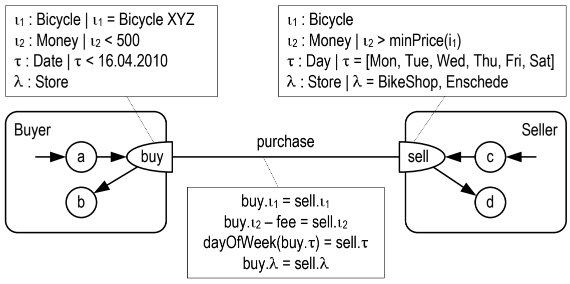


Figure 4-29
A concrete interaction structure and its context actions

4.6.2 Conformance requirements

To determine the conformance between an abstract interaction and a concrete interaction structure, the following conformance requirements are identified. We refer to Figure 4-30 for illustration.

Figure 4-30
A purchase interaction as in Figure 4-24


ι1 : Bicycle | ι1 = Bicycle XYZ
ι2 : Money | ι2 < 500
τ : Date | τ < 16.04.2010
λ : Store
ι1 : Bicycle
ι2 : Money | ι2 > minPrice(i1)
τ : Day | τ = [Mon, Tue, Wed, Thu, Fri, Sat]
λ : Store | λ = BikeShop, Enschede
Buyer
a
buy
b
purchase
Seller
sell
c
d
buy.ι1 = sell.ι1
buy.ι2 – fee = sell.ι2
dayOfWeek(buy.τ) = sell.τ
buy.λ = sell.λ

- *IR1: Preservation of causality relations*. The causality relations between the abstract interaction contribution of an abstract participant and its abstract context actions should be preserved by the (indirect) causality relations between the final interaction contribution(s) of the corresponding concrete participant and the concrete actions that implement the abstract context actions.
  For the buyer, the final interaction contribution(s) of a correct concrete interaction contribution structure should depend (indirectly) on the concrete action that implements abstract action *a*; and the concrete action that implements abstract action *b* should depend on those final interaction contribution(s).
  For the seller, the final interaction contribution(s) of a correct concrete interaction contribution structure should depend (indirectly) on the concrete action that implements abstract action *c*; and the concrete action that implements abstract action *d* should depend on those final interaction contribution(s).

- *IR2: Preservation of contribution constraints*. The contribution constraints of the abstract interaction contribution of an abstract participant should be preserved by the contribution constraints of the concrete interaction contribution structure of the corresponding concrete participant
  For the buyer, a correct concrete interaction contribution structure should have contribution constraints defining that the item to buy

should be a bicycle with a maximum price of EUR 500 (including fees, if any); and that the purchase should occur before 16th April 2010 in a store.
For the seller, a correct concrete interaction contribution structure should have contribution constraints defining that the item to sell should be a bicycle; that the bicycle should be sold at a price that is higher than its minimum price; that the purchase should occur in any day except Sunday; and that the purchase should occur at the seller's bicycle shop.

– *IR3: Preservation of distribution constraints*. The distribution constraints of an abstract interaction should be preserved by the distributon constraints, and possibly the contribution constraints, in a concrete interaction structure.
A correct concrete interaction structure should have distribution constraints, and possibly contribution constraints, defining that the bicycle bought by the buyer should be the bicycle sold by the seller; that the buying price minus some fee, if any, should be equal to the selling price; and that the purchase should occur in the same day and in the same store.

– *IR4: Preservation of interaction synchronisation*. The synchronisation that is provided by an abstract interaction should be preserved by the synchronisation that is provided by a concrete interaction structure.
A correct concrete interaction structure should define the time dependencies of the concrete actions that implement abstract actions *b* and *d* on the concrete actions that implement abstract actions *a* and *c*.

Relationships to conformance requirements for behaviour refinement
Table 4-3 shows the relationships between the conformance requirements for general behaviour refinement (in Section 4.1.1) and for interaction refinement.

Table 4-3
Relationships between conformance requirements for behaviour refinement and for interaction refinement

Behaviour refinement	**Interaction refinement**
BR1: preservation of causality relations	IR1: preservation of causality relations
BR2: preservation of attribute values	IR2: preservation of contribution constraints IR3: preservation of distribution constraints
–	IR4: preservation of interaction synchronisation

The causality target of a causality relation can be a target action or an interaction contribution. Conformance requirement *BR1* is concerned with preservation of causality relations involving target actions. Conformance requirement *IR1* is concerned with preservation of causality relations involving interaction contributions.

A result is represented by attribute values. It is specified by one or more result constraints. In modeling, one specifies result constraints, not attribute values. Therefore, conformance requirement *BR2* should be taken as '*preservation of result constraints*'.

In the conceptual model (as shown in Figure 3-44 in Section 3.6), *result constraint* is specialised into *contribution constraint* and *distribution constraint*. Conformance requirement *BR2* is hence specialised into conformance requirements *IR2* and *IR3*, which deal with the contribution and distribution constraints, respectively.

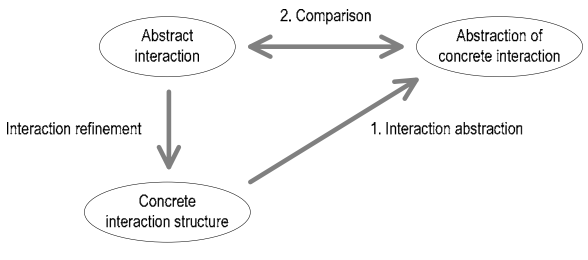
Interaction synchronisation is a property that is specific to an interaction. Hence, conformance requirement *IR4* is not related to conformance requirements *BR1* or *BR2*.

4.6.3 Assessment method

We define an assessment method for interaction refinement that uses the same idea as the assessment method for general behaviour refinement presented in Section 4.1.3. The conformance of the concrete interaction structure can be assessed in the following steps, as illustrated in Figure 4-31.

1. Determine the abstraction of the concrete interaction structure. This activity can be done by applying the abstraction rules and method presented in Section 4.6.4 in order to obtain an abstract interaction that is comparable to the original abstract interaction.
2. Compare the abstraction of the concrete interaction structure with the original abstract interaction. This activity checks whether both abstract interactions comply with a certain correctness relation. If this is the case, the concrete interaction structure conforms to the original abstract interaction. Otherwise, the concrete interaction structure does not conform to the original abstract interaction.

Figure 4-31
Conformance assessment of interaction refinement


2. Comparison
Abstract interaction
Abstraction of concrete interaction
Interaction refinement
1. Interaction abstraction
Concrete interaction structure

Like the assessment method for general behaviour refinement, a correctness relation can be

– an *equivalence relation*: a concrete interaction structure preserves all properties of an abstract interaction.
– a *partial ordering relation*: a concrete interaction structure preserves a subset of the properties of an abstract interaction.

4.6.4 Interaction abstraction

Interaction abstraction is a behaviour abstraction in which a concrete interaction structure is replaced with an abstract interaction. Given a concrete interaction structure, one can abstract it in different ways, resulting in different abstractions. Defining which design information in the concrete interaction structure is considered essential in the abstraction is therefore necessary. This design information has to be preserved when the concrete interaction structure is abstracted into an abstract interaction.

A method for interaction abstraction consists of the following steps.

1. Determine design information that will be preserved. It consists of
   – participants, which implement abstract participants; and
   – attributes of the interaction contributions of the preserved participants, which serve as the participants' views on the result as specified by an abstract interaction.

   This design information can be found in a correspondence relation.
2. Check if every final interaction depends on the same concrete context actions in the preserved participants. The final interactions can be found in the correspondence relation. This is to check whether the concrete interaction structure provides synchronisation as provided by an abstract interaction, i.e, conformance requirement *IR4*. If so, there is a possibility that the concrete interaction structure can be replaced with an abstract interaction. Otherwise, it cannot.
3. For each preserved participant, replace its concrete interaction contribution structure with an abstract interaction contribution. This is done by applying the abstraction rules and methods defined in Section 4.2. The abstract interaction contribution should preserve attributes, as identified in Step 1, and their possible values. A concrete interaction structure can be replaced with an abstract interaction only if the concrete interaction contribution structure in each preserved participant can be replaced with an abstract interaction contribution.
4. Form an abstract interaction. This step consists of the following steps.
   a. Connect all the abstract interaction contribution obtained in Step 4 to each other.
   b. Determine distribution constraints of the abstract interaction from the constraints in the concrete interaction structure. These distribution constraints are composed from

– the distribution constraints in the concrete interaction structure that involve the preserved attributes, and/or
– the contribution constraints of non-preserved participants that allows a preserved attribute to refer indirectly to another preserved attribute.

Attribute substitution is necessary to eliminate the attributes of non-preserved participants from the distribution constraints of the abstract interaction.

For illustration, we apply the abstraction method to the concrete interaction structure *Purchase* in Figure 4-32. In Step 1, we determine that participants *Buyer* and *Seller* should be preserved. The attributes of the concrete interactions as indicated in Table 4-2 should be preserved as their values correspond to the result of the abstract interaction.

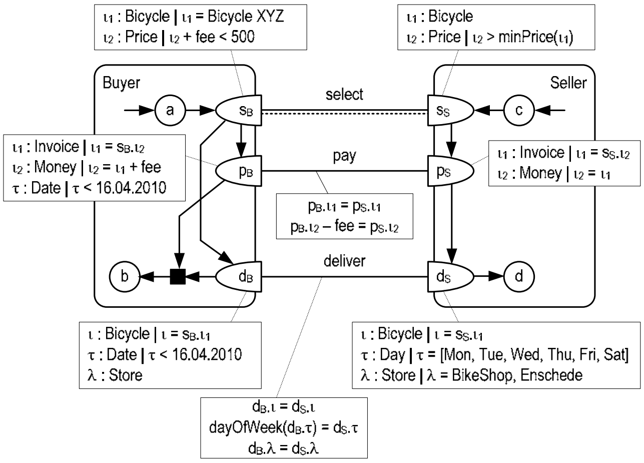

ι1 : Bicycle | ι1 = Bicycle XYZ
ι2 : Price | ι2 + fee < 500
ι1 : Bicycle
ι2 : Price | ι2 > minPrice(ι1)
Buyer
Seller
select
pay
deliver
a
b
c
d
sB
pB
dB
sS
pS
dS
ι1 : Invoice | ι1 = sB.ι2
ι2 : Money | ι2 = ι1 + fee
τ : Date | τ < 16.04.2010
ι1 : Invoice | ι1 = sS.ι2
ι2 : Money | ι2 = ι1
pB.ι1 = pS.ι1
pB.ι2 – fee = pS.ι2
ι : Bicycle | ι = sB.ι1
τ : Date | τ < 16.04.2010
λ : Store
ι : Bicycle | ι = sS.ι1
τ : Day | τ = [Mon, Tue, Wed, Thu, Fri, Sat]
λ : Store | λ = BikeShop, Enschede
dB.ι = dS.ι
dayOfWeek(dB.τ) = dS.τ
dB.λ = dS.λ

Figure 4-32
A concrete interaction structure *Purchase* as in Figure 4-26

Table 4-2 indicates that the final interactions are interactions *pay* and *deliver*. In Step 2, we observe that final interaction *pay* depends on actions *a* and *c* (via interaction *select*) and final interaction *deliver* depends on actions *a* and *c* (via interaction *select*). Each final interaction depends on the same context actions. Conformance requirement *IR4* is satisfied.

Step 3 results in abstract interaction contributions *q1* and *q2* in abstract participants *Buyer* and *Seller*, respectively, as depicted in Figure 4-33. The correspondences between the attributes of the concrete and abstract interaction contributions are listed in Table 4-4.

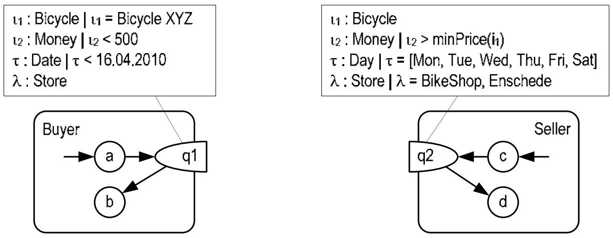

ι1 : Bicycle | ι1 = Bicycle XYZ
ι2 : Money | ι2 < 500
τ : Date | τ < 16.04.2010
λ : Store
ι1 : Bicycle
ι2 : Money | ι2 > minPrice(i1)
τ : Day | τ = [Mon, Tue, Wed, Thu, Fri, Sat]
λ : Store | λ = BikeShop, Enschede
Buyer
a
q1
b
Seller
q2
c
d

Figure 4-33
Abstract interaction contributions *q1* and *q2* obtained from Step 4

Table 4-4
Correspondences between attributes of concrete and abstract interaction contributions

Concrete interaction structure	Abstract interaction
$d_B.\iota$	$q1.\iota_1$
$p_B.\iota_2$	$q1.\iota_2$
$p_B.\tau$ or $d_B.\tau$	$q1.\tau$
$d_B.\lambda$	$q1.\lambda$
$d_S.\iota$	$q2.\iota_1$
$p_S.\iota_2$	$q2.\iota_2$
$d_S.\tau$	$q2.\tau$
$d_S.\lambda$	$q2.\lambda$

Step 4 results in abstract interaction *q* in Figure 4-34. Information attributes $d_B.\iota$ and $d_S.\iota$ are involved in the distribution constraint $[d_B.\iota = d_S.\iota]$. Given the attribute correspondences in Table 4-4, this constraint can be replaced with $[q1.\iota_1 = q2.\iota_1]$. Information attributes $p_B.\iota_2$ and $p_S.\iota_2$ are involved in the distribution constraint $[p_B.\iota_2 - \text{fee} = p_S.\iota_2]$. This constraint can be replaced with $[q1.\iota_2 - \text{fee} = q2.\iota_2]$. Time attributes $d_B.\tau$ and $d_S.\tau$ are involved in the distribution constraint $[\text{dayOfWeek}(d_B.\tau) = d_S.\tau]$. This constraint can be replaced with $[\text{dayOfWeek}(q1.\tau) = q2.\tau]$. Location attributes $d_b.\lambda$ and $d_s.\lambda$ are involved in the distribution constraint $[d_B.\lambda = d_S.\lambda]$. This constraint can be replaced with $[q1.\lambda = q2.\lambda]$.

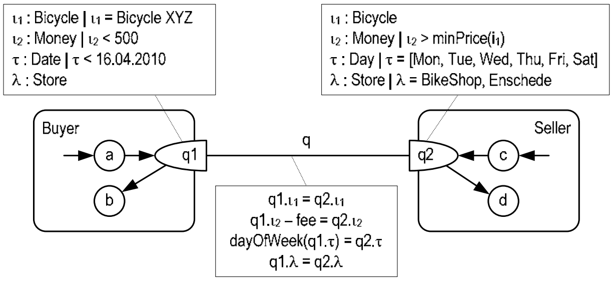

ι1 : Bicycle | ι1 = Bicycle XYZ
ι2 : Money | ι2 < 500
τ : Date | τ < 16.04.2010
λ : Store
ι1 : Bicycle
ι2 : Money | ι2 > minPrice(i1)
τ : Day | τ = [Mon, Tue, Wed, Thu, Fri, Sat]
λ : Store | λ = BikeShop, Enschede
Buyer
a
q1
b
q
Seller
q2
c
d
q1.ι1 = q2.ι1
q1.ι2 – fee = q2.ι2
dayOfWeek(q1.τ) = q2.τ
q1.λ = q2.λ

Figure 4-34
Abstract interaction *q* obtained from Step 4

Sections 4.7.2 and 4.7.4 show the examples of composing the distribution constraints of an abstract interaction from the contribution constraints of non-preserved participant.

4.6.5 Comparison to original abstract interaction

In Section 4.6.2, four conformance requirements *IR1*, *IR2*, *IR3*, and *IR4* are defined. During the abstraction process, conformance requirement *IR4* is checked. When the abstraction of a concrete interaction structure can be obtained, this means that conformance requirement *IR4* is satisfied. The comparison of the abstraction of the concrete interaction structure and the original abstract interaction is necessary to check whether conformance requirements *IR1*, *IR2*, and *IR3* are satisfied.

In Figure 4-34, the causality relations involving abstract interaction contributios are {a → q1, c → q2, q1 → b, q2 → d}. These causality relations are equivalent to the causality relations {a → buy, c → sell, c → sell, sell → d} respectively as depicted in Figure 4-24, in which interaction contributions *buy* and *sell* are replaced with *q1* and *q2*, respectively. Conformance requirement *IR1* is satisfied.

Contribution constraints of interaction *q* and contribution constraints of interaction *purchase* are equivalent. Conformance requirement *IR2* is satisfied. Distribution constraints of interaction *q* and distribution constraints of interaction *purchase* are equivalent. Conformance requirement *IR3* is satisfied.

All conformance requirements are satisfied. Interactions *purchase* and *q* comply with an equivalence correctness relation. We conclude that the concrete interaction structure *Purchase* conforms to abstract interaction *purchase*.

4.7 Patterns for interaction refinement

To give guidelines on possible interaction refinement, we identify four patterns for interaction refinement: *interface decomposition*, *functionality delegation*, *functionality distribution*, and *intermediary introduction*. These patterns are generic cases of interaction refinement. Each pattern captures a possible way to refine an abstract interaction. Other patterns are possible.

An interaction refinement may apply one of those patterns or a combination of them. In the following sections, those patterns are described and illustrated. Context actions are shown in order to indicate the relations between the resulting concrete interaction structure and its context actions.

4.7.1 Interface decomposition

In this pattern, an abstract interaction between abstract participants is replaced with a concrete interaction structure between concrete participants that corresponds to the abstract participants. All participants engage in every concrete interaction. This pattern allows one to replace an abstract interaction with, e.g., a sequence of concrete interactions representing intermediate steps to establish the result or a number of alternative concrete interactions to establish the result. The concrete interaction contributions in a concrete participant may form a structure that differs from the structure of the concrete interaction contributions in other concrete participants.

Figure 4-35 illustrates this pattern. An abstract interaction *q* between participants *B1* and *B2* is replaced with a concrete interaction structure *Q* consisting of interactions *q1*, *q2*, and *q3* between concrete participants *B1* and *B2*. Concrete participants *B1* and *B2* correspond to abstract participants *B1* and *B2*, respectively. The causality relations in participant *B1* differs from the causality relation in participant *B2*. Concrete context actions *a*, *b*, *c*, and *d* correspond to abstract context actions *a*, *b*, *c*, and *d*, respectively.

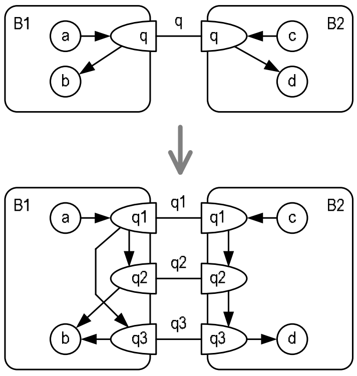


Figure 4-35
Interface decomposition pattern

Example

The refinement of abstract interaction *purchase* in Figure 4-24 into a concrete interaction structure *Purchase* in Figure 4-26 is an application of this pattern. The seller implements the abstract interaction in three sequential steps, i.e., *select*, *pay* and *deliver*. The buyer also implements the abstract interaction in three similar steps, but interactions *pay* and *deliver* can be done independent of each other.

The conformance assessment of the example is presented in Sections 4.6.4 and 4.6.5.

4.7.2 New participants introduction

In this pattern, an abstract interaction between abstract participants is replaced with a concrete interaction structure that consists only of one concrete interaction between concrete participants that corresponds to the abstract participants and one or more new concrete participant(s). This pattern allows one to delegate some functionality necessary to perform the interaction to the new concrete participant(s). For example, by introducing a bank, some functionality of a payment interaction can be delegated to that bank.

Figure 4-36 illustrates this pattern. An abstract interaction *q* between participants *B1* and *B2* is replaced with a concrete interaction structure consisting of single concrete interaction *q'* between concrete participants *B1*, *B2*, and *B3*. Concrete participant *B3* is introduced in the refinement.

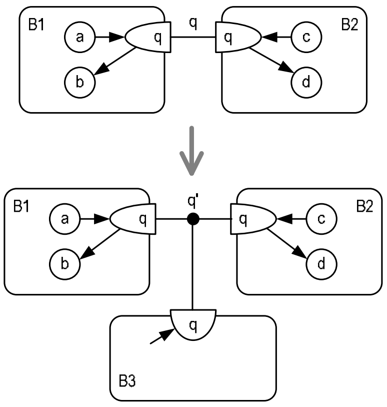
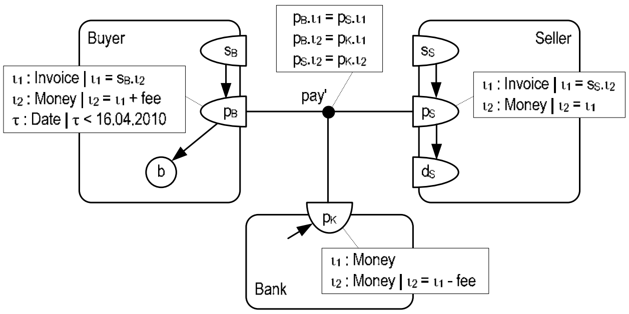


Figure 4-36
Functionality delegation pattern

Example

Figure 4-37 depicts a concrete interaction structure *pay* obtained from an application of this pattern in the refinement of abstract interaction *pay* in Figure 4-26. In the refinement, a bank is introduced. For brevity, the figure shows only the concrete interaction and its context actions. The bank charges the fee in interaction *pay*, as specified in the contribution constraint of interaction contribution p_K [$p_K.\iota_2 = p_K.\iota_1$ – fee]. As this functionality has been delegated to the bank, the fee is no longer specified in any distribution constraint. Table 4-5 depicts the correspondence relation between abstract interaction *pay* and concrete interaction *pay'*.

Figure 4-37
Example of functionality delegation



Buyer
sB
ι1 : Invoice | ι1 = sB.ι2
ι2 : Money | ι2 = ι1 + fee
τ : Date | τ < 16.04.2010
pB
b
pB.ι1 = pS.ι1
pB.ι2 = pK.ι1
pS.ι2 = pK.ι2
pay'
Seller
sS
ι1 : Invoice | ι1 = sS.ι2
ι2 : Money | ι2 = ι1
pS
dS
pK
ι1 : Money
ι2 : Money | ι2 = ι1 - fee
Bank

Table 4-5
Correspondence relation between abstract interaction *pay* and concrete interaction *pay'*

	Abstract interaction	**Concrete interaction structure**
Participants	Buyer	Buyer
	Seller	Seller
Attributes	$p_B.\iota_1$	$p_B.\iota_1$
	$p_B.\iota_2$	$p_B.\iota_2$
	$p_B.\tau$	$p_B.\tau$
	$p_S.\iota_1$	$p_S.\iota_1$
	$p_S.\iota_2$	$p_S.\iota_2$
Occurrences	pay	pay'

The conformance assessment is described as follows. In Step 1, we determine that participants *Buyer* and *Seller* and attributes $p_B.\iota_1$, $p_B.\iota_2$, $p_S.\iota_1$, and $p_S.\iota_2$ are the design information that should be preserved.

In Step 2, we observe that the only interaction *pay'* depends on context action s_B of *Buyer* and s_S of *Seller*. Conformance requirement *IR4* is satisfied.

Step 3 results in abstract interaction contributions *q1* and *q2* in abstract participants *Buyer* and *Seller*, respectively. The correspondences between the attributes of the concrete and abstract interaction contributions are listed in Table 4-4.

Table 4-6
Correspondences between attributes of concrete and abstract interaction contributions

Concrete interaction	**Abstract interaction**
$p_B.\iota_1$	$q1.\iota_1$
$p_B.\iota_2$	$q1.\iota_2$
$p_B.\tau$	$q1.\tau$
$p_S.\iota_1$	$q2.\iota_1$
$p_S.\iota_2$	$q2.\iota_2$

In Step 4, we observe that information attributes $p_B.\iota$ and $p_S.\iota$ are involved in distribution constraint $[p_B.\iota = p_S.\iota]$. Given the attribute correspondences in Table 4-6, this constraint can be replaced with $[q1.\iota_1 = q2.\iota_1]$. Information attribute $p_B.\iota_2$ and $p_S.\iota_2$ are involved in distribution constraints [$p_B.\iota_2 = p_K.\iota_1$] and [$p_S.\iota_2 = p_K.\iota_2$], respectively; while $p_K.\iota_1$ and $p_K.\iota_2$ are involved in contribution constraint [$p_K.\iota_2 = p_K.\iota_1 - \text{fee}$]. Substituting $p_B.\iota_2$ and $p_B.\iota_2$ for $p_K.\iota_1$ and $p_K.\iota_2$ in the contribution constraint, respectively, results in [$p_S.\iota_2 = p_B.\iota_2 - \text{fee}$]. This constraint can be replaced with [$q2.\iota_2 = q1.\iota_2 - \text{fee}$]. Time attribute $p_B.\tau$ is not involved in any distribution constraint. No distribution constraint of the abstract interaction can be specified for this time attribute.

The application of the abstraction method results in an abstract interaction that has an equivalence correctness relation with abstract interaction *pay*. Concrete interaction *pay'* conforms to abstract interaction *pay*.

4.7.3 Bilateral interactions transformation

In this pattern, an abstract interaction between three or more abstract participants is replaced with a concrete interaction structure in which every concrete interaction is performed by exactly two concrete participants that implement the abstract participants, i.e., a bilateral interaction. The functionality of the abstract interactions is distributed over interactions between pairs of concrete participants.

In an implementation, interactions are carried out by interaction mechanisms provided by, e.g., communication middleware. Current middleware mostly supports interaction mechanisms between two entities. This pattern allows one to replace an abstract interaction with a concrete interaction structure that will be implemented using such interaction mechanisms.

Figure 4-38 illustrates this pattern. An abstract interaction *q* between abstract participants *B1*, *B2*, and *B3* is replaced with a concrete interaction structure consisting of interactions *q1*, *q2*, and *q3*. Concrete interaction *q1* is an interaction between concrete participants *B1* and *B2*; concrete interaction *q2* is an interaction between concrete participants *B1* and *B3*; and concrete interaction *q3* is an interaction between concrete participants *B2* and *B3*. Each concrete interaction is performed by two concrete participants only.

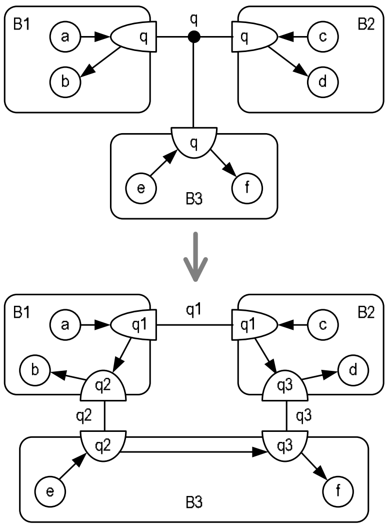


Figure 4-38
Functionality distribution pattern

Example

Figure 4-39 depicts a concrete interaction structure *Pay* obtained from an application of this pattern in the refinement of interaction *pay* in Figure 4-37. The seller first makes a payment request by sending an invoice to the buyer. The buyer then interacts with the bank to transfer the requested amount of money plus a transfer fee. After transfering the money, the buyer sends a notification indicating the date the money is transfered. The seller then checks her bank account's balance. If the money she received is equal to the money she requested, the seller confirms the buyer that the payment is successful. Table 4-7 depicts the correspondence relation between abstract interaction *pay'* and concrete interaction *Pay*.

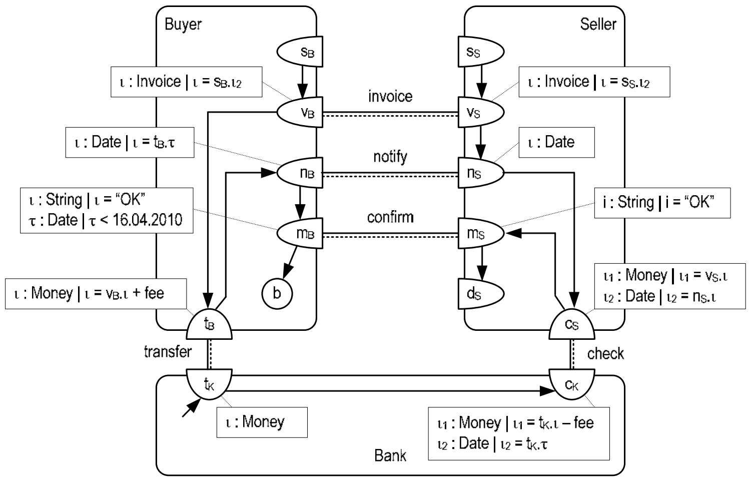


Figure 4-39
Example of functionality distribution

Table 4-7
Correspondence relation between abstract interaction *pay'* and concrete interaction structure *Pay*

	Abstract interaction	**Concrete interaction structure**
Participants	Buyer	Buyer
	Seller	Seller
	Bank	Bank
Attributes	$p_B.\iota_1$	$v_B.\iota$
	$p_B.\iota_2$	$t_B.\iota$
	$p_B.\tau$	$m_B.\tau$
	$p_S.\iota_1$	$v_S.\iota$
	$p_S.\iota_2$	$c_S.\iota_1$
	$p_K.\iota_1$	$t_K.\iota$
	$p_K.\iota_2$	$c_K.\iota_1$
Occurrences	pay'	confirm $\wedge$ check

The conformance assessment is described as follows. In Step 1, we determine that participants *Buyer*, *Seller*, and *Bank* and attributes $v_B.\iota$, $t_B.\iota$, $m_B.\tau$, $v_S.\iota$, $c_S.\iota_1$, $t_K.\iota$, and $c_K.\iota$ are the design information that should be preserved.

In Step 2, we observe that each final interaction depends on the same context actions, i.e., s_B of *Buyer*, s_S of *Seller*, and the start condition in participant *Bank*. Conformance requirement *IR4* is satisfied.

Step 3 results in abstract interaction contributions *q1*, *q2*, and *q3* in abstract participants *Buyer*, *Seller*, and *Bank*, respectively. The correspondences between the attributes of the concrete and abstract interaction contributions are listed in Table 4-8.

Table 4-8 Correspondences between attributes of concrete and abstract interaction contributions

Concrete interaction structure	Abstract interaction
$v_B.\iota$	$q1.\iota_1$
$t_B.\iota$	$q1.\iota_2$
$m_B.\tau$	$q1.\tau$
$v_S.\iota$	$q2.\iota_1$
$c_S.\iota_1$	$q2.\iota_2$
$t_K.\iota$	$q3.\iota_1$
$c_K.\iota_1$	$q3.\iota_2$

In Step 4, we observe that information attributes $v_B.\iota$ and $v_S.\iota$ are involved in distribution constraint $[v_B.\iota = v_S.\iota]$ that is implicit in remote interaction *invoice*. Given the attribute correspondences in Table 4-8, this constraint can be replaced with $[q1.\iota_1 = q2.\iota_1]$. Information attribute $t_B.\iota$ and $t_K.\iota$ are involved in distribution constraints $[t_B.\iota = t_K.\iota]$ that is implicit in remote interaction *transfer*. This constraint can be replaced with $[q1.\iota_2 = q3.\iota_1]$. Information attribute $c_S.\iota_1$ and $c_K.\iota_1$ are involved in distribution constraints $[c_S.\iota_1 = c_K.\iota_1]$ that is implicit in remote interaction *check*. This constraint can be replaced with $[q2.\iota_2 = q3.\iota_2]$. Time attribute $m_B.\tau$ is not involved in any distribution constraint. No distribution constraint of the abstract interaction can be specified for this time attribute.

The application of the abstraction method results in an abstract interaction that has an equivalence correctness relation with abstract interaction *pay'*. Concrete interaction *Pay* conforms to abstract interaction *pay'*.

4.7.4 Intermediary introduction

In this pattern, an abstract interaction between abstract participants is replaced with a concrete interaction structure that involves a new concrete participant acting as an intermediary between the concrete participants that correspond to the abstract participants. Concrete participants that correspond to the abstract participants interact only with the intermediary. Concrete participants do not interact directly, but via the intermediary.

This pattern allows one to put most of the collaboration logic in an intermediary, to keep simple the concrete participants that correspond to abstract participants. For example, in an interaction for a reservation of a holiday trip between a traveller, a hotel, and an airline, a travel agent can be introduced as an intermediary to carry out the negotiation between them.

This pattern also allows one to include the behaviour of communication middleware that enables interactions between concrete participants across distances. For example, an interaction for an online insurance application between a customer and an insurance company will be implemented using an asynchronous request-response mechanism based on callback. The inclusion of the behaviour of this interaction mechanism allows designers to model necessary interaction contributions to accommodate that interaction mechanism.

Figure 4-40 illustrates this pattern. An abstract interaction *q* between abstract participants *B1* and *B2* is replaced with a concrete interaction structure consisting of interaction *q1*, *q2*, *q3*, and *q4*. Concrete participant *B3* is introduced as an intermediary between concrete participants *B1* and *B2*. Concrete interactions *q1* and *q4* are between concrete participants *B1* and *B3*. Concrete interaction *q2* and *q3* are between concrete participants *B2* and *B3*.

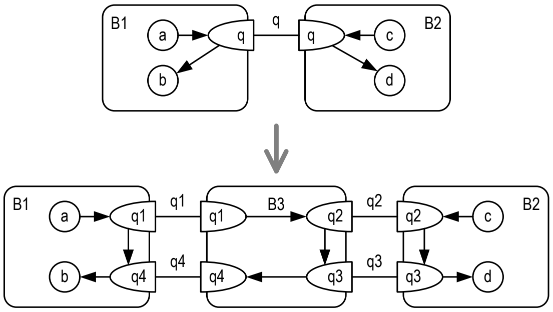


Figure 4-40 Intermediary introduction pattern

Example

Figure 4-41 depicts a concrete interaction structure *Confirm* obtained from an application of this pattern in the refinement of interaction *confirm* in Figure 4-39. This abstract interaction is implemented using a provider-confirmed message-passing mechanism. The seller first interacts with the middleware to send a confirmation message, i.e., "OK". The middleware then interacts with the buyer to pass that confirmation message. Finally, the middleware gives the seller an acknowledgment "ACK" indicating that the confirmation message has been passed successfully. Table 4-9 depicts the correspondence relation between abstract interaction *confirm* and concrete interaction *Confirm*.

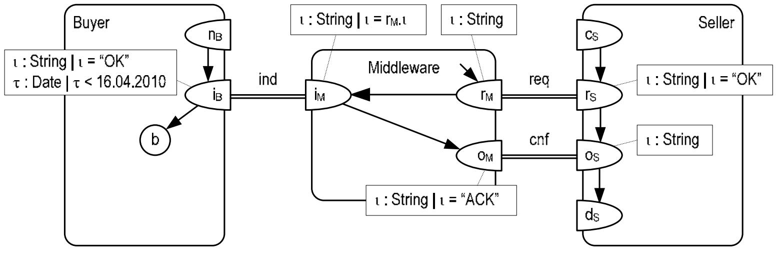


Figure 4-41 An example of intermediary introduction

Table 4-9
Correspondence relation between abstract interaction *confirm* and concrete interaction structure *Confirm*

	Abstract interaction	Concrete interaction structure
Participants	Buyer	Buyer
	Seller	Seller
Attributes	$m_B.\iota$	$i_B.\iota$
	$m_B.\tau$	$i_B.\tau$
	$m_S.\iota$	$r_S.\iota$
Occurrences	confirm	ind $\wedge$ cnf

The conformance assessment is described as follows. In Step 1, we determine that participants *Buyer* and *Seller* and attributes $i_B.\iota$, $i_B.\tau$, and $r_S.\iota$ are the design information that should be preserved.

In Step 2, we observe that each final interaction depends on the same context actions, i.e., n_B of *Buyer* and c_S of *Seller*. Conformance requirement *IR4* is satisfied.

Step 3 results in abstract interaction contributions *q1* and *q2* in abstract participants *Buyer* and *Seller*, respectively. The correspondences between the attributes of the concrete and abstract interaction contributions are listed in Table 4-10.

Table 4-10
Correspondences between attributes of concrete and abstract interaction contributions

Concrete interaction structure	Abstract interaction
$i_B.\iota$	$q1.\iota$
$i_B.\tau$	$q1.\tau$
$r_S.\iota$	$q2.\iota$

In Step 4, we observe that information attributes $i_B.\iota$ is involved in distribution constraint $[i_B.\iota = i_M.\iota]$ that is implicit in the remote interaction *ind*. Information attribute $r_S.\iota$ is involved in distribution constraint $[r_S.\iota = r_M.\iota]$ that is implicit in remote interaction *req*. Information attribute $i_M.\iota$ and $r_M.\iota$ are involved in contribution constraint $[i_M.\iota = r_M.\iota]$. Substituting $i_B.\iota$ and $r_S.\iota$ for $i_M.\iota$ and $r_M.\iota$ in this contribution constraint, respectively, results in $[i_B.\iota = r_S.\iota]$. Given the attribute correspondences in Table 4-10, this constraint can be replaced with $[q1.\iota = q2.\iota]$.

Time attribute $i_B.\tau$ is involved in distribution constraint $[i_B.\tau = i_M.\tau]$, but $i_M.\tau$. does not refer to or is not referred to by any other preserved time attribute. No distribution constraint of the abstract interaction can be specified for this time attribute.

The application of the abstraction method results in an abstract interaction that has an equivalence correctness relation with abstract interaction *confirm*. Concrete interaction *Confirm* conforms to abstract interaction *confirm*.

4.8 Related work

The MDA [90, 91] uses the idea of abstraction and refinement to allow “zooming” in and out of a model. Our work contributes to this area by providing a set of conformance requirements, an abstraction method, and a set of abstractions of interaction mechanisms.

Similar to [83, 114], our work considers an interaction as a first-class entity in a design process. Such an interaction can be used as a starting point for design refinement. Interaction refinement and its conformance assessment in [4] use the same assessment method as in Section 4.6.3. However, it does not define any conformance requirement or systematic abstraction method. Our work provides conformance requirements and a systematic abstraction method. Our work contributes to research towards interaction refinement, such as in [3, 27, 72], by providing interaction design transformations in the architectural domain.

In general, correct implementations can be obtained using two different refinement approaches: *correctness-by-construction* and *correctness-by-assessment*. In the first approach, as used in [43, 83], refinement is done by applying rules to construct correct implementations. This way of refinement however limits designers’ freedom because an implementation can only be obtained from an application of (a combination of) these rules. Our work supports the second approach, in which designers construct an implementation without necessarily following any refinement rule and assess the correctness of the implementation afterwards. If an implementation does not satisfy a set of correctness requirements, the implementation should be revisited and redeveloped. We believe that this refinement approach gives designers more freedom.

Our patterns of interaction refinement indicate possible ways of refinement without defining any refinement rule. These are not refinement rules as in [83]. Thus, a concrete interaction structure obtained from the application of our patterns may not conform to an abstract interaction. Conformance assessment should be performed on that concrete interaction structure.

An abstract interaction can be refined into a concrete interaction structure that complies with a structure defined as interaction patterns in [16, 54, 70]. Our work can be useful to check whether an interaction pattern used in an implementation results in a correct refinement.

[13, 126] present refinement of an interaction point in the entity domain. Interaction refinement in the behaviour domain is briefly discussed as its consequence. Our work can complement this work by providing interaction refinement in the behaviour domain.

4.9 Concluding remarks

In this chapter, we have presented behaviour transformations, i.e., refinement and abstraction, in ISDL. Interaction refinement was done indirectly in three steps. First, interacting behaviours are integrated into an abstract behaviour. Second, the abstract behaviour is refined into a concrete behaviour using action refinement and/or causality refinement. Finally, the concrete behaviour is decomposed using constraint-oriented behaviour structuring into interacting sub-behaviours. We have found out that this indirect interaction refinement cannot preserve the distribution of responsibility between participants. We have, therefore, defined direct interaction refinement that can maintain distribution of responsibility during the design process. A conformance assessment for interaction refinement is also defined.

In Section 3.1.2, three basic concepts for behaviour modelling were identified, i.e., *action*, *interaction*, and *causality relation*. However, ISDL supported only action refinement and causality refinement. Together with the interaction refinement presented in this chapter, refinement of all basic concepts is now supported. Concepts and methods, that have been defined for action and causality refinement, are reused in interaction refinement so as to give designers consistent ways of designing behaviours of distributed systems. Table 4-11 lists basic concepts in the behaviour domain and their support for behaviour refinement.

Table 4-11
Refinement of the basic concepts in the behaviour domain

Basic concept	**Behaviour refinement**
Action	Action refinement
Interaction	Interaction refinement
Causality relation	Causality refinement

Chapter 5

Abstract representations of interaction mechanisms

This chapter presents abstract representations of common interaction mechanisms that are suitable for modelling service compositions and distributed applications in general at higher abstraction levels. The abstract representations are obtained by applying the abstraction method defined in Chapter 4. This chapter is organised as follows: Section 5.1 motivates the need for suitable abstract representations of interaction mechanisms. Section 5.2 presents the approach we use to obtain abstract representations of interaction mechanisms. Section 5.3 shows how we apply the approach and our interaction abstraction method to obtain those abstract representations. Section 5.4 illustrates the use of the abstract representations in an example. Section 5.5 discusses related work. Finally, Section 5.6 presents some concluding remarks.

5.1 Motivation

The design of a service composition is a complex undertaking, especially if a designer is forced to deal immediately with the detailed behaviour of interaction mechanisms provided by communication middleware. It would be better if the designer could first focus on the essentials of the service composition using suitable abstract representations of interaction mechanisms.

In a service composition, service users and providers interact with each other. These service users and providers may be physically and geographically distributed. Communication middleware is therefore necessary to carry out the interactions between them. It provides generic interaction mechanisms, e.g., message passing and request-response interaction mechanisms, with which the interactions can be implemented.

Unfortunately, the detailed behaviour of interaction mechanisms contributes to the complexity of interactions in service compositions. When designing the interaction between service users and providers, a designer has to consider

- the characteristics of the interaction; and
- the behaviour of the interaction mechanism(s) that will be used to implement that interaction.

The interaction between a customer and a bank for an on-line loan application, for example, can be implemented either as synchronous or asynchronous request-response communication. The characteristic of this loan application interaction, i.e., the bank needs a couple of days to respond, leads the designers to choose the asynchronous request-response communication based on callback for implementing the interaction. The behaviour of this asynchronous communication has to be elaborated in the interaction design. This elaboration increases the complexity of the design. Such complexity makes it difficult to separate the behaviour of the interaction mechanism from the business or application logic of the interaction.

The use of related abstraction levels, as described in Chapter 1, allows a designer to first focus on the essentials of interactions between services, deferring the decision about possible alternative implementations to a later stage of the design process. At a high abstraction level, a designer defines only the intended result and requirements on the interaction and not the behaviour of the interaction mechanism that implements it. This approach needs a suitable abstract representation of the interaction mechanism.

Such an abstract representation should satisfy the following requirements [34].

- *Suitability*. An abstract representation of an interaction mechanism should be easy to use. In this thesis, an abstract representation is easy to use if it represents an interaction mechanism using a single interaction concept.
- *Platform independence.* An abstract representation of an interaction mechanism should not be specific to an implementation in a particular middleware platform. A platform-independent abstraction gives designers more implementation alternatives.
- *Correctness*. An abstract representation of an interaction mechanism should preserve the essential properties and represent correctly the behaviour of the interaction mechanism. This requirement means that the abstract representation can be refined back into the interaction mechanism.

5.2 Approach

In this section, we present our approach to obtain suitable representations of interaction mechanisms. We focus on the interaction mechanisms provided by CORBA [89] and Web Services [133].

5.2.1 Suitability and platform independence

In order to satisfy the requirements for suitability and platform independence, we follow the abstraction hierarchy as depicted in Figure 5-1.

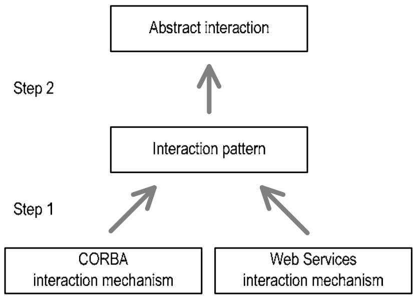

Abstract interaction
Step 2
Interaction pattern
Step 1
CORBA interaction mechanism
Web Services interaction mechanism

Figure 5-1
Abstraction hierarchy

In Step 1, we represent the behaviour of the CORBA and Web Services interaction mechanisms that are comparable to each other as an interaction structure that is independent of the details of the middleware platforms. We call that structure *an interaction pattern* because it models the similarity between the interaction mechanisms. This step results in a platform-independent interaction pattern.

Two interaction mechanisms are comparable to each other if the differences between them are not essential at a higher abstraction level. If the differences between them are essential, those interaction mechanisms are considered and modelled as two different interaction patterns. For example, the unconfirmed and provider-confirmed message-passing mechanisms have similarities, i.e., they pass a message from a sender to a receiver, but we consider that the differences between them are essential. Thus, we model them as two different interaction patterns.

In Step 2, if possible, we abstract an interaction pattern into an abstract interaction that is defined using the ISDL enhanced interaction concept. In this way, we can represent multiple interaction mechanisms by a single abstract concept and, therefore, satisfy the requirement of suitability as defined in Section 5.1. Since the interaction pattern is platform independent, its abstraction is also platform independent. The requirement of platform independence is also satisified.

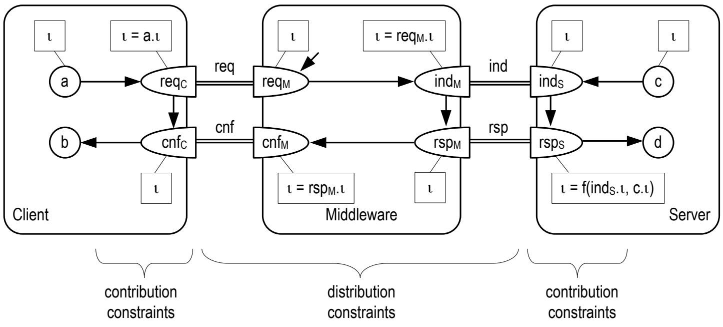
5.2.2 Correctness

In order to satisfy the requirement for correctness, we use the conformance requirements defined in Section 4.6.2:

- *IR1*: preservation of causality relations
- *IR2*: preservation of contribution constraints
- *IR3*: preservation of distribution constraints
- *IR4*: preservation of interaction synchronisation.

These conformance requirements, however, need a different interpretation because they are based on the properties of an interaction, not an interaction. The properties of an interaction cannot be found immediately in an interaction structure. The interpretation is explained in the following. We refer to Figure 5-2 for illustration. This figure depicts an interaction structure that represents the synchronous request-response mechanism between a client and server via communication middleware. For brevity, attribute types are omitted.

Figure 5-2
Contribution and distribution constraints in an interaction structure



The participants, that interact indirectly with each other via communication middleware, are called *remote participants*. In Figure 5-2, participants *Client* and *Server* are remote participants.

We aim to represent an interaction structure as an abstract interaction that abstracts from the detailed behaviour of the communication middleware. A remote participant is to be abstracted into an abstract participant. The interaction contribution structure in a remote participant is to be abstracted into an abstract interaction contribution in the corresponding abstract participant. Remote participants *Client* and *Server* in Figure 5-2 are to be abstracted into abstract participants *Client* and *Server*, respectively.

Attributes that represent essential information according to the purpose of the interaction mechanism should be preserved. Only such attributes in the remote participants are preserved. The interaction contributions of a remote participant that allow other actions of that remote participant to refer to the preserved attributes are the final interaction contributions of that remote participant. The interactions in which final interaction contributions are involved are the final interactions of the interaction structure.

In Figure 5-2, the purpose of a request-response mechanism is to establish a response message for a given request message. These messages are essential in the interaction mechanism. Therefore, information attributes that represent these messages should be preserved. The request message is represented by information attribute $req_C.\iota$ and $ind_S.\iota$; the response message is represented by information attribute $cnf_C.\iota$ and $rsp_S.\iota$. The final interaction contributions are cnf_C and rsp_S in remote participants *Client* and *Server*, respectively. The final interactions are hence *cnf* and *rsp*.

We model the causality context of an interaction mechanism as context actions *a*, *b*, *c*, and *d*. These context actions allow us to focus on an interaction structure without neglecting the dependencies between the interaction structure and its causality context. Context actions *a*, *b*, *c*, and *d* are to be abstracted into abstract context actions *a*, *b*, *c*, and *d*, respectively.

IR1: preservation of causality relations

The (indirect) causality relations between the final interaction contributions of a remote participant and the context actions should be preserved by the causality relations between the abstract interaction contribution of the corresponding abstract participant and the abstract actions that represent the context actions.

In Figure 5-2, the (indirect) causality relations between final interaction contribution cnf_C and context actions *a* and *b* in remote participant *Client* should be preserved by the causality relations between the abstract interaction contribution and abstract context actions *a* and *b* in abstract participant *Client*. The (indirect) causality relations between final interaction contribution rsp_S and context actions *c* and *d* in remote participant *Server* should be preserved by the causality relations between the abstract interaction contribution and abstract context actions *c* and *d* in abstract participant *Server*.

IR2: preservation of contribution constraints

The contribution constraints that specify the possible values of the preserved attributes of the interaction contribution structure in a remote participant should be preserved by the contribution constraints of the abstract interaction contribution of the corresponding abstract participant.

In Figure 5-2, the contribution constraints of preserved information attributes $req_C.\iota$ and $cnf_C.\iota$ in remote participants *Client* should be preserved by the contribution constraints of the abstract interaction contribution in abstract participants *Client*. The contribution constraints of preserved information attributes $\text{ind}_S.\iota$ and $\text{rsp}_S.\iota$ in remote participants *Server* should be preserved by the contribution constraints of the abstract interaction contribution in abstract participants *Server*.

IR3: preservation of distribution constraints

In an interaction structure, the relations between the attribute values in different remote participants are defined by

- the distribution constraints of the interactions between the communication middleware and remote participants; and
- the contribution constraints of the interaction contributions of the communication middleware.

The distribution and contribution constraints that define the relations between the preserved attributes in different remote participants should be preserved by the distribution constraints of the abstract interaction.

In Figure 5-2, the relation between the values of preserved information attributes $\text{req}_C.\iota$ and $\text{ind}_S.\iota$ in remote participant *Client* and *Server*, respectively, is defined by distribution constrains $[\text{req}_C.\iota = \text{req}_M.\iota]$ and $[\text{ind}_M.\iota = \text{ind}_S.\iota]$ of interactions *req* and *ind* (specified implicitly as local interactions) and contribution constraints $[\text{ind}_M.\iota = \text{req}_M.\iota]$ of interaction contribution ind_M of the middleware. These distribution and contribution constraints should be preserved by the distribution constraints of the abstract interaction.

IR4: preservation of interaction synchronisation

An interaction structure can be represented as an abstract interaction only if it provides time dependency as in an interaction. This dependency requires that every final interaction (indirectly) depends on the same context actions.

In Figure 5-2, final interaction contribution cnf_C of remote participant *Client* indirectly depends on context action *a*. Final interaction contribution rsp_S of remote participant *Server* indirectly depends on context action *c*. This interaction mechanism can be abstracted into an abstract interaction if every final interaction, i.e., *cnf* and *rsp*, depends on the set of context actions {a, c}.

5.2.3 Other interaction mechanism properties

The following properties of an interaction mechanism are not modelled (explicitly).

Time and location attributes

In an interaction structure, an interaction between a remote participant and the middleware establishes the same set of information values that are available from the same time moment and at the same location for the remote participant and middleware. We hence model the interaction as a local interaction (see Section 3.7.1).

The specifications of the CORBA and Web Services interactions mechanisms do not include constraints regarding time, e.g., delay or throughput, we hence do not include constraints on time attribute. Such constraints can be useful for modelling the quality of service (QoS) that is required from communication middleware. We leave this for future work.

We assume that all interactions between a remote partipant and communication middleware occur at the same location or address. For brevity, we do not include location attributes and constraints in the interactions.

Exceptions

An interaction mechanism may return an exception message when it cannot complete successfully. For example, in the CORBA synchronous request-response mechanism, an exception message can be returned to the client either by the server or middleware. An exception message returned by the server indicates that the server cannot process the request message sent by the client. An exception message returned by the middleware indicates that a problem occurs in the communication.

At a higher abstraction level, a designer may only be interested in whether an interaction occurs or does not occur, without considering any exception messages that might return. Moreover, the behaviours related to exception messages are middleware-specific. For example, in the Web Services synchronous request-response mechanism, the middleware cannot return an exception. To satisfy the requirement for platform independence, we exclude the behaviours that are related to exception messages. This exclusion can be done only if the exception messages have no function at an abstract level, i.e., they carry no essential information. We leave the inclusion of exception behaviour for future work.

5.3 Abstractions of interaction mechanisms

In this section, we present the applications of the interaction abstraction method defined in Chapter 4 to obtain abstract representations of common interaction mechanisms.

An interaction mechanism distinguishes different roles for its participants. A role determines the behaviour that a participant should perform to interact with other participant(s) using the interaction mechanism.

In the following sub-sections, a remote participant is named with the role it plays in an interaction mechanism. For example, a remote participant that plays the role of a client is named *Client*. It should be noted that the role is not attached to the participant, but to the contribution of the participant in the interaction mechanism. Therefore, in an interaction structure in which remote participants interact with each other using more than one interaction mechanisms, a remote participant can play the role of a client in one interaction mechanism and the role of a server in another interaction mechanism.

5.3.1 Unconfirmed message-passing

The purpose of this interaction mechanism is to pass a message from a sender to a receiver. The sender sends a message to the receiver and then continues its execution. In CORBA, this interaction mechanism is implemented using 'oneway' request-response communication. In Web Services, this interaction mechanism is implemented using 'one-way' operation.

The interaction pattern in Figure 5-3(i) models this interaction mechanism. Interaction *req* passes a message from the sender to the middleware. The sender determines the contents of this message [$req_S.\iota = f_S(a.\iota)$]. Interaction *ind* passes that message from the middleware to the receiver [$ind_M.\iota = req_M.\iota$].

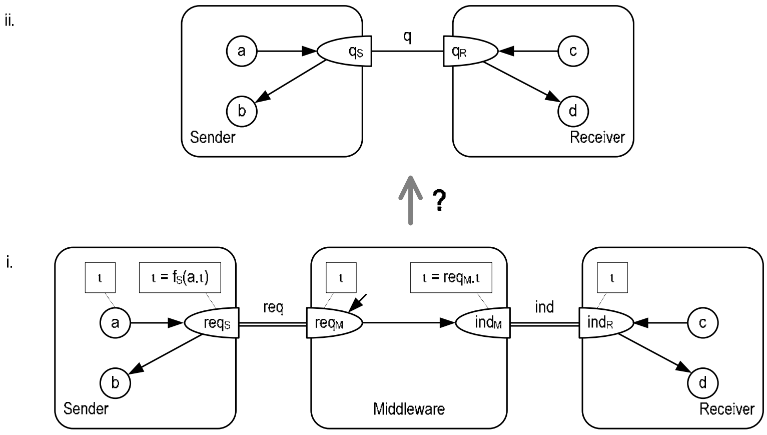


Figure 5-3
Unconfirmed message-passing and its intended abstract representation

We want to abstract the interaction pattern in Figure 5-3(i) into an abstract interaction in Figure 5-3(ii). The interaction abstraction method is applied as follows.

Step 1:
The design information that should be preserved is:

- remote participants *Sender* and *Receiver*; and
- information attributes $req_S.\iota$ and $ind_R.\iota$ as they represent the message that is passed from remote participants *Sender* to *Receiver*.

The relation between the preserved information attributes can be expressed as a single constraint [$req_S.\iota = ind_R.\iota$]. The calculation to derive this constraint is shown in Figure 5-4.

Figure 5-4
Calculation to derive the relation between the preserved information attributes in the unconfirmed message passing

$req_S.\iota = req_M.\iota$	; local interaction *req*
$ind_M.\iota = req_M.\iota$	; interaction contribution ind_M
$ind_M.\iota = ind_R\iota$	; local interaction *ind*
$req_S.\iota = ind_R.\iota$	

The final interaction contributions in remote participants *Sender* and *Receiver* are req_S and ind_R, respectively. The final interactions of this interaction pattern are hence interactions *req* and *ind*. Final interaction contributions req_S and ind_R depend on context actions *a* and *c*, respectively.

Step 2:
Final interaction *req* depends only on context action *a*. Final interaction *ind* depends on context action *a* via interaction *req* and on context action *c*. This means that final interactions *req* and *ind* do not depend on the same context actions. Conformance requirement *IR4* is therefore not satisfied. This interaction mechanism cannot be abstracted into a single abstract interaction.

To facilite interaction design, this interaction mechanism can be expressed using the shorthand notation for message-passing communication (see Section 3.7.3) because they have the same behaviour. This shorthand notation does not abstract from any design information of the message-passing interaction mechanism.

5.3.2 Provider-confirmed message-passing

The purpose of this interaction mechanism is to pass a message from a sender to a receiver. The sender sends a message to the receiver and then waits for a confirmation from the middleware that indicates that the message has been delivered to the receiver. The sender continues its execution only after it receives that confirmation.

The interaction pattern in Figure 5-5 models this interaction mechanism. Interaction *req* passes a message from the sender to the middleware [$req_S.\iota = f_S(a.\iota)$]. Interaction *ind* passes that message from the middleware to the receiver [$ind_M.\iota = req_M.\iota$]. Interaction *cnf* confirms to the sender that the message has been delivered to the receiver [$cnf_M.\iota = f_M(ind_M.\iota)$].

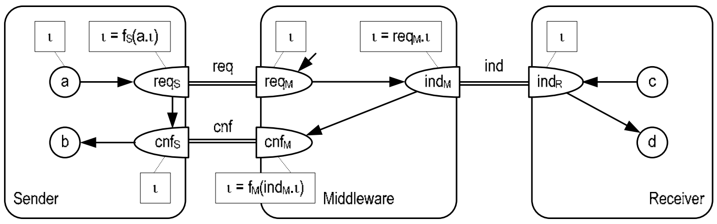


Figure 5-5
Provider-confirmed message-passing

The interaction abstraction method is applied as follows.

Step 1:

The design information that should be preserved is:

- remote participants *Sender* and *Receiver*; and
- information attributes $\text{req}_S.\iota$ and $\text{ind}_R.\iota$ as they represent the message that is passed from remote participants *Sender* to *Receiver*.

As in Section 5.3.1, the relation between the preserved information attributes can be expressed as a single constraint $[\text{req}_S.\iota = \text{ind}_R.\iota]$.

The final interaction contributions in remote participants *Sender* and *Receiver* are cnf_S and ind_R, respectively. The final interactions of this interaction pattern are hence interactions *cnf* and *ind*. Final interaction contribution cnf_S indirectly depends on context action *a*. Final interaction contribution ind_R depends on context action *c*.

Step 2:

Final interaction *cnf* depends on context action *a* via interaction *req* and on context action *c* via interaction *ind*. Final interaction *ind* depends on context action *a* via interaction *req* and on context action *c*. Every final interaction depends on the same context actions. Conformance requirement *IR4* is satisfied.

Steps 3 and 4:

These steps result in abstract interaction *q* between abstract participants *Sender* and *Receiver*, as depicted in Figure 5-6. Abstract interaction contributions q_S and q_R represent the interaction contribution structures in remote participant *Sender* and *Receiver*, respectively. Table 5-1 lists the correspondences between the design information of the interaction pattern and abstract interaction. Conformance requirement *IR1*, *IR2*, and *IR3* are satisfied. We successfully obtain an abstract representation of the provider-confirmed message-passing mechanism.

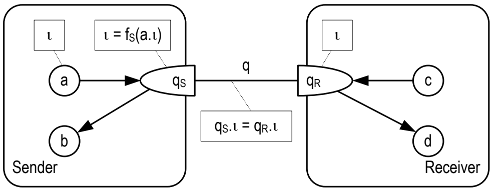


Figure 5-6
Abstract representation of the provider-confirmed message-passing

Table 5-1
Correspondences between the design information of the interaction pattern and abstract interaction

	Interaction pattern	Abstract interaction
Information attributes	$req_S.\iota$	$q_S.\iota$
	$ind_R.\iota$	$q_R.\iota$
Constraints	$req_S.\iota = f_S(a.\iota)$	$q_S.\iota = f_S(a.\iota)$
	$req_S.\iota = ind_R.\iota$	$q_S.\iota = q_R.\iota$

This abstract interaction can be represented as a remote interaction (see Section 3.7.2) because abstract participants *Sender* and *Receiver* see the same set of information attribute values that are available from different time moments and at different locations. Figure 5-7 depicts this interaction as a remote interaction.

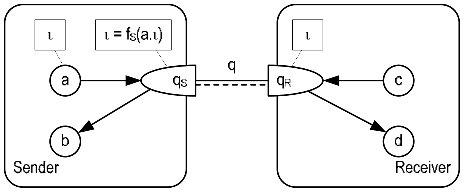


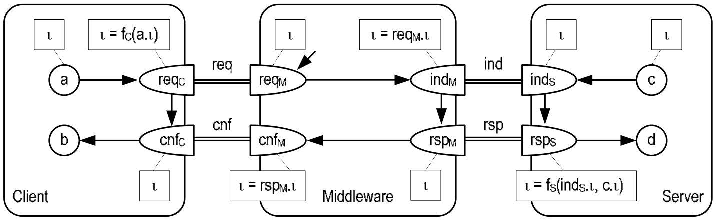
Figure 5-7
Provider-confirmed message-passing as a remote interaction

5.3.3 Synchronous request-response

The purpose of this interaction mechanism is to establish a response message for a given request message. A client sends a request message to a server and the server sends a response message back to the client. After sending a request message, the client waits for a response message before it continues its execution.

The interaction pattern in Figure 5-8 models this interaction mechanism. Interaction *req* passes a request message from the client to the middleware [$req_C.\iota = f_C(a.\iota)$]. Interaction *ind* passes that request message from the middleware to the server [$ind_M.\iota = req_M.\iota$]. Interaction *rsp* passes a response message from the server to the middleware. The server creates the response message based on the contents of the request message and the server's state [$rsp_S.\iota = f_S(ind_S.\iota, c.\iota)$]. Interaction *cnf* passes that response message from the middleware to the client [$cnf_M.\iota = rsp_M.\iota$].

Figure 5-8
Synchronous request-response



The interaction abstraction method is applied as follows.

Step 1:

The design information that should be preserved is:

- remote participants *Client* and *Server*;
- information attributes $\text{req}_C.\iota$ and $\text{ind}_S.\iota$ as they represent the request message; and
- information attributes $\text{cnf}_C.\iota$ and $\text{rsp}_S.\iota$ as they represent the response message.

The relation between the preserved information attributes that represent the request message can be expressed as a single constraint [$\text{req}_C.\iota = \text{ind}_S.\iota$]. The relation between the preserved information attributes that represent the response message can be expressed as a single constraint [$\text{cnf}_C.\iota = \text{rsp}_S.\iota$]. The calculation to derive these constraints is shown in Figure 5-9.

Figure 5-9
Calculation to derive the relation between the preserved information attributes in the synchronous request-response

$\text{req}_C.\iota = \text{req}_M.\iota$	; local interaction *req*
$\text{ind}_M.\iota = \text{req}_M.\iota$	; interaction contribution ind_M
$\text{ind}_M.\iota = \text{ind}_S.\iota$	; local interaction *ind*
$\text{req}_C.\iota = \text{ind}_S.\iota$	; the relation for request message
$\text{rsp}_S.\iota = \text{rsp}_M.\iota$	; local interaction *rsp*
$\text{cnf}_M.\iota = \text{rsp}_M.\iota$	; interaction contribution cnf_M
$\text{cnf}_M.\iota = \text{cnf}_C.\iota$	; local interaction *cnf*
$\text{cnf}_C.\iota = \text{rsp}_S.\iota$	; the relation for response message

The final interaction contributions in remote participants *Client* and *Server* are cnf_C and rsp_S, respectively. The final interactions of this interaction pattern are hence interactions *cnf* and *rsp*. Final interaction contributions cnf_C and ind_S indirectly depends on context actions *a* and *c*, respectively.

Step 2:

Final interaction *cnf* depends on context action *a* via interaction *req* and on context action *c* via interactions *rsp* and *ind*. Final interaction *rsp* depends on context action *a* via interactions *ind* and *req*; and on context action *c* via interaction *ind*. Every final interaction depends on the same context actions. Conformance requirement *IR4* is satisfied.

Steps 3 and 4:

These steps result in abstract interaction q between abstract participant *Client* and *Server*, as depicted in Figure 5-10. Abstract interaction contributions q_C and q_S represent the interaction contribution structures in remote participant *Client* and *Server*, respectively. Table 5-2 lists the correspondences between the design information of the interaction pattern and abstract interaction. Conformance requirement *IR1*, *IR2*, and *IR3* are satisfied. We successfully obtain an abstract representation of the synchronous request-response mechanism.

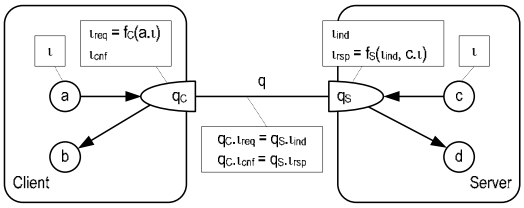
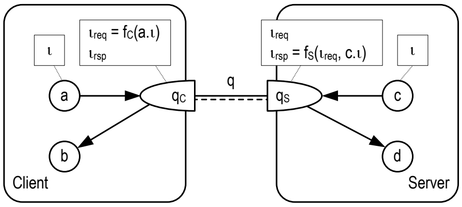


Figure 5-10
Abstract representation of the synchronous request-response

Table 5-2
Correspondences between the design information of the interaction pattern and abstract interaction

	Interaction pattern	Abstract interaction
Information attributes	$req_C.\iota$	$q_C.\iota_{req}$
	$cnf_C.\iota$	$q_C.\iota_{cnf}$
	$ind_S.\iota$	$q_S.\iota_{ind}$
	$rsp_S.\iota$	$q_S.\iota_{rsp}$
Constraints	$req_C.\iota = f_C(a.\iota)$	$q_C.\iota_{req} = f_C(a.\iota)$
	$rsp_S.\iota = f_S(ind_S.\iota, c.\iota)$	$q_S.\iota_{rsp} = f_S(q_S.\iota_{ind}, c.\iota)$
	$req_C.\iota = ind_S.\iota$	$q_C.\iota_{req} = q_S.\iota_{ind}$
	$cnf_C.\iota = rsp_S.\iota$	$q_C.\iota_{cnf} = q_S.\iota_{rsp}$

This interaction can be represented as a remote interaction (see Section 3.7.2) because abstract participants *Sender* and *Receiver* see the same set of information attribute values that are available from different time moments and at different locations. Figure 5-11 depicts this interaction as a remote interaction. Information attributes $q_S.\iota_{ind}$ and $q_C.\iota_{cnf}$ are renamed as $q_S.\iota_{req}$ and $q_C.\iota_{rsp}$, respectively.



ι
$\iota_{req} = f_C(a.\iota)$
ι_{rsp}
ι_{req}
$\iota_{rsp} = f_S(\iota_{req}, c.\iota)$
ι
q
a
q_C
q_S
c
b
d
Client
Server

Figure 5-11
Synchronous request-response as a remote interaction

5.3.4 Asynchronous request-response: callback

The purpose of this interaction mechanism is to establish a response message for a given request message. A client sends a request message to a server and the server sends a response message back to the client. After sending a request message (and receiving a confirmation from the middleware that indicates that the request message has been delivered to the server), the client may continue its execution, but eventually it has to wait for a response message. The Web Services callback mechanism is defined in the WS-Callback specification [18].

The interaction pattern in Figure 5-12 models this interaction mechanism. It is composed of two provider-confirmed message-passing mechanisms: one is for passing a request message from the client to the server (interactions req_1, ind_1, and cnf_1) and the other in the opposite direction is for passing a response message from the server to the client (interactions req_2, ind_2, and cnf_2).

Interaction req_1 passes a request message from the client to the middleware [$req_{1C}.\iota = f_C(a.\iota)$]. Interaction ind_1 passes that request message from the middleware to the server [$ind_{1M}.\iota = req_{1M}.\iota$]. Interactions cnf_1 confirms the client that the request message has been delivered to the server.

Interaction req_2 passes a response message from the server to the middleware [$req_{2S}.\iota = f_S(ind_{1S}.\iota, c.\iota)$]. Interaction ind_2 passes that response message from the middleware to the client [$ind_{2M}.\iota = req_{2M}.\iota$]. Interactions cnf_2 confirms the server that the response message has been delivered to the client.

The client may do other actions while waiting for the callback, i.e., between interaction contributions cnf_{1C} and ind_{2C}. However, inserting actions that are irrelevant to the interaction mechanism makes the design difficult to analyse because different concerns are mixed together. Such actions would be better done independently, possibly concurrently with the interaction mechanism.

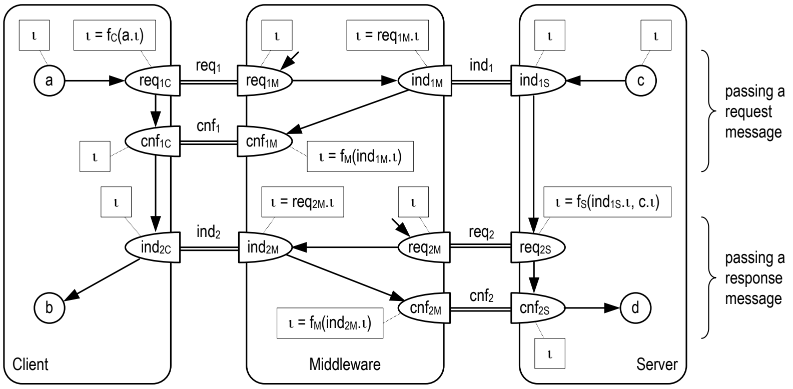


Figure 5-12
Asynchronous request-response based on callback

Instead of abstracting this interaction pattern into a single abstract interaction in a single abstraction activity, we abstract it in two sequential activities. First, we abstract each provider-confirmed message-passing mechanism into an abstract interaction as in Section 5.3.2. Second, we abstract the interaction structure that results from the first activity into an abstract interaction.

The first activity results in the interaction structure that is depicted in Figure 5-13. Interactions *req* and *rsp* represent the mechanisms for passing a request and response messages, respectively. Table 5-3 lists the correspondences between the design information of the interaction pattern and the abstract interactions.

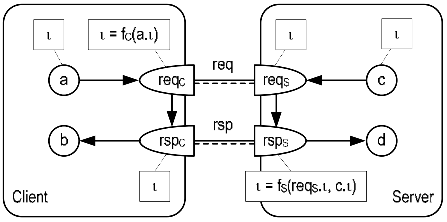


Figure 5-13
Callback using remote interactions

Table 5-3
Correspondences between the design information of the interaction pattern and abstract interactions

	Interaction pattern	**Abstract interactions**
Information attributes	$req_{1C}.ι$	$req_{C}.ι$
	$ind_{1S}.ι$	$req_{S}.ι$
	$req_{2S}.ι$	$rsp_{S}.ι$
	$ind_{2C}.ι$	$rsp_{C}.ι$
Constraints	$req_{1C}.ι = f_C(a.ι)$	$req_C.ι = f_C(a.ι)$
	$req_{1C}.ι = ind_{1S}.ι$	$req_C.ι = req_S.ι$
	$req_{2S}.ι = f_S(ind_{1S}.ι, c.ι)$	$rsp_S.ι = f_S(req_S.ι, c.ι)$
	$req_{2S}.ι = ind_{2C}.ι$	$rsp_S.ι = rsp_C.ι$

The second activity abstracts the interaction structure in Figure 5-13 into an abstract interaction *q*.

Step 1:

The design information that should be preserved is:

- participants *Client* and *Server*;
- information attributes $req_C.\iota$ and $req_S.\iota$ as they represent the request message; and
- information attributes $rsp_C.\iota$ and $rsp_S.\iota$ as they represent the response message.

The relation between the preserved information attributes that represent the request message is distribution constraint $[req_C.\iota = req_S.\iota]$. The relation between the preserved information attributes that represent the response message is distribution constraint $[rsp_C.\iota = rsp_S.\iota]$. These constraints are implicit in remote interactions *req* and *rsp*, respectively.

The final interaction contributions in participants *Client* and *Server* are rsp_C and rsp_S, respectively. The final interaction of this interaction pattern is hence interaction *rsp*. Final interaction contributions rsp_C and rsp_S indirectly depends on context actions *a* and *c*, respectively.

Step 2:

Final interaction *rsp* depends on context actions *a* and *c* via interaction *req* Conformance requirement *IR4* is satisfied.

Steps 3 and 4:

These steps result in abstract interaction *q* between abstract participant *Client* and *Server*, as depicted in Figure 5-14. Abstract interaction contributions q_C and q_S represent the interaction contribution structures in participant *Client* and *Server*, respectively. Table 5-4 lists the correspondences between the design information of the interaction structure and abstract interaction. Conformance requirement *IR1*, *IR2*, and *IR3* are satisfied. We successfully obtain an abstract representation of the asynchronous request-response mechanism based on callback.

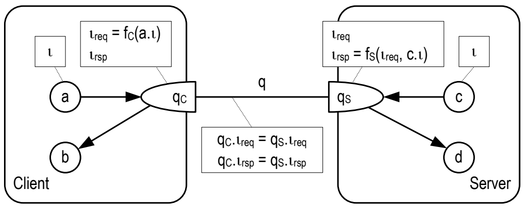


Figure 5-14
Abstract representation of the asynchronous request-response based on callback

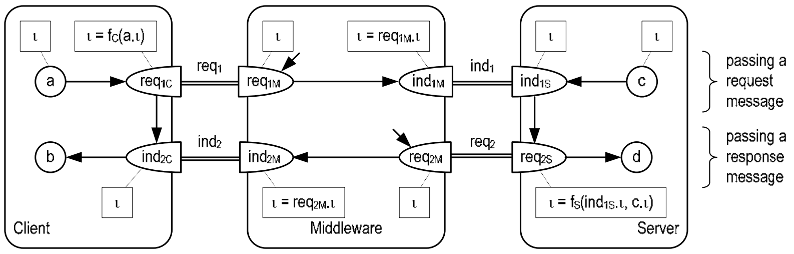
Table 5-4
Correspondences between the design information of the interaction structure and abstract interaction

	Interaction structure	**Abstract interaction**
Information attributes	$req_C.\iota$	$q_C.\iota_{req}$
	$req_S.\iota$	$q_S.\iota_{req}$
	$rsp_S.\iota$	$q_S.\iota_{rsp}$
	$rsp_C.\iota$	$q_C.\iota_{rsp}$
Constraints	$req_C.\iota = f_C(a.\iota)$	$q_C.\iota_{req} = f_C(a.\iota)$
	$req_C.\iota = req_S.\iota$	$q_C.\iota_{req} = q_S.\iota_{req}$
	$rsp_S.\iota = f_S(req_S.\iota, c.\iota)$	$q_S.\iota_{rsp} = f_S(q_S.\iota_{req}, c.\iota)$
	$rsp_S.\iota = rsp_C.\iota$	$q_S.\iota_{rsp} = rsp_C.\iota$

This abstract interaction can be represented as a remote interaction (see Section 3.7.2), as depicted in Figure 5-11.

Alternatively, this interaction mechanism can be modelled as the interaction pattern in Figure 5-15. It is composed of two unconfirmed message-passing mechanisms: one is for passing a request message from the client to the server (interactions *req_1* and *ind_1*) and the other in the opposite direction is for passing a response message from the server to the client (interactions *req_2* and *ind_2*). This interaction pattern can be abstracted into an abstract interaction in a similar way to the abstraction in Section 5.3.3. It results in an abstract interaction that is the same as the abstract interaction in Figure 5-14.

Figure 5-15
Callback using unconfirmed message-passings



5.3.5 Asynchronous request-response: remote-polling

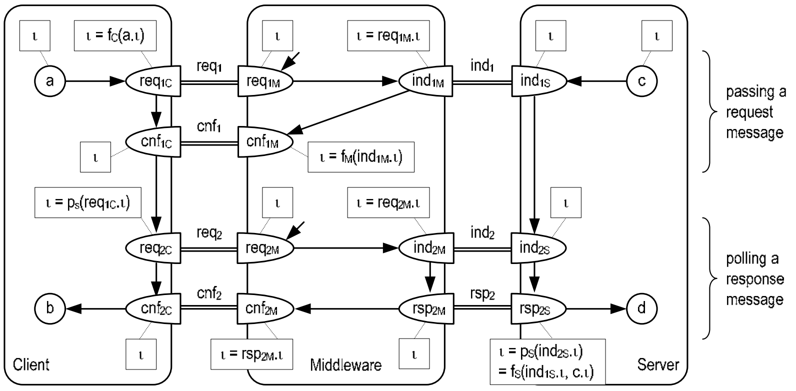
The purpose of this interaction mechanism is to establish a response message for a given request message. A client sends a request message to a server and then polls the server for a response message. After receiving a request message, the server waits for the client to poll a response message. The Web Services remote-polling mechanism is defined in the WS-Polling specification [140].

The interaction pattern in Figure 5-16 models this interaction mechanism. It is composed of a provider-confirmed message-passing mechanism for passing a request message from the client to the server (interactions *req_1*, *ind_1*, and *cnf_1*) and a synchronous request-response mechanism that is initiated by the client for polling a response message (interactions *req_2*, *ind_2*, *rsp_2*, and *cnf_2*).

Interaction *req_1* passes a request message from the client to the middleware [$req_{1C}.\iota = f_C(a.\iota)$]. Interaction *$ind_1$* passes that request message from the middleware to the server [$ind_{1M}.\iota = req_{1M}.\iota$]. Interactions *$cnf_1$* confirms the client that the request message has been delivered to the server.

Interaction *req_2* passes a polling message from the client to the middleware. The client creates this polling message based on the request message sent earlier [$req_{2C}.\iota = p_S(req_{1C}.\iota)$]. Interaction *$ind_2$* passes this polling message from the middleware to the server [$ind_{1M}.\iota = req_{1M}.\iota$]. It is used to poll a response message [$rsp_{2S}.\iota = p_S(ind_{2S}.\iota) = f_S(ind_{1S}.\iota, c.\iota)$]. Interaction *$cnf_2$* passes that response message from the middleware to the client [$cnf_{2M}.\iota = rsp_{2M}.\iota$].

Figure 5-16
Asynchronous request-response based on remote polling



The client may do other actions before polling the response message, i.e., between interaction contributions *cnf_{1C}* and *req_{2C}*. As motivated in Section 5.3.4, such actions would be better done independently, possibly concurrently with the interaction mechanism.

We abstract the interaction pattern in two sequential activities as in Section 5.3.4. First, we abstract each underlying interaction mechanism into an abstract interaction. Second, we abstract the interaction structure that results from the first activity into an abstract interaction.

The first activity results in the interaction structure that is depicted in Figure 5-13. Interaction *req* represents the mechanism for passing a request message. Interaction *rsp* represents the mechanism for polling a response message. It abstracts from the polling message because this message is not essential according to the purpose of the interaction mechanism. The second activity results in the same model as depicted in Figure 5-14.

Alternatively, an unconfirmed message-passing mechanism can be used for passing a request message. In this case, the interaction pattern should be done directly, not in two sequential activities. This is because an unconfirmed message-passing mechanism cannot be abstracted into an abstract interaction. The abstraction results in an abstract interaction as depicted in Figure 5-14 .

5.3.6 Asynchronous request-response: local-polling

The purpose of this interaction mechanism is to establish a response message for a given request message. A client sends a request message to a server and then polls the middleware for a response message. After receiving a request message, the server gives a response message to the middleware. The middleware waits for the client to poll that response message. Web Services do not have any specification for a local-polling mechanism.

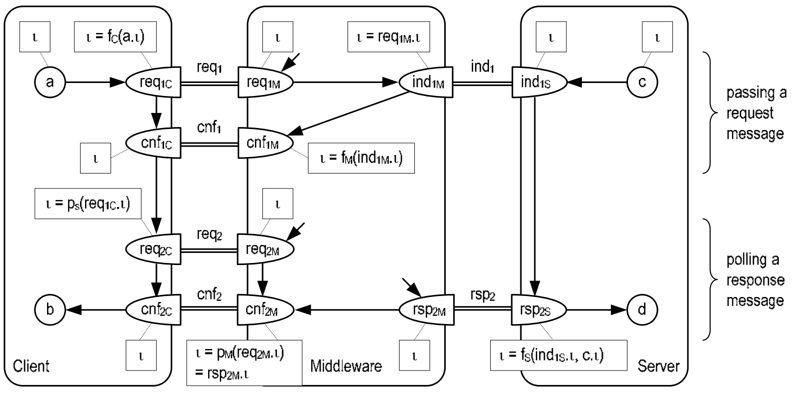
The interaction pattern in Figure 5-17 models this interaction mechanism. The client sends a request message to the server using a provider-confirmed message-passing mechanism (interactions req_1, ind_1, and cnf_1). Interaction req_1 passes a request message from the client to the middleware [$req_{1C}.\iota = f_C(a.\iota)$]. Interaction ind_1 passes that request message from the middleware to the server [$ind_{1M}.\iota = req_{1M}.\iota$]. Interactions cnf_1 confirms the client that the request message has been delivered to the server. When a response message is available, the server passes it to the middleware using interaction rsp_2 [$rsp_{2S}.\iota = f_S(ind_{1S}.\iota, c.\iota)$].

Interaction req_2 passes a polling message from the client to the middleware [$req_{2C}.\iota = p_S(req_{1C}.\iota)$]. Interaction cnf_2 passes a response message that is associated to that polling message from the middleware to the client [$cnf_{2M}.\iota = p_M(req_{2M}.\iota) = rsp_{2M}.\iota$].

The abstract representation of the provider-confirmed message-passing mechanism that is obtained in Section 5.3.2 cannot be reused here because the other part of this local-polling mechanism, i.e., for polling a response message, cannot be abstracted separately into an abstract interaction. This interaction pattern has to be abstracted into an interaction mechanism in a single abstraction activity.

The client may do other actions before polling the response message, i.e., between interaction contributions cnf_{1C} and req_{2C}. As motivated in Section 5.3.4, such actions would be better done independently, possibly concurrently with the interaction mechanism.

Figure 5-17
Asynchronous request-response based on local-polling



The interaction abstraction method is applied as follows.

Step 1:
The design information that should be preserved is:
- remote participants *Client* and *Server*;
- information attributes $\text{req}_{1C}.\iota$ and $\text{ind}_{1S}.\iota$ as they represent the request message; and
- information attributes $\text{cnf}_{2C}.\iota$ and $\text{rsp}_{2S}.\iota$ as they represent the response message.

The relation between the preserved information attributes that represent the request message can be expressed as a single constraint $[\text{req}_{1C}.\iota = \text{ind}_{1S}.\iota]$. The relation between the preserved information attributes that represent the response message can be expressed as a single constraint $[\text{cnf}_{2C}.\iota = \text{rsp}_{2S}.\iota]$. The calculation to derive these constraints is shown in Figure 5-18.

Figure 5-18
Calculation to derive the relation between the preserved information attributes in the asynchronous request-response based on local polling

$\text{req}_{1C}.\iota = \text{req}_{1M}.\iota$	; local interaction *req_1*
$\text{ind}_{1M}.\iota = \text{req}_{1M}.\iota$	; interaction contribution *ind_{1M}*
$\text{ind}_{1M}.\iota = \text{ind}_{1S}.\iota$	; local interaction *ind_1*
$\text{req}_{1C}.\iota = \text{ind}_{1S}.\iota$	; the relation for request message
$\text{rsp}_{2S}.\iota = \text{rsp}_{2M}.\iota$	; local interaction *rsp_2*
$\text{cnf}_{2M}.\iota = \text{rsp}_{2M}.\iota$	; interaction contribution *cnf_{2M}*
$\text{cnf}_{2M}.\iota = \text{cnf}_{2C}.\iota$	; local interaction *cnf_2*
$\text{cnf}_{2C}.\iota = \text{rsp}_{2S}.\iota$	; the relation for response message

The final interaction contributions in remote participants *Client* and *Server* are *cnf_{2C}* and *rsp_{2S}*, respectively. The final interactions of this interaction pattern are hence interactions *cnf_2* and *rsp_2*. Final interaction contributions *cnf_{2C}* and *ind_{2S}* indirectly depends on context actions *a* and *c*, respectively.

Step 2:

Final interaction *cnf_2* depends on context action *a* via interactions *req_2*, *cnf_1*, and *req_1*; and on context action *c* via interactions *rsp_2* and *ind_1*. Final interaction *rsp_2* depends on context action *a* via interaction *ind_1* and *req_1*; and on context action *c* via interaction *ind_1*. Every final interaction depends on the same context actions. Conformance requirement *IR4* is satisfied.

Steps 3 and 4:

These steps result in abstract interaction *q* between abstract participant *Client* and *Server*, as depicted in Figure 5-19. Abstract interaction contributions q_C and q_S represent the interaction contribution structures in remote participants *Client* and *Server*, respectively. Table 5-5 lists the correspondences between the design information of the interaction pattern and abstract interaction. Conformance requirement *IR1*, *IR2*, and *IR3* are satisfied. We successfully obtain an abstract representation of the asynchronous request-response mechanism based on local polling.

Figure 5-19
Abstract representation of the asynchronous request-response based on local-polling

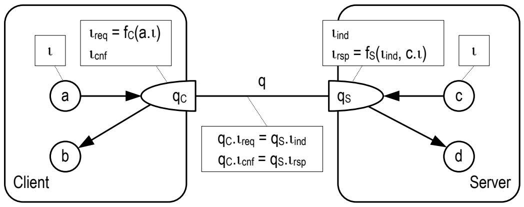

ι
$\iota_{req} = f_C(a.\iota)$
ι_{cnf}
ι_{ind}
$\iota_{rsp} = f_S(\iota_{ind}, c.\iota)$
ι
a
q_C
q
q_S
c
b
d
$q_C.\iota_{req} = q_S.\iota_{ind}$
$q_C.\iota_{cnf} = q_S.\iota_{rsp}$
Client
Server

Table 5-5
Correspondences between the design information of the interaction pattern and abstract interaction

	Interaction pattern	**Abstract interaction**
Information attributes	$req_{1C}.\iota$	$q_C.\iota_{req}$
	$ind_{1S}.\iota$	$q_S.\iota_{ind}$
	$rsp_{2S}.\iota$	$q_S.\iota_{rsp}$
	$cnf_{2C}.\iota$	$q_C.\iota_{cnf}$
Constraints	$req_{1C}.\iota = f_C(a.\iota)$	$q_C.\iota_{req} = f_C(a.\iota)$
	$req_{1C}.\iota = ind_{1S}.\iota$	$q_C.\iota_{req} = q_S.\iota_{ind}$
	$rsp_{2S}.\iota = f_S(ind_{1S}.\iota, c.\iota)$	$q_S.\iota_{rsp} = f_S(q_S.\iota_{ind}, c.\iota)$
	$cnf_{2C}.\iota = rsp_{2S}.\iota$	$q_C.\iota_{cnf} = q_S.\iota_{rsp}$

This abstract interaction can be represented as a remote interaction, as depicted in Figure 5-11. Information attributes $q_S.\iota_{ind}$ and $q_C.\iota_{cnf}$ are renamed as $q_S.\iota_{req}$ and $q_C.\iota_{rsp}$, respectively.

5.3.7 Multicast message-passing

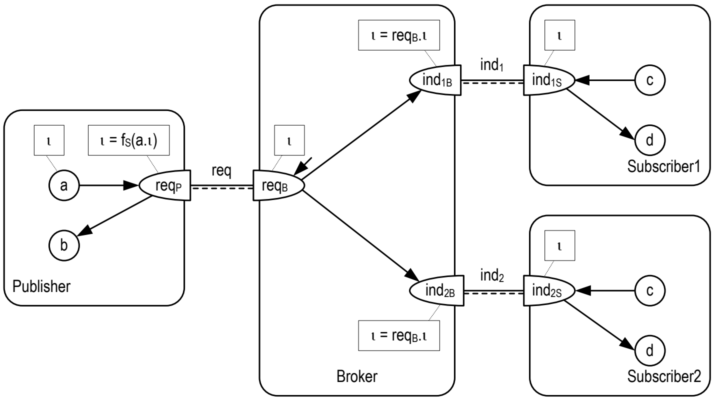
The purpose of this interaction mechanism is to pass copies of a message from a sender to multiple receivers. In CORBA and Web Services, this interaction mechanism is implemented using a publish/subscribe mechanism with a message broker playing the role of an intermediary. A publisher passes a message to the message broker and then the message broker passes copies of that message to multiple subscribers. A subscriber receives a copy of that message. The Web Services publish/subscribe mechanism is defined in the WS-Notification specification [84, 85].

A message is passed from a publisher to the message broker or from the message broker to a subscriber using either a 'push' or 'pull' strategy. In a 'push' strategy, a publisher pushes a message to the message broker. In a 'pull' strategy, the message broker pulls a message from a publisher. Similarly, the message broker can push a copy of a message to a subscriber; or a subscriber can pull a copy of a message from the message broker.

The interaction pattern in Figure 5-20 models this interaction mechanism. A publisher sends copies of a message to two subscribers. Interaction *req* passes a message from the publisher to the message broker. Interaction *ind*$_1$ and *ind*$_2$ passes copies of that message from the message broker to the subscribers.

This interaction pattern is already modelled at a higher abstraction level. It abstracts from the actual interaction mechanisms used in remote interactions *req*, *ind*$_1$, and *ind*$_2$. Each interaction can be implemented to support either a 'push' or 'pull' strategy. For example, if interaction *req* has to support a 'push' strategy, it can be implemented using the provider-confirmed message-passing mechanism, in which the publisher acts as a sender and the message broker acts as a receiver. If it has to support a 'pull' strategy, it can be implemented using the synchronous request-response mechanism, in which the message broker acts as a client and the publisher acts as a server.

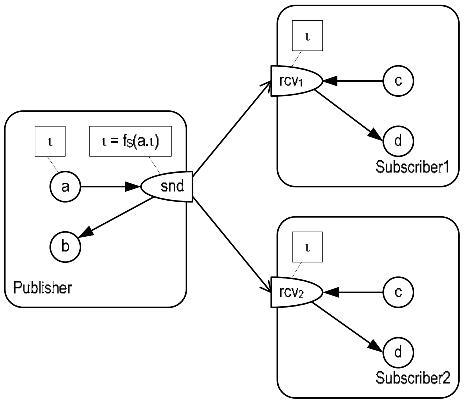
Figure 5-20
Multicast message-passing



Like the unconfirmed message-passing mechanism, the multicast message-passing mechanism does not provide synchronisation between remote participants. Conformance requirement *IR4* is therefore not satisfied. This interaction mechanism cannot be abstracted into a single abstract interaction.

To facilitate interaction design, we introduce a shorthand notation for multicast message-passing communication as depicted in Figure 5-21. This shorthand notation does not abstract from any design information of the multicast message-passing interaction mechanism.

Figure 5-21
Shorthand notation for multicast message-passing communication



5.3.8 Summary

Table 5-6 summarises the results of the abstractions in the previous subsections. 'Yes' means that an abstract interaction can be obtained to represent an interaction mechanism. 'No (shorthand)' means that an abstract interaction cannot be obtained and a shorthand notation is introduced to facilite interaction design.

Table 5-6
Summary of the abstract representations of interaction mechanisms

Section	Interaction mechanism	Abstraction
5.3.1	Unconfirmed message-passing	No (shorthand)
5.3.2	Provider confirmed message-passing	Yes
5.3.3	Synchronous request-response	Yes
5.3.4	Asynchronous request-response: callback	Yes
5.3.5	Asynchronous request-response: remote polling	Yes
5.3.6	Asynchronous request-response: local polling	Yes
5.3.7	Multicast message-passing	No (shorthand)

The fact that our interaction abstraction method cannot obtain abstract representations of every interaction mechanisms does not invalidate the abstraction method. Instead, it shows that the abstraction method can assess the possibility of a conformance relation between an interaction mechanism and an abstract interaction. When a conformance relation cannot be established, the abstraction method prevents an interaction mechanism from being represented as an abstract interaction.

It also shows that our interaction concept cannot represent every available interaction mechanism. This limitation is a consequence of choosing interaction synchronisation as a property of the interaction concept. The limitation makes our objective of obtaining abstract representations of common interaction mechanisms not fully achieved, i.e., some interaction mechanisms have to be expressed using shorthand notations. A shorthand notation provides a convenient graphical expression for an interaction mechanism, but it still requires a designer to think in terms of the detailed behaviour of the interaction mechanism.

5.4 Example of use

In this section, we illustrate the use of the abstract representations of interaction mechanisms that are obtained in Section 5.3 in an interaction design process. Figure 5-22 depicts an interaction design process of an interaction for applying for a credit card. Figure 5-22(i) depicts this credit card application as a single abstract interaction *ccApply*. The customer wants a credit limit that is higher than EUR 1000, while the bank only allows the maximum of EUR 5000. In Figure 5-22(ii), we refine interaction *ccApply* into a concrete interaction structure that consists of interactions *retrieve* and *apply* to model the retrieval of a credit card application form and the application for a credit card using that form, respectively.

We decide that interaction *retrieve* should be implemented as a synchronous request-response mechanism, in which the customer acts as the client and the bank acts as the server. The reason is that the bank can respond the customer's request by sending a requested application form back to the customer immediately. We decide that interaction *apply* should be implemented as an asynchronous request-response mechanism based on callback, in which the customer acts as the client and the bank acts the server. The reason is that the bank needs a couple of days for manual authorisation before sending a response to the customer.

To include these decisions into an interaction design, we annotate the design with information that indicates those decisions. In Figure 5-22(iii), annotations '(sync)' and '(async:cb)' indicate that the interactions must be implemented as a synchronous request-response and an asynchronous request-response based on callback, respectively. An interaction contribution has an indication of the role played by the participant. Annotation '(req)' or '(rsp)' that precedes an information attribute indicates that the annotated information attribute is the request or response message, respectively. Platform-specific information, e.g., of CORBA or

Web Services, can also be added to the interaction design (not shown in the figure).

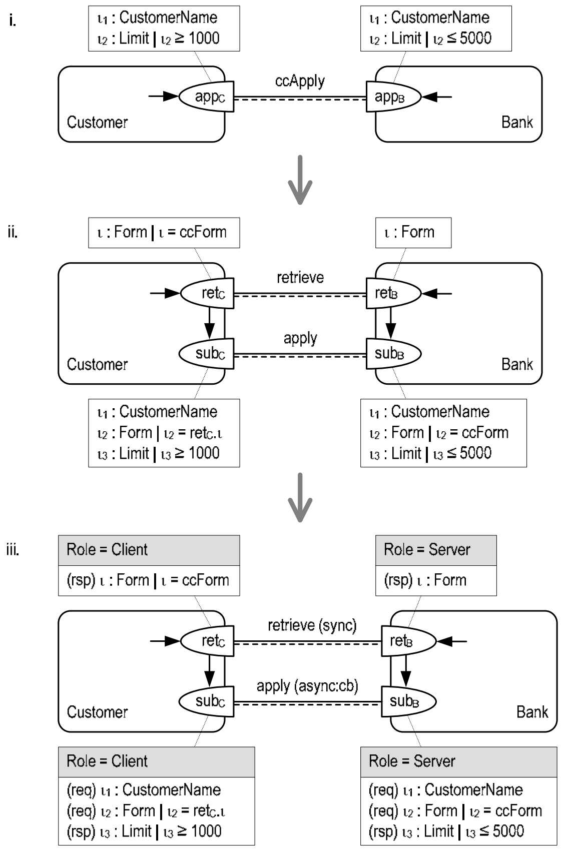


Figure 5-22
Design process of the interactions for a credit card application

This example shows that, when suitable and correct abstract representations of interaction mechanisms are available, a designer has only to develop an interaction design at a high abstraction level. Such abstract representations give the designer confidence that the abstract representations can be implemented using available standardised interaction mechanisms, without having to include explicitly the behaviour of interaction mechanisms in the interaction design. The abstract representations allow the designer to focus on the application or business logic of the interaction between the remote participants at his own abstraction level.

5.5 Related work

At a lower abstraction level, certain interaction structures may appear frequently [12]. Our abstraction approach can be used to obtain abstract representations of, e.g., the interaction patterns that are described in [16, 54, 70]. When such abstract representations are available, a designer does not have to define the same interaction structure multiple times in an interaction design. Instead, the designer can define them as abstract interactions (possibly with some indications about the targeted interaction patterns).

In [34], different interaction mechanisms are represented by different interaction design concepts. In contrast, our work represents abstract representations of different interaction mechanisms using the same interaction concept, i.e., the ISDL enhanced interaction concept defined in Chapter 3. This gives us two benefits as follows.

First, a composition of abstract representations can be further abstracted into an abstract interaction using the same abstraction method. We have shown this benefit when obtaining the abstract representations of the callback and remote-polling mechanisms in Section 5.3.4 and 5.3.5, respectively.

Second, at a higher abstraction level, a designer does not have to decide yet which interaction mechanisms should implement an abstract interaction. At a lower abstraction level, the designer has alternatives to implement an abstract interaction. We have shown this benefit in the example in Section 5.4.

[115, 116] uses the concept of *connector* to represent an abstract representation of an interaction mechanism. A connector is defined straightforwardly without examining the behaviour of an interaction mechanism. No conformance requirement or abstraction method is defined. This way of representation has no evidence on which to claim the correctness of a connector with respect to the behaviour of an interaction mechanism that it represents. Our abstraction approach defines conformance requirements between an interaction structure and its abstract representation. It uses a systematic abstraction method to check whether these conformance requirements are satisfied. In this way, one can have evidence that an abstract representation is correct.

A connector represents a specific interaction protocol, e.g., procedure calls, UNIX pipes, SQL links, or buffers [52, 78, 114], as a single concept. It is therefore suitable for representing an interaction mechanism. However, it forces a designer to think at an implementation level or about a specific implementation platform. Our abstract representations are independent from any interaction mechanism or implementation platform.

5.6 Concluding remarks

In this chapter, we have applied our abstraction method to obtain abstract representations of interaction mechanisms that are provided by communication middleware, i.e., CORBA and Web Services. These abstract representations allow a designer to specify interactions in a service composition, without being forced to consider the details of possible alternative interaction mechanisms at the early phase of a design process.

We have developed an approach to obtain an abstract representation of comparable interaction mechanisms provided by different communication middleware. The abstract representation is defined using the ISDL enhanced interaction concept. However, abstract representations of some interaction mechanisms, i.e., the unconfirmed message-passing and multicast message-passing mechanisms, cannot be obtained. To facilitate interaction design, we have introduced shorthand notations for those interaction mechanisms.

Since the behaviour of an interaction mechanism is pre-defined by communication middleware, the mapping between the interaction mechanism and its abstract representation can be defined. This mapping can be used to develop a transformation for refining an abstract interaction into an interaction structure that represents the interaction mechanism, by using (semi-)automatic transformation techniques. When such a transformation is available, a designer has only to indicate how the interaction mechanism should implement the abstract interaction, as illustrated by the example in Section 5.4. The transformation is then used to produce a correct implementation, i.e., a concrete interaction design or an executable implementation, based on this indication.

This chapter has shown that our interaction concept can be used to model precisely the behaviour of interaction mechanisms as a composition of interactions. The behaviour models of some interaction mechanisms can be abstracted into abstract interactions that preserve the essential properties of the interaction mechanisms. The interaction concept satisfies the requirement for modelling concrete interactions, as defined in Section 2.6.

Chapter 6

Transformation to executable implementations

The Model Driven Architecture (MDA) approach [90, 91], especially regarding automatic transformations, has been widely used and investigated to facilitate and speed up the implementation process of service composition [14, 22, 23, 35, 48, 61, 63, 64, 74]. For the same reason, we develop an automatic transformation tool to transform a service composition model to an executable implementation in BPEL (Business Process Execution Language, version 1.1 [20]). We call our transformation tool *the ISDL2BPEL transformation tool.*

This chapter explains the development of the ISDL2BPEL transformation tool. This chapter is organised as follows: Section 6.1 presents specification languages for service compositions that we use in the ISDL2BPEL transformation tool. Section 6.2 introduces our approach in the development of the ISDL2BPEL transformation tool. In this section, we argue that a service composition model must comply with certain modelling restrictions. Also, we present the decomposition of the transformation tool into three sub-transformations, namely *pattern recognition*, *constraint transformation*, and *model realisation*. Section 6.3 presents the modelling restrictions imposed on a service composition model that will be transformed using the transformation tool. Section 6.4 presents the pattern recognition. Section 6.5 presents the constraint transformation. Section 6.6 presents the model realisation. Section 6.7 discuses related work. Finally, Section 6.8 presents some concluding remarks.

6.1 Service compositions in Web Services

Web Services [133] have become a preferred implementation platform on which enterprises execute their services [42, 77]. This has motivated us to use Web Services as the target implementation platform of our transformation tool.

A service description describes the offered service of a service provider and/or the requested service of a service user. It contains, among others, a list of operations for interaction with a service provider or user. In a Web Service platform, a service description is specified in WSDL (Web Service Description Language [138, 139]).

As mentioned in Chapter 1, a service composition can be a choreography or an orchestration. In a Web Service platform, a choreography can be specified in WS-CDL (Web Services Choreography Description Language [135]), WSCI (Web Services Choreography Interface [136]), or WSCL (Web Services Conversation Language [137]). An orchestration can be specified in BPEL (Business Process Execution Language [20, 86]), WSFL (Web Services Flow Language [69]), or XLANG [124].

Nowadays, BPEL has become the de facto language for specifying Web Services orchestrations. Execution engines are available, e.g., [2, 97], to execute orchestrations that are specified in BPEL. Hence, an orchestration specified in BPEL is an executable implementation. The ISDL2BPEL transformation tool transforms an orchestration that is specified in ISDL to an executable implementation in BPEL.

6.1.1 WSDL

WSDL is a service description language that is based on XML [132]. A service description that is specified in WSDL is called a *WSDL description*. A WSDL description consists of the following definitions (we assume WSDL version 1.1 as it is used in the ISDL2BPEL transformation tool):

– *data types*
  Data types and elements are to be used in the message types definition below. Data types and elements are defined using XML Schema [142].
– *message types*
  A message type defines the type of messages that can be exchanged. A message consists of one or more logical parts; each of which has a type that can be either a data type or element that is defined in the data types definition above or an XML Schema type.
– *port types*
  A port type is a set of related operations. An operation includes a set of input, output and fault messages.
– *bindings*
  A binding defines communication protocols and message encoding to support the invocation of operations and the exchange of the associated messages. A port type may have multiple bindings.

– *ports*
  A port defines an endpoint for a binding. An endpoint specifies a network address at which a service that provides the port type is available.
– *services*
  A service groups a number of related ports.

The data types definition, message types definition, and port types definition make up the abstract part of a WSDL description. The bindings definition, ports definition, and services definition make up the concrete part [7]. The abstract part is independent of any protocol or message encoding. It is also independent of the location at which a service is available. Hence, it is reusable for different protocols, message encoding, and locations. The concrete part defines a specific protocol, message encoding, and location of a service.

WSDL defines four types of operations that an endpoint can support:
– *One-way operation*. The endpoint receives a message.
– *Request-response operation*. The endpoint receives a message and sends a correlated message.
– *Solicit-response operation*. The endpoint sends a message and receives a correlated message.
– *Notification operation*. The endpoint sends a message.

6.1.2 BPEL

BPEL is a language that is based on XML for specifying the coordinator of an orchestration on a Web Services platform. A coordinator that is specified in BPEL is called a *BPEL process*. A BPEL process is exposed to its users as a service provider that is described in WSDL.

The BPEL concepts that are used in the ISDL2BPEL transformation tool are as follows. (We assume BPEL version 1.1 as it is used in the ISDL2BPEL transformation tool)

Partner links

Service providers and users with which a BPEL process interacts are called *partners*. A logical connection between a BPEL process and partner is defined as a *partner link*. In a partner link, a BPEL process and partner play specific roles.

A partner link is an instance of a *partner link type*. A partner link type defines roles that have to be played by a BPEL process and partner. A role is associated with a WSDL port type. Only operations that are defined in that port type can be invoked within the partner link. Partner link types are defined as WSDL extensions, i.e., a partner link type is specified in a WSDL description to support a BPEL process.

Activities

BPEL distinguishes two kinds of activities: *basic activities* and *structured activities*. A basic activity represents certain functionality. A structured activity defines the execution order of other activities. The activities are as follows. (We only list BPEL activities that serve as target activities of the ISDL2BPEL transformation tool.)

The basic activities are:

– *invoke*
  This activity invokes an operation that is provided by a partner. If the invoked operation is a one-way operation, the execution of the BPEL process that specifies this activitiy continues immediately. If the invoked operation is a request-response operation, this activity then waits for and receives a response message. Upon the reception of the response message, the execution of the BPEL process continues.
– *receive*
  This activity waits for and receives a message from a partner. Upon the reception of a message, the execution of the BPEL process that specifies this activitiy continues.
– *reply*
  This activity sends a message as a response to a message that is received by a receive activity. It must be used together with a receive activity to provide a request-response operation.
– *assign*
  This activity assigns or updates variables with new values. A variable is associated with either an element defined in an XML schema, an XML Schema simple type, or a message type defined in a WSDL description.

The structured activities are:

– *sequence*
  This activity orders the execution of one or more activities sequentially.
– *flow*
  This activity allows concurrent execution of two or more activities.
– *switch*
  This activity allows alternative behaviours. It specifies a number of conditional activities and one optional default activity. The default activity is executed when none of the conditional activities can be executed, i.e., their conditions cannot be satisfied.
– *while*
  This activity executes another activity repeatedly as long as a repetition condition is satisfied.

– *pick*
  This activity waits for events to occur. An event can be the arrival of a message or the expiration of a timer. When an event occurs, an activity that is associated to that event is executed.

Compensation and fault handlers

Transaction processing in a BPEL process is defined using compensation. A compensation handler defines the activities for compensating the effect of another activity in a transaction. It can be executed only if the activity that it has to compensate completes normally.

A compensation handler can be invoked only within a fault handler or another compensation handler. A fault handler defines the activities that have to be executed when one or more faults occur during execution. It specifies the types of fault signals that can be handled.

Two activities to deal with compensation and faults are:

– *throw*
  This activity throws a fault signal that indicates that a fault occurs. This fault is to be handled by a fault handler. If a fault cannot be handled by a fault handler, it causes the execution of a BPEL process to terminate.
– *compensate*
  This activity invokes a compensation handler.

Remarks

As a specification language for orchestrations, BPEL provides sufficient support to implement activities for interacting with service providers and users, i.e., receive, reply, and invoke. On the other hand, BPEL provides limited support for data manipulation in the assign activity. By default, BPEL uses XPath 1.0 [141] for data manipulation. XPath is a query language for XML documents, which supports only simple arithmetic, boolean, and string manipulation. BPEL and XPath are not sufficient for complex data manipulation.

Several options have been proposed to deal with the limitation of BPEL on data manipulation, such as in [1, 19, 25, 63, 64, 87, 97]. In Section 6.5, we discuss those options and select one of them to be used in the ISDL2BPEL transformation tool.

6.2 Approach

In this chapter, we use the term ‘model’ to denote a design, in order to match to the term that is used by the MDA approach.

6.2.1 General approach

A service composition model, i.e., a model that specifies a service composition, is eventually transformed to an executable implementation on a target implementation platform. An implementation platform typically imposes certain requirements on the implementation that will be executed on it. These requirements determine the model transformation that should be done to transform a service composition model to an executable implementation. Figure 6-1 illustrates a transformation of a service composition model to an executable implementation that satisfies the requirements imposed by a target platform.

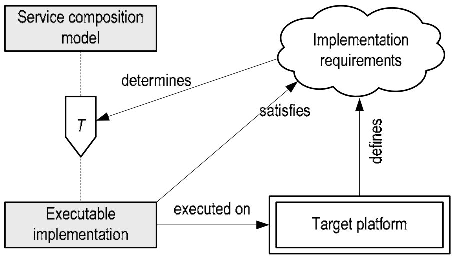

Service composition model
Implementation requirements
determines
T
satisfies
defines
Executable implementation
executed on
Target platform

Figure 6-1
Transformation of a service composition model to an executable implementation

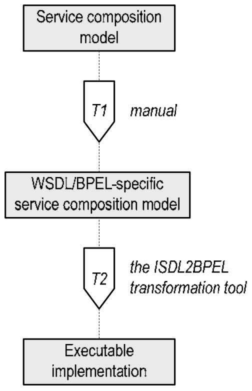
The definition of the solution for some implementation requirements cannot be automated because it involves a creative process for making design choices. For example, Web Services and BPEL support only message-passing and synchronous request-response interaction mechanisms. Any abstract interaction in a service composition model should be refined into these interaction mechanisms. This refinement involves a designer’s creative process that cannot be automated.

The definition of the solution for other implementation requirements can be automated, given sufficient platform-specific information. For example, an interaction that is annotated with information about the operation name that should be invoked, the port type in which that operation is defined, and the partner link that should be used for that operation invocation can be transformed to a complete BPEL invoke activity.

For those reasons, our approach to transforming a service composition model in ISDL to an executable implementation in BPEL consists of two smaller transformations, as depicted in Figure 6-2:

- *manual transformation T1*, which transforms a service composition model in ISDL to another service composition in ISDL that satisfies the implementation requirements imposed by BPEL of which the definition of their solution cannot be automated. This transformation includes annotation on the resulting model with WSDL/BPEL-specific information. The model resulting from this transformation is a WSDL/BPEL-specific model at an implementation level.
- *automatic transformation T2*, which transforms a service composition model resulted from transformation *T1* to an executable implementation in BPEL. The ISDL2BPEL transformation tool automates this transformation.

Figure 6-2
Manual and automated transformations


Service composition model
T1
manual
WSDL/BPEL-specific service composition model
T2
the ISDL2BPEL transformation tool
Executable implementation

Transformation *T1* can be considered as a transformation that prepares a service composition model that can be transformed by transformation *T2*. It produces a service composition model that complies to certain modelling restrictions. These restrictions include the restrictions on the design concepts that can be used, on the behaviour structures that can be formed, and on the way to specify constraints. Later in Section 6.3, we present these modelling restrictions in more detail.

BPEL is a language whose purpose is to specify the coordinator of an orchestration. Thus, the ISDL2BPEL transformation tool transforms only the coordinator model, i.e., the model of the coordinator of an orchestration, and not a complete service composition model.

6.2.2 Decomposition of the ISDL2BPEL transformation tool

This section gives an overview of the ISDL2BPEL transformation tool.

In order to deal with different tasks in transforming a coordinator model in ISDL to a BPEL process and WSDL extensions to support that BPEL process, we decompose the ISDL2BPEL transformation tool to three sub-transformation tools: *pattern recognition*, *constraint transformation*, and *model realisation*, as depicted in Figure 6-3. These sub-transformations are decoupled from each other by using intermediate models between sub-transformations. An intermediate model contains all the information that is necessary to produce a BPEL process. An intermediate model is also useful to examine whether a transformation produces a correct output model, given a certain input model. The language for specifying intermediate models is presented in Section 6.4.2.

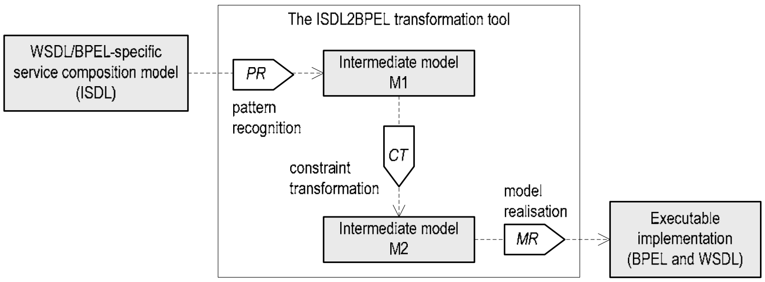

The ISDL2BPEL transformation tool
WSDL/BPEL-specific service composition model (ISDL)
PR
pattern recognition
Intermediate model M1
CT
constraint transformation
Intermediate model M2
model realisation
MR
Executable implementation (BPEL and WSDL)

Figure 6-3 Decomposition of the ISDL2BPEL transformation tool

Pattern recognition

The relations between activities in a coordinator model form a *behaviour structure*. This structure determines, amongst others, the execution order of the activities. In general, a behaviour structure is composed of generic structures representing well-known and frequently-used relations, such as sequence and concurrency. We use the term *behavioural patterns* to denote these generic structures. A pattern defines the relations between activities, without determining what activities are related. Patterns are typically nested to form a (more complex) behaviour structure.

In the transformation of a coordinator model to a BPEL process, behavioural patterns that are used in that coordinator model have to be recognised. In the ISDL2BPEL transformation tool, the recognised patterns are documented in an intermediate model. The task of recognising and documenting behavioural patterns that are used in a coordinator model is called *the pattern recognition*. It is presented in more detail in Section 6.4.

Constraint transformation

Constraints that are specified in a coordinator model can be contribution constraints (see Section 3.5.2), causality constraints (see Section 3.3.3), or repetition constraints (see Section 3.3.5). These constraints have to be transformed properly. The task of transforming constraints in a coordinator model is called *the constraint transformation*. It transforms an intermediate model that is produced by the pattern recognition to another intermediate model.

As mentioned in Section 6.1.2, BPEL supports limited data manipulation. We evaluate several options to overcome this limitation and select one to be used in the ISDL2BPEL transformation tool. The constraint transformation is to accomodate the selected option. It is presented in more detail in Section 6.5.

Model realisation

Finally, the intermediate model that is produced by the constraint transformation is realised as a BPEL process. This task is called *the model realisation*. It is presented in more detail in Section 6.6.

We developed the ISDL2BPEL transformation tool in Java by using the EMF (Ecplise Modelling Framework [40]). Chapter 8 uses the ISDL2BPEL transformation tool in a case study.

6.3 Modelling restrictions

This section presents the modelling restrictions, as mentioned in Section 6.2.1, that should be comply with by the coordinator model in ISDL that will be transformed to a BPEL process using the ISDL2BPEL transformation tool. The restrictions are derived from the implementation requirements imposed by Web Services and BPEL.

6.3.1 Activities

The modelling restrictions on the activities of a coordinator are as follows.

Restriction 1: *Each activity of a coordinator must be an activity that represents the contribution of that coordinator to the interaction with a service provide or user.*

A coordinator coordinates interactions between service users and providers in an orchestration. Activities in a coordinator are mainly activities for interacting with service providers and users. BPEL supports the implementation of a coordinator by providing basic activities for interaction, i.e., invoke, receive, and reply. BPEL assumes that any activity can be carried out as an interaction with other service provider or user.

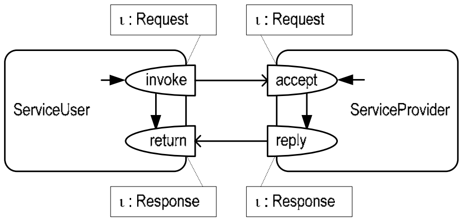
In ISDL, an interaction contribution represents the contribution of an entity to the interaction with other entitie(s). Activities in a coordinator are hence interaction contributions only.

Restriction 2: *Interaction contributions of a coordinator must represent operation calls and/or operation executions.*

The basic interaction in Web Services is an operation invocation that involves only two participants. One participant plays the role of a service provider. The other participant plays the role of a service user. The service user sends a request to call an operation that is provided by the service provider. The service provider executes that operation and returns the operation result as a response to the service user, if required. In an interaction, a coordinator plays either a service user or provider role.

We model an operation call as a pair of interaction contributions as depicted in Figure 6-4. Interaction contribution *invoke* sends a request and interaction contribution *return* receives the response. Similarly, we model an operation execution as a pair of interaction contributions. Interaction contribution *accept* receives a request and interaction contribution *reply* returns a response. A request sent by a service user via interaction contribution *invoke* is received by a service provider via interaction contribution *accept*. A response returned by the service provider via interaction contribution *reply* is received by the service user via interaction contribution *return*. Activities performed by the service provider during an operation execution can be inserted between interaction contributions *accept* and *reply*. In the service user, no activity may be inserted between interaction contributions *invoke* and *return*, because they model a synchronous operation call.

Figure 6-4
Operation call and execution

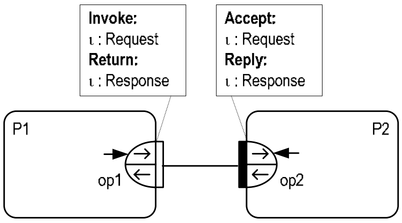


Restriction 3: *Operation calls and operation executions of a coordinator must be specified using their respective shorthands.*

Since all interactions must represent operation invocations, we define shorthands to specify operations calls and executions for convenience. Figure 6-5 depicts an interaction between two participants *P1* and *P2* using the shorthands for operation calls and executions. An operation call is graphically expressed as a segmented ellipse with a white rectangle attached to it. Interaction contributions *invoke* and *return* are indicated by arrows pointing toward and away from the white rectangle, respectively. An operation execution is graphically expressed as a segmented ellipse with a black rectangle attached to it. Interaction contributions *accept* and *reply* are indicated by arrows pointing away from and toward the white rectangle, respectively. In the figure, participant *P1* has an operation call and participant *P2* has an operation execution. Hence, in this operation invocation, participants *P1* and *P2* play the roles of a service user and provider, respectively.

Interaction contribution *invoke* of an operation call is called *the invoke part of an operation call*. Similarly, the other interaction contributions are called *the return part of an operation call*, *the accept part of an operation execution* and *the reply part of an operation execution*.

Figure 6-5
Shorthands for operation calls and operation executions

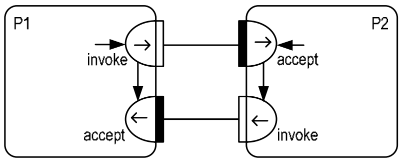


Attributes are specified in a text box attached to the segmented ellipse. In the figure, the invoke part of operation call *op1* has an information attribute of type *Request* and the return part of this operation call has an information attribute of type *Response*. The accept part of operation execution *op2* has an information of type *Request* and the reply part of this operation execution has an information of type *Response*.

A reference to an interaction contribution that is part of an operation can be made by appending the symbol '$' to the operation name followed by the name of the part, i.e., invoke, accept, reply or return. For example, *op1$invoke* refers to interaction contribution *invoke* of operation call *op1*.

The return part of an operation call and the reply part of an operation execution are optional. This is to allow us to model one-way operations. For example, the asynchronous request-response interaction mechanism based on callback can be modelled using two one-way operation invocation as depicted in Figure 6-6.

Figure 6-6
Callback as two one-way operation invocations



Besides modelling convenience, the shorthands facilitate the development of the ISDL2BPEL transformation tool. Given an operation call or execution, we know directly which interaction contributions correlate to each other. Otherwise, annotations are necessary to indicate correlations between interaction contributions [35].

Restriction 4: *Operation calls and executions of a coordinator must be annotated with WSDL/BPEL-specific information.*

An executable implementation includes WSDL/BPEL-specific information, such as operation name, port type, and partner link. This information are not available in a platform-independent coordinator model. Therefore, a coordinator model that will be transformed to a BPEL process must be annotated with that information. Annotations that must be given to a coordinator model are presented in Section 6.6.

6.3.2 Behaviour structure

The modelling restrictions on the behaviour stucture of a coordinator are as follows.

Restriction 5: *The behaviour structure of a coordinator must be a composition of allowed behavioural patterns, namely sequence, concurrency, selection, and repetition.*

This restriction is to allow the mapping from the allowed behavioural patterns onto the execution orders that are supported by BPEL. The sequence, concurrency, selection, and repetition patterns can be mapped onto the BPEL structured activity, i.e., sequence, flow, switch, and while, respectively. This restriction requires that any complex behaviour structure should be constructed as a composition of those patterns.

Figure 6-7 depicts the representations of the patterns in ISDL. For generality reason, the figure uses actions to represent activities. Figure 6-7(i) represents of the sequential execution of actions *a* and *b*. Figure 6-7(ii) represents the concurrent execution of actions *a* and *b*. Figure 6-7(iii) represents a choice or selection between the execution of action *a* and the execution of action *b*. Figure 6-7(iv) represents a repetition of zero or more instances of action *a*.

In the concurrency pattern, the conjunction is optional when no other activity has to be executed after the execution of all the activities in the concurrency pattern. Similarly, in the selection pattern, the disjunction is optional.

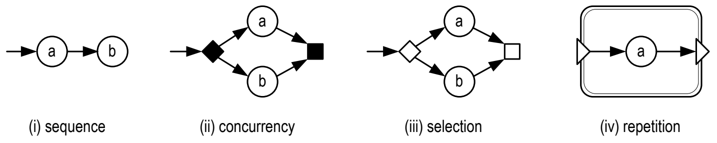


Figure 6-7
Representations of the behavioural patterns in ISDL

Restriction 6: *The concurrency, selection, and repetition patterns must be represented using their shorthands, i.e., and-split, or-split, and repetitive behaviour instantiation, respectively.*

The shorthands for *and-split*, *or-split*, and *repetitive behaviour instantiation* are described in Section 3.3.5. This restriction is to facilitate the recognition of behavioural patterns used in a coordinator model. For example, in Figure 6-8(i), the use of the shorthand for *and*-split makes it easy to recognise a concurrency pattern. An *and*-split explicitly indicates which activities can be executed concurrently, i.e., actions *b* and *c*. The behaviour in Figure 6-8(ii) is equivalent to the behaviour in Figure 6-8(i), but without the *and*-split shorthand. The concurrency of actions *b* and c is presented implicitly by two sequences {a → b} and {a → c}. Because concurrency is not modelled explicitly in this case, it is more difficult to recognise. Pattern recognition is described later in Section 6.4.

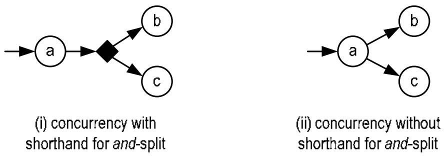
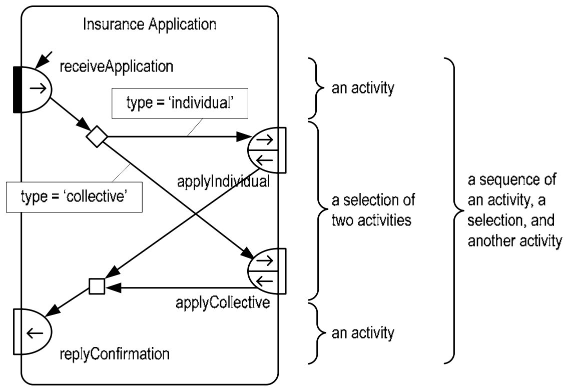

a
b
c
(i) concurrency with shorthand for and-split
(ii) concurrency without shorthand for and-split

Figure 6-8
Concurrencies with and without shorthand

Example

Figure 6-9 depicts the coordinator model of an insurance application. The coordinator receives an application from an applicant and checks whether the application is of type individual or collective. It then makes an operation call to a service provider according to the application type. When the coordinator receives a confirmation as a response, it forwards the confirmation to the applicant.

Figure 6-9
Patterns in the coordinator model of an insurance application



The behaviour structure of this coordinator model is composed of two patterns: sequence and selection patterns, in which the selection pattern is nested within the sequence pattern. The sequence pattern consists of operation execution *receiveApplication*, the composition of operation calls that forms the selection pattern, and operation call *replyConfirmation*. The selection pattern consists of two operation calls: *applyIndividual* and *applyCollective*. This coordinator model complies with the restrictions on activities and behaviour structure that have been discussed so far.

6.3.3 Constraints

The modelling restriction on the way to specify constraints, i.e., contribution constraints, causality constraints, or repetition constraints, in a coordinator is as follows.

Restriction 7: *Constraints in a coordinator must be specified as function calls.*

Simple constraints can be included in a coordinator model and leave the coordinator model easy to understand. Inclusion of complex constraints, however, potentially makes a coordinator model difficult to understand. To avoid that, some detail of a complex constraint can be encapsulated in a (parameterised) function that is specified in another document. In this way, complex constraints can be included in a coordinator model as function calls. A function specification can be informal, e.g., in natural languages, or formal, e.g., in mathematical expressions, pseudo-code, or programming languages.

Figure 6-10 depicts an example of a coordinator *Pricing* (interactions with other service providers are not shown). Operation execution *calculatePrice* receives an order and returns the total price of the order. The contribution constraint of the reply part of this operation execution specifies that the price must be equal to the result of function *calculatePrice()* with the received order as a parameter. This function is specified in another document, which is depicted in Figure 6-11. This document specifies the data manipulation that must be performed by the reply part of operation execution *calculatePrice* to establish its result.

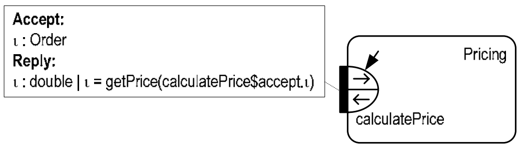


Figure 6-10
A function call in an attribute constraint

This approach separates the behaviour structure of a coordinator from the data manipulation that should be performed by that coordinator. The behaviour structure is defined by operations and behavioural patterns; the data manipulation is defined by function specifications.

Besides ease of understanding, constraints that are specified as function calls facilitate the development of the ISDL2BPEL transformation tool, i.e., they accommodate the option that we select to overcome the limitation of data manipulation in BPEL, as presented later in Section 6.5.

Figure 6-11
The specification of function *calculatePrice()* in pseudocode

```
function calculatePrice(Order order)

begin
  total_price = 0.0;
  for each line in order do
    total_price = total_price + (line.quantity × line.price);
  return total_price;
end.
```

6.4 Pattern recognition

This section presents an approach to recognise and document the behavioural patterns that form the behaviour structures of a coordinator model.

6.4.1 Tasks in behaviour structure transformation

The behaviour structure of a coordinator model is composed of behavioural patterns. Since behavioural patterns determine the execution order of activities, they must be transformed to BPEL structured activities.

In the transformation of the behaviour structure of a coordinator to BPEL activities, two successive tasks can be identified: *pattern recognition* and *pattern realisation*. The pattern recognition identifies patterns that form the behaviour structure of a coordinator model. The pattern realisation transforms the recognised patterns to BPEL structured activities. In the ISDL2BPEL transformation tool, the pattern realisation is a part of the model realisation as presented in Section 6.6.

ISDL represents a behavioural pattern as a set of causality relations. For example, ISDL represents the sequence of actions *a*, *b*, and *c* as two causality relations {a → b} and {b → c}. The pattern recognition should recognise that these casuality relations form a sequence of actions *a*, *b*, and *c*. In the ISDL2BPEL transformation tool, the pattern recognition also documents the recognised patterns in an intermediate model. This model is specified in a language that documents an allowed behavioural pattern, i.e., sequence, concurrency, selection, or repetition (see Restriction 5 in Section 6.3.2) as a single concept.

6.4.2 Common behavioural patterns language

The behavioural patterns in an ISDL coordinator model can be directly transformed to BPEL structured activities. In this case, the pattern recognition is combined with the pattern realisation. This direct transformation of an ISDL coordinator model to a BPEL process implies that ISDL interaction contributions must be transformed into BPEL invoke, receive, and/or reply activities at the same time. Furthermore, constraints in the coordinator model must be transformed at the same time. This approach results in a complex monolithic transformation tool. We decompose the ISDL2BPEL transformation tool, as depicted in Figure 6-3, in order to carry out those different tasks or sub-transformations separately. We use intermediate models between sub-transformations.

An intermediate model is specified in a language that is able to represent a behavioural pattern as a single concept. To facilitate and support other transformations, the language should be simple, yet able to contain all design information in a coordinator model. We define such a language and call it CBPL (Common Behavioural Pattern Language). It uses the term 'common' because the behavioural patterns that can be documented are common to many specification languages [10, 29, 144].

Figure 6-12 depicts a subset of the CBPL metamodel that is intended for documenting patterns and a behaviour structure. An *activity* is an abstract concept that represents an activity to be performed by a coordinator. An activity can be an interaction or a structured activity. An *interaction activity* represents an activity for interacting with another service. The possible types for an interaction activity are the parts of an operation call or execution, i.e., *invoke*, *return*, *accept*, and *reply*. A *structure activity* represents an activity that determines the execution order of other activities.

Behavioural patterns are specialisations of the structured activity. A *sequence* represents one or more ordered activities to be executed sequentially. A *concurrency* represents two or more activities that can be executed concurrently. A *selection* represent a choice or selection between one or more cases. A *case* represent a case constraint and an activity which is executed when the case constraint is satisfied. A *default case* is selected when other cases cannot be selected because their case constraints cannot be satisfied. A *repetition* contains an activity to be executed repeatedly while its repetition constraint holds.

A *behaviour* defines the behaviour of a coordinator. It contains only one activity to be performed. When a coordinator should perform many activities, the activity of the behaviour of this coordinator is a structured activity that is composed of those activities.

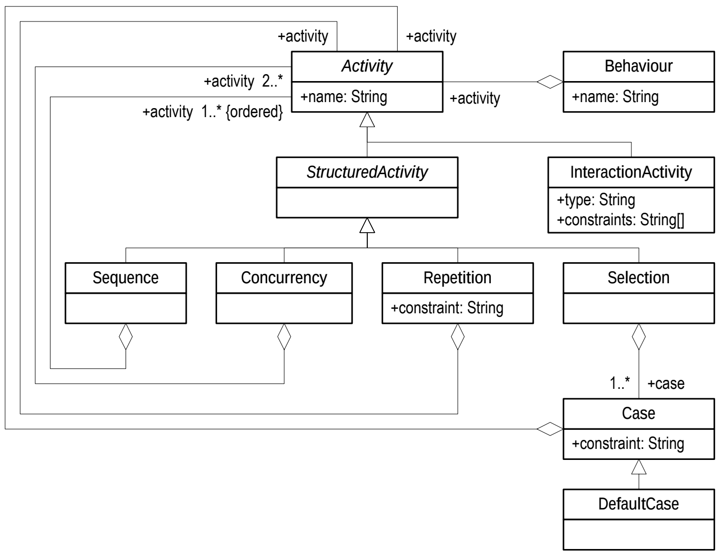


Figure 6-12
CBPL metamodel

An intermediate model must contain all the information that is necessary to produce a BPEL process. It therefore should also include representations of attributes of ISDL interaction contributions, parameters and parameter value assignments of ISDL entry and exit points, and placeholders for WSDL/BPEL-specific informations. We do not show those representations in Figure 6-12 because they are not related to pattern recognition and documentation.

Example

Figure 6-13 depicts an intermediate model in CBPL that documents the behavioural patterns used in the coordinator model in Figure 6-9. A CBPL intermediate model is expressed in XML.

Figure 6-13
An intermediate model in CBPL

```
<cbpl:sequence>
      <cbpl:interactionActivity
         name=”receiveApplication$accept” ... />
   <cbpl:selection>
      <cbpl:case constraint=”type = ‘individual’”>
         <cbpl:sequence>
            <cbpl:interactionActivity
               name=”applyIndividual$invoke” ... />
            <cbpl:interactionActivity
               name=”applyIndividual$return” ... />
         </cbpl: sequence>
      </cbpl:case>
      <cbpl:case constraint=”type = ‘collective’”>
         <cbpl:sequence>
            <cbpl:interactionActivity
               name=”applyCollective$invoke” ... />
            <cbpl:interactionActivity
               name=”applyCollective$return” ... />
         </cbpl: sequence>
      </cbpl:case>
   </cbpl:selection>
   <cbpl:interactionActivity
      name=”replyConfirmation$invoke” ... />
</cbpl:sequence>
```

6.5 Constraint transformation

Restriction 7 in Section 6.3.3 requires that a constraint must be specified as a function call. It separates the behaviour structure of a coordinator from the data manipulation that should be performed by that coordinator. We call their implementations *behaviour-structure implementation* and *function implementation*, respectively. The pattern recognition and realisation mentioned in Section 6.4 deal with the transformation of the behaviour structure of a coordinator model to a behaviour-structure implementation. This behaviour-structure implementation is specified as a BPEL process.

This section presents an approach to transform a function call to an operation call. In this operation call, the operation provides the function implementation. We use this approach to deal with the BPEL limitation on data manipulation. We select this approach after evaluating a number of options for function implementation.

6.5.1 Evaluation and selection criteria

A function specification can be informal, e.g., in natural languages, or formal, e.g., in mathematical expressions, pseudo-code, or programming languages. It is mainly obtained by manual transformation from the requirements of that function. To be executable, a function specification must be defined in a programming language. In this case, BPEL and XPath is considered as a programming language. To our knowledge, when a function specification is defined informally or in a mathematical expression or pseudo-code, there is not yet an effective and efficient way to transform a function specification to a function implementation automatically.

The behaviour-structure and function implementations of a coordinator model can be mapped onto the same or distinct implementation artefacts. In case they have to be mapped onto the same implementation artefact, an additional transformation is required to merge them into the same implementation artefact.

To analyse options for data manipulation in a BPEL process, we define the following criteria.

- *Feasibility*: What support is available for implementing a function specification?
- *Efficiency*: What is the execution efficiency of a function call?
- *Reusability*: Can a function implementation be reused by multiple behaviour-structure implementations?
- *Merging*: Does a function implementation have to be merged with a behaviour-structure implementation?
- *Portability*: Does a function implementation allow a behaviour-structure implementation to be portable between different BPEL execution engines? The function and behaviour-structure implementations do not have to be merged into the same implementation artefact.

Table 6-1 lists the quality and quantity values to be assigned to those criteria.

In the development of the ISDL2BPEL transformation tool, we give all criteria the same weight of importance. If possible, we want to select an option that has full feasibility, high efficiency, full reusability, no merging, and is portable. Quantitatively, we select an option that has the highest score.

One may define different weights of importance for different criteria when some criteria are considered more important than others. The selected option determines the constraint transformation that should be developed.

Table 6-1
Quantity and quality values for the evaluation and selection criteria

Criteria	Values	
	Quality	**Quantity**
Feasibility	Full	1.0
	Limited	0.5
	None	0.0
Efficiency	High	1.0
	Middle	0.5
	Low	0.0
Reusability	Full	1.0
	Limited	0.5
	None	0.0
Merging	Yes	0.0
	No	1.0
Portability	Yes	1.0
	No	0.0

To determine the score of an option, we define the following formula.

$$\text{Score} = (w_1 \times F) + (w_2 \times E) + (w_3 \times R) + (w_4 \times M) + (w_5 \times P)$$

where,	w_i ($i = 1..5$)	: weight of importance
	F , E, R, M, P	: quantity values of feasibility, efficiency, reusability, merging, and portability criteria, repectively

6.5.2 Options

The options for implementing function specifications are as follows.

Option 1: BPEL and XPath

A function implementation is defined in BPEL and XPath as part of a behaviour-structure implementation. Data manipulation is done in BPEL assign activities using XPath expressions. Structured activity constructs, e.g., *while* and *switch*, can also be used for data manipulation.

- *Feasibility*: BPEL structured activity and XPath provide limited support for implementing complex function specifications.
- *Efficiency*: The function implementation and behaviour-structure implementation are executed in the same execution instance. Overhead in calling a function is low; the execution efficiency is high.
- *Reusability*: The function implementation can only be used by the behaviour-structure implementation in which the function implementation is defined.

– *Merging*: The function implementation have to be merged with the behaviour-structure implementation to produce an executable implementation.
– *Portability*: The behaviour-structure implementation includes the function implementation. Since the function implementation is defined in BPEL and XPath, the behaviour-structure implementation is supported by, and hence portable between, different BPEL execution engines.

Option 2: Embedded code
A function implementation is defined in a general-purpose implementation language. The function implementation is then embedded in a behaviour-structure implementation. This option is supported by extensions to BPEL, such as BPELJ [19] and Java embedding [97].
– *Feasibility*: A general-purpose implementation language, e.g., Java, typically provides full support for implementing complex function specifications.
– *Efficiency*: The function implementation and the behaviour-structure implementation are executed in the same execution instance. Overhead in calling a function is low; the execution efficiency is high.
– *Reusability*: The function implementation can only be used by the behaviour-structure implementation in which the function implementation is embedded.
– *Merging*: The behaviour-structure implementation must be merged with the function implementation to produce an executable implementation.
– *Portability*: The behaviour-structure implementation can only be executed on a BPEL execution engine that supports the extension.

Option 3: Server functions
A function implementation is defined in a general-purpose implementation language. After compilation, the function implementation is deployed in a BPEL execution engine on which a behaviour-structure implementation will be executed. The function and behaviour-structure implementations are on different implementation artefacts. In execution, the behaviour-structure implementation calls the function implementation. This option is supported by extensions to BPEL, such as custom functions [1].
– *Feasibility*: A general-purpose implementation language, e.g., Java or C#, typically provides full support for implementing complex function specifications.
– *Efficiency*: The function implementation and the behaviour-structure implementation are executed in different execution instances. A function call establishes an interprocess communication between those execution instances. The execution efficiency is lower than the

execution efficiency of the previous options because of the higher calling overhead.
- *Reusability*: The function implementation can be used by multiple behaviour-structure implementations that run on the same BPEL execution engine. Behaviour-structure implementations on different BPEL execution engine cannot use the same function implementation.
- *Merging*: The behaviour-structure implementation and the function implementation are deployed separately. No merging is required.
- *Portability*: The behaviour-structure implementation can be executed only on a BPEL execution engine in which the function implementation is deployed. Not every BPEL execution engine supports this extension.

Option 4: Function call as operation call

A function implementation is defined as one or more operations that are provided by Web service(s). A function call in a constraint of a coordinator model is implemented as an operation call. This option is used in [25, 63, 64, 87].
- *Feasibility*: The function implementation can be defined in a general-purpose implementation language, e.g., Java or C#. Such an implementation language typically provides full support for implementing complex function specifications.
- *Efficiency*: The function implementation and the behaviour-structure implementation are executed in different execution instances. Possibly, they run on different execution engines. A function call establishes an interprocess communication (via a communication network) between those execution instances. Execution efficiency is low because of the high calling overhead.
- *Reusability*: Since the function implementation is provided as Web Services operations, it can be used by multiple behaviour-structure implementations.
- *Merging*: The behaviour-structure implementation and the function implementation are deployed separately. No merging is required.
- *Portability*: The behaviour-structure implementation is defined only in BPEL. Therefore, it is portable between different BPEL execution engines.

Summary

The analysis is summarised in Table 6-2. We conclude that option 4, function call as operation call, seems the best choice since it has the highest score. To improve its efficiency, option 4 can be combined with option 1, such that simple arithmetic operations are implemented in BPEL and XPath, instead of providing them as Web Service operations. For example, a repetition typically uses addition or subtraction operation to increase or decrease its repetition index. Implementing these arithmetic operations in BPEL and XPath will improve the execution efficiency significantly.

Table 6-2
Comparison between options

Criteria	Options			
	1. BPEL and XPath	2. Code embedding	3. Server functions	4. Function as operation
Feasibility	limited	full	full	full
Efficiency	high	high	middle	low
Reusability	none	none	limited	full
Merging	yes	yes	no	no
Portability	yes	no	no	yes
Score	2.5	2.0	3.0	4.0

6.5.3 Transformation rules

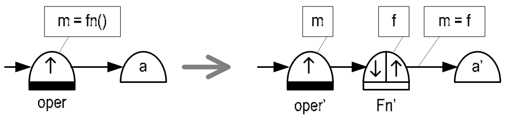
The ISDL2BPEL transformation tool uses option 4 (function call as operation call) to transform constraints in a coordinator model. Five transformation rules are defined and implemented; each of which deals with constraints that are associated with different model elements. The rules are independent from each other. They can be applied in any order.

These transformation rules are applied to a CBPL model and result in another CBPL model. To illustrate the rules, we use the ISDL notations because we do not define notations for the CBPL concepts. For generality reasons, an interaction contribution, in this section, is used to represent an operation call and execution.

Rule 1

This rule deals with *the contribution constraint of the accept or return part of an operation*. This contribution constraint determines the message that can be received by the accept or return part. When such a message is received, the execution of a coordinator may continue. Figure 6-14 illustrates this rule.

Figure 6-14
Rule 1



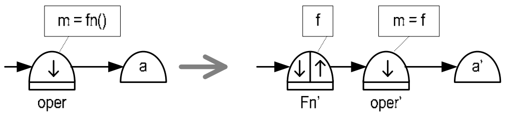
Operation execution *oper* enables interaction contribution *a*. The accept part of this operation execution has contribution constraint [m = fn()], where *m* is the message to be received and *fn()* is a function call. This operation execution only receives message *m* that is equal to the result of *fn()*.

The transformation should replace this structure with operation execution *oper'*, operation call *Fn'*, and interaction contribution *a'* that are to be executed sequentially. Operation execution *oper'* implements a part of operation execution *oper*, i.e., receiving a message. Operation call *Fn'* calls an operation that implements function *fn()* and returns a message *f*. Interaction contribution *a'* implements action *a*. The causality relation between operation call *Fn'* and interaction contribution *a'* has causality constraint [m = f]. This constraint is equivalent to original contribution constraint [m = fn()], in which *fn()* is replaced with the result of operation call *Fn'*.

Rule 2

This rule deals with *the contribution constraints of the invoke or reply part of an operation*. These contribution constraints determine the message that can be sent by the invoke or reply part. Figure 6-15 illustrates this rule.

Figure 6-15
Rule 2



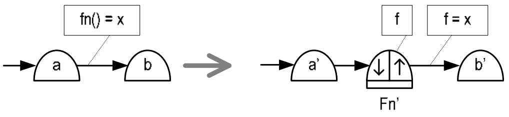
Operation call *oper* enables interaction contribution *a*. The invoke part of this operation call has contribution constraint [m = fn()]. This operation call only sends message *m* that is equal to the result of *fn()*.

The transformation should replace this structure with operation call *Fn'*, operation call *oper'*, and interaction contribution *a'* that are to be executed sequentially. Operation call *Fn'* calls an operation that implements function *fn()* and it returns a message *f*. Operation call *oper'* implements a part of operation call *oper*, i.e., sending a message. The invoke part of this operation call has contribution constraint [m = f]. This constraint is equivalent to original contribution constraint [m = fn()], in which *fn()* is replaced with the result of operation call *Fn'*.

Rule 3

This rule deals with *the causality constraints of an enabling condition*. Figure 6-16 illustrates this rule.

Figure 6-16
Rule 3



Interaction contribution *a* enables interaction contribution *b*. The enabling condition of interaction contribution *b* has causality constraint [fn() = x], where *x* is a certain value. Interaction contribution *b* can occur only if interaction contribution *a* occurs and this causality constraint is satisified.

The transformation should replace this structure with interaction contribution *a'*, operation call *Fn'*, and interaction contribution *b'* that are to be executed sequentially. Interaction contributions *a'* and *b'* implement interaction contributions *a* and *b*, respectively. Operation call *Fn'* calls an operation that implements function *fn()* and it returns a message *f*. The enabling condition of interaction contribution *b'* has causality constraint [f = x]. This constraint is equivalent to original causality constraint [fn() = x], in which *fn()* is replaced with the result of operation call *Fn'*.

Rule 4

This rule deals with *the causality constraints in an or-split*. Figure 6-17 illustrates this rule.

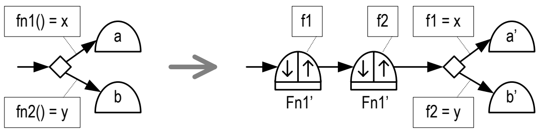


Figure 6-17
Implementation of causality constraint in OR-split

An or-split is used to define the choice between interaction contributions *a* and *b*. Interaction contribution *a* may occur if causality constraint [fn1() = x] is satisified, and interaction contribution *b* may occur if causality constraint [fn2() = y] is satisified.

The transformation should replace this structure with operation calls *Fn1'* and *Fn2'* that are to be executed sequentially, followed by a choice between interaction contributions *a'* and *b'*. Operation calls *Fn1'* and *Fn2'* call operations that implement functions *fn1()* and *fn2()*, respectively. They return a message *f1* and *f2*, respectively. Interaction contributions *a'* and *b'* implement interaction contributions *a* and *b*, respectively. The causality constraint of interaction contribution *a'* is [f1 = x] that is equivalent to the original causality constraint [fn1() = x], in which *fn1()* is replaced with the result of operation call *Fn1'*. The causality constraint of interaction contribution *b'* is [f2 = y] that is equivalent to the original causality constraint [fn2() = y], in which *fn2()* is replaced with the result of operation call *Fn2'*. If the causality constraints of interaction contributions *a* and *b* is defined by the same function *fn()*, e.g., [fn() = x] for interaction contribution *a* and [fn() = y] for interaction contribution *b*; only one operation call *Fn'* has to be defined preceeding the choice between interaction contributions *a'* and *b'*.

***Rule* 5**

This rule deals with *the repetition constraint of a repetitive behaviour instantiation*. Figure 6-18 illustrates this rule.

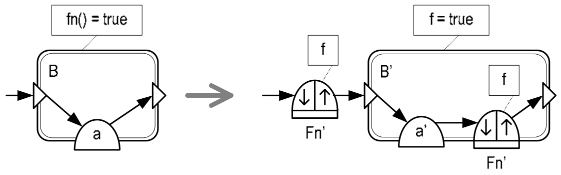


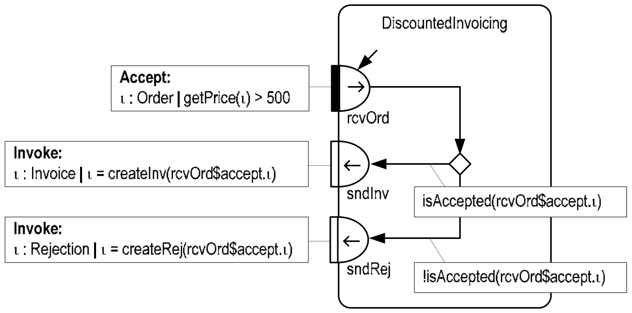
Figure 6-18
Rule 5 for unchanging condition

Repetitive behaviour instantiation *B* contains interaction contribution *a* to be performed repeatedly while repetition constraint [fn() = true] holds. In a general case, behaviour *B* may contain a number of related interaction contributions.

The transformation should replace this structure with operation call *Fn'* that is followed by repetitive behaviour instantiation *B'*. This behaviour contains interaction contribution *a'* that is followed by another operation call *Fn'*. Operation call *Fn'* returns a message *f*. Interaction contribution *a'* implement interaction contribution *a*. Repetitive behaviour *B'* has repetition constraint [f = true]. This constraint is equivalent to original causality constraint [fn() = true], in which *fn()* is replaced with the result of operation call *Fn'*.

6.5.4 Example

We apply the transformation rules to a service provider *DiscountedInvoicing* in Figure 6-19. The execution of this service provider starts when it receives an order from a customer with price higher than 500 euro [getPrice(order) > 500]. It checks whether the order can be accepted. If so, the service provider returns an invoice to the customer. Otherwise, the service returns a rejection message.



DiscountedInvoicing
Accept:
ι : Order | getPrice(ι) > 500
rcvOrd
Invoke:
ι : Invoice | ι = createInv(rcvOrd$accept.ι)
sndInv
isAccepted(rcvOrd$accept.ι)
Invoke:
ι : Rejection | ι = createRej(rcvOrd$accept.ι)
sndRej
!isAccepted(rcvOrd$accept.ι)

Figure 6-19
A service provider

First, we apply Rule 1 to the contribution constraint of operation execution *rcvOrd*. It results in coordinator *DiscountedInvoicing1* as depicted in Figure 6-20. For brevity, constraints and attributes that are not relevant to the application of this rule are omitted. The function call *getPrice()* is transformed to an operation call *getPrice* to a service provider that implements function *getPrice()*. The or-split is now enabled with a causality constraint that refers to the information attribute of the return part of operation call *getPrice* [getPrice$return.ι > 500].

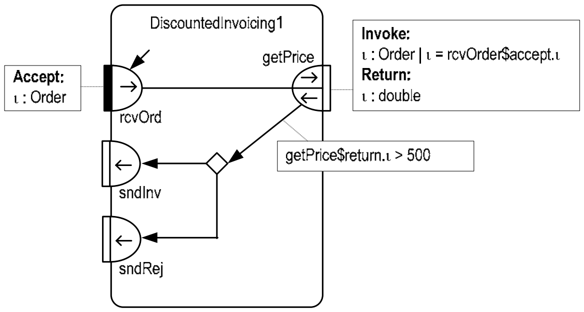


Figure 6-20
A coordinator resulted from Rule 1

Secondly, we apply Rule 4 to the causality constraints in the or-split. It results in coordinator *DiscountedInvoicing2* as depicted in Figure 6-21. Function call *isAccepted()* is transformed to operation call *isAccepted*. The causality constraints of operation call *sendInv* and *sendRej* now refer to the information attribute of the return part of operation call *isAccepted* [isAccepted$return.ι] and [!isAccepted$return.ι], respectively.

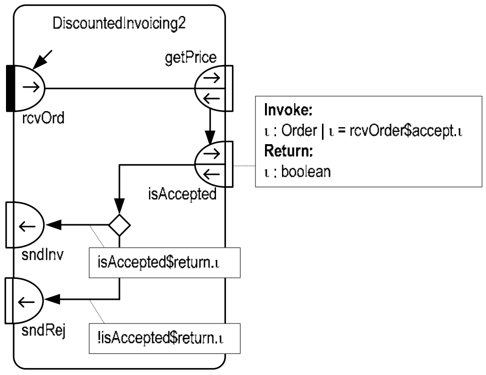


Figure 6-21
A coordinator resulted from Rule 4

Finally, we apply Rule 2 to the contribution constraints of operation calls *sndInv* and *sndRej*. It results in coordinator *DiscountedInvoicing3* as depicted in Figure 6-22. Function calls *createInv()* and *createRej()* are transformed to operation calls *createInv* and *createRej*, respectively. The contribution constraints of operation calls *sndInv* now refers to information attribute of the return part of operation call *createInv* [sndInv.ι = createInv$return.ι]. The contribution constraints of operation calls *sndRej* now refers to information attribute of the return part of operation call *createRej* [sndRej.ι = createRej$return.ι].

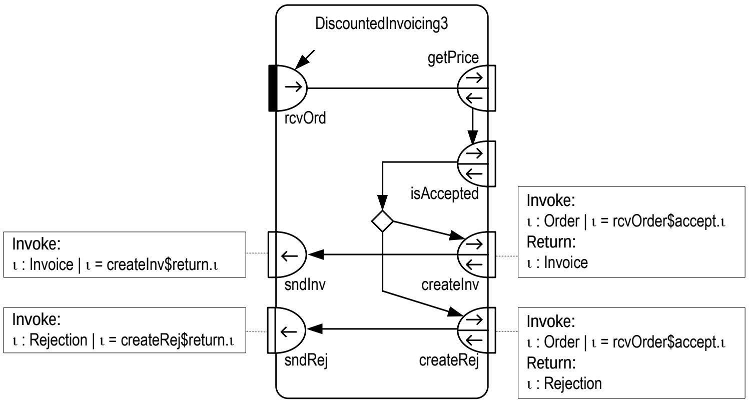

DiscountedInvoicing3
getPrice
rcvOrd
isAccepted
Invoke:
ι : Order | ι = rcvOrder$accept.ι
Return:
ι : Invoice
Invoke:
ι : Invoice | ι = createInv$return.ι
sndInv
createInv
Invoke:
ι : Rejection | ι = createRej$return.ι
sndRej
createRej
Invoke:
ι : Order | ι = rcvOrder$accept.ι
Return:
ι : Rejection

Figure 6-22
A coordinator resulted from Rule 2

A service provider that implements the specification of all functions called in service *DiscountedInvoicing3* has to be developed separately. Figure 6-23 depicts such a service provider called *FunctionService*.

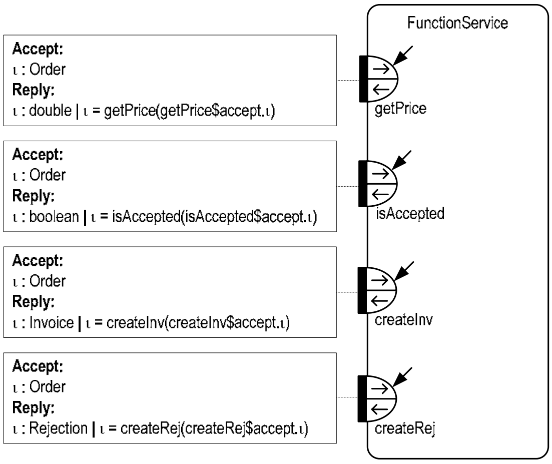

FunctionService
Accept:
ι : Order
Reply:
ι : double | ι = getPrice(getPrice$accept.ι)
getPrice
Accept:
ι : Order
Reply:
ι : boolean | ι = isAccepted(isAccepted$accept.ι)
isAccepted
Accept:
ι : Order
Reply:
ι : Invoice | ι = createInv(createInv$accept.ι)
createInv
Accept:
ι : Order
Reply:
ι : Rejection | ι = createRej(createRej$accept.ι)
createRej

Figure 6-23
A service provider that implements function specifications

6.6 Model realisation

This section presents an approach to transform a CBPL model to a BPEL process and WSDL extenstions.

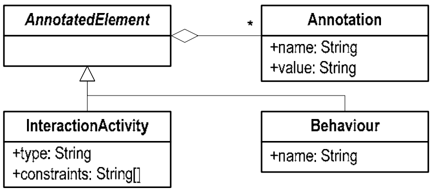
6.6.1 Annotations

Restriction 4 in Section 6.3.1 requires that an operation call or execution should be annotated with WSDL/BPEL-specific information. ISDL provides a stereotyping mechanism that is similar to the UML 1.3 stereotyping mechanism [94], i.e., the mechanism adds a stereotype and/or tagged values to a model element. We use the ability to add tagged values for annotating operation calls and executions with WSDL/BPEL-specific information.

When an ISDL coordinator model is transformed into a CBPL intermediate model, its annotations have to be maintained such that the CBPL intermediate model can be later transformed to a BPEL process. Figure 6-24 depicts a subset of the CBPL metamodel that is for maintaining annotations given in an ISDL coordinator model.

An *annotated element* can be an *interaction activity* or a *behaviour* (see Figure 6-12). An annotated element may have a number of annotations. An *annotation* consists of a *name* and a *value*. A behaviour can be annotated with, e.g., a namespace that should be given to the BPEL process that is represented by that behaviour.

Figure 6-24
Annotation in CBPL



6.6.2 Operations

Table 6-3 lists the annotations that must be given to an operation call or execution.

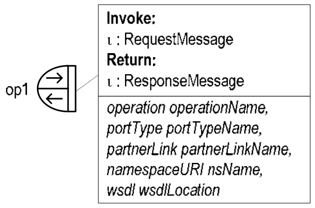
Table 6-3
Annotations for an operation

Name	Value
operation	The name of the operation to be called or executed. This annotation is optional when the operation call or execution already represents the name of the operation to be called or executed.
portType	The portType in which the operation is defined
partnerLink	The partnerLink in which the portType is used
namespaceURI	The namespace of the portType
wsdl	The location, i.e., URI (Uniform Resource Identifier), of the WSDL description in which the portType is defined

Operation call

Figure 6-25 depicts an operation call *op1* with annotations. The annotations are defined in the lower text boxt. This annotated operation call can be transformed into the BPEL process as depicted in Figure 6-26.

Figure 6-25
An annotated operation call



The operation call is transformed into a BPEL invoke activity of which the name is equal to the name of the operation call. The information attributes of the invoke and return part of the operation call are transformed into the input and output variables of the invoke activity, respectively. The values of annotations *operation*, *portType*, and *partnerLink* supply the values of the *operation*, *portType*, and *partnerLink* attributes of the BPEL invoke activity, respectively. For each information attribute of the operation call, a BPEL variable is created. Annotation *partnerLink* is also transformed into a BPEL partner link. This partner link specifies that the partner plays the role of 'provider'. Annotation *namespaceURI* is used to define a namespace and namespace alias in the BPEL process.

Figure 6-26
A BPEL process from the transformation of an operation call

```
<bpel:process
   xmlns:ns0="wsdlNamespace"
   xmlns:ns1=”nsName” ... >

   <bpel:partnerLink name=”partnerLinkName”
      partnerLinkType=”ns0:partnerLinkNamePLT”
      partnerRole=”provider” />

   <bpel:variable name=”op1.request”
      messageType=”ns1:RequestMessage” />

   <bpel:variable name=”op1.response”
      messageType=”ns1:ResponseMessage” />

   <bpel:invoke name=”op1”
      inputVariable=”op1.request”
      outputVariable=”op1.response”
      operation=”operationName”
      portType=”ns1:portTypeName”
      partnerLink=”partnerLinkName” />
```

```
</bpel:process>
```

When an operation call is used to model a one-way operation call, the transformation results in a BPEL invoke activity without attribute *outputVariable*.

The transformation also results in WSDL extensions as depicted in Figure 6-27. Annotation *partnerLink* is transformed into a partner link type. The name of this partner link type is equal to the value of this annotation that is appended with *PLT*. This partner link type defines a role named 'provider'. This role is associated with a port type that is specified in annotation *portType*. Annotations *namespaceURI* and *wsdl* are used to import the WSDL description that defines the port type.

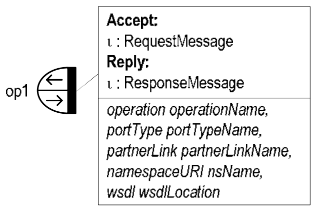
Figure 6-27
WSDL extension from the transformation of an operation call

```
<wsdl:definitions
   targetNamespace="wsdlNamespace"
   xmlns:ns1="nsName" ... >

   <plnk:partnerLinkType name="partnerLinkNamePLT">
      <plnk:role name="provider">
         <plnk:portType name="ns1:portTypeName"/>
      </plnk:role>
   </plnk:partnerLinkType>

   <wsdl:import
      location="wsdlLocation"
      namespace="nsName"/>

</wsdl:definitions>
```

Operation execution

An annotated operation execution as depicted in Figure 6-28 can be transformed into the BPEL process as depicted in Figure 6-29. The transformation also results in WSDL extensions that are equal to WSDL extensions as depicted in Figure 6-27.

Figure 6-28
An annotated operation execution



The operation execution is transformed into a BPEL receive and reply activity. The activity names are equal to the name of the operation execution appended with *_accept* and *_reply*, respectively. The information attributes of the invoke and return parts of the operation execution are transformed into the variables of the BPEL receive and reply activities, respectively. Annotation *partnerLink* is transformed into a BPEL partner link. This partner link specifies that the BPEL process plays the role of 'provider'. Other annotations are used in the same way as in the transformation of an operation call.

Figure 6-29
A BPEL process from the transformation of an operation execution

```
<bpel:process
   xmlns:ns0="wsdlNamespace"
   xmlns:ns1=”nsName” ... >

   <bpel:partnerLink name=”partnerLinkName”
      partnerLinkType=”ns0:partnerLinkNamePLT”
      myRole="provider" />

   <bpel:variable name=”op1.req”
      messageType=”ns1:RequestMessage” />

   <bpel:variable name=”op1.rsp”
      messageType=”ns1:ResponseMessage” />

   <bpel:receive name=”op1_accept”
      variable=”op1.request”
      operation=”operationName”
      portType=”ns1:portType”
      partnerLink=”partnerLinkName” />

   <!-
      activities for generating the response message are
      inserted here.
   -->

   <bpws:reply name=”op1_reply”
      variable=”op1.response”
      operation=”operationName”
      portType=”ns1:portType”
      partnerLink=”partnerLinkName” />

</bpel:process>
```

When an operation execution is used to model a one-way operation execution, the transformation results in a BPEL receive activity only.

Operation call and operation execution

The interaction between a service provider and user may consist of a combination of an operation call and execution. For example, a callback interaction mechanism can be modelled for the service provider as an operation execution that is eventually followed by an operation call. The operation execution is used only for receiving a request message from the service user and, therefore, it has only the accept part. The operation call is used only for sending a response message to the service user and, therefore, it has only the invoke part. Figure 6-30 depicts these operation execution and call. To indicate that the operation execution and call are related to each other, they must have the same value for annotation *partnerLink*.

The transformation of operation execution *op1* and operation call *op2* that are depicted in Figure 6-30 results in two different roles in a BPEL partner link, as depicted in Figure 6-31, and its partner link type, as depicted in Figure 6-32. The partnerlink specifies that this BPEL process plays the role of 'provider', while the partner plays the role of 'requester'.

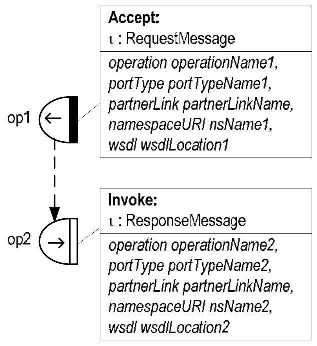


Figure 6-30
Callback interaction mechanism in a service provider

Figure 6-31
Partnerlink with two roles

```
<bpel:partnerLink name="Customer"
   partnerLinkType="ns0:partnerLinkNamePLT"
   myRole="provider"
   partnerRole="requester" />
```

Figure 6-32
Partner link type with two roles

```
<plnk:partnerLinkType name="partnerLinkNamePLT">
   <plnk:role name="provider">
      <plnk:portType name="ns1:portTypeName1"/>
   </plnk:role>

   <plnk:role name="requester">
      <plnk:portType name="ns2:portTypeName2"/>
   </plnk:role>
</plnk:partnerLinkType>
```

6.6.3 Compensation

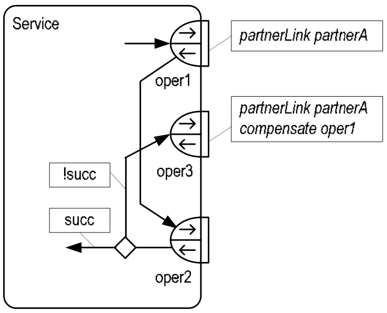
To indicate that an operation call compensates the effect of other operation calls, we define annotation *compensate* as depicted in Table 6-4.

Table 6-4
Annotation to indicate a compensation

Name	Value
compensate	The name of the operation to be compensated.

Figure 6-33 depicts the use of this annotation. Operation call *oper2* is performed after operation call *oper1*. If the result received by the return part of operation *oper2* indicates that the operation call does not succeed, operation call *oper3* should be executed to compensate for any changes made by operation call *oper1*. Operation call *oper3* is hence annotated with annotation *compensate* with value referring to the name of the operation call that has to be compensated, i.e., *oper1*. To refer to the same partner, an operation call and its compensation handler must have the same value for annotation *partnerLink*. This model can be transformed into the BPEL process as depicted in Figure 6-34.

Figure 6-33
Operation call *oper3* compensates operation call *oper1*



Operation call *oper3* is transformed into a compensation handler of operation call *oper1*. A causality relation that enables operation *oper3* is transformed into a BPEL throw activity. A fault name is generated by the transformation. A fault handler is defined to catch this fault. A BPEL compensate activity is defined in the fault handler to handle that fault.

Figure 6-34
Compensation in a BPEL process

```
<bpel:process ... >
   <bpel:faultHandlers>
      <bpel:catch faultName="fault1">
         <bpel:compensate scope="oper1" />
      </bpel:catch>
   </bpel:faultHandlers>

   <bpel:sequence>
```

```
      ...
      <bpel:invoke name="oper1" ... >
         <bpel:compensationHandler>
            <bpel:invoke name="oper3" ... />
         </bpel:compensationHandler>
      </bpel:invoke>

      <bpel:invoke name="oper2" ... />

      <bpel:switch>
         <bpel:case constraint="oper2 is success">
         ...
         </bpel:case>
         <bpel:case constraint="oper2 is not success">
            </bpel:throw name="fault1" />
         </bpel:case>
      </bpel:switch>

   </bpel:sequence>
</bpel:process>
```

6.6.4 Behavioural patterns

The pattern recognition (see Section 6.4) transforms the behavioural patterns in a coordinator model to CBPL structured activities. The pattern realisation, which is a part of the model realisation, transforms these CBPL structured activities to BPEL structured activities. The mapping between CBPL and BPEL structured activities is shown in Table 6-5.

Table 6-5
Mapping between CBPL structured activity and BPEL structured activity

CBPL structured activity	**BPEL structured activity**
Sequence	Sequence
Concurrency	Flow
Iteration - constraint	While - condition
Selection	Switch
Case - constraint	Case of Switch - condition
DefaultCase	Otherwise of Switch

Example

Using the mapping in Table 6-5, the CBPL intermediate model as depicted in Figure 6-13 can be transformed into a BPEL process as depicted in Figure 6-35.

Figure 6-35
Implementation in BPEL

```
<bpel:sequence>
   <bpel:receive name=”receiveApplication” ... />
   <bpel:switch>
      <bpel:case condition=
         ”bpws:getVariableProperty(’app’,’type’)=’individual’”>
         <bpel:sequence>
            <bpel:assign ... />
            <bpel:invoke name=”applyIndividual” ... />
            <bpel:assign ... />
         </bpel:sequence>
      </bpel:case>
      <bpel:case condition=
         ”bpws:getVariableProperty(’app’,’type’)=’collective’”>
         <bpel:sequence>
            <bpel:assign ... />
            <bpel:invoke name=”applyCollective” ... />
            <bpel:assign ... />
         </bpel:sequence>
      </bpel:case>
   </bpel:switch>
   <bpel:reply name=”replyConfirmation” ... />
</bpel:sequence>
```

6.7 Related work

A model that is given as an input to an automatic transformation tool is typically restricted. Modelling restrictions specify model elements that may be used, structures or patterns that may be formed, and annotations that should be added in order to make a transformation produce correct implementations. Such constraints should be explicitly defined as in [22, 23, 35, 63, 64, 125]. Otherwise, models may contain model elements or structures that cannot be transformed. An annotation is necessary if a model element can be mapped onto an alternative implementation construct. A set of constraints and annotations can be defined using a language profiling mechanism [94] as in [9, 35, 71, 92].

Workflow patterns [129] may serve as a set of allowed behavioural patterns in a coordinator model. It offers more patterns and these patterns are behavioural patterns that are frequently used in business process modelling. In this case, the pattern recognition has to identify the workflow patterns that are used in a coordinator model and document them in an intermediate model as a composition of the CBPL behavioural patterns. The mapping between some workflow patterns and the CBPL behavioural patterns has been defined in [36].

Transformations from a service composition model to an implementation can be found in [25, 63, 64, 101, 120]. None of them indicates how the transformations are (de)composed. Each one develops a transformation directly based on a transformation specification.

The need for transformation (de)composition is studied in [67]. The study addresses issues that should be considered in transformation decomposition and composition, such as order of rule execution, tangling and scattering concerns, and additive changes. The study focuses on the development of a transformation language that can handle those issues.

The (de)composition of transformations is also studied in the area of aspect orientation [8, 66, 118]. A transformation is decomposed according to concerns, e.g., logging, security, and transaction. Aspect orientation does not consider the structure and activities of a business process or service composition model as a concern and, hence, does not decompose a transformation according to them.

CBPL can be seen as an abstract platform [4] that offers a large set of alternative implementation languages to realise an intermediate model, such as BPEL, Java, or C/C++.

The UML 2.1.1 StructuredActivities package [96] supports traditional structured programming constructs. It provides the concepts of sequence, conditional, and loop nodes, which are similar to CBPL sequence, selection and repetition patterns, respectively. However, UML has no single concept for representing a concurrency. Furthermore, the CBPL metamodel is simpler than the metamodel of UML StructuredActivities. It better facilitates the development of the transformation tool.

6.8 Concluding remarks

In this chapter, we have explained the development of our ISDL2BPEL transformation tool. This tool automates the transformation of the coordinator model of an orchestration into an executable implementation in BPEL. A coordinator model that will be transformed should comply with several modelling restrictions and be annotated with WSDL/BPEL-specific information.

The ISDL2BPEL transformation tool is developed as a composition of smaller sub-transformation tools, namely: pattern recognition, constraint transformation, and model realisation, that have to be performed sequentially. Intermediate models are used to decouple the sub-transformations. We have defined a language called CBPL (Common Behavioural Pattern Language) to specify those intermediate models. The pattern recognition identifies the behavioural patterns that are used to compose the behaviour structure of a coordinator model. The constraint transformation transforms a function call in a coordinator model into an operation call. This transformation is to accommodate the approach that we select to deal with the BPEL limitation on data manipulation. The model realisation transforms a CBPL model into an executable implementation in BPEL.

The decomposition of the ISDL2BPEL transformation tool into sub-transformations tools opens the possibility of reusing the sub-transformation tools in other transformation tools, as indicated in [4, 36]. This possibility should be further investigated to evaluate whether such reuse is feasible.

We identify some possible improvements for the transformation tool.

- *Incremental transformation*. Currently, the transformation tool does not support incremental transformation. Any modification on the BPEL process that results from the transformation of a coordinator model is not preserved when the coordinator model (possibly with some allowed changes) is transformed again. In an incremental transformation, the modification on the BPEL process is preserved when the coordinator model is transformed again. Hence, the BPEL process reflects the modification that is done directly on it and the change made on the coordinator model.
- *Support for arbitrary behaviour structures*. The BPEL flow activity can be used to support an arbitrary behaviour structure, i.e., a behaviour structure that cannot be mapped onto the sequence, concurrency, selection, and/or repetition patterns. Such a structure can be formed by utilising the BPEL links between the activities that are defined in a BPEL flow activity. Further investigation is necessary to analyse whether the BPEL links can represent ISDL causality relations. If so, the mapping between (types of) arbitrary structures in ISDL and a BPEL flow activity with links can be defined.
- *Support for other types of ISDL causality conditions*. Currently, the transformation tool supports only the enabling and disabling conditions. The disabling condition, however, has to be used in a choice relation. Further investigation can be done on the definition of transformation rules of the disabling and synchronisation condition.

Chapter 7

Case study: travel reservation application

In this chapter, we apply our interaction design concept and transformations to a case study, namely a travel reservation application [134], and evaluate the concept and transformations to assess whether they serve their purposes well and can be used in practice. This case study demonstrates

- a top-down design process of a business collaboration, from an abstract interaction to a concrete interaction structure,
- preservation of interaction synchronisation in a concrete interaction structure, and
- preservation of the atomic property of an abstract interaction, that is implemented as transaction processing in a concrete interaction structure.

In this case study, the services to be composed do not exist yet.

This chapter is organised as follows: Section 7.1 presents the case description for this case study. Sections 7.2 and 7.3 present alternative design processes to design a travel reservation application. Section 7.4 discusses the design processes. Finally, Section 7.5 evaluates the interaction concept and design transformations.

7.1 Case description

A travel agent wants to offer its customers the ability to compose and book a vacation package. A vacation package consists of a return flight and hotel reservation. The flight and hotel reservations are performed with a flight and hotel reservation system, respectively. The payment of a reservation is done using a credit card system provided by a credit card company. The steps for booking a vacation package are as follows.

– *Select flight*
  A customer selects a return flight to his destination on preferred dates, i.e., the dates of outward and inward flights. The detailed description of this step is as follows. The customer provides a destination and dates to the travel agent. The travel agent queries the flight reservation system about the available flights; presents the customer a list of the available flights returned by the flight reservation system; and lets the customer select a return flight that is suitable for him. The travel agent then puts the selected flight on hold in the flight reservation system. The flight reservation system returns a confirmation with an expiration date. A held flight can only be booked before its expiration date.
– *Select hotel*
  The customer selects a hotel at his destination for staying a couple of days as indicated by the dates of his flight. It is assumed that the customer checks-in to the hotel on the date of his outward flight; and checks-out from the hotel on the date of his inward flight. Using information from the previous step, the travel agent finds a list of the available hotels from the hotel reservation system; presents the list to the customer; and lets the customer select his preferred hotel.
– *Book the composed vacation package*
  The customer books the composed vacation package. The customer provides his credit card information. The travel agent contacts the credit card system to request an authorisation that guarantees the payment of the total amount of the price of the composed vacation package. The credit card system indicates a successful authorisation with an authorisation identifier. The travel agent books the selected hotel with the authorisation identifier. The travel agent then confirms the selected flight with the authorisation identifier. The travel agent charges the customer a reservation fee and provides the customer with the booking codes of the hotel and flight. If the flight booking cannot be confirmed by the flight reservation system, the hotel booking should be cancelled. Any payment that might have been made should be paid back to the customer.

A vacation package consists of information about outward flight, inward flight, and hotel. Flight information consists of *flight code*, *departure place*, *destination place*, *date of flight*, and *price*. Availability information of a hotel includes *hotel name*, *hotel location*, *check-in* and *check-out dates*, and *price*. Figure 7-1 depicts these information types. We assume that function *getXyz(abc)* is available to return the value of attribute *xyz* of information *abc*. For example, *getDeparture(flight)* returns the value of attribute *departure* of *flight* information.

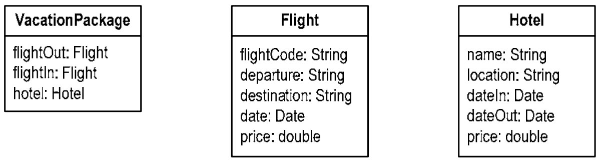


Figure 7-1
Information types

The original description of this case includes technological requirements, such as description language, ontologies, discovery technology, authentication technology, and encryption technology. These technological requirements are outside the scope of our case study.

Design approach
The case description provides a detailed scenario at an implementation level. Our interaction design concept and transformations are meant to enable and encourage interaction design at related abstraction levels. Thus, we carry out the case study by
– identifying the essential requirements for a travel reservation,
– modelling an abstract interaction that satisfies those essential requirements, and
– performing a top-down design process to develop an implementation as described above. Information items in the case description that are not identified as essential requirements are considered as implementation requirements.

We present two alternative design processes, which differ in the identified essential requirements. Different essential requirements result in different abstract interactions that serve as starting points for design processes. After presenting the design processes, we discuss our observations about those design processes.

7.2 Design process 1

In this section, we design a business collaboration between two essential entities. Non-essential entities are introduced during the design process.

7.2.1 Essential requirements

Two essential entities for making a reservation of a vacation package are identified: a *customer* and *travel agent*. The flight and hotel reservation systems are supporting entities that are used by the travel agent in order to deliver its service. The credit card system is not an essential entity because the payment can be done using other payment methods, e.g., cash, debit cards, or other online payments such as in [102].

The customer wants to book a vacation package consisting of a return flight to a specific destination and a stay in a hotel in that destination for a couple of days. To book the flight, the departure place should be known. The customer is willing to pay the price of the vacation package. The travel agent provides the flight and hotel reservation as a vacation package. The travel agent charges the customer the price of the vacation package plus a reservation fee. The travel agent identifies a customer by the customer's name. A vacation package is provided as a complete package of a return flight and hotel.

7.2.2 Abstract interaction

At a higher abstraction level, we model the collaboration between the customer and travel agent as an abstract interaction as depicted in Figure 7-2. For brevity, attribute types and constraints are omitted. Names are used to represent information attributes, instead of indexed information attributes, e.g., ι_1 and ι_2. The complete specification of this interaction design is textually expressed in Figure 7-3. Context actions are included to allow us to consider the dependency that might exist between the interaction and causality context. A vacation package is represented as information attributes *flightOut*, *flightIn*, and *hotel*.

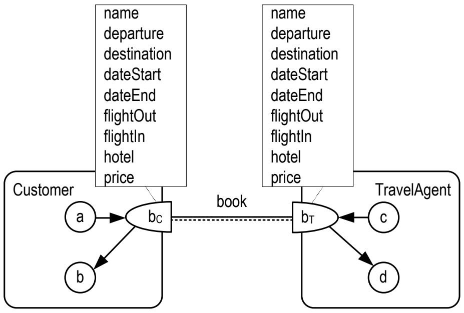


Figure 7-2
Abstract interaction between the customer and travel agent

Figure 7-3
Textual expression of the abstract interaction in Figure 7-2

```
Customer = {
    a → b_C (name: String, departure: String, destination: String, dateStart: Date, dateEnd: Date,
    flightOut: Flight, flightIn: Flight, hotel: Hotel, price: double)
        [departure = getDeparture(flightOut) = getDestination(flightIn),
        destination = getDestination(flightOut) = getDeparture(flightIn),
        destination = getLocation(hotel),
        dateStart  = getDate(flightOut) = getDateIn(hotel),
        dateEnd = getDate(flightIn) = getDateOut(hotel)],

    b_C → b
}
```

```
TravelAgent = {
    c → b_T (name: String, departure: String, destination: String, dateStart: Date, dateEnd: Date,
    flightOut: Flight, flightIn: Flight, hotel: Hotel, price: double)
        [flightOut in listFlights(departure, destination, dateStart),
        flightIn in listFlights(destination, departure, dateEnd),
        hotel in listHotels(destination, dateStart, dateEnd),
        price = getPrice(flightOut) + getPrice(flightIn) + getPrice(hotel) + fee],

    b_T → d
}
```

book (b_C: Customer.b_C, b_T: TravelAgent.b_T) [remote]

Interaction *book* is modelled as a remote interaction, because it should establish the same set of information values that can be available from different time moments and at different locations for different participants.

In the travel agent, function *listFlights(departure, destination, date)* returns a list of the available flights from a departure place to a destination place on a specific date. Function *listHotels(location, check-in, check-out)* returns a list of the available hotels in a specific location between check-in and check-out dates.

The contribution constraints of interaction contribution b_C of the customer specifies that

- C1: the customer's departure place is the departure place of the outward flight and is the same as the destination place of the inward flight
  [departure = getDeparture(flightOut) = getDestination (flightIn)];
- C2: the customer's destination place is the destination place of the outward flight and is the same as the departure place of the inward flight
  [destination = getDestination(flightOut) = getDeparture (flightIn)];
- C3: the hotel is located in the customer's destination place
  [destination = getLocation(hotel)];
- C4: the customer's vacation trip starts on the date of the outward flight and is the same as the check-in date to the hotel
  [dateStart = getDate(flightOut) = getDateIn(hotel)]; and
- C5: the customer's vacation trip ends on the date of the inward flight and is the same as the check-out date from the hotel
  [dateEnd = getDate(flightIn) = getDateOut(hotel)].

The contribution constraints of interaction contribution b_T of the travel agent specifies that

- T1: the outward flight should be in the list of the available flights from a departure place to a destination place on a specific date.
  [flightOut in listFlights(departure, destination, dateStart)];
- T2: the inward flight should be in the list of the available flights from a departure place to a destination place on a specific date.
  [flightIn in listFlights(destination, departure, dateEnd)];
- T3: the hotel should be in the list of the available hotels between check-in and check-out dates in a specific location
  [hotel in listHotels(destination, dateStart, dateEnd)]; and
- T4: the price charged to the customer is the sum of the prices of the outward flight, inward flight, hotel reservation, and a reservation fee
  [price = getPrice(flightOut) + getPrice(flightIn) + getPrice(hotel) + fee].

7.2.3 Refinement 1 (choreography)

We refine abstract interaction *book* into a concrete interaction structure as depicted in Figure 7-4. For brevity, attributes and constraints are omitted. The refinement is done by applying the interface decomposition pattern (see Section 4.7.1). This concrete interaction structure models the choreography between the customer and travel agent. The interactions in this model are described in Table 7-1. The complete specification of this intraction design is textually expressed in Figure 7-5.

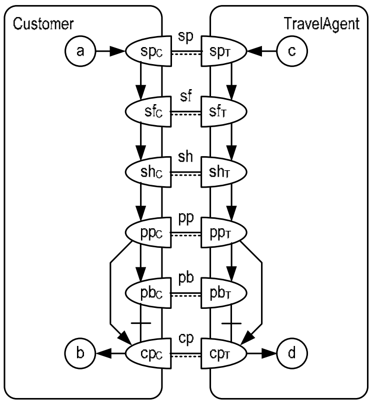


Figure 7-4
Choreography between the customer and travel agent

Table 7-1
Descriptions of interactions in Figure 7-4

Interaction	Description
sp	Start to compose a vacation package
sf	Select a return flight (i.e., outward and inward flights)
sh	Select a hotel
pp	Pay a composed vacation package
pb	Payback the payment for a vacation package
cp	Confirm the booking of a vacation package

Figure 7-5
Textual expression of the choreography in Figure 7-4

Customer = {

$a \rightarrow sp_C$ (departure: String, destination: String, dateStart: Date, dateEnd: Date),

$sp_C \rightarrow sf_C$ (flightOut: Flight, flightIn: Flight)
[getDeparture(flightOut) = getDestination(flightIn) = sp_C.departure,
getDestination(flightOut) = getDeparture(flightIn) = sp_C.destination,
getDate(flightOut) = sp_C.dateStart,
getDate(flightIn) = sp_C.dateEnd],

$sf_C \rightarrow sh_C$ (hotel: Hotel)
[getLocation(hotel) = sp_C.destination,
getDateIn(hotel) = sp_C.dateStart,
getDateOut(hotel) = sp_C.dateEnd],

$sh_C \rightarrow pp_C$ (name: String, price: double),

$pp_C \wedge \neg cp_C \rightarrow pb_C$ (payback: double)
[payback = pp_c.price],

$pp_C \wedge \neg pb_C \rightarrow cp_C$ (code: long[2]),

$cp_C \rightarrow b$

}

TravelAgent = {

$c \rightarrow sp_T$ (departure: String, destination: String, dateStart: Date, dateEnd: Date),

$sp_T \rightarrow sf_T$ (flightOut: Flight, flightIn: Flight)
[flightOut in listFlights(sp_T.departure, sp_T.destination, sp_T.dateStart),
flightIn in listFlights(sp_T.destination, sp_T.departure, sp_T.dateEnd)]

$sf_T \rightarrow sh_T$ (hotel: Hotel)
[hotel in listHotels(sp_T.destination, sp_T.dateStart, sp_T.dateEnd)],

```
    sh_T → pp_T (name: String, price: double)
        [price = getPrice(sf_T.flightOut) + getPrice(sf_T.flightIn) + getPrice(sh_T.hotel) + fee)],

    pp_T ∧ ¬cp_T → pb_T (payback: double)
        [payback = pp_T.price],

    pp_T ∧ ¬pb_T → cp_T (code: long[2])
        [code[0] = getCode(pp_T.name, sf_T.flightOut, sf_T.flightIn),
        code[1] =  getCode(pp_T.name, sh_T.hotel)],

    cp_T → d
}

sp (sp_C: Customer.sp_C, sp_T: TravelAgent.sp_T) [remote]
sf (sf_C: Customer.sf_C, sf_T: TravelAgent.sf_T) [remote]
sh (sh_C: Customer.sh_C, sh_T: TravelAgent.sh_T) [remote]
pp (pp_C: Customer.pp_C, pp_T: TravelAgent.pp_T) [remote]
cp (cp_C: Customer.cp_C, cp_T: TravelAgent.cp_T) [remote]
```

This choreography models the steps in a travel reservation. Interaction *sp* lets the travel agent know about the customer's preference. To book a vacation package, three interactions are used, i.e., interactions *pp, pb,* and *cp*. After the occurrence of interaction *pp* (i.e., pay a composed vacation package), either interaction *pb* (i.e., a complete vacation package cannot booked and the money is paid back to the customer) or interaction *cp* (i.e., the booking is successfully confirmed) occurs. The occurrences of interaction *cp* and *pb* represents the occurrence and non-occurrence of abstract interaction *book*, respectively.

All conformance assessment that is done for this case study is provided in Appendix A.

7.2.4 Refinement 2 (orchestration)

We refine the travel agent by introducing the supporting entities, i.e., the flight and hotel reservation systems as depicted in Figure 7-6. It models the travel agent as an orchestration between a travel agent coordinator (*TACoordinator*), flight reservation system (*FlightRS*) and hotel reservation system (*HotelRS*). For brevity, attributes and constraints are omitted. The interactions in this model are described in Table 7-2. The interactions between the customer and travel coordinator are the same as the interactions described in Table 7-1. The complete specification of this interaction design is textually expressed in Figure 7-7.

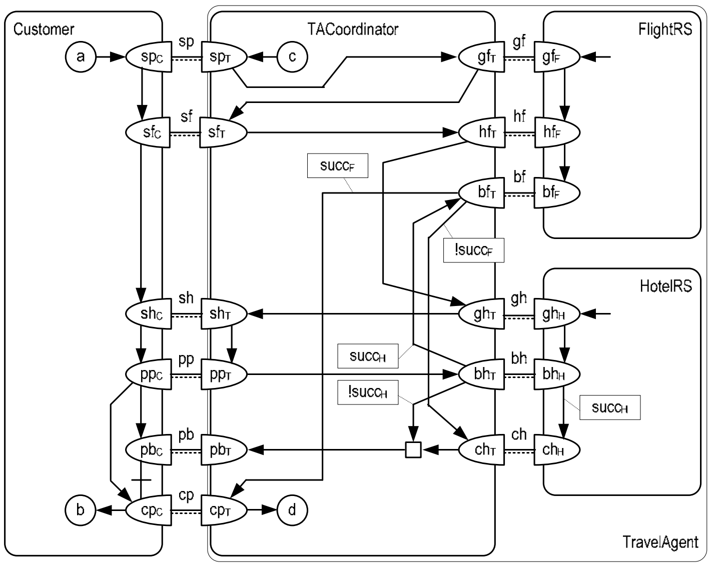


Figure 7-6
The travel agent as an orchestration

Table 7-2
Descriptions of interactions in Figure 7-6

Interaction	Participants	Description
cp	Customer – TACoordinator	Confirm a vacation package
gf	TACoordinator – FlightRS	Get the available flights
hf	TACoordinator – FlightRS	Hold a flight
bf	TACoordinator – FlightRS	Book a flight
gh	TACoordinator – HotelRS	Get the available hotels
bh	TACoordinator – HotelRS	Book a hotel
ch	TACoordinator – HotelRS	Cancel a hotel reservation

Figure 7-7
Textual expression of the orchestration in Figure 7-6

Customer = {
    …   ; the same behaviour as in Figure 7-5
}

FlightRS = {
    $\surd \rightarrow gf_F$ (departure: String, destination: String, dateOut: Date, dateIn: Date, flightsOut: Flight[], flightsIn: Flight[])
        [flightsOut = listFlights(departure, destination, dateOut),
        flightsIn = listFlights(destination, departure, dateIn)],

    $gf_F \rightarrow hf_F$ (flightOut: Flight, flightIn: Flight, expiryDate: Date)
        [flightOut in gf_F.flightsOut,
        flightIn in gf_F.flightsIn,
        expiryDate = getExpiryDate(currentDate())],

```
    $hf_F \rightarrow bf_F$ (name: String, flightOut: Flight, flightIn: Flight, price: double, code: long)
        [flightOut = $hf_F$.flightOut,
        flightIn = $hf_F$.flightIn,
        price = getPrice(flightOut) + getPrice(flightIn),
        code = (if currentDate() < $hf_F$.expiryDate
            then getCode(name, flightOut, flightIn) else -1)]
}

HotelRS = {
    $\surd \rightarrow gh_H$ (location: String, dateIn: Date, dateOut: Date, hotels: Hotel[])
        [hotels = listHotels(location, dateIn, dateOut)],

    $gh_H \rightarrow bh_H$ (name: String, hotel: Hotel, price: double, code: long)
        [hotel in $gh_H$.hotels,
        price = getPrice(hotel),
        code = getCode(name, hotel)],

    $bh_H$ [$bh_H$.code $\geq$ 0] $\rightarrow ch_H$ (code: long)
}

TACoordinator = {
    $c \rightarrow sp_T$ (departure: String, destination: String, dateStart: Date, dateEnd: Date),

    $sp_T \rightarrow gf_T$ (departure: String, destination: String, dateOut: Date, dateIn: Date, flightsOut:
    Flight[], flightsIn: Flight[])
        [departure = $sp_T$.departure,
        destination = $sp_T$.destination,
        dateOut = $sp_T$.dateStart,
        dateIn = $sp_T$.dateEnd],

    $gf_T \rightarrow sf_T$ (flightOut: Flight, flightIn: Flight)
        [flightOut in $gf_T$.flightsOut,
        flightIn in $gf_T$.flightsIn]

    $sf_T \rightarrow hf_T$ (flightOut: Flight, flightIn: Flight, expiryDate: Date)
        [flightOut = $sf_T$.flightOut,
        flightIn = $sf_T$.flightIn],

    $hf_T \rightarrow gh_T$ (location: String, dateIn: Date, dateOut: Date, hotels: Hotel[])
        [location = $sp_T$.destination,
        dateIn = $sp_T$.dateStart,
        dateOut = $sp_T$.dateEnd],
    gh_T → sh_T (hotel: Hotel)
        [hotel in gh_T.hotels],

    sh_T → pp_T (name: String, price: double)
        [price = getPrice(sf_T.flightOut) + getPrice(sf_T.flightIn) + getPrice(sh_T.hotel) + fee],

    pp_T → bh_T (name: String, hotel: Hotel, price: double, code: long)
        [name = pp_T.name,
        hotel = sh_T.hotel
        price = getPrice(hotel)],

    bh_T [bh_T.code ≥ 0] → bf_T (name: String, flightOut: flight, flightIn: Flight, price: double, code:
    long)
        [name = pp_T.name,
        flightOut = hf_T.flightOut,
        flightIn = hf_T.flightIn,
        price = getPrice(flightOut) + getPrice(flightIn)],

    bf_T [bf_T.code < 0] → ch_T (code: long)
        [code = bh_T.code],

    bh_T [bh_T.code < 0] ∨ ch_T  → pb_T (payback: double)
        [payback = pp_T.price],

    bf_T [bf_T.code ≥ 0] → cp_T (code: long[2])
        [code[0] = bf_T.code,
        code[1] = bh_T.code],

    cp_T → d
}

sp (sp_C: Customer.sp_C, sp_T: TACoordinator.sp_T) [remote]
sf (sf_C: Customer.sf_C, sf_T: TACoordinator.sf_T) [remote]
sh (sh_C: Customer.sh_C, sh_T: TACoordinator.sh_T) [remote]
pp (pp_C: Customer.pp_C, pp_T: TACoordinator.pp_T) [remote]
pb (pb_C: Customer.pb_C, pb_T: TACoordinator.pb_T) [remote]
cp (cp_C: Customer.cp_C, cp_T: TACoordinator.cp_T) [remote]

gf (gf_T: TACoordinator.gf_T, gf_F: FlightRS.gf_F) [remote]
hf (hf_T: TACoordinator.hf_T, hf_F: FlightRS.hf_F) [remote]
bf (bf_T: TACoordinator.bf_T, bf_F: FlightRS.sf_F) [remote]

gh (gh_T: TACoordinator.gh_T, gh_H: HotelRS.gh_H) [remote]
```

```
bh (bh_T: TACoordinator.bh_T, bh_H: HotelRS.bh_H) [remote]
ch (ch_T: TACoordinator.ch_T, ch_H: HotelRS.ch_H) [remote]
```

In Figure 7-6, causality constraints are defined as [succ_H], [$!\text{succ}_H$], [succ_F], and [$!\text{succ}_F$]. They represent successful hotel booking [$\text{bh}_T.\text{code} > 0$], unsuccessful hotel booking [$\text{bh}_T.\text{code} < 0$], successful flight booking [$\text{bf}_T.\text{code} \geq 0$], and unsuccessful flight booking [$\text{bf}_T.\text{code} < 0$], respectively. The occurrence of a booking interaction results in either a successful or an unsuccessful booking.

The choreography between the customer and travel agent is preserved by the customer and coordinator. The causality relations between the interaction contributions of the travel agent are refined. For example, the causality relation between interaction contributions sp_T and sf_T in Figure 7-4 is refined by inserting interaction *gf*. It should be emphasised here that refinement of the choreography into the orchestration is not interaction refinement, because no interaction in the choreography is refined into a concrete interaction structure.

7.2.5 Refinement 3 (choreography of booking hotel interaction)

We refine the interactions in the orchestration by focusing on the hotel booking interaction *bh* of Figure 7-6. By applying the interface decomposition pattern, this interaction is refined into a concrete interaction structure that consists of interactions *request*, *payment*, and, *confirm*, as depicted in Figure 7-8. For brevity, the figure shows only the concrete interaction structure and context interaction contributions; and omits attribute types. This concrete interaction structure models the choreography between the coordinator and hotel reservation system. Context interaction contributions are coloured in grey. Interaction *request* models a request to book a selected hotel. Interaction *payment* models the payment of the selected hotel. Interaction *confirm* models the confirmation of the hotel booking. The specification of this model is textually expressed in Figure 7-9.

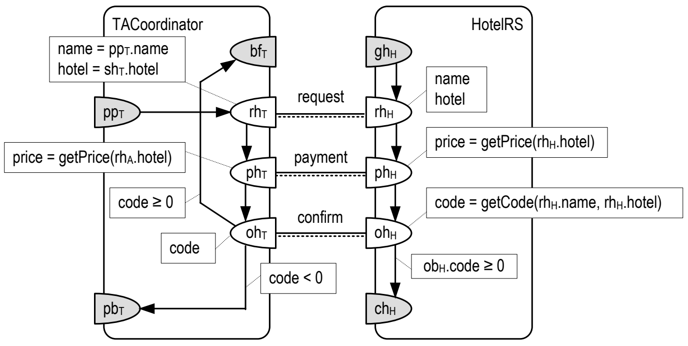


Figure 7-8
Choreography between the coordinator and hotel reservation system for booking a hotel

Figure 7-9
Excerpt of textual expression of the choreography in Figure 7-8

```
TACoordinator = {
    …
    $pp_T \rightarrow rh_T$ (name: String, hotel: Hotel)
        [name = $pp_T$.name,
        hotel = $sh_T$.hotel],

    $rh_T \rightarrow ph_T$ (price: double)
        [price = getPrice($rh_T$.hotel)],

    $ph_T \rightarrow oh_T$ (code: long),

    $oh_T$ [$oh_T$.code $\geq$ 0] $\rightarrow bf_T$,

    $oh_T$ [$oh_T$.code $<$ 0] $\rightarrow pb_T$
    …
}

HotelRS = {
    …
    $gh_H \rightarrow rh_H$ (name: String, hotel: Hotel)
        [hotel in $gh_H$.hotels],

    $rh_H \rightarrow ph_H$ (price: double)
        [price = getPrice($rh_H$.hotel)],

    $ph_H \rightarrow oh_H$ (code: long)
        [code = getCode($rh_H$.name, $rh_H$.hotel)],

    $oh_H$ [$oh_H$.code $\geq$ 0] $\rightarrow ch_H$
    …
}
```

request (rh_T: TACoordinator.rh_T, rh_H: HotelRS.rh_H) [remote]
payment (ph_T: TACoordinator.ph_T, ph_H: HotelRS.ph_H) [remote]
confirm (oh_T: TACoordinator.oh_T, oh_H: HotelRS.oh_H) [remote]

7.2.6 Refinement 4

We now include a solution for an implementation requirement regarding payment: *"The payment of a reservation is done using a credit card system provided by a credit card company"*. We focus on the payment for the hotel booking in Figure 7-8. We refine the coordinator by inserting interaction *auth*, as depicted in Figure 7-10. Its specification is textually expressed in Figure 7-11.

The customer should provide his credit card information. This credit card information, together with its authorisation identifier, is used to pay the booking of the selected hotel. For this purpose, in interaction *pp*, we introduce information attribute *ccNo* to represent credit card information. In interaction *payment*, we introduce information attributes *ccNo* and *authID* to represent credit card information and authorisation identifier, respectively. In addition, in interaction *confirm,* we introduce information attribute *rcpt* to represent the receipt of payment.

Figure 7-10
The credit card system is included in the orchestration

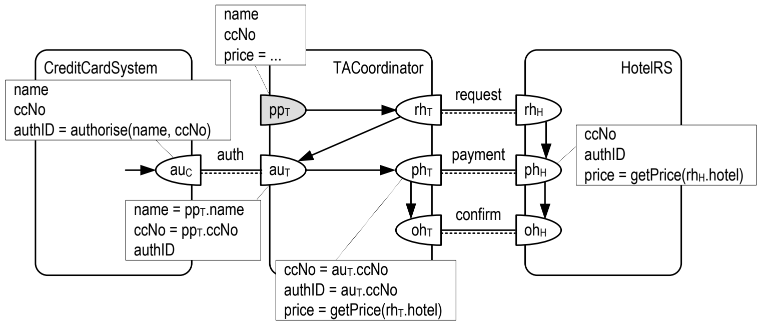


Figure 7-11
Excerpt of textual expression of the orchestration in Figure 7-10

TACoordinator = {
…
$rh_T \rightarrow au_T$ (name: String, ccNo: String, authID: long)
[name = pp_T.name,
ccNo = pp_T.ccNo],

$au_T \rightarrow ph_T$ (ccNo: String, authID: long, price: double)
[ccNo = au_T.ccNo,
authID = au_T.authID,
price = getPrice(rh_T.hotel)],

```
    $ph_T \rightarrow oh_T$ (code: long, rcpt: String),
    ...
}

HotelRS = {
    ...
    $rh_H \rightarrow ph_H$ (ccNo: String, authID: long, price: double)
        [price = getPrice($rh_H$.hotel)],

    $pa_H \rightarrow oh_H$ (code: long, rcpt: String)
        [code = getCode($rh_H$.name, $rh_H$.hotel),
        rcpt = makeReceipt($ph_H$.ccNo, $ph_H$.price),
    ...
}

CreditCardSystem = {
    $\surd \rightarrow au_C$(name: String, ccNo: String, authID: long)
        [authID = authorise(name, ccNo)]
}

payment ($ph_T$: TACoordinator.$ph_T$, $ph_H$: HotelRS.$ph_H$) [remote]
auth ($au_C$: CreditCardSystem.$au_C$, $au_T$: TACoordinator.$au_T$) [remote]
```

Alternative implementations for payment interaction

By modeling payment as an abstract interaction, i.e., interaction *pp* between the customer and coordinator in Figure 7-6 and interaction *payment* between the coordinator and hotel reservation system in Figure 7-8, we have a number of alternative implementations. As mentioned in Section 7.2.1, the payment can be done using other payment methods, e.g., money transfer via a bank or online payment.

Different payment methods can be combined in a reservation of a vacation package. For example, the payment interaction between the customer and coordinator can be done using money transfer via a bank, while the payment interaction between the coordinator and flight or hotel reservation system can be done using credit card.

7.3 Design process 2

In this section, we design a business collaboration between three essential entities. An intermediary entity is introduced during the design process. This design process is an alternative to design process 1.

7.3.1 Essential requirements

Three essential entities are identified: *a customer*, *a flight reservation system*, and *a hotel reservation system*. The travel agent and credit card system are considered non-essential entities. In real life, a customer can book a return flight and hotel to compose a vacation package without a travel agent. The credit card system is not an essential entity because the payment can be done using other payment methods.

The customer wants to book a vacation package for a couple of days at a destination. The vacation days are identified by the date he starts and ends his vacation trip. To plan his trip, his departure place should be known. The customer is willing to pay the price of his vacation package and a reservation fee, if any. The flight and hotel reservation systems provide a return flight and hotel reservation, respectively. The customer is identified by his name.

In a vacation package, both flight and hotel must all be successfully booked. If one of them cannot be booked, the other must not be booked either.

7.3.2 Abstract interaction

At a higher abstraction level, we model the collaboration between the essential entities as an abstract multilateral interaction, as depicted in Figure 7-12. The 'all or none' characteristic of the flight and hotel bookings is represented by the atomic property of the abstract interaction. For brevity, attribute types are omitted. The complete specification of this interaction design is textually expressed in Figure 7-13. Context actions are included to allow us to consider the dependency that might exist between the interaction and its causality context. A vacation package is represented as information attributes *flightOut*, *flightIn*, and *hotel*.

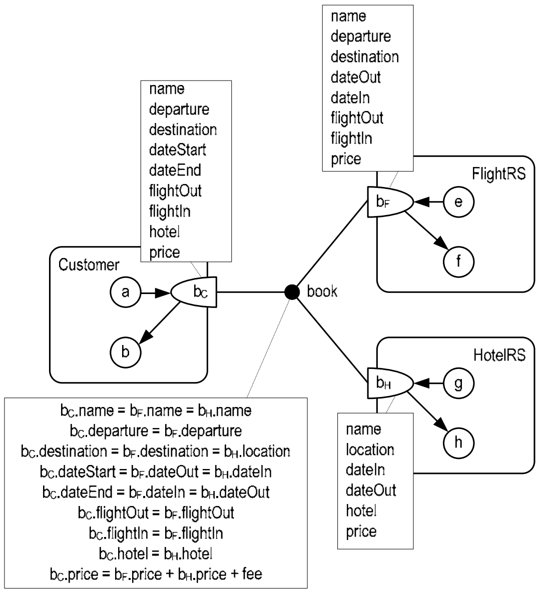

name
departure
destination
dateStart
dateEnd
flightOut
flightIn
hotel
price
Customer
a
b_C
b
book
name
departure
destination
dateOut
dateIn
flightOut
flightIn
price
FlightRS
b_F
e
f
HotelRS
b_H
g
h
name
location
dateIn
dateOut
hotel
price
b_C.name = b_F.name = b_H.name
b_C.departure = b_F.departure
b_C.destination = b_F.destination = b_H.location
b_C.dateStart = b_F.dateOut = b_H.dateIn
b_C.dateEnd = b_F.dateIn = b_H.dateOut
b_C.flightOut = b_F.flightOut
b_C.flightIn = b_F.flightIn
b_C.hotel = b_H.hotel
b_C.price = b_F.price + b_H.price + fee

Figure 7-12
An abstract multilateral interaction between a customer, flight, and hotel reservation system

Figure 7-13
Textual expression of interacting behaviours in Figure 7-12

Customer = {
  a → b_C (name: String, departure: String, destination: String, dateStart: Date, dateEnd: Date, flightOut: Flight, flightIn: Flight, hotel: Hotel, price: double)
    [departure = getDeparture(flightOut) = getDestination(flightIn),
    destination = getDestination(flightOut) = getDeparture(flightIn),
    destination = getLocation(hotel),
    dateStart = getDate(flightOut) = getDateIn(hotel),
    dateEnd = getDate(flightIn) = getDateOut(hotel)],

  b_C → b
}

FlightRS= {
  e → b_F (name:String, departure: String, destination: String, dateOut: Date, dateIn: Date, flightOut: Flight, flightIn: Flight, price: double)
    [flightOut in listFlights(departure, destination, dateOut),
    flightIn in listFlights(destination, departure, dateIn),
    price = getPrice(flightOut) + getPrice(flightIn)],

  b_F → f
}

HotelRS = {

```
    g → b_H (name: String, location: String, dateIn: Date, dateOut: Date, hotel: Hotel, price:
    double)
        [hotel in listHotels(location, dateIn, dateOut),
        price = getPrice(hotel)],

    b_H → h
}

book (b_C: Customer.b_C, b_F: FlightRS.b_F, b_H: HotelRS.b_H)
    [b_C.name = b_F.name = b_H.name,
    b_C.departure = b_F.departure,
    b_C.destination = b_F.destination = b_H.location,
    b_C.dateStart = b_F.dateOut = b_H.dateIn,
    b_C.dateEnd = b_F.dateIn = b_H.dateOut,
    b_C.flightOut = b_F.flightOut,
    b_C.flightIn = b_F.flightIn,
    b_C.hotel = b_H.hotel,
    b_C.price = b_F.price + b_H.price + fee]
```

The customer's behaviour is the same as the customer's behaviour in design process 1 (see Figure 7-2 and Figure 7-3). The contribution constraints of interaction contribution b_F of the flight reservation system specifies that

- F1: the outward flight should be in the list of the available flights from a departure place to a destination place on a specific date. [flightOut in listFlights(departure, destination, dateStart)];
- F2: the inward flight should be in the list of the available flights from a departure place to a destination place on a specific date. [flightIn in listFlights(destination, departure, dateEnd)]; and
- F3: the price is the sum of the prices of the outward and inward flights [price = getPrice(flightOut) + getPrice(flightIn)].

The contribution constraints of interaction contribution b_H of the hotel reservation system specifies that

- H1: the hotel should be in the list of the available hotels between check-in and check-out dates in a specific location [hotel in listHotels(location, dateIn, dateOut)]; and
- H2: the price is the price of the hotel reservation [price = getPrice(hotel)].

The distribution constraints of interaction *book* specifies that

- D1: the customer that flies with the flights is the customer that stays in the hotel

[b_C.name = b_F.name = b_H.name];

- D2: the customer's departure place is the departure place of the outward flight
  [b_C.departure = b_F.departure];
- D3: the customer's destination place is the destination place of his outward flight and is the same as the location of the hotel
  [b_C.departure = b_F.destination = b_H.location];
- D4: the customer's vacation trip starts on the date of the outward flight and is the same as the check-in date to the hotel
  [b_C.dateStart = b_F.dateOut = b_H.dateIn];
- D5: the customer's vacation trip ends on the date of the inward flight and is the same as the check-out date from the hotel
  [b_C.dateEnd = b_F.dateIn = b_H.dateOut];
- D6: the outward flight reserved by the customer is the outward flight provided by the flight reservation system
  [b_C.flightOut = b_F.flightOut];
- D7: the inward flight reserved by the customer is the inward flight provided by the flight reservation system
  [b_C.flightIn = b_F.flightIn];
- D8: the hotel reserved by the customer is the hotel provided by the hotel reservation system
  [b_C.hotel = b_H.hotel]; and
- D9: the price charged to the customer is the sum of the prices demanded by the flight and hotel reservation systems and a reservation fee
  [b_C.price = b_F.price + b_H.price + fee].

7.3.3 Concrete interaction structure

We refine abstract interaction *book* by applying the intermediary introduction pattern (see Section 4.7.4). A *travel agent* is introduced as an intermediary between the customer, flight and hotel reservation system, as depicted in Figure 7-14. The behaviour of the travel agent defines the business logic of the collaboration. For brevity, attributes and constraints are omitted.

From the customer's view, the occurrences of interactions *cp* and *pb* represent the occurrence and non-occurrence of abstract interaction *book*, respectively. From the flight reservation system's view, the successful flight booking in interaction *bf* represents the occurrence of abstract interaction *book*. The unsuccessful flight booking or the non-occurrence of interaction *bf* represents the non-occurrence of abstract interaction *book*. From the hotel reservation system's view, the successful hotel booking in interaction *bh* and the non-occurrence of interaction *ch* represent the occurrence of abstract interaction *book*. The successful hotel booking that is followed by the occurrence of interaction *ch*, the unsuccessful hotel booking, or the non-occurrence of interaction *bh* represents the non-occurrence of abstract interaction *book*.

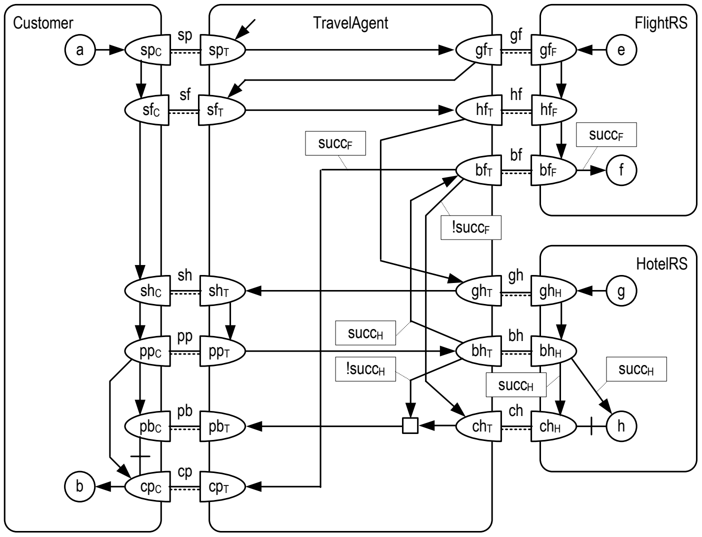


Figure 7-14
The travel agent as an intermediary between the customer, flight and hotel reservation systems

Transaction processing

To preserve the atomic property of abstract interaction *book*, the travel agent implements transaction processing with compensation [47, 99] for the flight and hotel booking. For modelling the transaction processing, we consider not only the occurrence and non-occurrence of an interaction, but also two possible outcomes of the occurrence of an interaction: a *positive* and a *negative result*. A positive result is the intended result of an interaction. A negative result is an anticipated but unintended result of an interaction. For example, the positive result of interaction *bf* represents a successful flight booking [succ_F]. The negative result represents an unsuccessful flight booking [$!\text{succ}_F$].

Compensation is used to cancel or reverse the effects of a completed interaction when another interaction establishes a negative result or does not occur. Compensation is application specific [99]. In Figure 7-14, interaction *ch* is to compensate interaction *bh*, when interaction *bf* establishes a negative result.

We define two requirements for transaction processing with compensation that preserves the atomic property of an abstract interaction.

- *TR1*: If a final interaction occurs with a positive result and is not cancelled; all other final interactions should occur with positive results and not be cancelled.
- *TR2*: If a final interaction occurs with a negative result, occurs with a positive result but is cancelled, or does not occur; every other final interaction should occur with a negative result, occur with a positive result but be cancelled, or not occur.

We check whether the model in Figure 7-14 satisfies requirements TR1 and TR2. We determine the following final interaction contributions:
- in the customer, interaction contribution cp_C,
- in the flight reservation system, interaction contribution bf_F, and
- in the hotel reservation system, interaction contribution bh_H.

Interactions *cp*, *bf*, and *bh* are hence the final interactions.

Requirement TR1:
- If interaction *cp* occurs, interactions *bf* and *bh* have occurred with positive results.
- If interaction *bf* occurs with a positive result, interaction *bh* has occurred with a positive result and interaction *cp* is enabled to occur.
- If interaction *bh* occurs with a positive result and is not cancelled, interaction *bf* has occurred with a positive result and interaction *cp* is enabled to occur.

Requirement TR1 is satisfied.

Requirement TR2:
- If interaction *cp* does not occur, interaction *bf* has occurred with a negative result or does not occur; or interacton *bh* has occurred with a negative result, has occurred with a positive result but has been cancelled, or does not occur.
- If interaction *bf* occurs with a negative result, interaction *bh* has occurred but will be cancelled. Interaction *cp* will not occur.
- If interaction *bf* does not occur, interaction *bh* has occurred with a negative result or does not occur. Interaction *cp* will not occur.
- If interaction *bh* occurs with a negative result, interactions *bf* and *cp* will not occur.
- If interaction *bh* occurs with a positive result but is cancelled, interaction *bf* has occurred with a negative result. Interaction *cp* will not occur.
- If interaction *bh* does not occur, interactions *bf* and *cp* will not occur.

Requirement TR2 is satisfied.

Both requirements TR1 and TR2 are satisfied. We conclude that the atomic property of abstract interaction *book* is preserved in the concrete interaction structure in Figure 7-14.

Interaction synchronisation

We check whether the concrete interaction structure in Figure 7-14 provides synchronisation as provided by abstract interaction *book* in Figure 7-12, i.e., whether conformance requirement *IR4* holds.

Final interaction *cp* depends on

- context action *a* via interactions *pp*, *sh*, *sf*, and *sp*;
- context action *e* via interactions *bf*, *hf*, and *gf*; and
- context action *g* via interactions *bf*, *bh*, and *gh*.

Final interaction *bf* depends on

- context action *a* via interactions *hf*, *gf*, and *sp*;
- context action *e* via interactions *hf* and *gf*; and
- context action *g* via interactions *bh* and *gh*.

Final interaction *bh* depends on

- context action *a* via interactions *pp*, *sh*, *sf*, and *sp*;
- context action *e* via interactions *gh*, *hf*, and *gf*; and
- context action *g* via interactions *gh*.

Every final interaction depends on the same context actions, i.e., concrete actions *a*, *e*, and g. We conclude that the concrete interaction structure provides synchronisation as provided by abstract interaction *book*. The complete conformance assessment of this concrete interaction structure is provided in Appendix A.

7.4 Discussion

In this section, we discuss issues regarding the design processes.

Essential requirements and design processes

Design processes 1 and 2 show that different choices of essential requirements leads to different design processes. In interaction design, the identification of essential entities must be done in the first place. Without knowing which entities are going to interact, we cannot identify the responsibility of each participating entity.

In design process 1, we choose the customer and the travel agent as the essential entities (see Figure 7-2). In design process 2, we choose the customer, flight and hotel reservation systems as the essential entities (see Figure 7-12).

Although they start from different sets of identified essential requirements and thus different abstract interactions, the implementation requirements in the case description lead the design processes to result in similar concrete interaction structures (see Figure 7-6 and Figure 7-14). Figure 7-15 illustrates this possibility. An abstract interaction can be refined into multiple alternative concrete interaction structures, e.g., abstract interaction *D1* can be refined into concrete interaction structures *D1.1* and *D1.2*. Abstract interaction *D2* can be refined into concrete interaction structures *D2.1* and *D2.2*. Concrete interaction structure *D1.2* can be similar to or the same as concrete interaction structure *D2.1*.

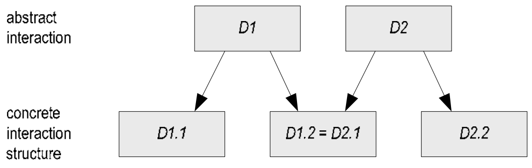


Figure 7-15
The same concrete interaction structure implements different abstract interactions

Transaction processing

In design process 1, the concrete interaction structure without transaction processing as depicted in Figure 7-16 conforms to the abstract interaction in Figure 7-2. The occurrences of interactions *cp* and *pb* represent the occurrence and non-occurrence of abstract interaction *book*, respectively. Abstract interaction *book* does not impose a requirement that transaction processing is necessary in an implementation. The transaction processing in Figure 7-6 is defined to satisfy an implementation requirement. The correctness of the transaction processing hence should be checked against that implementation requirement, and not against requirements *TR1* and *TR2* as defined in Section 7.3.3.

In design process 2, the atomic property of abstract interaction *book* in Figure 7-12 imposes a requirement that transaction processing is necessary in an implementation. The transaction processing in Figure 7-14 hence should be checked against requirements *TR1* and *TR2*.

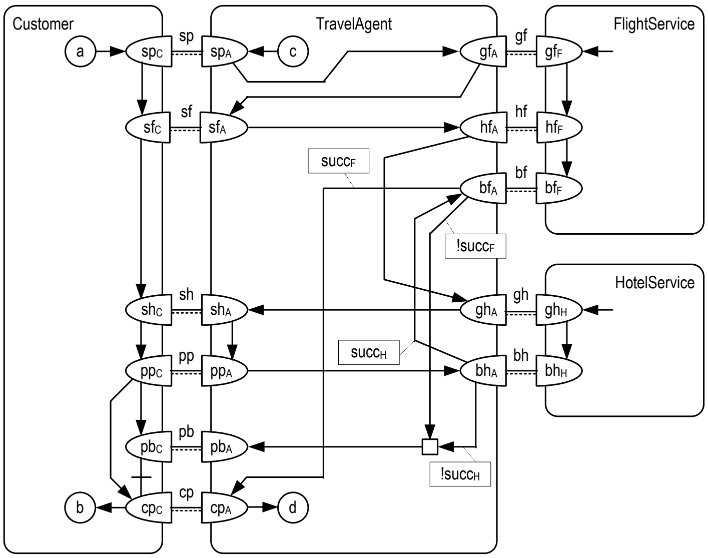


Figure 7-16
Orchestration of a customer, flight and hotel resercation system without transaction processing

Negative results and exceptions

We distinguish between a negative result and an exception message. A negative result is part of the application logic. An exception message is concerned with an unwanted event that is generated by (an) underlying service(s) during execution. An exception message can indicate, e.g., buffer overflow, connection termination, locked database, or insufficient memory space. We do not consider an exception message as part of the application logic. If an interaction returns an exception message, the interaction simply does not occur. Of course, an exception handler can be defined to allow that the execution of the application continues or terminates normally. Exceptions and exception handling are outside the scope of our case study and we leave them for future work.

7.5 Evaluation

In this section, we evaluate our interaction design concept and transformations to assess whether they serve their purposes well and can be used in practice.

Interaction concept

We evaluate our interaction concept from the perspectives of the targeted users: *business analysts* and *application designers*. A business analyst uses the interaction concept to model interactions at higher abstraction levels. An application designer uses the interaction concept to model interactions at lower abstraction levels. Regardless of the abstraction level at which a design is specified, a design should be complete and precise.

The interaction concept allows a business analyst to model a complete collaboration between business entities as a single abstract interaction. It also allows a business analyst to model a complete business collaboration as a concrete interaction structure, without having to deal with the implementation details of the interactions in that interaction structure.

The interaction concept allows an application designer to develop a complete interaction design at an implementation level, in which all interactions can be realised using available interaction mechanisms. All interactions in Figure 7-10, for example, are modelled as remote interactions. A remote interaction can be implemented as a synchronous request-response mechanism (see Chapter 5).

The contribution and distribution constraints of the interaction concept allow business analysts to model a business collaboration precisely. The contribution constraints of an abstract interaction allow a business analyst to specify precisely the participants' responsibilities in the establishment of the interaction result and their views on it. The distribution constraints allow the business analyst to specify precisely the relations between the participants' views.

Similarly, at lower abstraction levels, the contribution and distribution constraints allow an application designer to specify a concrete interaction precisely.

Design transformations

We have shown that our design transformation can be used in the design process of a service composition. However, when an abstract interaction is refined into a complex concrete interaction structure, the conformance assessment consumes a large manual effort. Tool supports can be developed to facilitate the conformance assessment in a (semi-)automatic way.

Chapter 8

Case study: enterprise application integration

In this chapter, we use our interaction concept and the ISDL2BPEL transformation tool to carry out a case study, namely enterprise application integration (EAI) for an order management system [123]. We evaluate whether the interaction concept and the transformation tool serve their purposes well and can be used in practice.

This chapter is organised as follows: Section 8.1 introduces EAI and presents an approach to design an integration solution. Section 8.2 describes the integration case. Section 8.3 presents the design of a solution for the integration case. Section 8.4 discusses the possibility of defining an integration solution at a high abstraction level. Finally, Section 8.5 evaluates the interaction concept and transformation tool.

8.1 EAI approach

This section introduces enterprise application integration (EAI) and presents an approach to design an integration solution that we use in the case study.

8.1.1 Introduction to EAI

Enterprise application integration (EAI) is an effort to make existing enterprise applications, that are usually designed separately, interoperable with each other. It results in an integration solution that enables the existing applications to interact with each other to achieve a specific goal.

Two integration architectures are identified: the *point-to-point* and *hub-and-spoke* architectures [42]. In the point-to-point architecture as illustrated in Figure 8-1, the existing applications interact directly with each other.

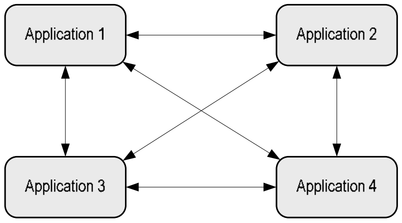


Figure 8-1
Point-to-point architecture

In the hub-and-spoke architecture as illustrated in Figure 8-2, the existing applications interact indirectly with each other via a mediator between them.

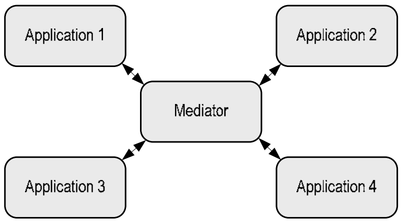


Figure 8-2
Hub-and-spoke architecture

Service-oriented computing emerges as a paradigm to support enterprise application integration [42, 77]. In service-oriented computing, an application exposes its external functionality without revealing the internals of the application. This allows an enterprise to make its applications interoperable with other enterprises' applications, while safely keeping its valuable assets, e.g., business logic and data, from the other enterprises' sight.

8.1.2 An integration approach

In this case study, we follow an integration approach [105, 112] that is for designing a mediator between the applications being integrated. This integration approach enables business analysts to participate actively in the design of an integration solution. Also, it facilitates automation of parts of the integration process.

The integration approach consists of the following steps. These steps are illustrated in Figure 8-3.

In Step 1, the platform-independent service descriptions of the applications being integrated are derived, by abstracting from platform-specific information. In terms of the MDA approach [90, 91], this means that the service PSMs (platform-specific models) of the applications being integrated are transformed to their respective service PIMs (platform-independent models).

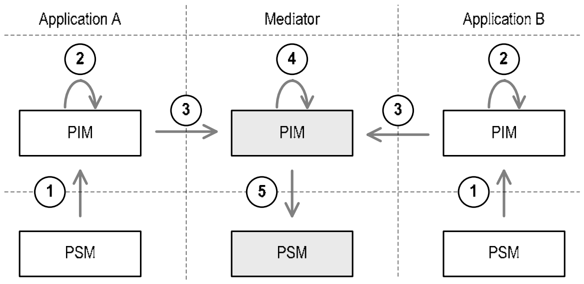


Figure 8-3
Integration approach

In Step 2, the service PIMs are semantically enriched by adding information that cannot be (automatically) derived from the service PSMs. For example, a service PSM may be complemented by some text document that describes part of the service in natural language. The purpose of semantic enrichment is to make the service PIMs complete and precise.

In Step 3, a mediator between the service PIMs is designed as an integration solution. Since the mediator is designed at the PIM level, this step enables the more active participation of business analysts. In Step 4, the correctness of the mediator is verified using one or more analysis techniques.

Finally, in Step 5, a mediator PSM is derived from the mediator PIM, by adding platform-specific information. Further, the mediator PSM is (automatically) transformed to an executable implementation.

We follow this integration approach by using our design concepts for modelling service PIMs and a mediator PIM. We then use our ISDL2BPEL transformation tool to transform the mediator PIM to an executable implementation in BPEL.

8.2 Case description

An ordering application (OA) of a customer company *Blue* will be integrated with a customer relation management (CRM) and an order management (OM) system of a manufacturing company *Moon*. Blue's OA and Moon's CRM and OM are Web Services applications. All interactions are implemented as Web Services operation invocations.

Blue's OA interacts using a simplified RosettaNet PIP3A4 message format. Blue's OA first sends a PIP3A4 Purchase Order Request (PIP3A4POR) to Moon. A purchase order request contains one or more items to be ordered. In return, Blue's OA receives an Acknowledgment of Receipt indicating that Moon has received the purchase order request. Blue's OA then waits for a PIP3A4 Purchase Order Confirmation (PIP3A4POC) from Moon with a status that indicates whether the purchase order request is *accepted*, *rejected*, or *pending*. Upon receipt of a purchase order confirmation, Blue's OA sends an Acknowledgment of Receipt to Moon. Figure 8-4 depicts the types of messages to interact with Blue's OA.

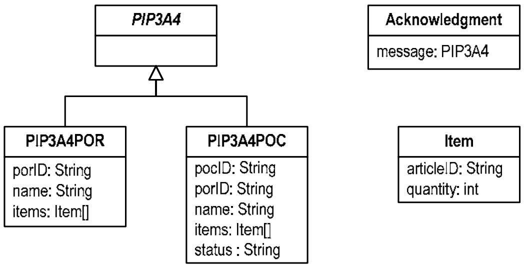


Figure 8-4
Messages to interact with Blue's OA

Moon's OM interacts using a proprietary data model and communication protocol. Moon's OM expects Blue to create a new order using a separate message containing a customer ID. It returns an order ID with which Blue can add the ordered items one-by-one to the created order. Moon's OM returns an acknowledgment each time an item is added to the order. After adding all items to the order, Blue must close the order. Moon's OM returns the number of items in the order. It then starts to send confirmations; each of which indicates the order status of an item in the order, i.e., *accepted*, *rejected*, or *pending*. Blue should return an acknowledgment for each confirmation. Figure 8-5 depicts the types of messages to interact with Moon's OM.

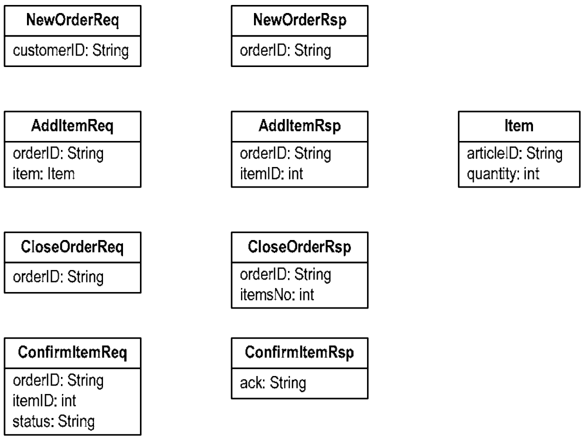
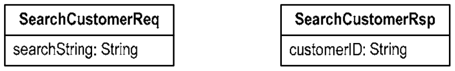


Figure 8-5
Messages to interact with Moon's OM

Moon's CRM returns the customer ID of a given customer. Figure 8-6 depicts the types of messages to interact with Moon's CRM.

Figure 8-6
Messages to interact with Moon's CRM



The information model of the original description of this integration case is larger than the information models that are depicted in Figure 8-4, Figure 8-5, and Figure 8-6, e.g., it contains information about customer address, shipment addresses, telephone numbers, e-mails, units of products, dates of order, dates of shipment, etc. As we focus on the design of the behaviour of an integration solution, we consider only the necessary information for defining an integration solution.

8.3 Integration solution

This section illustrates the application of the integration approach presented in Section 8.1.2 to the integration case described in the previous section. We use our interaction concept to specify the interactions in the integration solution.

Step 1: Abstraction from service PSMs to service PIMs

In this step, we derive the platform-independent service descriptions of Blue's OA and Moon's CRM and OM from their WSDL descriptions. This step results in the service PIMs that are depicted in Figure 8-7. We model the operation calls and operation executions of the applications using their shorthands (see Section 6.3.1). Names are used to represent information attributes, i.e., *req* and *rsp*, instead of indexed information attributes, e.g., ι_1 and ι_2.

Blue's OA has one operation call and one operation execution. Operation call *por* is for sending a purchase order request (PIP3A4POR) and receiving the acknowledgment for that purchase order request. Operation execution *poc* is for receiving a purchase order confirmation (PIP3A4POC) and sending the acknowledgment for that purchase order confirmation.

Moon's CRM has one operation execution only. Operation execution *srch* is for receiving a search string containing a customer's name and sending the customer ID of that customer.

Moon's OM has three operation executions and one operation call. Operation executions *new*, *add*, and *close* are for creating a new order, adding an item to an order, and closing an order, respectively. Operation call *conf* is for confirming the status of an item of an order

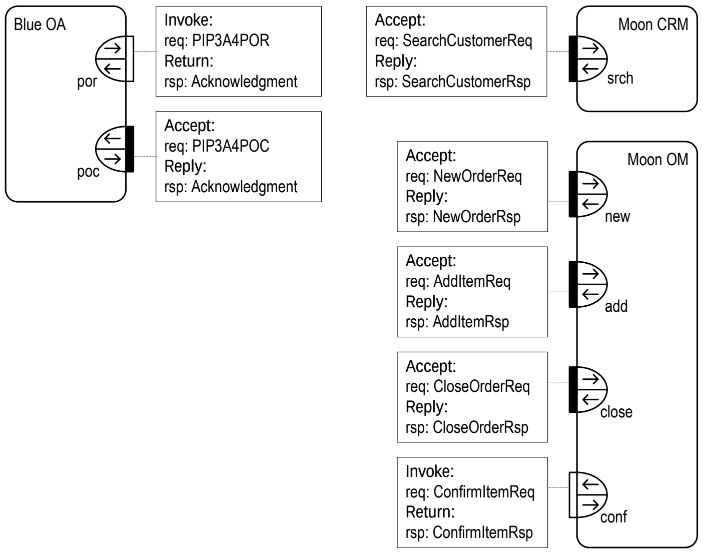


Figure 8-7
Service PIMs of Blue's OA and Moon's CRM and OM

Step 2: Semantic enrichment of the service PIMs

In this step, we define the relation between the operation calls and operation executions in Blue's OA and Moon's OM, based on the informal description in Section 8.2. Figure 8-8 depicts the semantically enriched service PIMs of Blue's OA and Moon's CRM and OM. The repetitive steps in Moon's OM are made explicit and modelled as repetitive behaviour instantiations.

Step 3: Design of the mediator PIM

In this step, we design a mediator PIM between the semantically enriched service PIMs of Blue's OA and Moon's CRM and OM. The definition of the mediator PIM consists of the definition of

– the offered and requested services of the mediator;
– the composition of these services by relating their respective operation executions and operation calls; and
– the information mapping between the information attributes of the operation executions and operation calls.

The mediator offers one service that must match the requested service of Blue's OA. This service can initially be defined as the 'complement' of the requested service of Blue's OA. The complement of a service is obtained by changing each operation call into an operation execution, and vice versa, while keeping the same information attributes. Analogously, the requested services of the mediator can be obtained by complementing the offered services of Moon's CRM and OM.

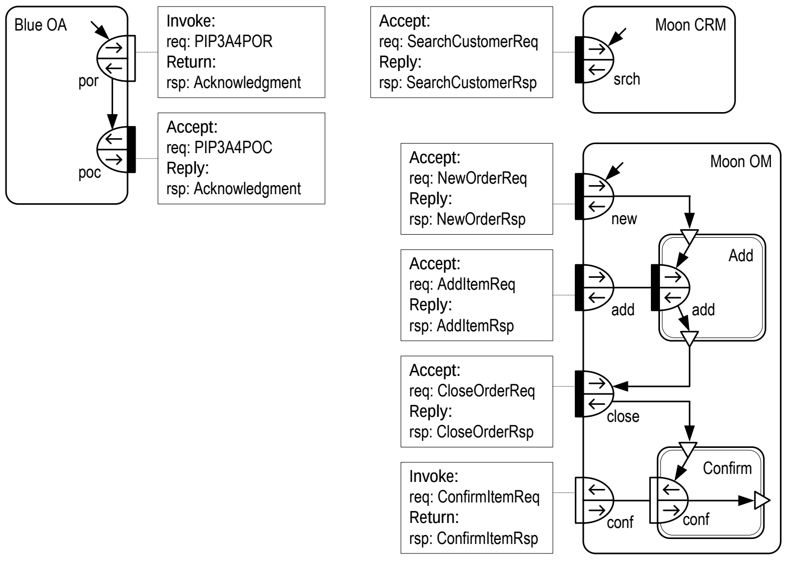


Figure 8-8
Semantically enriched service PIMs of Blue's OA and Moon's CRM and OM

The design of the mediator can now be approached as the search for a composition of the requested services from Moon's CRM and OM, which conforms to the offered services to Blue's OA. The structure of this composition is defined as causality relations among the operation calls and operation executions. Figure 8-9 depicts the obtained mediator PIM. For brevity, information attributes are omitted.

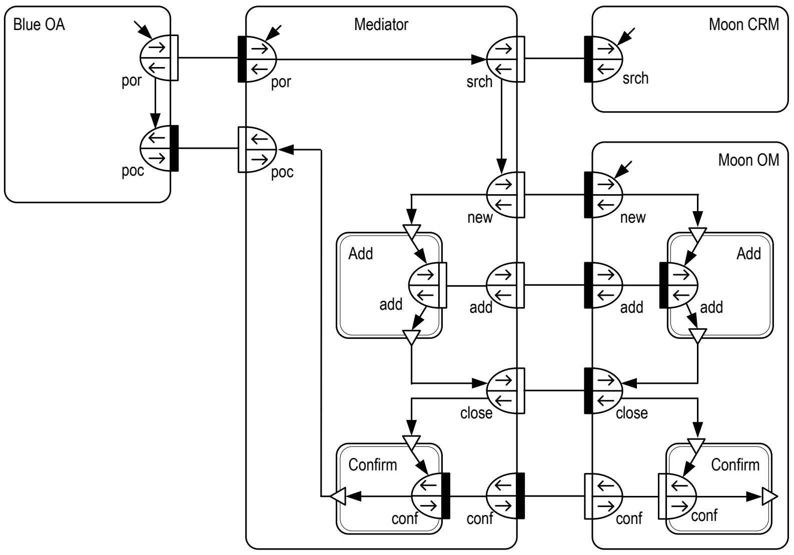
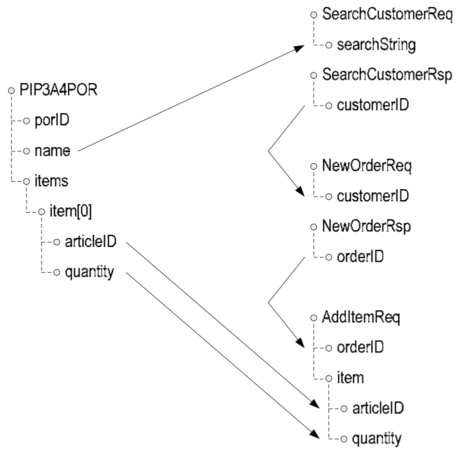


Figure 8-9
Mediator PIM

The definition of information mapping between information attributes of the operation calls and operation executions can be approached as a refinement of the causality relations among the operation calls and operation executions. This information mapping defines how the value of the information attribute of an operation call or operation execution is generated from the values of the information attributes of the other operation calls and/or operation executions.

The information mapping between the attributes of the operation executions and operation calls of the mediator is illustrated in Figure 8-10. The information mapping is used to specify constraints in the mediator PIM.

Figure 8-10
Information mapping



Step 4: Verification of the mediator PIM

In this step, we verify the mediator PIM by means of simulation. The simulation of ISDL behaviours is supported by the Grizzle tool [57, 109]. Simulation allows a designer to analyse the possible orderings of operation occurrences, as well as the information results that are established in these operations. In addition, the Grizzle tool provides hooks in the simulation process to execute application code upon execution of an operation. This enables the simulated mediator to perform real Web services invocations and to incorporate the results that are returned by Web services during the simulation. For this purpose, stub-code is linked to a modelled Web-services operation call. Furthermore, the simulator allows external applications to invoke a modelled Web-services operation execution.

Step 5: Derivation of the mediator PSM

In this final step, we transform the mediator PIM to a mediator PSM in terms of BPEL. Further, we transform the mediator PSM to an executable implementation in BPEL. The mediator PSM should contain WSDL/BPEL-specific information, as defined in Chapter 6.

We annotate the mediator PIM in Figure 8-9 with WSDL/BPEL-specific information. The following annotations are given to an operation call or operation execution: *operation*, *portType*, *partnerLink*, *namespaceURI*, and *wsdl* (see Table 6-3). Figure 8-11 illustrates the annotations that are given to operation execution *por*.

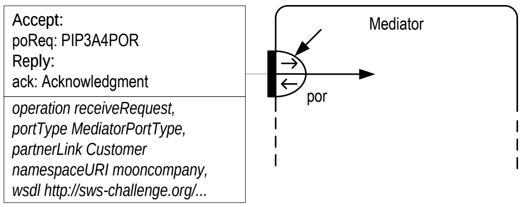

Accept:
poReq: PIP3A4POR
Reply:
ack: Acknowledgment
operation receiveRequest,
portType MediatorPortType,
partnerLink Customer
namespaceURI mooncompany,
wsdl http://sws-challenge.org/...
Mediator
por

Figure 8-11
Annotated operation execution *por*

After the model is properly annotated, the model is given as an input to the ISDL2BPEL transformation tool. The transformation tool produces a BPEL process that is ready to be deployed on a BPEL execution engine.

8.4 Discussion

In the previous section, we use our interaction concept to specify operation invocations. An integration solution is defined at that abstraction level. In this section, we discuss the possibility of using the interaction concept at a high abstraction level and, thus, defining an integration solution at a high abstraction level.

When the offered and/or requested services of an application being integrated can be represented as an abstract interaction contribution, as in [38], an integration solution can be modelled as a single abstract interaction. The information mapping is specified as the distribution constraints of the abstract interaction. Figure 8-12 depicts abstract interaction *order* that models an integration solution between Blue's OA and Moon's CRM and OM.

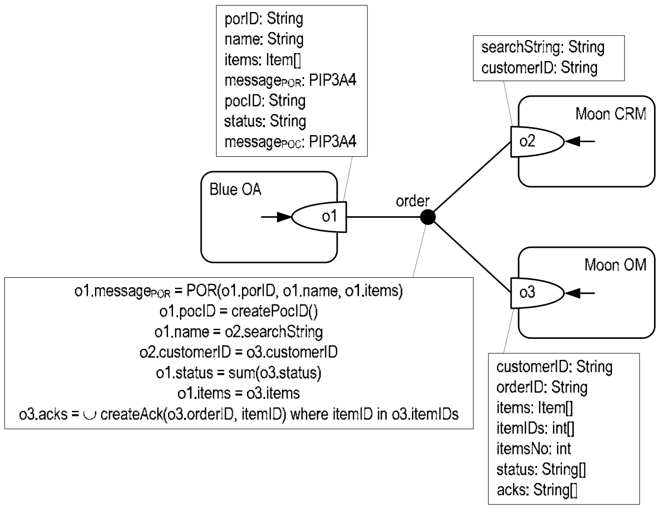


Figure 8-12
Integration solution as an abstract interaction

At an implementation level, the interaction synchronisation of abstract interaction *order* must be preserved by an integration solution. This interaction synchronisation specifies the causal dependency of Blue's OA and Moon's CRM and OM on each other. An integration solution that implements abstract interaction *order* will be more complex than the mediator in Figure 8-9 because it must preserve the interaction synchronisation of abstract interaction *order*. The mediator in Figure 8-9 does not specify the dependency of Moon's CRM on Moon's OM.

8.5 Evaluation

In this section, we evaluate our interaction concept and design transformations to assess whether they serve their purposes well and can be used in practice.

Interaction concept

We have shown that our interaction concept can be used in the integration approach that is presented in Section 8.1.2. The interaction concept is used to specify operation invocations at a low abstraction level, i.e., an interaction models the sending of a message between two participants. To facilitate the modelling of operation invocations, the shorthands for operation calls and operation executions are used. The shorthands make the interaction concept more usable in practice, because designers do not have to specify the same composition of interactions to represent operation invocations multiple times.

ISDL2BPEL transformation tool

We have shown that a mediator PSM that is properly annotated with WSDL/BPEL-specific information can be transformed to an executable implementation. It generates a BPEL process from the mediator PSM in a few seconds and, hence, it saves the development time and effort of an integration solution. The ISDL2BPEL transformation tool can be used in practice.

Chapter 9

Conclusions

This chapter presents the conclusions and contributions of this thesis and suggests directions for further research. The chapter is organised as follows: Section 9.1 presents the general conclusions of our work; Section 9.2 presents our main research contributions; and Section 9.3 suggests directions for further research.

9.1 General conclusions

This thesis proposes a concept and transformations for designing interactions in a service composition at related abstraction levels. The concept and transformations are aimed at helping designers to bridge the conceptual gap between the business and software domains. In this way, the complexity of an interaction design can be managed adequately.

A service is the establishment of some valuable effect through the interaction between a service user and service provider(s). In a service composition, a number of services are composed to deliver a new service. A service composition is specified as one or more interactions between a service user and service provider(s).

Section 1.2.3 identifies three research questions regarding the use of related abstraction levels in the development of a service composition. In the following, we answer those questions by referring to the chapters in which those questions are addressed and answered.

RQ1: What interaction design concept is suitable for modelling interactions at related abstraction levels? Are available interaction design concepts suitable for this purpose?

In Chapter 2, we define a set of suitability requirements to assess whether an interaction design concept is suitable for modelling interactions at related abstraction levels. Such an interaction design concept should allow a business analyst and application designer

- to model an interaction between two or more participants,
- to define different views of different participants on the established result,
- to specify the relation between different views of different participants, and
- to specify participants' requirements directly.

In addition, at an implementation level, an interaction design concept should be able to model interaction mechanisms precisely.

We use these suitability requirements to analyse the interaction design concepts that are used in a number of design methods for service compositions. The analysis concludes that none of the analysed interaction design concepts satisfies all the suitability requirements.

In Chapter 3, we define an interaction concept that satisfies these requirements by enhancing the ISDL interaction concept. We show that the interaction concept is suitable for modelling abstract interactions. In Chapter 5, we show that the interaction concept is suitable for modelling interaction mechanisms as concrete interaction structures and abstract interactions.

RQ2: How to transform interaction designs between related abstraction levels? How to assess the conformance between interaction designs at different abstraction levels?

In Chapter 4, we define two design transformations to support interaction design at related abstraction levels, i.e., direct interaction refinement and abstraction, as opposed to indirect interaction refinement and abstraction [107]. Direct interaction design transformation preserves the distribution of responsibility between participants. Indirect interaction design transformation performs the transformation in the integrated perspective, in which every interaction is modelled as an action. When an interaction is modelled as an action, information about the distribution of responsibility between participant disappears and, thus, cannot be preserved during transformation.

To assess the conformance between interaction designs at related abstraction levels, we define a set of conformance requirements and an assessment method to check whether those conformance requirements are satisfied. The interaction design transformations and conformance assessment method reuse and extend the operational concepts and methods for general behaviour transformations in ISDL.

RQ3: How to facilitate the development process of a service composition? How can the MDA approach contribute to that process?

To facilitate the development process of a service composition, we provide:

- patterns for interaction refinements.

In Chapter 4, we identify four patterns for interaction refinement. Each pattern indicates a possible way to refine an abstract interaction into a concrete interaction structure. The patterns are interface decomposition, new participants introduction, bilateral interactions transformation, and intermediary introduction. We show that every pattern can result in a concrete interaction structure that conforms to the original abstract interaction.

– abstract representations of interaction mechanisms.
  In Chapter 5, we represent the CORBA and Web Services interaction mechanisms as abstract interactions. These representations allow a business analyst and application designer to include the interaction mechanisms in an abstract interaction design without having to deal with the detailed behaviours of the interaction mechanisms. Following the MDA approach, those representations also abstract from the details of the technological platforms on which the interaction mechanisms are implemented. This allows them to be implemented with different supporting platforms.

– a transformation tool to transform an interaction design to an executable implementation.
  In Chapter 6, we develop a transformation tool to transform automatically an interaction design to an executable implementation in BPEL. The interaction design must comply with several modelling restrictions and be annotated with WSDL/BPEL-specific information. The use of modelling restrictions and annotations follow the MDA transformation approach that makes use of patterns (i.e., recognisable structures of elements) and markings (i.e., annotating a model with information that is specific to the target platform) [90].

9.2 Research contributions

Our work contributes to the area of service composition design and model-driven engineering. Specifically, it contributes to
– interaction design concepts,
– interaction design transformations,
– interaction mechanism representations, and
– model transformations.

Contributions to interaction design concepts
We enhance the ISDL interaction concept to make it suitable for modelling interactions at related abstraction levels. Specifically, the enhanced

interaction concept allows different participants to have different views on the interaction result. Distribution constraints are defined to relate these participants' views. The enhanced interaction concept allows us to represent a complex composition of interactions by a single interaction and refine it.

The enhanced interaction concept is generic with regard to abstraction levels and application domains. Although the interaction concept is developed for ISDL, it can be (partly) adopted by other design languages. In a broader scope, we contribute to the research toward concepts for representing interactions at various stages in a design process, such as in [3, 24, 58, 72, 73, 115].

Contributions to interaction design transformation

We define interaction refinement and abstraction methods that preserve the distribution of responsibility between participants. We define a set of conformance requirements and an assessment method to check the conformance between interaction designs at related abstraction levels. Also, we identify four patterns for interaction refinement. In a broader scope, we contribute to the research toward interaction refinement and abstraction, such as in [6, 13, 27, 28, 119, 126].

Contributions to interaction mechanism representations

The use of design patterns that are concerned with interactions, e.g., in [11, 16, 46, 54, 70], have become common practice to describe interaction structures that satisfy generic requirements in specific application domains. We model the behaviours of interaction mechanisms as interaction patterns. Further, we represent them as abstract interactions. These abstract representations contribute to the research towards interaction mechanism representations, such as in [30, 34, 68, 78, 115, 116].

Contributions to model transformations

Our transformation tool is developed as a composition of smaller transformations. The purpose of transformation (de)composition, as investigated in [5, 39, 67, 79], is to manage transformation complexity and to allow reuse. To allow reuse, we define a language for defining an intermediate model. This intermediate model documents the behavioural patterns that are found in a source model.

We define a framework for evaluating and selecting options for data manipulation in a service composition, based on the following criteria: feasibility, efficiency, reusability, merging, and portability. The framework defines the quality and quantity values for those criteria and a formula for selecting an option.

9.3 Directions for further research

We suggest the following directions for further research.

Quality-of-service modelling
The ISDL behavioural concepts include, among others, time and uncertainty attributes. Further investigation should aim at elaborating those attributes for modelling the quality-of-service (QoS) of an interaction, such as delay, throughput, and reliability. These qualities can be derived from a service level agreement (SLA) between the participants. An abstract interaction specifies the desired QoS and a concrete interaction structure specifies how to deliver that QoS. Conversely, one should be able to calculate the total QoS of a concrete interaction structure and represent it in an abstract interaction. This investigation can extend the work in [107, 110].

Verification of interaction designs
Conformance assessment checks whether a concrete interaction structure conforms to an abstract interaction. It does not check, for example, the possibility of a deadlock during execution. Further investigation should aim at the verification of a concrete interaction structure, e.g., for liveness, reachability, or boundedness properties. The verification of those properties can be done by using an available analysis language, such as Petri Nets. A mapping from the ISDL behavioural concepts to an analysis language is therefore necessary. Initial work [105] has been done on this mapping and further investigation can extend that work.

Tool support for conformance assessment
The case studies in Chapter 7 and 8 indicate the need of tool support for conformance assessment. Further investigation should aim at the development of such tool support. The tool support should include (semi-) automatic interaction abstraction and interaction design comparison. It would facilitate a design process and encourage designers to use our interaction design concept and transformations. This investigation can use and extend the ISDL formal semantics [107].

Mapping other design languages to ISDL
The aforementioned tool support could serve as a generic tool supporting conformance assessment, as illustrated in Figure 9-1. This would allow a business analyst and application designer to develop interaction designs using their preferred design languages, e.g., L1 and L2 in the figure, and then to check the conformance between the designs using the tool support. In addition, the ISDL2BPEL transformation tool could be used to

transform the interaction design, which is obtained from the translation, to an executable implementation. In this way, the ISDL behavioural concepts serve as a common semantic meta-model that relates design languages, analysis languages, and implementation languages [113]. Further investigation should aim at mapping the behavioural concepts of other design languages to the ISDL behavioural concepts.

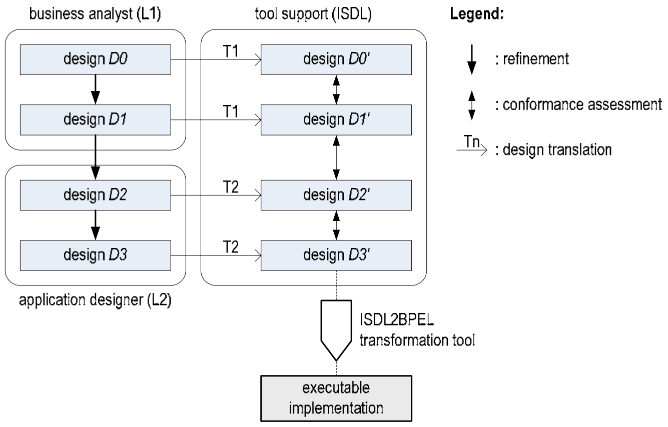


Figure 9-1
Conformance assessment of interaction designs that use different design languages

Appendix A

Conformance assessments in case study 1

This appendix provides the conformance assessment of the refinements that are done in case study 1 (travel reservation application).

A.1 Design process 1

Four refinements are done in this design process. Refinements 1 and 3 are interaction refinements. Refinement 2 and 4 are causality refinements that are followed by the refinement of inserted actions into interactions.

A.1.1 Refinement 1

Abstract interaction *book* in Figure 7-2 is refined into the concrete interaction structure in Figure 7-4. Table A-1 depicts the correspondence relation between them.

Step 1
The participants and attributes of the concrete interaction structure that are listed in Table A-1 should be preserved.

Step 2
The only final interaction *cp* depends on context actions *a* and *c* via interactions *pp*, *sh*, *sf*, and *sp*. The concrete interaction structure provides synchronisation as provided by abstract interaction *book*. Conformance requirement *IR4* is satisfied.

Step 3
The concrete interaction contribution structure in participant *Customer* is abstracted into abstract interaction contribution b_C that has the preserved

attributes of concrete participant *Customer* in Table A-1. The resulted abstraction is depicted in Figure A-1.

Table A-1
Correspondence relation in refinement 1

	Abstract interaction	**Concrete interaction structure**
Participants	Customer	Customer
	TravelAgent	TravelAgent
Attributes	b_C.name	pp_C.name
	b_C.departure	sp_C.departure
	b_C.destination	sp_C.destination
	b_C.dateStart	sp_C.dateStart
	b_C.dateEnd	sp_C.dateEnd
	b_C.flightOut	sf_C.flightOut
	b_C.flightIn	sf_C.flightIn
	b_C.hotel	sh_C.hotel
	b_C.price	pp_C.price
	b_T.name	pp_T.name
	b_T.departure	sp_T.departure
	b_T.destination	sp_T.destination
	b_T.dateStart	sp_T.dateStart
	b_T.dateEnd	sp_T.dateEnd
	b_T.flightOut	sf_T.flightOut
	b_T.flightIn	sf_T.flightIn
	b_T.hotel	sh_T.hotel
	b_T.price	pp_T.price
Occurrences	book	cb

Figure A-1
The abstraction of concrete participant *Customer*

```
Customer' = {
    a → b_C (name: String, departure: String, destination: String, dateStart: Date, dateEnd: Date,
    flightOut: Flight, flightIn: Flight, hotel: Hotel, price: double)
        [getDeparture(flightOut) = getDestination(flightIn) = departure,
        getDestination(flightOut) = getDeparture(flightIn) = destination,
        getLocation(hotel) = destination,
        getDate(flightOut) = getDateIn(hotel) = dateStart,
        getDate(flightIn) = getDateOut(hotel) = dateEnd],

    b_C → b
}
```

The concrete interaction contribution structure in participant *TravelAgent* is abstracted into abstract interaction contribution b_T that has the preserved attributes of concrete participant *TravelAgent* in Table A-1. The resulted abstraction is depicted in Figure A-2.

Figure A-2
The abstraction of concrete participant *TravelAgent*

```
TravelAgent' = {
    c → b_T (name: String, departure: String, destination: String, dateStart: Date, dateEnd: Date,
    flightOut: Flight, flightIn: Flight, hotel: Hotel, price: double)
        [flightOut in listFlights(departure, destination, dateStart),
        flightIn in listFlights(destination, departure, dateEnd),
        hotel in listHotels(destination, dateStart, dateEnd),
        price = getPrice(flightOut) + getPrice(flightIn) + getPrice(hotel) + fee],

    b_T → d
}
```

Step 4

The interactions in Figure 7-4 are defined as remote interactions. Their implicit distribution constraints determine the distribution constraints of abstract interaction *book'* between abstract participants *Customer'* and *TravelAgent'*, as depicted in Figure A-3.

Figure A-3
Abstract interaction *book'* between abstract participants *Customer'* and *TravelAgent'*

```
book' (b_C: Customer'.b_C, b_T: TravelAgent'.b_T) [
    b_C.name = b_T.name,
    b_C.departure = b_T. departure,
    b_C.destination = b_T.destination,
    b_C.dateStart = b_T.dateStart,
    b_C.dateEnd = b_T.dateEnd,
    b_C.flightOut = b_T.flightOut,
    b_C.flightIn = b_T.flightIn,
    b_C.hotel = b_T.hotel,
    b_C.price = b_T.price]
```

Alternatively, this abstract interaction can be defined as a remote interaction as depicted in Figure A-4.

Figure A-4
Abstract interaction *book'* as a remote interaction

```
book' (b_C: Customer'.b_C, b_T: TravelAgent'.b_T) [remote]
```

Abstract interaction *book'* has an equivalence correctness relation with the original abstract interaction *book*. The concrete interaction structure in Figure 7-4 conforms to abstract interaction *book* in Figure 7-2.

A.1.2 Refinement 2

The orchestration in Figure 7-6 is obtained from the refinements of the causality relations between the interaction contributions of abstract participant *TravelAgent* in Figure 7-4, that are followed by the refinement of the inserted actions into interactions. An inserted action that is refined into an interaction is called *an inserted interaction*. The inserted interactions are

- interaction *gf*, which is inserted between interaction contributions sp_T and sf_T;
- interactions *hf* and *gh*, which are inserted between interaction contributions sf_T and sh_T; and
- interactions *bh*, *bf*, and *ch*, which are inserted between interaction contributions pp_T, pb_T, and cp_T.

To abstract from an inserted interaction, we define the following steps.

1. Abstract an inserted interaction into an integrated interaction and model it as an action, i.e., an inserted action.
2. Abstract from the inserted action.

Interaction gf

Inserted interaction *gf* is modelled as inserted action *gf*. This inserted action is depicted in Figure A-5 and textually in Figure A-6.

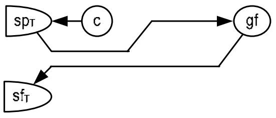


Figure A-5
Interaction *gf* as an inserted action

Figure A-6
Textual expression of Figure A-5

```
c → spT (departure: String, destination: String, dateStart: Date, dateEnd: Date),

spT → gf (departure: String, destination: String, dateOut: Date, dateIn: Date, flightsOut: Flight[],
flightsIn: Flight[])
    [departure = spT.departure,
    destination = spT.destination,
    dateOut = spT.dateStart,
    dateIn = spT.dateEnd,
    flightsOut = listFlights(departure, destination, dateOut),
    flightsIn = listFlights(destination, departure, dateIn)],

gf → sfT (flightOut: Flight, flightIn: Flight)
    [flightOut in gf.flightsOut,
    flightIn in gf.flightsIn]
```

The causality relations in Figure A-5 is abstracted from inserted action *gf*. This results in the causality relations that are depicted in Figure A-7 and textually in Figure A-8.

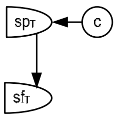


Figure A-7
Abstract from inserted action *gf*

Figure A-8
Textual expression of Figure A-7

c → sp$_T$ (departure: String, destination: String, dateStart: Date, dateEnd: Date),

sp$_T$ → sf$_T$ (flightOut: Flight, flightIn: Flight)
    [flightOut in listFlights(sp$_T$.departure, sp$_T$.destination, sp$_T$.dateStart),
    flightIn in listFlights(sp$_T$.destination, sp$_T$.departure, sp$_T$.dateEnd)]

The obtained causality relation between *sp$_T$* and *sf$_T$* has an equivalence correctness relation with the original causality relation.

Interactions hf and gh

Inserted interactions *hf* and *gh* are modelled as inserted actions *hf* and *gh*. These inserted actions are depicted in Figure A-9 and textually in Figure A-10.

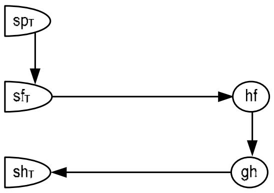


Figure A-9
Interactions *hf* and *gh* as inserted actions

Figure A-10
Textual expression of Figure A-9

sp$_T$ → sf$_T$ (flightOut: Flight, flightIn: Flight)
    [flightOut in listFlights(sp$_T$.departure, sp$_T$.destination, sp$_T$.dateStart),
    flightIn in listFlights(sp$_T$.destination, sp$_T$.departure, sp$_T$.dateEnd)]

sf$_T$ → hf (flightOut: Flight, flightIn: Flight, expiryDate: Date)
    [flightOut = sf$_T$.flightOut,
    flightIn = sf$_T$.flightIn,
    expiryDate = getExpiryDate(currentDate())],

hf → gh (location: String, dateIn: Date, dateOut: Date, hotels: Hotel[])
    [location = sp$_T$.destination,
    dateIn = sp$_T$.dateStart,
    dateOut = sp$_T$.dateEnd,
    hotels = listHotels(location, dateIn, dateOut)],

gh → sh$_T$ (hotel: Hotel)
    [hotel in gh.hotels]

The causality relations in Figure A-9 are abstracted from inserted actions *hf*. This results in the causality relations that are depicted in Figure A-11 and textually in Figure A-12.

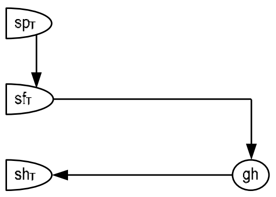


Figure A-11
Abstract from inserted action *hf*

Figure A-12
Textual expression of Figure A-11

sp$_T$ → sf$_T$ (flightOut: Flight, flightIn: Flight)
    [flightOut in listFlights(sp$_T$.departure, sp$_T$.destination, sp$_T$.dateStart),
    flightIn in listFlights(sp$_T$.destination, sp$_T$.departure, sp$_T$.dateEnd)]

sf$_T$ → gh (location: String, dateIn: Date, dateOut: Date, hotels: Hotel[])
    [location = sp$_T$.destination,
    dateIn = sp$_T$.dateStart,
    dateOut = sp$_T$.dateEnd,
    hotels = listHotels(location, dateIn, dateOut)],

gh → sh$_T$ (hotel: Hotel)
    [hotel in gh.hotels]

The causality relations in Figure A-11 are abstracted from inserted action *gh*. This results in the causality relations that are depicted in Figure A-13 and textually in Figure A-14.

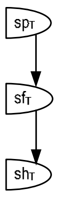


Figure A-13
Abstract from inserted action *gh*

Figure A-14
Textual expression of Figure A-13

sp$_T$ → sf$_T$ (flightOut: Flight, flightIn: Flight)
    [flightOut in listFlights(sp$_T$.departure, sp$_T$.destination, sp$_T$.dateStart),
    flightIn in listFlights(sp$_T$.destination, sp$_T$.departure, sp$_T$.dateEnd)]

sf$_T$ → sh$_T$ (hotel: Hotel)
    [hotel in listHotels(sp$_T$.destination, sp$_T$.dateStart, sp$_T$.dateEnd)]

The obtained causality relation between sf_T and sh_T has an equivalence correctness relation with the original causality relation.

Interactions bh, bf, and ch

Inserted interactions *bh*, *bf*, and *ch* are modelled as inserted actions *bh*, *bf*, and *ch*. These inserted actions are depicted in Figure A-15 and textually in Figure A-16.

Figure A-15
Interactions *bh*, *bf*, and *ch* as an inserted actions

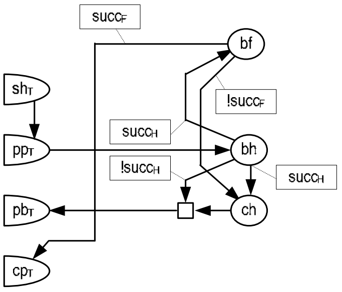


Figure A-16
Textual expression of Figure A-15

```
shT → ppT (name: String, price: double)
    [price = getPrice(sfT.flightOut) + getPrice(sfT.flightIn) + getPrice(shT.hotel) + fee],

ppT → bh (name: String, hotel: Hotel, price: double, code: long)
    [name = ppT.name,
    hotel = shT.hotel
    price = getPrice(hotel),
    code = getCode(name, hotel)],

bh [bh.code ≥ 0] → bf (name: String, flightOut: flight, flightIn: Flight, price: double, code: long)
    [name = ppT.name,
    flightOut = sfT.flightOut,
    flightIn = sfT.flightIn,
    price = getPrice(flightOut) + getPrice(flightIn),
    code ∈ {getCode(name, flightOut, flightIn), -1}],

bf [bf.code < 0] ∧ bh [bh.code ≥ 0] → ch (code: long)
    [code = bh.code],

bh [bh.code < 0] ∨ ch  → pbT (payback: double)
    [payback = ppT.price],

bf [bf.code ≥ 0] → cpT (code: long[2])
    [code[0] = bf.code,
    code[1] = bh.code]
```

The causality relations in Figure A-15 are abstracted from inserted action *bf*. This results in the causality relations that are depicted in Figure A-17 and textually in Figure A-18. The exclusive choice between cp_T and *ch* that is implied by the positive and negative results of action *bf* are modelled explicitly.

Figure A-17
Abstract from inserted action *bf*

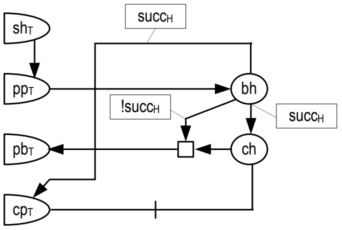


Figure A-18
Textual expression of Figure A-17

$sh_T \rightarrow pp_T$ (name: String, price: double)
    [price = getPrice(sf_T.flightOut) + getPrice(sf_T.flightIn) + getPrice(sh_T.hotel) + fee],

$pp_T \rightarrow$ bh (name: String, hotel: Hotel, price: double, code: long)
    [name = pp_T.name,
    hotel = sh_T.hotel
    price = getPrice(hotel),
    code = getCode(name, hotel)],

bh [bh.code $\geq$ 0] $\wedge \neg cp_T \rightarrow$ ch (code: long)
    [code = bh.code],

bh [bh.code < 0] $\vee$ ch $\rightarrow pb_T$ (payback: double)
    [payback = pp_T.price],

bh [bh.code $\geq$ 0] $\wedge \neg$ch $\rightarrow cp_T$ (code: long[2])
    [code[0] = getCode(pp_T.name, sf_T.flightOut, sf_T.flightIn),
    code[1] = bh.code],

The causality relations in Figure A-17 are abstracted from inserted action *ch*. This results in the causality relations that are depicted in Figure A-19 and textually in Figure A-20.

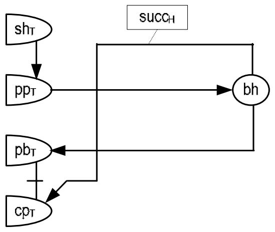


Figure A-19
Abstract from inserted action *ch*

Figure A-20
Textual expression of Figure A-19

$sh_T \rightarrow pp_T$ (name: String, price: double)
  [price = getPrice(sf_T.flightOut) + getPrice(sf_T.flightIn) + getPrice(sh_T.hotel) + fee],

$pp_T \rightarrow bh$ (name: String, hotel: Hotel, price: double, code: long)
  [name = pp_T.name,
  hotel = sh_T.hotel
  price = getPrice(hotel),
  code = getCode(name, hotel)],

$bh \wedge \neg cp_T \rightarrow pb_T$ (payback: double)
  [payback = pp_T.price],

$bh\ [bh.code \geq 0] \wedge \neg pb_T \rightarrow cp_T$ (code: long[2])
  [code[0] = getCode(pp_T.name, sf_T.flightOut, sf_T.flightIn),
  code[1] = bh.code],

The causality relations in Figure A-19 are abstracted from inserted actions *bh*. This results in causality relations that are depicted in Figure A-21 and textually in Figure A-22.

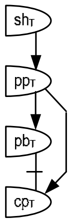


Figure A-21
Abstract from inserted action *bh*

Figure A-22
Textual expression of Figure A-21

$sh_T \rightarrow pp_T$ (name: String, price: double)
  [price = getPrice(sf_T.flightOut) + getPrice(sf_T.flightIn) + getPrice(sh_T.hotel) + fee],

$pp_T \wedge \neg cp_T \rightarrow pb_T$ (payback: double)
  [payback = pp_T.price],

$pp_T \wedge \neg pb_T \rightarrow cp_T$ (code: long[2])
  [code[0] = getCode(pp_T.name, sf_T.flightOut, sf_T.flightIn),

```
code[1] = getCode(pp_T.name, sh_T.hotel],
```

The obtained causality relation between pp_T, pb_T, and cp_T has an equivalence correctness relation with the original causality relation.

A.1.3 Refinement 3

Abstract interaction *bh* in Figure 7-6 is refined into the concrete interaction structure in Figure 7-8. Table A-2 depicts the correspondence relation between them.

Table A-2
Correspondence relation in refinement 3

	Abstract interaction	**Concrete interaction structure**
Participants	TACoordinator	TACoordinator
	HotelRS	HotelRS
Attributes	bh_T.name	rh_T.name
	bh_T.hotel	rh_T.hotel
	bh_T.price	ph_T.price
	bh_T.code	oh_T.code
	bh_H.name	rh_H.name
	bh_H.hotel	rh_H.hotel
	bh_H.price	ph_H.price
	bh_H.code	oh_H.code
Occurrences	bh	confirm

Step 1
The participants and attributes of the concrete interaction structure that are listed in Table A-2 should be preserved.

Step 2
The only final interaction *confirm* depends on context interaction contributions pp_T and gh_H via interactions *request* and *payment*. The concrete interaction structure provides synchronisation as provided by abstract interaction *bh*. Conformance requirement *IR4* is satisfied.

Step 3
The concrete interaction contribution structure in participant *TACoordinator* is abstracted into abstract interaction contribution bh_T that has the preserved attributes of the concrete participant *TACoordinator* in Table A-2. The resulted abstraction is depicted in Figure A-23.

Figure A-23
The abstraction of concrete participant *TACoordinator*

```
TACoordinator' = {
    …
    $pp_T \rightarrow bh_T$ (name: String, hotel: Hotel, price: double, code: long)
        [name = $pp_T$.name,
        hotel = $sh_T$.hotel
        price = getPrice(hotel)],

    $bh_T$ [$bh_T$.code $\geq$ 0] $\rightarrow bf_T$ (…)
    $bh_T$ [$bh_A$.code < 0] $\rightarrow pb_A$ (…)
    …
}
```

The concrete interaction contribution structure in participant *HotelRS* is abstracted into abstract interaction contribution bh_H that has the preserved attributes of the concrete participant *HotelRS* in Table A-2. The resulted abstraction is depicted in Figure A-24.

Figure A-24
The abstraction of concrete participant *HotelRS*

```
HotelRS' = {
    …
    $gh_H \rightarrow bh_H$ (name: String, hotel: Hotel, price: double, code: long)
        [hotel in $gh_H$.hotels,
        price = getPrice(hotel),
        code = getCode(name, hotel)],

    $bh_H$ [$bh_H$.code $\geq$ 0] $\rightarrow ch_H$ (…)
}
```

Step 4

The interactions in Figure 7-8 are defined as remote interactions. Their implicit distribution constraints determine the distribution constraints of interaction *bh'* between abstract participants *TACoordinator'* and *HotelRS'*, as depicted in Figure A-25.

Figure A-25
Abstract interaction *bh'* between abstract participants *TACoordinator'* and *HotelRS'*

```
bh' ($bh_T$: TACoordinator'.$bh_T$, $bh_H$: HotelRS'.$bh_H$) [
    $bh_T$.name = $bh_H$.name,
    $bh_T$.hotel = $bh_H$.hotel,
    $bh_T$.price = $bh_H$.price,
    $bh_T$.code = $bh_H$.code]
```

Alternatively, this abstract interaction can be defined as a remote interaction as depicted in Figure A-26.

Figure A-26
Abstract interaction *bh'* as a remote interaction

bh' (bh$_T$: TACoordinator'.bh$_T$, bh$_H$: HotelRS'.bh$_H$) [remote]

Abstract interaction *bh'* has an equivalence correctness relation with the original abstract interaction *bh*. The concrete interaction structure in Figure 7-8 conforms to abstract interaction *bh* in Figure 7-6.

A.1.4 Refinement 4

Interaction *auth* in Figure 7-10 is inserted in the causality relation between interaction contributions *rh$_T$* and *ph$_T$* of participant *TACoordinator* in Figure 7-8. We follow the steps defined in Section A.1.2 to abstract from this inserted interaction.

Inserted interaction *auth* is modelled as inserted action *auth*. This inserted action is depicted in Figure A-27 and textually in Figure A-28.

Figure A-27
Interaction *auth* as an inserted action

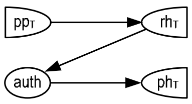


Figure A-28
Textual expression of Figure A-27

```
pp_T → rh_T (name: String, hotel: Hotel)
    [name = pp_T.name,
    hotel = sh_T.hotel],

rh_T → auth (name: String, ccNo: String, authID: long)
    [name = pp_T.name,
    ccNo = pp_T.ccNo,
    authID = authorise(name, ccNo)],

auth → ph_T (ccNo: String, authID: long, price: double)
    [ccNo = au_T.ccNo,
    authID = au_T.authID,
    price = getPrice(rh_T.hotel)]
```

The causality relations in Figure A-27 are abstracted from inserted action *auth*. This results in the causality relations that are depicted in Figure A-29 and textually in Figure A-30.

Figure A-29
Abstract from inserted action *auth*

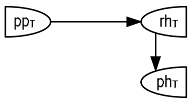


Figure A-30
Textual expression of Figure A-29

```
$pp_T \rightarrow rh_T$ (name: String, hotel: Hotel)
    [name = $pp_T$.name,
    hotel = $sh_T$.hotel],

$rh_T \rightarrow ph_T$ (price: double)
    [price = getPrice($rh_T$.hotel)]
```

The obtained causality relation between rh_T and ph_T has an equivalence correctness relation with the original causality relation.

A.2 Design process 2

Interaction *book* in Figure 7-12 is refined into the concrete interaction structure in Figure 7-14. Table A-3 depicts the correspondence relation between them.

Step 1
The participants and attributes of the concrete interaction structure that are listed in Table A-3 should be preserved.

Step 2
This step has been presented in Section 7.3.3. It concludes that conformance requirement *IR4* is satisfied.

Step 3
The concrete interaction contribution structure in participant *Customer* is abstracted into abstract interaction contribution b_C that has the preserved attributes of concrete participant *Customer* in Table A-3. The resulted abstraction is depicted in Figure A-31.

The concrete interaction contribution structure in participant *FlightRS* is abstracted into abstract interaction contribution b_F that has the preserved attributes of concrete participant *FlightRS* in Table A-3. The resulted abstraction is depicted in *Figure A-32*.

The concrete interaction contribution structure in participant *HotelRS* is abstracted into abstract interaction contribution b_H that has the preserved attributes of concrete participant *HotelRS* in Table A-3. The resulted abstraction is depicted in Figure A-33.

Table A-3
Correspondence relation in the refinement in design process 2

	Abstract interaction	Concrete interaction structure
Participants	Customer	Customer
	FlightRS	FlightRS
	HotelRS	HotelRS
Attributes	b_C.name	pp_C.name
	b_C.departure	sp_C.departure
	b_C.destination	sp_C.destination
	b_C.dateStart	sp_C.dateStart
	b_C.dateEnd	sp_C.dateEnd
	b_C.flightOut	sf_C.flightOut
	b_C.flightIn	sf_C.flightIn
	b_C.hotel	sh_C.hotel
	b_C.price	pp_C.price
	b_F.name	bf_F.name
	b_F.departure	gf_F.departure
	b_F.destination	gf_F.destination
	b_F.dateOut	gf_F.dateOut
	b_F.dateIn	gf_F.dateIn
	b_F.flightOut	bf_F.flightOut
	b_F.flightIn	bf_F.flightIn
	b_F.price	bf_F.price
	b_H.name	bh_H.name
	b_H.location	gh_H.location
	b_H.dateIn	gh_H.dateIn
	b_H.dateOut	gh_H.dateOut
	b_H.hotel	bh_H.hotel
	b_H.price	bh_H.price
Occurrences	book	cp $\wedge$ bf $\wedge$ bh

Figure A-31
The abstraction of concrete participant *Customer*

```
Customer' = {
    a → b_C (name: String, departure: String, destination: String, dateStart: Date, dateEnd: Date,
    flightOut: Flight, flightIn: Flight, hotel: Hotel, price: double)
        [getDeparture(flightOut) = getDestination(flightIn) = departure,
        getDestination(flightOut) = getDeparture(flightIn) = destination,
        getLocation(hotel) = destination,
        getDate(flightOut) = getDateIn(hotel) = dateStart,
        getDate(flightIn) = getDateOut(hotel) = dateEnd],

    b_C → b
}
```

Figure A-32
The abstraction of concrete participant *FlightRS*

```
FlightRS' = {
    e → $b_F$ (name: String, departure: String, destination: String, dateOut: Date, dateIn: Date,
    flightOut: Flight, flightIn: Flight, price: double)
        [flightOut in listFlights(departure, destination, dateOut),
        flightIn in listFlights(destination, departure, dateIn),
        price = getPrice(flightOut) + getPrice(flightIn)]

    $b_F$ → f
}
```

Figure A-33
The abstraction of concrete participant *HotelRS*

```
HotelRS' = {
    g → $b_F$ (name: String, location: String, dateIn: Date, dateOut: Date, hotel: Hotel, price:
    double)
        [hotel in listHotels(location, dateIn, dateOut),
        price = getPrice(hotel)]

    $b_F$ → h
}
```

Step 4

The interactions in Figure 7-14 are defined as remote interactions. Their implicit distribution constraints must be taken into account in the distribution constraints of abstract interaction *book'* between abstract participants *Customer'*, *FlightRS*', and *HotelRS'* as depicted in Figure A-34.

Figure A-34
Abstract interaction *book'* between abstract participants *Customer'*, *FlightRS'*, and *HotelRS'*

```
book' ($b_C$: Customer'.$b_C$, $b_F$: FlightRS'.$b_F$, $b_H$: HotelRS'.$b_H$) [
    $b_C$.name = $b_F$.name = $b_H$.name,
    $b_C$.departure = $b_F$.departure,
    $b_C$.destination = $b_F$.destination = $b_H$.location,
    $b_C$.dateStart = $b_F$.dateOut = $b_H$.dateIn,
    $b_C$.dateEnd = $b_F$.dateIn = $b_H$.dateOut,
    $b_C$.flightOut = $b_F$.flightOut,
    $b_C$.flightIn = $b_F$.flightIn,
    $b_C$.hotel = $b_H$.hotel,
    $b_C$.price = $b_F$.price + $b_H$.price + fee]
```

The calculation to determine those contribution constraints is shown in Figure A-35. Constraints that are preceded with the symbol '■' are the constraints of the abstract interaction. These constraints are obtained by replacing the attributes at the concrete level with the corresponding attributes at the abstract level.

Figure A-35
Calculation to determine the distribution constraints of abstract interaction *book'*

pp_C.name = pp_T.name = bh_T.name = bh_H.name = bf_T.name = bf_F.name
pp_C.name = bf_F.name = bh_H.name
■ b_C.name = b_F.name = b_H.name

sp_C.departure = sp_T.departure = gf_T.departure = gf_F.departure
sp_C.departure = gf_F.departure
■ b_C.departure = b_F.departure

sp_C.destination = sp_T.destination = gf_T.destination = gf_F.destination = gh_T.location = gh_H.location
sp_C.destination = gf_F.destination = gh_H.location
■ b_C.destination = b_F.destination = b_H.location

sp_C.dateStart = sp_T.dateStart = gf_T.dateOut = gf_F.dateOut = gh_T.dateIn = gh_H.dateIn
sp_C.dateStart = gf_F.dateOut = gh_H.dateIn
■ b_C.dateStart = b_F.dateOut = b_H.dateIn

sp_C.dateEnd = sp_T.dateEnd = gf_T.dateIn = gf_F.dateIn = gh_T.dateOut = gh_H.dateOut
sp_C.dateEnd = gf_F.dateIn = gh_H.dateOut
■ b_C.dateEnd = b_F.dateIn = b_H.dateOut

sf_C.flightOut = sf_T.flightOut = hf_T.flightOut = bf_T.flightOut = bf_F.flightOut
sf_C.flightOut = bf_F.flightOut
■ b_C.flightOut = b_F.flightOut

sf_C.flightIn = sf_T.flightIn = hf_T.flightIn = bf_T.flightIn = bf_F.flightIn
sf_C.flightIn = bf_F.flightIn
■ b_C.flightIn = b_F.flightIn

sh_C.hotel = sh_T.hotel = bh_T.hotel = bh_H.hotel
sh_C.hotel = bh_H.hotel
■ b_C.hotel = b_H.hotel

pp_C.price = pp_T.price = getPrice(sf_T.flightOut) + getPrice(sf_T.flightIn) + getPrice(sh_T.hotel) + fee
pp_C.price = bf_T.price + bh_T.price + fee
pp_C.price = bf_F.price + bh_H.price + fee
■ b_C.price = b_F.price + b_H.price + fee

Abstract interaction *book'* has an equivalence correctness relation with the original abstract interaction *book*. The concrete interaction structure in Figure 7-14 conforms to abstract interaction *book* in Figure 7-12.

References

1. Active Endpoints Inc., *ActiveBPEL Engine 2.0*, http://www.active-endpoints.com/active-bpel-engine-overview.htm
2. Active Endpoints Inc., *ActiveVOS*, http://www. activevos.com
3. R. Allen and D. Garlan, “A Formal Basis for Architectural Connection”, *ACM Transactions on Software Engineering and Methodology*, vol. 6, no.3, ACM Press, 1997, pp. 213-249.
4. J.P. Almeida, R. Dijkman, M. van Sinderen, and L. Ferreira Pires, “On the Notion of Abstract Platform in MDA Development”, *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)*, IEEE Computer Society Press, 2004, pp. 253-263.
5. J.P. Almeida, L. Ferreira Pires, and M. van Sinderen, “Cost and Benefits of Multiple Levels of Models in MDA Development”, *Proceedings of the 2nd European Workshop on Model Driven Architecture with an Emphasis on Methodologies and Transformations*, 2004, pp. 12-20.
6. J.P.A. Almeida, R. Dijkman, L. Ferreira Pires, D. Quartel, and M. van Sinderen, “Model Driven Design, Refinement and Transformation of Abstract Interactions”, *International Journal of Cooperative Information Systems*, vol. 15, no. 4, World Scientific, 2006, pp. 599-632.
7. G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services. Concepts, Architectures and Applications*, Springer, 2004.
8. P.A. Amaya Barbosa, C.F. Gonzalez Contreras, and J.M. Murillo Rodriguez, “MDA and Separation of Aspects: An Approach based on Multiple Views and Subject Oriented Design”, *Proceedings of the 6th International Workshop on Aspect Oriented Modelling*, 2005.
9. J. Amsden, T. Gardner, C. Griffin, and S. Iyengar, *Draft UML 1.4 Profile, for Automated Business Processes with a Mapping to BPEL 1.0, version 1.1*, IBM, June 2003.
10. G.R. Andrews and F.B. Schneider, “Concepts and Notations for Concurrent Programming”, *ACM Computing Survey*, vol. 15, no. 1, ACM Press, 1983, pp. 3-43.

11. Y. Aridor and D.B. Lange, "Agent Design Patterns: Elements of Agent Application Design", *Proceedings of the 2nd International Conference on Autonomous Agents*, ACM Press, 1998, pp. 108-115.
12. C. Atkinson and T. Kühne, "Aspect-Oriented Development with Stratified Frameworks", *IEEE Software*, vol. 20, no. 1, IEEE Computer Society Press, 2003, pp. 81-89.
13. C. Atkinson, T. Kühne, and C. Bunse, "Dimensions of Component-Based Development", *Proceedings of the Workshop on Object-Oriented Technology*, Lecture Notes in Computer Science, vol. 1743, Springer, 1999, pp. 185-186.
14. K. Baina, B. Benatallah, F. Casati, and F. Toumani, "Model-Driven Web Service Development", *Proceedings of the 16th International Conference of Advanced Information Systems Engineering (CAiSE'04)*, Lecture Notes in Computer Science, vol. 3084, Springer, 2004, pp. 290-306.
15. L. Baresi, R. Heckel, S. Thöne, and D. Varró, "Modeling and Validation of Service-Oriented Architectures: Application vs. Style", *Proceedings of the 4th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE 2003)*, ACM SIGSOFT Software Engineering Notes, vol. 28, no. 5, ACM Press, 2003, pp. 68-77.
16. A. Barros, M. Dumas, and A. ter Hofstede, "Service Interaction Patterns", *Proceedings of the 3rd International Conference on Business Process Management (BPM 2005)*, Lecture Notes in Computer Science, vol. 3649, Springer, 2005, pp. 302-318.
17. G. Baster, P. Konana, and J.E. Scott, "Business Components: A Case Study of Bankers Trust Australia Limited", *Communication of the ACM*, vol. 44, no. 5, ACM, 2001, pp. 92-98.
18. BEA Systems Inc., *WS-Callback Protocol (WS-Callback)*, version 0.91, 26 February 2003.
19. BEA Systems Inc. and IBM Corp., *BPELJ: BPEL for Java*, March 2004.
20. BEA Systems Inc., IBM Corp., Microsoft Corp., SAP AG, and Siebel Systems Inc., *Business Process Execution Language for Web Services*, version 1.1, 5 May 2003.
21. B. Benatallah, R.M. Dijkman, M. Dumas, and Z. Maamar, "Service Composition: Concepts, Techniques, Tools and Trends", Z. Stojanovic and A. Dahanayake (eds.), *Service-Oriented Software System Engineering: Challenges and Practices*, Idea Group Inc., 2005, pp.48-66.
22. J. Bezivin, S. Hammoudi, D. Lopes, and F. Jouault, "Applying MDA Approach to B2B Applications: A Road Map", *Proceedings of the Workshop on Model Driven Development (WMDD2004)*, Lecture Notes in Computer Science, vol. 3344, Springer, 2004, pp. 148-157.

23. J. Bezivin, S. Hammoudi, D. Lopes, and F. Jouault, "Applying MDA Approach for Web Service Platform", *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)*, IEEE Computer Society Press, 2004, pp. 58-70.
24. T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language LOTOS", *Computer Networks and ISDN Systems*, vol. 14, no. 1, Elsevier, 1987, pp. 25-59.
25. B. Bordbar and A. Staikopoulos, "On Behavioural Model Transformation in Web Services", *Proceedings of the 5th International Workshop on Conceptual Modelling Approaches for e-Business (eCOMO 2004),* Lecture Notes on Computer Science, vol. 3289, Springer, 2005, pp. 67-678.
26. S. Brahe, "BPM on Top of SOA: Experiences from Financial Industry", *Proceedings of the 5th International Conference on Business Process Management (BPM 2007)*, Lecture Notes on Computer Science, vol. 4714, Springer, 2007, pp. 96-111.
27. M. Broy, "Compositional Refinement of Interactive Systems", *Journal of the ACM*, vol. 44, no. 6, ACM Press, 1997, pp. 850-891.
28. M.V. Cengarle and A. Knapp, "UML 2. 0 Interactions: Semantics and Refinement," *Proceedings of the 3rd International Workshop Critical Systems Development with UML (CSDUML '04)*, 2004, pp. 85-99.
29. N. Chapin and S.P. Denniston, "Characteristics of a Structured Program", *ACM SIGPLAN Notices*, vol. 13, no. 5, ACM Press, 1978, pp. 36-45.
30. T.R. Dean and J.R. Cordy, "A Syntactic Theory of Software Architecture", *IEEE Transaction on Software Engineering*, vol. 21, no. 4, IEEE Computer Society Press, 1995, pp. 302-313.
31. R. Dijkman, *Consistency in Multi-Viewpoint Architectural Design*, Ph.D. thesis, University of Twente, The Netherlands, 2006.
32. R. Dijkman, *Choreography-Based Design of Business Collaborations*, BETA Working Paper WP-181, Eindhoven University of Technology, The Netherlands, 2006.
33. R. Dijkman and M. Dumas, "Service-Oriented Design: a Multi-Viewpoint Approach", *International Journal of Cooperative Information Systems*, vol. 13, no. 4, World Scientific, 2004, pp. 337-368.
34. R. Dijkman, T. Dirgahayu, and D.A.C. Quartel, "Towards Advanced Interaction Design Concepts", *Proceedings of the 10th IEEE International EDOC Enterprise Computing Conference (EDOC 2006)*, IEEE Computer Society Press, 2006, pp. 331-344.
35. T. Dirgahayu, *Model Driven Engineering of Web Service Compositions: A Transformation from ISDL to BPEL*, M.Sc. Thesis, University of Twente, The Netherlands, 2005.

36. T. Dirgahayu, D. Quartel, and M. van Sinderen, “”Development of Transformations from Business Process Models to Implementation by Reuse”, *Proceedings of the 3rd International Workshop on Model-Driven Enterprise Information System (MDEIS 2007)*, INSTICC Press, 2007, pp. 41-50.
37. T. Dirgahayu, D. Quartel, and M. van Sinderen, “An Abstract Interaction Concept for Designing Interaction Behaviour of Service Compositions”, *Proceedings of the 4th International Conference on Interoperability for Enterprise Software and Applications (I-ESA’08)*, Enterprise Interoperability III, Springer, 2008, pp. 261-273.
38. T. Dirgahayu, D. Quartel, and M. van Sinderen, “Designing Interaction Behaviour of Service-Oriented Enterprise Application Integrations”, *Proceedings of the 2008 ACM Symposium of Applied Computing (SAC 2008)*, ACM, pp. 1048-1054.
39. F. Drewes, P. Knirsch, H.-J. Kreowski, and S. Kuske, “Graph Transformation Modules and Their Composition”, *Proceedings of International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE’99)*, Lecture Notes in Computer Science, vol. 1779, Springer, 2000, pp. 111-119.
40. Eclipse Foundation, *Eclipse Modeling Framework Project (EMF)*, http:// www. eclipse.org/modeling/emf/
41. C. Emig, J. Weisser, and S. Abeck, “Development of SOA-Based Software Systems – an Evolutionary Programming Approach”, *Proceedings of Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT-ICIW’06)*, 2006, pp. 182-187.
42. N. Erasala, D.C. Yen, and T.M. Rajkumar, “Enterprise Application Integration in the Electronic Commerce World”, *Computer Standards and Interfaces*, vol 25, no. 2, Elsevier, 2003, pp. 69-82.
43. A. Farias and M. Sudholt, “On Components with Explicit Protocols Satisfying a Notion of Correctness by Construction”, *Proceedings of OTM 2002 Confederated International Conferences: CoopIS, DOA, and ODBASE*, Lecture Notes in Computer Science, vol. 2519, Springer, 2002, pp. 995-1012.
44. L. Ferreira Pires, *Architectural Notes: a Framework for Distributed Systems Development*, Ph.D. thesis, University of Twente, The Netherlands, 1994.
45. R. Fehling, “A Concept of Hierarchical Petri Nets with Building Blocks”, *Proceedings of the 12th International Conference on Applications and Theory of Petri Nets*, Lecture Notes in Computer Science, vol. 674, Springer, 1993, pp. 148-168.

46. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1995.
47. J. Gray and A. Reuter, *Transaction Processing Concepts and Techniques*, Morgan Kaufmann Publisher, 1993.
48. R. Grønmo, D. Skogan, I. Solheim, and J. Oldevik, "Model-driven Web Services Development", *Proceedings of 2004 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'04)*, 2004, pp. 42-45.
49. J. Grudin, "Interactive Systems: Bridging the Gaps between Developers and Users", *Computer*, vol. 24, no. 4, IEEE Computer Society Press, 1991, pp. 59-69.
50. S. Hallsteinsen and M. Paci (eds.), *Experiences in Software Evolution and Reuse. Twelve Real World Projects,* Springer, 1997.
51. R. Hamadi and B. Benatallah, "A Petri Net-Based Model for Web Service Composition", *Proceedings of the 14th Australasian Database Conference (ADC 2003)*, Australian Computer Society Inc., 2003, pp. 191-200.
52. D. Hirsch, S. Uchitel, and D. Yankelevich, "Towards a Periodic Table of Connectors", *Proceedings of the 3rd International Conference on Coordination Languages and Models (COORDINATION'99)*, Lecture Notes in Computer Science, vol. 1594, Springer, 1999, pp. 418-424.
53. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, 1985.
54. G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley Professional, 2004.
55. M.N. Huhns and M.P. Singh, "Service-Oriented Computing: Key Concepts and Principles", *IEEE Internet Computing*, vol. 9, no. 1, IEEE Computer Society Press, 2005, pp. 75-81.
56. IIBA, *A Guide to Business Analysis Body of Knowledge version 1.6*, 2006.
57. ISDL Home, *Grizzle Distributions*, http://isdl.ctit.utwente.nl/tools/grizzle/
58. ISO/IEC, *Information Technology – Open Distributed Processing – Reference Model*, International Standard ISO/IEC 10746.1-4, 1998.
59. ISO, *Information Processing Systems – Open System Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, ISO/IEC 8807, 1989.
60. M. Jackson, *Software Requirements & Specifications – A Lexicon of Practice, Principles and Prejudices*. ACM Press, 1995.
61. O. Kath, A. Blazarenas, M. Born, K.-P. Eckert, M. Funabashi, and C. Hirai, "Towards Executable Models: Transforming EDOC Behavior

Models to CORBA and BPEL", *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)*, IEEE Computer Society Press, 2004, pp. 267-274.

62. M. Keil and E. Carmel, "Customer-Developer Links in Software Development", *Communication of the ACM*, vol. 38, no. 5, ACM, 1995, pp. 33-44.
63. J. Koehler, R. Hauser, S. Kapoor., F.Y. Wu, and S. Kumaran, "A Model Driven Transformation Method", *Proceedings of the 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003)*, IEEE Computer Society Press, 2003, pp. 186-197.
64. B. Korherr and B. List, "Extending UML 2 Activity Diagram with Business Process Goals and Performance Measures and the Mapping to BPEL", *Proceedings of the 2nd International Workshop on Best Practices of UML (BP-UML 2006)*, Lecture Notes on Computer Science, vol. 4231, 2006, pp. -18.
65. G. Kramler, E. Kapsammer, W. Retschitzegger, and G. Kappel, "Towards Using UML 2 for Modelling Web Service Collaboration Protocols", *Proceedings of the 1st International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA'05)*, D. Konstantas, J.-P. Bourrières, M. Léonard, and N. Boudjlida (eds.), Interoperability of Enterprise Software and Applications, Springer, 2006, pp. 227-238.
66. V. Kulkarni and S. Reddy, "Separation of Concerns in Model-Driven Development", *IEEE Software*, vol. 20, no. 5, IEEE Computer Society Press, 2003, pp. 64-69.
67. I. Kurtev, *Adaptability of Model Transformation*, PhD. thesis, University of Twente, The Netherlands, 2005.
68. K.-K. Lau, L. Ling, V. Ukis, and P.V. Elizondo, "Composite Connectors for Composing Software Components", *Revised selected papers of the 6th International Symposium on Software Compositions (SC 2007)*, Lecture Notes on Computer Science, vol. 4829, Springer, 2007, pp. 266-280.
69. F. Leymann, *Web Services Flow Language (WSFL 1.0)*, IBM Corp., 2001.
70. J. Lind, "Patterns in Agent-Oriented Software Engineering", *Agent-Oriented Software Engineering III, Revised papers and invited contributions of the 3rd International Workshop on Agent-Oriented Software Engineering*, Lecture Notes on Computer Science, vol. 2585, Springer, 2002, pp. 47-58.
71. B. List and B. Korherr, "A UML2 Profile for Business Process Modelling", *Proceedings of the Workshop on Metamodelling and Model Driven Development*, Lecture Notes on Computer Science, vol. 3770, Springer, 2005, pp. 85-96.

72. D.C. Luckham and J. Vera, "An Event-Based Architectural Definition Language", *IEEE Transactions on Software Engineering*, vol. 21, no. 9, IEEE Computer Society Press, 1995, pp. 717-734.
73. J. Magee and J. Kramer, "Dynamic Structure in Software Architecture", *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 6, ACM Press, 1996, pp. 3-14.
74. I. Manolescu, M. Brambilla, S. Ceri, S. Comai, and P. Fraternali, "Model-Driven Design and Deployment of Service-Enabled Web Applications", *ACM Transactions on Internet Technology*, vol. 5, no. 3, ACM Press, 2005, pp. 439-479.
75. R.P. McAfee and J. McMillan, "Auctions and Bidding", *Journal of Economic Literature*, vol. 25 no. 2, American Economic Association, 1987, pp. 699-738.
76. M. Mecella, F.P. Presicce, and B. Pernici, "Modeling E-service Orchestration through Petri Nets", *Proceedings of the 3rd International Workshop on Technologies for E-Services (TES 2002)*, Lecture Notes on Computer Science, vol. 2444, Springer, 2002, pp. 109-134.
77. B. Medjahed, B. Benatallah, A. Bougouettaya, A.H.H. Ngu, and A.K. Elmagarmid, "Business-to-Business Interactions: Issues and Enabling Technologies", *The VLDB Journal*, vol. 12, no. 1, Springer, 2003, pp. 59-85.
78. N.R. Mehta, N. Medvidovic, and S. Phadke, "Towards a Taxonomy of Software Connectors", *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, ACM Press, 2000, pp. 178-187.
79. T. Mens and P. van Gorp, "A Taxonomy of Model Transformation", *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)*, Electronic Notes in Theoretical Computer Science, vol. 152, Elsevier, 2006, pp. 125-142.
80. Microsoft Corp., *.NET Framework Conceptual Overview*, http://msdn. microsoft.com/en-us/library/zw4w595w.aspx
81. D.E. Millard, Y. Howard, E.-R. Jam, S. Chennupati, H.C. Davis, L. Gilbert, and G.B. Wills, "FREMA Method for Describing Web Services in a Service-Oriented Architecture", *Technical Report ECSTR-IAM06-002*, University of Southampton, UK, 2006.
82. R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, vol 92, Springer-Verlag, 1980.
83. M. Moriconi, X. Qian, and R.A. Riemenschneider, "Correct Architecture Refinement", *IEEE Transactions on Software Engineering*, vol. 21, no. 4, IEEE Computer Society Press, 1995, pp. 356-372.
84. OASIS, *Web Services Base Notification 1.3 (WS-BaseNotification)*, OASIS Standard, 1 October 2006.

85. OASIS, *Web Services Brokered Notification 1.3 (WS-BrokeredNotification)*, OASIS Standard, 1 October 2006.
86. OASIS, *Web Services Business Process Execution Language version 2.0*, OASIS Standard, 11 April 2007.
87. OMG, *Business Process Modeling Notation (BPMN) version 1.2*, formal/ 2009-01-03, 2009.
88. OMG. *Business Process Modeling Notation (BPMN) FTF Beta 1 for version 2.0*, dtc/2009-08-14, 2009.
89. OMG, *Common Object Request Broker Architecture: Core Specification version 3.0.3*, formal/04-03-12, 2004.
90. OMG, *MDA Guide version 1.0.1*, omg/2003-06-01, 2003.
91. OMG, *Model Driven Architecture (MDA)*, ormsc/2001-07-01, 2001.
92. OMG, *Service Oriented Architecture Modeling Language (SoaML) – Specification for the UML Profile and Metamodel for Services (UPMS)*, ptc/2009-04-01, 2009.
93. OMG, *UML Profile for CORBA Specification version 1.0*, formal/02-04-01, 2002.
94. OMG, *Unified Modelling Language Specification version 1.3*, 2000.
95. OMG, *Unified Modeling Language: Infrastructure version 2.1.1*, formal/ 07-02-03, 2007.
96. OMG, *Unified Modeling Language: Superstructure version 2.1.1*, formal/ 2007-02-03, 2007.
97. Oracle Corp., *Oracle BPEL Process Manager*, http://www.oracle.com/ technology/bpel
98. B. Orriens, J. Yang, and M.P. Papazoglou, "Model Driven Service Composition", *Proceedings of International Conference on Service Oriented Computing (ICSOC 2003)*, Lecture Notes in Computer Science, vol. 2910, Springer, 2003, pp. 75-90.
99. M.P. Papazoglou, *Web Services: Principle and Technology*, Pearson Education, 2008.
100. M.P. Papazoglou and D. Georgakopoulos, "Service-Oriented Computing", Communications of the ACM, vol. 46, no. 10, ACM Press, 2003, pp. 25-28.
101. O. Patrascoiu, "Mapping EDOC to Web Services using YATL", *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)*, IEEE Computer Society, 2004, pp. 286-297.
102. PayPal Inc., *PayPal*, http://www.paypal.com
103. C. Peltz, "Web Services Orchestration and Choreography", *IEEE Computer*, vol. 36, no. 8, IEEE Computer Society Press, 2003, pp. 46-52.
104. J.L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.

105. S. Pokraev, *Model-Driven Semantic Integration of Service-Oriented Applications*, Ph.D thesis, University of Twente, The Netherlands, 2009.

106. S. Pokraev, D.A.C. Quartel, M.W.A. Steen, A. Wombacher, and M. Reichert, "Business Level Service-Oriented Enterprise Application Integration", *Proceedings of the 3rd International Conference on Interoperability for Enterprise Software and Applications (I-ESA 2007)*, Springer, 2005, pp. 507-518.

107. D. Quartel, *Action Relations. Basic Design Concepts for Behaviour Modelling and Refinement*, Ph.D. thesis, University of Twente, The Netherlands, 1998.

108. D. Quartel, R. Dijkman, and M. van Sinderen, "Methodological Support for Service-Oriented Design with ISDL", *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC 2004)*, ACM Press, 2004, pp. 1-10.

109. D. Quartel, T. Dirgahayu, and M. van Sinderen, "Model-Driven Design, Simulation and Implementation of Service Compositions in COSMO", *International Journal of Business Process Integration and Management*, vol. 4, no.1, Inderscience, 2009, pp. 18-34.

110. D. Quartel, L. Ferreira Pires, and M. van Sinderen, "On Architectural Support for Behaviour Refinement in Distributed Systems Design", *Journal of Integrated Design and Process Science*, vol. 6, no. 1, Society for Design and Process Science, 2002, pp.1-30.

111. D.A.C. Quartel, L. Ferreira Pires, M.J. van Sinderen, H.M. Franken, and C.A. Vissers, "On the Role of Basic Design Concepts in Behaviour Structuring", *Computer Networks and ISDN Systems*, vol. 29, no. 4, Elsevier, 1997, pp. 413-436.

112. D.A.C. Quartel, S. Pokraev, T. Dirgahayu, R. Mantovaneli Pessoa, M.W.A. Steen, and M. van Sinderen, "Model-Driven Development of Mediation for Business Services using COSMO", *Enterprise Information Systems*, vol. 3, no. 3, Taylor & Francis, 2009, pp. 319-345.

113. D.A.C. Quartel, M.W.A. Steen, S. Pokraev, and M.J. van Sinderen, "COSMO: A Conceptual Framework for Service Modelling and Refinement", *Information Systems Frontiers*, vol. 9, no. 2-3, Springer, 2007, pp. 225-244.

114. M. Shaw, "Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status", *Selected papers from the Workshop on Studies of Software Design*, Lecture Notes in Computer Science, vol. 1078, Springer, 1993, pp. 17-32.

115. M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to

Support Them", *IEEE Transactions on Software Engineering*, vol. 21, no. 4, IEEE Computer Society Press, 1995, pp. 314-335.

116. M. Shaw, R. DeLine, and G. Zelesnik, "Abstraction and Implementation for Architectural Connections", *Proceedings of the 3rd International Conference on Configurable Distributed Systems (ICCDS'96)*, 1996, pp. 2-10.

117. D. Skogan, R. Grønmo, and I. Solheim, "Web Service Composition in UML", *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)*, IEEE Computer Society Press, 2004, pp. 47-57.

118. A. Solberg, D. Simmonds, R. Reddy, S. Ghosh, and R. France, "Using Aspect Oriented Techniques to Support Separation of Concerns in Model Driven Development", *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, IEEE Computer Society Press, 2005, pp. 121-126.

119. N. Soundarajan, "Refining Interactions in a Distributed System", *Proceedings of the 1st International Workshop on Formal Approaches to Agent-Based Systems (FAABS 2000)*, Lecture Notes in Computer Science, vol. 1871, Springer, 2000, pp. 209-221.

120. T. Specht, J. Drawehn, M. Thranert, and S. Kuhne, "Modeling Cooperative Business Processes and Transformation to a Service Oriented Architecture", *Proceedings of the 7th IEEE International Conference on E-Commerce Technology*, IEEE Computer Society Press, 2005, pp. 249-256.

121. Sun Microsystems Inc., *Java Platform, Enterprise Edition 5 (Java EE 5)*, http://java.sun.com/javaee/technologies/javaee5.jsp

122. A. Sutcliffe, *The Domain Theory: Patterns for Knowledge and Software Reuse*. Lawrence Erlbaum Associates, Inc., 2002.

123. SWSC, *Semantic Web Service Challenge: Evaluating Semantic Web Services Mediation, Choreography and Discovery,* http://sws-challenge.org

124. S. Thatte, *XLANG: Web Services for Business Process Design*, Microsoft Corp., 2001.

125. S. Thöne, R. Depke, and G. Engels, "Process-Oriented, Flexible Composition of Web Services with UML", *Proceedings of the 3rd International Joint Workshop on Conceptual Modeling Approaches for E-Business (eCOMO 2002)*, Lecture Notes in Computer Science, vol. 2784, Springer, 2003, pp. 390-401.

126. E. Truyen, B.N. Jorgensen, W. Joosen, and P. Verbaeten, "On Interaction Refinement in Middleware", *Proceedings of the 5th International Workshop on Component-Oriented Programming (WCOP 2000)*, 2000, pp. 56-62.

127. M. Turner, D. Budgen, and P. Brereton, “Turning Software into a Service”, *IEEE Computer*, vol. 36, no. 10, IEEE Computer Society Press, 2003, pp. 38-44.
128. W.M.P. van der Aalst, M. Dumas, and A.H.M. ter Hofstede, “Web Service Composition Languages: Old Wine in New Bottles?”, *Proceedings of the 29th EUROMICRO Conference (EUROMICRO’03)*, 2003, pp. 298-305.
129. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, “Workflow Patterns”, *Distributed and Parallel Databases*, vol. 14, no. 3, Springer, 2003, pp. 5-51.
130. M. van Sinderen, *On the Design of Application Protocols*, Ph.D. thesis, University of Twente, The Netherlands, 1995.
131. C.A. Vissers, L. Ferreira Pires, D.A.C. Quartel, and M.J. van Sinderen, *The Architectural Design of Distributed Systems*, University of Twente, The Netherlands, 2002.
132. W3C, *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, W3C Recommendation, 26 November 2008.
133. W3C, *Web Services Architecture*, W3C Working Group Note, 11 February 2004.
134. W3C, *Web Services Architecture Usage Scenario*, W3C Working Group Note, 11 February 2004.
135. W3C, *Web Services Choreography Description Language version 1.0*, W3C Candidate Recommendation, 9 November 2005.
136. W3C, *Web Services Choreography Interfaces (WSCI) 1.0*, W3C Note, 8 August 2002.
137. W3C, *Web Services Conversation Language (WSCL) 1.0*, W3C Note, 14 March 2002.
138. W3C, *Web Services Description Language (WSDL) 1.1*, W3C Note, 15 March 2001.
139. W3C, *Web Services Description Language (WSDL) version 2.0 Part 0: Primer*, W3C Recommendation, 26 June 2007.
140. W3C, *Web Services Polling (WS-Polling)*, W3C Member Submission, 26 October 2005.
141. W3C, *XML Path Language (XPath) version 1.0*, W3C Recommendation, 16 November 1999.
142. W3C, *XML Schema Part 0: Primer second edition*, W3C Recommendation, 28 October 2004.
143. S.A. White, “Using BPMN to Model a BPEL Process”, *BPTrends*, vol. 3, no. 3, Business Process Trends, 2005, pp. 1-18.
144. M.H. Williams, “Generating Structured Flow Diagrams: The Nature of Unstructurness”, *The Computer Journal*, vol. 20, no. 1, British Computer Society, 1977, pp. 45-50.

145. J.M. Zaha, M. Dumas, A. ter Hofstede, A. Barros, and G. Decker, "Service Interaction Modeling: Bridging Global and Local Views", *Proceedings of the 10th IEEE International EDOC Enterprise Computing Conference (EDOC 2006)*, IEEE Computer Society Press, 2006, pp. 45-55.

146. J.M. Zaha, A. Barros, M. Dumas, and A. ter Hofstede, “Let’s Dance: A Language for Service Behavior Modeling”, *Proceedings of OTM 2006 Confederated International Conferences: CoopIS, DOA, GADA, and ODBASE*, Lecture Notes in Computer Science, vol. 4275, Springer, 2006, pp. 145-162.

Publications by the author

During the development of this thesis, the author has published various parts of his work in the following papers (listed in reverse chronological order):

- T. Dirgahayu, D. Quartel, and M. van Sinderen, "Interaction Refinement in the Design of Business Collaborations", *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC 2010)*, ACM, pp. 86-93, 22 - 26 March 2010, Sierre, Switzerland. ISBN: 978-1-60558-638-0. Best paper award for the Engineering theme.
- T. Dirgahayu, D. Quartel, and M. van Sinderen, "Abstractions of Interaction Mechanisms", *Proceedings of the 13th IEEE International EDOC Enterprise Computing Conference (EDOC 2009)*, IEEE Computer Society, pp. 173-182, 31 August - 4 September 2009, Auckland, New Zealand. ISBN: 978-0-7695-3785-6, ISSN: 1541-7719.
- D.A.C. Quartel, S. Pokraev, T. Dirgahayu, R. Mantovanelli Pessoa, M.W.A. Steen, and M. van Sinderen, "Model-Driven Development of Mediation for Business Services Using COSMO", *Enterprise Information Systems*, vol. 3, no. 3, pp. 319-345, August 2009. ISSN: 1751-7583.
- D. Quartel, T. Dirgahayu, and M. van Sinderen, "Model-Driven Design, Simulation and Implementation of Service Compositions in COSMO", *International Journal of Business Process Integration and Management*, vol. 4, no. 1, pp. 18-34, 2009. ISSN: 1741-8763.
- D.A.C. Quartel, S. Pokraev, T. Dirgahayu, R. Mantovaneli Pessoa and M. van Sinderen, "Model-driven Service Integration using the COSMO Framework", *Semantic Web Services Challenge: Proceedings of the 2008 Workshops*, Stanford Logical Group Technical Reports (LG-2009-01), pp. 77-88, 26 October 2008, Karlsruhe, Germany.
- T. Dirgahayu, D. Quartel, and M. van Sinderen, "Transforming Internal Activities of Business Process Models to Services Compositions", *Joint Proceedings of IWUC, MDEIS and TCoB 2008 - 4th International Workshop on Model-Driven Enterprise Information Systems*, INSTICC Press, pp. 56-63, 12-13 June 2008, Barcelona, Spain. ISBN: 978-989-8111-49-4.
- T. Dirgahayu, D. Quartel, and M. van Sinderen, "An Abstract Interaction Concept for Designing Interaction Behaviour of Service Compositions", *Enterprise Interoperability III – Proceedings of the 4th*

International Conference on Interoperability for Enterprise Software and Applications (I-ESA '08), Springer, pp. 261-273, 25-28 March 2008, Berlin, Germany. ISBN: 978-1-84800-220-3.

- T. Dirgahayu, D. Quartel, and M. van Sinderen, “Designing Interaction Behaviour of Service-Oriented Enterprise Application Integrations”, *Proceedings of the 2008 ACM Symposium of Applied Computing (SAC 2008)*, ACM, pp. 1048-1054, 16-20 March 2008, Fortaleza, Brazil. ISBN: 978-1-59593-753-7.
- R.M. Dijkman, T. Dirgahayu, and D.A.C. Quartel, “The Adequacy of Languages for Representing Interaction Mechanisms”, *Information Systems Frontiers*, vol. 9, no. 4, Springer, pp. 359-373, September 2007. ISSN: 1387-3326.
- T. Dirgahayu, D. Quartel, and M. van Sinderen, “Development of Transformations from Business Process Models to Implementations by Reuse”, *Proceedings of MDEIS 2007 - 3rd International Workshop on Model-Driven Enterprise Information Systems*, INSTICC Press, pp. 41-50, 12 June 2007, Funchal, Portugal. ISBN: 978-989-8111-00-5.
- R.M. Dijkman, T. Dirgahayu, and D.A.C. Quartel, “Towards Advanced Interaction Design Concepts”, *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06)*, IEEE Computer Society Press, pp. 331-344, 16-20 October 2006. Hong Kong. ISBN: 0-7695-2558-X.

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
[2009-30] Marcin Zukowski (CWI) Balancing vectorized query execution with bandwidth-optimized storage
[2009-31] Sofiya Katrenko (UVA) A Closer Look at Learning Relations from Text
[2009-32] Rik Farenhorst (VU) and Remco de Boer (VU) Architectural Knowledge Management: Supporting Architects and Auditors
[2009-33] Khiet Truong (UT) How Does Real Affect Affect Affect Recognition In Speech?
[2009-34] Inge van de Weerd (UU) Advancing in Software Product Management: An Incremental Method Engineering Approach
[2009-35] Wouter Koelewijn (UL) Privacy en Politiegegevens; Over geautomatiseerde normatieve informatie-uitwisseling
[2009-36] Marco Kalz (OUN) Placement Support for Learners in Learning Networks
[2009-37] Hendrik Drachsler (OUN) Navigation Support for Learners in Informal Learning Networks
[2009-38] Riina Vuorikari (OU) Tags and self-organisation: a metadata ecology for learning resources in a multilingual context

[2009-39] Christian Stahl (TUE, Humboldt-Universitaet zu Berlin) Service Substitution -- A Behavioral Approach Based on Petri Nets
[2009-40] Stephan Raaijmakers (UvT) Multinomial Language Learning: Investigations into the Geometry of Language
[2009-41] Igor Berezhnyy (UvT) Digital Analysis of Paintings
[2009-42] Toine Bogers (UvT) Recommender Systems for Social Bookmarking
[2009-43] Virginia Nunes Leal Franqueira (UT) Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients
[2009-44] Roberto Santana Tapia (UT) Assessing Business-IT Alignment in Networked Organizations
[2009-45] Jilles Vreeken (UU) Making Pattern Mining Useful
[2009-46] Loredana Afanasiev (UvA) Querying XML: Benchmarks and Recursion

2010
[2010-01] Matthijs van Leeuwen (UU) Patterns that Matter
[2010-02] Ingo Wassink (UT) Work flows in Life Science
[2010-03] Joost Geurts (CWI) A Document Engineering Model and Processing Framework for Multimedia documents
[2010-04] Olga Kulyk (UT) Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments
[2010-05] Claudia Hauff (UT) Predicting the Effectiveness of Queries and Retrieval Systems
[2010-06] Sander Bakkes (UvT) Rapid Adaptation of Video Game AI
[2010-07] Wim Fikkert (UT) A Gesture interaction at a Distance
[2010-08] Krzysztof Siewicz (UL) Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments
[2010-09] Hugo Kielman (UL) A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging
[2010-10] Rebecca Ong (UL) Mobile Communication and Protection of Children
[2010-11] Adriaan Ter Mors (TUD) The world according to MARP: Multi-Agent Route Planning
[2010-12] Susan van den Braak (UU) Sensemaking software for crime analysis
[2010-13] Gianluigi Folino (RUN) High Performance Data Mining using Bio-inspired techniques
[2010-14] Sander van Splunter (VU) Automated Web Service Reconfiguration
[2010-15] Lianne Bodenstaff (UT) Managing Dependency Relations in Inter-Organizational Models
[2010-16] Sicco Verwer (TUD) Efficient Identification of Timed Automata, theory and practice
[2010-17] Spyros Kotoulas (VU) Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications

[2010-18] Charlotte Gerritsen (VU) Caught in the Act: Investigating Crime by Agent-Based Simulation
[2010-19] Henriette Cramer (UvA) People's Responses to Autonomous and Adaptive Systems
[2010-20] Ivo Swartjes (UT) Whose Story Is It Anyway? How Improv Informs Agency and Authorship of Emergent Narrative
[2010-21] Harold van Heerde (UT) Privacy-aware data management by means of data degradation
[2010-22] Michiel Hildebrand (CWI) End-user Support for Access to Heterogeneous Linked Data
[2010-23] Bas Steunebrink (UU) The Logical Structure of Emotions
[2010-24] Dmytro Tykhonov () Designing Generic and Efficient Negotiation Strategies
[2010-25] Zulfiqar Ali Memon (VU) Modelling Human-Awareness for Ambient Agents: A Human Mindreading Perspective
[2010-26] Ying Zhang (CWI) XRPC: Efficient Distributed Query Processing on Heterogeneous XQuery Engines
[2010-27] Marten Voulon (UL) Automatisch contracteren
[2010-28] Arne Koopman (UU) Characteristic Relational Patterns
[2010-29] Stratos Idreos(CWI) Database Cracking: Towards Auto-tuning Database Kernels
[2010-30] Marieke van Erp (UvT) Accessing Natural History - Discoveries in data cleaning, structuring, and retrieval
[2010-31] Victor de Boer (UVA) Ontology Enrichment from Heterogeneous Sources on the Web
[2010-32] Marcel Hiel (UvT) An Adaptive Service Oriented Architecture: Automatically solving Interoperability Problems
[2010-33] Robin Aly (UT) Modeling Representation Uncertainty in Concept-Based Multimedia Retrieval
[2010-34] Teduh Dirgahayu (UT) Interaction Design in Service Compositions