CONSISTENCY IN MULTI-VIEWPOINT ARCHITECTURAL DESIGN

REMCO DIJKMAN

Consistency in Multi-Viewpoint Architectural Design

Remco M. Dijkman

Enschede, The Netherlands, 2006

CTIT Ph.D.-Thesis Series Number 06-80
Telematica Instituut Fundamental Research Series Number TI/FRS/17

Cover Design: Studio Oude Vrielink, Losser and Jos Hendrix, Groningen
Book Design: Lidwien van de Wijngaert and Henri ter Hofte
Printing: Universal Press, Veenendaal

Graduation Committee:

Chair, secretary:	prof. dr. ir. A.J. Mouthaan (Universiteit Twente)
Promotor:	prof. dr. ir. C.A. Vissers (Universiteit Twente / Telematica Instituut)
Assistant promotor:	dr. ir. D.A.C. Quartel (Universiteit Twente)
Members:	prof. dr. ir. W.M.P. van der Aalst (Technische Universiteit Eindhoven)
	prof. dr. ir. S.M.M. Joosten (Open Universiteit Nederland / Ordina)
	prof. dr. P.F. Linington (University of Kent)
	dr. ir. M.J. van Sinderen (Universiteit Twente)
	prof. dr. R.J. Wieringa (Universiteit Twente)

ISBN 90-75176-80-5
ISSN 1381-3617 (CTIT Ph.D.-Thesis Series Number 06-80)
ISSN 1388-1795 (Telematica Instituut Fundamental Research Series Number 17)

Centre for Telematics and Information Technology,
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.
Telephone: +31 (0)53 489 8031; Fax: +31 (0)53 489 1070

CONSISTENCY IN MULTI-VIEWPOINT ARCHITECTURAL DESIGN

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. W.H.M. Zijm,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 3 februari 2006 om 16.45 uur

door
Remco Matthijs Dijkman
geboren op 30 december 1976
te Amsterdam

Dit proefschrift is goedgekeurd door:
prof. dr. ir. C.A. Vissers (promotor)
dr. ir. D.A.C. Quartel (assistent-promotor)

Abstract

This thesis presents a framework that aids in preserving consistency in multi-viewpoint designs. In a multi-viewpoint design each stakeholder constructs his own design part. We call each stakeholder's design part the *view* of that stakeholder. To construct his view, a stakeholder has a *viewpoint*. This viewpoint defines the design concepts, the notation and the tool support that the stakeholder uses.

The framework presented in this thesis focuses on *architectural* multi-viewpoint design of *distributed systems*.

A *distributed system* is a system of which the parts execute on different physical system nodes. Interaction between the system parts plays an important role in such systems. An example of a distributed system is a mobile communication network. In such a network, the parts of the system execute on e.g. the mobile telephones of the clients, the desktops of the employees of the network operator and the mobile access points.

Architectural design is the area of design that focuses on higher levels of abstraction in the design process. The lowest level of abstraction that we consider is the level at which the system parts correspond to parts that can be deployed on communication middleware.

Using our framework, consistency is preserved through inter-viewpoint relations and consistency rules that must be specified by the stakeholders. The stakeholders use inter-viewpoint relations to specify how one view relates to another and they use consistency rules to specify what rules must at least be satisfied in a consistent design.

To aid in preserving consistency, our framework defines:

- a common set of basic design concepts;
- pre-defined inter-viewpoint relations;
- pre-defined consistency rules;
- a language to represent inter-viewpoint relations and consistency rules.

The basic design concepts that the framework defines have been adopted from earlier work. These concepts were developed by carefully examining

the area of distributed systems design. Using our framework, viewpoint-specific design concepts must be defined as compositions or specializations of these basic concepts. Hence, the basic concepts form a common vocabulary that the different stakeholders can use to understand each other's designs.

The framework pre-defines inter-viewpoint relations that can be re-used to specify how one view relates to another. The two main types of inter-viewpoint relations that it pre-defines are: refinement relations and overlap relations. Refinement relations exist between views that (partly) consider the same design concerns at different levels of abstraction. Overlap relations exist between views that (partly) consider the same design concerns at the same level of abstraction. We derived the pre-defined relations by examining existing frameworks for multi-viewpoint design and extracting frequently occurring relations between viewpoints in these frameworks.

If a pre-defined inter-viewpoint relation exists between two views, this implies that certain consistency rules must be satisfied. Specifically, if two views have a refinement relation, this implies that one must preserve the system properties specified by the other. If two views have an overlap relation, this implies that the two views must be equivalent with respect to the overlap that they have. Our framework pre-defines consistency rules that can be re-used to verify these properties.

We define an architecture for tool-support to aid in specifying view relations and consistency rules and to check whether the specified consistency rules hold. The architecture contains the pre-defined relations and consistency rules, such that they can be re-used.

As a case study for the framework we define adapted versions of the RM-ODP enterprise, computational and information viewpoints, using our framework. We define the concepts from these viewpoints as compositions of the basic concepts. Also, we define the relations between views from these viewpoints, as well as the corresponding consistency rules, using the relations and consistency rules that are pre-defined by the framework. The results of the case study support the claim that our framework aids in preserving consistency in multi-viewpoint designs.

Acknowledgements

This thesis marks the end of a four-year period, during which I met many people that have contributed to my development as a researcher. Some of these people I want to mention here especially.

First of all, I want to thank Chris, Dick and Marten for their supervision and especially Dick, for reading, and re-reading, my thesis several times. Also, I want to thank Luís, who was one of my supervisors during the first year.

I want to thank Stef, who inspired me to do research in the first place. The ideas on which this thesis is based are also for a large part due to him.

I want to thank the members of my promotion committee: professor Linington, professor Wieringa, professor van der Aalst (Wil) and the people mentioned above. I am deeply impressed by the work that each of you has done in his particular area of research. Therefore, I feel honoured that you agreed to be members of the committee.

I want to thank the people of the Business Process Management group at the Queensland University of Technology, and especially Marlon and Arthur, for providing me with the opportunity of working with them. The time with you was very productive and I learned a lot from you. Thanks for that.

To my colleagues and former colleagues, I realize that our group provides very pleasant working environment that will be difficult to find anywhere else. You are the ones to thank for that. Some of you I spend more time with, drinking coffee or something stronger or even doing some work together; I especially want to thank you.

Finally, I want to thank my family and friends and especially Martine for their company and support outside of the office.

Remco Dijkman
Hengelo, 4 January 2006

Contents

Telematica Instituut Fundamental Research Series

001 G. Henri ter Hofte, *Working apart together: Foundations for component groupware*

002 Peter J.H. Hinssen, *What difference does it make? The use of groupware in small groups*

003 Daan D. Velthausz, *Cost-effective network-based multimedia information retrieval*

004 Lidwien A.M.L. van de Wijngaert, *Matching media: information need and new media choice*

005 Roger H.J. Demkes, *COMET: A comprehensive methodology for supporting telematics investment decisions*

006 Olaf Tettero, *Intrinsic information security: Embedding security issues in the design process of telematics systems*

007 Marike Hettinga, *Understanding evolutionary use of groupware*

008 Aart T. van Halteren, *Towards an adaptable QoS aware middleware for distributed objects*

009 Maarten Wegdam, *Dynamic reconfiguration and load distribution in component middleware*

010 Ingrid J. Mulder, *Understanding designers, designing for understanding*

011 Robert J.J. Slagter, *Dynamic groupware services: modular design of tailorable groupware*

012 Nikolay K. Diakov, *Monitoring distributed object and component communication*

013 Cheun N. Chong, *Experiments in rights control expression and enforcement*

014 Cristian Hesselman, *Distribution of multimedia streams to mobile Internet users*

015 Giancarlo Guizzardi, *Ontological Foundations for Structural Conceptual Models*

016 Mark van Setten, *Supporting People in Finding Information: Hybrid Recommender Systems and Goal-based Structuring*

See also: http://www.telin.nl/publicaties/frs.htm

Chapter 1

Introduction

This chapter presents the background of our research towards a framework for distributed systems design from different, but related, viewpoints. The framework itself is presented in the remainder of this thesis. This chapter motivates the need for the framework. It presents the goals of the research that led towards the framework and the approach chosen to achieve these goals. Also, it describes the contributions that the research has with respect to other research in the area of distributed systems design. Finally, this chapter explains the structure of the remainder of the thesis.

1.1 Multi-Viewpoint Design

In any large-scale design, different people with different interests are involved. These people, or stakeholders as we call them, have their own way of looking at a system, for which they use their own modelling languages, techniques and tools. Informally, we call the way in which a stakeholder looks at a system: the *viewpoint* of that stakeholder.

Definition 1-1 Stakeholder

A stakeholder in a design is a person with a particular interest in that design and his own viewpoint on that design.

From his viewpoint, each stakeholder constructs his own design part, or *view*. However, because views are parts of the same *multi-viewpoint design*, we must maintain the consistency between the different view.

The task of maintaining the consistency between views in a multi-viewpoint design is especially complex, because the different stakeholders may use different terminology and different tools to construct their views. The use of a different terminology presents us with a communication problem in maintaining the consistency in a multi-viewpoint design. This problem manifests itself in different stakeholders using the same terms to mean

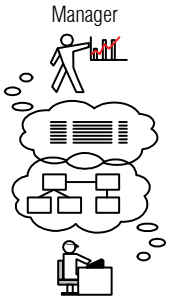


Figure 1-1 Example of a multi-viewpoint Design with Two Stakeholders

different things and using different terms to mean the same thing. The use of different tools presents us with a technology problem when maintaining the consistency in a multi-viewpoint design. This problem lies therein that, if we want to support consistency checks between different tools, we must be able to extract the models from these tools and relate them in a uniform way.

The framework for multi-viewpoint design that we propose aids in maintaining the consistency between views in a multi-viewpoint design. Especially, it addresses the communication and technology problems outlined above. The framework focuses on maintaining the consistency in the design of distributed systems that support administrative business processes.

Example 1-1 A Multi-Viewpoint Design

Figure 1-1 illustrates a multi-viewpoint design, in which two stakeholders are involved: a manager and an application developer. A manager typically has, among other things, a financial interest in the design. This interest could be represented from a viewpoint that covers the difference in costs of performing business activities manually and automatically with the system under design. The manager could model his interest as a table. An application developer has an interest in the functional specifications of the applications that are part of the distributed system; these functional specifications can be represented from another viewpoint than the viewpoint of the manager. The application developer could model his interest as a UML class diagram. Although the manager and the application developer have different viewpoints, their viewpoints are related. For example, the costs of performing a business activity depend on the level of automated support of that activity by software applications.

1.2 Role of Design Concepts in Multi-Viewpoint Design

Stakeholders construct a design (from their viewpoint) by combining instances of design concepts.

Definition 1-2 Design Concept

A design concept is an abstraction of some common and essential property of distributed systems.

Hence, a design concept represents some element in the Universe of Discourse (UoD). An example of a concept is the 'remote invocation' concept, which is an abstraction of a communication pattern in which one entity sends a request to another entity and a related response is sent in the opposite direction, or an error in the underlying communication mechanism is notified to the requesting party. Another example of a concept is the 'component' concept, which is an abstraction of an entity that encapsulates certain functionality. It makes this functionality available to its environment via communication mechanisms, such as the remote invocation mechanism.

Hence, a design can be constructed from instances of the 'component' and 'remote invocation' concept.

Often, a set of concepts is implicitly agreed upon by some stakeholders as part of the design culture.

Definition 1-3 Design Culture

A design culture is an environment of methods, procedures, tools and skills (Vissers, Ferreira Pires, & van de Lagemaat, 1995).

We say that design concepts are *implicitly* agreed upon in a design culture if they are used, but not explicitly defined. We argue that, for various reasons, the set of design concepts used in a design culture should be made explicit and defined precisely. These reasons include that:

- the explicit definition of the set of design concepts helps to understand those concepts unambiguously. If a design concept is not explicitly agreed upon, different people may have a different interpretation of it. For example, some people may interpret the aforementioned 'component' concept as an Enterprise Java Bean that only has two interfaces, others may interpret it as a more generic software component that can have any number of interfaces and still others may interpret it as a part of some whole that is not necessarily a software system;
- precisely defined design concepts can be used for validation and verification. For most realistic designs, it is unfeasible to assess the validity or evaluate certain properties of that design by hand. Therefore, we use automated tools to support validation and verification. The use of automated tools implies that the concepts that we use in a design must be implemented and therefore that we need to develop a precise enough understanding of these concepts to implement them.

In prior work (Quartel, 1998; Quartel, Ferreira Pires, van Sinderen, Franken, & Vissers, 1997; van Sinderen, 1995; Ferreira Pires, 1994) we developed a set of basic design concepts that is aimed towards the design of distributed systems across different viewpoints (focusing on structural, behavioural and information aspects of such systems).

Definition 1-4 Basic Design Concept

A basic design concept is an abstraction of some common, essential and elementary *property of distributed systems.*

Basic design concepts form the most elementary building blocks for distributed systems design. Since the basic design concepts are aimed towards the design of distributed systems across different viewpoints, an important criterion for selecting them is their *general applicability* to these viewpoints. Moreover, because, in our approach, each available concept is either a basic concept or a composition of basic concepts, the basic concepts determine the expressive power of the set of all available concepts. Hence, the basic

concepts should be *complete* with respect to the viewpoints they aim to address. Observing the generality and completeness criteria, the basic concepts should be expressive enough to construct designs from the viewpoints they aim to address.

Although the basic concepts may be expressive enough to address the selected viewpoints, it may be hard for stakeholders to accept such a general and elementary set of concepts, because the concepts may not be intuitively clear to the stakeholders or may be hard to apply. There are two reasons for this. Firstly, each stakeholder focuses on a specific part of the overall design. Therefore, each stakeholder considers a subset of the properties that are addressed in the overall design and a subset of the basic concepts that address these properties. Hence, the stakeholder may be confused by the other concepts that address properties that he or she does not consider. Secondly, stakeholders do not (only) consider elementary system properties. They consider properties that are specializations of elementary properties, like labour costs and administrative costs, which are specializations of the general property costs. Also, stakeholders consider properties that are compositions of elementary properties, like business processes, which are compositions of activities and goals, and stakeholders consider properties that are derived from elementary properties, like overall costs of performing a business process which are derived from the costs of individual activities. Since these properties can be expressed as compositions of basic concepts, there are no concepts that match them in a one-to-one fashion. Hence, the stakeholder may be confused, because the properties that he or she addresses are not represented by the available concepts in a one-to-one fashion.

For these reasons we develop a set of concepts for each stakeholder, such that there *is* an intuitive match between the properties that that stakeholder considers and the set of concepts that the stakeholder can use. To maintain the relation between the stakeholder-specific (or viewpoint-specific) concepts and the basic concepts, we define the stakeholder-specific concepts as specializations of basic concepts or as composite concepts. We can do that, provided that our set of basic concepts is expressive enough to represent the properties that the stakeholder considers.

Definition 1-5 Composite Design Concept

A composite design concept is a composition of basic design concepts or other composite design concepts.

Example 1-2 Basic Concepts and Viewpoint-specific Composite Concepts

Our basic design concepts include an 'interaction' concept that represents the successful completion of an activity that is performed by two or more behaviours in collaboration. However, an application developer may use his own set of concepts, including concepts like the 'component' and 'remote interaction' concepts mentioned above. Based on the basic 'interaction' concept, we can define the 'remote interaction' concept as a composition of a

'request', an 'indication', a 'response', a 'confirm' and an 'error notification' basic interaction. The 'request' interaction represents a basic interaction between the requestor and the underlying communication layer, used to pass the remote invocation request to the communication layer. The 'indication' represents the indication of the request to the receiving party, while the 'error notification' represents the notification of an error in the communication layer. The 'response' and 'confirm' interactions represent the passing of the response from the receiving party to the communication layer and from the communication layer to the requestor, respectively. Similarly, we can define the 'component' concept as a specialization of the basic 'entity' concept that represents a carrier of behaviour.

1.3 Consistency in a Multi-Viewpoint Design

To maintain the consistency in a multi-viewpoint design, we must define the *relations* between the viewpoints and corresponding *rules* to verify the consistency between designs from these viewpoints. We use consistency rules as more precise versions of the viewpoint relations. Where the viewpoint relations only allow us to specify the relation between viewpoints, the corresponding consistency rules provide the techniques to verify whether that relation holds for designs from those viewpoints. For example, if a viewpoint has a refinement relation to another viewpoint, the corresponding consistency rules allow us to verify whether a design from the first viewpoint is a correct refinement of a design from the second.

We claim that a set of basic concepts can play an important role in maintaining consistency between the viewpoints, because:

1. all stakeholder specific concepts either are basic concepts or are defined as compositions of the basic concepts. Therefore, the basic concepts provide the different groups of stakeholders with a common basis, which they can use to relate their respective sets of concepts and thereby their designs;
2. design relations and consistency rules can be defined on the basic concepts (as they are defined on the basic concepts, we also call them basic relations and basic consistency rules). The basic design relations and their corresponding consistency rules are automatically inherited by concepts that are defined as compositions of the basic concepts. Therefore, they can then be re-used to verify the consistency between different sets of viewpoints.

For these reasons we propose the use of a set of basic concepts. Figure 1-2 illustrates the relation between the basic concepts and viewpoint specific concepts. It also shows how the relation between two viewpoints can be defined as a composition of basic relations. For example a viewpoint can have a relation to another viewpoint that is the composition of the 'refine-

ment' and the 'part of' relation (and hence be a refinement of a part of that viewpoint).

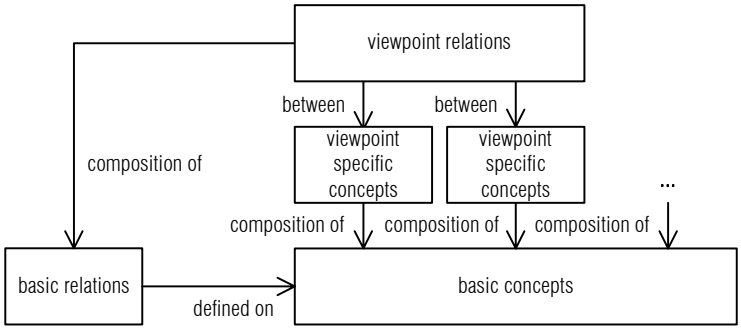


Figure 1-2 Basic Concepts, Relations and Viewpoints

1.4 Research Goals and Scope

The goal of this thesis is to develop a framework that supports the design of distributed systems from different, but related, viewpoints. For reasons outlined above, the framework proposes the use of a set of basic design concepts on which basic viewpoint relations and corresponding consistency rules can be defined. Viewpoint specific concepts can then be defined as compositions of basic concepts and relations between viewpoints can be defined as compositions of the basic relations.

To define viewpoints and relations between viewpoints in this way, the framework consists of the following parts:

1. a set of basic concepts that is sufficiently expressive to construct designs from the viewpoints that we consider (see the scope of the research below);
2. a technique to define viewpoint specific concepts as compositions of the basic concepts;
3. re-usable basic viewpoint relations on the basic concepts that can be used to define the relations between the viewpoints;
4. re-usable basic consistency rules, that correspond to the basic viewpoint relations, which can be used to verify the consistency between viewpoints that have a particular relation;
5. a technique to define viewpoint relations and the corresponding consistency rules as compositions of the basic viewpoint relations that are defined on the basic concepts.

The development of these parts are sub-goals of this thesis.

We limit ourselves to the behavioural, structural and information aspects of distributed systems design. Hence, we do not concern ourselves with aspects such as system performance and security. Also, because our

research is embedded in the area of distributed systems design for administrative enterprises, we limit ourselves to the levels of detail that are concerned with enterprise design in the form of business processes, to the design of services of distributed applications that support such business processes and to the design of the internal structure and behaviour of these applications. We address the internal structure of applications, up to (but not including) a level of detail at which they can be transformed to code in some programming language. We leave it to other modelling languages, such as the Unified Modelling Language (UML) (Object Management Group, 2004a; Object Management Group, 2003b), to address these levels of detail.

The basic concepts that we introduce are aimed towards design of aspects at levels of detail in the scope of our research. They may be expressive enough to address other aspects, such as security, or other levels of detail, such as application implementation modelling. However, we do not explicitly test them for these aspects and levels of detail.

It is the goal of this thesis to describe a framework, not a complete design process. Therefore, the framework that we propose is neutral with respect to the viewpoints that may be chosen in a particular design process and it does not define criteria for selecting viewpoints or viewpoint specific concepts. Also, the framework is neutral with respect to the order in which design steps are performed (top-down, bottom-up, waterfall, …) and therefore it is neutral with respect to the order in which designs are considered from the viewpoints. Since our framework is neutral with respect to the design process that is used, it can be used to complement any design process in which particular viewpoints (that are in the scope of the aspects and levels of detail that we consider) have already been chosen.

1.5 Research Approach

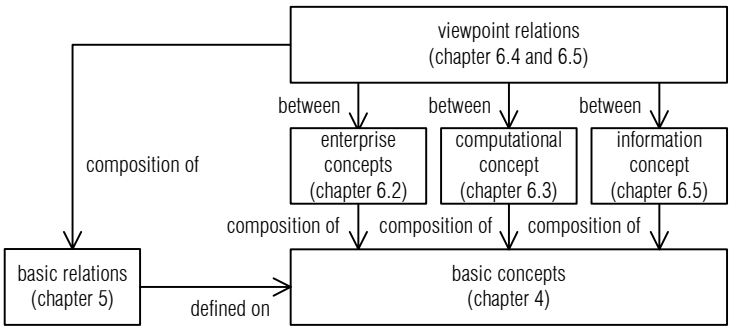
We use the following approach to achieve our goals.

Firstly, we survey the use of viewpoints in distributed systems design in literature. In the survey, we focus on the relations that may exist between viewpoints, techniques for enforcing and verifying the consistency between designs from these viewpoints and techniques for defining viewpoints and viewpoint specific concepts. Inspired by this survey and by the design relations that we defined in previous work (Quartel, Ferreira Pires, & van Sinderen, 2002; Quartel, Ferreira Pires, Franken, & Vissers, 1995), we identify re-usable basic viewpoint relations and develop consistency rules accordingly. Also, inspired by this survey, we develop a technique for defining viewpoint specific concepts as compositions of basic concepts and viewpoint relations as compositions of basic relations.

Secondly, we define the concepts from the enterprise, computational and information viewpoint of the Reference Model for Open Distributed Processing (RM-ODP) (ITU-T, & ISO/IEC, 1999; ITU-T, & ISO/IEC, 1995) as extensions of the general set of concepts that we defined in previous work (Quartel, 1998; Quartel et al., 1997; van Sinderen, 1995; Ferreira Pires, 1994). In the process of doing this, we propose some improvements to the RM-ODP viewpoints, such that the resulting viewpoints are not fully RM-ODP compliant. Therefore, we name the resulting viewpoints the enterprise, computational and information viewpoints (and not the *RM-ODP* enterprise and computational viewpoints). In accordance with the scope of this thesis, the enterprise viewpoint focuses on enterprise design in the form of business processes, the computational viewpoint focuses on the design of services of distributed applications that support such business processes and on the design of the internal structure and behaviour of these applications. The information viewpoint details the information aspects that are considered in business process and distributed application design. Also in accordance with the scope of this thesis, we restrict the use of the viewpoints to the concepts that address behavioural, structural and information aspects.

The definition of the enterprise, computational and information viewpoint concepts as extensions of the general set of concepts serves as a case study. We use this case study to evaluate whether the basic concepts are expressive enough to address the aspects and abstraction levels mentioned above. We also use this case study to evaluate the technique for defining viewpoint specific concepts in terms of basic concepts. We claim that the enterprise, computational and information viewpoints play an important role in the area of distributed systems design for administrative enterprises, because they address business process design, application service design and design of internal application structure and behaviour. Also, we claim that the RM-ODP enterprise, computational and information viewpoints are representative and that, therefore, they form a representative case study.

Figure 1-3 Steps in the Research Approach and Chapter Structure



Thirdly, we define the relation that exists between the enterprise, computational and information viewpoints in terms of basic viewpoint relations that we discovered in step 1. This exercise serves to evaluate our basic relations.

Figure 1-3 shows how our research approach addresses the research goals that are represented in Figure 1-2.

1.6 Thesis Structure

The structure of this thesis follows the research approach that we used to develop the framework. Figure 1-3 illustrates this. The remainder of this thesis is structured as follows:

Chapter 2 – *frameworks for multi-viewpoint design: an overview* – surveys existing frameworks for multiple-viewpoint design. The survey focuses on frameworks that target (at least) the behavioural, structural and information aspects of distributed systems design from the levels of detail that are concerned with business process design, application service design and design of internal application structure and behaviour.

Chapter 3 – *framework for multi-viewpoint design* – presents the actual framework. It explains the principles for multi-viewpoint design, namely how a design can be constructed from the viewpoints of multiple stakeholders and how consistency between viewpoints can be preserved in such a design. Also, it describes techniques that support the principles and that can be supported by tools.

Chapter 4 – *basic design concepts* – introduces the basic concepts of the framework. The set of concepts is partitioned into a set of concepts concerning the structure of the system, a set of concepts concerning the behaviour of the system and a set of concepts concerning the information handled by the system.

Chapter 5 – *pre-defined viewpoint relations* – Defines the basic viewpoint relations more precisely. Also, it defines reusable rules to verify the consistency between different views.

Chapter 6 – *enterprise, computational and information viewpoint* – presents the enterprise, computational and information viewpoints, which we derived from the RM-ODP viewpoints. It presents concepts to construct designs from the viewpoints and a notation to graphically represent these concepts. The concepts of the viewpoints are defined as compositions of basic concepts. Chapter 6 also defines the relations between

the viewpoints and rules for verifying consistency between designs that are constructed from the viewpoints.

Chapter 7 – *conclusions and future work* – discusses the merits and extent of the framework. It presents the conclusions of our work and directions for future work.

Chapter 2

Frameworks for Multi-Viewpoint Design: an Overview

This chapter presents an overview of existing frameworks for multi-viewpoint design. It addresses frameworks that discuss the relations between and consistency of different viewpoints and that consider both computational and enterprise viewpoints.

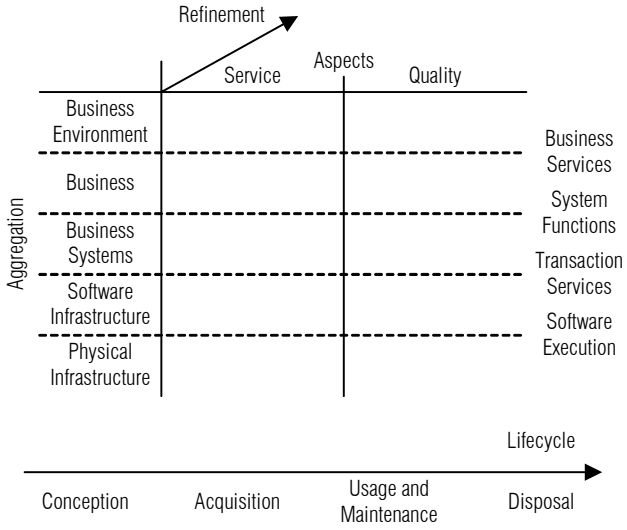


Figure 2-1 Dimensions in the GRAAL Framework

2.1 The GRAAL Framework

The goal of the GRAAL framework (van Eck, Blanken, & Wieringa, 2004; Wieringa, Blanken, Fokkinga, & Grefen, 2003) is to align parts of an overall system design. To this end, it presents dimensions according to which

viewpoints in a system design can be classified. In its most extended form, the GRAAL framework consists of four orthogonal dimensions. Figure 2-1 illustrates these dimensions. The fourth dimension is drawn separately to overcome difficulties with four-dimensional drawing.

The aspects dimension. The aspects dimension classifies viewpoints according to externally observable properties of the system that viewpoints address. It considers that a system offers *services* with a certain *quality* and classifies viewpoints according to these two aspects and further into sub-aspects. It addresses sub-aspects like: the behaviour aspect that presents possible ordering of service offerings and the service quality expected by the user.

The level of aggregation dimension. The level of aggregation dimension orders viewpoints according to the level of aggregation (of system parts) that viewpoints consider. At each level of aggregation some interacting system parts are represented. These interacting system parts provide services to the higher level of aggregation. The framework addresses levels of aggregation like:

- the physical infrastructure level that consists of physical system parts (e.g. PC, network, …) and provides services that allow higher levels of aggregation to execute;
- the business level that consists of parts of a business (e.g. actors, roles, …) and provides business services to an environment of clients.

The refinement dimension. The refinement dimension orders viewpoints according to the level of detail at which viewpoints describe the system. Adding detail to a viewpoint is said to refine the viewpoint. The framework does not consider decomposition a form of refinement, because the aggregation dimension deals with decomposition of a system into parts. Instead, each part in a system can be refined independently, by describing more information about it. For example, we can describe each part by describing its goal and we can refine it by describing how the part achieves its goal.

The lifecycle dimension. The lifecycle dimension orders viewpoints according to the stages in the lifecycle of the system that viewpoints address. The lifecycle dimension considers stages like: the conception of the system, the usage and maintenance of the system and the disposal of the system.

Relations between viewpoints. The GRAAL framework defines relations between viewpoints in terms of methodological guidelines. It describes guidelines like:

– define a software component for each business service that must be delivered;
– specify how business functions contribute to the business goal; and
– define how software applications fulfil roles in business processes and support business functions.

Guidelines are derived from relations between viewpoints that were found in case studies. The GRAAL framework presents relations at a high level of abstraction irrespective of the concepts to which they apply, because designers can freely choose the concepts that they use in each viewpoint. The GRAAL framework does not define means for verifying consistency.

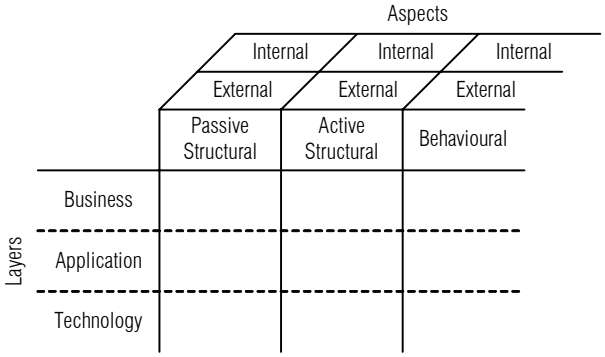


Figure 2-2 Layers and Aspects in ArchiMate

2.2 ArchiMate

The goal of ArchiMate (Lankhorst, 2005; Lankhorst, van Buuren, van Leeuwen, Jonkers, & ter Doest, 2004) is to describe the relations among different viewpoints (which they call domains) in a system design. To this end it presents concepts that can be used for design from these different viewpoints. It also presents relations that can be used to relate (instances of) concepts from the same and from different viewpoints. ArchiMate categorizes viewpoints according to which layers (business, application or technology) and which aspects they address. Figure 2-2 shows the different aspects and layers in a system design according to ArchiMate.

Architectural layers. Similar to the GRAAL framework, ArchiMate distinguishes levels of aggregation, which it calls layers, such that each layer provides services to the higher layers. It distinguishes three layers:
– the business layer that offers services to the client of the business;
– the application layer that offers application services to support the business;

– the technology layer that offers services that allow applications to execute.

Aspects. ArchiMate distinguishes aspects that are addressed in each layer. It distinguishes structural and behavioural aspects, where structural aspects are further classified into active and passive structural aspects. The active structural aspect represents parts that take initiative in performing activities (e.g. business actors or active software components). The passive structural aspect represents parts that do not take initiative (e.g. information objects or conference rooms). In the structural and behavioural aspects we can distinguish external and internal aspects. External aspects represent aspects that are observable by the users of a layer. Internal aspects represent aspects that are not.

Concepts and relations. ArchiMate presents abstract concepts and relations (Jonkers, Lankhorst, van Buuren, Hoppenbrouwers, Bonsangue, van der Torre, 2004) that can be used to construct a design for each of the layers. Abstract concepts and relations can only be used for design at high levels of abstraction. To construct more detailed designs, ArchiMate relies on viewpoint-specific languages. This is in line with the philosophy of ArchiMate, in which stakeholders can use their own languages and tools to construct designs from their viewpoints. Designs from the different viewpoints can be imported in ArchiMate at a high level of abstraction, therewith abstracting from details that are represented in the viewpoint specific languages. Subsequently, ArchiMate can be used to define relations between the viewpoints at a high level of abstraction. For example, a business process design can be imported in ArchiMate. The result is a design that represents activities and flow relations between these activities, but no details about the flow relations (e.g. whether they are OR-split or AND-splits). Subsequently, we can use ArchiMate to relate business process activities to the application services that support them, which are imported from some other language and tool.

2.3 RM-ODP

The Reference Model for Open Distributed Processing (ITU-T, & ISO/IEC, 1999; ITU-T, & ISO/IEC, 1995) (RM-ODP), which is currently being revised (ISO/IEC/JTC1/SC7, 2004), presents a reference model to define standards for the design and development of open distributed systems. It consists of concepts and functions that can be used to define RM-ODP compliant standards. Such standards include (but are not limited to):

standards for modelling open distributed systems and standards for open distributed processing components.

The RM-ODP concepts are structured into five viewpoints:

- the enterprise viewpoint, which defines concepts and concept relations to specify the role of an ODP system in an environment;
- the computational viewpoint, which defines concepts and concept relations to specify a functional decomposition of an ODP system;
- the information viewpoint, which defines concepts and concept relations to specify the structure of information in an ODP system and basic operations that can be performed on that information;
- the engineering viewpoint, which defines concepts and concept relations to specify the mechanisms and functions that support distributed interactions between ODP system parts;
- the technology viewpoint, which defines concepts and concept relations to specify the technology onto which an ODP system is implemented.

RM-ODP defines basic concepts that are used as a basis for defining the viewpoint concepts. A viewpoint concept is either a specialization of a basic concept or is defined in terms of basic concepts. For example, the 'computational object' concept is a specialization of the 'object' concept and a 'signal' is an 'atomic shared action', where 'atomic' and 'shared action' are basic concepts.

RM-ODP defines consistency rules that aid in keeping viewpoint specifications consistent. The consistency rules consist of correspondences that specify which concepts from one viewpoint 'correspond to' which concepts from another viewpoint. If concepts from one viewpoint correspond to concepts from another viewpoint, then specifications from those viewpoints must make correspondence statements. Correspondence statements specify which concept instances from one viewpoint specification correspond to which concept instances from another viewpoint specification. RM-ODP leaves it to other standards to define what is meant by concepts from one viewpoint 'corresponding to' concepts from another. For example, the enterprise object concept from the enterprise viewpoint corresponds to the computational object concept from the computational viewpoint. Hence, in an ODP specification, a correspondence statement must relate enterprise objects from the enterprise specification to corresponding computational objects from the computational specification.

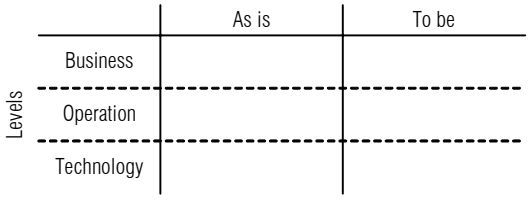


Figure 2-3 Organizational Levels in SEAM

2.4 SEAM

The Systemic Enterprise Architecture Methodology (SEAM) (Wegmann, 2003) proposes an approach to align an enterprise's strategies, processes and supporting systems. To this end it presents:
- organizational levels from which the enterprise can be designed;
- concepts for designing the enterprise from each of these levels;
- a method for designing the enterprise, proposing some design activities.

Organizational levels. SEAM proposes the organizational levels that are illustrated in Figure 2-3 to organize an overall design. Each of these levels considers a system of interacting entities. However, at each level the kind of entities that is considered differs. The business level is used to represent the business in its value chain, considering the business and its clients and providers as entities. The operation level is used to represent the operations of the business itself, considering the actors in the enterprise (including software applications) as entities. The technology level is used to represent the technical infrastructure of the business, considering software components as entities. Each of the levels can be represented as-is or to-be.

Concepts. SEAM uses the concepts from RM-ODP for design. Naumenko (Naumenko, 2002) defines an abstract syntax for RM-ODP in a language called Alloy (Jackson, 2002) that uses a set theoretic formal semantics. Balabko and Wegmann (2003) explain the abstract syntax for designing behaviour in more detail. They also explain how behaviour in RM-ODP can be represented in different modelling languages.

Traceability. SEAM aids in maintaining consistency by defining traceability relations. Traceability relations can be used to relate concept instances from different organizational levels. SEAM does not define rules for checking consistency between organizational levels. It relies on the designer to maintain the consistency.

2.5 The ViewPoints Framework

The ViewPoints framework (Finkelstein, Gabbay, Hunter, Kramer, & Nuseibeh, 1994; Nuseibeh, Kramer, & Finkelstein, 1994; Nuseibeh, Kramer, & Finkelstein, 1993) is one of the first and one of the most influential works on multi-viewpoint design and consistency. It is the first attempt to shift from the idea of views as abstractions of existing information to views that are developed separately by different stakeholders.

In the ViewPoints framework, each viewpoint is defined by five elements:

- a domain that specifies the universe of discourse of the viewpoint;
- a style that defines the modelling language used to construct a design from the viewpoint;
- a work plan that defines the process according to which a design can be constructed from the viewpoint;
- a specification that represents the design from the viewpoint, using the modelling language defined by the style;
- a work record that represents the information about the current state and history of the specification.

A complete system design consists of viewpoints and rules that define requirements for consistency between these viewpoints. Each rule defines a condition that must hold in a consistent specification and defines an action that must be taken if that condition is violated. The rules are specified as queries on the database that contains the design.

To verify consistency the ViewPoints framework assumes that the viewpoint specifications and the consistency rules are mapped onto a database that supports reasoning with first-order logic. The database will then perform the consistency check.

2.6 OpenViews

OpenViews (Boiten, Bowman, Derrick, Linington, & Steen, 2000) presents an approach to maintain consistency in RM-ODP based designs, although it can be applied in a more general context.

OpenViews defines that two views are consistent if a design can be found that is a refinement of both views. The views can be modelled using different modelling languages, in which case one of the views must be transformed, such that the views are represented in the same language. Specifically, OpenViews describes in more detail how to relate views that are represented in LOTOS (van Eijk, Vissers, & Diaz, 1989) and Object-Z (Smith, 2000). For that purpose Derrick, Boiten, Bowman, and Steen (1999) define how to transform a view represented in LOTOS into a view represented in Object-Z.

OpenViews differs from the approaches above, which rely on view relations and consistency rules. Also, it differs from the approaches above in that it defines when two views *are* consistent (namely when a common consistent refinement can be found). The approaches above define when two views *are not* consistent (namely when consistency rules are not satisfied).

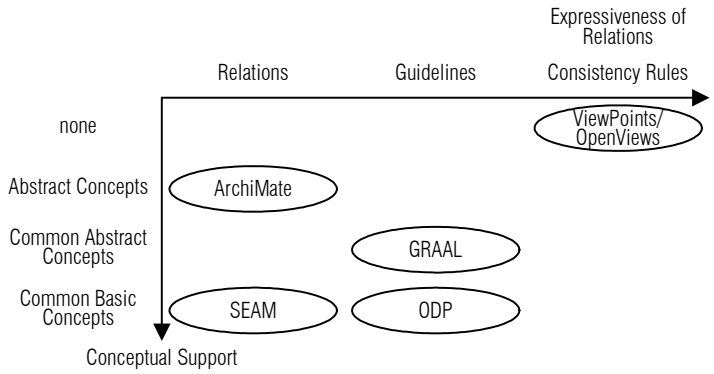


Figure 2-4 Consistency Verification Support of Frameworks

2.7 Conclusions

Figure 2-4 illustrates the different levels of support that the frameworks above have for defining view relations and for checking consistency between views. The figure classifies the support according to two criteria: expressiveness of view relations and conceptual support.

The expressiveness criteria distinguishes the frameworks according to the support that they have for defining consistency rules. At the lowest level, a framework supports the definition of relations between views, but not the (consistency) rules that apply to these relations. At the next level a framework supports the definition of consistency rules. However, advanced consistency rules, such as refinement, are not supported. At the highest level, a framework fully supports the definition of consistency rules.

A framework supports view relations conceptually, if it provides concepts for the views considered in the framework. These concepts provide a frame of reference that helps designers to think about relations between views. First, designers must define the relation between viewpoint-specific concepts and the framework's concepts. Second, they must define the relation between concepts from different viewpoints, using the relations between the framework's concepts. A framework can provide conceptual support to a varying degree.

Abstract concepts provide abstractions of concepts that may be used in each of the views (covered by the framework). Since they are abstractions, they may not cover all design properties in detail. Common abstract concepts have the additional property that they are shared between the views, where regular abstract concepts are different for each of the views. (Common) basic concepts cover all design properties in detail.

We aim to support the definition of view relations at the level of expressiveness that fully allows for the definition of consistency rules. Also, we aim to support the view relations by providing common basic concepts.

Chapter 3

Framework for Multi-Viewpoint Design

An earlier version of the work presented in this chapter was published in (Dijkman, Quartel, Ferreira Pires, & van Sinderen, 2003)

This chapter presents our framework to support multi-viewpoint design and maintain consistency between different views in such a design. The framework allows stakeholders to use their own design concepts, tools and modelling languages for their view.

This chapter is structured in two parts. The first part discusses the principles of multi-viewpoint design. This part defines the viewpoint concept precisely and explains how a multi-viewpoint design should be constructed. It explains how we can choose the viewpoints that we want to use in a multi-viewpoint design, such that the viewpoints are aligned with the design methodology that is used and such that the multi-viewpoint design is complete in the sense that it covers all relevant aspects. The first part also considers how we can maintain the consistency between views in a multi-viewpoint design. The second part explains how the principles for multi-viewpoint design can be supported by tools. To this end it explains a standardized syntax for defining viewpoints and modelling languages. Also, it explains how the principles for multi-viewpoint design can be realized by manipulation of this syntax. This leads to requirements for tool-suites for multi-viewpoint design.

3.1 Principles of Multi-Viewpoint Design

The principles for multi-viewpoint design include principles for: (i) defining and using a viewpoint; (ii) choosing a set of viewpoints for a multi-viewpoint design and describing the relations between these viewpoints; (iii)

graphically or textually representing a view (modelling); and (iv) assessing the consistency of a multi-viewpoint design.

3.1.1 Viewpoints and Views in a Design

A stakeholder in a design has a particular interest in that design. More specifically, we say that a stakeholder focuses on certain design concerns and considers these concerns at a certain level of abstraction or certain levels of abstraction. This observation is shared by most of the frameworks that we considered in chapter 2. We define design concern as follows:

Definition 3-1 Concern

A concern is a class of system properties.

For example, the behaviour concern is the class of properties that address the behaviour of a system, such as the activities that can occur in the system and the relations between these activities.

We define level of abstraction as follows:

Definition 3-2 Level of Abstraction

A level of abstraction, also called a level of detail, is a relative position in the design process that prescribes what design information is essential at that position in the design process.

Figure 3-1 Adding Detail in a Design Process

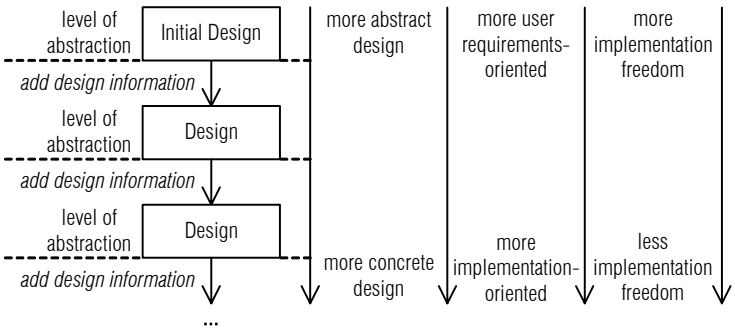


Figure 1-2 illustrates the design process and the relation between levels of abstraction and the design process. A design process starts out with an initial design that is an initial representation of a (distributed) system that solves a given problem. The initial design is gradually transformed into a design that contains sufficient information to start building the system. This transformation can be performed in successive design steps, where design information is added in each step. Design information is represented by concept instances. A level of abstraction prescribes what design information should be considered in a design at the end of a design step. A level of abstraction is more abstract or higher than another level of abstraction, if it prescribes less design information. It is more concrete or lower, if it pre-

scribes more design information. Similarly, a design is more abstract than another design if it prescribes less design information and a design is more concrete if it prescribes more design information. Each design step takes us from a more user requirements-oriented design to a more implementation-oriented design, because we add design information that describes in more detail how the system must be implemented. For that same reason implementation freedom decreases with each design step.

Example 3-1 Adding Design Information to a Design

The design that prescribes that 'activity *b* must occur after activity *a*' is more abstract than the design that prescribes that 'activity *b* must occur *within 300 milliseconds* after action *a*'. The more concrete design allows for less implementation freedom, because only an implementation that makes action *b* occur *within 300 milliseconds* after action *a* satisfies these requirements. Less implementations satisfy this requirement than the requirement that *b* must occur after action *a*. From the more abstract to the more concrete design the design information *within 300 milliseconds* is added.

Note that multiple designs can address the same level of abstraction, because a level of abstraction is a prescription of design information that must be provided. Multiple designs can satisfy this prescription.

Focusing on certain concerns and a certain level of abstraction, a stakeholder considers only a part of an overall design and a subset of the concepts that are used to construct that design. We call this combination of concerns, levels of abstraction and concepts the viewpoint of a stakeholder.

Definition 3-3 Viewpoint

The viewpoint of a stakeholder is the combination of the concerns and levels of abstraction that the stakeholder addresses and the set of concepts that the stakeholder uses to construct his or her part of an overall design.

Hence, to describe the viewpoint from which a stakeholder observes a design, we must describe:

1. the concerns that are addressed from that viewpoint;
2. the level of abstraction at which those concerns must be addressed; and
3. the concepts, and rules for combining instances of these concepts, that can be used to construct a design part from that viewpoint.

A view coincides with a part of a design that represents the interest of a stakeholder.

Definition 3-4 View

The view of a stakeholder on a design is the part of a design, constructed from a viewpoint, that represents the interest of that stakeholder.

A stakeholder constructs a view by combining instances of design concepts that are defined by the viewpoint of that stakeholder. The view must represent the concerns that the viewpoint from which it is constructed prescribes

at the level of detail that the viewpoint from which it is constructed prescribes.

Figure 3-2 Viewpoint Terminology and its Relations

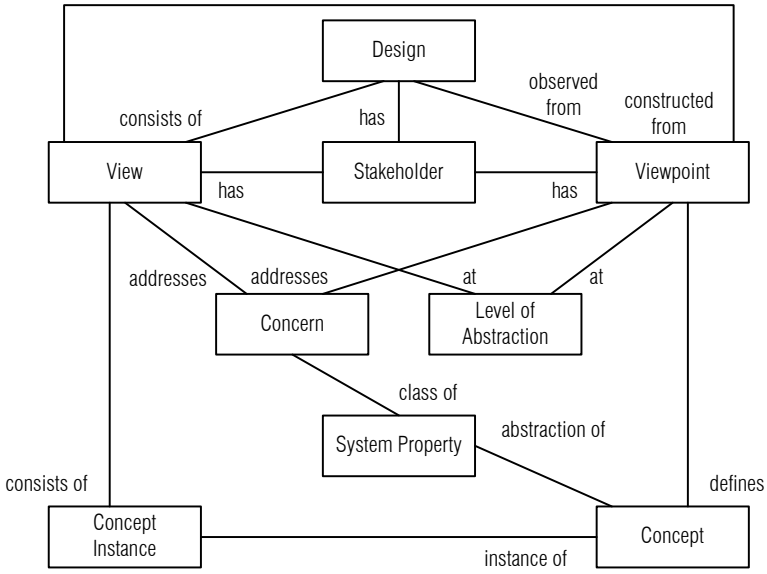


Figure 3-2 illustrates the terms that we introduced in this section and the relationships between these terms. It shows that a design is observed from the viewpoints that the stakeholders in that design have. Viewpoints address concerns at a level or some levels of abstraction, where a concern is a class of system properties. A viewpoint defines concepts that can be used to construct a part of a design from that viewpoint. We call such a part of a design a view. A view consists of instances of concepts that are defined by its viewpoint. Also, it addresses the concerns that its viewpoint addresses at the level of abstraction at which its viewpoint prescribes. Our outline of multi-viewpoint architectural design can be used as an extension of the approach for multi-viewpoint architectural description defined by IEEE (2000), because we also consider relations between viewpoints and concepts that can be used for modelling from a viewpoint.

Example 3-2 A Viewpoint

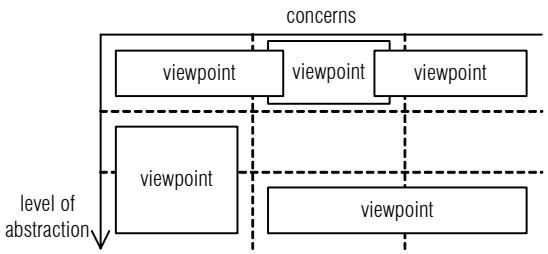
A business process viewpoint is an example of a viewpoint. This viewpoint is a way of looking at a system for a stakeholder that is interested in the way in which work is performed in an enterprise. This stakeholder considers the tasks that are performed in the enterprise and the relations between those tasks. We refer to that as the process concern. The stakeholder also considers who performs which tasks, which we refer to as the organizational concern. Both the process and the organizational concern are considered at a level of abstraction at which we consider the individual employees in a company. The organizational concern addresses individual employees and the process concern addresses tasks that these individual employees

have to perform. Examples of design concepts that are used from the business process viewpoint are: *task*, *sequence relation*, *choice relation* and *actor*. Hence, according to the definition of the business process viewpoint, a business process view consists of instances of the *task*, *sequence relation*, *choice relation* and *actor* concepts. An example of a view is the view that contains two instances of the *task* concept. One task has the description 'fill out application form' and the other has the description 'evaluate application'. These tasks are related by one instance of the *sequence relation* concept and are performed by the instance of the *actor* concept with the name 'John Jameson'.

3.1.2 Relating Views in a Design

Since viewpoints consider certain concerns at a certain level of abstraction, we can position the viewpoints of a design relative to each other. Figure 3-3 illustrates the relative position of some viewpoints in a table. The columns of the table represent the different concerns of stakeholders in the design, while the rows represent the levels of abstraction at which these concerns are considered. The rows in the table are ordered, such that the level of abstraction decreases (and therefore the level of detail increases) as we get to lower rows in the table.

Figure 3-3 Table that Contains the Viewpoints of a Design



System properties that are prescribed at some level of abstraction do not lose their validity at lower levels of abstraction. They can only be *refined*. Therefore, if a concern is covered at a certain level of abstraction, then the properties that it prescribes must also be covered at lower levels of abstraction. To prevent that parts of a design must be repeated without change at successive levels of detail, we allow such parts to be left out. We do this by assuming that, if a concern is not covered at a certain level of abstraction, while it is covered at a higher level of abstraction (as is the case in the table from Figure 3-3), then the properties that held for that concern at the higher level of abstraction, also hold at the lower levels of abstraction. For example, at the higher levels of abstraction, a stakeholder can consider Quality of Service constraints, such as 'a client must receive an answer within 5 seconds'. These Quality of Service constraints may not have to be considered explicitly at some successive intermediate levels of abstraction and be considered again once an implementation platform has been chosen.

However, the Quality of Service constraints do not lose their validity at the intermediate levels of abstraction. Therefore the Quality of Service constraints described at the higher levels of abstraction must hold at each intermediate level.

Since a view is constructed from a viewpoint, by combining concepts that that viewpoint defines, the views in a design are also positioned relative to each other. The relative position of views can be represented in the same table as the table that organizes the viewpoints.

Basic types of relations between views. Since different views in a design consider the same system, views are related in one way or another. In our framework the designer must explicitly prescribe the relations that exist between views, because defining the relation between views explicitly is a prerequisite to verifying the consistency between those views.

The basic types of relations that viewpoints, and therefore views, can have with each other can be inferred from the relative position that views can have with respect to each other. Because a viewpoint considers certain design concerns at a certain level of abstraction, different viewpoints can consider different design concerns and different levels of abstraction. Therefore, we distinguish between two basic types of view relations: the refinement relation and the overlap relation.

Definition 3-5 Refinement Relation

Two views have a refinement relation, if they consider the same concerns at different levels of abstraction.

In case of a *refinement relation* between two views, the more concrete view is a refinement of the more abstract view. For example, one view may consider the behaviour of a system as a whole, while another view also considers the internal behaviour of that system in terms of the interactions between its components. These views are related because the internal view is a refinement of the system view, i.e., adds design detail, by providing a decomposition of the system that prescribes how the system can be implemented.

Two views may consider different concerns. In this case we say that they are *orthogonal*. For example, one view may consider the structure of a system in terms of interconnected components, while another view considers the behaviour of each component. These views are orthogonal in the sense that the structural view identifies the components and their interconnections, whereas the behavioural view considers the behaviour of the component. In general, it may be difficult, if not impossible, to separate concerns such that they are fully orthogonal, in the sense that they have no system properties in common. Therefore, views that consider different concerns at the same level of abstraction are likely to have overlap. For example, the

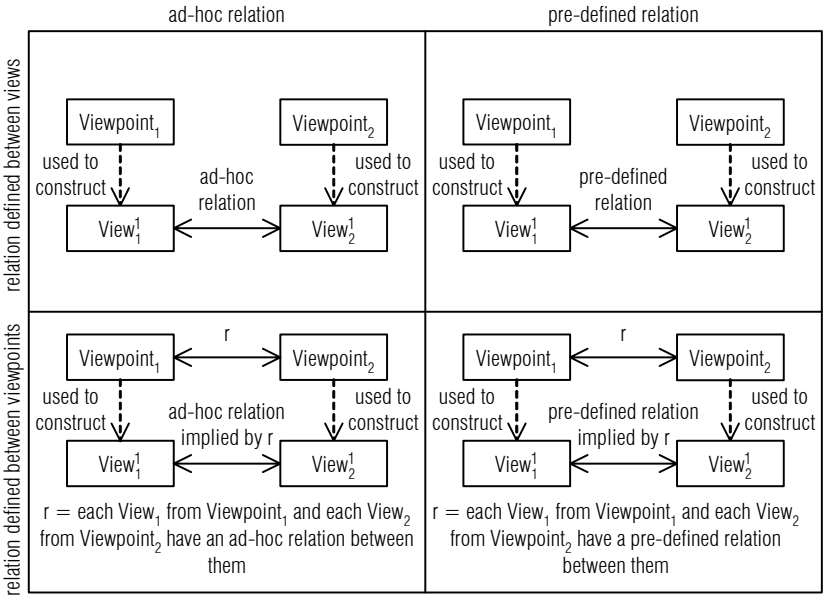
structural and behavioural view are not fully orthogonal, because the behavioural view should specify the behaviour of each component identified in the structural view. Hence, both views address the system component properties, although one focuses on the interconnections of the system components, while the other focuses on the behaviour of the system components. If two views have partly overlapping concerns we say that they have an *overlap relation*.

Definition 3-6 Overlap Relation

Two views have an overlap relation if they partly consider the same system properties.

We can prescribe more design information about the relation between two views, by prescribing which properties that are represented by the views are related and how. In case of a refinement relation between two views, we can prescribe what design information is added to which properties in one view and in which, more concrete, properties this results in the other view. In case of an overlap relation between two views, we can prescribe which properties in one view have overlap with which properties in another view and what the nature of the overlap is (the properties are the same, the properties are partly the same, …). We can represent the design information about view relations as associations between concept instances, because concept instances represent (instances of) properties.

Table 3-1 Possibilities to Define View Relations



Different ways to prescribe view relations. Table 3-1 illustrates four possibilities for prescribing the relation between views, depending on

whether these relations are prescribed in an ad-hoc or pre-defined manner and depending on whether they are described between viewpoints or between views. Relations are pre-defined if they are defined for re-use in different designs. They are ad-hoc if they are defined specifically for a single design.

Whether an ad-hoc or a pre-defined relation is used has an impact on the re-usability of view relations between designs as well as on the flexibility in adapting a relation. On the one hand, using a pre-defined relation has the benefit of being able to re-use an existing relation, but also has the drawback of not being completely free to define the relation. On the other hand, using an ad-hoc relation has the benefit of being completely free to define that relation, but also has the drawback of having to define the relation by oneself.

Whether a relation is defined between views or between viewpoints, has an impact on the re-usability of a view relation when views are modified as well as on the flexibility in adapting a relation to two views. On the one hand, if a relation is defined between viewpoints rather than directly between views, then that relation is not affected when views are modified. Hence, a relation that is defined at a viewpoint level can be re-used for different views that are constructed from that viewpoint. On the other hand, it may not be possible to define (a part of) a relation between viewpoints, because it is not possible to define unambiguously at a viewpoint level which concept instances the relation relates. For example, at a viewpoint level, we may be able to specify that each behaviour from (a view from) one viewpoint is a refinement of a single behaviour from (a view from) another viewpoint. However, that does not provide us with enough information at a view level to determine exactly *which* behaviour is a refinement of which. Therefore, this relation must partly be defined at a view level.

For these reasons, we propose that, in a multi-viewpoint design, both ad-hoc and pre-defined relations are used and relations are defined between viewpoints as well as between views. This combines the re-usability of pre-defined and viewpoint-level relations with the flexibility of ad-hoc and view-level relations. In section 3.2 we explain how these approaches can be combined if a particular syntax is used to represent viewpoints. In chapter 5 we pre-define some re-usable view(point) relations.

3.1.3 Consistency in a Multi-Viewpoint Design

If a relation is prescribed between two views, those views must observe the requirements that the relation between them implies. A refinement relation between two views implies that the more detailed view can be obtained from the less detailed view by adding design information. An overlap relation between views implies that the views are equivalent with respect to

their overlap. More detail can be prescribed about the requirements that are implied by a view relation, such as the (type of) design information that is added in a refinement relation, or the nature of the overlap between views. We separate view relations from *consistency rules*.

Definition 3-7 Consistency Rule

A consistency rule is a rule that represents a requirement that is implied by a view relation.

The separation between view relations and their consistency rules is merely a representation choice, because the consistency rules, being implied by the view relations, are an inherent part of those relations. We make this representation choice, such that the design activity of defining relations between views can be separated from the design activity of verifying the consistency between views. However, these activities can still be combined if the designer so desires. Chapter 5 pre-defines the consistency verification rules that accompany our pre-defined view relations.

We call the verification of the requirements implied by the relation, according to the consistency rules, *consistency verification*. If a view is related to another view by means of a refinement relation, design detail is added to get from the less detailed to the more detailed view. Hence, two views are consistent according to a refinement relation if, after removing the added design detail from the more detailed view, the more detailed view is equivalent to the less detailed view. Two views are consistent according to an overlap relation, if they are equivalent (according to some notion of equivalence) with respect to their overlap.

3.1.4 Choosing Viewpoints

Before a design is constructed, viewpoints must be selected to construct the design from. It is not our aim to prescribe exactly which concerns, levels of abstraction and viewpoints a design must cover. Our framework aims to complement existing design methodologies that typically already imply particular concerns and levels of abstraction. However, we do provide guidelines for selecting viewpoints from which a design can be constructed. These guidelines can be used to improve existing design methodologies, by making these methodologies adhere to the guidelines. They can also be used to add viewpoints to a design, if the methodology of choice does not cover all concerns or levels of abstraction that are relevant to the stakeholders.

Align viewpoints with the methodology used. The aim of our framework is that it is used in combination with an existing design methodology. However, to use our framework in combination with a methodology, we

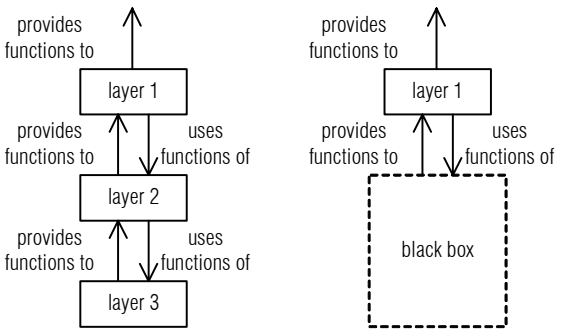
must align it with that methodology. We do that by aligning the viewpoints to the *design milestones* and *layers of functionality* that may be defined in the context of a methodology.

Definition 3-8 Design Milestone

A design milestone is a well-defined design objective.

Since a design milestone is a design objective, it can be considered a level of abstraction, because it also prescribes what must be represented (to reach the objective). Therefore, to align our viewpoints to a methodology, we can define levels of abstraction that match the milestones in a design methodology. Even though we align our levels of abstraction to milestones, we are still allowed to define other levels of abstraction as well. Also, because a level of abstraction is prescriptive, we are allowed to define levels of abstraction such that multiple milestones correspond to the same level of abstraction. For example, we could define a level of abstraction that addresses the behaviour of a system as a whole and a milestone that corresponds to that level of abstraction. The designer may present a view at that level of abstraction to the client and modify the view based on comments from the client, therewith reaching a new milestone. Although design information is added in that process, the resulting view is still at the level of abstraction that addresses the behaviour of a system as a whole. Hence, both milestones correspond to that level of abstraction. In all evolutionary design approaches, where a design process is iterated several times until the designer is satisfied with the result (Pressman 2003), this situation can occur. Even though design detail is added in each iteration, the stakeholders observe that the levels of abstraction that they defined are addressed once for each iteration.

Figure 3-4 Layers of Functionality



i. Layered functionality ii. Layered functionality with black box

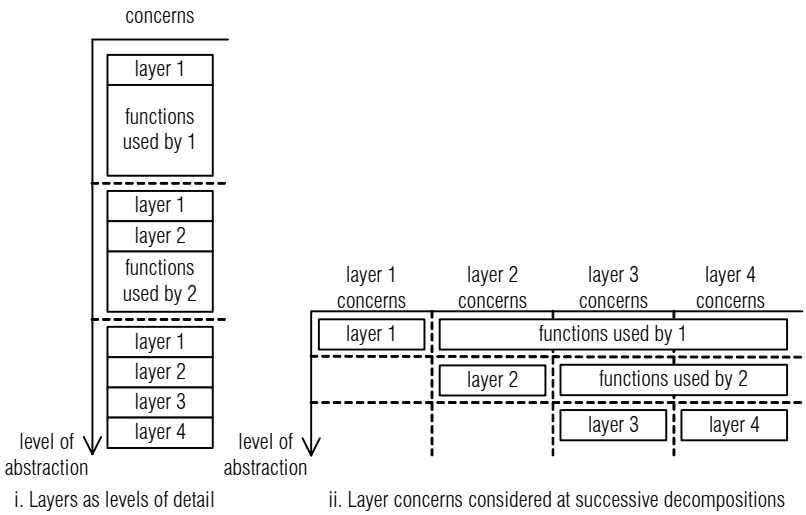
We define layer of functionality as follows:

Definition 3-9 Layer of Functionality

A layer of functionality is a set of functions that can be used by other sets of functions and use other sets of functions, but that cannot both use (directly or indirectly) and be used (directly or indirectly) by the same other set of functions.

We call a set of functions that cannot both use and be used by the same other set of functions: a *layer*, because it can be *layered* on top of another layer and other layers can be layered on top of it, such that the higher layers only use, but are not used by, the lower layers and the lower layers are only used by, but do not use, the higher layers. Figure 3-4.i illustrates this. We can gradually introduce layers of functionality, during the design process. We can start out with a description of what functions we need, considering the layers that provide them as a black box. Subsequently, we can introduce the layer of functionality that provides the required functions and uses the functions from another black box. Figure 3-4.ii illustrates this step in the design. We can apply this process recursively, by opening up each black box and defining it as a layer of functionality and another black box. The process ends when we can directly implement the black box as a layer of functionality. Each introduction of a layer of functionality introduces design detail with respect to the black box, because each black box describes only *what* functions are provided. The decomposition of the black box into a layer of functionality and another black box also describes *how* these functions are realized. Hence, we can define a level of abstraction that matches the composition of the layers of functionality that were identified so far *and* the black box that provides the functions required by those layers. For example, Figure 3-4.ii shows a design at a level of abstraction after the identification of one layer. Figure 3-5.i illustrates the levels of abstraction that we can identify for different layers of functionality in a design methodology.

Figure 3-5 Incorporating Layers of Functionality as Viewpoints



An alternative to match viewpoints to layers of functionality is by considering that each layer addresses its own concerns. Hence, we create a concern for each layer of functionality. This concern is addressed by the layer itself and by the black box that contains the layer. The black box considers the concerns at a higher level of abstraction, because it specifies what concerns the layer addresses and abstracts from how the layer implements these concerns. Figure 3-5.ii illustrates this relation between layers of functionality and viewpoints. The figure shows that the concerns of *layer 1* are considered by the view that covers that layer at the highest level of abstraction. The concerns of *layer 2* are considered by the view that covers that layer at the second highest level of abstraction. The concerns of *layer 2* are also considered by the black box that incorporates that layer, but abstracts from contributions of the individual layers, *functions used by 1*. The levels of abstraction in this figure represent the successive levels of decomposition.

Align viewpoints to requirements of stakeholders. According to our definition, viewpoints address the concerns of stakeholders at the levels of abstraction that these stakeholders consider. Therefore, we can add viewpoints, levels of abstraction and concerns as required by the stakeholders. If the viewpoints, levels of abstraction and concerns were originally derived from milestones and layers of functionality as described in the previous paragraphs, this may involve modifying those viewpoints. For example, in Figure 3-5 the concerns from each of the layers could be split-up into several more fine grained concerns. Similarly, we can introduce levels of abstraction between two levels of abstraction that were derived from milestones.

Separation of concerns. The generic architectural principle of separation of concerns (where concerns are not necessarily concerns as they are defined above) states that design problems should be broken up into subproblems that can be solved relatively independent, after which the solutions can be combined again into a complete solution. The principle of separation of concerns also applies to the concerns and viewpoints that we identify. Applied to concerns and viewpoints, it implies that we should reduce the overlap between the concerns and viewpoints that we identify to a minimum. Based on this principle, we could remove concept instances from one viewpoint that have an equivalent in another viewpoint. Such concepts do not add information, because the property that they represent is already addressed by another view. The relation that exists between removed concept instances and other concept instances from that view can be incorporated into the overlap relation that exists between the views. Note, however, that the existence of equivalent concepts may be intentional and useful, because they improve the clarity of the individual viewpoints. In that

case the concepts should not be removed. The separation of concerns principle may also motivate that concerns are split up into several more fine grained concerns.

Derive generic concerns. If concerns are (partial) specializations of the same concern, then there may be a benefit in grouping them under this generic concern. The benefit is that the relation between the concerns at different levels of abstraction becomes more clear. Hence, there is mainly a benefit when the concerns are addressed at different levels of abstraction. For example, the *customer satisfaction* concern that addresses the timeliness and mistake-rate of the service that is delivered to the customers and the *application quality of service* concern can (partly) be considered specializations of the same generic concern *quality of service*. Hence, they can be grouped under this concern. In this way, it becomes more clear that customer satisfaction is (partly) realized by application quality of service at a lower level of abstraction. Eventually all concerns are specializations of the concern *a concern in the design*, so they could be grouped into that single concern. However, this violates the principle of separation of concerns. This illustrates that there is a trade-off between separation and generalization of concerns. The designer has to choose the concerns at the appropriate level of generality.

Observe generic viewpoint relations. Viewpoints are in principle always related, either directly or indirectly, because they are used to represent design parts of the same system. Hence, we should define our viewpoints in such a way that no groups of viewpoints exist between which there are no relations. Also, viewpoints must always be assigned to the levels of abstraction and concerns that they address. Since, for each two levels of abstraction, one is always more concrete than the other, viewpoints are either (partly) related by refinement, orthogonal or overlapping. This implies that two viewpoints at different levels of abstraction cannot consider the same concern and be unrelated by refinement.

Frequently occurring concerns and levels of abstraction. Each of the frameworks presented in chapter 2 presents its own concerns and levels of abstraction (or layers of functionality). These concerns and levels of abstraction can be used as a guideline for developing one's own concerns and levels of abstraction. Also, we can use the concerns and levels of abstraction that are suggested by enterprise architecture frameworks, such as the Zachman framework (Zachman, 1987; Sowa, & Zachman, 1992) and The Open Group Architecture Framework (TOGAF) (The Open Group, 2005).

3.1.5 Viewpoints and Modelling Languages

Viewpoints and their concepts are meant to construct mental images of a part of the design. Since it is hard to discuss and share designs in terms of mental images that only exist in the minds of the stakeholders, we should make our designs concrete in the form of models. A *model* is a textual or graphical representation of a design (part). Hence, design takes place in three related 'worlds': the real world, where the real system and its environment exist or will be constructed, the conceptual world, which is the conception of the real world in our minds, and the symbolic world, which is the concrete representation of the conceptual world on some medium (e.g., paper or a computer screen). Views exist in the conceptual world, because they are mental images of the stakeholders. Each view may be represented by different models that use different symbolisms.

Just as a viewpoint defines the means to construct a view in the form of concepts, a *modelling language* defines the means to construct models. These means consist of *language concepts*, which define what can be modelled, and *notational elements* to represent (express) the language concepts graphically or textually. A language concept represents some system properties, similar to a design concept. For example, UML Activity Diagrams define the language concept 'Action', which is represented by the graphical notational element 'Rounded Rectangle'. The UML language concept 'Action' represents some unit of activity that can be executed by a system.

A *representation relation* between language concepts and the notational elements defines how each language concept is *represented* by a single notational element or a composition of notational elements. Similarly, it shows how notational elements can be *interpreted* in terms of language concepts. A modelling language may associate more than one notation with the same concept. For example, it is common for modelling languages to define both a graphical and a textual notation with the same concept, because the graphical notation is easier to understand by humans, while the textual notation is easier to use for processing by tools.

The benefit of distinguishing language concepts from notational elements is that we can clearly separate the conceptual aspects from the notational aspects of a modelling language. This is especially useful, because the notational elements often contain properties that are used by modelling tools to draw the model. Examples of such properties are the 'height' and 'width' of a rounded rectangle. These properties are not relevant from a conceptual perspective. In contrast, language concepts *are* relevant to stakeholders, because they imply a viewpoint. Moreover, because language concepts have their own semantics, some languages are suitable to represent a viewpoint, while others are not.

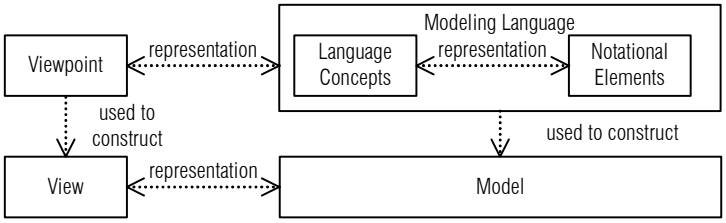


Figure 3-6 Viewpoints, Modelling Languages and their Relations

The relation between viewpoints and modelling languages. Figure 3-6 illustrates the relation between views, viewpoints, models and modelling languages. It is consistent with the relation between concepts and notational elements that is defined in (Ferreira Pires, 1994). In order to use a modelling language to represent (the views according to) some viewpoint, we must define the relationship between the language concepts of the modelling language and the design concepts of the viewpoint. This relationship should clearly define how (compositions of) language concepts represent (compositions of) the design concepts from the viewpoint.

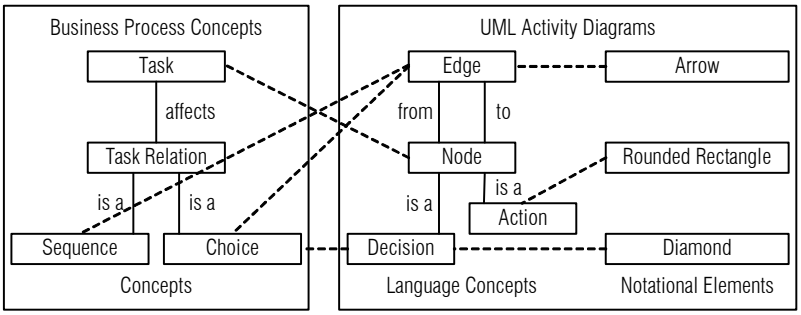
In the ideal case modelling language concepts correspond to viewpoint concepts in a one-to-one fashion. Since this means that the sets of concepts are equivalent, we only need one set of concepts. Hence, there is no need for a representation relation in this case. However, in general language concepts differ from viewpoint concepts, because they are developed separately (by different expertise groups). For example, modelling languages are often developed by tool builders, who want to make their tool suitable for as many stakeholders as possible. Therefore, the language concepts that they use are generic. In contrast, viewpoints are defined by stakeholders in a system design, who want to define their concepts to match the properties they consider important as closely as possible. Therefore, the viewpoint concepts that they define are more viewpoint specific (although stakeholders may (partly) adopt more generic concepts). This implies that modelling languages can often be used to represent more than one viewpoint. For these reasons, the representation relation between viewpoints and modelling languages can take different forms, depending on how much the viewpoint concepts differ from the language concepts in available modelling languages, how much stakeholders of the viewpoint are willing to adapt their viewpoint concepts and how easy it is to adapt the language concepts and their corresponding tools.

In the extreme case language and viewpoint concepts are made to correspond to each other in a one-to-one fashion, either by forcing the stakeholders to adapt to the available modelling language concepts or by forcing the modelling language to adapt to the viewpoint concepts. The first approach is often chosen in practice, where a set of popular modelling languages, like the UML, is selected and the definition of viewpoint concepts is

left implicit. The stakeholders are then not free to choose their own viewpoint concepts. Instead, the viewpoint concepts are implied by the language concepts. The second approach is the approach proposed by Domain Specific Languages (DSL) (van Deursen, Klint, & Visser, 2000). A DSL is a modelling language developed especially for a particular application domain (such as a viewpoint or a set of viewpoints). Hence, a DSL does not define its own concepts, but instead uses the concepts from the viewpoint(s) that it represents.

We claim that it is not good practice to force stakeholders to use the concepts from a modelling language, because viewpoint concepts should be motivated by the properties that a stakeholder wants to represent. Language concepts should adapt to these concepts. People have employed this approach for the area of workflow design. For this area, an inventory of the viewpoint concepts (patterns) used by stakeholders has been made (van der Aalst, ter Hofstede, Kiepuszewski, & Barros, 2003; Russell, van der Aalst, ter Hofstede, & Edmond, 2005; Russell, ter Hofstede, Edmond, & van der Aalst, 2005). The ability of existing languages to represent these concepts has been evaluated. This has led to the development of a new language that supports all the concepts (van der Aalst, & ter Hofstede, 2005).

Figure 3-7 Example of Related Concepts and Language



Example 3-3 A Representation Relation

Figure 3-7 illustrates a complex representation relation. The figure shows some business process design concepts and a strongly simplified version of UML activity diagrams. It represents concepts in boxes. Lines represent relations between concepts, while dashed arrows represent the representation relation. The business process concepts consider tasks and two relations between tasks: the sequence relation, that prescribes that the affected tasks are performed in a sequence, and the choice relation, that represents that there is a choice between the affected tasks. UML activity diagrams consider nodes and directed edges between nodes. We consider action nodes and decision nodes. A decision node represents that a decision takes place after the actions are performed from which edges are directed to the decision. After the decision one of the actions is performed to which edges are directed from the decision. Edges in a UML diagram are represented by arrows, actions by rounded rectangles and decisions by diamonds. We use UML actions to represent business process tasks and UML decisions to represent business process choices. However, a business process choice directly specifies the

tasks between which it represents a choice, while a UML decision is related to UML actions by means of UML edges. Therefore, the representation relation needs to take into account that a business process choice is represented as a composition of a UML decision with edges leading to and from it. Hence, the representation relation specifies that for each instance of the UML action concept there exists an instance of the task concept. It also specifies that for each instance of the UML 'DecisionNode' concept with instances of the 'ActivityEdge' concept leading to and from it, there exists an instance of the choice concept. This choice instance relates task instances in a similar way as the 'ActivityEdge' instances relate action instances.

3.2 Tool-Support for Multi-Viewpoint Design

The principles for multi-viewpoint design that we explained in the previous section can be implemented in design tools. We implement the principles for multi-viewpoint design as syntactic manipulations of models. We use existing standards for both the syntax and the syntactic manipulations as much as possible. At the end of the chapter, we give an overview of a tool suite for multi-viewpoint design, in which the proposed syntax and syntactic manipulations are used.

3.2.1 A Standardized Syntax to Define Concepts and Modelling Languages

We claim that a standardized syntax is needed to define concepts and modelling languages. We fulfil this need by using the Meta-Object Facility (MOF) that is standardized by the Object Management Group (2002a).

The need for a standardized syntax. Just as we need concepts and model elements to create a common understanding for constructing views and models, we need meta-concepts to create a common understanding for constructing viewpoints and modelling languages. This is especially important if we want to relate (views from) different viewpoints. Because, if different viewpoints are defined using the same meta-concepts, we can use techniques that are defined on the meta-concepts to relate these viewpoints.

Definition 3-10 Meta-concept

A meta-concept is a concept that is used to define other concepts.

A concept is an instance of a meta-concept or a composition of instances of meta-concepts. Typical meta-concepts are 'concept', 'attribute' and 'composition rule'. For example, to define the business process viewpoint from Figure 3-7, we create the instances 'task' and 'sequence' of the meta-concept concept. Hence, 'task' and 'sequence' are themselves concepts. We

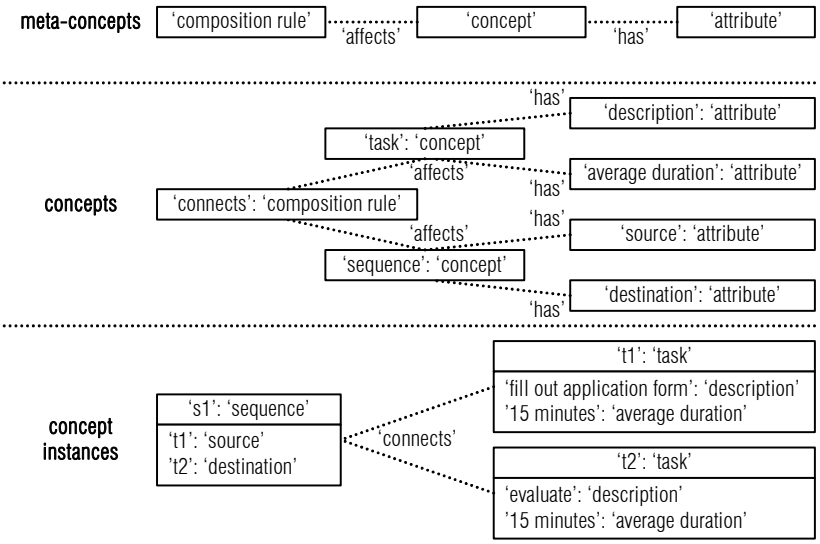


Figure 3-8 The Means to Define Concepts and their Instances

create the instances 'description' and 'average duration' of the 'attribute' meta-concept that we associate with the 'task' concept. Also, we create the instances 'source task' and 'destination task' of the 'attribute' meta-concept that we associate with the 'sequence' concept. Figure 3-8 illustrates the relation between meta-concepts, concepts and concept instances, using the business process viewpoint example.

Just as we need meta-concepts to define concepts, we need meta-meta-concepts to define meta-concepts, meta-meta-meta-concepts to define meta-meta-concepts and so on. However, if we follow this line of reasoning, we would always have to define the next meta-level and we would never get around to the actual design. To solve this problem, we define meta-concepts as instances of themselves. This still requires us to postulate a set of meta-concepts. However, these meta-concepts are defined as instances of themselves. Hence, they are their own meta-meta-concepts and we do not need to define a set of meta-meta-concepts. For example, we can de-

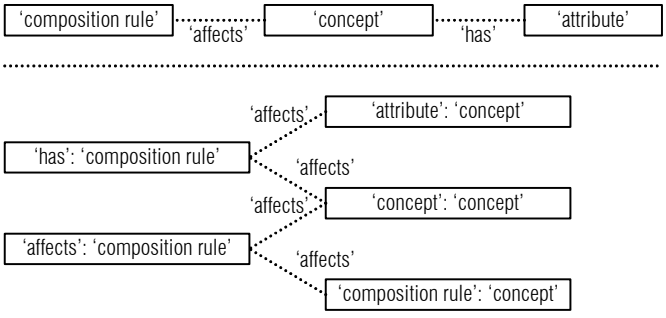
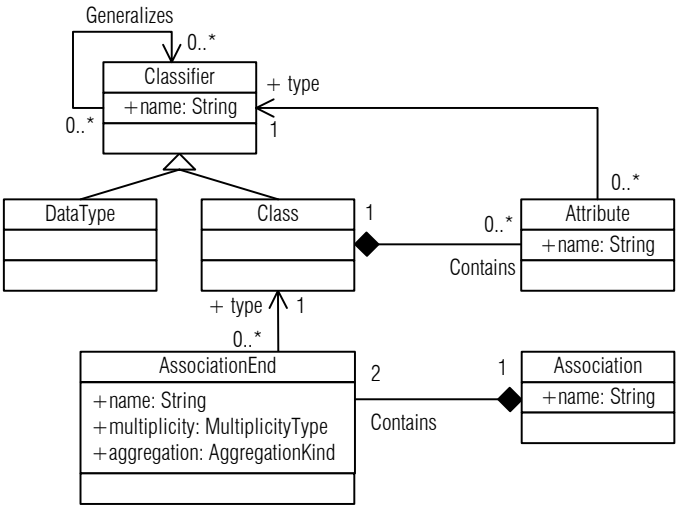


Figure 3-9 How we Define Meta-Concepts in Terms of Themselves

fine the meta-concepts 'concept' and 'attribute' as instances of the meta-concept 'concept' and relate them via the instance of the 'composition rule' meta-concept: 'a concept can have a number of properties'. Figure 3-9 illustrates how meta-concepts can be defined as instances of themselves.

The Meta-Object Facility. We use the MOF to define the concepts for our viewpoints and modelling languages syntactically. A benefit of using the MOF is that it associates its meta-concepts with a graphical modelling language, namely a specialized version of the UML (Object Management Group, 2004b), that is widely supported by design tools. Therefore, it does not only provide us with a set of meta-concepts for defining our concepts syntactically, but also with a modelling language and tools that we can use for doing that. In addition to a graphical language, the MOF associates a textual language with its meta-concepts (Object Management Group, 2002b). This textual language can be used for interchange of designs and meta-designs between tools.

Figure 3-10 MOF Concepts



Here, we introduce the meta-concepts from the MOF that we use further on in this thesis and the graphical elements that are associated with these concepts. The MOF meta-concepts are represented in Figure 3-10 in a strongly simplified version. For further detail, we refer to the MOF specification.

The 'Class' concept represents the meta-concept 'concept' from Figure 3-8. Hence, we can instantiate 'Class' to represent concepts. An instance of a 'Class' has a unique name and is graphically represented as a box with three compartments, the name of the concept represented in the top compartment. The graphical representation of a 'Class' instance is illustrated in Figure 3-11.i.

An instance of a 'Class' can contain instances of 'Attribute'. 'Attributes' of a 'Class' correspond to the attributes that are associated to meta-concepts in Figure 3-8. An instance of 'Attribute' represents a system property, of which the value can vary per concept instance. It has a name that identifies it uniquely in the context of a 'Class' instance. It also has a type that represents the degree of variation. To this end, a type identifies another 'Class' instance or an instance of 'Data Type'. This 'Class' or 'Data Type' instance identifies the set of values that the attribute instance can take. These values represent the properties that the attribute instance can represent. 'Data Type' is a particular kind of concept that represents (structured) information, such as numbers or documents. 'Attribute' instances are graphically represented in the second compartment of a 'Class' instance, by names followed by a semi-colon and the name of their type. Figure 3-11.ii illustrates this.

An instance of 'Association' represents a composition rule between concepts, similar to the composition rule meta-concept from Figure 3-8. If an 'Association' instance exists between two concepts, the instances of those concepts can be composed via an instance of that 'Association'. As will be explained below, we can specify additional rules that apply to instances of an 'Association'. An 'Association' instance has a name and two instances of 'Association End' that represent the concepts that participate in the 'Association' instance. The concepts at the 'Association End' instances are the types of those instances. An 'Association End' instance has a name. It also has a multiplicity that determines how many instances of the concept at that end can be related to each concept instance at the other end. The multiplicity of an 'Association End' instance is determined by a minimum and a maximum number. An 'Association' instance is graphically represented as a line that is attached to the 'Class' instances that represent the concepts that it associates. We draw the name of the 'Association' instance somewhere close to the line. The attachment of the line to a 'Class' instance represents an 'Association End' instance. We draw the name and multiplicity of the 'Association End' instance close to it. A maximum multiplicity of * repre-

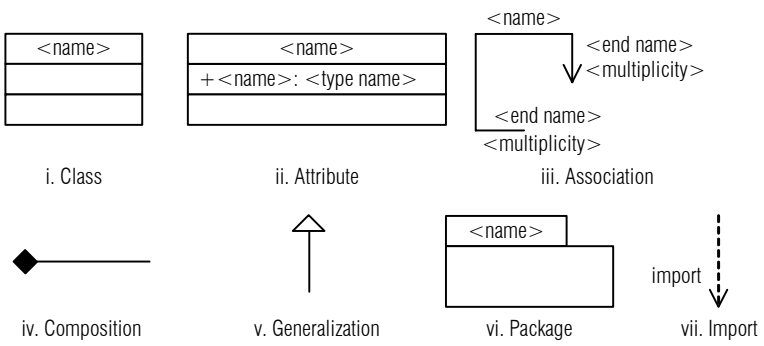


Figure 3-11 Graphical Representation of MOF Concepts

sents that there is no maximum. If the minimum and the maximum multiplicity is the same, they are represented by a single number. Figure 3-11.iii illustrates how an 'Association' instance is represented graphically. A special kind of association is 'Composition'. A 'Composition' instance represents that each concept instance at the 'Association End' instance with 'aggregation' set to 'composite' is a composition of concept instances from the other 'Association End' instance. We represent a 'Composition' graphically by drawing a diamond at the composite end of the 'Association' instance. This is illustrated in Figure 3-11.iv. An 'Association' instance can be directed, representing that it should be read *from* one end *to* another. This is graphically represented by an arrowhead at the *to* end. The

MOF considers that a concept can be a generalization of another concept. This is represented by a 'Generalization' instance from the more general to the more specific concept. If a 'Generalization' instance exists between two concepts, each instance of the specific concept has a generic counterpart in an instance of the generic concept. A 'Generalization' instance is graphically represented by a line with an open arrowhead at the more general end, as illustrated in Figure 3-11.v. If a 'Class' instance is a more specific version of another, more general, 'Class' instance, it inherits all 'Attribute' and 'Association' instances of the more general 'Class' instance.

The definition of a set of concepts can be structured by the MOF by putting them and their relations inside instances of 'Package'. Concepts can only 'see' concepts from the same 'Package' instance and 'Association' instances can only be defined between concepts that can 'see' each other. A 'Package' instance makes concepts from another 'Package' instance visible, if it contains or imports that 'Package' instance. A 'Package' instance has a name. It is graphically represented as a box that contains 'Class', 'Package' and Association' instances. This is illustrated in Figure 3-11.vi. A 'Package' instance that imports another 'Package' instance is graphically represented as a dashed line from the importing to the imported 'Package' instance. Figure 3-11.vii illustrates this. In case a 'Class' or 'Association' instance is contained in a 'Package' instance, it can be referenced inside a 'Package' instance that contains or imports that 'Package' instance, using: '<name of package instance name that contains the concept or association instance>::<name of concept or association instance>'.

Figure 3-10 both represents the MOF meta-concepts and instances of the MOF meta-concepts, because the MOF concepts are instances of themselves. Hence, it also illustrates how the MOF meta-concepts can be used to define concepts.

If we define concepts merely as instances of meta-concepts, we only define the terms that we use to identify concepts and the relations that these terms have with each other. Hence, we do not define the semantics of the

concepts. The semantics must define precisely what system properties are represented by each concept.

3.2.2 Syntactical Means to Relate Viewpoint Concepts to Basic Concepts and to Modelling Languages

Chapter 1 explains that a viewpoint concept can be defined as a composition of basic concepts. A viewpoint concept can also be a *specialization* of a single basic concept or a composition of basic concepts. If a viewpoint concept is a specialization of a single concept or a composition of concepts, it describes more properties than that concept or composition. For example, the business task concept can be defined as a specialization of the basic concept action. It represents all the properties that the action concept represents as well as the property that the business task occurs in some business environment.

The syntactical constructs of the MOF that we use to represent the relations between viewpoint and basic concepts are *generalization* and *transformation*. The same syntactical constructs have been used to relate viewpoints to modelling languages by Akehurst, Derrick, & Waters (2003).

Relating via generalization. We can use the 'Generalization' from the MOF to represent that a viewpoint concept is a specialization of a single concept. This concept may, in turn, represent a composition of concepts, such that we can represent a specialization of a composition. We represent that a viewpoint concept is a specialization of another concept, by drawing an instance of 'Generalization' from the more specific viewpoint concept to the more general concept.

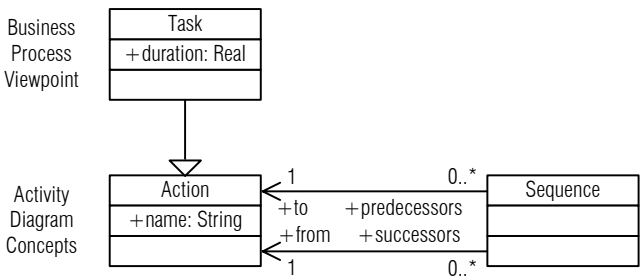
We can also use 'Generalization' to relate viewpoint concepts to language concepts by defining the viewpoint concepts as specializations of the language concepts. As with relating viewpoint concepts to basic concepts via generalization, it is the responsibility of the designer to ensure that, semantically, the viewpoint concepts are specializations of the language concepts that they specialize.

The viewpoint concepts can introduce additional attributes and associations with respect to basic and language concepts, because they are defined as specializations. The attributes represent the additional system properties that the viewpoint concept considers, while the associations represent additional composition rules. Also, the viewpoint specific concepts can define additional constraints on how their instances can be composed. Using these mechanisms, the stakeholders can tailor their basic and language concepts to match the required viewpoint concepts.

Ideally, we want modelling tools to support generalization between viewpoint concepts and modelling languages, such that when we draw a

notational element there is a one-to-one correspondence with the viewpoint concept instance that it represents. However, there are two problems with this. Firstly, the notational elements are associated with language concepts rather than viewpoint concepts and, in case a viewpoint concept is a specialization of a language concept, it can be traced to that language concept, but not the other way around. Hence, if we draw a notational element, the tool has no way of knowing what viewpoint concept we mean. For example, consider a situation where we have multiple concepts from multiple viewpoints as specializations of the same language concept. In that case, if we draw the notional element that is associated with that language concept, which of the specializing viewpoint concepts does that language concept represent? Secondly, the associated language concepts do not define the attributes that are only part of the specialized viewpoint concepts. Hence, these attributes are not graphically represented by the notational elements. For these reasons, we allow a notational element to be annotated with the name of the specialized viewpoint concept that it represents and values for the attributes that viewpoint concept has. We derived this technique of relating viewpoint concepts to modelling languages from the technique of 'Stereotyping' that can be used to specialize concepts that are defined by the Unified Modelling Language. Figure 3-12.i shows an example of how the UML activity diagram concept 'Action' could be specialized to represent the business process viewpoint concept 'Task' that has an additional attribute 'duration'. Figure 3-12.ii shows what the graphical representation of a business process view would then look like.

Relating via transformation. We can use model transformation specifications to represent that a viewpoint concept is (a specialization of) a composition of basic concept. We can do that by defining a transformation from



i. Business process concepts as specializations of activity concepts

ii. A model in the annotated notation

Figure 3-12 Concepts Related via the Generalization Relation

each of the viewpoint concepts to the basic concepts of which it is a composition. A transformation is a set of rules that define how, given some concept instances, instances of other concepts must be generated. We can also use this technique to relate modelling languages to viewpoints. Such an approach is, for example, proposed by Akehurst, Derrick, & Waters (2003). They propose this approach to use UML modelling languages to represent viewpoints from RM-ODP.

At the time of writing, a standard transformation technique and language have not been defined for the MOF. However, standardization is in progress (Object Management Group, 2002c) and early versions of transformation techniques and languages are available (Object Management Group, 2005; Akehurst, Kent, & Patrascoiu, 2003). In the remainder of this thesis we use the transformation language YATL (Patrascoiu, 2004a; Patrascoiu, 2004b), which stands for Yet Another Transformation Language, because tool support exists for this language and because it is compliant to the MOF. Since YATL is compliant with the MOF, it can be used to define transformations for designs that have their concepts defined syntactically in the MOF. Here, we explain the YATL features that we use in this thesis.

YATL makes extensive use of the Object Constraint Language (OCL) (Object Management Group, 2003c), a language that can be used to describe constraints on how concepts can be used. It can also be used to query a design to verify that a constraint holds on that design. YATL uses OCL expressions to yield a particular set of concept instances as indicated by the query. OCL can be used on concepts that are defined using the MOF meta-concepts. Hence, we can define expressions and queries over designs that are compositions of instances of concepts that are defined syntactically with the MOF meta-concepts. Here, we only explain some basic OCL properties.

OCL. An OCL constraint is defined in a context. Here we consider only concepts as contexts. If an OCL constraint is defined in the context of a concept, it must hold for each instance of that concept. Each time the constraint is evaluated for an instance of the concept, the variable named `self` refers to that instance.

To specify an OCL constraint, we must specify a truth expression. To this end, we can use:

- Integer and Real operators and relations, such as `+`, `-` and `=`, `<`;
- Boolean operators and relations, such as `and`, `xor` and `=`;
- String operators and relations, such as '$string_1$.concat($string_2$)'; and
- Set operators and relations, such as 'set->includes(element)', 'set->isEmpty()' and 'set_1->includesAll(set_2)'. Operations and relations on sets must always be prefixed by the set to which they are applied and the symbol '->'.

OCL expressions can yield the attribute values of a concept instance and the concept instances with which a concept instance is composed via association instances. An attribute value of a concept instance is the result of the expression '<concept instance>.<attribute name>'. The set of concept instances to which a concept instance is associated can be obtained using the expression '<concept instance>.<association end name>', where the association end name is the name of the other end of an association in which the concept instance participates. If the association end has no name, the name of the concept at the other end can be used. If the multiplicity of the association end has a maximum of 1, then a single concept instance is returned instead of a set of concept instances. We can also get the set of attribute values of a set of concept instances, using the expression <set of concept instances>.<attribute name>.
Similarly, we can get the set of concept instances to which a set of concept instances is related, using <set of concept instances>.<association end name>.

Example 3-4 OCL Expressions

An example of an expression that returns attribute values and related concept instances, is the expression, defined on an instance of the 'Action' concept from Figure 3-12, that yields the names of all actions that follow that action sequentially:

```
self.successors.to.name
```

An example of an OCL constraint, applied to the meta-model from Figure 3-12, is the constraint that states that a sequence cannot be from and to the same 'Action':

```
context Action inv: self.from <> self.to
```

In this expression <> yields true if its arguments are not equal. inv stands for 'invariant', used here as synonym for constraint.

YATL. A YATL transformation has a name and consists of a set of transformation rules. These rules are performed in the order in which they are invoked by the rule that is declared the start rule. Each rule has a name, it optionally has a match part and it has a body part. The match part identifies a MOF 'Class' by its name and optionally defines an OCL expression over that concept. The body part of the rule is evaluated for each instance that is selected in the match part. For each execution of the body part self takes the values of one of these instances.

The body part contains a sequence of statements that must be performed. A let statement, 'let <name>: <classifier name>', declares a variable by the given <name> of the type given by the <classifier name>. An assignment statement, '$<expr_1>$:= $<expr_2>$', assigns the value of '$<expr_2>$ to $<expr_1>$'. A track statement is used to store and recall relations between concept instances. 'track($<ci_1>$, <relation name>, $<ci_2>$)' stores a relation between the concept instances $<ci_1>$ and $<ci_2>$ in the set with the name <relation name>. We also refer to this set as the tracking relation. The tracking relation must be functional, such that each $<ci_1>$ can be assigned to at most one other concept instance.

'track(<ci_1>, <relation name>, null)' returns the concept instance that is related to concept instance <ci_1> by the tracking relation that is identified by <relation name>. A tracking relation is visible in each rule in an entire transformation. A new statement, 'new <class name>', creates a new instance of the MOF 'Class' by the specified name.

YATL transformations can be structured by defining them in the context of namespaces. A namespace identifies the MOF 'Packages' that contain the MOF 'Classes' that are the source and the target of the transformation, respectively.

Example 3-5 A Transformation in YATL

Figure 3-13 shows an example of a transformation in YATL. It creates an instance of the 'Action' concept from figure Figure 3-12 for each instance of the 'Task' concept. The created 'Action' has the same name as the 'Task' from which it is created. The transformation assumes that the 'Action' concept is declared in a 'Package' called 'ActPackage' and the 'Task' concept is declared in a 'Package' called 'BPPackage'.

Figure 3-13 Example of a Transformation Declaration in YATL

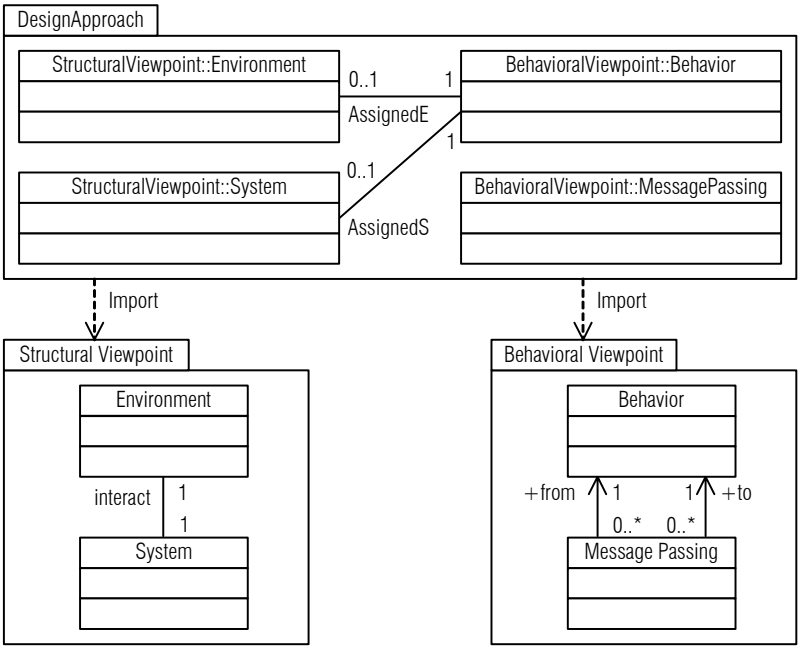
```
start example::transformationname::main;
namespace example(BPPackage, ActPackage){
    transformation transformationname{
        rule task2action match BPPackage::Task(){
            let action: ActPackage::Action;
            action := new ActPackage::Action;
            track(self, task2action_relation, action);
        }
        rule taskname2actionname match BPPackage::Task(){
            let action: ActPackage::Action;
            action := track(self, task2action_relation, null);
            action.name := self.name;
        }
        rule main(){
            task2action();
            taskname2actionname();
        }
    }
}
```

3.2.3 Syntactical Means to Define View Relations

To represent relations between viewpoints, we define 'Association' instances between MOF 'Class' instances that represent the concepts of the corresponding viewpoints. Then, we can represent the relations between (concept instances from) views, using instances of these 'Association' instances. Using this approach, a part of the view relations is defined at a view level, while another part is defined at a view*point* level, as explained in section 3.1.2. At a viewpoint level we can represent additional requirements on how (concept instances from) views must be related, using multiplicity

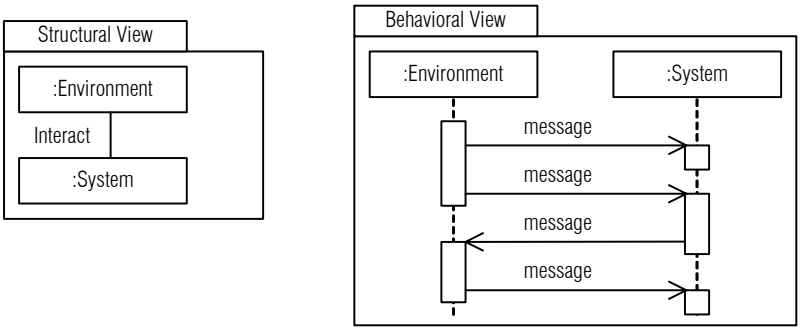
and OCL constraints. This approach can be combined with both ad-hoc and pre-defined view relations, as explained in section 3.1.2.

Figure 3-14 Viewpoints, Views and Their Relations



context Behavioral Viewpoint::Behavior inv:
not ((self.Environment->size() = 0) and (self.System->size() = 0))

i. Viewpoints and their associations



ii. Views and their associations

Example 3-6 Related Viewpoints

Figure 3-14 shows an example of a structural viewpoint and a behavioural viewpoint that are related in a design approach. The viewpoints are represented as 'Packages'. The structural viewpoint defines the 'System' and 'Environment' concepts that are related by an 'Interact' association, representing that interacting systems and environments can exist in a design. The behavioural viewpoint defines the 'Behaviour' and 'Message Passing' concepts that represent behaviours and messages that are passed from one behaviour to another, respectively. The

'Design Approach' package represents the pre-defined relations that exist between views. The relations between the viewpoints relates environments to their behaviour and systems to their behaviour. Multiplicity and OCL constraints represent the additional requirements that an environment and the system each have one behaviour and that a behaviour can either be the behaviour of a system or the behaviour of an environment, but not both. Figure 3-14.ii shows two views according to these viewpoints, represented as an object and a sequence diagram. The object diagram shows that there is exactly one system and one environment that interact. Behaviours are represented by the lifelines from the sequence diagrams. Lifelines are associated to objects, thus representing the relation between the views according to the predefined assignment relations.

3.2.4 Syntactical Means to Define and Verify Consistency Rules

During our state-of-the-art analysis we identified four approaches (Dijkman, Quartel, Ferreira Pires, & van Sinderen, 2003) to represent and verify consistency rules that accompany view relations.

Approach 1. The first approach is to represent consistency rules in natural language along with the relation between the views or viewpoints. In this approach consistency of the views is made plausible by informal reasoning in natural language. Hence, the consistency verification must remain simple, because for complex consistency verification mathematical models and tools are necessary. As an example consider the behavioural and structural view from Figure 3-14. A consistency rule between these views could be that messages can only be passed in the behavioural view, if the behaviours between which they are passed belong to a system and an environment that are interconnected in the structural view.

Approach 2. The second approach to represent consistency rules is to represent them in terms of expressions in a relational algebra over concept instances and relations between those instances. To do this, we interpret concepts as sets (of their instances) and relations between concepts as relations over these sets. We also consider the relations between the views, defined as relations between the concept instances from these views, as relational algebraic relations. Subsequently, we can express constraints on these sets and relations. Languages such as the Object Constraint Language are specifically meant for describing relational algebra-like expressions on designs, views and models.

As an example consider the behavioural and structural view from Figure 3-14 and the consistency rule from the previous paragraph that: messages can only be passed in the behavioural view, if the behaviours between which they are passed belong to a system and an environment that are interconnected in the structural view. We can specify this constraint in OCL as follows:

```
context BehavioralViewpoint::MessagePassing inv:
    (self.from.System.Environment = self.to.Environment.System) or
    (self.to.System.Environment = self.from.Environment.System)
```

This approach can only be used in combination with relations that are defined at a viewpoint level.

Approach 3. The third approach to represent and verify consistency rules between views, assumes that the views have a common formal semantics.

Definition 3-11 Formal Semantics

A formal semantics is a mapping from a set of concepts and their relations to a mathematical model.

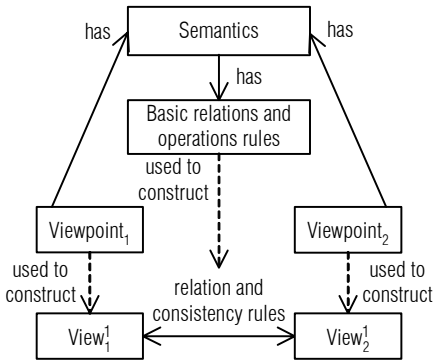


Figure 3-15 Another Approach to Describe Consistency Rules between Views

We also call such a mathematical model a *formalism*. The mapping rules are defined at a viewpoint level and can be used to transform each view into a mathematical model. The relations and operations that are defined on the formal semantics can be re-used in the views and viewpoints to represent relations between them. Figure 3-15 illustrates this approach. We cannot directly apply relations and operations that are defined on the formalism to concept instances from the views. Instead, we have to apply them to the *mathematical representation* of those concept instances, as it is defined by the mapping rules. We use the mathematical representation operator [ci] to obtain the mathematical representation of a concept instance ci.

The second approach can be considered a special case of this approach, where the formal semantics is a relational algebraic model and the basic relations and operation are relational algebraic relations and operations. The third approach is more general, because it can also use other formalisms as a semantics.

We can use more than one formalism to define consistency rules between views. In that case, the operations and relations from each formalism to which views are mapped, are available. However, since this means that a

single concept instance can have more than one mathematical representation, we must indicate which mathematical representation of concept instance ci we mean in the mathematical representation operator [ci]. We do this by subscripting the operator with the name of the formalism: $[ci]_{name}$. Since the second approach is a special case of this approach, we can also combine the second approach with this approach, such that consistency rules can both use operations and relations from a mathematical algebra and operations and relations from other formalisms to define consistency rules.

Example 3-7 Describing Consistency Rules via a Formal Semantics

As an example of relating views via the third approach, suppose that we have a view that represents the entire behaviour of a system, containing (among others) the concept instance behaviour, and another view that represents some scenarios that we want the system to perform, containing (among others) the concept instances 'scenario$_1$' and 'scenario$_2$'. A consistency rule between these views is that both 'scenario$_1$' and 'scenario$_2$' can be performed by 'behaviour'. Suppose that we have a formal semantics of both behaviours and scenarios in terms of Petri nets, and a relation $<$, such that, for a Petri net p_1 and a Petri net p_2: $p_1 < p_2$, if each possible execution of p_1 is also an execution of p_2. Then we can prescribe the consistency rule as:

$[scenario_1] < [behaviour]$ and $[scenario_1] < [behaviour]$.

Approach 4. The fourth approach to represent consistency rules assumes that each of the viewpoint concepts is defined as a composition of basic concepts. The definition of viewpoint concepts as compositions of basic concepts, provides a mapping from those viewpoint concepts to the basic concepts. Hence, similar to the use of a formal semantics, the basic concepts provide a common semantics for the various viewpoints. The relations and operations that are defined on that semantics can be reused to define view relations. Figure 3-15 illustrates this. Also similar to the use of a formal semantics, we cannot apply relations and operations that are defined on the basic concepts directly on the viewpoint concepts. Therefore, we use a general representation operator $[<ci>]_{<name>}$ that yields the representation of a concept instance <ci>, either in terms of a mathematical model or in terms of basic concepts as identified by the mapping named <name>.

The third and fourth approach are very similar. The third approach involves a mapping of viewpoint concepts to mathematics, while the fourth approach involves a mapping of viewpoint concepts to basic concept. However, we can observe that formalisms are often aimed towards use for particular concerns and levels of abstraction in a design. Therefore, as a foundation for verifying consistency between views a collection of formalisms may be required that each addresses its own concerns and levels of abstraction. This presents us with the additional challenge of maintaining the consistency between *formalisms*. Furthermore, formalisms are mainly aimed to-

wards mathematical rigour, rather than easy of use and understanding, which are important qualities when trying to understand viewpoint concepts and how they are related.

For these reasons, we propose the use of a set of basic concepts as a common semantics for the viewpoints, while we provide the basic concepts with a formal semantics in terms of one or more formalisms. This approach has the benefit that, while the definers of the viewpoint concepts do not have to be concerned with the formalisms or their mutual consistency, the viewpoints still inherit the mathematical rigor and the analysis and verification techniques from the formalisms. The formalisms are completely shielded from the viewpoints by the set of basic concepts. Figure 3-16 illustrates this approach. It shows that the basic concepts provide the semantics for the viewpoints, while the basic concepts are associated with a formal semantics. The figure also shows that the formal semantics provides the formal means to define the basic relations and consistency rules. Consistency can then be verified in the tools that are provided with the formalisms.

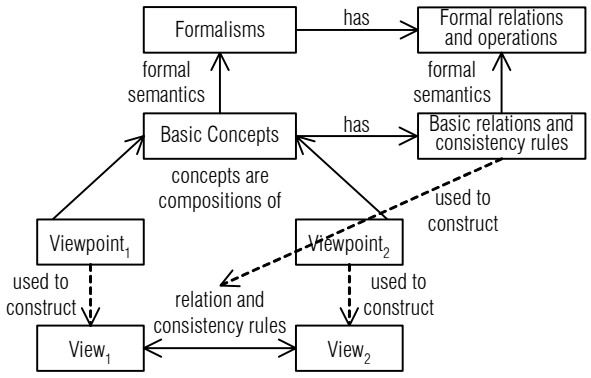


Figure 3-16 Proposed Approach to Describe Consistency Rules between Views

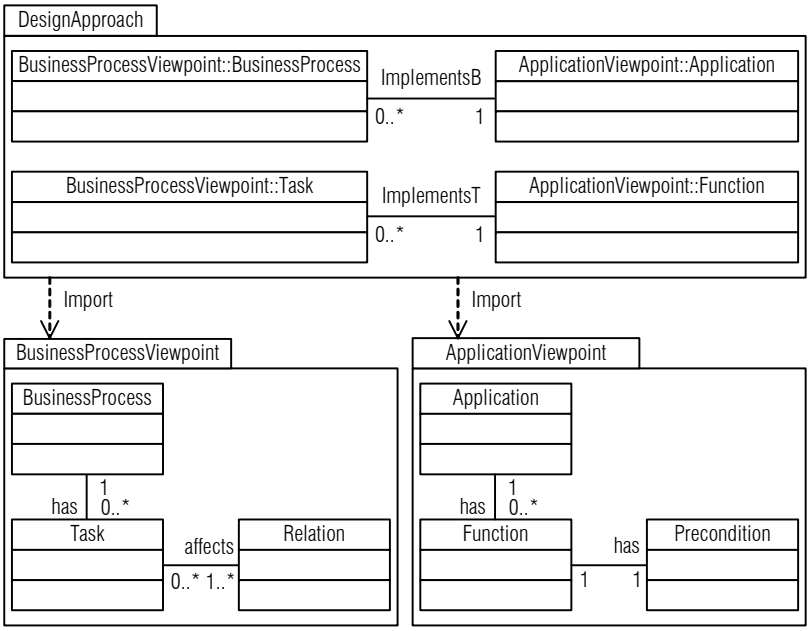
Combined approach. We use a combination of the second and fourth approach to consistency rule prescription. We represent consistency rules as OCL constraints. The benefit of using this approach is that it combines an established approach to describing constraints, namely OCL, with predefined consistency rules on the basic concepts.

We enable the re-use of pre-defined consistency rules, by adding these rules as operations on the basic concepts. Since OCL constraints can call operations on objects, this means that the consistency rules can be re-used in OCL constraints. For example, we define the abstract operator (that we explain in more detail in chapter 5) in as follows:

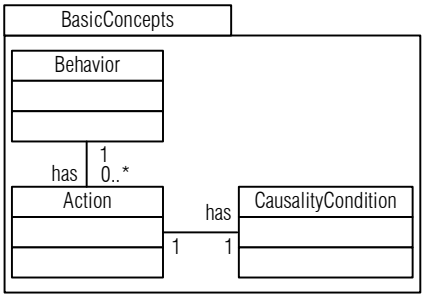
```
context BehaviourInstantiation def:
    abstract(as: Set): BehaviourInstantiation
```

The 'abstract' operator takes an instance of the 'BehaviourInstantiation' class and an instance of the 'Set' data type that contains instances of the 'ActionInstantiation' class. It returns an instance of the 'BehaviourInstantiation' class, which abstracts from the instances of the 'ActionInstantiation' class.

Figure 3-17 Viewpoints with Consistency Rules Defined via Basic Concepts



i. Viewpoints and their relations



ii. Basic concepts

```
context DesignApproach::BusinessProcess inv:
    self.bp2basic_bp2behavior.equivalent(
        self.Application.ap2basic_ap2behavior.abstract(
            (self.Application.has – self.has.Function).ap2basic_fun2action
        )
```

iii. Consistency rule

To be able to derive, for a concept instance, the basic concept instances that describe its semantics, we automatically transform YATL tracking relations into MOF associations. The name of the association end on the 'to' side of the tracking relation is the same as the name of the tracking relation. For example, example 3-6 defines a tracking relation 'task2action_relation' that allows one to track instances of 'Task' to instances of 'Action'. We transform this relation into a MOF association between the concepts 'Task' and 'Action', such that for an instance t of 'Task', t.task2action_relation yields the 'Action' to which it is related by the tracking relation.

Example. Figure 3-17.i shows an example of two related viewpoints, for which the consistency rules are defined via relations and operations that are defined on the basic concepts. The figure shows a business process viewpoint and an application viewpoint. The business process viewpoint contains concepts for representing the business processes and tasks in the organization, as well as relations between those tasks. The application viewpoint contains concepts for representing the application services in an organization, functions that can be performed by these application services and preconditions that must be met before the functions can be invoked. The viewpoints are represented by packages. These packages are imported in a package 'Design Approach' that represents the relations between the viewpoints that are represented by these packages. A relation exists that relates a business process to the application that implements it. This relation is represented by the 'ImplementsB' association. Another relation exists that relates a task to the application function that implements it. This relation is represented by the 'ImplementsT' association. To keep the example simple, we assume that each business process is completely implemented by a single application and that each task is implemented by a single application function. An application can still implement multiple business processes and a function can implement multiple tasks.

We assume that transformations exist by the names 'bp2basic' and 'ap2basic'. These transformations transform views from the business process and application viewpoint into designs in the basic concepts from Figure 3-17.ii. Business processes and applications are transformed into behaviours, tasks and functions are transformed into actions and relations and preconditions are transformed into causality conditions. Tracking relations, named 'bp2behavior', 'ap2behavior' and 'fun2action' store the relations between business processes and behaviours, applications and behaviours and functions and actions, respectively.

We can now specify the consistency rule from Figure 3-17.iii, in which the 'equivalent' operation represents an equivalence relation between basic behaviours. The rule states that, if an application implements a business process, then the behaviour of that application must be equivalent to the

behaviour of the business process, after we abstract from functions in the application that are not considered in the business process. Note that we assume that the equivalence relation knows the relation between tasks and the functions that implement them (for example because they have the same name).

3.2.5 Overview of a Tool Suite for Multi-Viewpoint Design

Based on the means that we described in this section to implement the principles for multi-viewpoint design, we provide an overview of a tool suite for multi-viewpoint design. Figure 3-18 shows that overview. It shows the software components that the tool suite should (at least) contain, the data that is stored by these components and the flow of data from one component to the other.

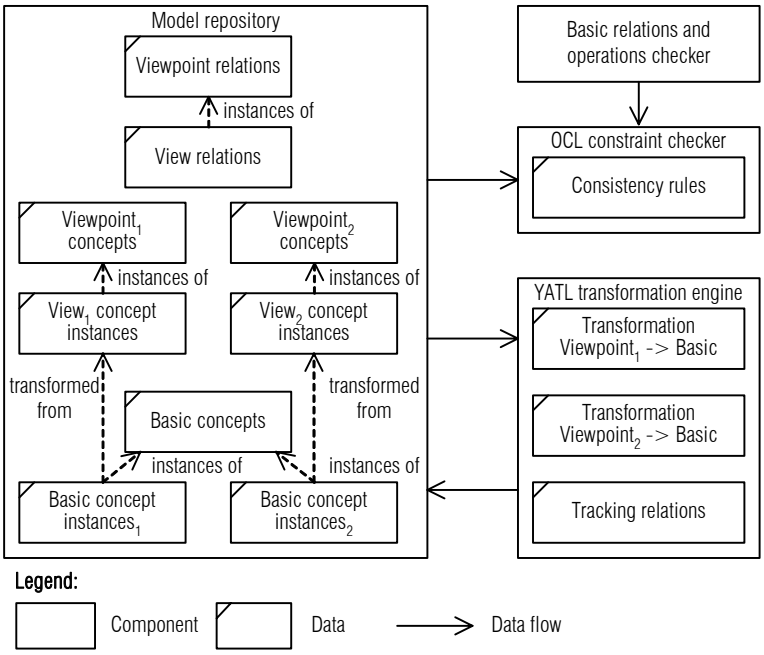


Figure 3-18 Overview of a Tool Suite for Multi-viewpoint Design

The tool suite contains a model repository in which the designer stores the concepts and their relations that the viewpoints define, as well as the concept instances and their relations that represent the views that are constructed according to the viewpoints. The designer also stores the view and viewpoint relations in the repository. The model repository is MOF compliant, in the sense that concepts and their relations are instances of MOF meta-concepts. We store our basic concepts and their relations, as we will define them in chapter 4, in the repository. The designs that are con-

structed by composing instances of these basic concepts can also be stored in the repository.

The YATL transformation component can transform views that are compositions of viewpoint concept instances into views that are compositions of basic concept instances. To this end it contains descriptions of such transformations. The transformation from view concept instances to basic concept instances is necessary to obtain the composition of basic concept instances in terms of which a view concept instance is defined. The YATL transformation component stores this relation between view concept instances and basic concept instances in tracking relations. These tracking relations are transformed into MOF associations that are stored in the MOF model repository (after the transformation was performed).

The OCL constraint checker can verify consistency rules that the designer prescribes in the form of OCL constraints. The OCL constraint verification component uses the following components:

1. The basic relations and operation checker to evaluate relations and operations that are defined on the basic concepts.
2. The model repository to obtain concepts and their relations as well as their instances and to obtain viewpoint relations and their instances.
3. The model repository to obtain the basic concept instances that belong to some view concept instance according to a tracking relation.

As an example, consider how the elements of the design approach from Figure 3-17 would fit into the overview from Figure 3-18, as illustrated in Figure 3-19. The concepts from the business process and application viewpoints that Figure 3-17.i defines would be stored in the model repository along with the relations between the viewpoints that the design approach defines and the basic concepts. Views that are defined according to the viewpoints could also be stored in the model repository. The consistency rule that is defined in Figure 3-17.iii would be stored in the OCL constraint checker. This consistency rule assumes the existence of transformations 'bp2basic' and 'ap2basic' and tracking relations 'bp2basic_bp2behavior', 'ap2basic_ap2behavior' and 'ap2basic_fun2actions'. These transformations and tracking relations must be stored in the YATL transformation engine. Associations that are created from the tracking relations are stored in the model repository.

To verify the consistency rule, the YATL transformation engine would first have to perform the transformations. This transforms the viewpoint concept instances into basic concept instances in the model repository. Also, it stores the relations between the viewpoint concept instances and the basic concept instances into which they are transformed into the tracking relations. After the transformations are performed, the consistency rule can be verified. To this end the OCL constraint checker must call:

1. The basic relations and operation checker to evaluate the equivalence relation (equivalent) and the abstraction operation (abstract).
2. The model repository to obtain the instances of the concepts (e.g.: 'BusinessProcess' and 'Application') and relations that are referenced in the OCL constraint (e.g.: 'DesignApproach::ImplementsB').
3. The model repository to obtain the basic behaviour instances into which business process instances are transformed (according to the tracking relation 'bp2basic_bp2behavior'), the basic behaviour instances into which applications instances are transformed (according to the tracking relation 'ap2basic_ap2behavior') and the basic action instances into which application functions are transformed (according to the tracking relation 'ap2basic_fun2actions').

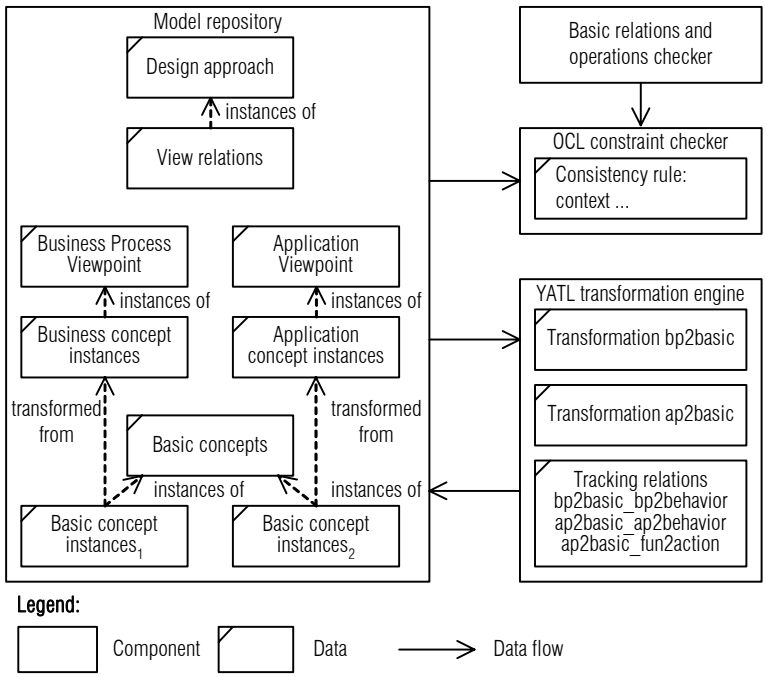


Figure 3-19 Example of a Multi-viewpoint Design in a tool

Chapter 4

Basic Design Concepts

This chapter defines concepts that can be used for distributed systems design across application domains and levels of abstraction. We categorize these concepts into structural, behavioural and information concepts. Structural concepts can be used to represent the structure of a distributed system, behavioural concepts to represent the behaviour of a distributed system and information concepts to represent the information that is manipulated by the system.

The concepts presented in this chapter are based on earlier work (Quartel, 1998; Quartel et al., 1997; van Sinderen, 1995; Ferreira Pires, 1994). We extended and improved these concepts on several points, as we explain further on in this chapter.

This chapter successively defines the structural, behavioural and information concepts, as well as the relations between these concepts.

4.1 System Structure and Structural Concepts

The structure of a system is the aggregate of the system's parts in their relationships to each other (adapted from (Merriam-Webster, 2005)). We consider two kinds of relationships between system parts. The first kind is the connection relationship that exists between parts that interact via some communication mechanism. The second kind is the part-whole relationship that exists between a system and its parts. We can consider each part as a (sub-)system. Therefore, this relation can also exist between a part (subsystem) and its sub-parts.

It is possible that the structure of a system changes during the lifetime of that system, because parts or relations between parts are added, removed or changed. For example, a router can be added or removed from the internet or a mobile user can disconnect from one wireless access point and connect to another. We call the stable structure of a system during a certain

time interval a structural *snapshot* of that system. If the structure of the system changes, it transitions from one snapshot to another.

Although the structure of a system can change, rules can apply to the system that restrict the allowed structures. We consider the allowed structures of a system by observing that parts in a system, as well as their relations, are of particular *types*. Restrictions apply to how parts of particular types can be connected and composed, using relations of particular types.

Finally, if we consider that the structure of a system can change during the lifetime of that system, we can also consider what causes these changes and when. We call this the *structural dynamics*.

A designer can use structural concepts to prescribe structural snapshots, allowed structures and structural dynamics. The concepts from (Quartel, 1998; Quartel et al., 1997; van Sinderen, 1995; Ferreira Pires, 1994) partly support the representation of these concerns. These concepts use entities to represent systems and parts, interaction points to represent connections between system parts and containment between entities to represent part-whole relations. We explain these concepts below in more detail and we explain how we extended them, to allow a designer to prescribe the allowed structures of a system. Also, we added precision to the part-whole relation by allowing a designer to prescribe that parts can implement connections of the whole. We leave the addition of concepts for representing structural dynamics for future work.

4.1.1 Structural Snapshot Concepts

To represent a snapshot of the structure of a system, we use the concepts from Figure 4-1.

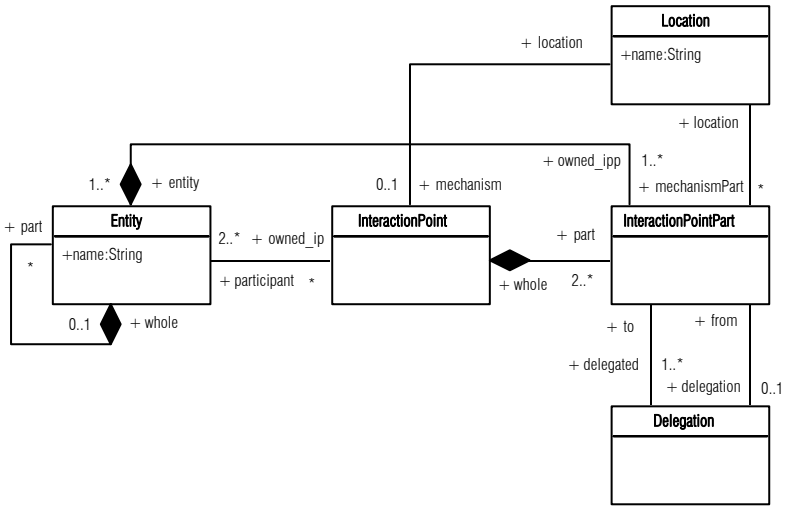
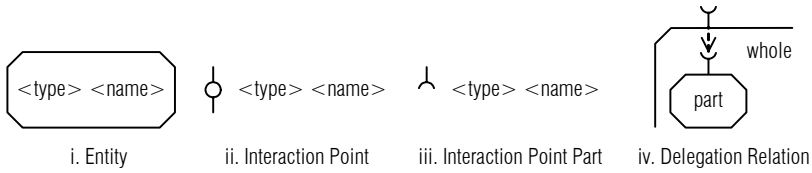


Figure 4-1 Structural Snapshot Concepts

Figure 4-2 Structural Snapshot Notation



Entity. We use the *entity* concept to represent a logical or physical part of a system that carries behaviour. Hence, components and objects can be represented as entities and business units and teams can also be represented as entities. An entity has a name that identifies it uniquely in the context of a design. An entity also has a type, which characterizes a collection of entities that are the same with respect to this type. This will be explained in more detail in subsection 4.1.2.

We represent an entity graphically as a box with cut-off corners, as shown in Figure 4-2.i. The entity's name and the name of the type of which it is an instance must be drawn inside the box. For brevity, the name of the entity's type can be omitted. In case the name of the entity's type is omitted, the entity's name must equal the name of the entity's type with some suffixed natural number (e.g.: $User_1$, where *User* is a typename).

Interaction point. We use the *interaction point* concept to represent a shared mechanism that two or more entities can use to interact. Such a mechanism consists of a part of these entities and some *means of interaction* that connects them. However, at the level of abstraction at which we represent the interaction point, we abstract from these constituents. For example, we can use an interaction point to represent an Ethernet connection between two computers. Such a connection consists of the Ethernet cards that are part of the connected computers and a means of interaction. This means of interaction can be as simple as a cross-link cable or as complex as a complete Ethernet network that consists of several cables, hubs and switches. However, at the level of abstraction at which we modelled the interaction point we do not have to consider that.

An interaction point is associated with a location at which the mechanism that it represents is accessible to all its participating entities. The location has a name that identifies it uniquely in the context of a design. Similar to an entity, a location has a type. We identify an interaction point by the name of the location with which it is associated.

We represent an interaction point graphically as a circle with lines protruding from it, as shown in Figure 4-2.ii. The lines that leave the circle must be attached to the entities that participate in the interaction point. The name of the location that identifies the interaction point and the interaction point's typename must be drawn close to the interaction point. For

brevity, the name of the type of the interaction point may be omitted, similar to an entity's typename.

Interaction point part. We use the *interaction point part* concept to represent an entity's participation in a shared communication mechanism. We need the interaction point part concept to represent that an entity is ready to form an interaction point, but does not yet do so. Each interaction point contains the interaction point parts of its participating entities and cannot exist without these interaction point parts. Similarly, an interaction point part cannot exist apart from an entity, since it is a part of an entity. An interaction point part has a location at which the mechanism that it represents is available to the entity that it is a part of. This location is the same as the location of the interaction point of which it is, or will become, a part. An interaction point part also has a type.

We represent an interaction point part graphically as a circle half with a line protruding from it, as shown in Figure 4-2.iii. The line that leaves the circle half must be attached to the entity of which the interaction point part is a part. The name of the location at which the interaction point part is available and the location's typename must be drawn close to the interaction point part. Like the name of an interaction point type, the name of an interaction point part type may be left out.

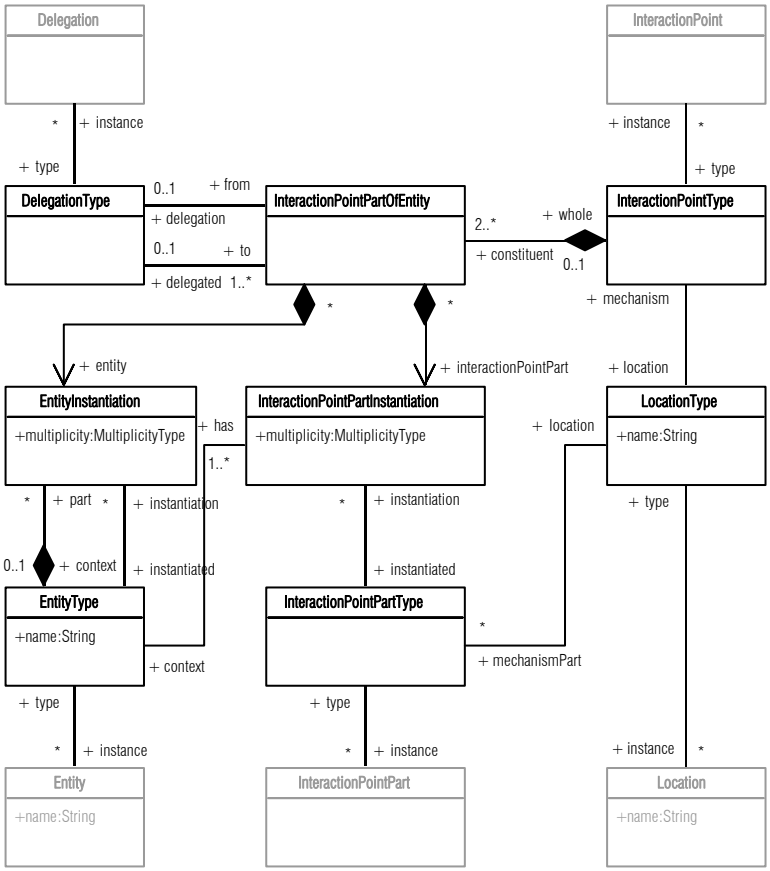
Composition. We use the *composition* relation to represent a part-whole relation in a system. The composition relation can be applied recursively, such that a part in one composition relation becomes a whole in another composition relation. In this way a sub-part can consist of sub-sub-parts and so on. The part-whole relation between the system and its parts can be represented explicitly, by representing the system as an entity and specifying a composition relation between the system and its parts.

We graphically represent the composition relation by drawing the entities that represent the parts inside the entity that represents the whole. The entities that represent the parts can have interaction points with each other, but they cannot have interaction points with entities that are not parts of the same whole.

Delegation. An entity's part in a communication mechanism can be implemented by one or more of that entity's constituents. For example, an Ethernet card implements a computer's part in an Ethernet connection. We also say that the entity that represents the whole *delegates* the implementation of the communication mechanism to one or more of its parts. We use the *delegation* relation to represent this. The delegation relation relates the interaction point part of the whole to the interaction point parts of the constituents that realize the communication mechanism.

We represent the delegation of an interaction point part graphically as a dashed arrow from the interaction point part of the whole to the interaction point parts of the whole's constituents, as shown in Figure 4-2.iv.

Figure 4-3 Structural Type Concepts



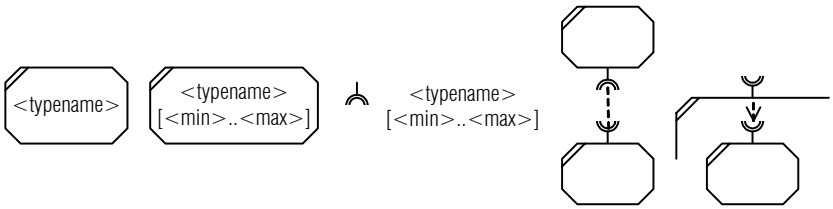

```
context DelegationType inv:
  let minimumMultiplicities = self.to->including(self.from)->collect(entity.multiplicity.lower*interactionPointPart.multiplicity.lower),
      maximumMultiplicities = self.to->including(self.from)->collect(entity.multiplicity.upper*interactionPointPart.multiplicity.upper) in
    minimumMultiplicities->forAll(m1, m2: Integer | m1 = m2) and
    maximumMultiplicities->forAll(m1, m2: Integer | m1 = m2)
context InteractionPointType inv:
  let minimumMultiplicities = self.constituent->collect(entity.multiplicity.lower*interactionPointPart.multiplicity.lower),
      maximumMultiplicities = self.constituent->collect(entity.multiplicity.upper*interactionPointPart.multiplicity.upper) in
    minimumMultiplicities->forAll(m1, m2: Integer | m1 = m2) and
    maximumMultiplicities->forAll(m1, m2: Integer | m1 = m2)
```

4.1.2 Structural Type Concepts

To represent the allowed structures of a system, we use the concepts from Figure 4-3. We included the structural snapshot concepts in grey, to represent the relation between the structural type and snapshot concepts.

The structural type concepts are based on the dichotomy between types and instances. A *type* represents a template, according to which we can create things. We call what is created an *instance* of that type and the process of creating it *instantiation*.

Figure 4-4 Structural Type Notation



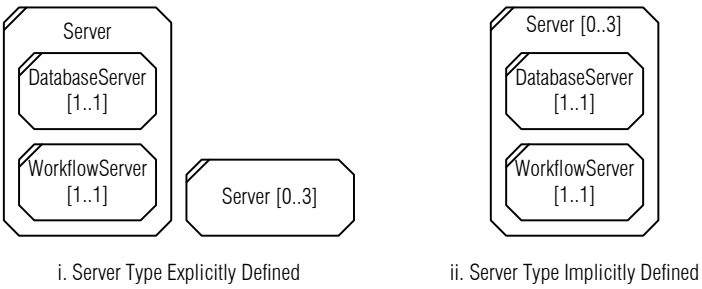
Entity type. An entity type specifies the properties that each of its instances, which are entities, will have. Specifically, it specifies the interaction point parts and the constituent entities that these entities will have. For example, a designer can define the entity type 'Workstation' according to which various workstation entities can be instantiated. The type defines the parts that each of the workstation entities has, namely a main board, a processor and a hard disk. It also defines the interaction point parts that each of the workstation entities has, namely a keyboard, a mouse, a screen and a network adapter. Later on, we associate an entity type with a behaviour type that represents the behaviour that entities of a particular type have. An entity type has a name that identifies it uniquely in the context of a design. We represent an entity type graphically by an entity symbol with an additional line in the top left corner, as shown in Figure 4-4.i. The name of the type must be drawn inside the box.

Interaction point part type. An interaction point part type specifies the properties that each of its instances, which are interaction point parts, will have. Specifically, it defines the locations at which its instances will be available. These locations are specified by associating an interaction point part type with a location type. The location type represents a template according to which locations can be instantiated. Each time an interaction point part is instantiated a location is instantiated along with it.

Instantiation. We use the instantiation concept to specify constraints on the possible structures of a system. An entity instantiation can be defined in

the context of an entity type or in the context of a design. An instantiation represents that entities of the instantiated type can be created in the associated context. If the instantiation is defined in the context of that entity, the instantiated entities will be contained in that entity. Interaction point part instantiations must be defined in the context of some entity type, representing that each entity of that type *can* instantiate interaction point parts of the instantiated type, but do not necessarily have to.

An instantiation must have a multiplicity. The multiplicity defines the minimum and the maximum number of instances that can be instantiated. The maximum number can also be undefined. For practical purposes we define the minimum and maximum number as 'Integers' and let an asterisk represent an undefined maximum number of created entities. In this way, entity and interaction point part instantiations specify the entities that can exist in the system and how many entities of a particular type can exist at any time. It also specifies the communication mechanism parts and the internal structure that entities can have.



i. Server Type Explicitly Defined

ii. Server Type Implicitly Defined

Figure 4-5 Multiple Instantiations with the Same Structure

We represent entity instantiation graphically by an entity symbol with an additional line in the top left corner, as shown in Figure 4-4.ii. The name of the type of the instantiation must be drawn inside the box, along with the minimum and maximum multiplicity. Entity instantiation in the context of an entity type is graphically represented by drawing the instantiation inside an instantiation of that type. For example, Figure 4-5.i represents that three servers may exist, each of which contains one database server and one workflow server. For brevity, we allow an entity type to be represented implicitly by one of its instantiations. For example, Figure 4-5.ii implicitly defines the server type, by prescribing that up to three instances of the server type can exist. We represent interaction point part instantiation graphically by a double circle half with a line protruding from it, as shown in Figure 4-4.iii. The line that leaves the interaction point part instantiation must be attached to an instantiation of the entity type in the context of which it is associated. The name of the type of the instantiation must be

drawn close to the instantiation, along with the minimum and maximum multiplicity.

Interaction point type. An interaction point type specifies the properties that each of its instances, which are interaction points, will have. Specifically, it defines the locations at which interaction points of its type will be available and the interaction point parts that are used to form the interaction point. To this end an interaction point type is associated with two or more interaction point part instantiations of particular entity instantiations. This represents that an interaction point *can* be formed by combining one interaction point part of each of these instantiations. Interaction point types do not specify how many interaction points *must* be created. If the designer wants to specify that, he must do so in additional constraints. We associate an interaction point type with an interaction point part instantiation *of* an entity instantiation, because interaction points are formed between interaction point part instances *of* entity instances. We represent an interaction point type graphically by a dashed line that connects interaction point part instantiations, as show in Figure 4-4.iv. Although the notation may suggest that the dashed line represents the means of interaction that the connected entities use to interact, this is not the case.

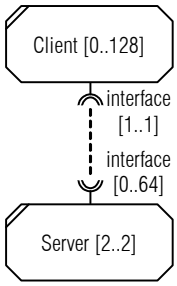


Figure 4-6 Example of Matching Multiplicities

Since interaction point parts that form an interaction point must have the same location, interaction point types must connect interaction point part instantiations that have the same location type. Also, the multiplicity of the interaction point part instantiations that are related by an interaction point type must match, because, if the multiplicities do not match, some interaction point parts can never be part of an interaction point. More specifically, because an interaction point type is associated with interaction point part instantiations *and* entity instantiations, the multiplicity of the interaction point part instantiation times the multiplicity of the entity instantiation should be the same for each combination that is associated with the interaction point type. Figure 4-3 represents this constraint in OCL.

Example 4-1 Matching Multiplicities

Figure 4-6 shows an example of a structure in which two servers exist that can each serve up to 64 clients via an interface. In the system up to 128 clients can exist, each of which has exactly one interface. The interaction point type connects the interface interaction point parts of the client and server type, representing that interaction points can be formed, each with one client and one server interface. Hence, the minimum multiplicity of the server instantiation times the minimum multiplicity of the server interface instantiation (2·0) must match the minimum multiplicity of the client instantiation times the minimum multiplicity of the client interface instantiation (0·1). Similarly, the maximum multiplicity of the server instantiation times the maximum multiplicity of the server interface instantiation (2·64) must match the maximum multiplicity of the client instantiation times the maximum multiplicity of the client interface instantiation (128·1). Suppose that each server could only serve up to 63 clients, then only 126 clients could be served. In that case, the multiplicities would not match and the model would not be well-formed.

Delegation type. A *delegation type* specifies the properties that each of its instances, which are delegations, will have. Specifically, it defines interaction point parts between which it delegates. To this end it is associated with a source interaction point part instantiation and one or more target interaction point part instantiations. This represents that each delegation of this type delegates from an interaction point part of the source instantiation to the interaction point parts of the target instantiations. A delegation type is graphically represented as dashed arrows from the source interaction point part instantiation to the target interaction point part instantiations, as show in Figure 4-4.v. Similar to interaction point types, delegation types must connect interaction point part instantiations that have the same location type. Also, the multiplicities of the connected interaction point instantiations must match. Figure 4-3 represents this constraint in OCL.

4.1.3 Examples

In two examples, we show how the structural type level can be used to describe the possible structures of a system and how the structural snapshot level can be used to represent the stable structure of a system during a particular interval in the system's lifetime.

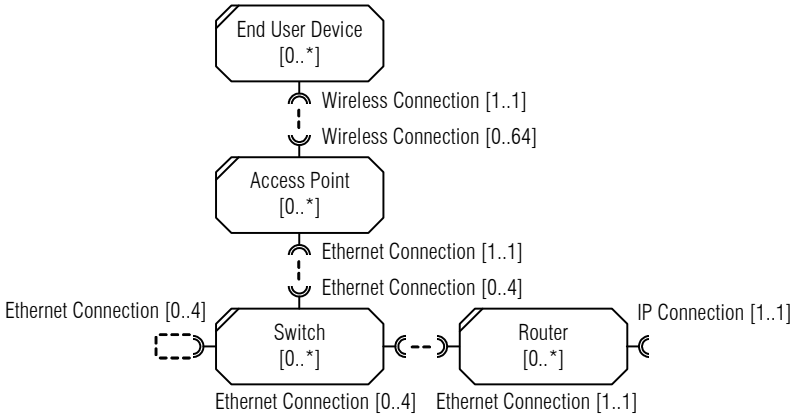


Figure 4-7 The Possible Structures of a Wireless Network

As an example, consider the wireless network of a fictitious company. This network consists of a number of end-user devices, such as laptops and workstations that are equipped with a wireless network card. The end-user devices can connect to wireless access points via the wireless network. Each wireless access point can serve up to 64 end-user devices at a time. The wireless access points are connected to Switches via an Ethernet connection. A Switch has four Ethernet ports to which access points, other Switches and Routers can connect. A Router has an Ethernet connection to

the wireless network and a connection to the rest of the network that we do not consider here. Figure 4-7 represents this structure graphically.

Note that the figure represents that a Switch can connect to four other Switches *and* four access points *and* four Routers, while we wanted to express that it can only connect to four other Switches, access points or Routers in total. Hence, the figure allows for more possible structures than we intended and the figure underspecifies the network. We could solve this problem by adding the constraint that "each entity of type Switch has exactly four interaction point parts of type Ethernet Connection". However, we did not define a language to express such constraints. This is left for future work. Figure 4-8 represents a possible structure of the system during a particular time interval.

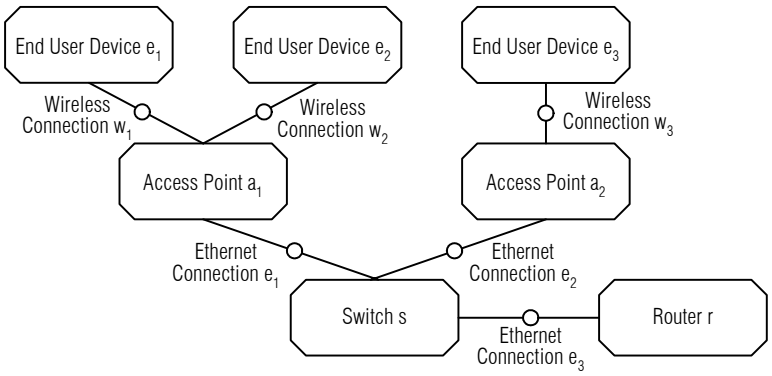


Figure 4-8 A Wireless Network during a Particular Interval

As another example, consider the system that consists of some interacting business partners. The goal of the system is to provide a mortgage to clients via a mortgage broker. In the system there is a single mortgage broker that communicates with clients that require a mortgage and with banks that provide mortgages. The broker selects the best mortgage for each client, based on the client's wishes and the mortgages that are available. To close the deal, the broker and the clients interact with a notary that draws up the final deed. Finally, the banks interact with the bureau of credit registration to verify if the client has a history of non-payment.

The broker itself is completely internet-based. It consists of a web-server to which clients can connect to inquire about a mortgage. The web-server interacts with a database in which information about available mortgages is stored. When the client selects a mortgage, the web-server interacts with a transaction manager that coordinates the interaction between the bank, the broker and the notary to draw up the final deed. Figure 4-9 represents the possible structures of the system graphically. It shows that we consider only one broker that connects to one notary and a number of banks and clients. Also, it shows the internal structure of the broker. Figure

4-10 shows a possible configuration of the system during a particular interval.

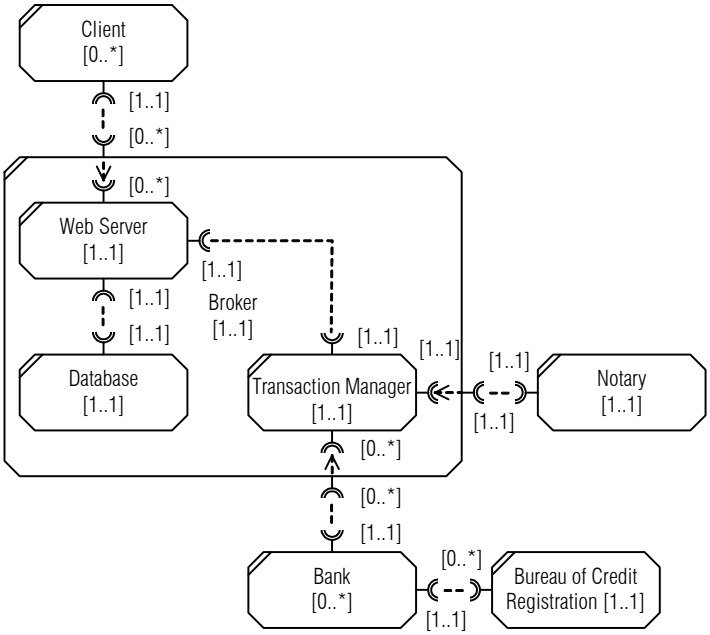


Figure 4-9 The Possible Structures of a Business Partnership

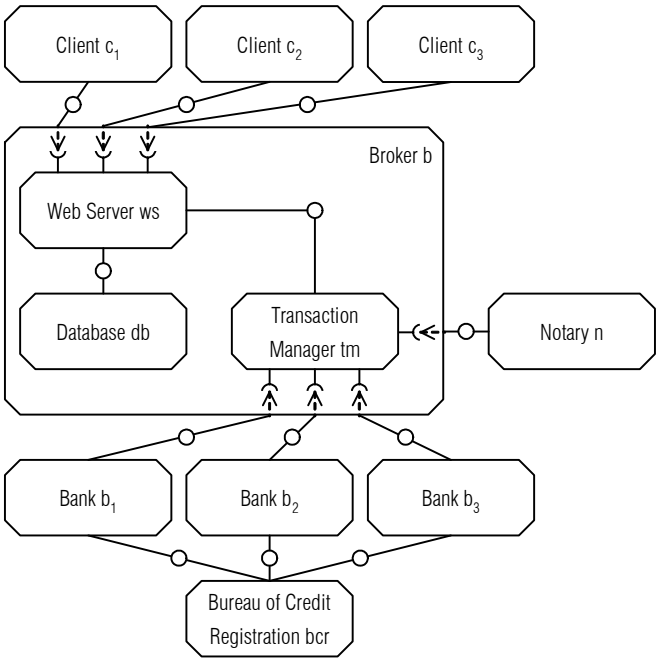


Figure 4-10 A Business Partnership during a Particular Interval

4.1.4 Connection Patterns

There are a number of frequently occurring patterns for entities to form interaction points. Table 4-1 shows the connection patterns that we consider. Since interaction points are formed between interaction point part *instantiations* of entity *instantiations*, the figure distinguishes patterns, based on the multiplicities of those instantiations. The combinations that are crossed out are impossible, because the multiplicities of the entity and interaction point part instantiations do not match.

Table 4-1 Connection Patterns between Two Entity Types

		entity instantiation of type E_1 with maximum multiplicity 1, owning:		entity instantiation of type E_1 with maximum multiplicity n > 1, owning:	
		interaction point part instantiation of type IPP_1 with maximum multiplicity 1	interaction point part instantiation of type IPP_1 with maximum multiplicity n > 1	interaction point part instantiation of type IPP_1 with maximum multiplicity 1	interaction point part instantiation of type IPP_1 with maximum multiplicity n > 1
entity instantiation of type E_2 with maximum multiplicity 1, owning:	interaction point part instantiation of type IPP_2 with maximum multiplicity 1	one-to-one			
	interaction point part instantiation of type IPP_2 with maximum multiplicity n > 1		replicated interaction points	one-to-many	replicated interaction points
entity instantiation of type E_2 with maximum multiplicity n > 1, owning:	interaction point part instantiation of type IPP_2 with maximum multiplicity 1		one-to-many	many one-to-one	many one-to-many
	interaction point part instantiation of type IPP2 with maximum multiplicity n > 1		replicated interaction points	many one-to-many	many-to-many

The one-to-one connection pattern represents that, in each possible structure, one entity of a particular type exists (or can exist in case of a minimum multiplicity of 0) that can connect to one entity of another type via

one interaction point. Figure 4-11.i shows an example entity and entity type model that correspond to this pattern.

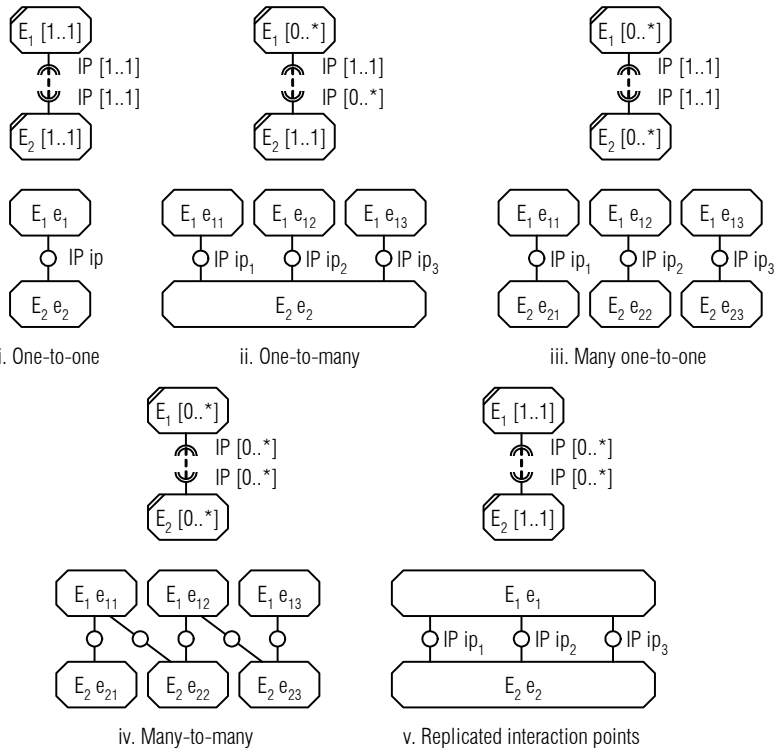


Figure 4-11 Connection Patterns between Two Entity Types

The one-to-many connection pattern represents that, in each possible structure, one entity of a particular type exists that can connect to many entities of another type via as many interaction points. This pattern will be commonly used in client-server architectures where there is one server to which many clients can connect. Each client then gets its own interaction point with the server. Variations of this pattern can specify a different multiplicity for the entity and the interaction point part of which multiple instantiations exist, indicating that there are restrictions to the number of entities or interaction points of each type that can exist. Figure 4-11.ii shows an example entity and entity type model that correspond to this pattern.

The many one-to-one connection pattern represents that, in each possible structure, many entities of a particular type exists that can be paired with entities of another type. Hence, it represents a set of possible structures in which the one-to-one connection pattern occurs multiple times. This pattern is not very useful on its own, because the paired entities do not have connections with other pairs. Hence, the resulting structural snapshots

represent collections of loose systems rather than a single coherent system. However, this pattern can occur in combination with a one-to-many or many-to-many pattern. For example, each of the clients in a one-to-many client-sever architecture can be paired with a GUI entity. Variations of this pattern can further constrain the minimum and maximum multiplicities that are associated with the instantiations. Figure 4-11.iii shows an example entity and entity type model that correspond to this pattern.

Similar to the many one-to-one connection patters, the many one-to-many connection pattern represents a set of possible structures in which the one-to-many connection pattern occurs multiple times. This connection pattern should also be used in combination with other connection patterns.

The many-to-many connection pattern represents that, in each possible structure, many entities of a particular type exists that can be connected with many entities of another type. Variations of this pattern can further constrain the minimum and maximum multiplicities that are associated with the instantiations. Figure 4-11.iv shows an example entity and entity type model that correspond to this pattern.

The replicated interaction points pattern represents that, in each possible structure, many interaction points of the same type can exist between the same two entities. This pattern can exist in combination with each of the other connection patterns. Variations of this pattern can further constrain the minimum and maximum multiplicities that are associated with the instantiations. Figure 4-11.v shows an example, where three interaction points of the same type exist in a one-to-one connection.

4.1.5 Communication Mechanism Abstraction

Different stakeholders may consider communication mechanisms at different levels of abstraction. Stakeholders that focus on high levels of abstraction consider that some entities interact through some communication mechanism. However, these stakeholders typically abstract from the particulars of the soft- and hardware that constitutes this communication mechanism and from the constraints that this mechanism imposes on the interaction (e.g.: the constraint that interaction can only take place using a request/response type of mechanism). Stakeholders at lower levels of abstraction *can* consider the soft- and hardware that constitutes a communication mechanism. They must also consider the constraints that the mechanism imposes and implement measures that ensure that the mechanism, with its constraints, implements the interaction at the higher levels of abstraction correctly.

At the different levels of abstraction, the communication mechanism and its parts can be represented by different compositions of entities and

interaction points. At the highest level of abstraction the communication mechanism can be represented by a single interaction point. At lower levels of abstraction, it can be represented by several entities, representing the parts of the communication mechanism, with interaction points between them.

At all levels of abstraction, the locations of interaction points must be described at the same abstraction level as those interaction points. It is the responsibility of the designer to ensure this. A common pitfall when choosing locations is represent a communication mechanism by an interaction point, but to assign this interaction point a location that corresponds to only one 'endpoint' of the communication mechanism. The problem with this construction is that an interaction point represents an entire communication mechanism and abstracts from the 'endpoints' that this mechanism has.

Example 4-2 Mismatching Abstraction Levels

As an example of locations of interaction points that exist at a different level of abstraction than those interaction points, consider a client-server architecture where several clients can interact with a server via TCP/IP. The designer may be tempted to use the IP addresses and ports of the clients to identify the interaction points. However, each interaction point represents an entire TCP connection between the client and the server, while the locations suggest that only the client's side of the connection is represented. We *can* choose to identify the interaction points between the server and its clients as tcp-connection$_1$, tcp-connection$_2$, …, or identify an interaction point by both the client's address and port *and* the server's address and port. Also, we can adapt the entity model to match the addressing scheme, by representing the TCP connection as an entity that has an interaction point with the server that is identified by the server's address and port and an interaction point with the client that is identified by the client's address and port.

The ability to abstract from the particulars of the communication mechanism is in line with the philosophy of the Model Driven Architecture (MDA) (Object Management Group, 2003a). The MDA proposes that there is an abstraction level in a design, also called the platform independent level, at which the particulars of the communication mechanism are not decided upon. At a lower level of abstraction, also called the platform specific level, the communication mechanism is considered in sufficient detail to have a straightforward mapping between design and implementation.

Figure 4-12 shows some common abstractions of message-oriented or remote procedure call (RPC) based middleware. It shows a design that represents two functional parts that communicate via some abstract communication mechanism. It also shows how this design can be refined in two steps into a design that can be implemented by middleware. Such middleware typically distinguishes between provided interfaces, at which entities can receive messages and RPC indications and send RPC responses, and required interfaces, at which entities can send messages and RPC requests and receive RPC confirmations. The intermediate step shows the abstrac-

tion level at which modelling languages, such as UML (Object Management Group, 2004a; Object Management Group, 2003b) and SDL (ITU-T, 2002), commonly represent their communication mechanisms. At this level an interaction point is formed by a provided interface of one part and a required interface of another part.

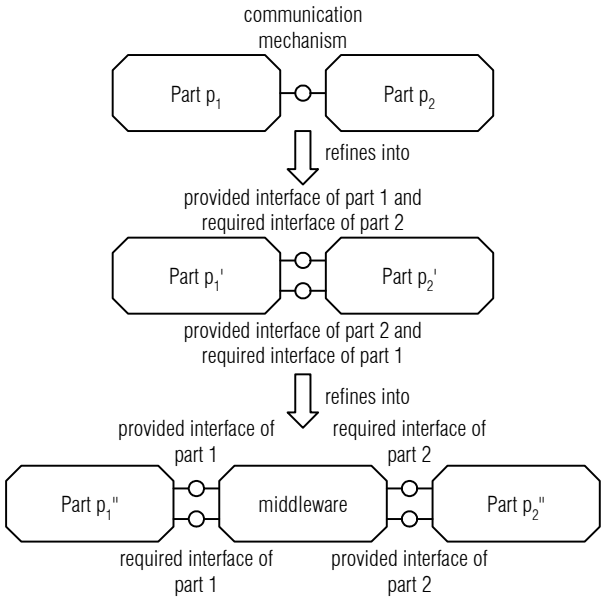


Figure 4-12 Common Abstractions of Communication Mechanisms

4.1.6 Related Work

System structure design is also addressed by languages known as Architectural Description Languages (ADLs) and UML 2.0 (Object Management Group 2004a). Medvidovic and Taylor (2000) wrote a survey on ADLs. Here, we discuss the ADLs Wright (Allen, & Garlan, 1997; Allen, & Garlan, 1994) and Rapide (Luckham, Kenney, Augustin, Vera, Bryan, & Mann, 1995; Luckham, & Vera, 1995), because, like our concepts, these ADLs support the description of the behaviour of a system. We also discuss Darwin (Magee, Dulay, Eisenbach, & Kramer, 1995), because it has strong support for (composite) structure design and for representing dynamic change in a system's structure.

Wright. Wright describes the structure of a system using the component and connector concepts. A component represents a part of the system that performs some computation. A connector represents a part of the system that realizes the communication between computational system parts. Connectors are not intended to perform any computation. Wright distinguishes between types and instances of both components and connectors. Hence,

multiple instances of a single component type or connector type can exist. Components have ports at which they make their computational functions available in the form of actions that can be invoked on these ports. The ports that a component can have are specified as port instantiations in a component type. Each port can be attached to a connector role. A connector role represents the role that a communicating party has in the communication that the connector represents. Connector roles are specified as instantiations in a connector type. Wright uses Communicating Sequential Processes (Hoare, 1985) to describe the behaviour of components and connectors in terms of the possible sequences in which actions can occur on the ports of components and on the roles of connectors. The behaviour is defined on the component and connector types. Figure 4-13 shows how a part of the wireless network from Figure 4-7 and Figure 4-8 could be represented in Wright.

Figure 4-13 A Design in Wright

System WirelessNetwork

Component EndUser
Port Antenna

Component AccessPoint
Port $Antenna_{1..64}$
Port UTPPort

Connector RadioLink
Role OneEnd
Role OtherEnd

Instances
$e_{1..3}$: EndUser
$a_{1..2}$: AccessPoint
$rl_{1..3}$: RadioLink

Attachments
e_1.Antenna as rl_1.OneEnd
a_1.$Antenna_1$ as rl_1.OtherEnd
e_2.Antenna as rl_2.OneEnd
a_1.$Antenna_2$ as rl_2.OtherEnd
e_3.Antenna as rl_3.OneEnd
a_2.$Antenna_1$ as rl_3.OtherEnd

End WirelessNetwork

Wright provides similar expressive power as our structural concepts. The main difference is that Wright connectors have their own behaviour and exist between components, while the behaviour of interaction points is completely defined by the behaviour of their participants. Hence, our structural concepts allow us to abstract from what occurs *between* entities, while we can also choose to represent that by representing a connector as a specific type of entity. Therefore, the potential for abstraction with our con-

cepts is greater and our concepts are more suitable for design at higher levels of abstraction. Moreover, Wright is intended to be used for software structure design rather than structure design in general. Other differences between our structural concepts and Wright are that Wright does not support the design of a component as a composition of sub-components and that Wright only partly supports the design of the possible structures of a system.

Figure 4-14 A Design in Rapide

```
type RadioLink is interface
end RadioLink;

type UTPPort is interface
end UTPPort;

type EndUser is interface
   service rl: RadioLink;
end EndUser;

type AccessPoint is interface
   service rl [1..64]: dual RadioLink;
   service utp: UTPPort;
end AccessPoint;

with EndUser, AccessPoint;
architecture WirelessNetwork is
   endusers: array [1..3] of EndUser;
   accesspoints: array [1..2] of AccessPoint;
connect
   enduser [1].rl to accesspoints [1].rl [1];
   enduser [2].rl to accesspoints [1].rl [2];
   enduser [3].rl to accesspoints [2].rl [1];
end WirelessNetwork;
```

Rapide. Strictly speaking, Rapide only defines the *behaviour* of a system. It does this by specifying the externally observable behaviour of the system parts, which it calls the interfaces of parts. Interfaces can be bound by means of connect statements that specify which events or function calls on one interface cause which events or function calls on another. Hence, the interfaces of parts and the bindings between these interfaces imply the structure of the system, which Rapide calls its architecture. Similar to the way in which actions can be grouped at ports in Wright, events and functions can be grouped into services in Rapide.

Rapide provides a mechanism for representing hierarchical composition of parts, by allowing an architecture, which consists of bound interfaces, to provide an interface itself. This interface can in turn be used in a composite architecture. Also, Rapide provides mechanisms for representing the dynamic creation of interfaces and services. In structural terms this corresponds to the dynamic creation of parts and interaction point parts at which

these parts can interact. Hence, Rapide can represent dynamic changes in the structure of a system.

Figure 4-14 shows how a part of the wireless network from Figure 4-7 and Figure 4-8 could be represented in Rapide.

Darwin. Darwin is similar to Wright in that it also allows for the description of the structure of a system. It also supports component types that have port instantiations and distinguishes between provided and required ports. However, it differs from Wright in that it does not consider connectors as a separate concept. Instead, provided and required ports of instances of components can be bound in a binding statement.

In addition, Darwin provides a mechanism for representing hierarchical composition of its components, by allowing a component to consist of other components. Also, it provides a mechanisms for representing the dynamic creation of components. Hence, structural dynamics can partly be represented in Darwin. Darwin's formal semantics is described in the Pi-calculus (Milner, 1999), a formalism that can partly represent structural dynamics.

Figure 4-15 shows how a part of the wireless network from Figure 4-7 and Figure 4-8 could be represented in Darwin.

Figure 4-15 A Design in Darwin

```
component EndUser{
   require RadioLink;
}

component AccessPoint{
   provide RadioLink [64];
   require NetworkCable;
}

component WirelessNetwork{
inst
   array e[3]: EndUser;
   array a[2]: AccessPoint;
bind
   e[0].RadioLink – a[0].RadioLink[0];
   e[1].RadioLink – a[0].RadioLink[1];
   e[2].RadioLink – a[1].RadioLink[0];
}
```

UML 2.0 Components. In UML 2.0 the structure of a system at a high level of abstraction can be described by means of connected components. Several components of a particular type can exist. UML distinguishes between interfaces and ports. An interface can be used to describe the behaviour that a component requires from or provides to its environment, in terms of the events and functions that this interface can handle. A port is a location at which a component can communicate with its environment. An interface can be associated with a port, indicating that a component pro-

vides or requires certain behaviour at that port, or directly with the component itself. A required and a provided interface can be connected by an assembly connector. Also, more than one required and one provided interface can be connected. In this case an interaction that occurs at the connection, occurs between all components that can support that interaction. The hierarchical composition of components can be specified, by allowing a component (type) to contain instantiations of other components. The containing component can delegate its ports and interfaces to the contained components.

Similar to the our structural concepts, UML distinguishes between structural snapshot and structural type concepts. The structural type concepts can describe component types and possible dependencies between components of those types. They can be used to construct a component diagram. The structural snapshot concepts can describe the component that constitute a system during a particular interval. They can be used to construct a composite structure diagram.

Figure 1-3 shows how a part of the wireless network from Figure 4-7 and Figure 4-8 could be represented in UML. It shows how the possible structures can be represented in a component diagram and how a structural snapshot can be represented in a composite structure diagram.

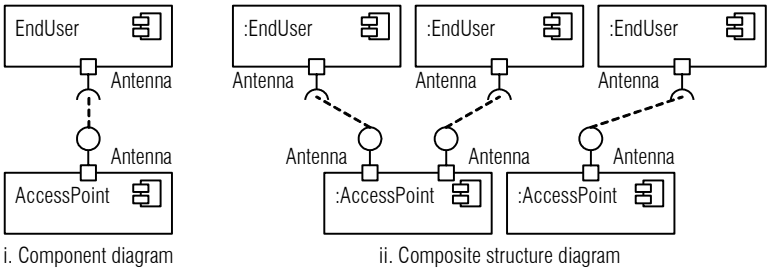


Figure 4-16 A Design in UML 2.0

4.2 System Behaviour and Behavioural Concepts

The behaviour of a system consists of the *activities* that can be performed by the system and the relations between these activities. The behaviour of a system can be structured into sub-behaviours to improve modularity of the behaviour. The sub-behaviours can either represent the behaviour of a system part or a logical unit of behaviour. Sub-behaviours can in turn be structured into sub-sub-behaviours and so on.

An activity can be performed either by a single system part or by some system parts in collaboration. It produces a tangible or intangible result that is available to all parts that engage in the activity. This result is available to

the parts at some logical or physical location. An activity takes time to be performed. Hence, it starts and finishes at particular time moments.

Two activities are related if the occurrence of one depends on the (non-) occurrence of the other. If two activities are related there is an implicit time relation between them that we will elaborate on further in this chapter. Also, if the occurrence of an activity depends on the occurrence of another activity, then its result may depend on the result of the activity on which it depends. For example, two activities are related if they must be performed in a sequence, because then the occurrence of the second depends on the occurrence of the first. Since the second occurs after the first, there is a time relation between them. Also, the result of the second activity can be a function of the result of the first activity.

The concepts from (Quartel, 1998; Quartel et al., 1997; van Sinderen, 1995; Ferreira Pires, 1994) support the representation of the behavioural concerns mentioned above. A designer can use behavioural concepts to prescribe the behaviour that a system must have.

As opposed to the structural concern for which designs can be constructed using both instance and type concepts, only type concepts can be used to construct designs for the behavioural concern. We impose this restriction because behaviours can contain an infinite number of activities. Such behaviours cannot be represented by a finite number of instances. However, assuming that realistic behaviours perform a finite number of activities and then repeat themselves, we can represent infinite behaviours by types that are instantiated multiple times. For example, a behaviour that successively performs an infinite number of activities 'a', can be represented by a behaviour type that performs 'a' and then repeats itself.

4.2.1 Behaviour

A *behaviour* represents a group of activities and the relations between these activities. For example, a behaviour can group the activities that belong to a single system part or the activities that are related to a particular phase in the behaviour of the entire system, such as the initialization phase or the data transmission phase. If a behaviour groups the activities that belong to a single system part, then it must be associated to the entity that represents that system part.

A *behaviour type* represents a template according to which behaviours can be created. The creation of a behaviour according to a behaviour type is represented by *behaviour instantiation* (of a behaviour type). We call the form of behaviour instantiation where a behaviour type instantiates itself *behaviour recursion*. We call the form of behaviour instantiation where a behaviour type instantiates another behaviour type *behaviour declaration*. If a behaviour type represents a template for the behaviours of some system parts, then it must

be associated with the entity type that represents these system parts. If a behaviour declaration represents the instantiation of the behaviour of a system part, then it must be associated with the entity instantiation that represents the instantiation of that part.

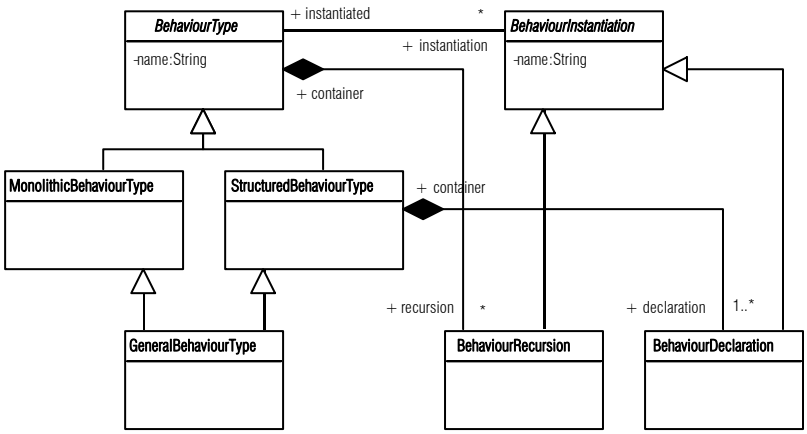


Figure 4-17 The Concepts Related to the Behaviour Concept

We distinguish between *monolithic* and *structured* behaviour types. Structured behaviour types are intended to represent behaviour structures. Therefore, they can only contain behaviour instantiations and no activities. In contrast, monolithic behaviours can only contain activities and behaviour recursions, but no behaviour declarations. When a behaviour type contains a behaviour instantiation, each behaviour of that type contains a behaviour of the instantiated type. Hence, containment of instantiations at the behaviour type level implies containment of behaviours at the behavioural instance level. Since we may want to combine a behaviour that contains activities with a behaviour that is structured, we also define the *general behaviour type*. A general behaviour type can both contain behaviour instantiations and activities.

Figure 4-17 shows the behavioural type concepts that we explained above in a meta-model.

Figure 4-18 shows how the different behavioural concepts can be graphically represented. A behaviour type is represented graphically as a rounded rectangle. The name of a behaviour type must be drawn inside the rounded rectangle. A behaviour instantiation is also represented as a rounded rectan-

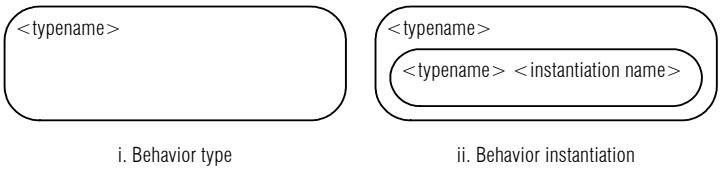


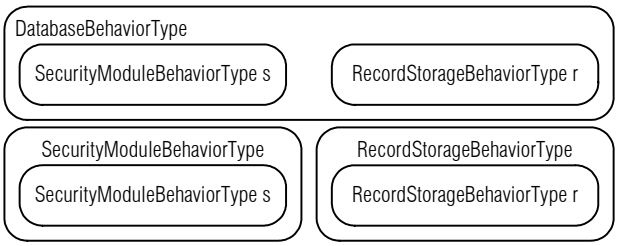
Figure 4-18 Graphical Representation of Behaviour Type and Instantiation

gle. The name of the behaviour type that it instantiates and the name of the instantiation must be drawn inside the rounded rectangle. We draw a behaviour instantiation inside the behaviour type that causes the instantiation.

Example 4-3 Example Behaviour

Figure 4-19 shows an example of the structured behaviour type that represents a database server. This behaviour type consists of behaviour declarations that represent the database's parts. The behaviour type of each of the parts is also defined. Both behaviour types contain recursive instantiations of themselves, representing that they can be repeated.

Figure 4-19 Behaviour Structure of a Database Server



4.2.2 Action

An *action* represents the successful completion of an activity that is performed by a single entity. Consequently, an action must be assigned to the behaviour of the entity that performs it. Each activity must be represented by a unique action. For example, the activity of sending an e-mail message with subject 'my account' is considered separate from the activity of sending an e-mail message with the subject 'how was your.

An *action type* represents a template according to which actions can be created. The creation of an action according to an action type is represented by *action instantiation* (of an action type). For example, we can define a type for the actions of sending e-mail.

A behaviour type contains action instantiations. The containment of an action instantiation in a behaviour type represents that an action is instantiated when a behaviour of the containing type is instantiated. As an example, consider a behaviour type that prescribes the behaviour of an e-mail system. This behaviour type contains an instantiation of the action type 'send e-mail' and a recursive instantiation of itself. Hence, the behaviour type represents the repeating behaviour of an e-mail system that can perform a 'send e-mail' action for each repetition.

Figure 4-20 Graphical Representation of Action Concepts

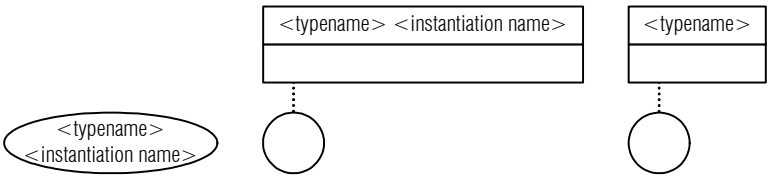


Figure 4-20 shows how action types and action instantiations are graphically represented. They are represented by an ellipse that contains the name of the type and the instantiation or by a circle to which a box containing the type and instantiation name is attached. In the case where only a single instantiation of a type exists, the name of the instantiation can be omitted. In that case, we assume that the name of the instantiation is the same as the name of the type. An instantiation must be drawn inside the behaviour type that instantiates it. Figure 4-21 shows an example of a behaviour type that contains two instantiations of the same action type.

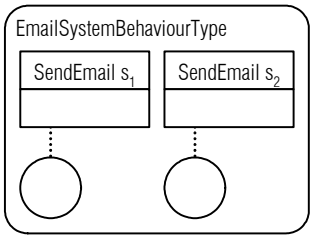


Figure 4-21 Behaviour Structure of a Database Server

4.2.3 Interaction

An *interaction* represents the successful completion of an activity that is performed by two or more entities in collaboration. An interaction either occurs, representing successful completion, or does not occur, representing failure to complete successfully, for all participating entities. If it occurs, all participating entities can refer to the result. If it does not occur, none of the participating entities can refer to any (intermediate) result that may have been established. An interaction is supported by a communication mechanism. Therefore, it can only occur between entities that have an interaction point.

Interaction contribution. We call the participation of an entity in an interaction an *interaction contribution*. Hence, an interaction consists of the interaction contributions of its participating entities. An interaction contribution must be assigned to the behaviour of the entity of which it represents the participation. For example, consider a 'submit application' interaction between a bank and a client. This interaction consists of a contribution of the bank, which is associated to the behaviour of the bank, and a contribution of the client, which is associated to the behaviour of the client.

An *interaction contribution type* represents a template according to which interaction contributions can be created. The creation of an interaction contribution according to an interaction contribution type is represented by *interaction contribution instantiation* (of an interaction contribution type). An interaction contribution instantiation is associated with a behaviour type,

representing that each behaviour that is created according to that behaviour type contains an interaction contribution of the instantiated type.

Interaction type. An *interaction type* represents a template according to which interactions can be instantiated. However, we do not explicitly represent interaction instantiations. Instead, an interaction type is implicitly instantiated if its interaction contributions are allowed to occur.

We define an interaction type in the context of a structured behaviour type, representing that it is an interaction of that structured behaviour type. Interactions of the structured behaviour type are formed by interaction contribution instantiations of behaviour declarations that the structured behaviour contains.

Since interactions can only occur between entities that have an interaction point, interaction types can only be specified between behaviour instantiations of which the entity instantiations can have an interaction point.

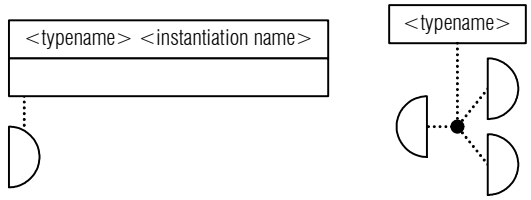


Figure 4-22 Graphical Representation of Interactions and their Contributions

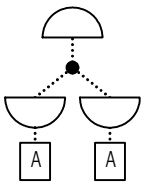


Figure 4-23 Shorthand Notation for Interaction Type Names

Figure 4-22 shows how interaction types and interaction contribution types and instantiations can be graphically represented. Interaction contribution types and instantiations are graphically represented in the same way as action types and instantiations. However, they are represented by an ellipse half instead of an ellipse. Interaction contribution instantiations must be drawn on the border of the behaviour type by which they are instantiated, with the flat part pointing outwards. An interaction is represented by a dot and lines that connect the dot to the interaction contribution instantiations that form the interaction. If an interaction contains only two interaction contribution instantiations, then the dot can be left out. The name of the type of an interaction is drawn inside a box and attached to it with a dashed line. For brevity, the name of an interaction type or any of its parts can be left out. In that case, the names of the interaction type and its parts are assumed to be the same. Figure 4-23 illustrates this case. In this figure, we assume that the interaction type and its parts have the type name 'A'.

Structured interaction contribution. Since a behaviour can consist of sub-behaviours, we may want to express that sub-behaviours contribute to an activity of their containing behaviour. To allow for this, we introduce the *structured interaction contribution* concept. A structured interaction contribu-

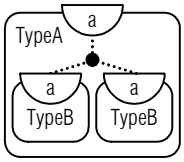


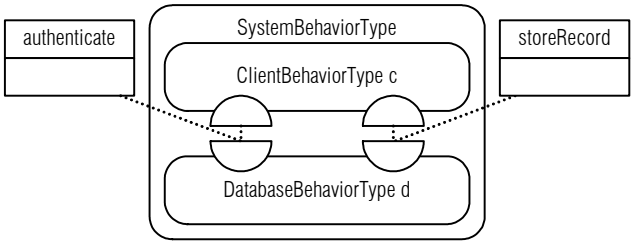
Figure 4-24 Graphical Representation of Structured Interaction Contributions

tion represents an interaction contribution of a structured behaviour, which is delegated to interaction contributions of the behaviour's parts. The successful completion of a structured interaction contribution coincides with the successful completion of its constituents.

Figure 4-24 shows how a structured interaction contribution must be graphically represented. It is graphically represented in the same way as a regular interaction contribution instantiation. It is attached to the interaction contributions that form it, similar to the way in which an interaction type is attached to the interaction contribution instantiations that form it.

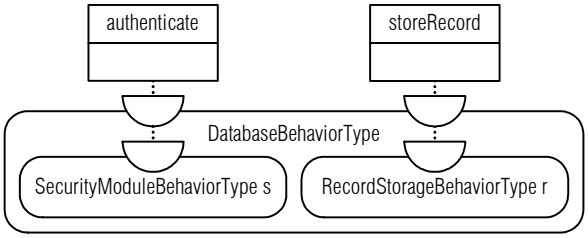
The structured interaction contribution is the behavioural counterpart of the structural delegation instantiation concept. Therefore, the association between a structured interaction contribution and its constituents must correspond to a delegation instantiation between the entity type and instantiations that perform the contribution.

Figure 4-25 Example of Interaction Types



As an example, consider that the behaviour of the database server from Figure 4-25 contains the behaviours of a security module and a record storage module. The security module participates in the *authenticate* interaction contribution of the database server. Therefore the *authenticate* interaction contribution instantiation is defined as structured and is associated with an *authenticate* interaction contribution instantiation of the security module's behaviour. Figure 4-26 illustrates this behaviour structure.

Figure 4-26 Example of Structured Interaction Contributions

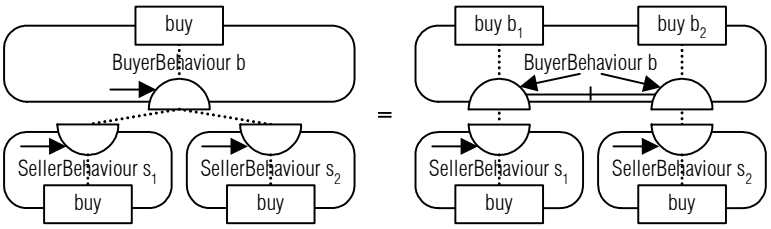


Alternative interaction shorthand. To facilitate in constructing simpler models, we developed shorthands. A *shorthand* is a single notational element for frequently occurring compositions of other notational elements. A shorthand can be used in place of such a composition.

We developed a shorthand for an interaction with alternatives. A common activity may be performed by alternative groups of participants. Each alternative group of participants performs an alternative of the interaction. We call this alternative an *alternative interaction*. We represent an alternative interaction graphically in the same way as we represent a interaction. However, unlike an interaction, an alternative may share an interaction contribution instantiation with other alternatives (of the same interaction), representing that that interaction contribution instantiation contributes to *each* of these alternatives.

Figure 4-27 shows an example of alternative interactions of the interaction 'buy'. In this example a buyer can perform an interaction 'buy' with either one of two sellers. Figure 4-27 also illustrates how to rewrite an interaction with alternatives into a design that uses only basic concepts, by splitting up a shared interaction contribution into as many interaction contribution instantiations as there are alternatives. Each of these instantiations contributes to one of the alternatives. The instantiations must disable each other, because the shared interaction contribution instantiation from which they were derived can occur only once.

Figure 4-27 Example of Alternative Interaction Contributions



Interaction meta-model. Figure 4-28 shows a meta-model of the concepts that are explained in this section. It shows that an interaction type consists of interaction participations. An interaction participation represents the participation of a behaviour in an interaction type. The meta-model shows that a structured interaction contribution is specified in the same way as an interaction. It is also specified by the participations that it contains.

4.2.4 Attributes

An *attribute* represents a result of an activity, the time at which the activity is finished and the location at which the results of the activity are available. We represent these properties of activities with the *information*, *time* and *location* attribute, respectively. When an activity has completed, its results are established and its attributes have a value.

Each attribute has a name and a type. The type of an attribute represents the structure of the result and the range of values that the result can

have. The type that is associated with an information attribute can be freely defined by the designer. The types that are associated with the time and location attributes have pre-defined properties. How information, time and location types can be defined is explained in more detail in section 4.3, along with the definition of the properties of the time and location attributes. A value of the location type is a location, such that a location attribute identifies (the location type of) an interaction point type and a value assigned to a location attribute identifies (the location of) an interaction point. In this way, we can specify the interaction point at which an interaction can occur or has occurred.

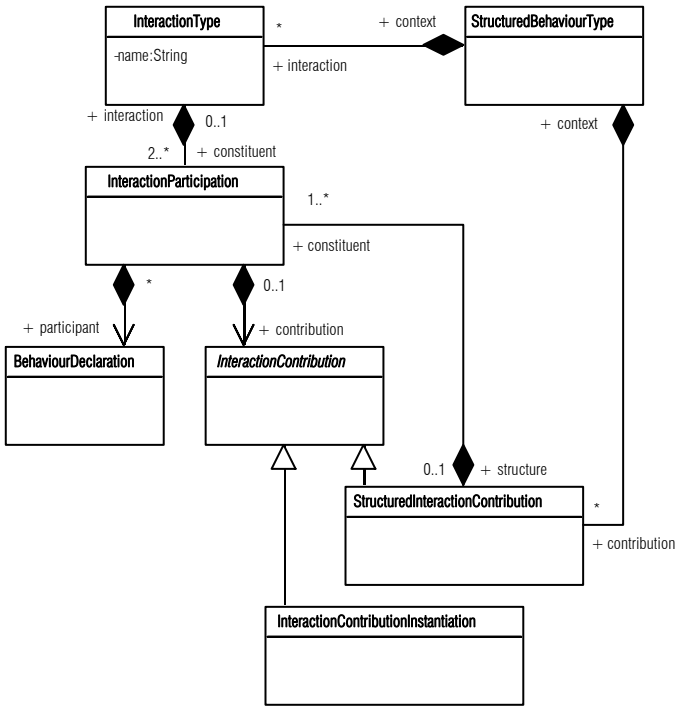


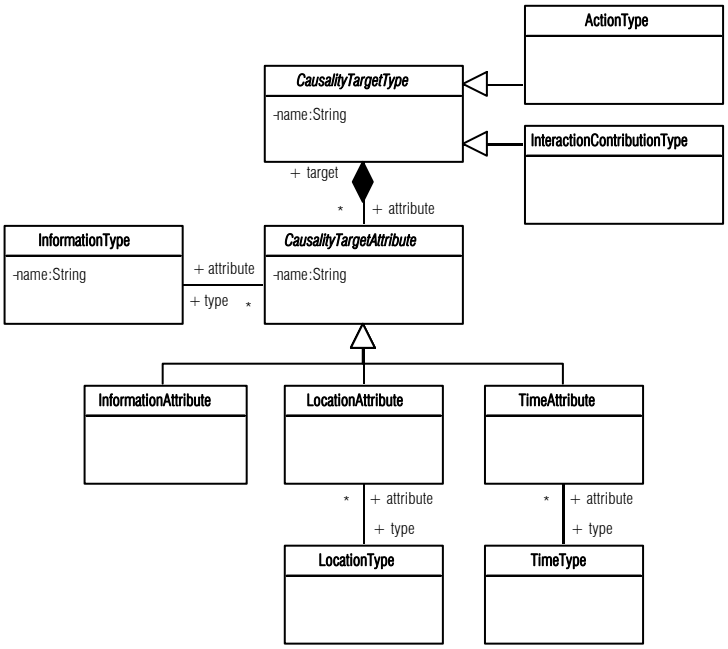
Figure 4-28 The Concepts Related to the Interaction Concept

An attribute is associated with an action or interaction contribution type, representing that each instance of that type has an attribute with the specified name and type. Since the same result is available to all participants in an interaction at the same time and at the same location, the same attributes must be associated with all interaction contributions of the same interaction. When an interaction occurs, all interaction contributions that belong to the interaction have the same values for their attributes.

Figure 4-29 shows the concepts that are related to the attribute concept. The causality target type represents either an action type or an interaction contribution type. Each causality target type can have a number of

attributes. Each attribute can either be an information attribute, a time attribute or a location attribute.

Figure 4-29 The Concepts Related to the Attribute Concept



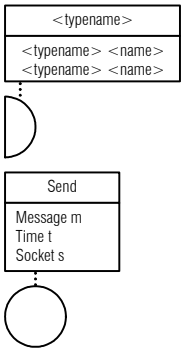


Figure 4-30 Graphical Representation Attributes

Figure 4-30 shows how attributes are graphically represented and an example. The attributes that belong to an action or interaction contribution type are represented in a box and attached to an instantiation of that type by a dashed line. For each attribute, both the name of the type of that attribute and the name of the attribute is shown.

4.2.5 Causality Relation

A *causality relation* associates an action or interaction contribution instantiation with a condition for the occurrence of that action or interaction contribution instantiation. In this section we provide a brief overview of causality relations. We refer to Quartel (1998) for a more detailed discussion, as well as a formal semantics of causality relations in terms of partially ordered sets. We refer to Katoen (1995). A causality relation consists of:

1. the action or interaction contribution instantiation, also called the causality target instantiation, for which it describes the condition;
2. a *causality condition*, which describes what causality targets must (not) have occurred for the associated causality target instantiation to occur; and

3. *constraints*. These constraints can describe what values other causality target instantiations must have established for the associated causality target instantiation to occur, in which case we call them as *causality constraints*. Constraints can also describe restrictions on the values that the associated causality target instantiation can establish, in which case we call them *attribute constraints*.

We associate a causality relation with interaction contribution instantiations rather than interactions, such that each behaviour that is involved in an interaction can specify its own condition for the occurrence of the interaction.

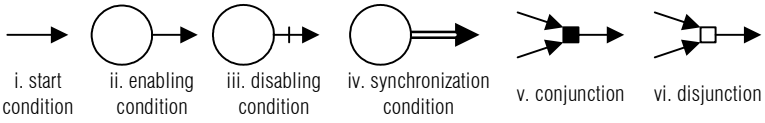


Figure 4-31 Graphical Representation of Causality Conditions

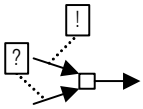


Figure 4-32 Graphical Representation of Uncertainty Attributes

Causality conditions. The causality condition for an instance of some causality target instantiation *a* is defined in terms of four basic causality conditions:

1. the *start condition* represents that the instance of *a* is always enabled to occur (or enabled for short);
2. the *enabling condition* represents that the instance of *a* is enabled, if an instance of some other causality target instantiation, specified by the condition, has occurred;
3. the *disabling condition* represents that the instance of *a* is enabled, if an instance of some other causality target instantiation, specified by the condition, has not yet occurred nor occurs at the same time; and
4. the *synchronization condition* represents that the instance of *a* is enabled, if an instance of some other causality target instantiation, specified by the condition, occurs at the same time.

These basic causality conditions can be combined into:

1. a *conjunction* that represents that all associated conditions must be satisfied to enable the occurrence of an instance of *a*;
2. a *disjunction* that represents that at least one of the associated conditions must be satisfied to enable the occurrence of an instance of *a*; or
3. a combination of conjunctions or disjunctions.

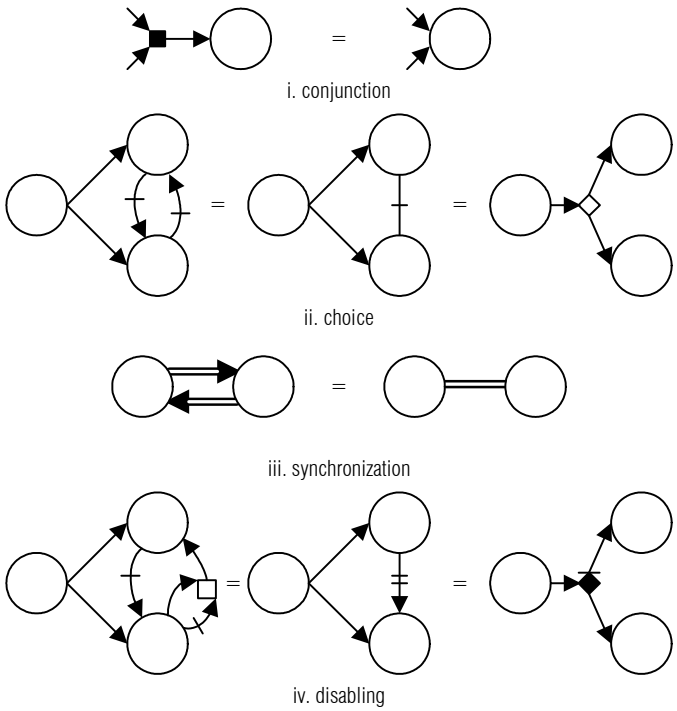
Figure 4-31 shows how causality conditions can be graphically represented. The arrowhead must point towards the instantiation of which the instances are enabled.

If a causality condition is expressed as a disjunction of conjunctions of basic causality conditions, we say that it is in the *disjunctive normal form*. We call a part of a condition in the disjunctive normal form an *alternative causality condition*, or *alternative* for short, if it is a sufficient condition for a causal-

ity target to occur. For example, for the causality condition "*b* has occurred and either *c* or *d* has occurred", "*b* and *c* have occurred" and "*b* and *d* have occurred" are alternative causality conditions. Although multiple alternatives can be true at the same time, an activity can only occur as a consequence of one alternative.

An activity either *must* or *may* occur if its causality condition is satisfied. To represent that, we use the *uncertainty attribute*. The uncertainty attribute is associated with each alternative causality condition. If the alternative condition is satisfied, the uncertainty attribute specifies whether the occurrence of the associated action is certain or not. If the uncertainty attribute has the value *must*, then the activity will eventually occur if the causality condition is satisfied. If value is *may*, then the activity may, or may not, occur if the causality condition is satisfied. The uncertainty attribute imposes no restrictions on *when* the activity will occur. Constraints on the time attribute should be used to represent such restrictions. Figure 4-32 shows how an uncertainty attribute is graphically represented. A *must* condition is represented by an exclamation mark that is associated to an alternative by a dashed line. A *may* condition is represented by a question mark.

Figure 4-33 Shorthands



Shorthand Causality Conditions. Figure 4-33 shows some of the shorthands that we developed for causality conditions.

A conjunction of conditions can be represented as conditions pointing to a filled box, or as conditions pointing directly to the causality target instantiation.

We can represent a choice between causality target instantiations by those causality target instantiations mutually disabling each other. Alternatively, we can represent two causality target instantiations mutually disabling each other by connecting those causality target instantiations by a line with a small line intersecting it. If some causality target instantiations mutually disable each other and depend on the enabling by some other causality target instantiation, we can represent this as an open diamond. This diamond has an arrow from the enabling causality target instantiation to it, and from it to all causality target instantiations between which there is a choice.

Some causality conditions imply other causality conditions. For example, if one causality target instantiation 'a' must occur at the same time as 'b' (synchronization condition), then 'b' must occur at the same time as 'a'. Figure 4-33.iii shows this. Since these relations will frequently be used in combination, we developed the shorthand shows in Figure 4-33.iii.

Similarly, if one causality target instantiation 'a' disables another causality target instantiation 'b', then 'b' cannot occur after 'a' has occurred, nor at the same time. This implies a condition for 'a', namely that 'a' cannot occur at the same time as 'b'. We can represent this by adding the condition that 'a' can occur either if 'b' has occurred or if 'b' has not yet occurred nor is occurring. Figure 4-33.iv shows this. Because each disabling of a causality target instantiation implies a relation in the opposite direction, this composition occurs frequently. Therefore, we developed the shorthands that are also shown in Figure 4-33.iv.

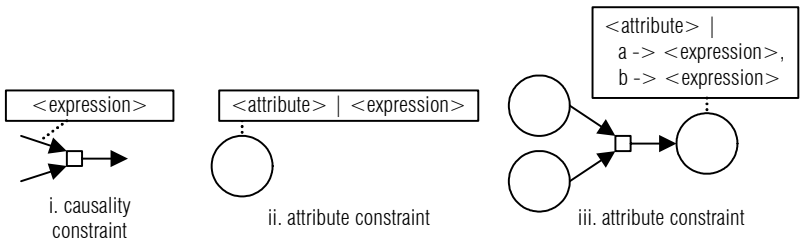


Figure 4-34 Graphical Representation of Constraints

Constraints. An *alternative causality constraint* specifies what attribute values must have been established by activities that appear in an enabling or synchronization condition, for the associated alternative causality condition to be satisfied. Hence, a causality target is only enabled if one of its alternative causality conditions is satisfied *and* the alternative causality constraint that is associated with that alternative is satisfied. An alternative causality constraint cannot depend on attribute values of activities that appear in a disabling condition, because these activities must not occur for the associated

activity to occur. Hence, their attributes do not have values. Figure 4-34.i shows how an alternative causality constraint is represented by an expression that is associated with an alternative by a dashed line. The language that we use to express the alternative causality constraints on the attributes is explained further in section 4.3.

An *alternative attribute constraint* specifies constraints on the values that can be established for an attribute. These constraints can also represent the relations between the results of two activities. Similar to alternative causality constraints, we associate alternative attribute constraints with alternative causality conditions. This represents that if a causality target occurs as a consequence of the alternative causality conditions, then the attributes of that causality target are constrained by the alternative attribute constraint. Figure 4-34.ii shows how an alternative attribute constraint is represented as an expression that follows an attribute. In this representation, the constraint must hold for all alternatives. If the constraint must only hold for some alternatives, it can be associated with that alternative in the same way as a causality constraint (Figure 4-34.i), or by prefixing the constraint with a textual representation of the alternative (Figure 4-34.iii). In this textual representation:

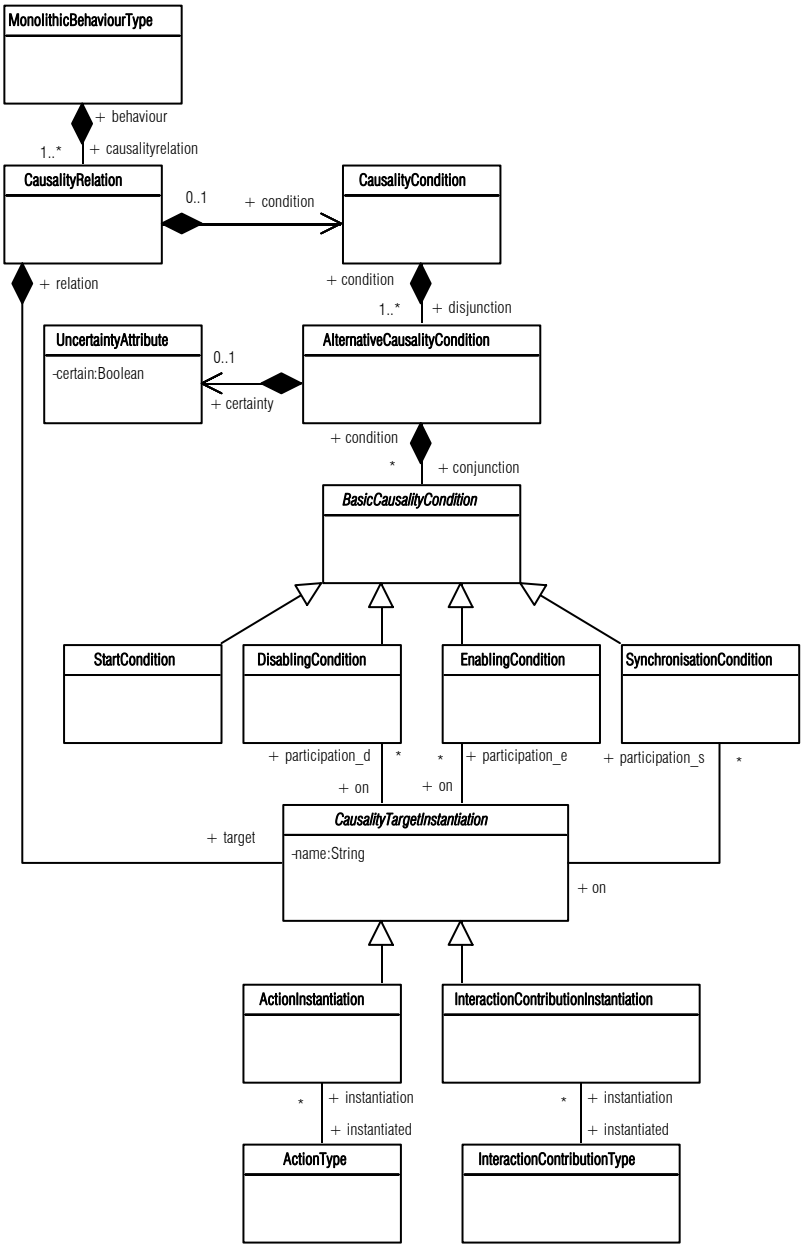
- the start condition is represented as $\surd$;
- the enabling condition is represented by the name of the associated causality target instantiation (e.g.: a);
- the disabling condition is represented by $\neg$ followed by the name of the associated causality target instantiation (e.g.: $\neg a$);
- the synchronization condition is represented by $=$ followed by the name of the associated causality target instantiation (e.g.: $=a$);
- the conjunction is represented by the logical conjunction, $\wedge$, of conditions (e.g.: $\neg a \wedge b$); and
- the disjunction is represented by the logical disjunction, $\vee$, of conditions (e.g.: $\neg a \vee b$).

The language that we use to express the attribute constraints is the same as the language that we use to express the causality constraints. It is explained further in section 4.3.

A causality relation of an action or interaction contribution can imply time relations with other actions and interactions, as well as causality relations for other actions and interactions and values for the uncertainty attributes of other causality relations. For example, if action a is synchronous with action b then a must occur at the same time as b. Hence, the time attribute of a must have a value that is equal to the time attribute of b. Also, b must be synchronous with a. Therefore, the causality relation for b must include the condition that it occurs synchronously with a. Finally, the uncertainty attribute for the causality conditions of the two actions must be the same, because if one of them must occur if its causality condition is

satisfied, then so must the other. More details about implied relations between actions and interactions can be found in (Quartel, 1998).

Figure 4-35 Concepts Related to the Causality Relation Concept



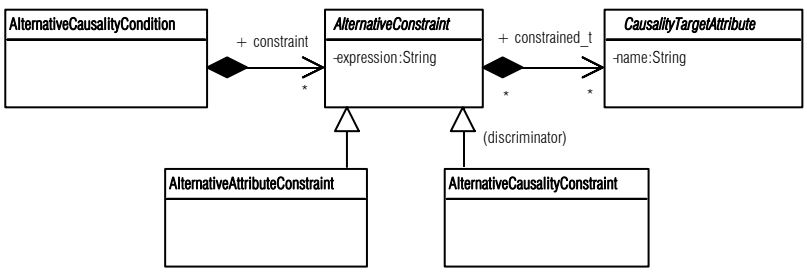
Causality relation meta-model. Figure 4-35 shows a meta-model of the concepts that are related to the causality relation concept. The meta-model

only allows for modelling in the disjunctive normal form. It shows that each causality condition consists of a disjunction of alternative causality conditions. Each alternative causality condition is a conjunction of basic causality conditions. Each alternative causality condition is associated with an uncertainty attribute and can be associated with attribute and causality constraints on the attributes of the target of the causality relation. Each enabling, disabling or synchronization condition refers to the action or interaction contribution that causes the enabling, disabling or synchronization with the target of the causality relation. Causality relations are specified in the context of a monolithic behaviour. Hence, only monolithic behaviours can contain causality relations and therefore action and interaction contribution instantiations. Figure 4-36 shows a meta-model of the concepts that are related to the causality constraint concept.

Example 4-4 An Example of Causality Relations

Figure 4-37 shows an example of some causality relations that are defined in the context of a behaviour type. The figure represents a behaviour in which a local or a remote request can occur with some parameters. A remote request has to be sent and received, while a local request can be processed immediately. However, after a remote request is sent, a communication error can occur. Both a local and a remote request can only be processed, if the parameter that is sent along with the request is valid. The action instantiations from Figure 4-37 represent these activities. *Remote request send* and *local request* have a start condition. Therefore they are enabled from the moment the behaviour is instantiated. The *communication error* action is enabled by *remote request send* and disabled by *remote request receive*. Hence, it can occur after a request was sent and while the request was not received on the other side. *Remote request receive* has a similar condition. Therefore, *remote request receive* and *communication error* exclude each other. The value of the parameter that *remote request receive* establishes must be equal to the value that *remote request send* establishes. *calculate response* can either be caused by *remote request receive* or by *local request*. However, both alternative conditions have the constraint that the parameter must be valid before *calculate response* can occur. The result that *calculate response* establishes is a function of the parameter of *remote request receive* or the parameter of *local request*, depending on which of the two caused the occurrence of *calculate response*.

Figure 4-36 Concepts Related to the Causality Constraint Concept



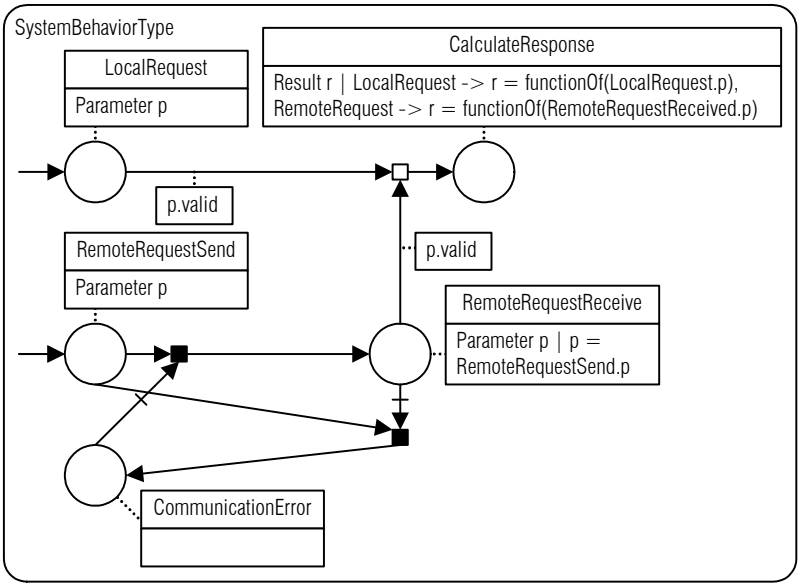


Figure 4-37 A Behaviour with Causality Relations

4.2.6 Behaviour Structuring

We define two techniques for structuring a behaviour, using the structured behaviour type concept: *causality oriented structuring* and *constraint oriented structuring*. These structuring techniques differ with respect to the way in which they relate the constituent behaviours of a structured behaviour. They can be used in combination if necessary.

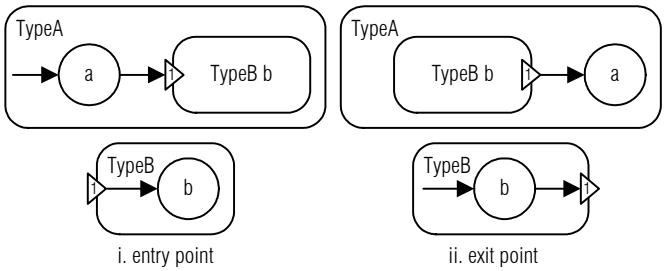


Figure 4-38 Graphical Representation of Entry and Exit Points

Causality oriented structuring. The causality oriented structuring technique is a purely syntactic structuring technique. It allows the causality target instantiations from one behaviour instantiation to appear in the causality condition of causality target instantiations from another behaviour. Therewith 'splitting up' a causality relation between several behaviour instantiations. This behaviour structuring technique can be used for both forms of behaviour instantiation: behaviour declaration and behaviour re-

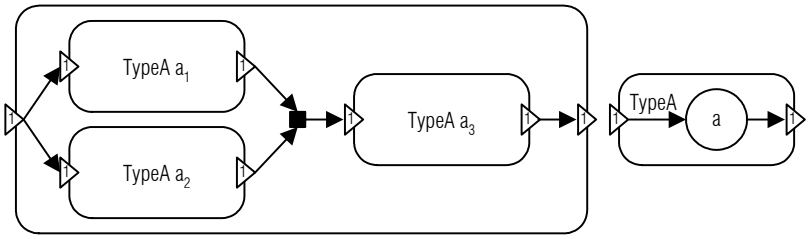
cursion. To split up a causality condition between several behaviours, we use *entry points* and *exit points*.

An entry point represents a causality condition from *outside* the behaviour type of which it is a part. The behaviour type of which it is a part can use the entry point as a part of its own causality conditions. The behaviour type that instantiates the behaviour type of the entry point must specify the condition that the entry point represents.

An exit point represents a causality condition from *inside* the behaviour type of which it is a part. The behaviour type must define the condition that the exit point represents. The behaviour that instantiates the behaviour type of the exit point, can use the exit point in its own causality conditions.

Figure 4-38 shows how that entry and exit points are graphically represented as a triangles pointing into or out of a behaviour type, respectively. An entry point represents a condition from the instantiat*ing* behaviour type, being used in the instantiat*ed* behaviour type. Hence, Figure 4-38.i represents that the condition for *b* is the occurrence of *a*. An exit point represents a condition from the instantiat*ed* behaviour type, being used in the instantiat*ing* behaviour type. Hence, Figure 4-38.ii represents that the condition for *a* is the occurrence of *b*.

Figure 4-39 Example of Entry and Exit Points



Example 4-5 Entry and Exit Points

Figure 4-39 shows a structured behaviour type that replicates behaviour type *Type A* three times. The entry points of the first and the second replica are associated with the condition represented by entry point of the instantiating behaviour. The entry point of the third replica is associated with the conjunction of the conditions that are represented by the exit points of the first two replicas. Since these exit points represent the enabling by action *a*, the entry point of the third replica represents the conjunction of the enabling of action *a* from the first replica and action *a* from the second replica. Hence, action *a* from the third replica can occur after both action *a* from the first and from the second replica has occurred. The exit point of the structured behaviour represents the enabling by action *a* from the third replica.

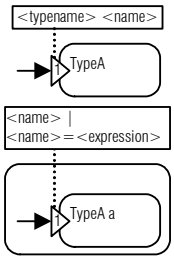


Figure 4-40 Graphical Representation of Parameters

To make behaviours completely modular, we do not allow an attribute from one behaviour to depend on an attribute from another behaviour, since referring to attributes from another behaviour would mean that a behaviour could 'look inside another behaviour' and hence violate the principles of modularity. To allow behaviours to make use of the values that are established in other behaviours, these values can be passed as *parameters* of entry

and exit points. The parameter of an entry point represents a value that is assigned by the behaviour that instantiates the behaviour that owns the entry point. The parameter of an exit point represents a value that is assigned by the behaviour that owns the exist point. A *parameter constraint* represents an expression that defines the value that can be assigned to the parameter. The parameter constraint can reference attributes or other parameters.

Figure 4-41 Example of Parameters

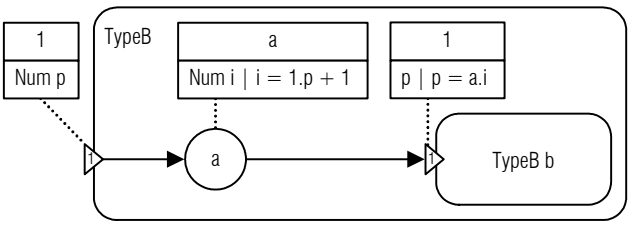


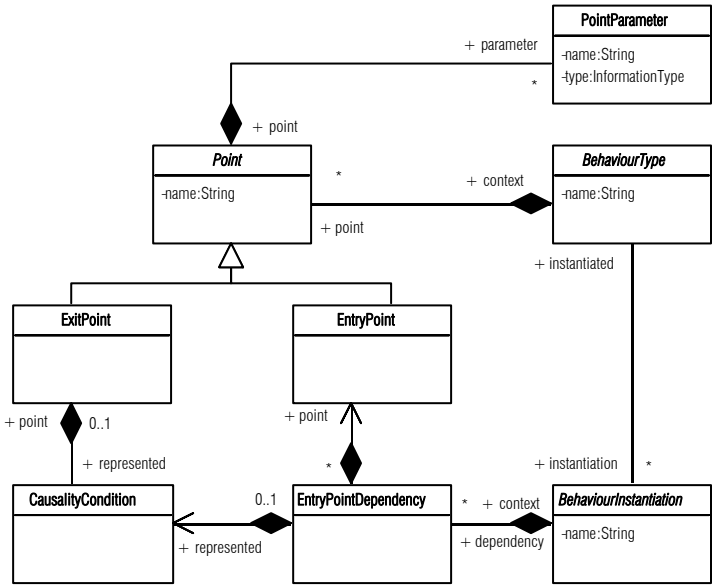
Figure 4-40 shows that parameters are graphically represented by their name and type attached to the point of which they are a parameter. Parameter constraints are graphically represented as expressions that are associated to the point by the behaviour type that imposes the constraint. Hence, in the case of an entry point, the constraint is associated with the behaviour type that instantiates the behaviour type of the entry point, because this behaviour type imposes the constraint. In the case of an exit point, the constraint is associated with the behaviour type that owns the exit point, because this behaviour type imposes the constraint.

Example 4-6 Parameters

Figure 4-41 shows a recursive behaviour type that has an entry point with a parameter named *p* with type *Num*. The result of action *a* equals the value of the parameter *p* plus 1. The value of the parameter for the recursive instantiation of the behaviour is constrained, using a parameter constraint, such that it equals the result of action *a*. The instance that is created by this instantiation has another action *a*, which will increase the parameter again and pass the new value on to the next instance and so on.

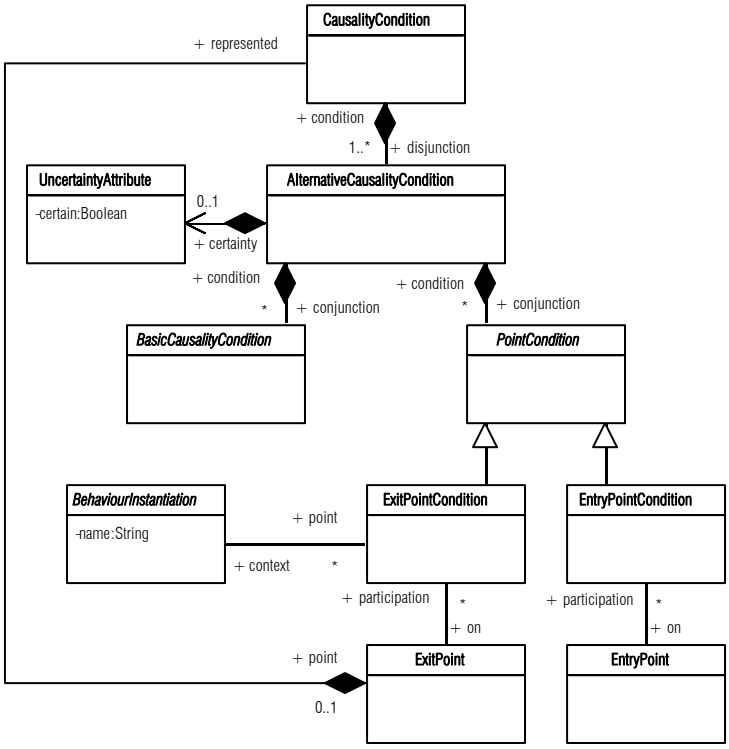
Causality-oriented structuring meta-model. Figure 4-42 shows a meta-model of the concepts that are related to causality-oriented behaviour structuring. It shows that each behaviour type can have entry and exit points. We do not distinguish between entry and exit point types and entry and exit point instantiations, because each entry and exit point type is instantiated exactly once for each behaviour instantiation. Hence, a point represents a combination of a point type and a point instantiation. An exit point is associated with the causality condition that it represents. A structured behaviour type associates each entry point of the behaviour instantiations that it defines with a causality condition. Therefore, an entry point dependency relates a causality condition to an entry point *of* a behaviour instantiation.

Figure 4-42 Concepts Related to Behaviour Structuring



Since causality conditions can refer to conditions that are represented by entry and exit points, we extend the causality relation concepts with the

Figure 4-43 Concepts Related to Point Conditions



concepts from Figure 4-43. The figure shows that an alternative causality condition is a conjunction of basic causality conditions and point conditions. A point condition either refers to the entry point of the behaviour type that instantiates the causality target, or it refers to an exit point of one of the behaviour instantiations in that behaviour type. Figure 4-44 shows a meta-model of the concepts that are related to causality constraints on point parameters.

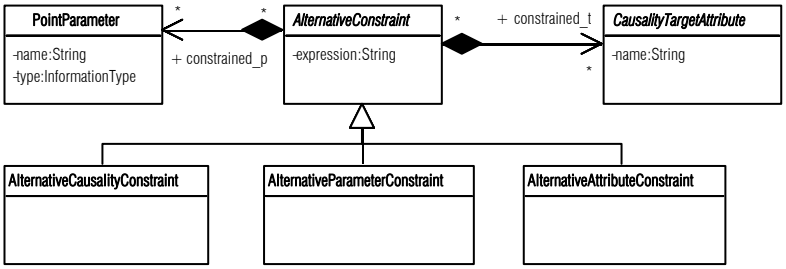


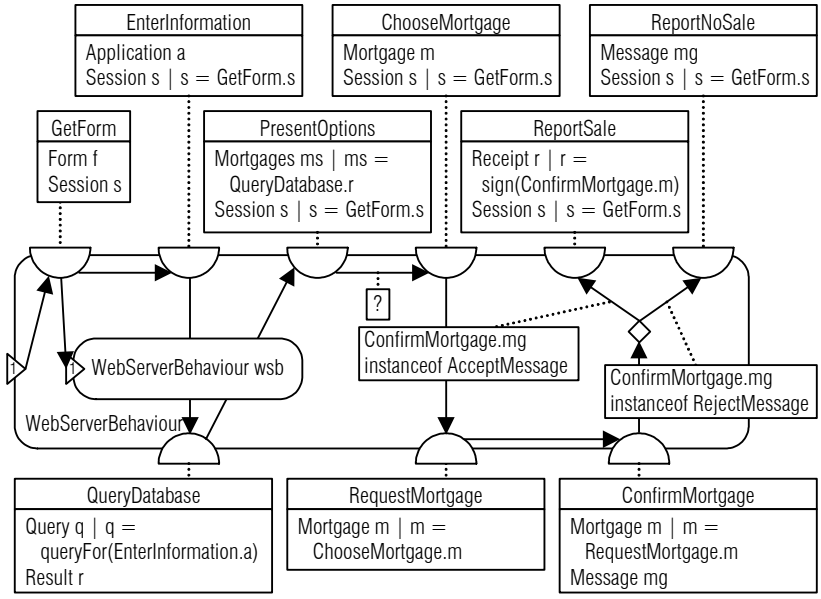
Figure 4-44 Concepts Related to Point Constraints

Constraint oriented structuring. The constraint oriented structuring technique allows the conditions for an activity to occur to be distributed over several interacting behaviours. For this purpose, the activity has to be represented as an interaction for which each participating behaviour can specify its own constraints. The constraint oriented structuring technique can, for example, be used to logically distribute the conditions for the occurrence of an activity. It can also be used to distribute the responsibility of enforcing those conditions over different parts.

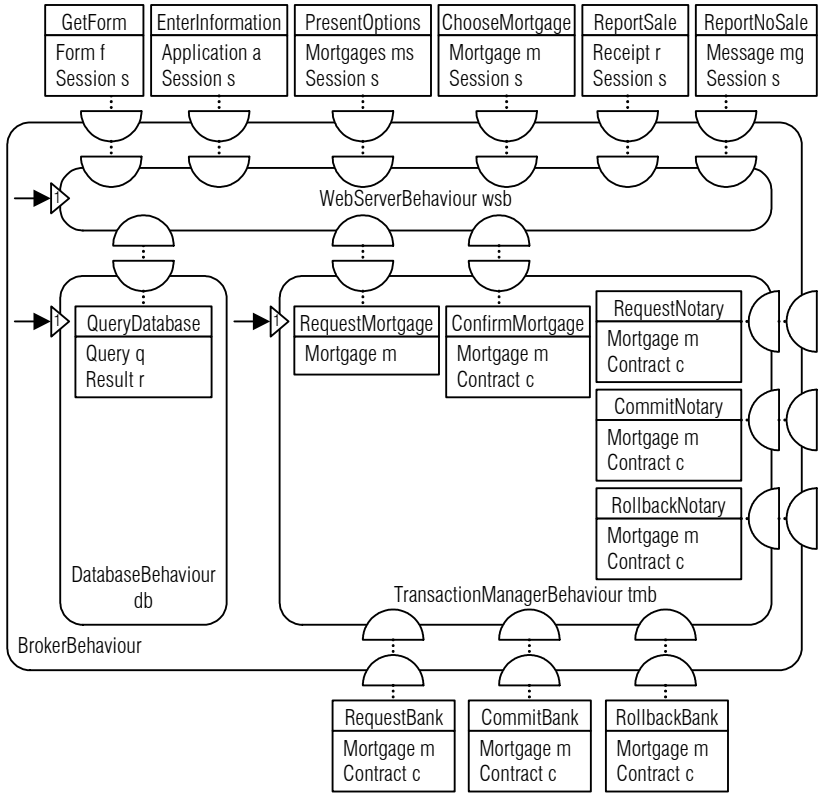
4.2.7 Example

Figure 4-45.i defines the behaviour of a web-server that can be used by its clients to search for an appropriate mortgage. The web-server allows a client to obtain a form via a session that the web-server has with that client. After the client has obtained the form, the web-server instantiates a new web-server behaviour, such that another client can also obtain a form. After a client has received a form, the behaviour only accepts interactions via the same session though which the client obtained the form. In this way each behaviour instance can be related to a single session with a client and not accidentally process parts of a session with another client. The client can fill out the form, resulting in an application. Based on this application, the web-server constructs a query on the database. It performs this query on the database and receives the result. The result is a set of mortgages, which is presented to the client. The client may select one of the mortgages, after which the mortgage is passed to the transaction manager that will take care of closing the deal with the bank and the notary. If it succeeds it returns an accept message and if it fails it returns a reject message.

Figure 4-45 A Behaviour Design



i. Web server behaviour type



ii. Broker behaviour type

The web-server behaviour is used in the behaviour of the mortgage broker from Figure 4-45.ii. The mortgage broker behaviour combines the behaviour of a web-server with the behaviour of a database and a transaction manager, by instantiating these behaviours. All instantiations have a start condition associated with their entry points, representing that they are enabled from the moment the broker behaviour is instantiated. The broker behaviour also defines the interaction *query database* between the web-server and the database and the interactions *request mortgage* and *confirm mortgage* between the web-server and the transaction manager. Finally, it defines how its sub-behaviours participate in interactions that it has with its environment.

4.3 Information and Information Concepts

The (values of) attributes of an action or interaction represent the result that that action or interaction establishes, as well as the time and location at which the result is available. Those values carry information about the result and the time and location at which it is available.

We use the information concepts to describe the information about the result in more detail. In particular, we consider that information has structure and can be of a certain information type. An information type identifies information that is similar with respect to this type.

We consider that information can be structured by logically grouping it into different information elements. In turn, the information elements in each of these logical groups can be grouped into logical sub-groups and so on and so forth. Hence, we also say that information can be *hierarchically* structured.

It is our goal to focus on concepts for prescribing information structure and constraints, not to define an information modelling language for graphically representing them. Therefore, rather than defining a modelling language ourselves, we allow a designer to define his own representation relation to describe which modelling elements he wants to use to describe which concepts.

The graphical representation of causality and attribute constraints depends on the graphical representation of attributes to which these constraints apply. Therefore, these same binding that binds information concepts to modelling elements must bind constraint concepts to modelling elements.

At the end of this section we show a binding between the information concepts and UML.

4.3.1 Information Concepts

Figure 4-46 shows the concepts that we can use to represent information structures and types.

An *information value* represents (a part of) a possible result of an activity. After an activity is performed its result is represented by information values assigned to its attributes. The *information type* concept represents a set of information values that are similar with respect to that type and the structure of these information values. We say that information values in the set that is represented by an information type are *of that type* or *satisfy that type*. We say that an information type *a* is a subtype of another type *b*, if all information values that satisfy *a* also satisfy *b*. If *a* is a subtype of *b*, *b* is a supertype of *a*. The information type that is associated with an attribute constrains the information values that can be assigned to that attribute, because the value must satisfy the type. An information type is uniquely identified by a name in the context of a design.

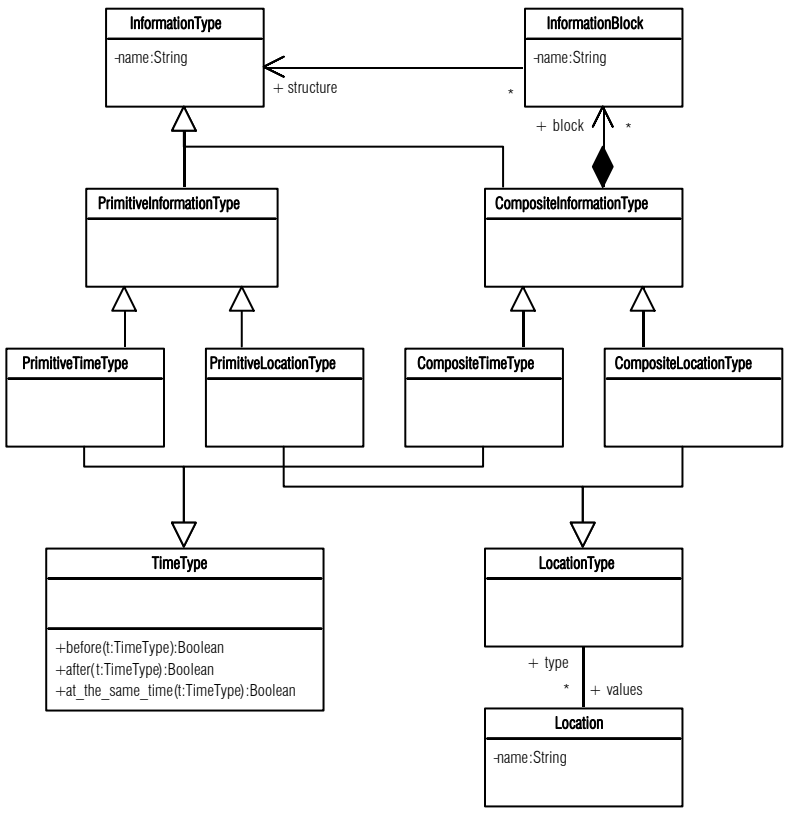


Figure 4-46 Information Structure Concepts

To represent structure of information values, an information type can be structured into *information blocks*, in which case we refer to it as a *composite information type*. Each block is associated with an information type itself, constraining the information values that that part can have and, optionally, its further structuring into blocks. An information block is uniquely identified by a name in the context of a composite information type. If an information type is unstructured, we refer to it as a *primitive information type*.

Example 4-7 A Composite Information Type

As an example of a composite information type, consider the *personal information* type that can be structured into the blocks *name* and *address*. The *name* block has a finite list of characters as its information type. The *address* block has a type that can be further structured into a block *street* with a finite list of characters as its information type and a *number* block with a positive natural number as its information type. As an example of an information value, consider an information value that is the result of an activity that establishes a result of type *personal information*. This value must have a block *name* that, for example, has the value *John Jameson*. It must have a block *address* that, for example, has the value *Bowstreet* for *street* and 7 for *number*.

Two special information types are the *time type* and the *location type*. These types can be used to represent the information structure and values of the time and the location at which the result of an activity is available. An information value of the location type is called a *location* and an information value of the time type is called a *time moment*. The location and location type concepts are the same concepts as those from the structural system design. Therefore, each location type must correspond to a location type in the structural type design. The values of that type are locations of that location type in a structural snapshot design.

4.3.2 Binding of Modelling Language to Information Concepts

We allow a designer to use a modelling language of his choice to represent information concepts. However, before he can do so, a binding between the modelling language and the information concepts has to be defined. A *binding* defines which modelling elements can be used to represent which information concept. If a modelling element is used to represent an information concept in the context of a binding, we also say that the information concept and the modelling element are bound.

Since the elements of a modelling language have a semantics of their own, the binding of modelling elements and information concepts is not arbitrary. It must be defined such that the semantics of the modelling elements is consistent with the semantics of the information concepts to which they are bound. In particular it must observe the rules below.

Binding to Information Type Concepts. A binding must describe which modelling elements are used to represent the information type con-

cept and the information value concept. The modelling element that is bound to the information value concept and the modelling element that is bound to the information type concept must have a type-instance relationship, because the information type and information value concept have such a relationship.

A binding must prescribe which modelling elements are used to represent the composite and primitive information type concepts. The composite information type concept puts additional constraints on the modelling elements that can be bound to it. Modelling elements that the designer binds to the composite information type concept must be able to contain modelling elements that can be bound to information blocks. These modelling elements must be able to have an identifier (name). Also, they must have an association with the modelling elements that are bound to the information type concept. This relationship must represent that the block can be assigned a value that satisfies the type represented by the associated modelling elements that are bound to the information type.

The designer defines the required time and location types in the modelling language of his choice as a part of a design. Each time type must represent a continuous or discrete timeline for which at least a notion of 'before', 'after' and 'at the same time' exists. These notions are required to match the time relations that are implied by the basic causality conditions. *a enables b* implies that *b* occurs after *a*, *a disables b* implies that, if *b* occurs, it occurs before *a* and *a synchronous with b* implies that *a* and *b* occur at the same time. A model of a location type must represent a set of *values* that correspond to the locations of that type in the structural design.

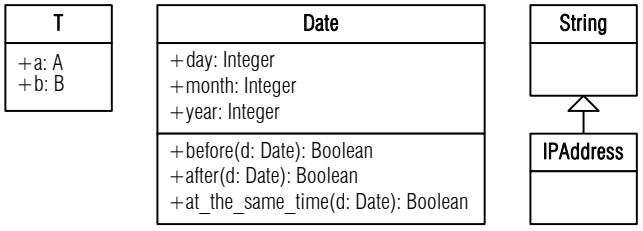


Figure 4-47 An Example of Information Types Represented with a Binding to UML

A binding must bind modelling elements to the attribute and parameter concepts. The modelling element that is bound to these concepts must be able to have a name, a type and a value. The value must take the form of an instance of the modelling element that is bound to the information value concept. It must satisfy the type that is associated to the corresponding attribute or parameter. Note that a value of an attribute or parameter is associated with an *instance* of an action, interaction or behaviour type, and not with an *instantiation*. Figure 4-48 illustrates this case. In this figure, each action instance of *a* can have its own value for *n*.

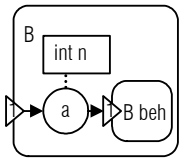


Figure 4-48 Example of Different Values for the Same Attribute

Example 4-8 A Binding

As an example of a binding, consider UML as a language for representing information. UML classes and objects have a template-instance relationship. Therefore we can bind the UML class modelling element to the information type concept and the UML object modelling element to the information value concept. We can also bind UML classes to composite information types, such that their attributes represent the blocks of composite information types.

For example, consider a composite information type *T* that contains a block by the name *a* with an information type by the name *A* and a block by the name *b* with an information type by the name *B*. This information type can be modelled in UML by a class *T* that has an attribute with name *a* that is an object of class *A* and an attribute with name *b* that is an object of class *B*. We can model a composite time type in UML as a class *Date* with attributes *day*, *month* and *year* of type *Integer* and methods *before*, *after* and *at_the_same_time* that take another *Date* object as an argument and return a *Boolean*. As another example, we can model a primitive location type that represents IP addresses as UML class *IPAddress* that is a subclass of the UML primitive type *String*. Figure 4-47 shows these types.

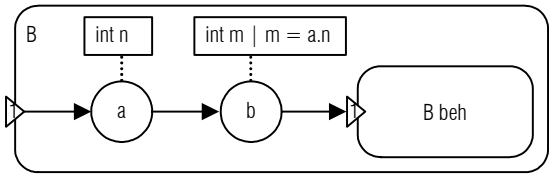
Binding to Constraint Concepts. The constraint concept contains an *expression* that can be used to model a constraint in the modelling language of choice. Hence, the binding must describe which modelling elements can be used to model a constraint. A constraint is a truth expression that must evaluate to true. In this way attribute and parameter constraints restrict the values that can be assigned to an attribute or parameter, because only values that make the corresponding constraint true are allowed. Causality constraints restrict the alternative causality conditions by which a causality target can be enabled, because only alternative causality conditions for which the constraint evaluates to true can enable a causality target.

Parameter constraint bindings can be treated in the same way as attribute constraint bindings. However, parameter constraints can only be equivalence constraints, because parameter constraints can only be used to pass information from one behaviour to another. If information can only be passed, then we have to know *exactly* which information is passed. Hence, a parameter constraint must evaluate to a single value.

A constraint can refer to attributes of action or interaction instantiations and to parameters of entry or exit points. Therefore, we allow expressions that represent these constraints to refer to the modelling elements that represent attributes and parameters. Since different instantiations can have attributes with the same name, we refer to attributes as: <the name of the instantiation>.<the name of the attribute>. This prevents naming conflicts between instantiations with attributes that have the same name. We use the same naming scheme to refer to parameters of entry points. We refer to parameters of exit points as: <the name of the behaviour instantiation of the exit point>.<the name of the exit point>.<the name of the parameter>. This prevents naming conflicts between behaviours that have exit points with the same name. If a constraint refers to an attribute or pa-

rameter of the instantiation or point to which it is attached, the name of that instantiation or point can be left out for brevity. Finally, an expression can refer to blocks in an information type using <the name of the attribute or parameter>.<the name of the block>. If the block is structured itself, the 'dot' notation can be used further to identify those blocks and so on. For example, consider the two actions from Figure 4-49. The constraint states that the value of attribute *m* of the action instance that is created by instantiation *b* must be equal to the value of attribute *n* of the action instance that is created by instantiation *a*.

Figure 4-49 An Example of Attribute Constraints



The same model elements that are bound to attribute and parameter constraints can be bound to causality constraints.

Table 4-2 UML Binding to Information Concepts

Information Concept	UML 2.0 Modelling Element
InformationValue	InstanceSpecification
InformationType	Classifier
PrimitiveInformationType	DataType
CompositeInformationType	Class
InformationBlock	Property of Class
Attribute/Parameter	Property of Class that represents the result of an Action or Interaction
Causality, Attribute or Parameter Constraint	OCL Constraint

4.3.3 Binding of UML Modelling Elements to Information Concepts

Table 4-2 shows how we bind UML modelling elements to information concepts.

We bind an information value to a UML instance specification and an information type to a UML classifier. Since classifiers and instance specifications have a type-instance relation, this satisfies the binding rules.

We bind a primitive information type to a UML data type. Hence, values of a primitive information type are bound to UML data values. Four pre-defined UML data types exist:

– The data type 'Boolean', which represents the logical truth values: 'true' and 'false'.

– The data type 'Integer', which represents integer values (…, -2, -1, 0, 1, 2, …).
– The data type 'String', which represents sequences of characters.
– The data type 'UnlimitedNaturals', which represents natural numbers (0, 1, 2, …) and infinity. We represent infinity as an asterisk (*).

In addition to that, UML data types include enumeration types. An enumeration type is a data type of which the values can be defined freely, by enumerating them.

We bind a composite information type to a UML class. Hence, the values of a composite information type are bound to UML objects. We bind the blocks of a composite information type to properties of the correspond-

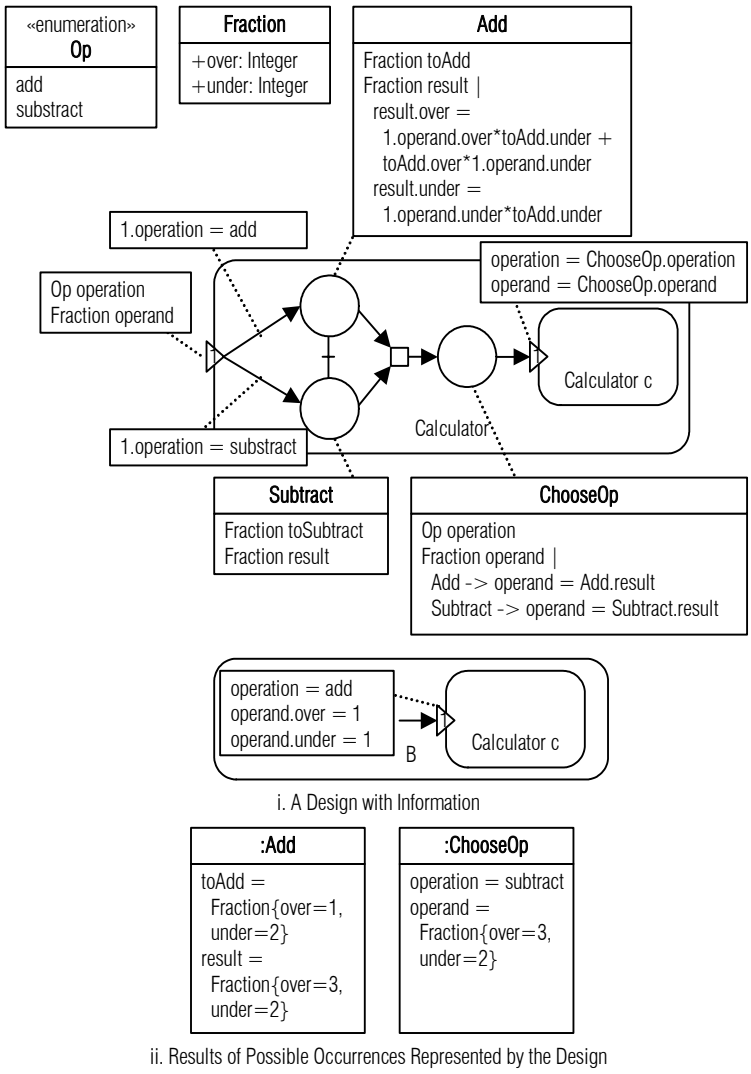


Figure 4-50 An Example of a Behaviour with the Information Concern Represented in UML

ing class. These properties can have a name and a type. The type identifies a UML class that is bound to an information type. Hence, the class structure matches that of the information types and the binding rules are satisfied.

We bind the attributes of a causality target instantiation to properties of a UML class. The name of an attribute is bound to the name of the UML property and the type of the attribute or property is bound to the type of the UML property. We do not explicitly represent the class that contains the properties in UML. Instead, we assume that such a class exists for each causality target instantiation. We assume that this class has the same name as the causality target instantiation that it represents. When an instance of a causality target instantiation has occurred, then the result that is established is represented by an instance of the class. The parameters of a point are represented by properties of a UML class in the same way.

We bind a causality, attribute or parameter constraint to an OCL constraint. In case of an attribute or parameter constraint, the OCL constraint must be specified in the context of the UML class that contains (a UML representation of) the corresponding attribute or parameter. In case of a causality constraint, the OCL constraint must be specified in the context of a newly defined UML Class. Each instance of this class represents the enabling of the corresponding alternative causality constraint. Like the classes that represent attributes or parameters, this class is not modelled explicitly, but we assume that it exists.

Example 4-9 A Behaviour with the Information Concern Represented in UML

Figure 4-50.i shows a behaviour that contains information types and constraints specified in UML. The behaviour represents a calculator that can either perform an addition or a subtraction operation on fractions. After an addition or subtraction, the next operation can be chosen and the behaviour is repeated. Two information types are defined: 'Op' and 'Fraction'. 'Op' is a primitive type that represents an addition or subtraction operator. It is represented by a UML enumeration. 'Fraction' is a composite type that represents a fraction. It has two blocks 'over' and 'under' that represent the numerator and the denominator of the fraction, respectively. Both blocks are of the primitive type 'Integer' that is pre-defined in UML. 'Fraction' is represented by a UML class.

Whether an addition or a subtraction operation is performed is determined by the causality constraints on the enabling conditions of the actions that represent those operations. According to those causality constraints, if the parameter 'operation' is set to 'add', the 'Add' action is enabled. If the parameter 'operation' is set to 'subtract', the 'Subtract' action is enabled. The 'Add' action establishes a result 'toAdd' that represents the fraction to add to the parameter 'operand'. It also establishes a result 'result' that represents the addition of 'toAdd' to the parameter 'operand'. The 'Subtract' action establishes similar results.

The 'ChooseOp' action establishes a result 'operand' that represents the operand for the next operation to perform. 'operand' is set to the result established in either the 'Add' or the 'Subtract' operation, depending on which operation was performed.

Figure 4-50.ii shows the results of some instances of the action instantiations from Figure 4-50.i. ':Add' represents the results of an occurrence of the action 'Add' and ':ChooseOp' represents the results of an occurrence of the action 'ChooseOp'.

To create valid OCL constraints to represent parameter, causality and attribute constraints, we must make some assumptions about the existence of UML constructs in a design. For example, '1.operand=add' in the example from Figure 4-50 is not a valid OCL constraint, unless we assume that an association exists between the class that is the context of the constraint and the class that represents the parameters of '1'.

We assume that the UML classes and data types that represent the information types are contained in a UML package. Also, we assume that a UML package exists for each behaviour declaration. This package has the same name as the declaration and represents the behaviour declaration in UML. It contains the classes that correspond to attributes of causality target instantiations from the represented behaviour declaration. It contains packages that correspond to behaviour declarations inside the represented behaviour declaration. Also, it imports the UML package that represents the information types.

If a causality target instantiation or point causally depends on another causality target instantiation or point, we add an association between the UML classes that represent the corresponding attributes or parameters. An instance of such an association represents that one of the corresponding causality targets caused the other. The association has '0..1' multiplicity on the causer side, because a causality target can be caused by at most one instance of another causality target instantiation.

An attribute constraint of an alternative condition only applies when the occurrence of its causality target is caused by that alternative condition. We assume that this requirement is incorporated into the OCL constraint that represents the attribute constraint. We can incorporate these requirements using OCL expressions on the associations that represent that one causality target causes another. For example, in Figure 4-50 'operand = Add.result' only applies if 'ChooseOp' was caused by 'Add'. We can add this requirement to the constraint, as: 'not Add.OclIsUndefined() implies operand = Add.result'. This means that if an association exists between 'self' and an instance of the class 'Add', then the alternative attribute constraint applies.

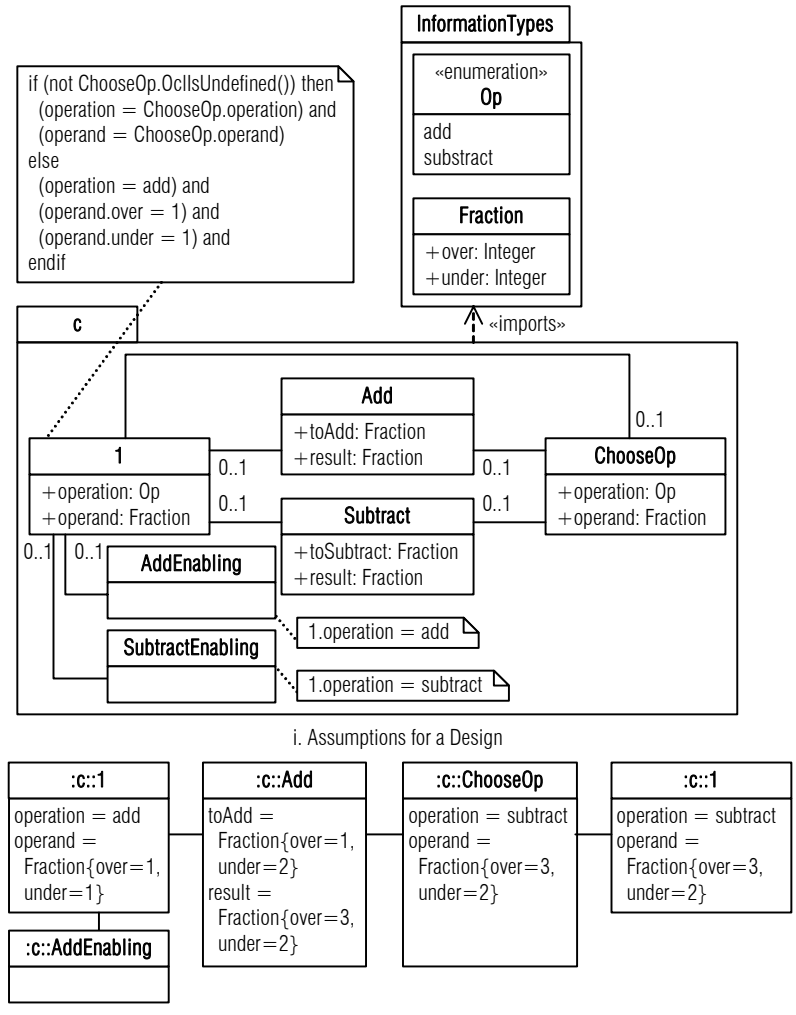
Example 4-10 Assumptions for a UML Binding of Information Concepts

Figure 4-51.i shows some of the assumptions that must be made to correctly represent the information types from Figure 4-50 in UML. It shows that packages are assumed to exist for the definition of the information types and for the behaviour declaration 'Calculator c'. The package for the behaviour declaration contains classes that represent the results of each of the points and causality targets in the corresponding behaviour type. It also contains the classes that are the contexts for the causality constraints for the 'Add' and 'Subtract' actions. 'Add' has an association to '1', because its action may be caused by (the condition represented by) '1'. 'ChooseOp' has an association to both 'Add' and 'Subtract', because its action may be caused by those 'Add' or by 'Subtract'. '1' has a parameter constraint. This constraint either sets the parameters of '1' to a pre-defined value or to the results of the 'ChooseOp' action, depending on

whether '1' was enabled by the 'ChooseOp' action or a start condition.

Figure 4-51.ii shows the occurrences of some actions that are specified by the design. In these occurrences an instance of '1' is enabled, such that the class that represents the parameters of '1' is instantiated. '1' has parameters that represent the addition operator and the fraction 1/1. '1' enables and causes an occurrence of the 'Add' action, such that the class that represents the causality condition of 'Add' and the class that represents the results of 'Add' are instantiated. These instances are associated to '1', because '1' causes the enabling and instantiation of 'Add'. Further, the figure shows that an instance of 'ChooseOp' occurs and that another instance of '1' is enabled. These occurrences and their results meet the OCL constraints specified in the design.

Figure 4-51 Example of Assumptions for a UML Binding of Information Concepts



Chapter 5

Pre-Defined Viewpoint Relations

This chapter investigates the notions of refinement and overlap from chapter 3 in more detail and identifies some frequently occurring cases of refinement and overlap. Based on these frequently occurring cases, this chapter pre-defines some refinement and overlap relations that a designer can re-use to prescribe a relation between viewpoints. The pre-defined relations are associated with consistency rules that a designer can re-use to verify consistency between views.

The pre-defined refinement relations and consistency rules focus on the conditions for the occurrence of actions and interactions. They do not consider attribute constraints, nor do they consider other design concerns, such as structural concerns or security concerns. We leave verification of consistency with respect to attribute constraints and with respect to other concerns for future work.

The work described in sections 5.1, 5.2.1, 5.2.3 and 5.2.4 is an extension of the work described by Quartel et al. (2002) and Quartel (1998). It contains a more algorithmic interpretation of that work, which is necessary for the implementation of the work in tool support. The work described in sections 5.2.5, 5.2.6 and 5.3 is a new contribution.

We define the viewpoint relations and consistency rules on a textual concrete syntax of the basic concepts that we explain in section 5.1, along with some basic operations and relations on the concrete syntax that are used further in the chapter. Subsequently, we explain refinement and overlap in sections 5.2 and 5.3, respectively.

5.1 Textual Concrete Syntax for Basic Concepts

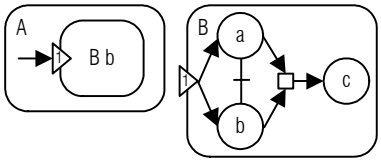
We define a textual concrete syntax to represent basic behaviours. Also, we define some rules, relations and operations that apply to behaviours.

Textual concrete syntax for behaviour types. In the textual concrete syntax, action, interaction contribution, interaction, point and behaviour instances as well as their types are represented by the names of those instances or types. Instantiations are represented by their name, suffixed by the name of the type that they instantiate between brackets. The name of an entry point is prefixed by >. The name of an exit point is suffixed by >. If only one instantiation of a particular action or interaction contribution type exists in a behaviour type, the type of that action or interaction contribution can be left out. If we refer to a point or interaction contribution instantiation *of* a particular behaviour instantiation, we refer to it as <name of behaviour instantiation>.<name of interaction contribution instantiation or point>. For example, the name of exit point *1* of behaviour instantiation *b* of behaviour type *B* becomes *b(B).1*>. We can also use this notation to uniquely identify two instances or instantiations with the same name that belong to different behaviours. By convention, we use upper case letters to represent types, lower case letters to represent instances and instantiations and numbers to represent points.

A behaviour type consists of action and interaction contribution instantiations, as well as the (recursive) behaviour instantiations that can occur in the context of that behaviour type and the exit points that the behaviour type makes available. We represent the relation between a behaviour type and its constituents by representing the behaviour type as a set that contains these constituents. Each of a behaviour type's constituents is prefixed by an arrow and a causality condition for its occurrence, or, in case of an exit point, the causality condition that it represents. Figure 1-2 illustrates the textual representation of two behaviour types and their constituents. It also illustrates the equivalent graphical representation of the behaviour types.

Figure 5-1 Textual Representation of Behaviour Types

$$
\begin{aligned}
A &= \{\surd \rightarrow b(B).>1\} \\
B &= \{\ >1 \wedge \neg b \ \rightarrow a, \\
&\qquad >1 \wedge \neg a \ \rightarrow b, \\
&\qquad a \vee b \qquad \rightarrow c\}
\end{aligned}
$$



Causality conditions are represented as follows. An enabling condition is represented by the name of the instantiation by which the target is enabled. A disabling condition is represented by the logical not (¬) followed by the name of the instantiation by which the target is disabled. The start condition is represented by √. The disjunction of conditions is represented by the logical or (∨) of those conditions. The conjunction of conditions is represented by the logical and (∧) of those conditions.

By convention we refer to an arbitrary basic causality condition as γ and to an arbitrary disjunction or conjunction as Γ.

Commutativity, associativity and distributivity. Some rules apply to causality conditions. The disjunction and conjunction of conditions are associative and commutative, such that for each causality condition Γ_1, Γ_2 and Γ_3:

$$\Gamma_1 \wedge (\Gamma_2 \wedge \Gamma_3) = (\Gamma_1 \wedge \Gamma_2) \wedge \Gamma_3 \quad (1)$$
$$\Gamma_1 \vee (\Gamma_2 \vee \Gamma_3) = (\Gamma_1 \vee \Gamma_2) \vee \Gamma_3 \quad (2)$$

$$\Gamma_1 \wedge \Gamma_2 = \Gamma_2 \wedge \Gamma_1 \quad (3)$$
$$\Gamma_1 \vee \Gamma_2 = \Gamma_2 \vee \Gamma_1 \quad (4)$$

Conjunction distributes over disjunction, such that for causality conditions Γ_1, Γ_2 and Γ_3:

$$\Gamma_1 \wedge (\Gamma_2 \vee \Gamma_3) = (\Gamma_1 \wedge \Gamma_2) \vee (\Gamma_1 \wedge \Gamma_3) \quad (5)$$

Disjunction does not distribute over conjunction.

The conjunction or disjunction of two syntactically equivalent conditions is equivalent to one of these conditions:

$$\Gamma \wedge \Gamma = \Gamma \quad (6)$$
$$\Gamma \vee \Gamma = \Gamma \quad (7)$$

Impossible conditions. A user may inadvertently specify a causality condition that is impossible to satisfy, such that the target of the condition can never occur. We specify some rules to detect impossible conditions, such that we can remove them. We use the symbol † to denote the impossible condition. The equations below are the equations for calculating with impossibility.

For each action or interaction contribution instantiation a:

if $\Gamma \rightarrow a$ and a appears in Γ then a can be replaced by † in Γ (8)

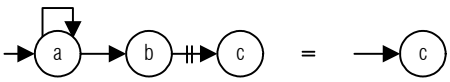
if $\Gamma \rightarrow a$ and $\neg a$ appears in Γ then $\neg a$ can be replaced by † in Γ (9)

$$a \wedge \neg a = \dagger \quad (10)$$
$$\dagger \wedge \Gamma = \dagger \quad (11)$$
$$\dagger \vee \Gamma = \Gamma \quad (12)$$
$$\neg\dagger = \surd \quad (13)$$

if $\dagger \rightarrow a$ then a can be replaced by † in each condition Γ (14)

Equations 8 and 9 represent that an action or interaction contribution cannot depend on the (non-)occurrence of itself. Hence, if the enabling or disabling of an action or interaction contribution appears in the condition of that action or interaction contribution, it can be replaced by the impossible condition. Equation 10 represents that the conjunction of the enabling and disabling of an action or interaction contribution cannot be satisfied and is therefore impossible. The other equations represent simplification rules for the impossible condition. Equation 11 represents that the conjunction of the impossible condition and another condition is equivalent to the impossible condition. The conjunction of a condition and the impossible condition is impossible, because in the conjunction of some conditions *all* conditions must be satisfied for that condition to be satisfied, while the impossible condition can never be satisfied. Equation 12 represents that in the disjunction of the impossible condition and another condition, the impossible condition can be removed. This is so, because the target of the disjunction can never occur as a result of the impossible condition. Equation 13 represents that the disabling of the impossible condition is equivalent to the start condition. This is so, because an impossible action or interaction contribution is always disabled. Finally, equation 14 represents that if the causality condition of an action or interaction contribution is impossible, then a reference to that action or interaction contribution in a causality condition can be replaced by the impossible condition. After simplifying a behaviour according to the rules above, we can remove the actions and interaction contributions from that behaviour that are impossible.

Figure 5-2 Example of a Behaviour with Impossible Actions



Example 5-1 A Behaviour with Impossible Actions

Figure 5-2 shows an example of a behaviour in which impossible actions exist. Action *a* is impossible, because it is enabled by itself. Action *b* is impossible, because it depends on the impossible action *a*. The behaviour can be simplified by replacing the condition of action *c* by the start condition and removing the impossible actions *a* and *b*.

Formally, the behaviour can be simplified as follows:

$$\{\surd \wedge a \rightarrow a, a \wedge (c \vee \neg c) \rightarrow b, \neg b \rightarrow c\} =_{8}$$
$$\{\surd \wedge \dagger \rightarrow a, a \wedge (c \vee \neg c) \rightarrow b, \neg b \rightarrow c\} =_{11}$$
$$\{\dagger \rightarrow a, a \wedge (c \vee \neg c) \rightarrow b, \neg b \rightarrow c\} =_{14}$$
$$\{\dagger \rightarrow a, \dagger \wedge (c \vee \neg c) \rightarrow b, \neg b \rightarrow c\} =_{11}$$
$$\{\dagger \rightarrow a, \dagger \rightarrow b, \neg b \rightarrow c\} =_{14,13}$$
$$\{\dagger \rightarrow a, \dagger \rightarrow b, \surd \rightarrow c\}$$

Disjunctive normal form. Many of the formulae in this chapter assume that the causality conditions of a behaviour type are in the disjunctive normal form, because the disjunctive normal form is a convenient form to perform calculations over causality conditions.

Definition 5-1 Disjunctive Normal Form

The disjunctive normal form of a causality condition is the causality condition that is specified as a disjunction of conjunctions.

More precisely, a causality condition Γ for a causality target a is in the disjunctive normal form, if it has the form: $\Gamma_1 \vee \Gamma_2 \vee \ldots \vee \Gamma_n$ such that each Γ_i does not contain a disjunction. We call each Γ_i an *alternative causality condition* for a, because it is a sufficient condition for a to occur. We define the function 'alternatives' to obtain the set of alternative conditions for a causality condition in the disjunctive normal form. Hence, for a condition $\Gamma = \Gamma_1 \vee \Gamma_2 \vee \ldots \vee \Gamma_n$ in the disjunctive normal form, we define the alternatives of Γ as follows:

$$\text{alternatives}\ \Gamma = \{\Gamma_1, \Gamma_2, \ldots, \Gamma_m\}$$

A behaviour is in the disjunctive normal form, if each of its causality conditions is in the disjunctive normal form.

We define the function 'dnf' to compute the disjunctive normal form for a causality condition as follows:

$$\text{dnf}\ \Gamma_1 \vee \Gamma_2 = (\text{dnf}\ \Gamma_1) \vee (\text{dnf}\ \Gamma_2)$$

$$\text{dnf}\ \Gamma_1 \wedge \Gamma_2 = \bigvee_{\substack{\Gamma_1' \in (\text{alternatives.dnf})\,\Gamma_1, \\ \Gamma_2' \in (\text{alternatives.dnf})\,\Gamma_2}} (\Gamma_1' \wedge \Gamma_2')$$

$$\text{dnf}\ \gamma = \gamma$$

The 'dot' represents function composition. For example, 'alternatives.dnf' represents that we first apply the function 'dnf' and then we apply the function 'alternatives' to the result of 'dnf'. The function 'dnf' takes a causality condition as input. If the causality condition is a disjunction of two parts, then the condition is already in the disjunctive normal form except (possibly) for its parts. Hence, in that case the formula returns the disjunction of the disjunctive normal forms of the two parts. If the causality condition is the conjunction of two parts, then the condition is not yet in the disjunctive normal form. Hence, in that case the formula first computes the disjunctive normal form of the parts. This results in two causality conditions in disjunctive normal form. The conjunction of each pair of conditions, where one condition is an alternative causality condition of one part and the other is an alternative causality condition of the other part, is an alternative of the

resulting causality condition. If the causality condition is a basic condition, then the condition is already in the disjunctive normal form and nothing has to be done. Hence, in that case the formula returns the causality condition without changes.

Example 5-2 Calculating the Disjunctive Normal Form

As an example of calculating the disjunctive normal form, consider the behaviour:

$B = \{ \surd \rightarrow a, a \wedge (c \vee \neg c) \rightarrow b, \neg b \rightarrow c, (a \vee b) \wedge (a \vee c) \rightarrow d \}$

The condition for b, $a \wedge (c \vee \neg c)$, is not in the disjunctive normal form. We can use the function 'dnf' to compute its disjunctive normal form as follows:

$$\text{dnf } a \wedge (c \vee \neg c) =$$

$$\bigvee_{\substack{\Gamma_1' \in (\text{alternatives.dnf})\, a \\ \Gamma_2' \in (\text{alternatives.dnf})\, c \vee \neg c}} (\Gamma_1' \wedge \Gamma_2') =$$

$$\bigvee_{\substack{\Gamma_1' \in \{a\} \\ \Gamma_2' \in \{c, \neg c\}}} (\Gamma_1' \wedge \Gamma_2') =$$

$$(a \wedge c) \vee (a \wedge \neg c)$$

The condition for d, $(a \vee b) \wedge (a \vee c)$ also is not in the disjunctive normal form. We can compute the disjunctive normal form for that condition as follows:

$$\text{dnf } (a \vee b) \wedge (a \vee c) =$$

$$\bigvee_{\substack{\Gamma_1' \in \{a,b\} \\ \Gamma_2' \in \{a,c\}}} (\Gamma_1' \wedge \Gamma_2') =$$

$$(a \wedge a) \vee (a \wedge c) \vee (b \wedge a) \vee (b \wedge c)$$

Alternative behaviours and their combination. Just as the specification of causality conditions in the disjunctive normal form makes it easier to perform computations over causality conditions, the specification of a behaviour as a set of *alternative behaviours* makes it easy to perform calculations over behaviours. Therefore, we introduce the notion of alternative behaviour.

Definition 5-2 Alternative Behaviour

An alternative behaviour of a behaviour is a combination of one alternative causality condition of each of the causality relations in that behaviour.

Hence, for a behaviour $B = \{ \Gamma_{a1} \rightarrow a_1, \Gamma_{a2} \rightarrow a_2, \ldots, \Gamma_{an} \rightarrow a_n \}$ in the disjunctive normal form the set of alternative behaviours is the set of each possible combination of alternative causality conditions. We introduce the function 'Alt' to compute the set of alternative behaviours of B:

$$\begin{aligned}\text{Alt } B = \{\{\Gamma_1 \to a_1, \Gamma_2 \to a_2, \ldots, \Gamma_n \to a_n\} \mid\ & \Gamma_1 \in \textit{alternatives}\ \Gamma_{a_1},\\ & \Gamma_2 \in \textit{alternatives}\ \Gamma_{a_2},\\ & \ldots,\\ & \Gamma_n \in \textit{alternatives}\ \Gamma_{a_n}\}\end{aligned}$$

After we perform a computation over a set of alternative behaviours, these alternative behaviours must be combined again into a regular behaviour. Alternative behaviours can be combined, by considering each condition for a particular action in an alternative behaviour an alternative condition for that action. For a set of alternative behaviours

$$\begin{aligned}BS = \{\ & \{\Gamma_{11} \to a_1, \Gamma_{21} \to a_2, \ldots, \Gamma_{n1} \to a_n\},\\ & \{\Gamma_{12} \to a_1, \Gamma_{22} \to a_2, \ldots, \Gamma_{n2} \to a_n\},\\ & \ldots,\\ & \{\Gamma_{1m_1} \to a_1, \Gamma_{2m_2} \to a_2, \ldots, \Gamma_{nm_n} \to a_n\}\}\end{aligned}$$

we introduce the function '$\circ$' to compute their combination:

$$\begin{aligned}\circ BS = \{\ & \Gamma_{11} \vee \Gamma_{12} \vee \ldots \vee \Gamma_{1m_1} \to a_1,\\ & \Gamma_{21} \vee \Gamma_{22} \vee \ldots \vee \Gamma_{2m_2} \to a_2,\\ & \ldots,\\ & \Gamma_{n1} \vee \Gamma_{n2} \vee \ldots \vee \Gamma_{nm_n} \to a_n\}\end{aligned}$$

The alternatives of a behaviour or the combination of alternative behaviours cannot be calculated for the class of *one-and-a-half sided causality relations*. This is the class of relations where, in some alternatives, an action a depends on another action b and b depends on a, while, in other alternatives a depends on b but b does not depend on a. More precisely, it is the class of behaviours B, where for some actions a and b:

$\Gamma_{a1} \vee \Gamma_{a2} \to a \in B$ and $\Gamma_{b1} \vee \Gamma_{b2} \to b \in B$
such that b or $\neg b$ appears in Γ_{a1} and b or $\neg b$ appears in Γ_{a2}
a or $\neg a$ appears in Γ_{b1} but not in Γ_{b2}

Even though one-and-a-half sided relations do not appear in a behaviour, they may appear in the alternatives of that behaviour as a consequence of some operation performed on those alternatives. A one-and-a-half sided relation manifests itself in a set of alternative behaviours as follows. In all

alternative behaviours b depends on a, but in some alternative behaviours a depends on b, while in others a does not depend on b. More precisely, alternative behaviours BS contain a one-and-a-half-sided relation between a and b if:

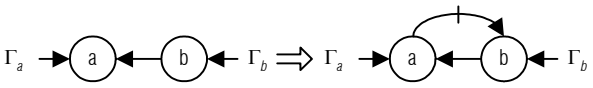
In all alternative behaviours $AB \in BS$
$\Gamma_a \rightarrow a \in AB$ such that b appears in Γ_a
and there exist alternative behaviours $AB \in BS$ for which
$\Gamma_b \rightarrow b \in AB$ such that a or $\neg a$ appears in Γ_b
and there exist alternative behaviours $AB \in BS$ for which
$\Gamma_b \rightarrow b \in AB$ such that a or $\neg a$ does not appear in Γ_b

Well-formedness rules demand that, if $\neg b$ appears in Γ_a in some alternative behaviour, a or $\neg a$ appears in Γ_b in that same alternative behaviour (Quartel, 1998). Therefore, we only need to consider one-and-a-half sided relations where b appears in Γ_a. Current experience shows that we can solve the problem of not being able to combine alternative behaviours that contain a one-and-a-half-sided relations in the following way (Quartel, 1998). If some alternative behaviours BS contain a one-and-a-half-sided relation between a and b, then:

in all alternative behaviours $AB \in BS$ in which
$b \wedge \Gamma_a \rightarrow a \in AB$ and $\Gamma_b \rightarrow b \in AB$
such that a or $\neg a$ does not appear in Γ_b
replace:
$\Gamma_b \rightarrow b$ by $\neg a \wedge \Gamma_b \rightarrow b$ in AB

Graphically, this replacement is represented in Figure 5-3.

Figure 5-3 Replacing One-and-a-half-sided Relations



Example 5-3 Decomposition and Combination of Behaviour

Figure 5-4 and Figure 5-5 show an example of the decomposition of a behaviour into alternative behaviours and the composition of alternative behaviours into a behaviour, respectively. Figure 5-4 shows the behaviour: $B = \{\neg b \rightarrow a, \neg a \rightarrow b, a \vee b \rightarrow c, a \vee b \rightarrow d\}$. Each possible combination of alternative conditions for c and d yields an alternative behaviour. Hence, the set of alternative behaviour BS is:

$BS = \{ \{\neg b \rightarrow a, \neg a \rightarrow b, a \rightarrow c, b \rightarrow d\},$
$\{\neg b \rightarrow a, \neg a \rightarrow b, a \rightarrow c, a \rightarrow d\},$
$\{\neg b \rightarrow a, \neg a \rightarrow b, b \rightarrow c, b \rightarrow d\},$
$\{\neg b \rightarrow a, \neg a \rightarrow b, b \rightarrow c, a \rightarrow d\}\}$

The combination of these alternatives behaviour can be computed using the '$\circ$' operator, as

follows:

$$\begin{aligned}\circ BS &= \{\ \neg b \vee \neg b \vee \neg b \vee \neg b \to a,\\ &\quad\ \neg a \vee \neg a \vee \neg a \vee \neg a \to b,\\ &\quad\ a \vee a \vee b \vee b \to c,\\ &\quad\ b \vee a \vee b \vee a \to d\ \}\\ &=_{4,7} \{\ \neg b \to a, \neg a \to b, a \vee b \to c, b \vee a \to d\ \}\end{aligned}$$

Figure 5-5 shows two alternative behaviours that imply a one-and-a-half-sided relation between *a* and *b*, because *b* depends on *a* in both alternatives, while *a* depends on *b* in one alternative but not in the other. Hence, before we can combine these alternatives, we must add the condition $\neg b$ to the condition of *a* in the alternative behaviour in which *a* does not depend on *b*. After which we can combine the alternative behaviours. More precisely:

$$\begin{aligned}BS &= \{\{\surd \to \mathrm{a}, \mathrm{a} \to \mathrm{b}\ \}, \{\neg \mathrm{b} \to \mathrm{a}, \neg \mathrm{a} \to \mathrm{b}\ \}\}\\ &= \{\{\surd \wedge \neg \mathrm{b} \to \mathrm{a}, \mathrm{a} \to \mathrm{b}\ \}, \{\neg \mathrm{b} \to \mathrm{a}, \neg \mathrm{a} \to \mathrm{b}\ \}\}\end{aligned}$$

After which:

$$\circ BS = \{\ (\surd \wedge \neg b) \vee \neg b \to a, a \vee \neg a \to b\ \}$$

Figure 5-4 Example of Decomposition of Behaviour into Alternative Behaviours

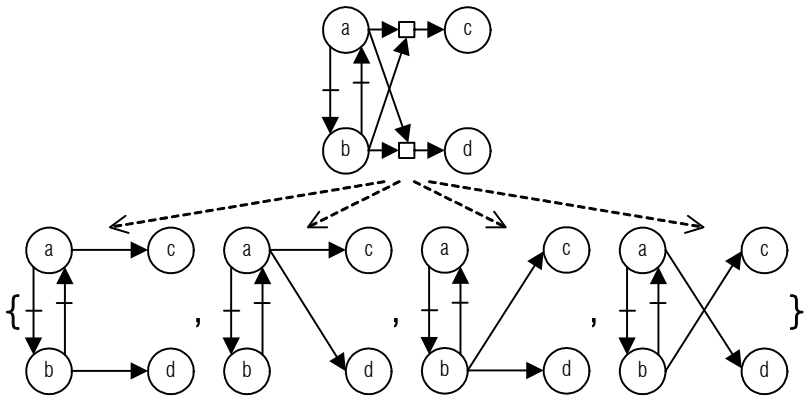
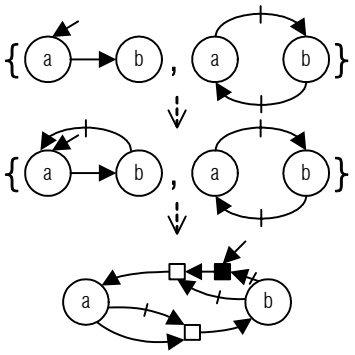


Figure 5-5 Example of Combination of Alternative Behaviours



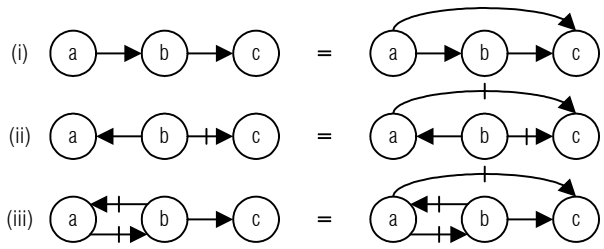


Figure 5-6 Implied Relations between Actions

Dependency closure. A relation between actions *a* and *b* and between actions *b* and *c* may imply a relation between actions *a* and *c*. If a relation is implied, it can be added to the behaviour, because the behaviour in which the implied relation is represented explicitly is equivalent with the behaviour in which the implied relation is not added explicitly. We distinguish three possible implied relations. Figure 5-6 illustrates these relations. (i) If *a* enables *b* and *b* enables *c*, then *a* implicitly enables *c*. This follows from the observation that *c* can only occur if *a* has occurred and, if *c* occurs it (indirectly) depends on *a*. (ii) If *b* enables *a* and *b* disables *c*, then *a* implicitly disables *c*. This follows from the observation that *c* can only occur if *a* has not yet occurred, because *a* can only occur if *b* has occurred and that would mean that *c* is disabled (by the occurrence of *b*). (iii) If *a* disables *b* and *b* disables *a* and *b* enables *c*, then *a* implicitly disables *c* (and *c* implicitly disables *a*, but that follows from the previous rule). This follows from the observation that *c* can only occur as long as *a* has not occurred, because if *a* occurs *b* cannot yet have occurred and after *a* has occurred *b* cannot occur anymore. Since *c* can only occur after *b* has occurred, this means that *c*, like *b*, depends on the non-occurrence of *a*. Note that, in this case, also *c* implicitly disables *a*, if we apply rule (ii).

More precisely, for an *alternative behaviour* AB, in which $\Gamma_a \rightarrow$ a, $\Gamma_b \rightarrow$ b and $\Gamma_c \rightarrow$ c:

$$\text{if a appears in } \Gamma_b \text{ and b appears in } \Gamma_c, \text{ then } \Gamma_c = a \wedge \Gamma_c \tag{15}$$

$$\text{if b appears in } \Gamma_a \text{ and } \neg b \text{ appears in } \Gamma_c, \text{ then } \Gamma_c = \neg a \wedge \Gamma_c \tag{16}$$

$$\text{if } \neg b \text{ appears in } \Gamma_a, \neg a \text{ appears in } \Gamma_b \text{ and b appears in } \Gamma_c, \text{ then } \Gamma_c = \neg a \wedge \Gamma_c \tag{17}$$

The explicit inclusion of an implicit relation, may give rise to the explicit inclusion of another relation. For example consider the behaviour: $B = \{ a \rightarrow b, b \rightarrow c, c \rightarrow d \}$, after applying the first rule (twice) this behaviour becomes $B = \{ a \rightarrow b, a \wedge b \rightarrow c, b \wedge c \rightarrow d \}$. Now *a* appears in the condition of *c* and *c* appears in the condition of *d*. Hence, we can apply the first rule again to yield the behaviour: $B = \{ a \rightarrow b, a \wedge b \rightarrow c, a \wedge b \wedge c \rightarrow d \}$. We can use these equations to calculate the dependence closure of a behaviour. Or, in the textual notation:

$$\begin{aligned} \mathcal{B} = \quad & \{\, a \rightarrow b,\ b \rightarrow c,\ c \rightarrow d \,\} =_{15} \\ & \{\, a \rightarrow b,\ a \wedge b \rightarrow c,\ b \wedge c \rightarrow d \,\} =_{15} \\ & \{\, a \rightarrow b,\ a \wedge b \rightarrow c,\ a \wedge b \wedge c \rightarrow \mathrm{d} \,\} \end{aligned}$$

Definition 5-3 Dependency Closure

The dependency closure of a behaviour is the behaviour in which all causality conditions have been rewritten to include all implicit causality conditions.

The dependency closure of an alternative behaviour can be calculated, using the algorithm 'ADep' below. In this algorithm AB' := AB represents the assignment of the causality relations of AB to AB'. AB = AB' is true if the causality condition of each causality target in AB is equivalent to the condition of that causality target in AB' using associative, commutative and distributive rewriting and rewriting of impossible conditions (using equations 1-14 defined above).

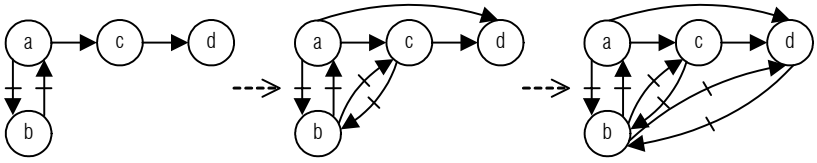
```
ADep AB:
    repeat
        AB' := AB
        for each Γa → a, Γb → b, Γc → c ∈ AB:
            if a appears in Γb, b appears in Γc and a does not appear in Γc
                then
                    replace Γc by a ∧ Γc in AB
            if b appears in Γa, ¬b appears in Γc
                and ¬a does not appear in Γc then
                    replace Γc by ¬a ∧ Γc in AB
            if ¬b appears in Γa, ¬a appears in Γb, b appears in Γc
                and ¬a does not appear in Γc then
                    replace Γc by ¬a ∧ Γc in AB
    until AB' = AB
    return AB'
```

We introduce the function 'Dep' to compute the dependency closure of a behaviour $\mathcal{B}$, which is the dependency closure of each of the alternatives of this behaviour:

Dep $\mathcal{B}$ = ° (map ADep (Alt $\mathcal{B}$))

Where 'map' is the function that applies a function to each element of a set.

Figure 5-7 Example of Calculating Dependency Closure



Example 5-4 Calculating the Dependency Closure of a Behaviour

Figure 5-7 shows an example of calculating the dependency closure of a behaviour in two steps. In the first step, the implied enabling condition between actions *a* and *d* according to equation 15, is added to the behaviour. Also, the implied disabling conditions that exist between actions *b* and *c* according to equations 16 and 17, are added to the behaviour. In the resulting behaviour two disabling conditions are implied between actions *b* and *d* according to equations 16 and 17. These conditions are added to the behaviour in the second step. In the textual notation this can be represented as:

$$
\begin{aligned}
B = \quad & \{ \neg b \rightarrow a, \neg a \rightarrow b, a \rightarrow c, c \rightarrow d \} =_{15,16,17} \\
& \{ \neg b \rightarrow a, \neg a \wedge \neg c \rightarrow b, a \wedge \neg b \rightarrow c, a \wedge c \rightarrow d \} =_{16,17} \\
& \{ \neg b \rightarrow a, \neg a \wedge \neg c \wedge \neg d \rightarrow b, a \wedge \neg b \rightarrow c, a \wedge \neg b \wedge c \rightarrow d \}
\end{aligned}
$$

Strong behaviour equivalence. We define strong equivalence ($\sim$) between behaviours as follows.

Definition 5-4 Strong Behaviour Equivalence

Two behaviours are strongly equivalent if they prescribe syntactically equivalent (implicit) causality conditions for their causality targets.

Hence, two behaviours are strongly equivalent if one can be derived from the other, using the equations for commutativity, associativity, distributivity and implicit conditions, defined above. We present the following algorithm to compute strong equivalence. For two behaviour B_1 and B_2 in the disjunctive normal form:

$B_1 \sim B_2$:
  $BS_1 :=$ ((map ADep).Alt) B_1
  $BS_2 :=$ ((map ADep).Alt) B_2
  if
    for each $AB_1 \in BS_1$ there exists an $AB_2 \in BS_2$ such that
      for each $(\Gamma_1 \rightarrow a) \in AB_1$ there exists an $(\Gamma_2 \rightarrow a) \in AB_2$,
        such that equivalentalternatives Γ_1 Γ_2
      and
      for each $(\Gamma_2 \rightarrow a) \in AB_2$ there exists an $(\Gamma_1 \rightarrow a) \in AB_1$,
        such that equivalentalternatives Γ_1 Γ_2
    and
    for each $AB_2 \in BS_2$ there exists an $AB_1 \in BS_1$ such that
      for each $(\Gamma_1 \rightarrow a) \in AB_1$ there exists an $(\Gamma_2 \rightarrow a) \in AB_2$,
        such that equivalentalternatives Γ_1 Γ_2
      and
      for each $(\Gamma_2 \rightarrow a) \in AB_2$ there exists an $(\Gamma_1 \rightarrow a) \in AB_1$,
        such that equivalentalternatives Γ_1 Γ_2
  then return true else return false

The algorithm first computes the dependency closure of each alternative of the two behaviours. Subsequently, it returns true if for each alternative of B_1 there exists an equivalent alternative in B_2. Equivalence between alternatives is assessed, by checking if for each condition in one alternative, an equivalent condition exists in the other alternative. For this purpose, the algorithm uses the 'equivalentalternatives' function that returns true if and only if two alternative conditions are syntactically equivalent, not regarding the order in which their basic conditions are specified. More precisely:

$$\text{equivalentalternatives } \Gamma_1\ \Gamma_2 = ((\text{basicconditions } \Gamma_1) = (\text{basicconditions } \Gamma_2))$$

$$\text{basicconditions } \Gamma_1 \vee \Gamma_2 = (\text{basicconditions } \Gamma_1) \cup (\text{basicconditions } \Gamma_2)$$
$$\text{basicconditions } \Gamma_1 \wedge \Gamma_2 = (\text{basicconditions } \Gamma_1) \cup (\text{basicconditions } \Gamma_2)$$
$$\text{basicconditions } \gamma = \{\gamma\}$$

Concrete syntax for interactions. In the textual concrete syntax, we represent an interaction by its name prefixed by a double arrow ($\Rightarrow$) and the interaction contributions and behaviours that participate in the interaction. We represent the interaction contributions and behaviours that participate in the interaction by the logical and ($\wedge$) of the interaction contributions that must be enabled for the interaction to occur. The interaction contributions are identified by the name of the behaviour instantiation of which they are a part 'dot' their name. If alternative interaction contributions exist, the alternative interaction contributions are separated by a logical or ($\vee$).

Figure 5-8 Textual Representation of Interactions

$$(a.a_1 \wedge b.a) \vee (a.a_2 \wedge b.a \wedge c.a) \Rightarrow a$$

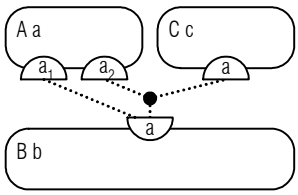


Figure 5-8 shows an example of the textual representation of an interaction as well as the graphical representation of that interaction. It shows an interaction by the name *a* that is made up of two alternatives.

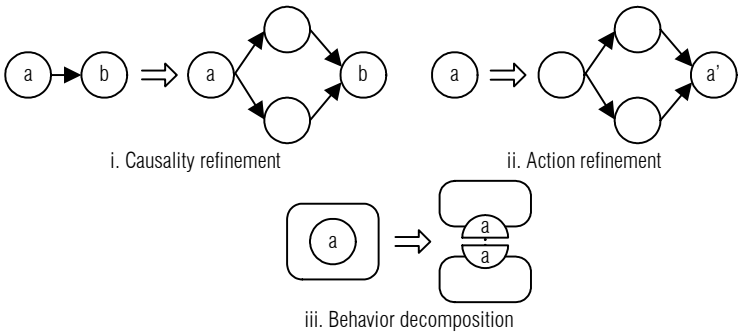
Interactions are defined in a way that is very similar to causality relations in the textual notation. Therefore, we also speak of the condition for the occurrence of an interaction. A difference between a causality condition and an interaction condition is that an interaction depends on the *enabling* of interaction contributions, while an action or interaction contribution depends on the *occurrence* of other actions or interaction contributions. Also,

there is no equivalent to the disabling condition in an interaction condition. However, we can derive the disjunctive normal form of an interaction condition. For this, we can use the same algorithm that we defined for deriving the disjunctive normal form of a causality condition. We will also speak of the alternatives of an interaction condition in the same way as we speak of the alternatives of a causality condition. By convention, we will refer to interaction conditions using the Greek letter theta (θ or Θ).

5.2 Pre-Defined Refinement Relations and Consistency Rules

We distinguish three basic ways in which (the behavioural concern of) a design can be refined into a more detailed design. Based on these cases of refinement, we pre-define means to specify consistency rules between views.

Figure 5-9 Different Forms of Refinement



5.2.1 Cases of Refinement

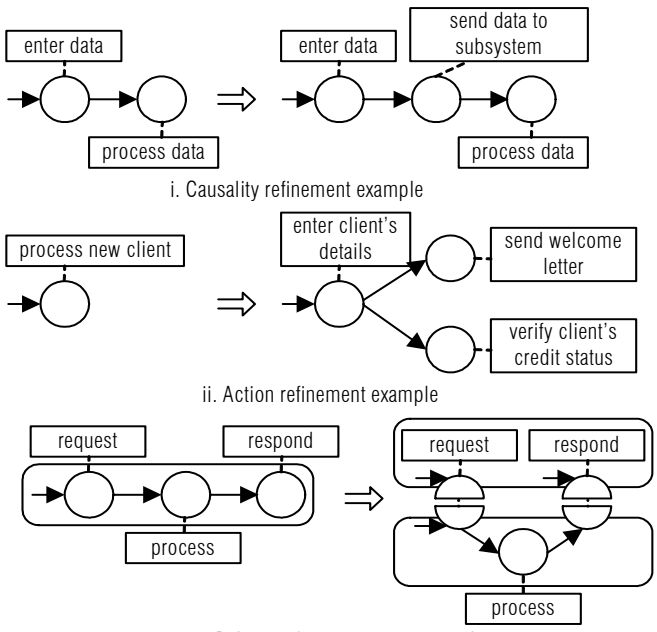
We consider that a behaviour can be refined in the following ways (Quartel, 1998):

1. A behaviour can be refined by decomposing a relation between two of its activities into multiple relations, optionally introducing activities to connect these relations. We represent this case of refinement as *causality refinement*, illustrated in Figure 5-9.i. In case of causality refinement a causality constraint is split up into two or more constraints. Optionally, actions are inserted to represent the activities that connect the refining constraints. These actions are called *inserted actions*, because they do not coincide with the completion of any of the original actions. Figure 5-10.i shows an example of causality refinement. In the example, the action '*send data to subsystem*' is inserted as the result of refining the constraint of the '*process data*' action.

2. A behaviour can be refined by describing one of its activities as a composition of more fine-grained activities. We represent this case of refinement as *action refinement*, illustrated in Figure 5-9.ii. In case of *action refinement* an action is refined into multiple actions with relations between them. Figure 5-10.ii shows an example of action refinement. When an action is refined, some of the actions from the refinement coincide with the completion of the original action. We call these actions *final actions*. In the example, '*send welcome letter*' and '*verify client's status*' are final actions for '*process new client*', because the original action completes if both these actions complete. Actions from the refinement that do not coincide with the completion of the original action are inserted actions. In the example, '*enter client details*' is an inserted action.
3. The behaviour of an entity can be refined by decomposing it into multiple behaviours that represent the individual contributions of parts of the original entity. As a result of this decomposition, the assignment of responsibilities of parts for performing activities are also considered. We represent this case of refinement as *behaviour decomposition*, illustrated in Figure 5-9.iii. In case of behaviour composition a behaviour is refined into multiple behaviours with interactions between them. Figure 5-10.iii shows an example of behaviour decomposition. In the example, actions '*request*' and '*respond*' are refined into interactions.

In general a behaviour will be refined, using a combination of the basic cases identified here.

Figure 5-10 Example of Refinement



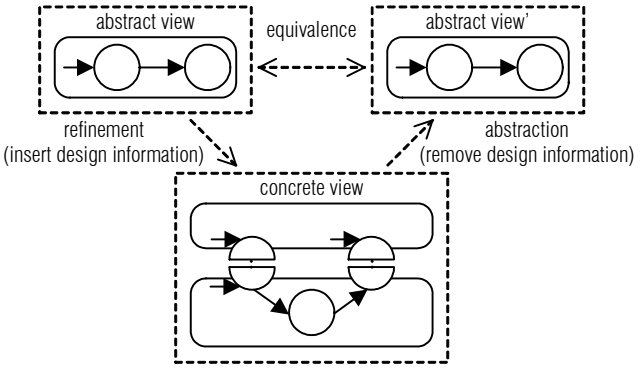


Figure 5-11 Refinement and Consistency Verification

5.2.2 Consistency Verification and Specifying Relations between Views

Having identified the ways in which an abstract design may be refined into a concrete design, we distinguish two approaches to check that a concrete design correctly refines (and hence is consistent with) an abstract design:

1. by ensuring that designers only use the aforementioned refinement operations, in which case the refinement may be considered consistent by construction, and;
2. by checking afterwards whether the concrete design can be reached by applying the refinement operations.

Since a designer may experience the refinement rules as overly restrictive, in particular because refinement is a creative activity, we opt for the latter approach. However, it is not feasible to verify consistency between a concrete design and an abstract design by trying to get from the abstract design to the concrete design by applying the refinement operators. The reason for this is that the refinement operators may be applied on any part of a design and in any combination.

Therefore, we use an approach in which we apply inverted refinement operators to abstract from the design details that were inserted during the refinement. After the application of the inverted refinement operations, we have to check if the resulting design is equivalent to the original design. Figure 5-11 illustrates this approach.

Inverted refinement operators. We introduce the following inverted refinement operators, which are explained in more detail in the following sections.

We introduce the *action abstraction* rule as the inverse of the causality refinement rule. The action abstraction rule removes the specified inserted actions (a_1, a_2, …) from a design. For example, in Figure 5-10.i we can abstract from the inserted action '*send data to subsystem*'. The actions that we

must abstract from can be derived from the 'final action of' relation, since each action that is not a final action of another must be an inserted action.

We introduce two rules as the inverse of action refinement: the action abstraction and *action integration* rule. Action abstraction abstracts from the specified inserted actions in an action refinement. Action integration integrates the specified final actions from the action refinement into a single action. Figure 5-10.ii shows an example in which action abstraction must be applied to action '*enter client details*' and action integration must be applied to the actions '*send welcome letter*' and '*verify client's credit status*'. The actions that we must integrate can be derived from the 'final action of' relation, the actions that are final actions of the same action must be integrated.

We introduce the *behaviour composition* rule as the inverse of the behaviour decomposition rule. The composition rule composes the specified behaviours into a single behaviour and composes the interactions between these behaviours into internal actions. Figure 5-10.iii shows an example of behaviour composition. The behaviours that must be composed can be derived from a 'part of' relation, because behaviours that are a part of the same abstract behaviour must be composed.

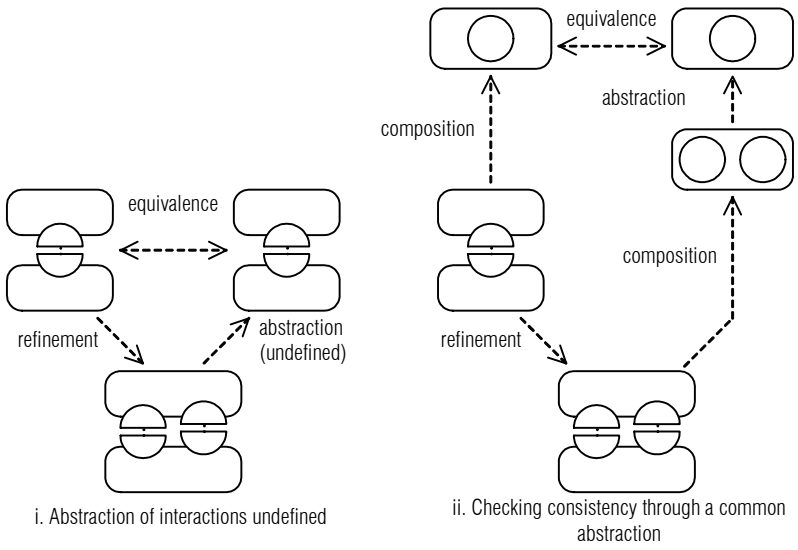


Figure 5-12 Checking Consistency of Interaction Refinement

Checking consistency of interaction refinement. The action abstraction and action integration rules are only defined for actions and interaction contributions that are not part of an interaction. The latter are contributions to interactions for which other behaviours impose no conditions for their occurrence. Checking consistency between designs in which interactions are refined is not possible, because abstraction and integration rules

are not defined for interactions. Figure 5-12.i illustrates this. The definition of abstraction and integration rules for interactions is left for future work.

We can partly check the consistency of an interaction refinement, by using a common abstraction in which the behaviours that participate in the refined interaction are composed. Using this approach, the interactions appear as actions in the composed behaviour and we can integrate and abstract these actions, as illustrated in Figure 5-12.i. We can use this approach, because a composed behaviour is equivalent to its decomposition, except that the distribution of causality conditions among interacting parties is not considered. Hence, the limitation of this approach is that we cannot verify the correctness of the re-distribution of causality conditions among interacting parties in a refinement.

Concrete syntax for specifying consistency rules. To specify consistency rules between views, we introduce the following OCL operations. These operations correspond to the inverted refinement operators above.

The 'abstract' operator takes an object of the behaviour instantiation meta-class (defined in chapter 4) and an object *as* of the OCL Set meta-class that contains objects of the causality target instantiation meta-class (defined in chapter 4). It returns an object of the behaviour instantiation meta-class, in which the causality target instantiations which are members of *as* are abstracted from. Hence, defined in OCL, its signature is as follows:

```
context BehaviourInstantiation def:
    abstract(as: Set): BehaviourInstantiation
```

Since the 'abstract' operator creates a new object of the behaviour instantiation meta-class, we must maintain the relation between causality target instantiations from the behaviour instantiation before abstraction and causality target instantiations from the behaviour instantiation after abstraction. The tracking relation named 'basicrules_abstract_target2target' maintains this relation.

The 'integrate' operator takes an object of the behaviour instantiation meta-class and an object *ccs* of the OCL Set meta-class that contains objects of the OCL String meta-class. It returns an object of the behaviour instantiation meta-class, in which causality target instantiations referenced in *ccs* are integrated according to completion conditions (that we explain in section 5.2.4) that are also specified in *ccs*. Hence, its signature is as follows:

```
context BehaviourInstantiation def:
    integrate(ccs: Set): BehaviourInstantiation
```

Each completion condition *cc* in *ccs* must conform to the following syntax:

```
<cc>          → <condition> TO <cti name>
<condition>   → <alternative> OR <alternative> | <alternative>
<alternative> → <cti name> AND <cti name> | <cti name>
```

Where <cti name> refers to the name of a causality target instantiation.

The tracking relation named 'basicrules_integrate_target2target' maintains the relation between the causality target instantiations from the behaviour instantiation before integration and the causality target instantiations into which they are integrated in the result.

The 'compose' operator takes two objects of the behaviour instantiation meta-class. It returns an object of the behaviour instantiation meta-class that is the result of the composition of those two objects. Hence, its signature is as follows:

```
context BehaviourInstantiation def:
    compose(bi: BehaviourInstantiation): BehaviourInstantiation
```

The tracking relation named 'basicrules_compose_action2interaction' maintains the relation between causality target instantiations in the newly created behaviour instantiation that are the result of composing interactions and the interactions of which they are a composition. The tracking relation 'basicrules_compose_target2target' maintains the relation between other causality target instantiations in the newly created behaviour instantiation and their counterparts.

5.2.3 Action Abstraction

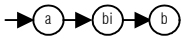


Figure 5-13 Example of an Implicit Causality Condition

The action abstraction operator can be used to remove inserted actions from a design. However, implicit causality relations in which an inserted action is involved must be preserved if that action is removed. Therefore, before removing an inserted action, we explicitly add the implicit causality conditions in which this action is involved. For example, Figure 5-13 shows that action *a* implicitly enables *b* through inserted action *bi*. Before removing *bi*, we must add that enabling condition.

We remove an inserted action *a* by removing its causality relation and all the basic conditions in which it appears. If this results in an empty causality condition, because an action *b* only depended on the inserted action *a*, then that causality condition is replaced by the start condition (√). We define the function 'remove' to remove an inserted action *a* from an alternative behaviour *AB*:

remove a AB
$= \{$ (removeconditions $\{a, \neg a\}$ $\Gamma_b) \rightarrow b \mid (\Gamma_b \rightarrow b) \in AB, b \neq a \}$

removeconditions as $\gamma_1 \wedge \gamma_2 \wedge \ldots \wedge \gamma_n$
$= \wedge \{\gamma \mid \gamma \in \{\gamma_1, \gamma_2, \ldots, \gamma_n\}, \gamma \notin as \}$
, if $\exists_{\gamma \in \{\gamma_1, \gamma_2, \ldots, \gamma_n\}}$ $\gamma \notin as$
$= \surd$, otherwise

To optimize action abstraction, we do not calculate *all* implicit causality relations, but only the implicit causality relations that the inserted actions participate in. For an inserted action a, we can derive these implicit causality relations from:

1. the causality relation of a;
2. alternative causality relations in which the enabling or disabling of a appears; and
3. causality relations of actions that appear in the condition of a, but only the alternatives of those causality relations in which the enabling or disabling of a appears.

We call these causality relations the *causality context* of a in behaviour B. We define the function 'Con' to compute the causality context of an action a in a behaviour B in the disjunctive normal form.

Con a B =
$\{ \Gamma_a \rightarrow a \}$
$\cup$
$\{$(alternativeswith a $\Gamma_b) \rightarrow b \mid (\Gamma_b \rightarrow b) \in B, a \neq b,$ (appearsin a $\Gamma_b)\}$
$\cup$
$\{$(alternativeswith a $\Gamma_b) \rightarrow b \mid (\Gamma_b \rightarrow b) \in B, a \neq b,$ (appearsin b $\Gamma_a)\}$
where $(\Gamma_a \rightarrow a) \in B$

appearsin a Γ
$= (a \in$ (basicconditions $\Gamma)) \vee (\neg a \in$ (basicconditions $\Gamma))$

alternativeswith a $\Gamma_1 \vee \Gamma_2 \vee \ldots \vee \Gamma_n$
$= \vee \{\Gamma \mid \Gamma \in \{\Gamma_1, \Gamma_1, \ldots, \Gamma_n\},$ appearsin a $\Gamma \}$,
if $\exists_{\Gamma \in \{\Gamma_1, \Gamma_1, \ldots, \Gamma_n\}}$ appearsin a Γ
$= \surd$, otherwise

After we abstracted from inserted action a in its causality context, we have to integrate the result with B again. We do that by computing the causality relations that are *not* in the causality context of a in behaviour B and calculating their disjunction with the causality context after abstraction. If an action only appears as a causality target in the causality context or only in

the rest of the behaviour, its causality relation appears in the result unchanged. If an action appears both in the causality context and in the rest and its condition in the causality context is not equal to the start condition, the result contains the disjunction of both conditions. If an action appears both in the causality context and in the rest and its condition in the causality context is equal to the start condition, the result contains the causality relation of that action from the rest of the behaviour. The removal of start conditions in this way is necessary, because start conditions were added when computing the causality context, to yield a well-formed behaviour. We define the function 'NCon' to compute the causality relations that are *not* in the context of action a in behaviour B:

$$
\begin{aligned}
&\text{NCon } a\ B = \\
&\quad \{\Gamma_b \to b \mid (\Gamma_b \to b) \in B,\ a \neq b,\ \neg(\text{appearsin } a\ \Gamma_b),\ \neg(\text{appearsin } b\ \Gamma_a)\} \\
&\quad \cup \\
&\quad \{(\text{alternativeswithout } a\ \Gamma_b) \to b \mid (\Gamma_b \to b) \in B,\ a \neq b,\ (\text{appearsin } a\ \Gamma_b)\} \\
&\quad \cup \\
&\quad \{(\text{alternativeswithout } a\ \Gamma_b) \to b \mid (\Gamma_b \to b) \in B,\ a \neq b,\ (\text{appearsin } b\ \Gamma_a)\ \} \\
&\qquad \text{where } (\Gamma_a \to a) \in B
\end{aligned}
$$

$$
\begin{aligned}
&\text{alternativeswithout } a\ \Gamma_1 \vee \Gamma_2 \vee \ldots \vee \Gamma_n \\
&\quad = \vee\{\Gamma \mid \Gamma \in \{\Gamma_1, \Gamma_1, \ldots, \Gamma_n\},\ \neg(\text{appearsin } a\ \Gamma)\ \}, \\
&\qquad\quad \text{if } \exists_{\Gamma \in \{\Gamma_1, \Gamma_1, \ldots, \Gamma_n\}}\ \neg(\text{appearsin } a\ \Gamma) \\
&\quad = \surd, \text{ otherwise}
\end{aligned}
$$

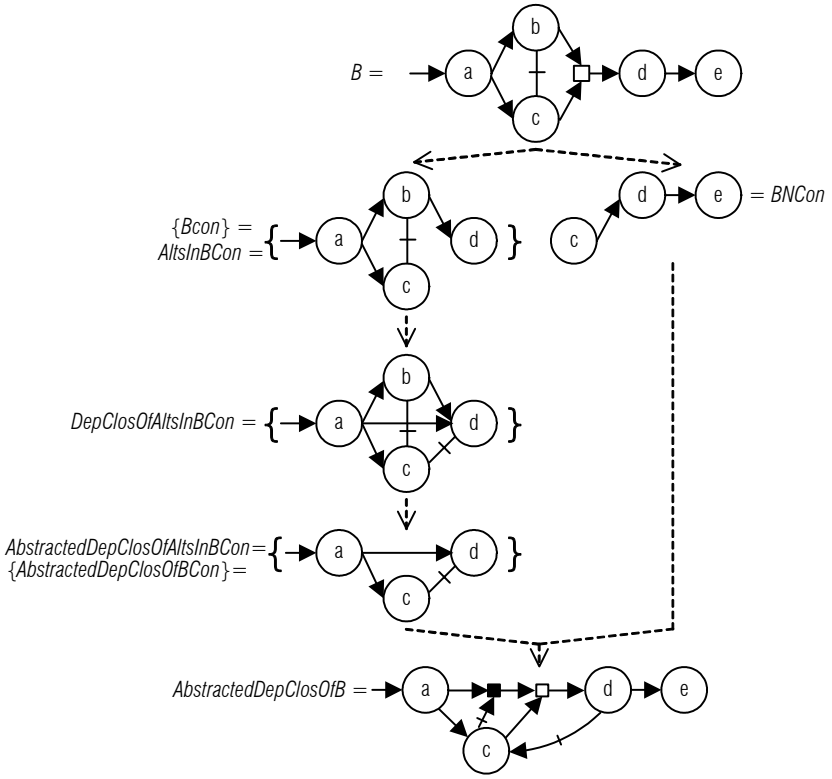
We define the function 'ICon' to integrate the (changed) causality context *BCon* of action a in behaviour B with the causality relations *BNCon* that are not in the causality context.

$$
\begin{aligned}
&\text{ICon } BCon\ BNCon\ a\ B = \\
&\quad \{\ \Gamma_a \to a \mid (\Gamma_a \to a) \in BCon,\ \neg\exists_{(\Gamma_b \to b) \in (BNCon)}\ a = b\} \\
&\quad \cup \\
&\quad \{\ \Gamma_b \to b \mid (\Gamma_b \to b) \in (BNCon),\ \neg\exists_{(\Gamma_a \to a) \in BCon}\ a = b\} \\
&\quad \cup \\
&\quad \{\ \Gamma_b \to b \mid (\Gamma_b \to b) \in (BNCon),\ (\Gamma_a \to a) \in BCon,\ a = b,\ \Gamma_a = \surd\} \\
&\quad \cup \\
&\quad \{\ \Gamma_a \vee \Gamma_b \to b \mid (\Gamma_b \to b) \in (BNCon),\ (\Gamma_a \to a) \in BCon,\ a = b,\ \Gamma_a \neq \surd\}
\end{aligned}
$$

Hence, the following algorithm can be used for the abstraction of a single inserted action a in a behaviour B (the abstraction of multiple inserted actions can be obtained by successive applications of the algorithm).

abstract *a B*:
BCon := Con *a B*
BNCon := NCon *a B*
AltsInBCon := (Alt.dnf) *BCon*
DepClosOfAltsInBCon := map ADep *AltsInBCon*
AbstractedDepClosOfAltsInBCon := map (remove *a*) *DepClosOfAltsInBCon*
AbstractedDepClosOfBCon := ° *AbstractedDepClosOfAltsInBCon*
AbstractedDepClosOfB := ICon *AbstractedDepClosOfBCon BNCon a B*
return *AbstractedDepClosOfB*

Figure 5-14 Example of Abstraction from an Action



Example 5-5 Abstraction from an Action

Figure 5-14 illustrates the abstraction from action *b* in behaviour *B*, where *B* is defined as follows:

$B = \{ \surd \rightarrow a, a \wedge \neg c \rightarrow b, a \wedge \neg b \rightarrow c, b \vee c \rightarrow d, d \rightarrow e \}$

We split up this behaviour into the causality context of *b* and the rest of the behaviour that is not the causality context of *b*:

$BCon = \text{Con}\ b\ B = \{ \surd \rightarrow a, a \wedge \neg c \rightarrow b, a \wedge \neg b \rightarrow c, b \rightarrow d \}$

BNCon = NCon *b B* = { $c \rightarrow d, d \rightarrow e$ }
Since there are no disjunctions in the causality context of *b* in *B*, there is only one behaviour in the set of alternative behaviours of *BCon*. This behaviour is equal to *BCon*. We can compute the implicit relations in the causality context of *b* in *B*, resulting in the behaviour *DepClosOfAltsInBCon*. Then we can abstract from *b* in that behaviour, resulting in:

AbstractedDepClosOfAltsInBCon =
map (remove *a*) *DepClosOfAltsInBCon* =
{ $\surd \rightarrow a, a \wedge \neg d \rightarrow c, a \wedge \neg c \rightarrow d$ }

Since there is only one alternative behaviour, the integration of alternative behaviours (*AbstractedDepClosOfBCon*) is equal to that alternative. Next, we can combine the causality context in which we abstracted from *b*, with the rest of the behaviour (*BNCon*). This results in:

AbstractedDepClosOfB =
ICon *AbstractedDepClosOfBCon BNCon b B* =
{ $\surd \rightarrow a, a \wedge \neg d \rightarrow c, (a \wedge \neg c) \vee c \rightarrow d, d \rightarrow e$ }

5.2.4 Action Integration

The action integration operator can be used to integrate final actions in a design.

There are different ways in which the completion of final actions corresponds to the completion of an abstract action. For example, the completion of *all* final actions corresponds to the completion of the abstract action, or the completion of *any* of the final actions corresponds to the completion of the abstract action. Therefore, we require the specification of a *completion condition* that represents which of the final actions must have completed, for the integrated action to complete. A completion condition can use conjunctions, represented by $\wedge$, to represent that all actions in the conjunction must have completed for the abstract action to complete. It can use disjunctions, represented by $\vee$, to represent that any of the actions in the disjunction must have completed for the abstract action to complete. And, it can use combinations of conjunctions and disjunctions. We assume that a completion condition is specified in the disjunctive normal form. An example of a completion condition is: $a_1 \vee (a_2 \wedge a_3)$. This condition represents that the completion of some abstract action corresponds to the completion of final action a_1 or the completion of final actions a_2 and a_3.

To integrate final actions, we must derive the causality condition of the integrated action. Also, we must change the causality conditions that other actions have on the final actions into conditions on the integrated action. These causality conditions depend on the completion condition of the integrated action.

Causality condition of an integrated action. To determine the causality condition of an integrated action, we distinguish three basic cases of the completion condition:

1. The completion of the integrated action corresponds to the completion of a single final action (the completion condition looks like: a). In that case the condition of the integrated action is the condition of that single final action.
2. The completion of the integrated action corresponds to the completion of a conjunction of final actions (the completion condition looks like: $a_1 \wedge a_2 \wedge \ldots \wedge a_n$). In that case the integrated action occurs when *all* final actions occur. Hence, the causality condition of the integrated action is the *conjunction* of the conditions of the final actions. To keep the condition of the integrated action in the disjunctive normal form, the condition is the disjunction of all possible conjunctions of alternative conditions of the final actions. More precisely, if $\Gamma_1 \rightarrow a_1$, $\Gamma_2 \rightarrow a_2$, …, $\Gamma_n \rightarrow a_n$ and the completion condition is $a_1 \wedge a_2 \wedge \ldots \wedge a_n$, then the causality condition of the integrated action is:

$$\vee\{\Gamma_{i1} \wedge \Gamma_{i2} \wedge \ldots \wedge \Gamma_{in} \mid \Gamma_{i1} \in (\text{alternatives } \Gamma_1), \Gamma_{i2} \in (\text{alternatives } \Gamma_2), \ldots, \Gamma_{in} \in (\text{alternatives } \Gamma_n)\}$$

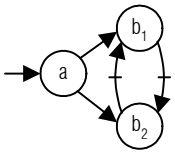


Figure 5-15 Example of an Implicit Causality Condition

3. The completion of the integrated action corresponds to the completion of a disjunction of final actions (the completion condition looks like: $a_1 \vee a_2 \vee \ldots \vee a_n$). In that case the integrated action occurs when *any* of the final actions occurs. Hence, the causality condition of the integrated action is the *disjunction* of the conditions of the final actions. From these conditions, we must remove conditions on other final actions (like final action b_1 has a condition on the non-occurrence of final action b_2 in Figure 5-15), because conditions on the occurrence of other final actions are irrelevant in the condition of the integrated action. At the level of abstraction of the integrated action, these conditions are internal to the integrated action and therefore abstracted from. More precisely, if $\Gamma_1 \rightarrow a_1$, $\Gamma_2 \rightarrow a_2$, …, $\Gamma_n \rightarrow a_n$ and the completion condition $cc = a_1 \vee a_2 \vee \ldots \vee a_n$, then the causality condition of the integrated action is:

$$\vee\{ \text{removeconditions (conactionsin } cc) \; \Gamma_i \mid \Gamma_i \in (\text{alternatives } \Gamma), \Gamma \in \{\Gamma_1, \Gamma_2, \ldots, \Gamma_n\} \}$$

Where we define 'conactionsin' (which stands for: conditions on actions in) function as follows:

$$\text{conactionsin } cc = \{ a, \neg a \mid a \in (\text{actionsin } cc) \}$$

$$\text{actionsin } A_1 \vee A_2 = (\text{actionsin } A_1) \cup (\text{actionsin } A_2)$$
$$\text{actionsin } A_1 \wedge A_2 = (\text{actionsin } A_1) \cup (\text{actionsin } A_2)$$
$$\text{actionsin } a = \{a\}$$

In general the completion condition is a disjunction of conjunctions of final actions. We define the function 'conditionfor' that determines the condition for an integrated action, based on a completion condition, as follows. For a completion condition *cc* and a behaviour *B*:

$$\text{conditionfor } (A_1 \vee A_2)\ B = (\text{conditionfor } A_1\ B) \vee (\text{conditionfor } A_2\ B)$$
$$\text{conditionfor } (A_1 \wedge A_2)\ B =$$
$$\vee\{\Gamma_i \wedge \Gamma_j \mid \Gamma_i \in (\text{alternatives } (\text{conditionfor } A_1\ B)),\ \Gamma_j \in (\text{alternatives } (\text{conditionfor } A_2\ B))\}$$
$$\text{conditionfor } a\ B =$$
$$\vee\{\ \text{removeconditions } (\text{conactionsin } cc)\ \Gamma_i \mid \Gamma_i \in (\text{alternatives } \Gamma)\}$$
$$\text{where } \Gamma \rightarrow a \in B$$

The definition of this function follows logically from the basic cases of the completion condition above.

Replacing conditions on final actions. To replace the conditions on final actions, we distinguish the same three basic cases of the completion condition as above:

1. The completion of the integrated action corresponds to the completion of a single final action (the completion condition looks like: *a*). In that case the enabling condition on the single final action must be replaced by the enabling condition on the integrated action. Similarly, the disabling condition on the single final action must be replaced by the disabling condition on the integrated action.
2. The completion of the integrated action corresponds to the completion of a conjunction of final actions (the completion condition looks like: $a_1 \wedge a_2 \wedge \ldots \wedge a_n$).

   In that case, we can replace the enabling of all final actions that appear in a condition Γ by the enabling of the integrated action, because the completion of the integrated action implies the completion of each of the final actions. We define the 'replace2a' function to perform that replacement for a completion condition $cc = a_1 \wedge a_2 \wedge \ldots \wedge a_n$ and an integrated action *a'* in a condition Γ:

$$\text{replace2a } cc\ a'\ \Gamma =$$
$$\vee\{\ a' \wedge (\text{removeconditions } (\text{actionsin } cc)\ \Gamma_i) \mid \Gamma_i \in (\text{alternatives } \Gamma), \text{partof } cc\ \Gamma_i\ \}$$
$$\vee$$
$$\vee\{\ \Gamma_i \mid \Gamma_i \in (\text{alternatives } \Gamma), \neg\text{partof } cc\ \Gamma_i\ \}$$

$$\text{partof}\ \gamma_1 \wedge \gamma_2 \wedge \ldots \wedge \gamma_n\ \Gamma = \forall_{\gamma' \in \{\gamma_1, \gamma_2, \ldots, \gamma_n\}}\ \gamma' \in (\text{targets}\ \Gamma)$$

Informally, if the completion condition is a part of an alternative condition of Γ, then we remove the completion condition from the alternative and include the integrated action. If the completion condition is not a part of an alternative condition, then that alternative remains unchanged. If some final actions are part of an alternative condition, those final actions will not be removed. This leads to failure of the consistency check, when checking the equivalence of the behaviour to the abstract behaviour (that does not contain final actions). This failure is caused by an incorrect refinement, because an action *must* be refined by *all* of the final actions.

Also, we can replace the disjunction of disablings on the final actions ($\neg a_1 \vee \neg a_2 \vee \ldots \vee \neg a_n$) by the disabling of the integrated action. We can do that, because the non-occurrence of the integrated action implies the non-occurrence of (at least) one of the final actions. Algorithmically, we perform the replacement, by trying to rewrite a condition Γ into:

$$(x_1 \wedge (\neg a_1 \vee \neg a_2 \vee \ldots \vee \neg a_n)) \vee (x_2 \wedge (\neg a_1 \vee \neg a_2 \vee \ldots \vee \neg a_n)) \vee \ldots \vee y$$

Then we can rewrite each occurrence of ($\neg a_1 \vee \neg a_2 \vee \ldots \vee \neg a_n$) by the integrated action. The function below does this by looking for an x and, if such an x can be found, rewriting the condition and looking for the next x. If such an x cannot be found, either there is no final action in the condition and therefore no replacements are necessary, or not all final actions appear in the condition. In the latter case, no replacements can be made. Hence, some final actions will remain in the condition. This leads to failure of the consistency check, when checking the equivalence of the behaviour to the abstract behaviour (that does not contain final actions). This failure is caused by an incorrect refinement, because a condition on the non-occurrence of an action *must* be refined by a condition on the non-occurrence of *any* of the final actions.

We define the function 'replace2b' to perform that replacement for a completion condition $cc = a_1 \wedge a_2 \wedge \ldots \wedge a_n$ and an integrated action a' in a condition Γ (that is not the condition of the integrated action). The function is defined in such a way that it is performed for one x that satisfies the conditions above. If such an x can be found, the replacement is performed for that x. Then the function recursively instantiates itself to search for another x that meets the conditions.

replace2b *cc a'* Γ =
  if we can construct a condition *x* for which
    $\forall_{a \in (\text{actionsin } cc)}(\exists_{\Gamma' \in (\text{alternatives } \Gamma)} \text{equivalentalternatives } \Gamma' (\neg a \wedge x))$
  then, for that *x*
    $\Gamma' := \vee\{\neg a' \wedge (\text{removeconditions } \neg a\ \Gamma'') \mid \Gamma'' \in (\text{alternatives } \Gamma),$
      $a \in (\text{actionsin } cc), \text{equivalentalternatives } \Gamma'' (\neg a \wedge x)\}$
    $\vee$
    $\vee\{\ \Gamma'' \mid \Gamma'' \in (\text{alternatives } \Gamma),$
      $\neg\exists_{a \in (\text{actionsin } cc)} \text{equivalentalternatives } \Gamma'' (\neg a \wedge x)\}$
    return replace2b *cc a'* Γ' =
  otherwise return Γ

3. The completion of the integrated action corresponds to the completion of a disjunction of final actions (the completion condition looks like: $a_1 \vee a_2 \vee \ldots \vee a_n$). The approach to replace final actions in this case is similar to the approach to replace final actions in case the completion condition is a conjunction of final actions. We can replace the disabling of all of the final actions by the disabling of the integrated action, because the non-occurrence of the integrated action implies the non-occurrence of all of the final actions. Also, we can replace the disjunction of enablings of the final actions by the enabling of the integrated action, because the occurrence of the integrated action implies the occurrence of one (or more) of the final actions. These replacements are covered by the functions 'replace3a' and 'replace3b', respectively. Hence, for a completion condition $cc = a_1 \vee a_2 \vee \ldots \vee a_n$, an integrated action *a'* and a condition Γ (that is not the condition of the integrated action):

replace3a *cc a'* Γ =
  $\vee\{\neg a' \wedge (\text{removeconditions } \neg(\text{actionsin } cc)\ \Gamma_i) \mid$
    $\Gamma_i \in (\text{alternatives } \Gamma), \forall_{a \in (\text{actionsin } cc)} \text{partof } \neg a\ \Gamma_i\ \}$
  $\vee$
  $\vee\{\ \Gamma_i \mid \Gamma_i \in (\text{alternatives } \Gamma), \neg\forall_{a \in (\text{actionsin } cc)} \text{partof } \neg a\ \Gamma_i\ \}$

replace3b *cc a'* Γ =
  if we can construct a condition *x* for which
    $\forall_{a \in (\text{actionsin } cc)} (\exists_{\Gamma' \in (\text{alternatives } \Gamma)} \text{equivalentalternatives } \Gamma' (a \wedge x))$
  then, for that *x*
    $\Gamma' := \vee\{a' \wedge (\text{removeconditions } a\ \Gamma'') \mid \Gamma'' \in (\text{alternatives } \Gamma),$
      $a \in (\text{actionsin } cc), \text{equivalentalternatives } \Gamma'' (a \wedge x)\}$
      $\vee$
      $\vee\{\ \Gamma'' \mid \Gamma'' \in (\text{alternatives } \Gamma),$
      $\neg\exists_{a \in (\text{actionsin } cc)} \text{equivalentalternatives } \Gamma'' (a \wedge x)\}$
    return replace3b *cc a'* Γ'
  otherwise return Γ

In general the completion condition (in the disjunctive normal form) is a disjunction of conjunctions of final actions. Therefore, we must combine the functions above, which either cover the disjunction *or* the conjunction of final actions, but not the general case. We define the functions 'replaceen' and 'replacedis'. 'replaceen' replaces a conjunction of enabling conditions on final actions or disabling conditions on final actions by an integrated action. 'replacedis' replaces a disjunction of enabling conditions on final actions or disabling conditions on final actions by an integrated action. For a completion condition *cc* and an integrated action *a*' in a condition Γ (that is not the condition of the integrated action):

replaceen *cc a*' Γ =
  if we can construct a condition *x* for which
    $\forall_{acc \in (\text{altccs } cc)}\ (\exists_{\Gamma' \in (\text{alternatives } \Gamma)}$ eqtoaltcc Γ' *x acc*)
  then, for that *x*
    Γ' := $\vee\{$ *a*' $\wedge$ (removeconditions *acc* Γ") |
        Γ" $\in$(alternatives Γ),
        acc $\in$(altccs *cc*), eqtoaltcc Γ" *x acc*$\}$
      $\vee$
      $\vee\{$ Γ" | Γ" $\in$(alternatives Γ),
        $\neg\exists_{acc \in (\text{altccs } cc)}$ eqtoaltcc Γ" *x acc*$\}$
    return replace *cc a*' Γ' =
  otherwise return Γ

replacedis *cc a*' Γ =
  if we can construct a condition *x* for which
    $\forall_{adcc \in (\text{altdccs } cc)}\ (\exists_{\Gamma' \in (\text{alternatives } \Gamma)}$ eqtoaltcc Γ' *x adcc*)
  then, for that *x*
    Γ' := $\vee\{$ $\neg$*a*' $\wedge$ (removeconditions $\neg$*adcc* Γ") |
        Γ" $\in$(alternatives Γ),
        adcc $\in$(altdccs *cc*), eqtoaltcc Γ" *x adcc*$\}$
      $\vee$
      $\vee\{$ Γ" | Γ" $\in$(alternatives Γ),
        $\neg\exists_{adcc \in (\text{altdccs } cc)}$ eqtoaltcc Γ" *x adcc*$\}$
    return replace *cc a*' Γ' =
  otherwise return Γ

Where we define the function 'altccs' (which stands for: alternative completion conditions) that returns, for a completion condition, the set of conjunctions of enabling conditions that can be replaced by an integrated action:

altccs $A_1 \vee A_2$ = (altccs A_1) $\cup$ (altccs A_2)
altccs A = $\{A\}$

We define the function 'altdccs' to do the same for the conjunctions of disabling conditions:

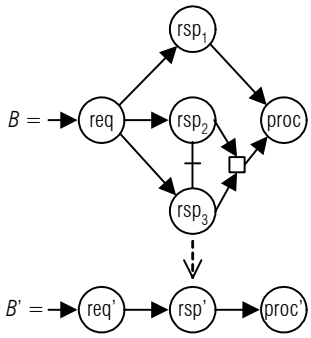
$$\text{altdccs } A = (\text{altccs } (\text{dnf } (\text{helper } A)))$$

$$\begin{aligned} \text{helper } A_1 \vee A_2 &= (\text{helper } A_1) \wedge (\text{helper } A_2) \\ \text{helper } A_1 \wedge A_2 &= (\text{helper } A_1) \vee (\text{helper } A_2) \\ \text{helper } a &= \neg a \end{aligned}$$

We define the function 'eqtoaltcc' (which stands for: equivalent to alternative completion condition). This function returns true if and only if an alternative condition Γ is equivalent to x or the conjunction of x with a or $\neg a$:

$$\begin{aligned} &\text{eqtoaltcc } \Gamma\ x\ a = \\ &\quad \text{equivalentalternatives } (\text{removeconditions } (\text{basicconditions } a)\ \Gamma)\ x \\ &\quad \wedge \\ &\quad \text{partof } (\text{removeconditions } (\text{basicconditions } x)\ \Gamma)\ a \end{aligned}$$

Figure 5-16 Example of Integration of Final Actions



Example 5-6 Integration of Final Actions

As an example of integrating final actions, consider Figure 5-16. This figure shows two behaviours *B* and *B*'. In *B* a request (*req*) is sent, after which a response rsp_1 and a choice between response rsp_2 and rsp_3 is returned. Finally, the response is processed (*proc*). We integrate the different responses (rsp_1, rsp_2 and rsp_3) into a single response (*rsp*'), using the completion condition $(rsp_1 \wedge rsp_2) \vee (rsp_1 \wedge rsp_3)$. This yields *B*'.
The causality condition Γ for *rsp*' can be determined as follows, using the formula 'conditionfor' that is defined for that purpose.

$$\begin{aligned} \Gamma &= \text{conditionfor } ((rsp_1 \wedge rsp_2) \vee (rsp_1 \wedge rsp_3))\ B \\ &= (\text{conditionfor } (rsp_1 \wedge rsp_2)\ B) \vee (\text{conditionfor } (rsp_1 \wedge rsp_3)\ B)\ B \\ &= \vee\{\Gamma_1 \wedge \Gamma_2 \mid \Gamma_1 \in (\text{alternatives.conditionfor } rsp_1\ B), \Gamma_2 \in (\text{alternatives.conditionfor } rsp_2\ B)\} \\ &\quad \vee \\ &\quad \vee\{\Gamma_1 \wedge \Gamma_2 \mid \Gamma_1 \in (\text{alternatives.conditionfor } rsp_1\ B), \Gamma_2 \in (\text{alternatives.conditionfor } rsp_3\ B)\} \\ &= \vee\{\Gamma_1 \wedge \Gamma_2 \mid \Gamma_1 \in \{req\}, \Gamma_2 \in \{req\}\} \vee \vee\{\Gamma_1 \wedge \Gamma_2 \mid \Gamma_1 \in \{req\}, \Gamma_2 \in \{req\}\} \end{aligned}$$

$= (req \wedge req) \vee (req \wedge req)$
$= req$

Only the causality condition of *proc* depends on final actions. We replace the final actions in the causality condition of *proc*', which is the single final action for *proc* in *B*', using the formula 'replaceen' that is defined for that purpose.
replaceen $((rsp_1 \wedge rsp_2) \vee (rsp_1 \wedge rsp_3))$ *rsp*' $((rsp_1 \wedge rsp_2) \vee (rsp_1 \wedge rsp_3))$:
If we choose $x = \surd$, then:
$\forall_{acc \in \{rsp1 \wedge rsp2,\ rsp1 \wedge rsp3\}}$ $(\exists_{\Gamma \in \{rsp1 \wedge rsp2,\ rsp1 \wedge rsp3\}}$ eqtoaltcc Γ *x acc*)
For example:
eqtoaltcc $rsp_1 \wedge rsp_2$ $\surd$ $rsp_1 \wedge rsp_2$
$=$ equivalentalternatives(removeconditions (basicconditions $rsp_1 \wedge rsp_2$) $rsp_1 \wedge rsp_2$) $\surd$
$\wedge$
partof (removeconditions (basicconditions $\surd$) $rsp_1 \wedge rsp_2$) $rsp_1 \wedge rsp_2$
$=$ equivalentalternatives(removeconditions $\{rsp_1, rsp_2\}$ $rsp_1 \wedge rsp_2$) $\surd$
$\wedge$
partof (removeconditions $\{\surd\}$ $rsp_1 \wedge rsp_2$) $rsp_1 \wedge rsp_2$
$=$ equivalentalternatives $\surd$ $\surd$ $\wedge$ partof $rsp_1 \wedge rsp_2$ $rsp_1 \wedge rsp_2$
$=$ True
Therefore perform:
Γ' := $\vee\{$ *rsp*' $\wedge$ (removeconditions $\{rsp_1, rsp_2, rsp_3\}$ Γ |
$\Gamma \in \{rsp_1 \wedge rsp_2,\ rsp_1 \wedge rsp_3\}$,
$\exists_{acc \in \{rsp1 \wedge rsp2,\ rsp1 \wedge rsp3\}}$ eqtoaltcc Γ *x acc*$\}$
$\vee$
$\vee\{\Gamma \mid \Gamma \in \{rsp_1 \wedge rsp_2,\ rsp_1 \wedge rsp_3\}, \neg\exists_{acc \in \{rsp1 \wedge rsp2, rsp1 \wedge rsp3\}}$ eqtoaltcc Γ *x acc*$\}$
$=$ $\vee\{$ *rsp*' $\wedge$ (removeconditions $\{rsp_1, rsp_2, rsp_3\}$ $\{rsp_1 \wedge rsp_2\}$,
rsp' $\wedge$ (removeconditions $\{rsp_1, rsp_2, rsp_3\}$ $\{rsp_1 \wedge rsp_3\}\}$
$\vee$
$\vee \varnothing$
$=$ $\vee\{rsp' \wedge \surd, rsp' \wedge \surd\} \vee \vee \varnothing$
$=$ *rsp*' $\wedge$ $\surd$
return replaceen $((rsp_1 \wedge rsp_2) \vee (rsp_1 \wedge rsp_3))$ *rsp*' (*rsp*' $\wedge$ $\surd$)

Hence, we have to perform another iteration of 'replaceen' over the newly defined causality (*rsp*' $\wedge$ $\surd$) of *proc*'.

replaceen $((rsp_1 \wedge rsp_2) \vee (rsp_1 \wedge rsp_3))$ *rsp*' (*rsp*' $\wedge$ $\surd$):
There is no *x* for which:
$\forall acc \in \{rsp_1 \wedge rsp_2,\ rsp_1 \wedge rsp_3\} \bullet \exists \Gamma \in \{rsp' \wedge \surd\} \bullet$ eqtoaltcc Γ *x acc*
Therefore perform:
return *rsp*' $\wedge$ $\surd$

5.2.5 Behaviour Composition

Constituents of a structured behaviour can be composed into a single behaviour, using the composition operator. We developed a composition op-

erator for certain classes of structured behaviours. This section describes the composition operator for those classes.

The class that contains non-structured, non-recursive sub-behaviours. If the behaviour instantiations that we compose are neither structured nor recursive, they do not contain declarations of other behaviour instantiations, nor recursive instantiations of themselves. This class of behaviour instantiations represents finite behaviours.

In this class of structured behaviours an interaction is instantiated and enabled if all interaction contribution instantiations that it consists of are enabled. Each interaction contribution instantiation is enabled exactly once. Hence, the condition for the occurrence of the interaction is the conjunction of the conditions of its interaction contribution instantiations. Therefore, if we compose the structured behaviour, its interactions can be replaced by action instantiations. The condition for the occurrence of such an action instantiation is the conjunction of the conditions of the interaction contribution instantiations that constituted the original interaction.

An interaction can consist of alternatives. Each alternative denotes a group of interaction contribution instantiations that, when enabled, instantiate and enable the interaction. However, each interaction contribution instantiation can occur only once. Therefore, if an interaction occurs as a consequence of an interaction contribution, other alternatives that contain that interaction contribution cannot occur anymore. Hence, action instantiations in a composition, that are derived from interaction alternatives that share interaction contribution instantiations, disable each other. Another property of alternative interactions is the following. If an action or interaction contribution instantiation is enabled or disabled by an interaction contribution instantiation that appears in multiple alternative interactions, it is enabled or disabled by each of these alternative interactions.

Therefore, we can define the composition of an interaction with alternatives as follows. For each alternative, create an action instantiation. Such an action instantiation is enabled by the conjunction of the causality conditions of the interaction contribution instantiations from which it was derived. Also, it is disabled by action instantiations created from other alternatives with which it shares one or more interaction contribution instantiations. Another causality condition can contain an enabling condition on an interaction contribution instantiation that appears in multiple alternatives. In the composite behaviour such an enabling condition must be replaced by the disjunction of enabling conditions on composite actions in which the interaction contribution appears. Similarly, if another causality condition contains a disabling condition on an interaction contribution that appears in multiple alternatives. In the composite behaviour this disabling condition

must be replaced by the conjunction of disabling conditions on composite actions in which the interaction contribution appears.

Figure 5-17 Example of Finite Behaviour Composition

Figure 5-17 and Figure 5-18 show two examples of the composition of behaviours that are not structured nor recursive. Interaction c in the first example consists of interaction contribution instantiations $b_1.c$ and $b_2.c$. Hence, the causality condition for the action that is the composition of the interaction is the conjunction of the causality condition of $b_1.c$ and the causality condition of $b_2.c$. We name the resulting action instantiation c after the interaction from which it was derived. In the textual notation, the example from Figure 5-17 looks as follows:

$B_1 = \{\surd \rightarrow a, a \rightarrow c\}$
$B_2 = \{\surd \rightarrow b, b \rightarrow c\}$
$b_1.c \wedge b_2.c \Rightarrow c$
$B = \{\surd \rightarrow a, \surd \rightarrow b, a \wedge b \rightarrow c\}$

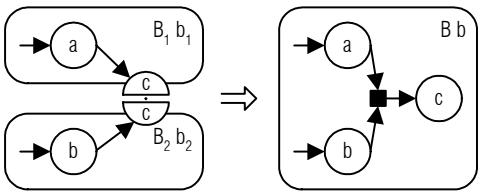
Figure 5-18 shows an interaction c that is instantiated and enabled, if $b_1.c$ and $b_2.c_1$ are enabled, or $b_1.c$ and $b_2.c_2$ are enabled. Hence, in the example, interaction c can occur as a consequence of $b_1.c$ and $b_2.c_1$, or as a consequence of $b_1.c$ and $b_2.c_2$. We compose this interaction into two actions c_1 and c_2 that represent the alternative interactions '$b_1.c$ and $b_2.c_1$' and '$b_1.c$ and $b_2.c_2$', respectively. Since the alternative interactions share the interaction contribution '$b_1.c$', one disables the other. Therefore, the actions that correspond to the alternative interactions also disable each other.

In Figure 5-18 $b_1.b$ is enabled by $b_1.c$. $b_1.c$ appears in the alternative interaction formed by $b_1.c$ and $b_2.c_1$ and the alternative interaction formed by $b_1.c$ and $b_2.c_2$. Therefore, in the composite behaviour b, action instantiation b is enabled by (the disjunction of) both actions that correspond to these alternatives. In contrast, $b_2.e$ is enabled by $b_2.c_1$, which only occurs in the alternative interaction formed by $b_1.c$ and $b_2.c_1$. Therefore, in the composite behaviour b, action instantiation e is only enabled by the instantiation action that correspond to that alternative.

The equivalent of the example from Figure 5-18 in the textual notation is as follows:

$B_1 = \{\surd \rightarrow a, a \rightarrow c, c \rightarrow b\}$
$B_2 = \{\surd \rightarrow d, d \rightarrow c_1, c_1 \rightarrow e, \surd \rightarrow f, f \rightarrow c_2, c_2 \rightarrow g\}$
$b_1.c \wedge (b_2.c_1 \vee b_2.c_2) \Rightarrow c$
$B = \{\surd \rightarrow a, \surd \rightarrow d, d \wedge a \wedge \neg c_2 \rightarrow c_1, c_1 \rightarrow e, \surd \rightarrow f, f \wedge a \wedge \neg c_1 \rightarrow c_2, c_2 \rightarrow g, c_1 \vee c_2 \rightarrow b\}$

Example 5-7 Composition of Non-Structured, Non-Recursive Behaviours



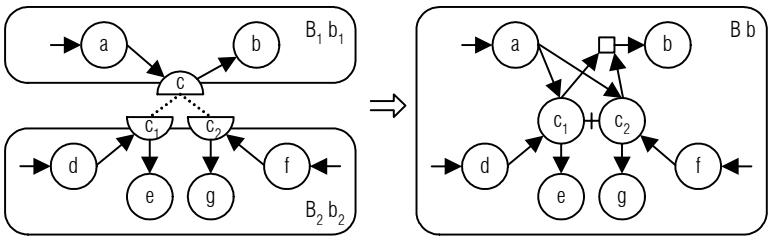


Figure 5-18 Example of Behaviour Composition with Alternative Interactions

Based on these observations, we can define the composition $\oplus$ of behaviour instantiations $\{b_1(B_1), b_2(B_2), \ldots, b_n(B_n)\}$ into a behaviour instantiation $b(B)$. $b_1(B_1), b_2(B_2), \ldots, b_n(B_n)$ have the interactions defined in the set I.

$$\oplus_I \{b_1(B_1), b_2(B_2), \ldots, b_n(B_n)\} = b(B), \text{ such that}$$
$$\forall_{i \in \{1,\ldots,n\}, (\Gamma \to t) \in Bi \mid bi.t \notin \cup\{\text{contributions } \Theta \mid (\Theta \Rightarrow int) \in I\}}$$
$$(((\text{rename.}(\text{prefixall } b_i))\ \Gamma) \to b_i.t) \in B$$
$$\forall_{(\Theta \Rightarrow int) \in I, \theta \in (\text{alternatives } \Theta)}$$
$$(\quad (\quad \wedge_{b.ic \in (\text{contributions } \theta)} (\text{rename.conditionof})\ b.ic$$
$$\wedge$$
$$\wedge_{\theta' \in (\text{alternatives } \Theta), \theta \neq \theta', (\text{contributions } \theta) \cap (\text{contributions } \theta') \neq \varnothing}$$
$$\neg(\text{contributions } \theta').int$$
$$) \to (\text{contributions } \theta).int$$
$$) \in B$$

The composite operator yields a behaviour type that is built from two rules.

The first rule applies to actions and interaction contributions that do not appear in an interaction between the composed behaviour instantiations. These actions and interaction contributions appear in the composite unchanged. We prefix each of these actions and interaction contributions, as well as actions and interactions in their conditions, with the name of the behaviour instantiation in which they were defined. In this way, names of actions or interaction contributions that appear in more than one behaviour instantiation remain unique. The 'prefixall' helper function does this. Finally, we have to change enabling and disabling conditions on interaction contributions that appear in interactions from I. We have to change these conditions into conditions on the actions that are derived from those interaction contributions. The 'rename' helper function does this.

The second rule applies to interactions in I. For each interaction alternative in I we create an action. The condition for that action is the conjunction of two parts. The first part is the conjunction of the conditions of the interaction contributions that form the alternative. These conditions have to be renamed with the 'rename' function. The second part is the conjunction of the disablings of all actions with which this action shares interaction contributions. Each action name is prefixed with the names of all interaction

contributions from which it is derived. This is to distinguish it from each action that was derived from the same interaction, but from a different alternative.

The composition operator uses the helper functions defined below. The 'contributions' helper function yields all interaction contributions in an interaction condition. This function assumes that the interaction condition is in the disjunctive normal form (although it could easily be rewritten to accept a condition in any form). The 'contributionof' function delivers the condition of the interaction contribution in a particular behaviour instantiation. It prefixes each action and interaction contribution in that condition with the name of the behaviour instantiation. The 'prefixall' function prefixes a name to each action and interaction contribution in a condition. The 'rename' function renames an enabling or disabling condition on an interaction contribution from I into a condition on the action or actions in which it appears.

contributions $\Theta_1 \wedge \Theta_2$ = (contributions Θ_1) $\cup$ (contributions Θ_2)
contributions $\Theta_1 \vee \Theta_2$ = (contributions Θ_1) $\cup$ (contributions Θ_2)
contributions $b.ic$ = $\{b.ic\}$, otherwise

conditionof $b.ic$ = prefixall b Γ
where $(\Gamma \rightarrow ic) \in B$ and
$b(B)$ is one of the composed behaviour instantiations

prefixall b $(\Gamma_1 \wedge \Gamma_2)$ = (prefixall b Γ_1) $\wedge$ (prefixall b Γ_2)
prefixall b $(\Gamma_1 \vee \Gamma_2)$ = (prefixall b Γ_1) $\vee$ (prefixall b Γ_2)
prefixall b a = $b.a$
prefixall b $\neg a$ = $\neg b.a$
prefixall b $\surd$ = $\surd$

rename $\Gamma_1 \wedge \Gamma_2$ = (rename Γ_1) $\wedge$ (rename Γ_2)
rename $\Gamma_1 \vee \Gamma_2$ = (rename Γ_1) $\vee$ (rename Γ_2)
rename $b.a$ = $b.a$, if $\neg\exists_{(\Theta \Rightarrow int) \in I,\ (b.a)' \in (\text{contributions } \Theta)}\ (b.a)' = b.a$
$= \vee_{(\Theta \Rightarrow int) \in I,\ \theta \in (\text{alternatives } \Theta),\ (b.a)' \in (\text{contributions } \theta),\ (b.a)' = b.a}$ contributions $\theta).int$, otherwise

rename $\neg b.a$
$= \neg b.a$, if $\neg\exists_{(\Theta \Rightarrow int) \in I,\ (b.a)' \in (\text{contributions } \Theta)}\ (b.a)' = b.a$
$= \wedge_{(\Theta \Rightarrow int) \in I,\ \theta \in (\text{alternatives } \Theta),\ (b.a)' \in (\text{contributions } \theta),\ (b.a)' = b.a}$ (contributions $\theta).int$, otherwise
rename $\surd$ = $\surd$

Example 5-8 Composition using the Composition Operator for Finite Behaviours

Using the composition operator as it is defined above, the composition of $b_1(B_1)$ and $b_2(B_2)$ from Figure 5-18, with interactions $I = \{ (b_1.c \wedge b_2.c_1) \vee (b_1.c \wedge b_2.c_2) \Rightarrow c \}$ is $b(B)$, such that:

$$
\begin{array}{lll}
B = \{ & \surd & \rightarrow b_1.a, \\
 & \{b_1.c, b_2.c_1\}.c \vee \{b_1.c, b_2.c_2\}.c & \rightarrow b_1.b, \\
 & \surd & \rightarrow b_2.d, \\
 & \{b_1.c, b_2.c_1\}.c & \rightarrow b_2.e, \\
 & \surd & \rightarrow b_2.f, \\
 & \{b_1.c, b_2.c_2\}.c & \rightarrow b_2.g, \\
 & b_1.a \wedge b_2.d \wedge \neg\{b_1.c, b_2.c_2\}.c & \rightarrow \{b_1.c, b_2.c_1\}.c, \\
 & b_1.a \wedge b_2.f \wedge \neg\{b_1.c, b_2.c_1\}.c & \rightarrow \{b_1.c, b_2.c_2\}.c\}
\end{array}
$$

Figure 5-18 shows the same composition. Only in Figure 5-18 $b_1.a$ is named a, $b_1.b$ is named b, $b_2.d$ is named d, $b_2.e$ is named e, $b_2.f$ is named f, $b_2.g$ is named g, $\{b_1.c, b_2.c_1\}.c$ is named c_1 and $\{b_1.c, b_2.c_2\}.c$ is named c_2.

As a consequence of composing behaviours, some of the composite actions may be impossible. This is not an error in the composition operator, but rather reflects an impossible interaction that already existed.

The class that contains non-structured, recursive sub-behaviours. To describe infinite behaviours, we can use the recursive behaviour instantiation concept as described in section 4.3. If an interaction is specified on an interaction contribution instantiation of a recursive behaviour that interaction can occur if *any* instance of that instantiation is enabled. We say *any* instance, because, from the perspective of the behaviour with which the recursive behaviour interacts, it does not matter if it interacts with the first, second or n^{th} instance of the instantiation. However, interactions that consist of different interaction contribution instances are *not* identical. Also, they may have different values for their attributes and/or different causality conditions, in which case they are not equivalent either.

The observations above lead to the conclusion that an interaction in which an interaction contribution instantiation of a recursive behaviour appears represents an interaction with a potentially infinite number of alternatives and instances. We can represent an interaction with an infinite number of alternatives, using the following notation. For any behaviour b of type B, b^i represents: b if $i = 1$, $b.b$ if $i = 2$, $b.b.b$ if $i = 3$, ... In this way b^i refers to successive instances (or instantiations) of behaviour type B by the name b^i, because instances (or instantiations) can be referred to by the name of the behaviour to which they belong 'dot' their own name. Hence, in the context of a behaviour instance b of type B, we represent the instantiation b of type B as $b.b(B)$, leading to a behaviour instance $b.b$. We repre-

sent the instantiation of b of type B in the context of behaviour instance $b.b$ as $b.b.b(B)$, leading to a behaviour instance $b.b.b$, and so on.

We can use this notation in conditions as follows. We represent the conjunction and disjunction of an infinite number of instances of action a, from successive instances b^1, b^2, b^3, … of behaviour type B, as

$$\bigwedge_{i=1}^{\infty} b^i.a \text{ and } \bigvee_{i=1}^{\infty} b^i.a \text{, respectively.}$$

We can represent (infinite) combinations of conjunctions and disjunctions in a similar fashion.

Example 5-9 Composition of Infinite Behaviours

Figure 5-19.i and ii illustrate how an interaction a can occur between any instance $b_1^i.a$ of B_1 and any instance $b_2^j.a$ of B_2. If we use the notation outlined above, we can represent the interaction:

$(b_1^1.a \vee b_1^2.a \vee b_1^3.a \vee \ldots) \wedge (b_2^1.a \vee b_2^2.a \vee b_2^3.a \vee \ldots) \Rightarrow a$
as:
$$(\bigvee_{i=1}^{\infty} b_1^i.a) \wedge (\bigvee_{j=1}^{\infty} b_2^j.a) \Rightarrow a$$

Figure 5-19.iii shows how we can specify the composite of the structured behaviour type that contains instantiations b_1 and b_2 as an infinite behaviour type. If behaviour types B_1 and B_2 are specified as:

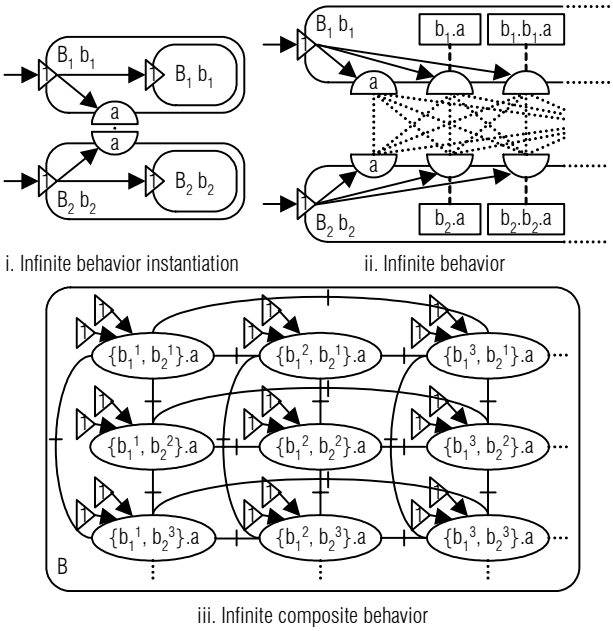
$$B_1 = \{ b_1^i.>1 \rightarrow a, b_1^i.>1 \rightarrow b_1^{i+1}(B_1).>1 \mid i > 0\}$$
$$B_2 = \{ b_2^i.>1 \rightarrow a, b_2^i.>1 \rightarrow b_2^{i+1}(B_2).>1 \mid i > 0\}$$

The textual equivalent of the composed behaviour type, using the notation outlined above, is, for $i, j > 0$:

$$B = \{b_1^1. > 1 \wedge b_2^1. > 1 \wedge (\bigwedge_{\substack{k=1 \\ k \neq j}}^{\infty} \neg\{b_1^i, b_2^k\}.a) \wedge (\bigwedge_{\substack{k=1 \\ k \neq i}}^{\infty} \neg\{b_1^k, b_2^j\}.a) \rightarrow \{b_1^i, b_2^j\}.a\}$$

Each action has a name that allows it to be identified as a composition of an interaction contribution of a particular instance of b_1 and an interaction contribution of a particular instance of b_2. For example, $\{b_1^2, b_2^3\}.a$ is the action a that is the composite of interaction contributions $b_1^2.a$ and $b_2^3.a$. Each action has two entry point conditions, because both behaviours from which it was derived assign it an entry point condition. The entry points are not named and not attached to the border of the behaviour to simplify the drawing. Each action $\{b_1^i, b_2^j\}.a$ is disabled by all actions $\{b_1^i, b_2^k\}.a$ $(k \neq j)$, because these actions share the same interaction contribution b_1^i (and an interaction contribution can only occur once). Similarly, each action $\{b_1^i, b_2^j\}.a$ is disabled by all actions $\{b_1^k, b_2^j\}.a$ $(k \neq i)$.

Figure 5-19 Composition of Infinite Behaviours

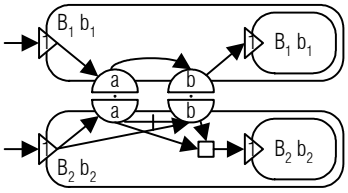


i. Infinite behavior instantiation ii. Infinite behavior

iii. Infinite composite behavior

Specifying infinite conditions in this way is not allowed by the syntax from chapter 4, we merely present it here to explain what we do when we compose infinite behaviours. The syntax from chapter 4 only allows for the specification of infinite behaviours using the recursive behaviour instantiation concept. Infinite conditions can then be represented by exit points of these behaviours. Therefore, we can only specify the composition of two or more infinite behaviours, if it can be expressed as a recursive behaviour or a finite behaviour.

Figure 5-20 Example of Interacting Repeating Behaviours



Example 5-10 Interacting Repeating Behaviours

Figure 5-20 shows an example of two repeating behaviours. Behaviour b_1 is repeating, because it can recursively instantiate itself after both interaction contribution *a* and interaction contribution *b* have occurred. Behaviour b_2 is repeating, because it can only recursively instantiate itself either after *a* has occurred and *b* is disabled or after *b* has occurred and *a* is disabled. The textual representation of the example is as follows:

$B_1 = \{>1 \rightarrow a, a \rightarrow b, b \rightarrow b_1(B_1).>1\}$
$B_2 = \{>1 \wedge \neg b \rightarrow a, >1 \wedge \neg a \rightarrow b, a \vee b \rightarrow b_2(B_2).>1\}$
$b_1.a \wedge b_2.a \Rightarrow a$
$b_1.b \wedge b_2.b \Rightarrow b$

The class that contains non-structured, repeating sub-behaviours. We present an algorithm to compose two or more *repeating* behaviour instantiations into a behaviour instantiation that can be represented as a recursive behaviour. Such a behaviour *can* be represented using the syntax from chapter 4. A repeating behaviour is a behaviour that can only recursively instantiate itself after all its actions and interaction contributions either have occurred or cannot occur anymore. Such a behaviour is a repeating behaviour, because it completes before it instantiates itself (repeats).

We can compose two or more repeating behaviours b_1, b_2, …, if, after i_1 recursive instantiations of b_1, i_2 recursive instantiations of b_2, …, they all have an entry point enabled that allows them to recursively instantiate. If this situation occurs, all behaviours have completed, because, by definition, all their actions and interaction contributions either have occurred or cannot occur anymore. Therefore, if we recursively instantiate the behaviours from that point, their joint behaviour is equivalent to their behaviour before that point. Hence, we can derive the composite behaviour as the composite behaviour after i_1 recursive instantiations of b_1, i_2 recursive instantiations of b_2, …, that recursively instantiates itself after that.

The algorithm for composing repeating behaviours follows from that claim. We recursively instantiate the individual behaviours b_1, b_2, … that *can* be instantiated recursively, until the point at which they can *all* be instantiated recursively. This results in i_1 instances of b_1, i_2 instances of b_2, …. The composition of the repeating behaviours then is the composition of those i_1 instances of b_1, i_2 instances of b_2, …. The conditions for the causality targets in the next instance of the composition are equivalent to the conditions for the causality targets in the i_1+1^{st} instance of b_1, the i_2+1^{st} instance of b_2, …. Therefore, we can use the same entry-points and point conditions on the composed behaviour as on the original behaviours.

These observations give rise to the definition of a 'substitution' operator (σ). The substitution operator substitutes a recursive behaviour instantiation by the instantiated behaviour itself. Note that, in the substitution of a behaviour instantiation, the action and interaction contribution instantiations of that behaviour must be pre-fixed with the name of the instantiation. To simplify the resulting behaviour, the entry points of the recursive instantiation can be replaced by the conditions that they represent. We define the substitution operator σ on an instantiation of type B' in behaviour instantiation b of type B as follows:

$$\sigma\ B'\ b(B) = b(B'')$$
$$\text{where } B'' = ((\text{replaceentries } b'(B')\ B).(\text{prefix } b'))\ B' \cup \text{removeentries } b'(B')\ B$$

b' is the name of the instantiation of B' in B

The substitution operator states that the substitution of the recursive instantiation b' in $b(B)$ is the behaviour instantiation $b(B")$. $B"$ is the union of the causality relations in B' and the causality relations in B. Each action, interaction contribution and entry point in B' is prefixed with b' to distinguish it from similar elements in B. Subsequently, each entry point of B' is replaced by the condition that it represents. Consequently, the entry points of B' in B can be removed.

To this end, the substitution operator uses the following helper functions. The 'prefix' function prefixes all actions, interaction contributions and entry points in B' with b'. It uses the 'prefixall' function that we defined earlier. This function has to be modified, because it originally did not consider entry points. The 'removeentries' function removes the entries of $b'(B')$ in B. The 'replaceentries' function replaces entry point conditions in $B"$ by the conditions that the entry points represent. The latter conditions are defined in B as the conditions of entry points of $b'(B')$.

$$\text{prefix } b\ B = \{(\text{prefixall } b\ \Gamma) \rightarrow b.t \mid (\Gamma \rightarrow t) \in B\}$$
$$\text{prefixall } b\ (\Gamma_1 \wedge \Gamma_2) = (\text{prefixall } b\ \Gamma_1) \wedge (\text{prefixall } b\ \Gamma_2)$$
$$\text{prefixall } b\ (\Gamma_1 \vee \Gamma_2) = (\text{prefixall } b\ \Gamma_1) \vee (\text{prefixall } b\ \Gamma_2)$$
$$\text{prefixall } b\ a = b.a$$
$$\text{prefixall } b\ \neg a = \neg b.a$$
$$\text{prefixall } b\ \surd = \surd$$
$$\text{prefixall } b\ >e = >b.e$$

$$\text{removeentries } b'(B')\ B = \{\Gamma \rightarrow t \mid (\Gamma \rightarrow t) \in B, t \neq b'(B').>e\}$$

$$\text{replaceentries } b'(B')\ B\ B" = \{\ (\text{re } \Gamma\ B\ b'(B')) \rightarrow t \mid (\Gamma \rightarrow t) \in B"\}$$

$$\text{re } (\Gamma_1 \wedge \Gamma_2)\ B\ b'(B') = (\text{re } \Gamma_1\ B\ b'(B')) \wedge (\text{re } \Gamma_2\ B\ b'(B'))$$
$$\text{re } (\Gamma_1 \vee \Gamma_2)\ B\ b'(B') = (\text{re } \Gamma_1\ B\ b'(B')) \vee (\text{re } \Gamma_2\ B\ b'(B'))$$
$$\text{re } >b'.e\ B\ b'(B') = \Gamma, \text{ where } (\Gamma \rightarrow b'(B').>e) \in B$$
$$\text{re } a\ B\ b'(B') = a, \text{ otherwise}$$

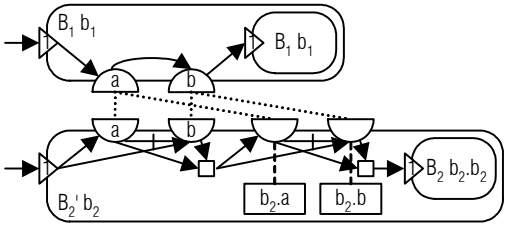


Figure 5-21 Interacting Repeating Behaviours after One Substitution of b_2

Example 5-11 Substitution of Behaviour Instantiations

Figure 5-21 shows the result of the substitution of $b_2(B_2)$ in B_2 from Figure 5-20. It shows how the interaction contributions in the recursive instantiation are prefixed with the name of the recursive instantiation (b_2) and how the entry point of the recursive instantiation ($b_2(B_2).>1$) is replaced by its condition ($a \vee b$). We can calculate this substitution, using the formulae above, as follows:

$$\sigma\, B_2\, b_2(B_2) = b_2(B_2''), \text{ where}$$
$$B_2'' = ((\text{replaceentries } b_2(B_2)\ B_2).(\text{prefix } b_2))\ B_2 \cup \text{removeentries } b_2(B_2)\ B_2$$

$$\text{prefix } b_2\ B_2 = \{\ b_2.>1 \wedge \neg b_2.b \rightarrow b_2.a,\ b_2.>1 \wedge \neg b_2.a \rightarrow b_2.b,\ b_2.a \vee b_2.b \rightarrow b_2.b_2(B_2).>1\}$$

$$\text{replaceentries } b_2(B_2)\ B_2\ (\text{prefix } b_2\ B_2) = \{\ (a \vee b) \wedge \neg b_2.b \rightarrow b_2.a,\ (a \vee b) \wedge \neg b_2.a \rightarrow b_2.b,\ b_2.a \vee b_2.b \rightarrow b_2.b_2(B_2).>1\}$$

$$\text{removeentries } b_2(B_2)\ B_2 = \{\ >1 \wedge \neg b \rightarrow a,\ >1 \wedge \neg a \rightarrow b\}$$

$$B_2'' = \{(a \vee b) \wedge \neg b_2.b \rightarrow b_2.a,\ (a \vee b) \wedge \neg b_2.a \rightarrow b_2.b,\ b_2.a \vee b_2.b \rightarrow b_2.b_2(B_2).>1,\ >1 \wedge \neg b \rightarrow a,\ >1 \wedge \neg a \rightarrow b\}$$

To define the composition operator for repeating behaviours, we use the composition operator for finite behaviours ($\oplus$). However, since the composition operator for finite behaviours did not consider entry points, we have to extend that operator. The redefined operator prefixes the names of the entry points with the names of the behaviours to which they originally belonged. In this way, the names of the entry points remain unique in the composite behaviour. We redefine the operator as follows:

$$\oplus_I \{b_1(B_1), b_2(B_2), \ldots, b_n(B_n)\} = b(B), \text{ such that}$$
$$\forall_{i \in \{1,\ldots,n\}, (\Gamma \rightarrow t) \in Bi \mid bi.t \notin \cup\{\text{contributions } \Theta \mid (\Theta \Rightarrow int) \in I\}}$$
$$t = b'(B').>e \Rightarrow (((\text{rename.}(\text{prefixall } b_i))\ \Gamma) \rightarrow b'(B_i).>b_i.e) \in B$$
$$t = a \Rightarrow (((\text{rename.}(\text{prefixall } b_i))\ \Gamma) \rightarrow b_i.a) \in B$$
$$\forall_{(\Theta \Rightarrow int) \in I,\ \theta \in (\text{alternatives } \Theta)}$$
$$(\ (\ \wedge_{b.ic \in (\text{contributions } \theta)} (\text{rename.conditionof})\ b.ic \wedge \wedge_{\theta' \in (\text{alternatives } \Theta),\ \theta \neq \theta',\ (\text{contributions } \theta) \cap (\text{contributions } \theta') \neq \varnothing} \neg(\text{contributions } \theta').int\ ) \rightarrow (\text{contributions } \theta).int\ ) \in B$$

Using the substitution operator and the redefined composition operator for finite behaviours, we define the composition operator ($\otimes$) for *repeating* behaviour instantiations $\{b_1(B_1), b_2(B_2), \ldots, b_n(B_n)\}$ that have interactions I as follows.

$$\otimes_I \{b_1(B_1), b_2(B_2), \ldots, b_n(B_n)\} = \otimes\text{Helper}_I \{(B_1, b_1(B_1)), (B_2, b_2(B_2)), \ldots, (B_n, b_n(B_n))\}$$

$\otimes\text{Helper}_I \{(B_1', b_1(B_1)), (B_2', b_2(B_2)), \ldots, (B_n', b_n(B_n))\} = b(B)$, such that

if $\forall_{i \in \{1, \ldots, n\}} (\neg\forall_{(\Gamma \to b'(Bi').>e) \in B"} \Gamma = \dagger)$

or

$\forall_{i \in \{1, \ldots, n\}} (\forall_{(\Gamma \to b'(Bi').>e) \in B"} \Gamma = \dagger)$

then

$b(B) = b"(\text{replacerecursions } \{B_1', B_2', \ldots, B_n'\}\ b"(B"))$

otherwise

$b(B) = \otimes\text{Helper}_I \{(B_1', b_1(B_1")), (B_2', b_2(B_2")), \ldots, (B_n', b_n(B_n"))\}$

where

$b"(B") = \oplus_{I'} \{b_1(B_1), b_2(B_2), \ldots, b_n(B_n)\}$

$b_i(B_i") = \sigma B_i' b_i(B_i)$, if $\neg\forall_{(\Gamma \to b'(Bi').>e) \in B"} \Gamma = \dagger$

$= b_i(B_i)$, otherwise

$I' = \text{map (explode } b_i.b') I$, for each $b'(B_i')$ for which $\neg\forall_{(\Gamma \to b'(Bi').>e) \in B"} \Gamma = \dagger$

$\text{replacerecursions } \{B_1', B_2', \ldots, B_n'\}\ b"(B") =$

$\{\Gamma \to t \mid (\Gamma \to t) \in B", t \neq b'(B').>e, B' \in \{B_1', B_2', \ldots, B_n'\}\}$

$\cup$

$\{\Gamma \to b"(B").>e \mid (\Gamma \to t) \in B", t = b'(B').>e, B' \in \{B_1', B_2', \ldots, B_n'\}\}$

$\text{explode } b.b' (\Theta \Rightarrow int) = (\text{ex } b.b' \Theta) \Rightarrow int$

$\text{ex } b.b' (\Theta_1 \wedge \Theta_2) = (\text{ex } b.b' \Theta_1) \wedge (\text{ex } b.b' \Theta_2)$

$\text{ex } b.b' (\Theta_1 \vee \Theta_2) = (\text{ex } b.b' \Theta_1) \vee (\text{ex } b.b' \Theta_2)$

$\text{ex } b.b' b".ic = b".ic \vee b.b'.ic$, if $b.b' = b".x$

$= b".ic$, otherwise

where x is a variable that does not contain a dot (.)

This composition operator for repeating behaviours is defined recursively. Textually, it reads as follows:

if, in the composition ($\oplus$) of behaviours

all behaviours repeat

or

none repeat,

then
    the result is the regular composition.
Otherwise,
    the result is the composition ($\otimes$) of the behaviours
where
    the behaviours that *can* recursively instantiate are substituted.

The repeated composition 'explodes' the original interaction contributions, because the substituted interaction contributions now also participate in the interactions. Hence, each interaction contribution of which another instance is created during the substitution of its behaviour, must be replaced by the disjunction of itself and the newly created instance. For example, during the substitution of behaviour $b_2(B_2)$, which results in the behaviour from Figure 5-21, a new instance $b_2.b_2.a$ is created of $b_2.a$. Hence, the interaction $b_1.a \wedge b_2.a \Rightarrow a$ must be replaced by $b_1.a \wedge (b_2.a \vee b_2.b_2.a) \Rightarrow a$.

Example 5-12 Composition of Repeating Behaviours

As an example of using the composition operator for repeating behaviours, we compose the repeating behaviours from Figure 5-20, resulting in the behaviour that is shown in Figure 5-22.i. The textual representation of this composite behaviour is:

$$\begin{aligned} B = \{\ & b_1.{>}1 \wedge b_2.{>}1 \wedge \neg\{b_1, b_2\}.b \rightarrow \{b_1, b_2\}.a, \\ & \{b_1, b_2\}.a \wedge \neg\{b_1, b_2\}.a \wedge b_2.{>}1 \rightarrow \{b_1, b_2\}.b, \\ & \{b_1, b_2\}.b \rightarrow b(B).b_1.{>}1, \\ & \{b_1, b_2\}.a \vee \{b_1, b_2\}.b \rightarrow b(B).b_2.{>}1\} \end{aligned}$$

Only behaviour b_2 can instantiate itself recursively in this composition. This can intuitively be seen, if we observe that a single instantiation of b_1 must perform a and then b before it can instantiate recursively, while a single instantiation of b_2 can only perform either a or b. Hence, the composite behaviour of one instance of b_1 and one instance of b_2 can only perform a, after which b_2 can instantiate recursively, but b_1 must still perform b before it can instantiate recursively. The following calculation shows how we can compute that b_1 cannot recursively instantiate in the composition of one instance of b_1 and one instance of b_2:

$$\{b_1, b_2\}.a \wedge \neg\{b_1, b_2\}.a \wedge b_2.{>}1 \rightarrow \{b_1, b_2\}.b =_{10,11} \dagger \rightarrow \{b_1, b_2\}.b$$

$$\{b_1, b_2\}.b \rightarrow b(B).b_1.{>}1 =_{14} \dagger \rightarrow b(B).b_1.{>}1$$

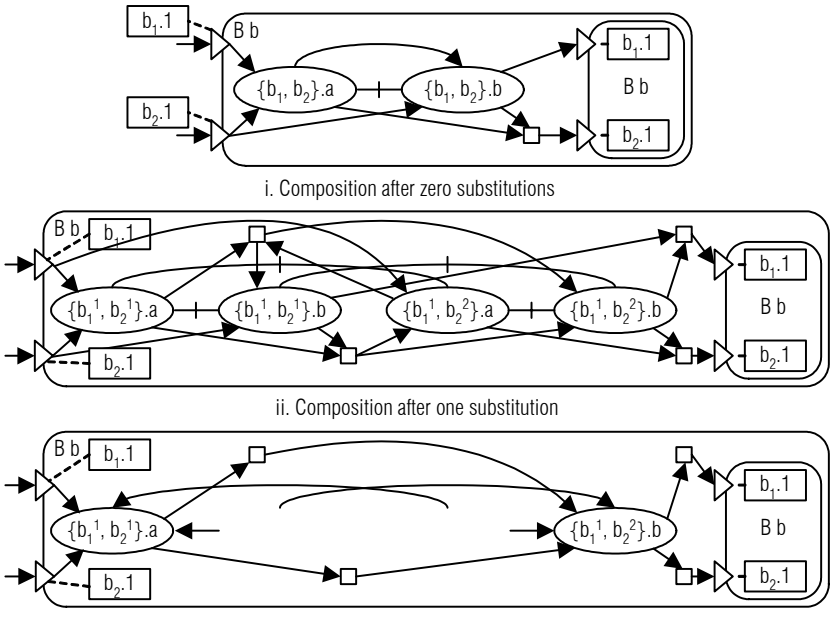
Since, in the example, b_2 recursively instantiates itself, but b_1 does not, we substitute instantiation b_2 in B_2. Then we calculate the composite behaviour of one instance of b_1 and two instances of b_2 to see if both behaviours repeat in the result. Figure 5-21 shows the result of substituting instantiation b_2 in B_2 and Figure 5-22.ii shows the composition of the result. The textual representation of this composition is:

$$B = \{\ b_1.{>}1 \wedge b_2.{>}1 \wedge \neg\{b_1^1, b_2^2\}.a \wedge \neg\{b_1^1, b_2^1\}.b \rightarrow \{b_1^1, b_2^1\}.a,$$

$b_1.>1 \wedge \neg\{b_1^1,b_2^1\}.a \wedge (\{b_1^1,b_2^1\}.a \vee \{b_1^1,b_2^2\}.a) \wedge \neg\{b_1^1,b_2^2\}.b \rightarrow \{b_1^1,b_2^1\}.b,$
$b_2.>1 \wedge (\{b_1^1,b_2^1\}.a \vee \{b_1^1,b_2^1\}.b) \wedge \neg\{b_1^1,b_2^1\}.a) \wedge \neg\{b_1^1,b_2^2\}.b \rightarrow \{b_1^1,b_2^2\}.a,$
$(\{b_1^1, b_2^1\}.a \vee \{b_1^1, b_2^2\}.a) \wedge (\{b_1^1, b_2^1\}.a \vee \{b_1^1, b_2^1\}.b)$
$\wedge \neg\{b_1^1,b_2^2\}.a) \wedge \neg\{b_1^1,b_2^1\}.b \rightarrow \{b_1^1,b_2^2\}.b,$
$\{b_1^1,b_2^1\}.b \vee \{b_1^1,b_2^2\}.b \rightarrow b(B).b_1.>1,$
$\{b_1^1,b_2^2\}.a \vee \{b_1^1,b_2^2\}.b \rightarrow b(B).b_2.>1\}$

After simplification of the behaviours, using the rules for calculating with impossibility, it looks as shown in Figure 5-22.iii. This figure shows that the recursive instantiation of both the original behaviours is possible. Therefore, this figure represents the final result. This figure also shows that, in the compositions, the actions named $\{b_1^1, b_2^1\}.b$ and $\{b_1^1, b_2^2\}.a$ are impossible. Intuitively, this can be seen, because these actions represent the composition of $b_1^1.b$ and $b_2^1.b$ and the composition of $b_1^1.a$ and $b_2^2.a$, respectively. Since in the first and only instance of b_1 (b_1^1) interaction contribution *a* must occur and then interaction contribution *b*, *a* must occur in the first instance of b_2 (b_2^1) and *b* must occur in the second instance of b_2 (b_2^2). Therefore, because *a* and *b* disable each other in b_2, *b* can never be performed in the first instance of b_2, while *a* can never be performed in the second instance of b_2. The simplified composed behaviour can be specified in the textual notation as follows:

$B = \{ \ b_1.>1 \wedge b_2.>1 \wedge \surd \wedge \surd \rightarrow \{b_1^1, b_2^1\}.a,$
$\{b_1^1, b_2^1\}.a \wedge \{b_1^1, b_2^1\}.a \wedge \surd \wedge \surd \rightarrow \{b_1^1, b_2^2\}.b,$
$\{b_1^1, b_2^2\}.b \rightarrow b(B).b_1.>1,$
$\{b_1^1, b_2^2\}.b \rightarrow b(B).b_2.>1\}$



i. Composition after zero substitutions

ii. Composition after one substitution

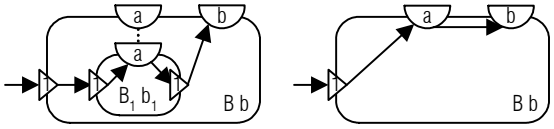
iii. Simplification of the composition after one substitution

Figure 5-22 Simplification of Composition of Repeating Behaviours

Note that we can compose two or more repeating behaviours b_1, b_2, …, if, after i_1 recursive instantiations of b_1, i_2 recursive instantiations of b_2, …, they all have an entry point enabled. This statement implies an important constraint, because we may not find an i_1, i_2, … that satisfies this constraint. In that case we cannot calculate the composition. This leads to a problem, because in case the individual behaviours continue repeating, but never repeat at the same time, the algorithm will end up in a live-lock. It continuously substitutes the behaviour instantiations that can occur, but never stops. Until now, we have not been able to find a solution for this problem.

The class that contains structured, repeating sub-behaviours with finite sub-sub-behaviours. In case a sub-behaviour contains an instantiation of a finite behaviour, it can be composed with other behaviours, after we substituted the instantiation by the actual behaviour. For this purpose, we can use the substitution operator that we defined in the previous paragraph. Figure 5-23.i shows an example of a structured behaviour that contains an instantiation of a finite behaviour. Figure 5-23.ii shows the same behaviour, after we substituted the instantiation for the behaviour that it represents.

Figure 5-23 Example of a Structured Behaviour



i. Structure with finite part ii. Finite part substituted

The class that contains structured, repeating sub-behaviours with repeating sub-sub-behaviours. We also develop an algorithm for the case in which a sub-behaviour *B* contains an instantiation of a repeating behaviour *BC* that satisfies the following constraints.

1. If *B* is repeating, then *B* must disable (all causality targets in) *BC* before *B* repeats. Otherwise, *BC* may have enabled causality targets when *B* repeats. Therewith violating the constraint that *B* is repeating.
2. *BC* (indirectly) interacts with any number of finite monolithic behaviours or at most one repeating monolithic behaviour.

In case these constraints are satisfied, we can compute the composition by first composing *BC* with the behaviours with which it interacts and then composing *B* with the behaviours with which it interacts.

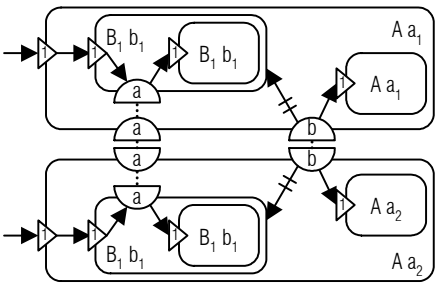
Example 5-13 Structured Behaviour with a Repeating Sub-Behaviour

Figure 5-24.i shows an example of a structured sub-behaviour that contains a repeating behaviour. The behaviours a_1 and a_2 are repeating, but interaction *b* disables repeating behaviour b_1, both in a_1 and in a_2, via the disabling relation that points to that behaviour. Therewith it

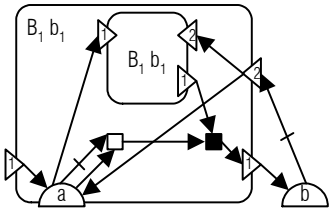
satisfies the first constraint. A disabling relation to a behaviour is a shorthand that represents a disabling relation to all parts of that behaviour. Hence, b disables $b_1.a$, $b_1.b_1.a$, …. Figure 5-24.ii shows how the shorthand can be decomposed into a basic behaviour. Repeating behaviour b_1 in behaviour a_1 interacts with repeating behaviour b_1 of behaviour a_2. Since it does not interact with any other behaviour, the second constraint is satisfied. Therefore, we can compose a_1 and a_2 by first composing b_1 and b_2 and then a_1 and a_2 themselves. In the textual notation, B_1 and A can be specified as follows:

$$
\begin{aligned}
B_1 = \{\ & >1 \wedge >2 && \rightarrow a \\
& a && \rightarrow b_1(B_1).>1, \\
& >2 && \rightarrow b_1(B_1).>2 \\
& (a \vee \neg a) \wedge b_1(B_1).1> && \rightarrow 1>\}
\end{aligned}
$$

$$
\begin{aligned}
A = \{\ & >1 && \rightarrow b_1(B_1).>1, \\
& \neg b && \rightarrow b_1(B_1).>2, \\
& b_1(B_1).1> && \rightarrow b, \\
& b && \rightarrow a_1(A_1).>1\}
\end{aligned}
$$



i. Structure with composable infinite part



ii. Relation represented by behavior disabling

Figure 5-24 Example of a Structured Behaviour with a Repeating Sub-Behaviour

The algorithm for composing repeating behaviours relies on knowing if an entry point can be enabled, to recursively instantiate the associated behaviour. Therefore, we must be able to calculate for each entry point if it can be enabled, or if it is impossible. This is difficult for an entry point that depends on the exit point of a repeating behaviour, because the condition for that exit point may be infinite. Hence, we must define an approach to compute if such an exit point can be enabled.

We have not defined such an approach for the general case. This is left for future work. However, we show for three frequently occurring infinite conditions if the exit point can be enabled.

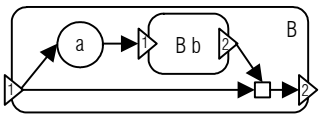
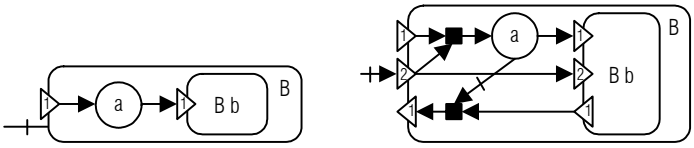


Figure 5-25 Convergent Condition for Dependence on Entry *or* Self

Case I. Figure 5-25 shows a behaviour with an exit point that has an infinite condition, because the condition of the exit point is the disjunction of the entry point of the behaviour and itself (in the recursive instantiation). This behaviour represents a 'while' loop as it can be used in programming languages. While a certain 'while condition' is satisfied, action *a* is performed and the behaviour is repeated, if that condition is not satisfied, the loop exits. Hence, assuming that the 'while condition' eventually becomes false, the exit point is eventually enabled once the entry point is enabled.

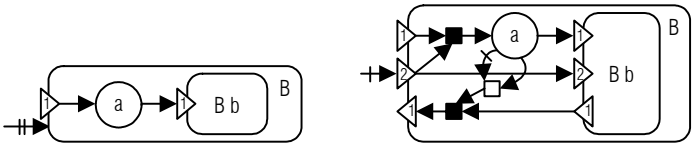
Figure 5-26 Convergent Condition for Choice between Behaviours



i. Structure with infinite condition shorthand ii. Shorthand exploded

Case II. Figure 5-26.i shows a behaviour *B* that has a choice relation with another behaviour or an action or an interaction contribution. Figure 5-26.ii shows what this behaviour looks like after the shorthand is rewritten in terms of basic concepts, showing that 1> represents an infinite condition, because it depends on itself. However, exit point 1> is enabled as long as none of the *a*'s have happened.

Figure 5-27 Convergent Condition for Disabling of Behaviour



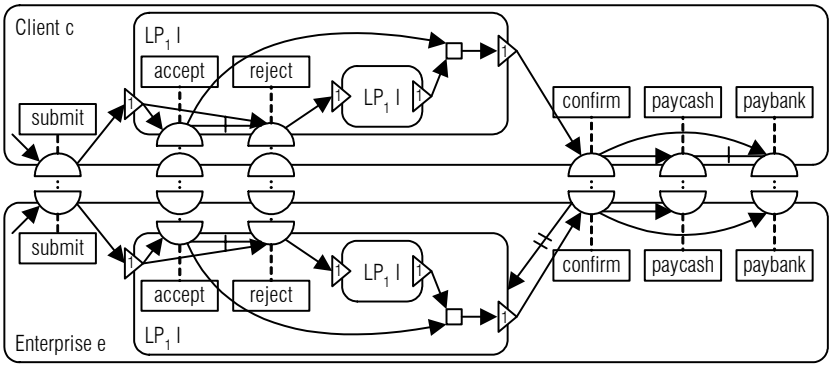
i. Structure with infinite condition shorthand ii. Shorthand exploded

Case III. Figure 5-27.i shows a behaviour that is disabled by another behaviour, an action or an interaction contribution. Figure 5-27.ii shows what this behaviour looks like after the shorthand is rewritten in terms of basic concepts. Exit point 1> has an infinite condition, because it depends on itself. However, exit point 1> is enabled as long as none of the *a*'s is happening at the same time.

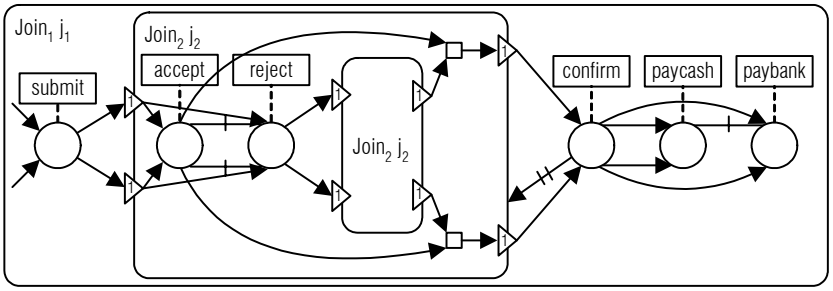
Examples of composition. Figure 5-28.i shows two interacting behaviour instantiations that represent the composite behaviour of a client and an enterprise. On the client side, the client can submit an application, then the client will negotiate the application. During this process, the client may receive several rejections, before the application is accepted. The client can then confirm and finally pays either via the bank or in cash. On the enterprise side, the enterprise can engage in the submission of an application.

After that the enterprise can engage in the negotiation process. The enterprise can continue that process until a confirmation is received. Finally, the enterprise accepts payment by bank and cash.

Figure 5-28 Business Process Example



i. Business process with interacting parties



ii. Composed business process

Both behaviour instantiations are structured, but not recursive. Contained behaviour *l* in the client instantiation only interacts with *l* in the enterprise instantiation and vice versa. Hence, the behaviour instantiations satisfy the constraints that allow them to be composed and we can compose *c* and *e* by first composing *c.l* and *e.l* and then *c* and *e*. The composition of *c* and *e* then is the behaviour from Figure 5-28.ii.

Figure 5-29.i shows an example that represents a question and answer service. There is a questioner (*Q*) that poses a question request (*qreq*) and awaits an answer confirmation (*acnf*), an answerer (*A*) that awaits a question indication (*qind*) and then gives an answer response (*arsp*) and a transporter (*T*) that transports the question from the questioner to the answerer and the answer from the answerer to the questioner. These behaviours are defined as follows:

$$
\begin{aligned}
Q = \{ > 1 \quad &\rightarrow qreq, \\
qreq \quad &\rightarrow acnf, \\
acnf \quad &\rightarrow q(Q).>1\}
\end{aligned}
$$

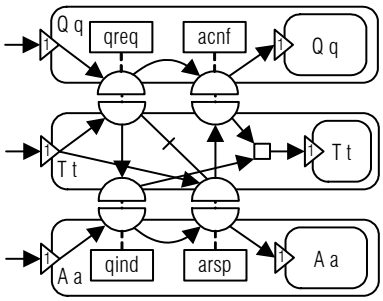
$$T = \{ > 1 \wedge \neg arsp \rightarrow qreq,$$
$$> 1 \wedge \neg qreq \rightarrow arsp,$$
$$qreq \rightarrow qind,$$
$$arsp \rightarrow acnf,$$
$$qind \vee acnf \rightarrow t(T).>1\}$$

$$A = \{ > 1 \rightarrow qind,$$
$$qind \rightarrow arsp,$$
$$arsp \rightarrow a(A).>1\}$$

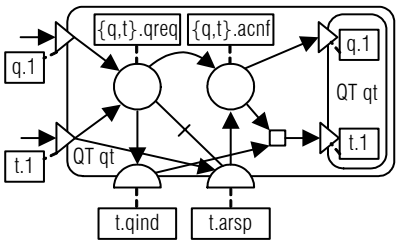
The interactions between these behaviours are:

$$I = \{ q.qreq \wedge t.qreq \Rightarrow qreq,\ q.acnf \wedge t.acnf \Rightarrow acnf,$$
$$q.qind \wedge t.qind \Rightarrow qind,\ q.arsp \wedge t.arsp \Rightarrow arsp\}$$

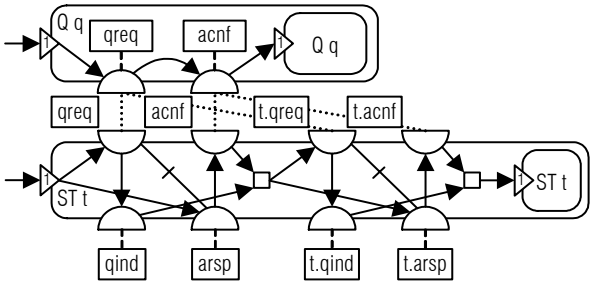
Figure 5-29 Question and Answer Service



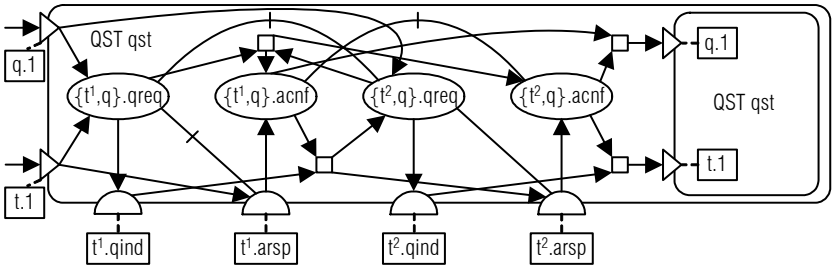
i. Question and answer service



ii. Questioner and transporter composed



iii. Questioner and transporter after one instantiation



iv. Questioner and transporter composed after one instantiation

Hence, using the composition operator, we can define the composition of Q and T as shown in Figure 5-29.ii. In the textual notation this corresponds to (for brevity we prefix actions only with the behaviour names of the interaction contributions from which they are derived):

$$
\begin{array}{l}
q(Q) \oplus t(T) = qt(QT), \text{ where} \\
\quad QT = \{ \quad q.{>}1 \wedge t.{>}1 \wedge \neg t.arsp \rightarrow \{q, t\}.qreq, \\
\qquad\quad \{q, t\}.qreq \wedge t.arsp \quad\;\; \rightarrow \{q, t\}.acnf, \\
\qquad\quad t.{>}1 \wedge \neg\{q, t\}.qreq \quad\;\; \rightarrow t.arsp, \\
\qquad\quad \{q, t\}.qreq \quad\;\; \rightarrow t.qind, \\
\qquad\quad \{q, t\}.acnf \quad\;\; \rightarrow qt(QT).q.{>}1, \\
\qquad\quad t.qind \vee \{q, t\}.acnf \quad\;\; \rightarrow qt(QT).t.{>}1\}
\end{array}
$$

Using the rules for calculating with impossibility, we can derive that, in the composition only T can repeat and Q cannot repeat. This can be done as follows (for brevity, we only show some of the causality relations in QT):

$$
\begin{array}{lll}
QT & =_{17} & \{\ldots, \neg\{q, t\}.qreq \wedge \{q, t\}.qreq \wedge t.arsp \rightarrow \{q, t\}.acnf, \ldots\} \\
& =_{10} & \{\ldots, \dagger \wedge t.arsp \rightarrow \{q, t\}.acnf, \ldots\} \\
& =_{11} & \{\ldots, \dagger \rightarrow \{q, t\}.acnf, \ldots\} \\
& =_{14} & \{\ldots, \dagger \rightarrow \{q, t\}.acnf, \dagger \rightarrow qt(QT).q.{>}1, \ldots\}
\end{array}
$$

Since T can repeat, but Q cannot, we have to substitute T in $t(T)$. This also changes the interactions in which T engages. Figure 5-29.iii shows how the recursive instantiation of T affects T and Q and their interactions. Using the textual notation, the result is:

$$
\begin{array}{l}
\sigma\, T\, t(T) = t(ST), \text{ where} \\
ST = \{ \quad {>}1 \wedge \neg arsp \quad\;\; \rightarrow qreq, \\
\qquad\quad {>}1 \wedge \neg qreq \quad\;\; \rightarrow arsp, \\
\qquad\quad qreq \quad\;\; \rightarrow qind, \\
\qquad\quad arsp \quad\;\; \rightarrow acnf, \\
\qquad\quad (qind \vee acnf) \wedge \neg t.arsp \quad\;\; \rightarrow t.qreq, \\
\qquad\quad (qind \vee acnf) \wedge \neg t.qreq \quad\;\; \rightarrow t.arsp, \\
\qquad\quad t.qreq \quad\;\; \rightarrow t.qind, \\
\qquad\quad t.arsp \quad\;\; \rightarrow t.acnf, \\
\qquad\quad t.qind \vee t.acnf \quad\;\; \rightarrow t(ST).{>}1\}
\end{array}
$$

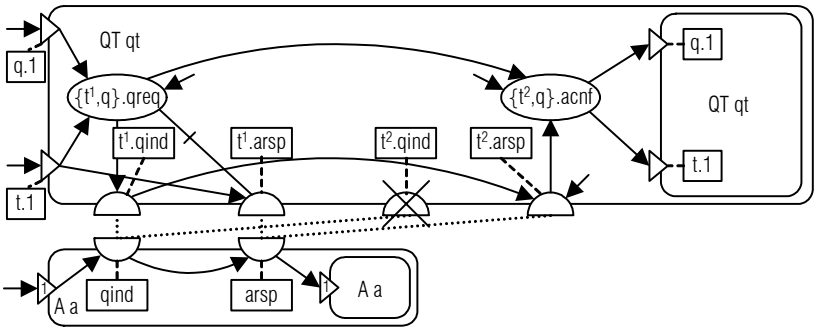
$$
\begin{array}{l}
(t^1.qreq \vee t^2.qreq) \wedge q.qreq \Rightarrow qreq \\
(t^1.acnf \vee t^2.acnf) \wedge q.acnf \Rightarrow acnf
\end{array}
$$

Hence, the composition of Q and ST, shown in Figure 5-29.iv, is:

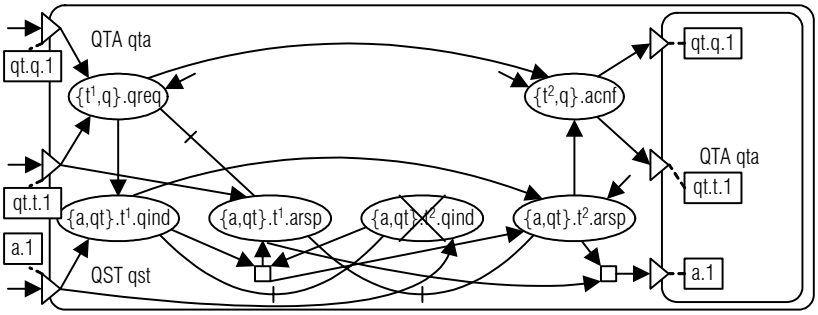
$q(Q) \oplus t(ST) = qt(QST)$, where

$$
\begin{aligned}
QST = \{\ & t.>1 \wedge \neg t.arsp \wedge q.>1 \wedge \neg\{t^2,q\}.qreq && \rightarrow \{t^1,q\}.qreq,\\
& (t.qind \vee \{t^2,q\}.acnf) \wedge \neg t^2.arsp \wedge q.>1 \wedge \{t^1,q\}.qreq && \rightarrow \{t^2,q\}.qreq,\\
& t.arsp \wedge (\{t^1,q\}.qreq \vee \{t^2,q\}.qreq) \wedge \neg\{t^2,q\}.acnf && \rightarrow \{t^1,q\}.acnf,\\
& t^2.arsp \wedge (\{t^1,q\}.qreq \vee \{t^2,q\}.qreq) \wedge \neg\{t^1,q\}.acnf && \rightarrow \{t^2,q\}.acnf,\\
& t.>1 \wedge \neg\{t^1,q\}.qreq && \rightarrow t^1.arsp,\\
& \{t^1,q\}.qreq && \rightarrow t^1.qind,\\
& (t.qind \vee \{t^1,q\}.acnf) \wedge \neg\{t^2,q\}.qreq && \rightarrow t^2.arsp,\\
& \{t^2,q\}.qreq && \rightarrow t^2.qind,\\
& t^2.qind \vee \{t^2,q\}.acnf && \rightarrow t.>1,\\
& \{t^1,q\}.acnf \vee \{t^2,q\}.acnf && \rightarrow q.>1\}
\end{aligned}
$$

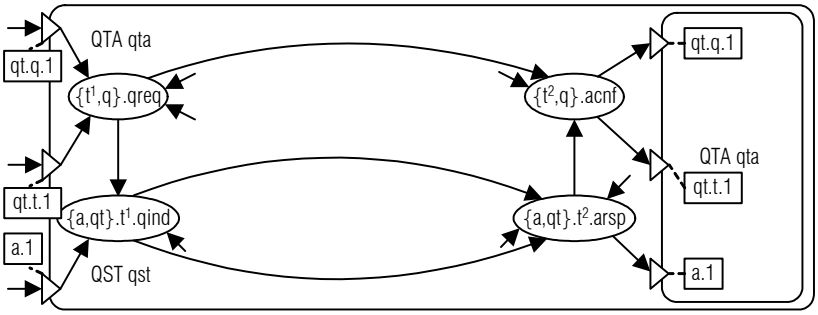
Figure 5-30 Question and Answer Service (continued)



i. Simplified questioner/transporter and answerer



ii. Simplified questioner/transporter and answerer composed



iii. Final simplified composition

Using the rules for calculating with impossibility, we can derive that in the composition of Q and ST, both Q and ST can repeat. Also, we can simplify the composite behaviour into the behaviour shown in Figure 5-30.i (the crossed-out interaction contribution in the figure represents an impossible interaction contribution). This can be done as follows (again, we only show causality relations that change, at the end we show the entire behaviour):

$$
\begin{aligned}
&QST = {}_{17}\\
&\{\ \ldots,\\
&\quad \neg\{t^1,q).qreq \wedge arsp \wedge (\{t^1,q\}.qreq \vee \{t^2,q\}.qreq) \wedge \neg\{t^2,q\}.ancf \rightarrow \{t^1,q\}.ancf\\
&\quad \ldots\}\\
&= {}_{5,\,10,\,11,\,12}\\
&\{\ \ldots, \neg\{t^1,q).qreq \wedge arsp \wedge \{t^2,q\}.qreq \wedge \neg\{t^2,q\}.ancf \rightarrow \{t^1,q\}.ancf,\\
&\quad \ldots\}\\
&= {}_{17}\\
&\{\ \ldots, \neg\{t^1,q).qreq \wedge arsp \wedge \{t^2,q\}.qreq \wedge \neg\{t^2,q\}.ancf \rightarrow \{t^1,q\}.ancf,\\
&\quad \neg qind \wedge (qind \vee \{t^1,q\}.ancf) \wedge \neg t.arsp \wedge q.{>}1 \wedge \neg\{t^1,q\}.qreq \rightarrow \{t^2,q\}.qreq,\\
&\quad \ldots\}\\
&= {}_{5,\,10,\,11,\,12}\\
&\{\ \ldots, \neg\{t^1,q).qreq \wedge arsp \wedge \{t^2,q\}.qreq \wedge \neg\{t^2,q\}.ancf \rightarrow \{t^1,q\}.ancf,\\
&\quad \neg qind \wedge \{t^1,q\}.ancf \wedge \neg t.arsp \wedge q.{>}1 \wedge \neg\{t^1,q\}.qreq \rightarrow \{t^2,q\}.qreq,\\
&\quad \ldots\}\\
&= {}_{15}\\
&\{\ \ldots, \neg\{t^1,q).qreq \wedge arsp \wedge \{t^2,q\}.qreq \wedge \neg\{t^2,q\}.ancf \rightarrow \{t^1,q\}.ancf,\\
&\quad \{t^2,q\}.qreq \wedge \neg qind \wedge \{t^1,q\}.ancf \wedge \neg t.arsp \wedge q.{>}1\\
&\qquad\qquad\qquad\qquad \wedge \neg\{t^1,q\}.qreq \rightarrow \{t^2,q\}.qreq,\\
&\quad \ldots\}\\
&= {}_{8,\,11}\\
&\{\ \ldots, \neg\{t^1,q).qreq \wedge arsp \wedge \{t^2,q\}.qreq \wedge \neg\{t^2,q\}.ancf \rightarrow \{t^1,q\}.ancf,\\
&\quad \dagger \rightarrow \{t^2,q\}.qreq,\ \ldots\}\\
&= {}_{14,\,11}\\
&\{\ \ldots, \dagger \rightarrow \{t^1,q\}.ancf,\ \dagger \rightarrow \{t^2,q\}.qreq,\ \ldots\}\\
&=\\
&\{\ t.{>}1 \wedge \neg t.arsp \wedge q.{>}1 \wedge \surd \rightarrow \{t^1,q\}.qreq,\\
&\quad \dagger \rightarrow \{t^2,q\}.qreq,\\
&\quad \dagger \rightarrow \{t^1,q\}.ancf,\\
&\quad t^2.arsp \wedge \{t^1,q\}.qreq \wedge \surd \rightarrow \{t^2,q\}.ancf,\\
&\quad t.{>}1 \wedge \neg\{t^1,q\}.qreq \rightarrow t.arsp,\\
&\quad \{t^1,q\}.qreq \rightarrow t.qind,\\
&\quad t.qind \wedge \surd \rightarrow t^2.arsp,\\
&\quad \dagger \rightarrow t^2.qind,\\
&\quad \{t^2,q\}.ancf \rightarrow qt(QST).t.{>}1,\\
&\quad \{t^2,q\}.ancf \rightarrow qt(QST).q.{>}1\}
\end{aligned}
$$

We can compose this resulting behaviour with A. Note that the interactions between T and A have also changed, when T was recursively instantiated, into:

$$a.qind \wedge (qt.t^1.qind \vee qt.t^2.qind) \Rightarrow qind$$
$$a.arsp \wedge (qt.t^1.arsp \vee qt.t^2.arsp) \Rightarrow arsp$$

Figure 5-30.ii shows the result of the composition. Figure 5-30.iii shows the result of the composition, after eliminating all impossible actions and simplifying the behaviour. The composition and the simplification can be calculated as follows:

$qt(QST) \oplus a(A) = aqt(AQT)$, where
$AQT =$
$\{\ qt.t.>1 \wedge \neg\{a,qt\}.t^1.arsp \wedge qt.q.>1 \wedge \surd \rightarrow \{t^1,q\}.qreq,$
$\quad \dagger \rightarrow \{t^2,q\}.qreq,$
$\quad \dagger \rightarrow \{t^1,q\}.ancf,$
$\quad \{a,qt\}.t^2.arsp \wedge \{t^1,q\}.qreq \wedge \surd \rightarrow \{t^2,q\}.ancf,$
$\quad a.>1 \wedge \{t^1,q\}.qreq \wedge \neg\{a,qt\}.t^2.qind \rightarrow \{a,qt\}.t^1.qind,$
$\quad a.>1 \wedge \dagger \wedge \neg\{a,qt\}.t^1.qind \rightarrow \{a,qt\}.t^2.qind,$
$\quad (\{a,qt\}.t^1.qind \vee \{a,qt\}.t^2.qind) \wedge qt.t.>1$
$\qquad \wedge\neg\{t^1,q\}.qreq \wedge \neg\{a,qt\}.t^2.arsp \rightarrow \{a,qt\}.t^1.arsp,$
$\quad (\{a,qt\}.t^1.qind \vee \{a,qt\}.t^2.qind) \wedge \{a,qt\}.t^1.qind$
$\qquad \wedge \surd \wedge \neg\{a,qt\}.t^1.arsp \rightarrow \{a,qt\}.t^2.arsp,$
$\quad \{t^2,q\}.acnf \rightarrow aqt(AQT).qt.t.>1,$
$\quad \{t^2,q\}.acnf \rightarrow aqt(AQT).qt.q.>1,$
$\quad \{a,qt\}.t^1.arsp \vee \{a,qt\}.t^2.arsp \rightarrow aqt(AQT).a.>1\}$
$=_{11}$
$\{\ \ldots, \dagger \rightarrow \{a, qt\}.t^2.qind, \ldots\}$
$=_{14, 13, 12}$
$\{\ \ldots, a.>1 \wedge \{t^1,q\}.qreq \wedge \surd \rightarrow \{a,qt\}.t^1.qind,$
$\quad \{a,qt\}.t^1.qind \wedge qt.t.>1 \wedge \neg\{t^1,q\}.qreq \wedge \neg\{a,qt\}.t^2.arsp \rightarrow \{a,qt\}.t^1.arsp,$
$\quad \{a,qt\}.t^1.qind \wedge \{a,qt\}.t^1.qind \wedge \surd \wedge \neg\{a,qt\}.t^1.arsp \rightarrow \{a,qt\}.t^2.arsp,$
$\quad \ldots\}$
$=_{17}$
$\{\ \ldots, \neg\{a,qt\}.t^1.qind \wedge \{a,qt\}.t^1.qind \wedge qt.t.>1$
$\qquad \wedge \neg\{t^1,q\}.qreq \wedge \neg\{a,qt\}.t^2.arsp \rightarrow \{a,qt\}.t^1.arsp,$
$\quad \ldots\}$
$=_{10, 11}$
$\{\ \ldots, \dagger \rightarrow \{a,qt\}.t^1.arsp, \ldots\}$
$=_{14, 11, 13}$
$\{\ \ldots, \{a,qt\}.t^1.qind \wedge \{a,qt\}.t^1.qind \wedge \surd \wedge \surd \rightarrow \{a,qt\}.t^2.arsp,$
$\quad \{a,qt\}.t^2.arsp \rightarrow aqt(AQT).a.>1, \ldots\}$

$$
\begin{array}{ll}
= & \\
\{\ qt.t.{>}1 \wedge \surd \wedge qt.q.{>}1 \wedge \surd & \rightarrow \{t^1,q\}.qreq, \\
\quad \dagger & \rightarrow \{t^2,q\}.qreq, \\
\quad \dagger & \rightarrow \{t^1,q\}.ancf, \\
\quad \{a,qt\}.t^2.arsp \wedge \{t^1,q\}.qreq \wedge \surd & \rightarrow \{t^2,q\}.ancf, \\
\quad a.{>}1 \wedge \{t^1,q\}.qreq \wedge \surd & \rightarrow \{a,qt\}.t^1.qind, \\
\quad \dagger & \rightarrow \{a,qt\}.t^2.qind, \\
\quad \surd & \rightarrow \{a,qt\}.t^1.arsp, \\
\quad \{a,qt\}.t^1.qind \wedge \{a,qt\}.t^1.qind \wedge \surd \wedge \surd & \rightarrow \{a,qt\}.t^2.arsp, \\
\quad \{t^2,q\}.acnf & \rightarrow aqt(AQT).qt.t.{>}1, \\
\quad \{t^2,q\}.acnf & \rightarrow aqt(AQT).qt.q.{>}1, \\
\quad \{a,qt\}.t^2.arsp & \rightarrow aqt(AQT).a.{>}1\}
\end{array}
$$

5.2.6 Discussion

We defined the operators for abstraction and for equivalence assessment in this section for a subset of the possible behaviours that we can specify using the basic concepts from chapter 4.

We defined the action abstraction and integration operators and the equivalence relation for the class of monolithic, possibly recursive, behaviours. In addition we can use these operators to verify consistency in structured behaviours, if we first compose these behaviours into monolithic behaviours. However, this has the limitation that we cannot verify consistency with respect to (re-)distribution of responsibility for enforcing causality conditions among interacting parties. We defined the behaviour composition operator for structured behaviours that either contain finite sub-behaviours or repeating sub-behaviours.

We defined the abstraction operators and the equivalence relation in such detail that we can implement them in tools and provided prototype implementations. We tested (the implementation of) the operators for some example behaviours, but not extensively. Also, we have not proven the correctness of the operators formally. A suggestion for assessing the correctness of the operators is to use the operational semantics for the basic concepts (Quartel, 1998) and verify the correctness of the operators with respect to this semantics. Also, correctness of the operators can be assessed via extensive testing of the implementations. The composition operator for the class of behaviours that contain structured sub-behaviours is defined informally and not implemented.

5.3 Pre-Defined Overlap Relations and Consistency Rules

Overlap exists between concerns and viewpoints that partly consider the same system properties. With respect to the properties that they both consider, views of overlapping viewpoints must be equivalent. We distinguish three frequently occurring cases of overlap between views. Based on these cases of overlap, we pre-define four possible overlap relations between concept instances from overlapping views. Also, we illustrate how consistency rules can be specified on (the MOF associations that represent) these relations.

5.3.1 Cases of Overlap

If two viewpoints overlap, either: (i) they consider exactly the same properties; (ii) one viewpoint considers all the properties that the other considers as well as some other properties; or (iii) they have some properties in common but both also consider other properties. These different forms of overlap are also identified by Spanoudakis, Finkelstein, & Till (1999). They are illustrated in Figure 5-31. The figure shows the properties that are addressed on the horizontal axis. It represents concerns on the property axis between dashed lines, representing that they are used to represent the properties between these dashed lines. Viewpoints that address these concerns are drawn between dashed lines as well, representing that they address the concerns that belong to the dashed lines.

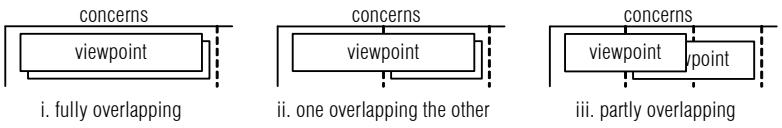


Figure 5-31 Different kinds of Overlap between Viewpoints

One could argue that it is a mistake to have completely overlapping viewpoints or concerns or to have one viewpoint or concern completely overlapping the other, because that means that one of the viewpoints or concerns does not add any design information, since all design information is present in one of the viewpoints. However, we argue that there may be a benefit in (graphically) representing the same system properties in a different fashion. Representing the same system properties differently may improve the understanding of the designers. For example, we can represent the behaviour of a system and the execution traces that that behaviour produces as two separate concerns and in two separate views. Although the execution traces that the behaviour of the system produces are completely described by the behaviour of the system, describing these traces separately may improve the understanding of the behaviour of the system.

One could also argue that it is a mistake to have one viewpoint or concern completely overlapping another, because then the overlapping viewpoint or concern adds design information with respect to the overlapped viewpoint and, therefore, should be considered a refinement rather than an overlap of the overlapped viewpoint. However, we argue that this is a matter of how you choose your concerns and levels of detail. If the concerns and levels of detail are chosen as in Figure 5-31.ii the viewpoints *must* overlap and *not* have a refinement relation. This is so because, by definition, a refinement relation can only exist between the same properties at *different* levels of detail, while the viewpoints from Figure 5-31.ii consider the same properties at *the same* level of detail. In contrast, we could have chosen the viewpoints such that they address the same concern at different levels of detail. Both ways of positioning the viewpoints in the framework have their own benefit, because they stress different design relations. One stresses the refinement relation, therewith positioning the viewpoints in a relation to each other that can be traced back to the design process (gradually adding detail). The other stresses the overlap relation, such that the viewpoints can be considered at the same level of abstraction in the design process.

5.3.2 Cases of Overlap between Concept Instances

Since overlap between views exists between the parts of those views that represent the same properties, we can represent overlap between views by relating concept instances (not necessarily of the same concept) that represent the same property. However, concept instances may not only represent the same property, but other properties as well. Also, concept instances may only represent a property in a composition. Figure 5-32 illustrates different overlap relations that correspond to different ways in which two concept instances represent the same property. Below we explain these overlap relations in more detail.

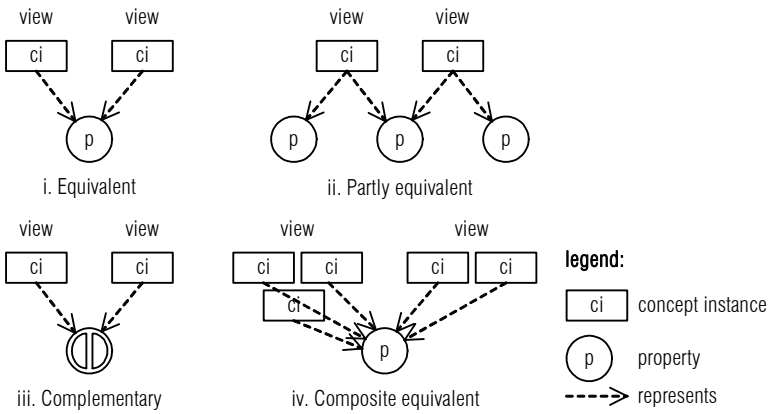
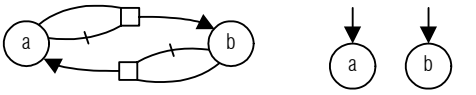


Figure 5-32 Correspondences between Concept Instances from Overlapping Views

Equivalent concept instances. If two concept instances represent exactly the same property they are, by definition, equivalent. Figure 5-32.i illustrates this case. Equivalent concept instances can have a different graphical or textual representation. Moreover, the concept instances may be instances of concepts with different names (e.g. Task versus Activity). The concepts can also differ in that one concept may represent a property explicitly as an attribute, while the other does not. Such representation choices do not determine the semantics of the concept and therefore do not determine whether two concepts are equivalent. Only the property represented determines whether two concepts are equivalent.



i. A and B interleaved ii. A and B concurrent

Figure 5-33 Equivalent Behaviours under Some Assumptions

Different notions of equivalence. It is possible that two concept instances only represent the same property, if we make certain assumptions about (what properties we consider relevant in) the real world domain. We then abstract from other properties. In that case, we can say that two concept instances are equivalent *under these assumptions*. For example, consider the behaviours from Figure 5-33. These behaviours can be equivalent if we abstract from causal relations between actions and from full-concurrency properties (whether two actions can occur independent of each other). These assumptions are often used for equivalence notions on finite state machines. A large variety of equivalence relations for behaviour exists, which make different assumptions about properties they consider relevant (van Glabbeek, & Goltz, 2001). We pre-defined the notion of strong behaviour equivalence on our basic concepts from chapter 4. The designer can use this equivalence relation to prescribe that two behaviours must be equivalent with respect to the causal relations that exist between activities. The question how this notion of equivalence relates to the notions distinguished by van Glabbeek, & Golz remains for further study. Obviously, the actions from the behaviours in Figure 5-33.i and Figure 5-33.ii do not have the same causality conditions and therefore are not equivalent under the assumptions of the notion of strong behaviour equivalence.

Example 5-14 Equivalent Concept Instances

As an example of equivalent concept instances, consider a UML Class Diagram and a UML State Machine Diagram. UML Class Diagrams can be used to represent the classes that exist in an object-oriented program and methods that can be invoked on those classes. UML State Machine Diagrams can be used to represent when methods can be invoked. Hence, if we consider UML Class Diagrams and UML State Machine Diagrams as different viewpoints with their own sets of concepts, then there is an equivalence relation between each method in a UML State Machine

Diagram and a method in a UML Class Diagram. We can represent this relation by a MOF association between the 'Method' meta-classes in both viewpoints. Also, we can prescribe consistency rules in OCL to check this form of equivalence. For example, we can define that associated methods must have the same signature.

As another example of equivalence, consider a UML Activity Diagram and a UML State Machine Diagram as two views that represent the same behaviour. However, a UML Activity Diagram can represent that two actions occur at the same time, while in a UML State Machine actions are always interleaved. Hence, they can only represent the same behaviour under the assumption that it does not matter whether actions occur at the same time or interleaved. Of course, the designer is free to prescribe a strong equivalence relation between the behaviours that are represented by the diagrams. However, this is likely to result in inconsistencies (that can be verified by the consistency rules), because the designer makes a claim that is not true.

Partly equivalent concepts instances. If a concept instance from one view represents partly the same property as a concept instance from another view as well as some other properties, then these concept instances are only equivalent with respect to the property that they have in common. Hence, we say that they are partly equivalent. Figure 5-32.ii illustrates this case. For partly equivalent concept instances, the designer has to define his own relation and prescribe exactly with respect to which part the concept instances are equivalent. The designer can re-use the strong equivalence notion on behaviours to prescribe that two behaviours must be, in part, equivalent.

Example 5-15 Partly Equivalent Concept Instances

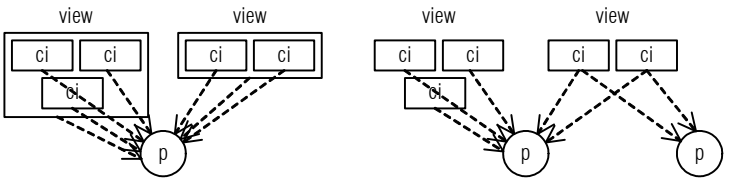
The behaviour of a service and the behaviour of that service at a particular interaction point *ip* are partly equivalent. The designer can represent this form of partial equivalence, by prescribing that the service behaviour, after abstracting from all interactions that occur at other interaction points, must be equivalent to the service behaviour at interaction point *ip*. We can prescribe this consistency in OCL, using the 'abstract' operator and the 'strong equivalence' relation defined earlier in this chapter.

Complementary concept instances. If two concept instances represent different properties of the same entity, we say that they are complementary. The concept instances 'overlap', because they deal with the same entity. Figure 5-32.iii illustrates this case. In this case the concepts instances must be related by some other relation than an equivalence relation, because they do not represent the same property. The designer must define this relation. As an example, consider a view that considers causality conditions and a view that considers causality constraints. The causality conditions describe the condition for an action to occur in terms of the (non-)occurrence of other actions, while a causality constraint describes the condition for an action to occur in terms of constraints on the information that is established by other actions. Causality conditions and constraints complement each other to represent the overall condition for an action to occur. They

are related by an association that associates a causality constraint to an alternative causality condition in the meta-model from chapter 4.

Composite equivalent concept instances. If a composition of concept instances from one view represents the same property as a composition of concept instances from another view, these compositions are equivalent. Figure 5-32.iv illustrates this case. Since, using the MOF, we can only relate individual concept instances, we can only represent equivalence between a composition of concept instances and some other concept instance, after we added a single concept instance that represents the composition as a whole. This concept instance must be related (directly or indirectly) to the concept instances that form the composition. Subsequently we can relate the concept instance that represents the composition via an equivalence relation to another concept instance (that represents another composition as a whole). Figure 5-34.i illustrates this situation. As an example, consider two views that prescribe sets of actions and their relations. To represent that these sets of actions and their relations are equivalent, we can add a behaviour concept instance to each of the views and draw the equivalence relation between those behaviours. The behaviours must be related to the actions of which they are a composition.

Figure 5-34 Complex Correspondences between Concept Instances



i. Composite equivalence and equivalence ii. Composite equivalence and part equivalence

We can also combine the basic overlap relations from Figure 5-32 into more complex relations. Figure 5-34 illustrates two more complex correspondences. Figure 5-34.i illustrates the case that we described in the previous paragraph, in which the same property is represented by both a composition of concept instances and a single concept instance. Figure 5-34.ii illustrates equivalence between a composition of concept instances in one view and a composition of concept instances in another view that represents the same property as well as another property. The correspondence between these compositions is such that the composition on the left is equivalent to a part of the composition on the right.

Chapter 6

Enterprise, Computational and Information Viewpoint

An earlier version of the work presented in this chapter was published in (Dijkman, Quartel, Ferreira Pires, & van Sinderen, 2004).

As a case study for the framework outlined in this thesis, this chapter presents three viewpoints: an enterprise, a computation and an information viewpoint. We based our viewpoints on the corresponding RM-ODP viewpoints.

This chapter is structured as follows. Section 6.1 presents the goal and scope of the case study. Sections 6.2 and 6.3 present our enterprise and computational viewpoint, respectively. Section 6.4 presents the relations between these viewpoints and the consistency rules that apply to these relations. Section 6.5 presents our information viewpoint and its relations to the other two viewpoints along with the consistency rules that apply to these relations.

6.1 Goal and Scope of the Case Study

The goal of the case study is to show that the framework outlined in this thesis can be applied to realistic viewpoints. The main goal is to show that the framework aids in defining rules to check the consistency between views in a design, using a set of basic concepts and re-usable consistency rules. In addition, the case study serves to illustrate:

- how to define viewpoints in the framework;
- how to define an abstract syntax, that consists of concepts and their relations, for each viewpoint;
- how to relate a graphical notation (concrete syntax) to a collection of concepts, to represent a view according to a viewpoint;

– how to define viewpoint-specific concepts as compositions of basic concepts; and
– how to re-use view relations and consistency rules that were defined on the basic concepts to define view relations and to check consistency between views.

The focus of the case study is on the evaluation of the *framework* rather than the evaluation of the viewpoints that we define in this chapter.

Currently, the pre-defined viewpoint relations and consistency rules address relations and consistency between *behaviours*. Therefore, this chapter focuses on *behaviour* relations and consistency.

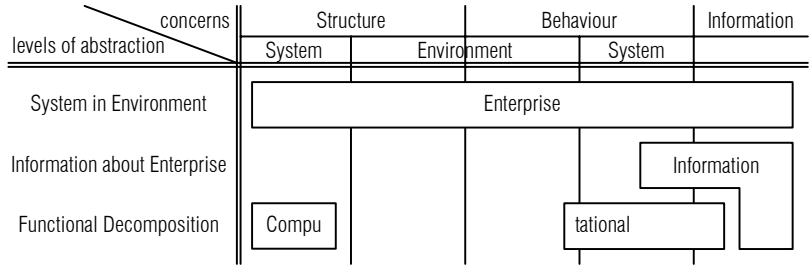


Figure 6-1 Relative Position of ODP Viewpoints

Figure 6-1 shows the viewpoints that we present in this chapter and their relative position with respect to the concerns and abstraction levels that we consider (as explained in chapter 1). We base the positions of the viewpoints on the correspondences that RM-ODP prescribes and on the definitions of the viewpoints themselves (ITU-T, & ISO/IEC, 1999; ITU-T, & ISO/IEC, 1995).

The enterprise viewpoint addresses the structural, behavioural and information concerns of the enterprise, covering both the system and its environment.

The computational viewpoint addresses the structure, behaviour and information concerns of a functional decomposition of the system. Since it considers a decomposition of the system, it refines the part of the enterprise viewpoint that represents the system.

The information viewpoint focuses on the structure and values of information that is used in the system and its environment. It refines the enterprise viewpoint with respect to the information concern. The computational viewpoint makes use of the information viewpoint to define the structure and values of the information that it manipulates. Therefore, the information and computational viewpoint overlap with respect to this concern.

6.2 Enterprise Viewpoint

The enterprise viewpoint is used to design a system in its environment. This is the concern of a stakeholder that wants to have an overview of how the system is embedded in an environment, such as a director, a quality assurance manager or an enterprise architect. To represent the system in its environment, an enterprise view represents the community of which the system is a part. A *community* is a configuration of objects that is formed to meet an objective. The system can be represented by one or more objects in that community. To represent the community in more detail, the enterprise viewpoint addresses the following concerns:

- the objective of the community;
- the structure of the community;
- the behaviour of the community; and
- the policies that govern the structure and behaviour of the community.

It addresses these concerns at the level of abstraction at which the community's objects represent people, business units or applications. The behaviour of the community (and its objects) is represented in terms of business tasks.

6.2.1 Enterprise Viewpoint Concepts

Figure 6-2 shows the enterprise viewpoint concepts and their relations, represented in the UML Profile for MOF 1.4. This conceptual model is an adaptation of a part of the conceptual model that is defined in (ITU-T, & ISO/IEC, 2005). We selected the part that considers the structure and behaviour of the system, but not the policies that govern the structure and behaviour, nor the concepts that relate to accountability. Inclusion of these concepts is left for future work. Other conceptual models for the RM-ODP enterprise viewpoint exist (Steen, & Derrick, 2000). We chose the one from (ITU-T, & ISO/IEC, 2005), because it is the most recent and because it is a part of an international standardization activity. We made some changes to the conceptual model, to better match the RM-ODP enterprise viewpoint specification. We explain these changes below. Also, we made the following changes, to match the syntax prescribed in the UML profile for MOF:

1. We changed ternary associations into (multiple) binary associations, because the MOF 1.4 does not support ternary associations.
2. We changed the way in which roles of classes in associations are graphically represented. In the UML profile for MOF 1.4 the role of a class in an association is drawn where the association is attached to that class, while in the conceptual model the role is drawn on the other side of the association. we changed the names of the roles accordingly.

3. We added a *name* attribute to some of the concepts, such that their instances can be more easily recognized in a MOF repository.

Figure 6-2 shows a MOF meta-model that represents the enterprise concepts and their relations. For brevity, we prefixed the concept names with 'E_' instead of 'Enterprise' in the figure. Figure 6-2 represents that a community is a configuration of *enterprise objects*. An enterprise object represents a carrier of behaviour in the enterprise. A community can be represented as a single *community object*, such that its constituents are not revealed. Since a community object is an object itself, it can participate in another commu-

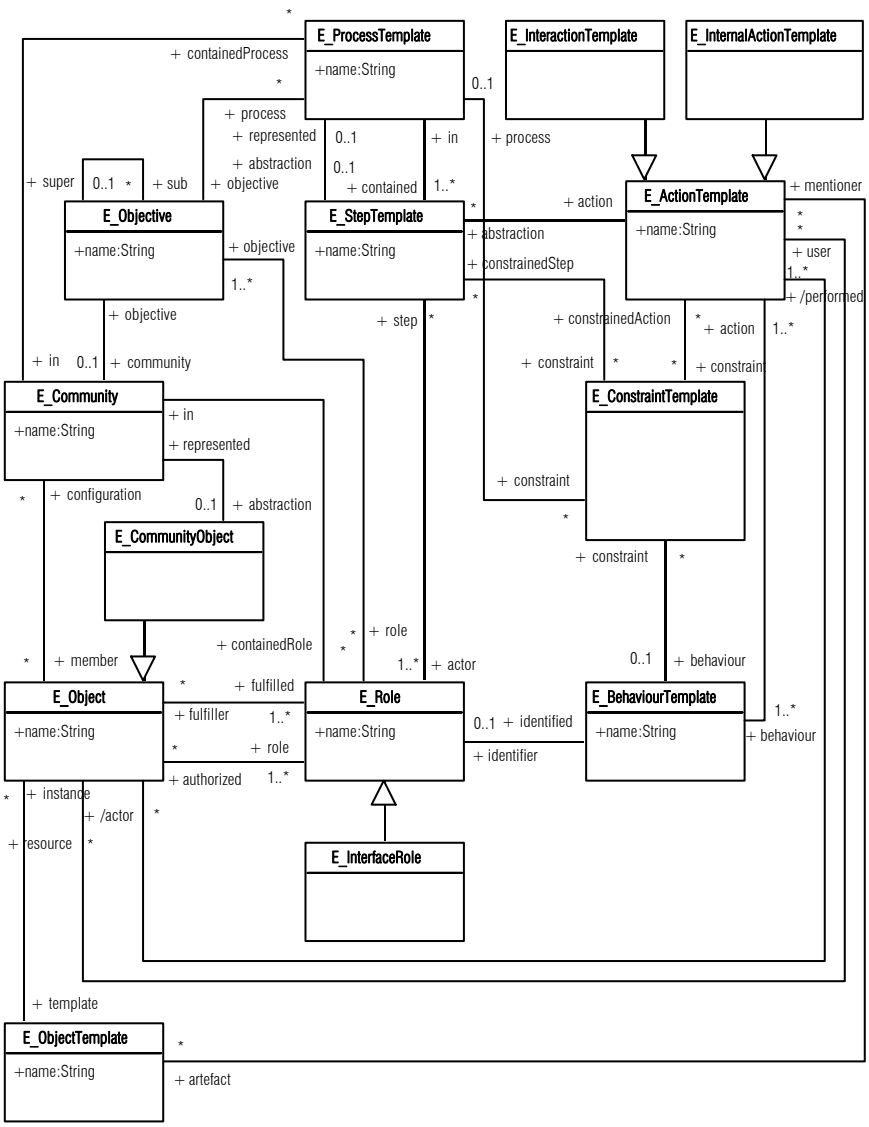


Figure 6-2 Conceptual Model for the Enterprise Viewpoint

nity. An *enterprise object template* represents properties of enterprise objects in such detail that new objects can be created according to it. Each newly created enterprise object is an instance of its template.

Figure 6-2 represents the relation between a community and its objective as an association. In the model from (ITU-T, & ISO/IEC, 2005), a community is associated to its objective via a contract. However, we chose not to represent the contract concept explicitly, because a contract is completely defined by its constituents: the enterprise's objective, its behaviour and the policies that govern the enterprise. The objective of a community can be described in more detail by some sub-objectives.

The behaviour of a community is described by the roles and processes that are performed in that community, which, therefore, have a containment relation with the community.

A *role* is an identifier for a behaviour template. An object that fulfils that role performs an instance of the template. Our definition of role deviates from the one given in RM-ODP (ITU-T, & ISO/IEC, 1995, part 2 clause 9.14), where "*A role is an identifier for a behaviour …*", rather than a behaviour template. We adapted our role concept from the role concept that is used in the workflow area (where a role is: "*A group of participants exhibiting a specific set of attributes, qualifications and/or skills.*" (Workflow Management Coalition, 1999)). In this definition a role identifies a group of objects and their behaviours, rather than a single object behaviour. We motivate this change to the role concept by the ability to model the four-eye-principle. The four-eye-principle involves two people in the same role cross-checking each others work. Using our role concept, we can represent this as two objects fulfilling the same role. Using the RM-ODP role concept, we must represent this as two objects that fulfil different roles (of the same type). An *interface role* is a role that includes interactions with objects outside of the community. A role contributes to achieving an objective or sub-objective of the community. Our conceptual model differs from the original conceptual model in this respect, because the original conceptual model assigns an objective to behaviours instead of roles. However, (ITU-T, & ISO/IEC, 1999, clause 7.7) states that "*A sub-objective may be assigned to a collection of roles*". Whether or not an enterprise object can fulfil a role at any moment in time is determined by an assignment policy. Although we do not consider policies in detail, we add to the conceptual model that an enterprise object may be allowed, or *authorized*, to fulfil a role, because such authorization is usually part of a business process modelling language (Workflow Management Coalition, 1999).

In RM-ODP a *behaviour* is a collection of actions with a set of constraints on when they may occur. An *action* is something that happens. An action can either be performed by a single object, in which case we refer to it as an *internal action*, or by some objects in collaboration in which case we refer to

it as an *interaction*. RM-ODP does not prescribe how the behaviour constraints must be specified. It allows specifications that are based on RM-ODP to prescribe their own constraint specification concepts and techniques. We explain further on in this chapter how we apply UML activities to specify constraints.

Enterprise objects can be involved in an action in three ways:

1. An object is an *actor* in an action, if it participates in carrying out that action. Whether an enterprise object is involved in an action as an actor can be derived from whether it fulfils a role that identifies a behaviour that performs the action.
2. An object is a *resource* in an action, if the action uses the object and possibly makes it unavailable.
3. An object is an *artefact* in an action, if the action refers to the enterprise object.

A *process* is a collection of steps taking place in a prescribed manner and leading to an objective. A *step* is an abstraction of an action, in that it may leave the objects that participate in it unspecified. A process can be performed in a community as often as required. Therefore, the conceptual model represents process templates rather than process instances. The conceptual model from (ITU-T, & ISO/IEC, 2005) allows that a step is a member of more than one process. We do not allow this. Instead, we allow that an action is represented by more than one step, such that each reference to an action corresponds to a step. In this way each step is a member of exactly one process, such that it is easy to create and destroy steps along with the process to which they belong. Each step must be associated with a role that performs it. A process may itself be considered as a single step, such that its constituent steps are not revealed.

6.2.2 Representation of Enterprise Views

We use UML 2.0 to represent enterprise views by means of models. We define the representation relation between enterprise concepts and UML 2.0 modelling elements, by stereotyping the UML 2.0 modelling element.

We deviate from the representation relation that is defined in (ITU-T, & ISO/IEC, 2005), because this standard represents interactions between roles by sending and receiving signals. We claim (Almeida, Dijkman, Ferreira Pires, Quartel, & van Sinderen, 2005) that representing interactions in this way forces a designer to make choices with respect to interaction mechanisms too early in the design process. Moreover, the way in which (ITU-T, & ISO/IEC, 2005) represents interactions forces the designer to order the exchange of artefacts in an interaction, while the conceptual model from Figure 6-2 does not prescribe that such an ordering must be provided. Other techniques to represent the ODP enterprise view-

point in UML exist as well (Steen, & Derrick, 2000; Aagedal, & Milosevic, 1999). However, these are based on earlier versions of UML.

Table 6-1 Representation of Enterprise Viewpoint Concepts in UML 2.0

Enterprise Viewpoint Concept	UML 2.0 Modelling Element
E_Community	Class stereotyped E_Community
E_CommunityObject	Class stereotyped E_CommunityObject
E_Object	Class stereotyped E_Object
E_ObjectTemplate	Class stereotyped E_ObjectTemplate
E_Role	Actor stereotyped E_Role
E_InterfaceRole	Actor stereotyped E_InterfaceRole
E_BehaviourTemplate	Activity stereotyped E_BehaviourTemplate
E_Objective	Class stereotyped E_Objective
E_ProcessTemplate	Activity stereotyped E_ProcessTemplate
E_StepTemplate	Activity stereotyped E_ProcessTemplate, if the E_StepTemplate represents a E_ProcessTemplate Activity stereotyped E_StepTemplate, otherwise
E_ActionTemplate	Activity stereotyped E_ActionTemplate
E_ConstraintTemplate	Flow in Activity and localPrecondition/localPostcondition of Action

Table 6-1 presents our representation relation between the enterprise viewpoint concepts from Figure 6-2 and the UML 2.0 modelling elements that we use to represent them. We represent the community, community object, enterprise object, enterprise object template and objective concepts by stereotypes of the UML singleton class (which is a class that has a single instance) modelling element. We represent the name of an instance of one of these concepts by the name of the class that represents it. We cannot represent these concepts using UML objects, because we are cannot define UML Associations between UML objects, while we must use UML Associations to represent relations between concept instances. We represent the role and interface role concepts by stereotypes of the UML actor modelling element and the names of their instances by the names of the actors. We represent the behaviour template, process template, step template and action template concepts by stereotypes of the UML activity modelling element. We represent the names of their instances by the names of the activities. We represent the containment of a step template in a process template as the activity that represents the process template calling the activity that represents the step. Such a call is represented by a UML 'call behaviour action'. We represent the containment of an action template in a behaviour template in the same way. We allow an activity that represents a step template or an action template to be implicitly defined by a call behaviour action. We represent constraints in a behaviour or process template by flows

in the UML activity and pre- and postconditions on the UML call behaviour actions. We explain this in more detail in the next subsection.

Table 6-2 Representation of Enterprise Viewpoint Relations in UML 2.0

Enterprise Viewpoint Relation	UML 2.0 Modelling Element
E_Object or E_Role in E_Community	aggregation stereotyped memberObject and memberRole, respectively
E_ProcessTemplate in E_Community	aggregation stereotyped processOf
E_Objective of E_Community	association stereotyped objectiveOf
E_CommunityObject representing E_Community	E_Community and E_CommunityObject having the same name
E_Object fulfilling E_Role	association stereotyped fulfils
E_Object authorized to fulfil E_Role	association stereotyped authorizedForRole
E_Role identifying E_BehaviourTemplate	association stereotyped identifies
E_BehaviourTemplate containing E_ActionTemplate	activity containing call behaviour action, which calls the activity that represents the E_ActionTemplate
instance of E_ObjectTemplate as artefact in E_ActionTemplate	class representing the object template as the type of a parameter of the activity
E_Object as actor in E_ActionTemplate	implicitly represented via the associations between the object, its roles, the behaviours identified by the roles and the actions contained in those behaviours
E_Object as resource in E_ActionTemplate	association stereotyped resourceIn
sub-objective of objective	aggregation stereotyped subObjective
E_Role helping to achieve E_Objective	association stereotyped helpsToAchieve
E_ProcessTemplate containing E_StepTemplate	activity containing call behaviour action, which calls the activity that represents the E_StepTemplate
E_ProcessTemplate towards E_Objective	association stereotyped towards
E_StepTemplate representing E_ProcessTemplate	represented by the activity being stereotyped E_ProcessTemplate rather than E_StepTemplate
E_StepTemplate representing E_ActionTemplate	represented by the activities that represent the E_StepTemplate and the E_ActionTemplate having the same name
E_Role authorized to perform E_StepTemplate	association stereotyped authorizedForStep

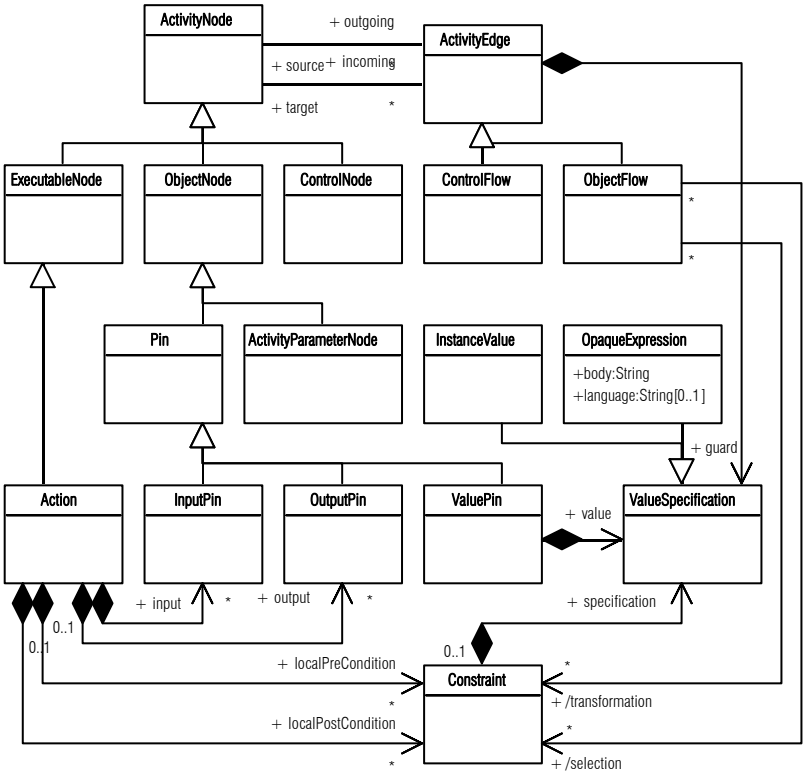
Table 6-2 illustrates the representation relation between the enterprise viewpoint relations from Figure 6-2 and the UML 2.0 modelling elements that we use to represent these relations.

A UML association represents a general form of relation. We use it to represent most of the relations from Figure 6-2. We specialize the associations with stereotypes, as indicated in Table 6-2, to represent specific rela-

tions between the enterprise viewpoint concepts. We represent some of the relations by aggregation associations to indicate that the concept instance on the aggregate side of the aggregation *contains* the concept instance on the other side of the aggregation. We represent the relation between a community object and the community of which it is an abstraction, by the community object and the community having the same name. Similarly, we represent the relation between a step template and the action template of which it is an abstraction, by step template and the action template having the same name. We represent that a process template contains a step template by the activity that represents the process template containing a call to the activity that represents the step template. We represent that a behaviour template contains an action template in the same way. We represent that an enterprise object is referenced as an artefact of an action template, by representing the enterprise object as a parameter of the activity that represents the action template.

Chapter 3 explains the graphical representation of MOF classes and associations, which is the same notation used for UML classes and associations. A UML actor is graphically represented as a puppet. We explain the graphical representation of UML activities in the next subsection.

Figure 6-3 UML Activity Diagram Concepts



6.2.3 Representing Constraints with UML Activities

By representing behaviour, action, process and step templates as UML activities, we inherit the concepts from UML activities to represent behaviour constraints. Figure 6-3 shows some of the concepts from UML activities. We restrict ourselves to these concepts for representing behaviour constraints. UML allows for a lot of freedom in the notation and the use of these concepts. Below, we use them in a more restrictive way. Addition of other concepts and loosening restrictions is left for future work.

We use UML flows and pre- and postconditions to represent behaviour constraints. *UML control flows* and *UML object flows* represent the flow of control and information, respectively, from one action in a system to another. We only use *UML call behaviour actions* (which in our case represents a step template of an enterprise action template). A flow passes tokens from one action to the other. In a control flow, the flow of tokens represents that the action at the source of the flow enables the action at the target of the flow. In an object flow, the flow of tokens represents the flow of information from one action to the next. Flows constrain behaviour, because an action can only start if all its incoming flows have a token on them. If an action completes, it puts a token on all its outgoing flows. An object flow must connect an output pin or activity parameter node to an input pin or activity parameter node. An *output pin* represents the output information of an action, while an *input pin* represents the input information of an action. Pins have a name and a type. If the action is a UML call behaviour action (an action that represents the execution of some behaviour, such as a UML activity), the input and output pins must correspond to the parameters of the called behaviour. A *value pin* is a special kind of input pin, which represents a pre-defined input value. We can represent this value as an instance of a UML data type (e.g. String), in which case we represent it as a *UML instance value*. We can also represent this value using an expression in an arbitrary language, in which case we represent it as a *UML opaque expression*. We will use OCL for these purposes. An *activity parameter node* represents the input or output of the activity. If the activity is started via a call behaviour action, the input pins of the call behaviour action must correspond to parameter nodes of the called activity. The tokens that arrive on the input pins are passed to the activity parameter nodes. The output pins of the call behaviour action must correspond to output parameter nodes of the called activity. Tokens that arrive on the output parameter nodes are passed to the output pins.

An object flow can have a flow transformation that represents that the flow transforms the information value that is put on it. A flow transformation transforms some input into some output. The input of the flow is represented by the output pin that is the source of the flow, while the output

of the flow is represented by the input pin that is the target of the flow. An OCL constraint is associated with the transformation. This constraint represents the postcondition of the transformation behaviour. The OCL constraint can refer to the input and output pins that the flow connects, as if they were variables with their name and type specified by the pins. Similarly, a selection represents that only particular values can be put on a flow.

The activity that an action represents can also be specified in more detail, by representing the pre- and postconditions of that action in OCL. A precondition represents a condition on the input of an action that must be satisfied before the action can be performed. A postcondition represents a condition that the output of the action satisfies.

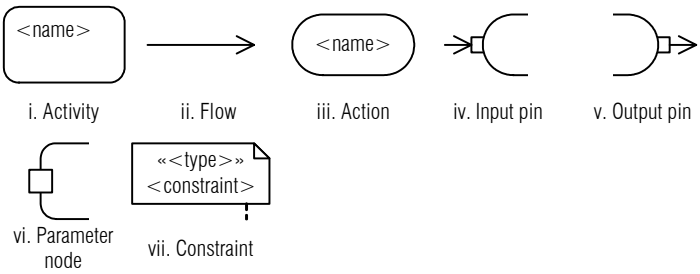


Figure 6-4 Graphical Notation for UML Activities

Figure 6-4 shows how we graphically represent UML activities. We represent an action or an activity as a rounded rectangle with the name of the action or activity inside it. In case of a call behaviour action, the name of the action is the name of the behaviour (e.g. the activity) that is called. We represent a flow as an arrow. We represent input and output pins as blocks on an action that only have incoming or outgoing flows, respectively. The value of a value pin must be represented close to the pin. Similarly, we represent input and output parameter nodes as blocks on an activity that only have outgoing or incoming flows, respectively. We represent constraints inside a comment box that is attached to the model element that it constrains. The comment box is stereotyped with the type of constraint that it represents (local pre- or post-condition, selection, or transformation).

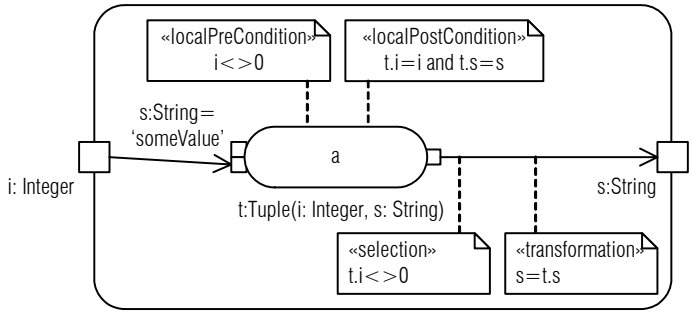
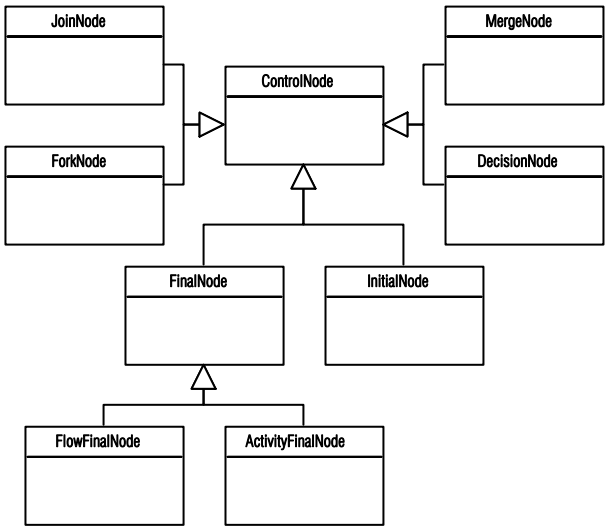


Figure 6-5 Example of a UML Activity

Example 6-1 A UML Activity

Figure 6-5 shows a UML activity with an input parameter of type integer and an output parameter of type string. An object flow from the input parameter node to an input pin of action 'a' passes a token that represents an integer value to 'a'. The precondition of 'a' defines that the value should not be zero. 'a' has a value pin that is assigned the string 'someValue'. The postcondition of 's' ensures that the output value of 'a' is a tuple with an integer and a string value that are equal to the inputs of 'a'. The object flow that leaves the output pin of 'a' accepts only tokens that represent a tuple with an integer value that is not zero, as specified by the selection of the object flow. The transformation of the object flow ensures that a token that only contains the string value of the tuple is delivered to the output parameter.

Figure 6-6 Control Nodes

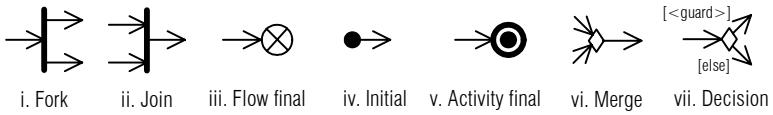


We can represent more advanced behaviour constraints, using control nodes. A control node is a node that can be the source and target of flows and that affects the tokens on these flows in a pre-defined manner. Figure 6-6 shows the different kinds of control nodes that exist. Their ways of affecting tokens are as follows:

- if a token is put on the incoming flow of a fork node, it removes that token and puts tokens (representing the same value) on all of its outgoing flows. Figure 6-7.i shows the graphical representation for a fork node;
- if a join node has tokens on all its incoming flows, it removes those tokens and puts one token on its outgoing flow. Possible information values are combined. However, this is outside the scope of this thesis. Figure 6-7.ii shows the graphical representation for a join node;
- a flow final node removes all tokens that arrive on its incoming flows. Figure 6-7.iii shows the graphical representation for a flow final node;

– an initial node initially has tokens on all its outgoing flows. Figure 6-7.iv shows the graphical representation for an initial node;
– an activity final node removes all tokens from the activity if a token arrives on one of its incoming flows. Figure 6-7.v shows the graphical representation for an activity final node;
– if a token is put on an incoming flow of a merge node, it removes that token and puts a token (representing the same value) on its outgoing flow. Figure 6-7.vi shows the graphical representation for a merge node;
– if a token is put on the incoming flow of a decision node, it removes that token and puts a token (representing the same value) on the outgoing flow of which the value satisfies the attached OCL constraint (also called guard). A special guard, named 'else', exists that is only satisfied if none of the other constraints are satisfied. No two guards should be satisfied at the same time. Figure 6-7.vii shows the graphical representation for a decision node.

Figure 6-7 Graphical Notation for Control Nodes

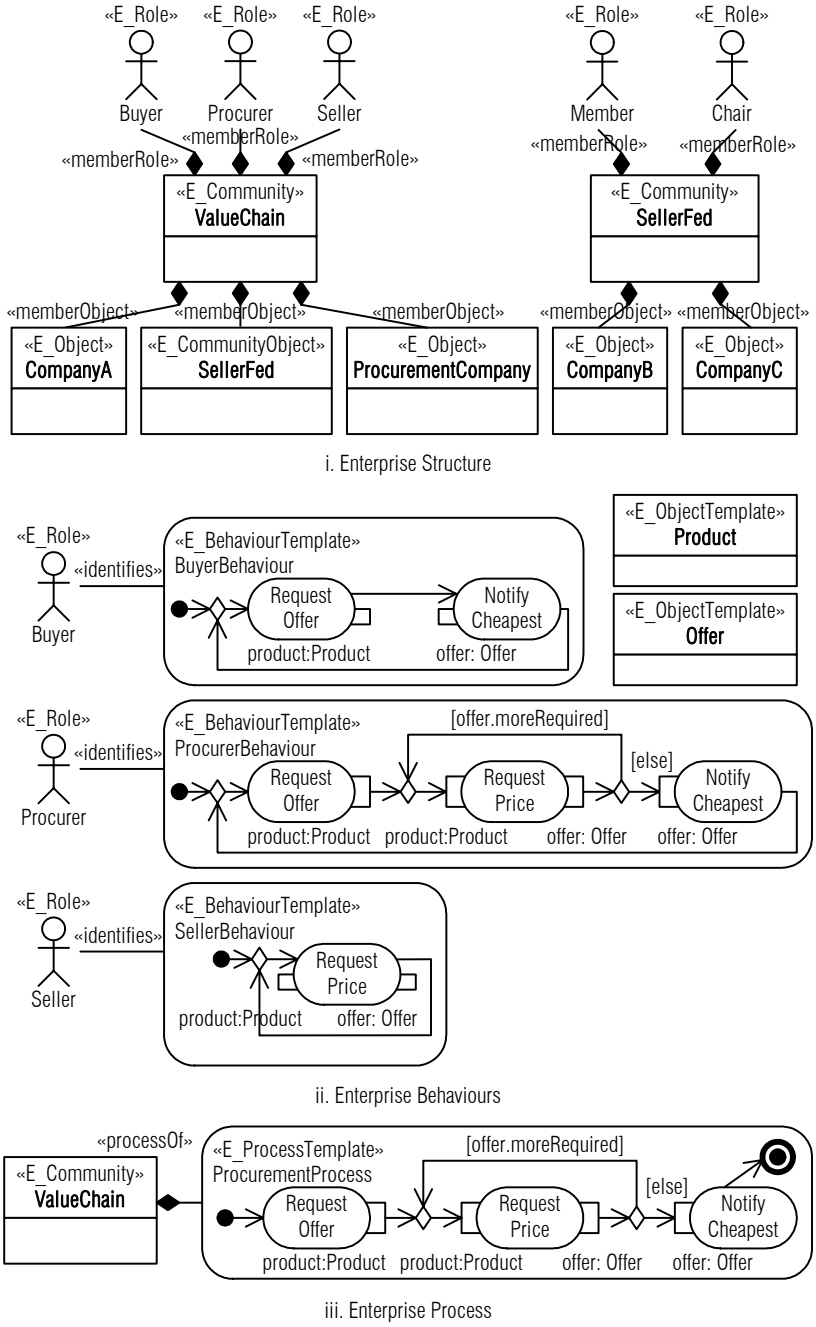


6.2.4 Example of an Enterprise View

Figure 6-8 shows an example of an enterprise view that uses the concepts from subsection 6.2.1 and the notation from subsection 6.2.2. Figure 6-8.i shows an enterprise community that contains three enterprise objects and three roles. One of the objects is a community object that is further detailed as a community. Figure 6-8.ii represents the behaviour templates identified by the roles in the 'ValueChain' community as activity diagrams. The 'Product' and 'Offer' enterprise object templates that are used by the action templates are represented as parameters to the call behaviour actions. Hence, their instances represent artefacts that are used by the actions represented by the action templates. The 'RequestOffer' and 'NotifyCheapest' behaviour call actions appear both in the 'BuyerBehaviour' and in the 'ProcurerBehaviour', indicating that they represent interactions between behaviours of those templates. Similarly, the 'RequestPrice' behaviour call action appears both in the 'ProcurerBehaviour' and the 'SellerBehaviour', indicating that it represents an interaction between behaviours of those templates. Figure 6-8.iii shows the business process template that is part of the 'ValueChain' community. The step templates bear the same names as the action templates from the behaviours, indicating that they represent these action templates. Figure 6-8.iv shows which roles are authorized to perform steps of which templates, which enterprise objects are authorized to perform which roles and which enterprise object fulfils which role.

Figure 6-8.v illustrates the contributions of roles and processes to the goal for which the 'ValueChain' community exists.

Figure 6-8 An Enterprise View Represented in UML



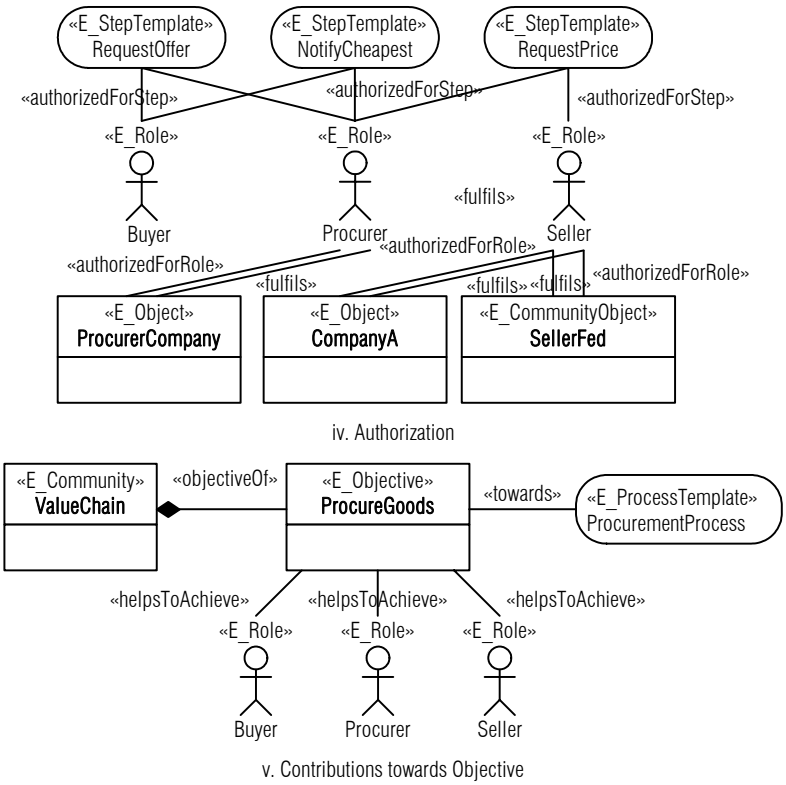


Figure 6-8 An Enterprise View Represented in UML (continued)

6.2.5 Relation of Enterprise Viewpoint Concepts to Basic Concepts

We define the relation between the enterprise viewpoint concepts and the basic viewpoint concepts in terms of a transformation. The transformation defines how a view in terms of enterprise viewpoint concepts can be transformed into a view in terms of basic concepts. We explain the transformation informally, i.e. in natural language. The role and objective concepts are not supported by the basic concepts, because basic concepts (or compositions thereof) that represent the same properties do not exist. Therefore, these concepts cannot be transformed into basic concepts.

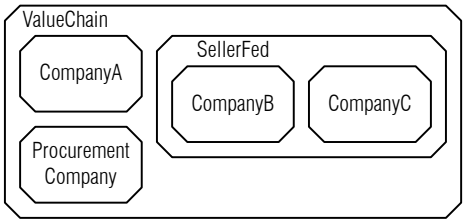


Figure 6-9 An Example of Enterprise Structure Transformation

Transformation of structural concepts. We transform communities and enterprise objects (that are not community objects) into entities. We transform the membership of an enterprise object to a community into an instantiation of the entity that represents the enterprise object by the entity that represents the community. Similarly, we transform the membership of a community object o (that represents some community c_1) to a community c_2 into the containment of the entity representing c_1 in the entity representing c_2. Figure 6-9 illustrates the transformation of the enterprise structure from Figure 6-8.i into basic concepts.

Since the entity model that is the result of the transformation does not contain interaction points, it cannot be considered a well-formed entity model. In future work, we must create rules to add interaction points to the basic model. However, for now we focus on the behaviour concern.

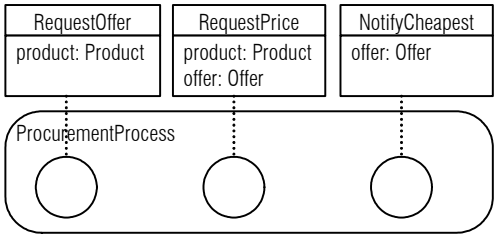


Figure 6-10 An Example of a Behaviour Template Transformation

Transformation of behaviour and process templates. We transform behaviour templates into basic behaviour types and internal action templates into basic action instantiations. We transform an interaction template into one basic interaction contribution instantiation for each behaviour template in which the interaction template is contained. A basic action instantiation or interaction contribution instantiation must be contained in the basic behaviour type that represents the behaviour template in which the corresponding internal action or interaction is contained.

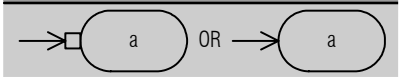
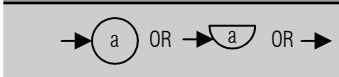
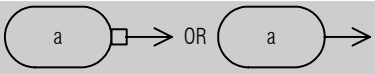
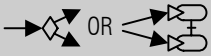
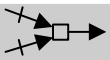
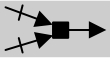
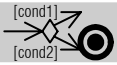
We transform process templates into basic behaviour types and step templates (that do not represent process templates) into basic action instantiations. We transform a step template that *does* represent a process template into an instantiation of the basic behaviour type that corresponds to that process template.

To transform an artefact of an action into an instance of a basic concept, we transform the UML pin that represents the artefact. We transform a pin into an information attribute with the same name and information type as the pin. If an attribute represents a value pin, we add an OCL constraint that states that the value of the attribute must be equal to the value of the value pin.

Figure 6-10 shows an example in which the process template from Figure 6-8.iii is transformed into a basic behaviour type. The type contains

the steps in the process as basic action instantiations. The pins that represent the parameters of the steps are transformed into information attributes of the corresponding basic action instantiations.

Table 6-3 Transformation of Activity Concepts to Basic Concepts

Activity Concept	Basic Concept
	
	
	
	
	
	
	
	Removed
	
	
	
	

Transformation of constraints. To transform constraints into basic concepts, we transform the UML flows and pre- and postconditions that represent these constraints. The transformation consists of the following steps:

1. transform flows and nodes into basic causality conditions;
2. rewrite invalid disjunctive conditions;
3. add guards, transformations, selections and pre- and postconditions.

Step 1. The first step in the transformation is the transformation of flows and nodes into basic causality conditions. Table 6-3 shows this transformation. The transformations are the following:

– We transform a flow to an UML action into a causality condition of the corresponding action instantiation, interaction contribution instantiation or behaviour instantiation. If multiple flows point to a UML action, the causality condition of the corresponding basic concept is the con-

junction of the causality conditions into which the flows can be transformed.

- We transform a flow from an UML action into a basic enabling condition on the corresponding action, interaction contribution or behaviour instantiation, because a flow from an UML action enables the execution of its target after the UML action has completed.
- We transform a flow from an initial node into a start condition, because each flow from an initial node initially gets a token, such that its target can be performed (depending on other flows that have the same target).
- We do not transform a flow to a flow final node, because the flow final node does not affect the execution of any other nodes.
- We transform flows to a join node into the conjunction of the conditions represented by those flows, because the join node enables its outgoing flow if all its incoming flows are enabled.
- We transform a flow to a fork node into the corresponding basic condition targeting each of the basic concept instances that the flow points to. We can do this, because a join node enables each of its targets when it is enabled.
- We either transform a decision between some UML actions into a basic choice between the corresponding targets or into the enabling of newly created behaviour instantiations (and other targets) that mutually disable each other. We prefer to use the first construct, because it is easier. However, the second construct must be used in the case where one (or more) of the flows point to a fork node. This is because, in this case, using the basic choice would cause the actions that are the target of the fork node to mutually disable each other (as illustrated in Figure 6-11), while they should not. If a decision is part of a loop (it (implicitly) depends on itself), actions and interaction contributions that are part of that loop (that (implicitly) depend on the decision and on themselves) are enabled by the condition of the decision. However, in this case, we do not add mutual disabling relations between these actions and interaction contributions. We will do that in the next step. Figure 6-12 illustrates this case. The decision node in the figure indirectly depends on itself. Hence, it is part of a loop. Action *a* depends the decision and on itself and therefore is part of the same loop. Therefore, we do not add the mutual disabling between *a* and the behaviour that contains *b* and *c*.
- We transform flows to a merge node into a disjunction between the corresponding basic conditions. Note, however, that, while a disjunction only enables the execution of its target once, the merge can enable the execution of its target once *for each flow pointing to it*. We fix this discrepancy between the semantics of the two concepts in step 2.
- Regarding activity final nodes, we focus on the case in which the activity final node only receives a token if no other tokens exist in the activity

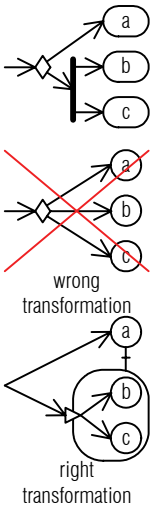


Figure 6-11 Transformation of Decision Nodes

anymore. In this case, the activity final node represents a milestone (the completion of the activity) and does not affect the execution of other parts of the activity. Therefore, we do not transform it into a basic causality condition.

Step 2. The second step in the transformation is rewriting disjunctions that were transformed from a merge node through which more than one token could flow. This is the case if:

1. tokens can be put on more than one of its incoming flows, for example, because these flows (indirectly) all depend on an initial node; or
2. the merge node is part of a loop, such that a token can be put on it, after a token was already processed by it once. Figure 6-12 illustrates a merge node for which this is the case.

In the basic concepts, an action, interaction contribution or behaviour can only be repeated if the behaviour type of which it is a part is instantiated more than once. Therefore, we change the disjunction that is transformed from a merge through which multiple tokens flow as follows:

1. We remove the disjunction and everything that causally depends on it from its behaviour type and put it in a new behaviour type.
2. On that behaviour type we add an entry point for the disjunction. The condition of the entry point is the condition of the disjunction. Also, we add an entry point for each disjunction that both causally depends on the first disjunction and on which the first disjunction depends. Their conditions are the conditions of the disjunctions for which they are created. We add these entry points because the merges from which they (their disjunctions) are transformed are part of the same loop as the first disjunction. Therefore, they must be part of the same instantiation.
3. We replace each alternative condition of the original disjunction(s) by an instantiation of the new behaviour type, such that the condition of the entry point of that instantiation equals the alternative condition.
4. In step one we did not add disabling relations to actions and interaction contributions that were part of a loop when transforming a decision node. Now that we have transformed the loop, we can do that.

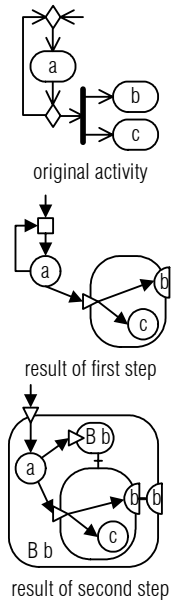


Figure 6-12 Transformation of Loops

Figure 6-12 illustrates this step. For the disjunction that was transformed from the merge that caused the repeated occurrence of *a*, create an entry point of a new behaviour type. This behaviour can both be instantiated by the start condition and by the occurrence of *a*. The recursive instantiation of the behaviour and the behaviour that contains *b* and *c* mutually disable each other, such that, after *a* has occurred once, there is a choice between executing the behaviour that contains *b* and *c*, or repeating the entire behaviour.

If interaction contributions are put inside a new behaviour type, they must be added as structured interaction contributions to the behaviour type

that contains one or more instantiations of the new behaviour type. The interaction contribution of each contained behaviour instantiation is an alternative interaction contribution for the structured interaction contribution. Figure 6-12 illustrates this for interaction contribution *b* that must be added as a structured contribution to the overall behaviour.

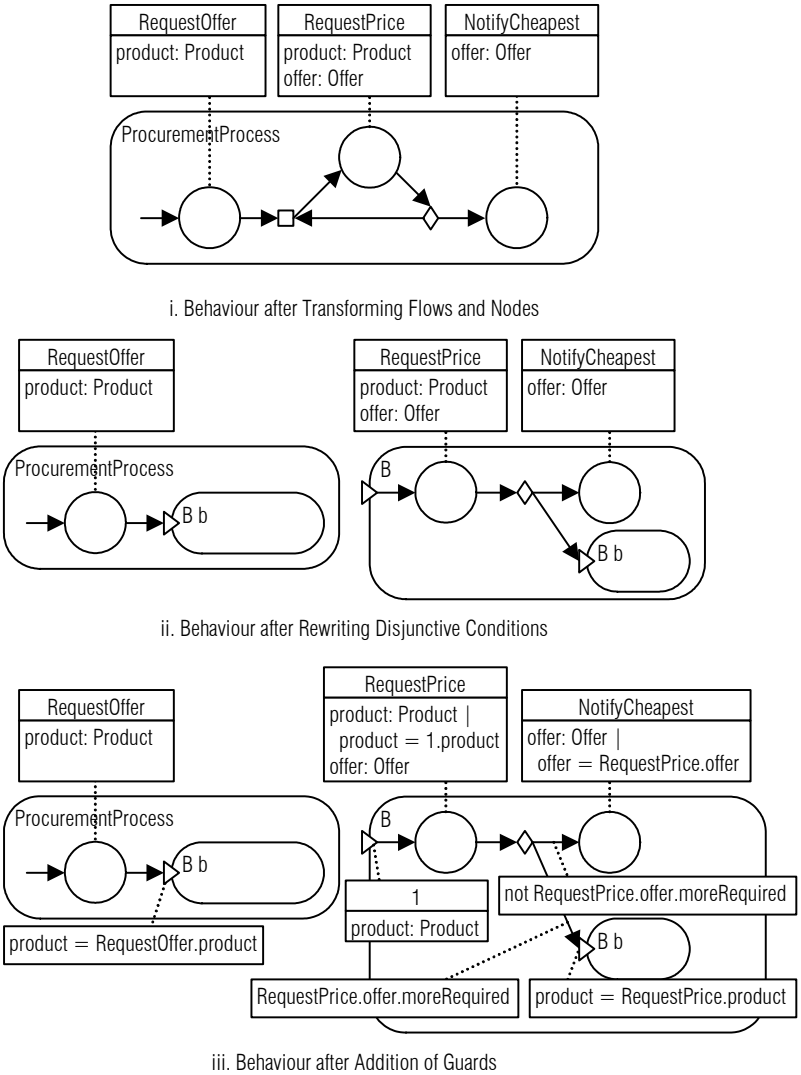
Also, if actions or interaction contributions are put inside a new behaviour type, they may causally depend on actions or interaction contributions outside the behaviour as a result. Causality relations between actions or interaction contributions outside and inside a behaviour must be split up using point conditions.

Step 3. The third step in the transformation is the addition of guards, transformations, selections and pre- and postconditions into causality and attribute constraints. We transform a guard or a selection on a flow into a basic causality constraint on the corresponding basic condition. Assuming that guards and the selections are represented as OCL constraints and that the UML binding to the basic concepts is used, the basic constraint is equal to the guard or selection from which it must be transformed. An 'else' guard must be transformed into a constraint that is the 'not' of the disjunction of the conditions on the other flows that leave the decision node. We must do this, because the causality condition that corresponds to the flow with the 'else' guard is enabled if none of the other flows are. We transform a transformation on a flow into a constraint on the attribute that represents the input pin to which the flow points. We transform pre- and postconditions into constraints on the corresponding attribute. An object flow also represents a constraint, in that the value of the pin that the flow points to must be the same as the value of the pin that the flow comes from. Therefore, we transform an object flow into an equivalence constraint on the attributes that correspond to the pins that the flow connects.

We replace an object that is referenced by the name of the pin, by <the name of the action of which the object is a result>.<the name of the pin>, because the basic concepts require an information value to be referenced by <the name of an action or interaction contribution>.<the name of an attribute of that action or interaction contribution>.

A constraint can reference an attribute from another behaviour, as the result of a disjunction being transformed into an entry point of another behaviour. In that case, we must pass the value of the attribute as a parameter of the entry point. The parameter of the entry point has the same name and type as the attribute and the constraint that its value must be equal to the value of the attribute. Also, the constraint that references the attribute must reference the parameter from the entry point instead.

Figure 6-13 An Example of Constraint Transformation



Example 6-2 Constraint Transformation

Figure 6-13 shows an example in which the constraints from the process template from Figure 6-8.iii are transformed into basic behaviour constraints. Figure 6-13.i shows the result of the first step, in which UML flows are transformed into causality conditions according to Table 6-3.

Figure 6-13.ii shows the result of the second step, in which the disjunctive condition of the 'RequestPrice' action instantiation is transformed into two behaviour instantiations of a new behaviour type. We added an entry point to the new behaviour type for the disjunction. The behaviour type contains all actions that depended on the disjunction. Both alternative conditions of the disjunction ('RequestOffer has occurred' and 'RequestPrice has occurred, but NotifyCheapest has not yet occurred') are transformed into conditions for instantiations of the new behaviour type.

Figure 6-13.iii shows the result of the third step, in which the guard conditions are

transformed. We transformed the 'else' guard into the 'not' of the disjunction of the guards on the other flows (offer.moreRequired). We prefixed 'offer' with 'RequestPrice', because that is the way in which we must refer to the result of an action or interaction. The 'offer' attribute value must be equal for the 'RequestOffer' and 'RequestPrice' action instantiations and the 'product' attribute value must be equal for the 'RequestPrice' and 'NotifyCheapest' action instantiations, because the corresponding attributes were originally passed as objects in the UML activity.

Transformation into behaviour instantiations. We transformed enterprise processes and behaviours into basic behaviour types. However, behaviour types cannot be performed. Only their instances can be. Therefore, we must create instances of the behaviour types that represent enterprise processes and behaviours.

Multiple instances of a process can be active in an enterprise at the same time. However, since a process is only aimed at representing the activities performed to achieve a certain goal, it does not consider the relations between these instances. Therefore, we cannot construct a complete design with multiple instantiations of the same process. However, as we see later, creating multiple instantiations is not necessary for verifying conformance, we create only a single instantiation for each process.

Each enterprise object that fulfils an enterprise role, performs a behaviour of the type identified by that role. Therefore, in each community, we create an instantiation of the basic behaviour represented by a role, for each enterprise object fulfilling that role. We create an interaction between (interaction contributions of) the behaviour instantiations for each enterprise interaction. We assume that, if multiple instantiations of the same behaviour type exist, *any* of these instantiations can contribute to an interaction. Hence, each instantiation adds an alternative interaction contribution to the interaction.

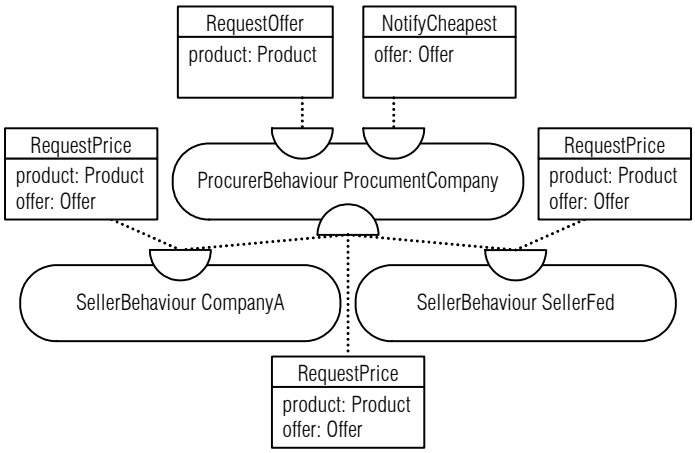


Figure 6-14 An Example of Transformation into Behaviour Instantiations

Example 6-3 Transformation into Behaviour Instantiations

Figure 6-14 illustrates the transformation of enterprise objects that fulfil roles according to Figure 6-8.iv into instantiations of the basic behaviour types identified by those roles. The figure shows one instantiation of the behaviour associated with the 'Procurer' role, because there is only one object that fulfils that role. There are two instantiations of the behaviour associated with the 'Seller' role, because there are two objects that fulfil that role.

The 'RequestPrice' interaction can be performed by the 'ProcurementCompany' in the 'Procurer' role and either 'CompanyA' in the 'Seller' role or the 'SellerFed' in the 'Seller' role. Therefore, the 'RequestPrice' basic interaction has two alternatives that correspond to these two options.

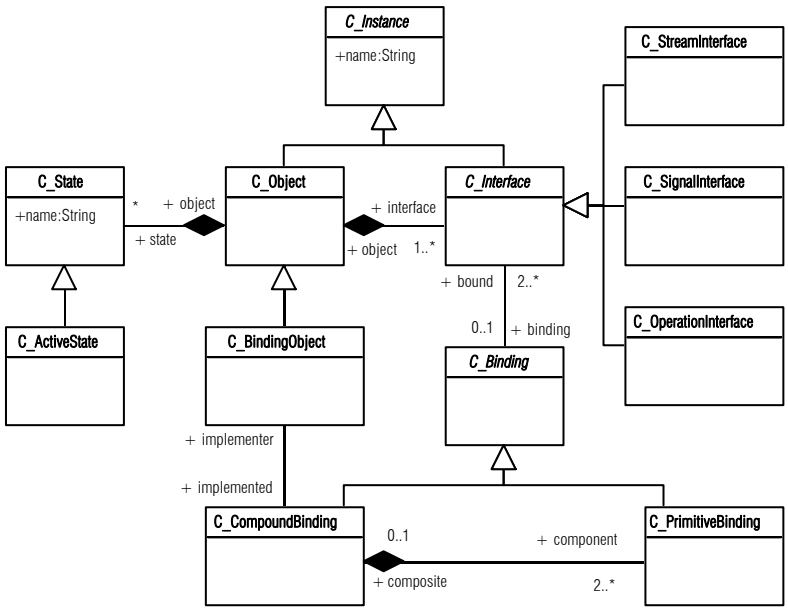
6.3 Computational Viewpoint

The computational viewpoint is used to design a functional decomposition of the system. This is the concern of a stakeholder that wants to have an overview of the constituents of the system at a level of abstraction above the software implementation. To represent the functional decomposition of the system, the computational viewpoint addresses the following concerns:

- the structure of the system;
- the behaviour of (the parts of) the system; and
- contracts that govern the behaviours of the parts and their interconnections.

The computational viewpoint assumes that the design will be implemented, using some object-oriented or component-based middleware to implement

Figure 6-15 Conceptual Model for Computational Instance Concepts



the interactions between the system parts. Hence, it represents the system at a level of abstraction at which such interactions must be considered.

6.3.1 Computational Viewpoint Concepts

As a conceptual model for the computational viewpoint, we use an adapted version of the conceptual model from (ITU-T, & ISO/IEC, 2005). We adapted the conceptual model syntactically in the same way as we adapted the conceptual model for the enterprise viewpoint. We separated the conceptual model into three diagrams:

- The computational instances diagram (Figure 6-15) that incorporates the concepts that represent the system parts and their properties;
- The computational templates diagram (Figure 6-16 and Figure 6-17) that incorporates the concepts that represent templates according to which instances can be constructed; and
- A diagram that relates the instance concepts to the template concepts (Figure 6-18).

The distinction between the computational templates and computational objects is also made in (ITU-T, & ISO/IEC, 2005) and (Akehurst, et al., 2003). To make the distinction more strict than in (ITU-T, & ISO/IEC, 2005), we added the behaviour specification and contract specification concepts, as the template counterparts of behaviour and contract. We also included a relation between those concepts and the action template and constraint template concepts, representing that the behaviour and contract specification templates contain the related action and constraint templates.

A computational view consists of a configuration of *computational objects*. A computational object has a behaviour that is determined by the template according to which it is instantiated. The behaviour of an object is defined by the actions that the object can perform, possibly in collaboration with other objects, and the constraints on the occurrence of those actions. A computational object also has a state. The *state* of an object is the mode that determines the actions in which an object can take part from a certain moment in time. The state of an object can change when it performs an action. We consider the state of an object to be implicitly defined by the behaviour of that object and the actions that an object has performed, along with the result that was established during these actions. We can do this, because the behaviour of an object and (the results of) the actions that an object has performed, determine which actions it can take part in next. We call the current state of an object the *active state*.

An object has interfaces. An *interface* is an abstraction of the behaviour and environment contract of an object. It is an abstraction by considering only the interactions that an object may have at a particular binding. We explicitly added the relation between the object and interface concept,

which was not contained in (ITU-T, & ISO/IEC, 2005). We distinguish three different kinds of interfaces:

1. The *stream interface*, which is an interface that contains only interactions that are abstractions of a sequence of interactions. Such interactions are called *flows*.
2. The *signal interface*, which is an interface that contains only interactions that represent an atomic shared activity between two objects. Such interactions are called *signals*.
3. The *operation interface*, which is an interface that contains only interactions that follow the request/response pattern. Such interactions are called *operations*. We consider an operation as consisting of a request,

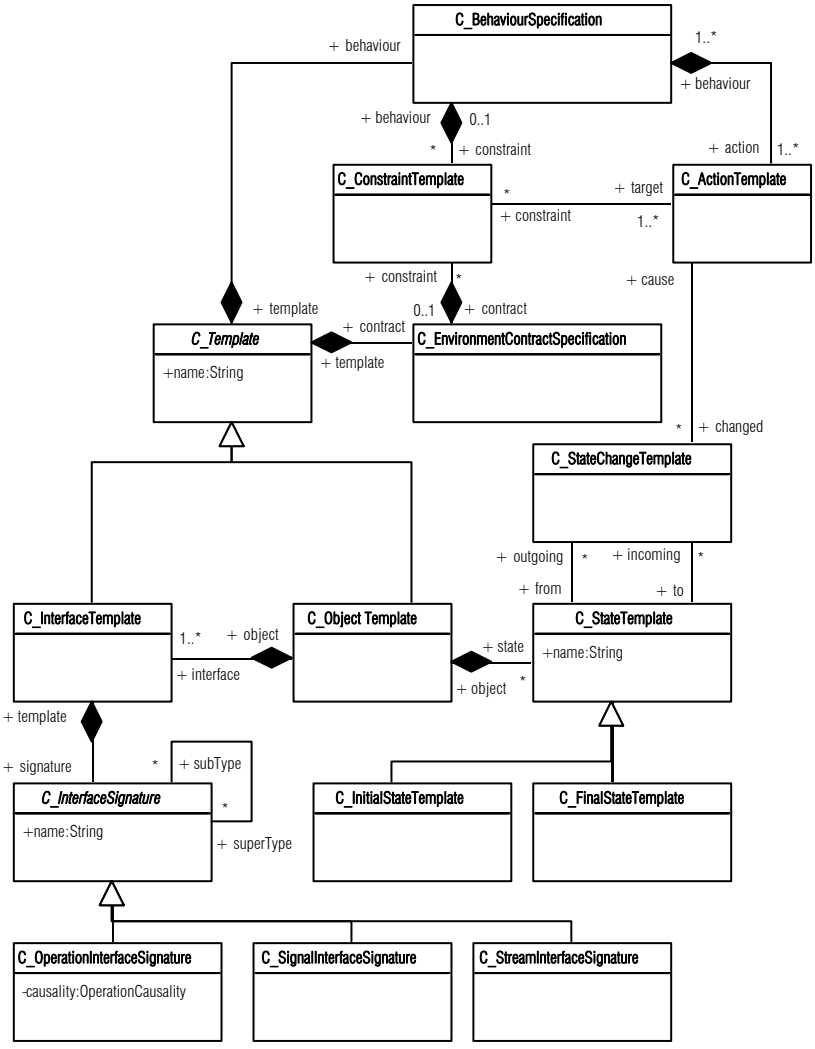


Figure 6-16 Conceptual Model for Computational Template Concepts

which we call *invocation*, and optionally a response, which we call *termination*. In case an operation has a response we call it an *interrogation*, otherwise we call it an *announcement*.

A *binding* between interfaces represents a contractual context in which joint activities of objects can occur. A joint activity cannot occur between two objects if no binding exists between (interfaces of) those objects. We distinguish between a *primitive binding* that binds only two interfaces and a *compound binding* that can bind any number of interfaces. A compound binding can be represented by a computational object, a so-called *binding object*, and primitive bindings between the binding object and the objects that are bound by the compound binding. In that case we also say that the computational object implements the compound binding.

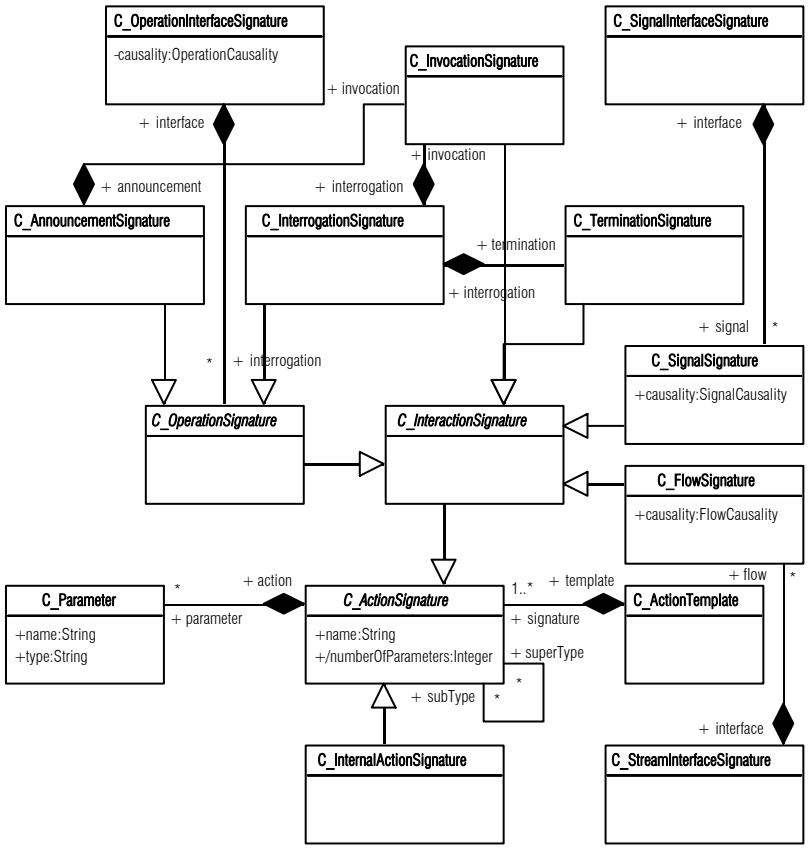


Figure 6-17 Conceptual Model for Computational Template Concepts: Interface Signatures

The conceptual model from Figure 6-16 and Figure 6-17 shows the concepts that represent templates according to which computational objects can be created. Computational objects are created according to a computational object template. Their actions are created according to action tem-

plates. The behaviour of a computational object conforms to the behaviour specification of the corresponding object template, and so on. The behaviour of an object is governed by an *environment contract*, which is an agreement that governs the collective behaviour of that object and objects that want to interact with it. The contract consists of constraints, such as Quality of Service constraints, on interactions between the object and its environment. The environment contract specification specifies the environment contract that an object of the associated template must observe.

The state and state change template concepts represent the states that an object of a particular template can be in, the actions that it can perform in this state and the states that an object can change into upon performing an action, respectively. The initial state and final state templates represent the state that an object is in directly after its creation and the state in which it has completed its behaviour, respectively.

Signatures represent additional information for templates to create instances. An action signature represents the name of the action that must be created, as well as *parameters* that represent the information that is established in an action. Each interface in which the action can appear has its own signature of that action. Different kinds of action signatures exist that correspond to the different kinds of actions that can exist in a computational object. Flow action signatures have a causality that represents whether the object that owns the interface associated with the flow is the producer or the consumer of the flow. Signal action signatures have a causality that represents whether the object that owns the interface associated with the signal is the initiator or the responder of the signal. An interface signature represents the name of the interface that must be created and the actions that it must contain. Different kinds of interface signatures exist that correspond to the different kinds of interfaces that a computational object can have. An operation interface signature has a causality that represents whether the object's interface is the server, such that it responds to the operation calls at the interface, or the client, such that it initiates the operation calls at the interface. Although (ITU-T, & ISO/IEC, 2005) represents operation causality as a property of the operation concept, we added operation causality as an attribute of the operation interface concept, because the RM-ODP computational viewpoint prescribes that.

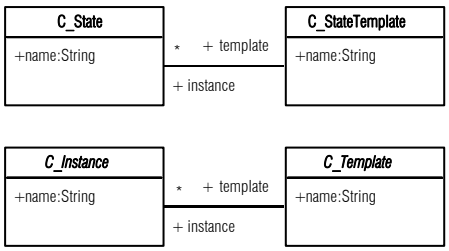


Figure 6-18 Relations between Computational Instances and Templates

Figure 6-18 represents the relations between the computational instances from Figure 6-15 and the computational templates from Figure 6-16.

Table 6-4 Representation of Computational Viewpoint Concepts in UML 2.0

Computational Viewpoint Concept	UML 2.0 Modelling Element
C_Object	Component Instance stereotyped C_Object
C_SignalInterface	Port with one or two Interfaces stereotyped C_SignalInterface
C_OperationInterface	Port with Interface stereotyped C_OperationInterface
C_BindingObject	Component Instance stereotyped C_BindingObject
C_PrimitiveBinding	One Assembly Connector stereotyped C_PrimitiveBinding if binding operation interfaces, two Assembly Connectors if binding signal interfaces
C_CompoundBinding	Implicitly represented by a binding object
C_ObjectTemplate	Component stereotyped C_ObjectTemplate
C_InterfaceTemplate	Implicitly represented by C_InterfaceSignature
C_OperationInterfaceSignature	Port with required Interface stereotyped C_OperationInterfaceSignature if the causality is 'client', Port with provided interface stereotyped C_OperationInterfaceSignature if the causality is 'server'
C_SignalInterfaceSignature	Port with at most one provided and at most one required Interface stereotyped C_SignalInterfaceSignature
C_ActionTemplate	Implicitly represented by C_ActionSignature
C_AnnouncementSignature	Operation stereotyped C_AnnouncementSignature
C_InterrogationSignature	Operation stereotyped C_InterrogationSignature
C_TerminationSignature	Implicitly represented as part of C_InterrogationSignature
C_InvocationSignature	Implicitly represented as part of C_OperationSignature
C_SignalSignature	Signal stereotyped C_SignalSignature
C_InternalActionSignature	Implicitly represented by a CallBehaviourAction in a C_BehaviourSpecification
C_Parameter	A parameter of an Operation, an Attribute of a Signal, a Parameter of a CallBehaviourAction or an Attribute of a Class that represents the return type of an Operation.
C_BehavourSpecification	Activity stereotyped C_BehaviourSpecification
C_EnvironmentContractSpecification	Class stereotyped C_EnvironmentContractSpecification
C_ConstraintTemplate	Flow in Activity and localPrecondition/localPostcondition of Action

6.3.2 Representation of Computational Views

Like for enterprise views, we use UML 2.0 to represent computational views by means of models. We deviate from the representation explained in

(ITU-T, & ISO/IEC, 2005), because this standard prescribes a way to represent computational views with the EDOC profile for UML (Object Management Group, 2002d), rather than with UML itself. This approach means that people have to learn another profile, while this is not necessary. Blair and Stefani (1998) also developed a notation for the computational viewpoint. However, they use a proprietary modelling language. Romero and Vallecillo (2005) developed their profile for modelling the RM-ODP computational viewpoint with UML 2.0 at the same time as we developed ours.

Table 6-4 illustrates the representation relation between the computational viewpoint concepts and UML 2.0 modelling elements. We stereotype the UML modelling elements, as indicated in Table 6-4, to make them identifiable as computational viewpoint concepts. We do not address stream interfaces and flows, because there exist no modelling elements in UML 2.0 that have a semantics that matches these concepts. Representing streams and flows is left for future work. Also, we do not explicitly represent the states of an object, because we consider state to be defined implicitly by the behaviour of the objects.

We represent computational objects and binding objects by UML component instances, their interfaces by UML ports with required and provided UML interfaces and we represent primitive bindings by UML assembly connectors. We represent a signal interface by (at most) two UML interfaces: a required interface and a provided interface. We must do this, because a required UML interface only sends signals and a provided UML interface only receives signals, while a computational signal interface can both send and receive signals. A compound binding is completely represented by its binding object.

We represent the structural concepts of the computational template concepts, as well as their relations, by a UML component diagram. We represent an object template by a UML component. We represent an operation interface by a UML port with a provided UML interface if the interface has server causality and a required UML interface if the interface has client causality. We represent a signal interface by a UML port with at most one provided and at most one required UML interface, for reasons discussed above. If multiple instances of an interface template can be attached to a single object, we represent this by setting the multiplicity of the UML ports accordingly. UML ports do not represent all properties of computational interfaces, because computational interfaces can be created and destroyed at any time, while UML ports can only be created or destroyed along with the UML component to which they belong (Bordbar, Derrick, & Waters, 2002). However, since we do not consider the dynamics of a structure further on in this thesis, we do not consider this a problem. Also, UML port 'types' do not exist. Therefore, if we use ports to represent computational

interfaces, the name of a computational interface is equal to the name of its template.

We represent operations by UML operations. We represent the termination of an interrogation as the reply of a UML operation. We represent the parameters of the invocation of an operation as the parameters of the UML operation and we represent the parameters of a termination as a UML class that contains UML attributes that correspond to the parameters. We represent a signal as a UML signal and the parameters of the signal as UML attributes of the UML signal. We represent that a signal has 'initiator' causality, by assigning it to a required UML interface. We represent that a signal has 'responder' causality, by assigning it to a provided UML interface.

Figure 6-19 Notation for the Computational Viewpoint

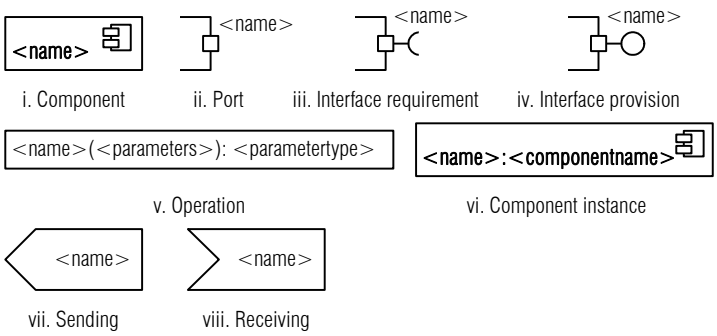


Figure 6-19 shows how UML graphically represents component concepts. A UML component is graphically represented as a box that carries the name of the component. A port is represented as a square on the border of a component. The name of the port must be drawn close to the square. An interface is represented as a UML class stereotyped 'interface'. If a component requires an interface this is denoted by the symbol shown in Figure 6-19.iii and the name of the required interface drawn close to the symbol. Similarly, a provided interface is denoted by the symbol shown in Figure 6-19.iv. A signal is represented by a class stereotyped 'signal'. An operation or signal on an interface is textually represented inside the box that represents the interface. A signal is represented by its name. An operation is represented by a name, followed by the parameters of the request, followed by the name of the parametertype of the response. The parameters of the request are denoted as: <name>:<typename>. A component instance is graphically represented in the same way as a component. However, the box contains the name of the component instance, followed by the name of the component.

Table 6-5 Representation of Computational Viewpoint Relations in UML 2.0

Computational Viewpoint Concept	UML 2.0 Modelling Element
C_Template owning C_BehaviourSpecification	Aggregation stereotyped behaviourOf
C_Template owning C_EnvironmentContractSpecification	Aggregation stereotyped contractOf
C_ActionTemplate referenced in C_BehaviourSpecification	Action referencing the corresponding Operation or Signal

Similar to enterprise behaviour templates, we represent computational behaviour specifications as UML activities and we represent their constraint templates by flows and pre- and postconditions. Table 6-5 shows how we represent the relations between the structural and behavioural computational viewpoint concepts. We represent the ownership of a behaviour specifications or environment contract by a UML aggregation. We represent that an action in a behaviour or contract by a UML action in the UML activity that represents the behaviour or contract. This UML action must represent the initiation of or the response to the UML operation or signal that represents the action's signature in the behaviour. Therefore, the following rules apply when representing actions:

- if the action's signature is a signal with 'initiator' causality it must be represented by a UML SendSignalAction;
- if the action's signature is a signal with 'responder' causality it must be represented by a UML AcceptEventAction;
- if the action's signature is an operation on an interface with 'client' causality it must be represented by a UML CallOperationAction;
- if the action's signature is an announcement on an interface with 'server' causality it must be represented by a UML AcceptCallAction;
- if the action's signature is an interrogation on an interface with 'server' causality it must be represented by a UML AcceptCallAction that corresponds to the invocation and a UML ReplyAction that corresponds to the termination.

SendSignalActions, AcceptEventActions, AcceptCallActions and ReplyActions have a special notation associated with them (other UML actions can be denoted using the regular action notation). SendSignalActions and ReplyActions are denoted as shown in Figure 6-19.vii. The box carries the name of the signal or operation response that is sent. Optionally, the name is prefixed with the name of the port on which the action occurs. Each SendSignalAction must have an input pin for each parameter of the sent signal. This pin must have the same name and type as the attribute. A ReplyAction must have an input pin for the return parameter of the operation. This pin must have the same type as the parameter. A ReplyAction must also have an input pin named 'returnInformation'. The data received on this pin must come from the pin with the same name on the corresponding AcceptCallAction. AcceptEventActions and AcceptCallActions are denoted

as shown in Figure 6-19.viii. The box carries the name of the signal or operation invocation that is received. Optionally, the name is prefixed with the name of the port on which the action occurs. An AcceptCallAction must have an output pin named 'returnInformation'. An AcceptCallAction must also have one output pin for each parameter of the invocation of which it represents the reception. Each pin must have the same name and type as the parameter to which it corresponds. An AcceptEventAction that represents the reception of a signal must have an output pin that represents the signal. This pin must have the signal as its type.

If we use UML, we cannot represent invocation and termination action templates on a 'client' interface by separate actions. Instead, these actions templates but must be represented by a single UML action that represents

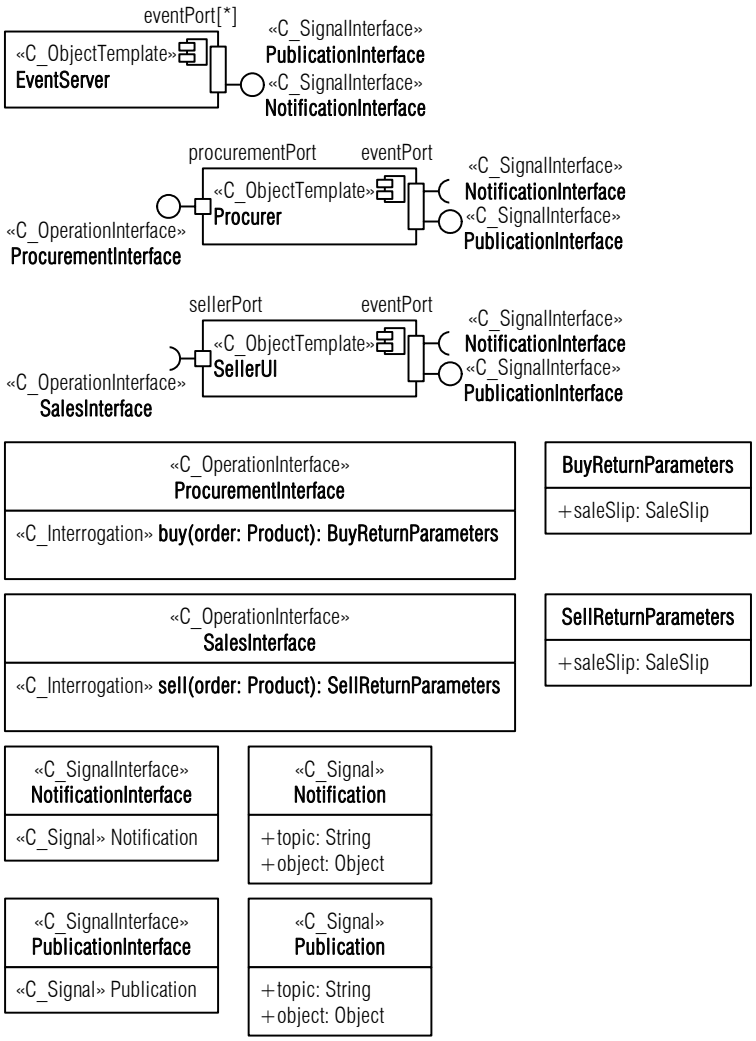


Figure 6-20 Structure of a Computational View Represented in UML

both the sending of the invocation and the reception of the termination. Similarly, we cannot represent an interrogation action template on a 'server' interface by a single UML action. Instead, we must represent an interrogation action template by two UML actions that represent the reception of the invocation and the sending of the termination.

We do not consider the representation of environment contracts in this thesis. This is left for future work.

6.3.3 Example of a Computational View

Figure 6-20 shows the structural aspects of the object templates in a computational view. It shows three object templates that can be used to create computational objects. The 'Procurer' object template is a template for computational objects that receive orders and select the best seller to fulfil each order. It has a 'ProcurementInterface' operation interface with 'server' causality that it uses to communicate with potential buyers. This interface has the interrogation 'buy' that has a single parameter 'order' with type 'Product' for the invocation and a single parameter 'saleSlip' of type 'SaleSlip' for the termination. The 'Procurer' object template has a signal interface at which it exchanges events. At this signal interface it can initiate 'Publication' signals that have a topic and an item on that topic. Also, it can respond to 'Notification' signals that also have a topic and an item on that topic. An event server ensures that events that are published via a 'Publication' signal are notified to computational objects that are subscribed to the topic. The event server notifies objects via a 'Notification' signal. To this end, it has multiple signal interfaces to which publishers and subscribers can bind. The subscription of computational objects to topics is outside the scope of this example. We assume that 'SellerUI' objects are subscribed to events on the 'order' topic and 'Procurer' objects are subscribed to events on the 'offer' topic. The 'SellerUI' object template represents a template to construct objects that represent the user interface for a seller.

Figure 6-21 shows the behaviour specifications that are attached to the 'Procurer' and 'EventServer' object templates, respectively (the associations that represent the relations between the object templates and the behaviour specifications are not shown). A procurer initially receives an invocation. The action that represents the reception of this invocation has a parameter of type 'Product'. The subsequent send signal action publishes the 'Product' object as an event on the 'order' topic. After the order is published, the procurer awaits a response in the form of an event on the 'offer' topic. In parallel, it creates an object that holds an empty offer and a counter (set to zero) that represents the number of offers received. When it receives an offer, the 'evaluate offers' action accepts the empty offer, the counter and the received offer. It then evaluates the best offer and increases the counter

by one. If the procurer has not yet received 3 offers, it awaits the next offer and compares this offer with the best offer received so far. If the procurer has received 3 offers, it sends the best offer via the procurement interface.

The event server receives a notification and, based on that, creates an object that consists of a publication and a counter. As long as the counter has not reached the number of subscribers to the topic, the publication is sent to the next subscriber, which is identified in the 'get target' action. Concurrently the counter is increased and the publication step is started again.

Figure 6-21 Behaviour of a Computational View Represented in UML

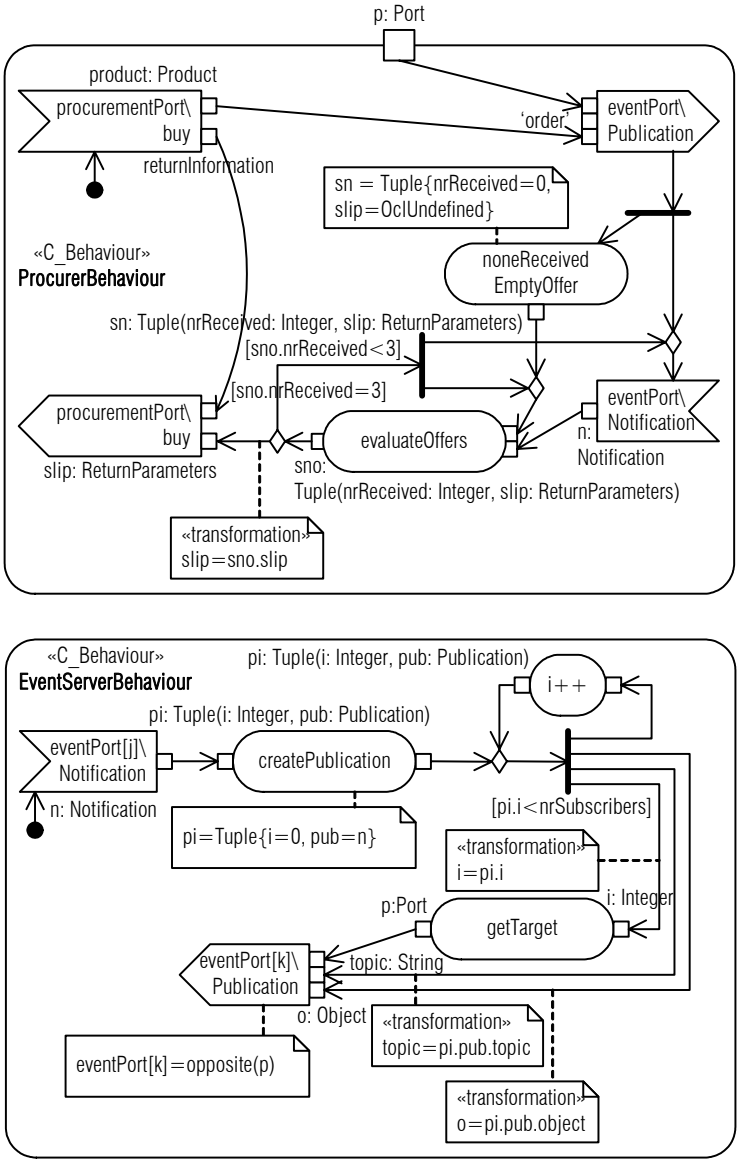


Figure 6-22 Instances of a Computational View represented in UML

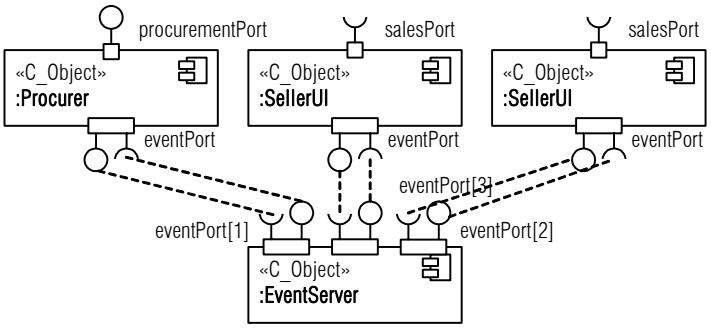
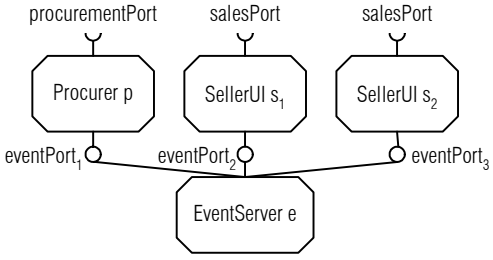


Figure 6-22 shows an example in which instances of the computational object templates from Figure 6-20 are shown. The figure shows one procurer, one event server and two seller user interfaces. To interact with the procurer as well as the sellers, the event server has three event ports. For brevity, the figure does not show the names of interfaces.

6.3.4 Relation of Computational Viewpoint Concepts to Basic Concepts

We define the relation between the computational viewpoint concepts and the basic viewpoint concepts in terms of a transformation. The transformation defines how a view in terms of computational viewpoint concepts can be transformed into a view in terms of basic concepts.

Figure 6-23 An Example of Instance Transformation



Transformation of instance concepts. We transform computational objects into entities and we transform computational interfaces into interaction point parts of those entities. We transform a primitive binding between two interfaces into an interaction point that consists of the interaction point parts that correspond to the bound interfaces. We transform the name of an object into the name of the corresponding entity. If the object does not have a name, we create (a unique) one. We transform the name of an interface into the location of the corresponding interaction point part. If a binding binds interfaces with different names, we must create a single unique location for the corresponding interaction point and interaction

point parts. We must do this because a binding can bind interfaces with different names, while the corresponding interaction point cannot have interaction point parts with different locations. If we change the name of an interface in this way, references to that interface in the behaviour must be changed accordingly. We do not transform a compound binding, but we transform the binding object that represents it.

Figure 6-23 shows the transformation of the computational instances from Figure 6-22 into basic concepts.

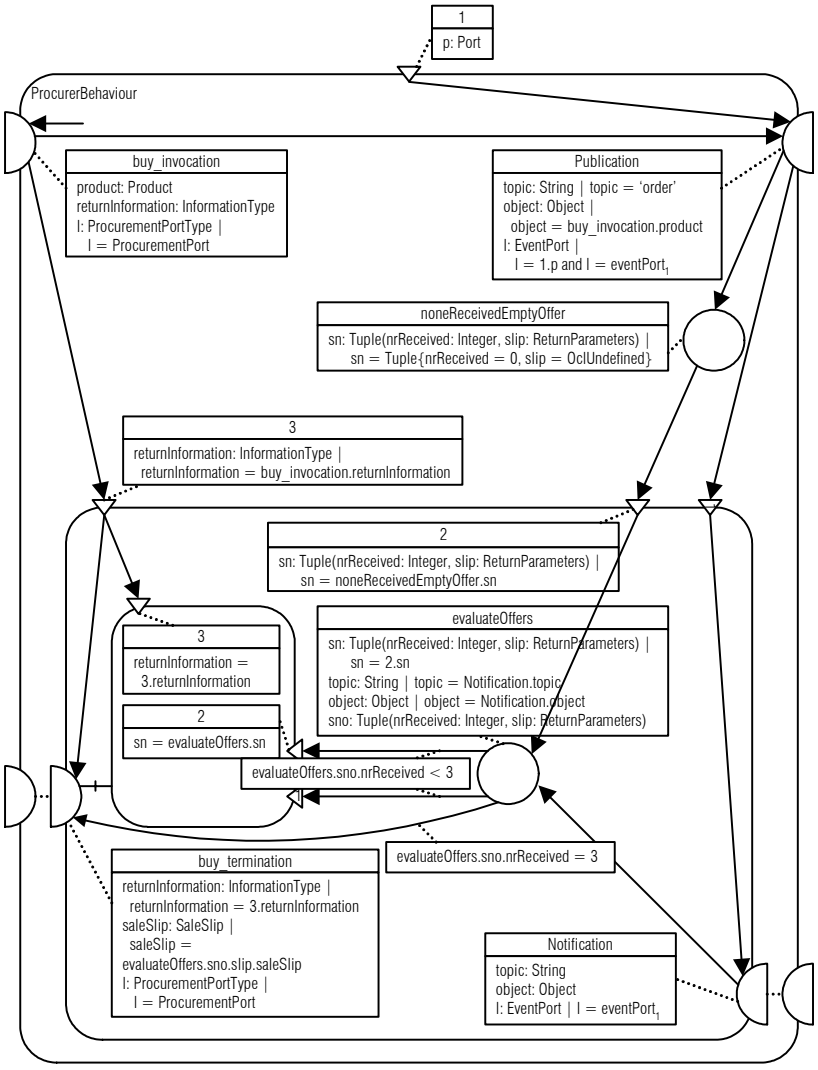


Figure 6-24 An Example of Template Transformation

Transformation of template concepts. We transform object templates into entity types and we transform interface templates into interaction

point part types. We transform the name of an interface template's signature into the location type of the corresponding interaction point part type. If the instances of one interface template are meant to be bound to instances of another interface template (which we can only derive by checking if there *are* instances for which this is true), the location types of the corresponding interaction point part types must be the same. Hence, we may change the names of the interface templates, similar to the way we change the names for interfaces.

We transform the behaviour specification of an object template into a basic behaviour type of the corresponding entity type. We transform the action templates that constitute those behaviours as follows:

1. If the action template represents an internal action (it is associated with an internal action signature), we transform it into a basic action instantiation. The parameters of the internal action are transformed into attributes of the basic action instantiation, such that the name of the parameter corresponds to the name of the attribute and the type of the parameter to the type of the attribute.
2. If the action template represents a signal (it is associated with two signal signatures), we transform it into a basic interaction contribution instantiation. The parameters of the signal are transformed into attributes of the basic interaction contribution instantiation. Optionally, the port on which the signal occurs is transformed into the value of the location attribute. The value of the location attribute must be equal to the location into which the port is transformed. The basic concepts have no way of representing whether the interaction contribution represents the initiating or responding part of the signal. Therefore, these design properties are lost during the transformation and consistency with respect to these properties cannot be verified using the basic concepts.
3. If the action template represents an operation (it is associated with two operation signatures), we transform the invocation part of the operation into an interaction contribution instantiation. If the operation has a termination, we transform the termination into an interaction contribution instantiation as well. The condition of that interaction contribution instantiation must include the condition that the 'interaction contribution instantiation that corresponds to the invocation must have occurred'. The parameters of the invocation and termination are transformed into the attributes of the corresponding interaction contribution instantiations. Optionally, the port on which the signal occurs is transformed into the value of the location attribute.

The constraints of behaviours can be transformed in the same way as we transform behaviour constraints in the enterprise viewpoint, because, as in the enterprise viewpoint, they are represented using UML activities.

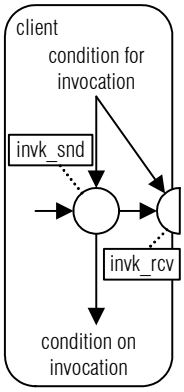


Figure 6-25 Condition on Invocation on the Client's Side

In a UML activity diagram, a flow from an operation invocation action to some other action represents the fact that the other action is enabled immediately after the operation has been sent. We do not wait for the invocation to be received on the server side. However, in ISDL, if we prescribe that an action is enabled by an interaction contribution that represents an operation invocation, then we represent that the action is only enabled after the invocation has been received on the server side. Therefore, we cannot transform a flow from an operation invocation action into an enabling condition on that action. To solve this problem, we insert an action that represents sending an invocation for each invocation action, as illustrated in Figure 6-25. We transform a flow from the invocation action into an enabling condition on that inserted action.

For the same reason, we insert a new action, which represents sending a termination, for each termination action on the server side. A flow leaving an operation termination action on the server side must be transformed into an enabling condition on the inserted action.

Since the transformation transforms bound interfaces into a single interaction point, the constraints that apply to the interfaces must be transformed into constraints on the interaction point.

Example 6-4 Transformation of Templates

Figure 6-24 shows an example in which the 'ProcurerBehavour' behaviour template from Figure 6-21 is transformed. The transformation follows the algorithm outlined in section 6.2.5. Note that the name of the location at which the procurer interacts with the event server is changed to '$eventPort_1$'. This is the result of the transformation of instance concepts, in which bound interfaces, which may have different names, are transformed into interaction points, which must have a single location (with a single name). The constraints that applied to the two interfaces are transformed into constraints on the interaction point. Also note that, while the UML actions that represent the 'buy invocation' and 'buy termination' interactions both have the name 'buy' in the UML activity, the basic interactions into which they are transformed have the names 'buy_invocation' and 'buy_termination', respectively. This is because UML represents invocations and terminations, using the name of the operation to which they belong. Instead, the basic concepts into which invocations and terminations are transformed use the names of these invocations and terminations themselves. Another difference that stems from the way in which UML represents computational concepts is the difference between the reference of parameters in UML and in the basic concepts. UML references the parameters of a received signal using the name of that signal, while the basic concepts directly reference these parameters. Similarly, UML references the parameters of a termination using a separate class, while the basic concepts directly reference these parameters.

Transformation into behaviour instantiations. As in the enterprise viewpoint, we must create instances of behaviour types, such that those behaviour types can be performed.

For each object, we create an instance of the behaviour type that represents the behaviour of the corresponding object template.

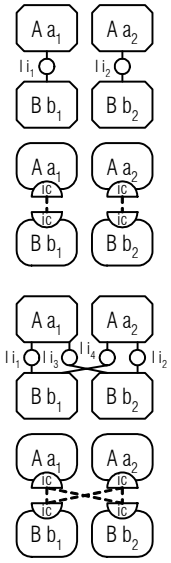
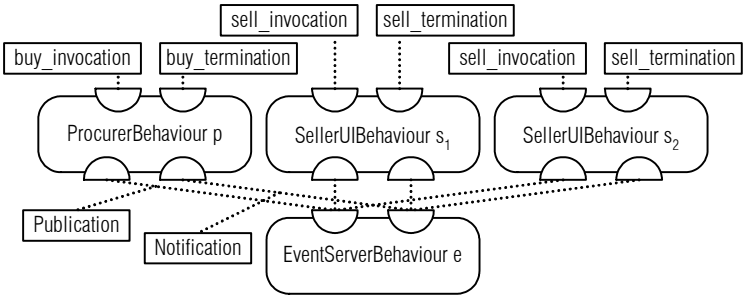


Figure 6-26 Instantiation of Interactions

For each action template that represents an interaction (i.e. it is associated with an interaction signature), we create a basic interaction, but only if there exist bound instances of the interfaces at which this interaction can occur. For each pair of bound interfaces at which the interaction can occur, we create an alternative of the interaction. Each alternative consists of a pair of interaction contributions that are contributed by objects with bound interfaces. Figure 6-26 illustrates this. It shows two examples in which objects of templates *A* and *B* (with behaviour templates that have the same names) can have an interaction of template *ic*. Objects of these templates can interact at interfaces that are transformed into interaction points of type *I*. In the first example, only a_1 and b_1, and a_2 and b_2 are bound through interfaces. Therefore, there are only alternative interactions between a_1 and b_1, and between a_2 and b_2. In the second example, a_1 and b_2, and a_2 and b_1 are bound as well. Therefore, there also exist alternative interactions between a_1 and b_2 and between a_2 and b_1.

Figure 6-27 shows the instantiations of behaviour types that correspond to the behaviours of the objects from Figure 6-22. The figure assumes that interaction templates 'Publication' and 'Notification' exist between 'ProcurerBehaviour' and 'EventServerBehaviour', and between 'SellerUIBehaviour' and 'EventServerBehaviour'. A 'Publication' or 'Notification' interaction can occur between the 'EventServerBehaviour' and either the 'ProcurerBehaviour' or one of the 'ServerUIBehaviours'. This is consistent with how the 'Procurers' and 'SellerUIs' are bound to the 'EventServer' in Figure 6-22.

Figure 6-27 An Example of Transformation into Instantiations



6.4 Relations between Enterprise and Computational Views

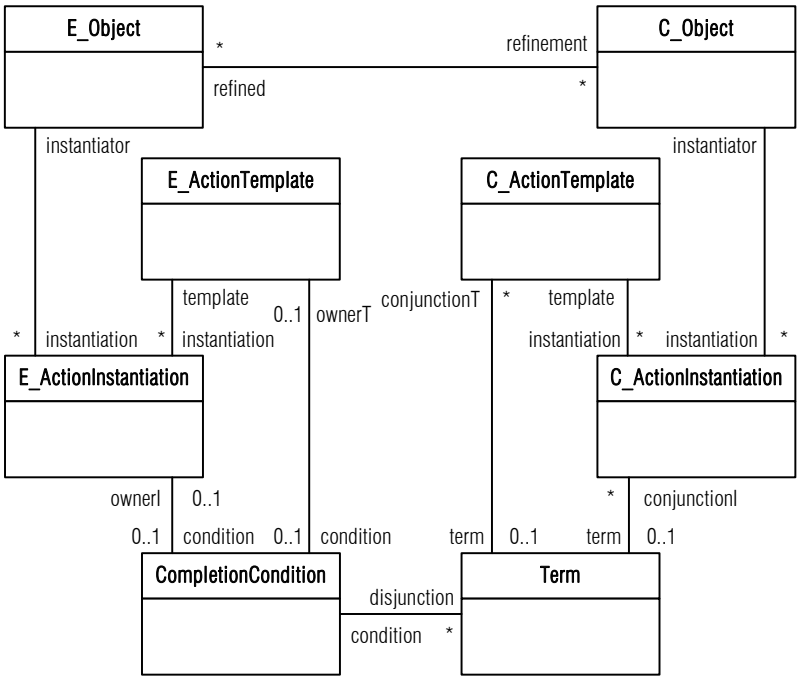
In this section we consider the refinement relation between enterprise and computational views in more detail. We precisely represent the relations and consistency rules between:

– enterprise and computational objects;
– enterprise role-based behaviour and computational behaviour; and

– enterprise process-based behaviour and computational behaviour.

We base these relations and consistency rules on the viewpoint correspondences that RM-ODP defines.

Figure 6-28 Relations between Enterprise and Computational Views



Enterprise and Computational Objects. In the relations between enterprise and computational views, computational objects refine enterprise objects that represent (parts of) the system under design. The computational objects describe the system in more detail, by decomposing it into functional parts and, optionally, describing the activities that the system can perform in more detail. We represent this relation by a MOF association between the enterprise object and computational object concepts. Figure 6-28 shows this association.

We do not associate consistency rules with this relation. The designer is free to relate enterprise objects to computational objects that refine them.

Enterprise Role-Based Behaviour and Computational Behaviour. If a computational object refines an enterprise object, its behaviour refines the behaviour of that enterprise object. Hence, actions of the computational object can be final actions for actions of the enterprise object, according to some completion condition.

We represent this relation and the associated consistency rules, using operators on the basic concepts. The operators on the basic concepts allow

us to represent that action *instantiations* are final actions for action *instantiations*. However, the enterprise and computational viewpoints only contain concepts that correspond to the basic action *template* concept (namely the enterprise action template and the computational action template concepts). Therefore, we should introduce enterprise and computational concepts that correspond to the basic action instantiation concept. Since each object that performs (the behaviour that contains) an action template has an *instantiation* of that template, we introduce action instantiation concepts, as shown in Figure 6-28. We relate those action instantiation concepts via a completion condition. The completion condition is a disjunction of a conjunction of final (computational) action instantiations.

Enterprise and computational action templates that represent a joint activity between two or more objects correspond to a basic interaction. Since we did not define refinement rules for interactions, we cannot relate action templates that correspond to basic interactions. Instead, we use the approach described in subsection 5.2.2 to represent and verify refinement relations between interactions. In this approach, we compose (the behaviours that contain contributions to) the interactions. Subsequently, we verify the refinement of the resulting action instantiations. Using this approach, each interaction is composed into one or more action instantiations. Therefore, we allow enterprise or computational action templates that correspond to basic interactions to participate in a completion condition, representing that the action instantiation that they correspond to (after composition) participate in the completion condition.

If a computational action template corresponds to more than one basic action instantiation after composition, we assume that *each* of these action instantiations is a final action for the related enterprise action. We do not cover the case in which an enterprise action template corresponds to more than one basic action instantiation after composition. This case is left for future work.

Example 6-5 Enterprise and Computational View Relations

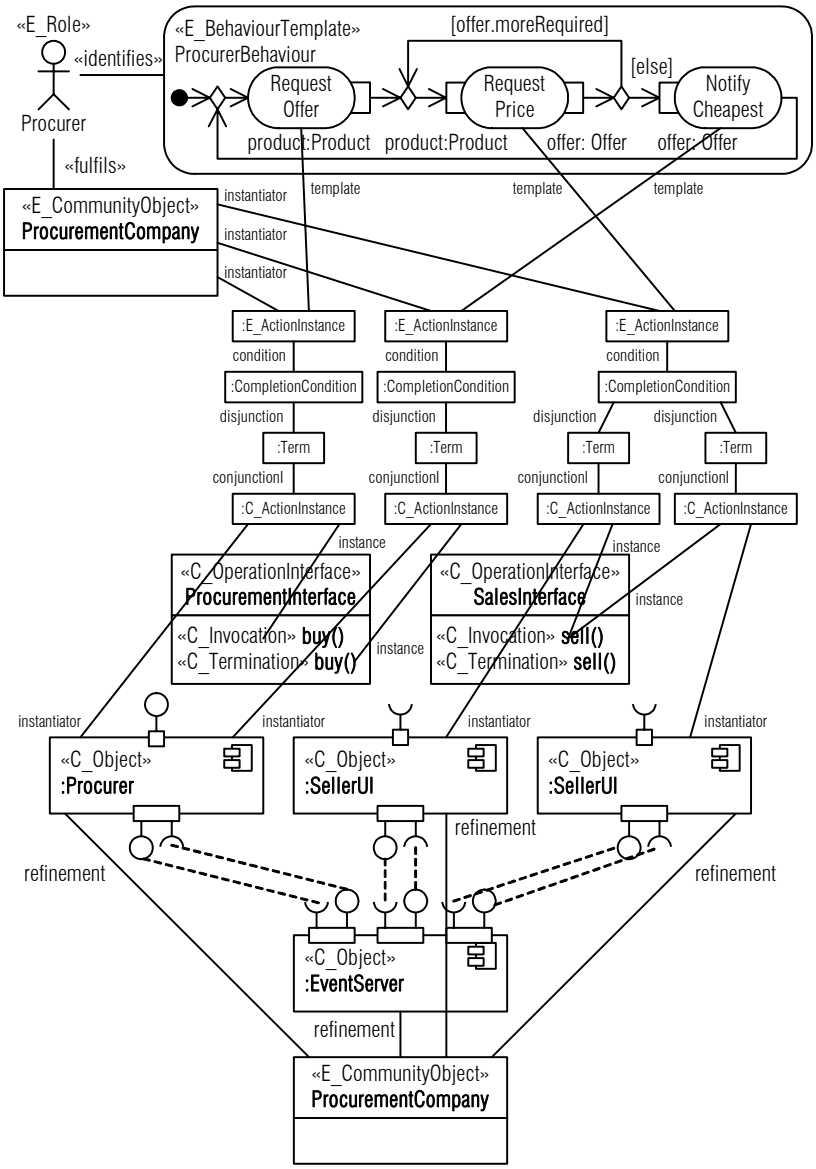
Figure 6-29 shows the relations between the enterprise view from Figure 6-8 and the computational view from Figure 6-22. We represented the 'ProcurementCompany' enterprise object twice to make the model more clear. Also, we split up each interrogation into an announcement and a termination, to be able to relate these two parts separately.

The figure shows a relation between the 'ProcurementCompany' enterprise object and the 'Procurer', 'EventServer' and both 'SellerUI' computational objects. This relation represents that the 'ProcurementCompany' enterprise object is refined by those computational objects.

The figure also shows a relation between the 'RequestOffer' enterprise action instantiation, as it is performed by the 'ProcurementCompany' enterprise object, and the 'buy_invocation' computational action instantiation as it is performed by the 'Procurer' computational object. These action instantiations are related via a completion condition that represents that the completion of a 'buy_invocation' action instantiation corresponds to the completion of a

'RequestOffer' action instantiation. Similarly, the 'buy_termination' computational action instantiation, as it is performed by the 'Procurer' computational object, is related to the 'NotifyCheapest' enterprise action instantiation. The 'RequestPrice' enterprise action instantiation, as it is performed by the 'ProcurementCompany' enterprise object, is related to the 'sell' action instantiations, as they are performed by the 'SellerUI' objects. The completion condition that relates them represents that the completion of either one of the 'sell' action instantiations corresponds to the completion of 'RequestPrice' enterprise action instantiation.

Figure 6-29 Example of Enterprise and Computational View Relations



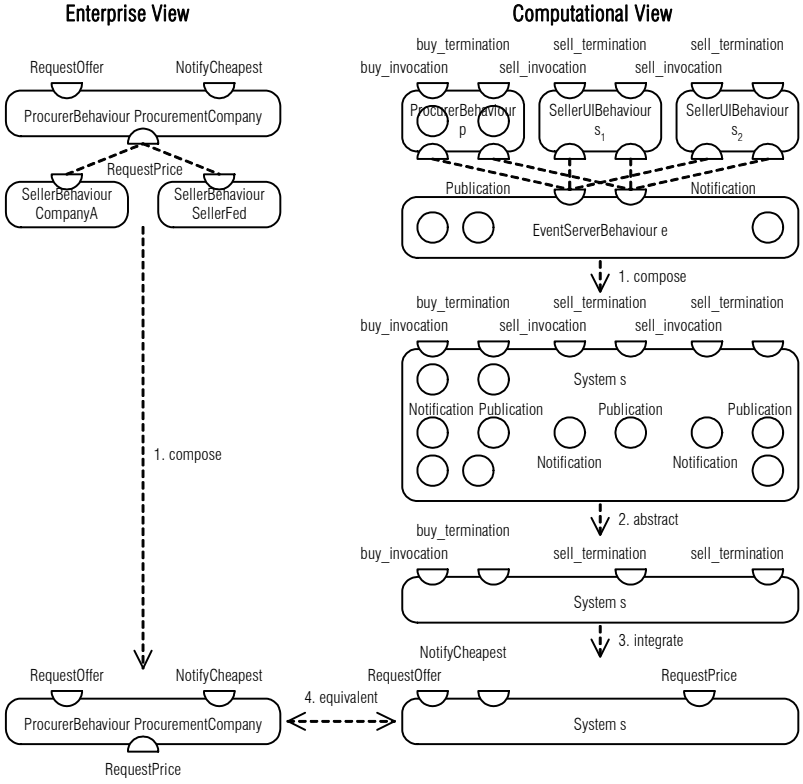
The following consistency rule applies to the relations between the enterprise and computational viewpoint. Computational objects refine the enterprise objects that represent the system. Therefore, in each community, the joint behaviour of the computational objects implementing that community must conform to the joint behaviour of the enterprise objects that they refine. We verify this consistency rule in four steps.

1. For each community, we compose all basic behaviour instantiations that represent behaviour performed by computational objects that implement that community. Also, we compose all basic behaviour instantiations that represent behaviour performed by enterprise objects in that community, but only those enterprise objects that represent parts of the system rather than its environment. These are enterprise objects that are related to computational objects via the association from Figure 6-28. We compose the behaviour instantiations, because the following steps (the abstract and integrate operators) can only be performed on a single behaviour instantiation, rather than on multiple interacting behaviour instantiations. We compose only enterprise behaviour instantiations that represent parts of the system, because the computational viewpoint only represents the behaviour of the system. Therefore, we can only verify consistency of behaviour instantiations that represent behaviour of the system.
2. In the resulting computational behaviour instantiation, we abstract from basic action and interaction contribution instantiations that *do not* correspond to the completion of an enterprise action instantiation. These are basic action and interaction contribution instantiations that do not track (via a tracking relation that is stored during the transformation into basic concepts) to a computational action instantiation or template that is related to an enterprise action instantiation or template. Such a relation is represented by the 'completion condition' and 'term' concepts from Figure 6-28 and their associations.
3. In the resulting computational behaviour instantiation, we integrate basic action and interaction contribution instantiations that *do* correspond to the completion of an enterprise action instantiation. The 'completion condition' from Figure 6-28 represents the precise relation that such basic action and interaction contributions have to their enterprise viewpoint counterparts.
4. We verify whether the resulting computational behaviour instantiation is (strongly) equivalent to the enterprise behaviour instantiation. If the behaviour instantiations are equivalent, the views are consistent with respect to this consistency rule.

Appendix A.1 defines an OCL constraint that represents this consistency rule. The OCL constraint follows the steps outlined above. It assumes that, when an enterprise or computational view is transformed into a basic de-

sign, tracking relations are maintained that can relate a basic behaviour instantiation to the viewpoint behaviour instantiation from which it was created. It also assumes that tracking relations are maintained that can relate a basic action instantiation, interaction contribution instantiation or interaction to the viewpoint action instantiation or template from which it was created.

Figure 6-30 Example of Checking Behaviour Consistency between Enterprise and Computational Views



Example 6-6 Checking Behaviour Consistency between Enterprise and Computational Views

Figure 6-30 illustrates how this consistency rule can be verified in four steps. The figure shows the basic behaviour instantiations that are the result of a transformation of the enterprise view from Figure 6-8 and the computational view from Figure 6-22 into basic concepts. We can verify the consistency between these basic designs as follows:

Step 1: compose the enterprise and computational behaviour instances that represent the behaviour of the system. The relations that are specified in Figure 6-29 imply that only the enterprise 'ProcurementBehaviour' represents behaviour of the system, because only the 'ProcurementCompany' is refined by computational objects. Therefore, composition is not needed in the enterprise view. In the computational view, composition results in composition of the 'Notification' and 'Publication' signal interactions.

Step 2: abstract from computational actions that do not correspond to the completion of an enterprise action. The relations that are specified in Figure 6-29 imply that only the

'buy_invocation', 'buy_termination' and 'sell_termination' interactions correspond to the completion of enterprise actions.
Step 3: integrate the computational actions that correspond to the completion of an enterprise action, as represented by completion conditions that are specified in Figure 6-29.
Step 4: verify that the resulting enterprise and computational behaviour instantiations are equivalent.

Enterprise Process-Based Behaviour and Computational Behaviour. Each process in an enterprise view is prescriptive for the computational view. This relation between enterprise processes and computational behaviours is implicitly represented by the relation between enterprise actions and computational actions from Figure 6-28, because enterprise actions correspond to steps in an enterprise process.

The consistency rule associated with this relation is that each computational behaviour must conform to the processes that it refines, but only to the part thereof that is performed by the system.

Two differences between the concerns that enterprise processes and computational behaviours address, complicate verifying this consistency rule. Firstly, an enterprise process design on the one hand typically covers only a single customer. It does not cover mechanisms that allow multiple customers to engage in the same business process. A computational behaviour design on the other hand also cover mechanisms that allow multiple customers to engage in (the computational realization of) the same business process. Therefore, a computational behaviour specification and an enterprise behaviour specification in terms of processes can only be compared after we remove such concurrency mechanisms from the computational behaviour specification. We can only remove these concurrency mechanisms, if they are clearly indicated in the computational view. Secondly, an enterprise process design does not consider relations between steps that belong to different processes, while a computational design *does* consider relations between actions of which abstractions belong to different processes. Therefore, we can only compare the behaviour of an enterprise process to a computational behaviour, after abstracting from such relations.

We verify the consistency rule, using the four steps outlined above. However, instead of verifying equivalence of the computational behaviour to the composed enterprise behaviour, we verify equivalence to a business process. Also, we only consider parts of the computational view that do not represent concurrency mechanisms. To this end we assume that computational action and constraint templates can be marked as representing a part of such mechanisms. To verify consistency with business processes, we apply a modified version of the transformation from the computational view to basic concepts. This transformation does not transform action and constraint templates that are marked. A better solution would be to mark the

basic concepts rather than not transform them. We leave this for future work. Appendix A.2 defines an OCL constraint that represents this consistency rule.

6.5 Information Viewpoint

The information viewpoint is used to design the structure of information in a system and its environment and basic operations that can be performed on this information. This is the concern of a stakeholder that focuses on the information in the system, such as a database analyst. Also, we rely on the information viewpoint to represent the structure of parameters of actions in the computational viewpoint. To represent the structure and processing rules of information, the information viewpoint addresses the following concerns:

- the structure of information objects;
- the invariants that must hold for information objects; and
- how actions use, change, create and remove information objects.

6.5.1 Information Viewpoint Concepts

The conceptual model from Figure 6-31 represents the information viewpoint concepts, using the UML profile for MOF. The conceptual model deviates from the one provided by ITU-T, & ISO/IEC (2005). As in the computational viewpoint, we make a more strict distinction between template and instance concepts. We consider the state of an information object to be implicitly determined by the information values of that object, because the information values of that object completely determine the actions that it can take part in next. Therefore, we do not explicitly consider the state concept in our conceptual model. Also, we leave the addition of the type concept for future work. In RM-ODP a type is a predicate. We consider the template of an object to be the type of that object.

Information objects represent information that is established in the system. An information object can be either *atomic* or *composite*. If it is atomic, it is not composed of other information objects. If it is composite, it is composed of component information objects, each of which has a name that identifies it within the composite information object. Unlike (ITU-T, & ISO/IEC, 2005) we explicitly consider atomic and composite information object concepts in our conceptual model, because the distinction between the two concepts is made explicitly in RM-ODP. The template according to which an information object can be created is composite or atomic, depending on the structure of the objects that it creates. We can associate information object templates with *invariants*. An invariant represents a con-

straint that must hold at all times for each object that is constructed according to that template.

Information actions represent activities that change the state of some collection of information objects. Information actions are constructed from information action templates. Such a template provides more information about how actions of that template can affect information. To this end it prescribes information that is input to such actions and information that is output from such actions. The information is identified by a name and references an information object template to represent the structure of the input or output information. An action template specifies a *precondition* that represents the state that the original information objects must be in, for actions of that template to be allowed to occur. Also, an action template specifies a *postcondition* that represents the state that resulting information objects must be in, after actions of that template have occurred.

6.5.2 Representation of Information Views

Like for enterprise and computational views, we use UML 2.0 to represent information views by means of models. We deviate from (ITU-T, &

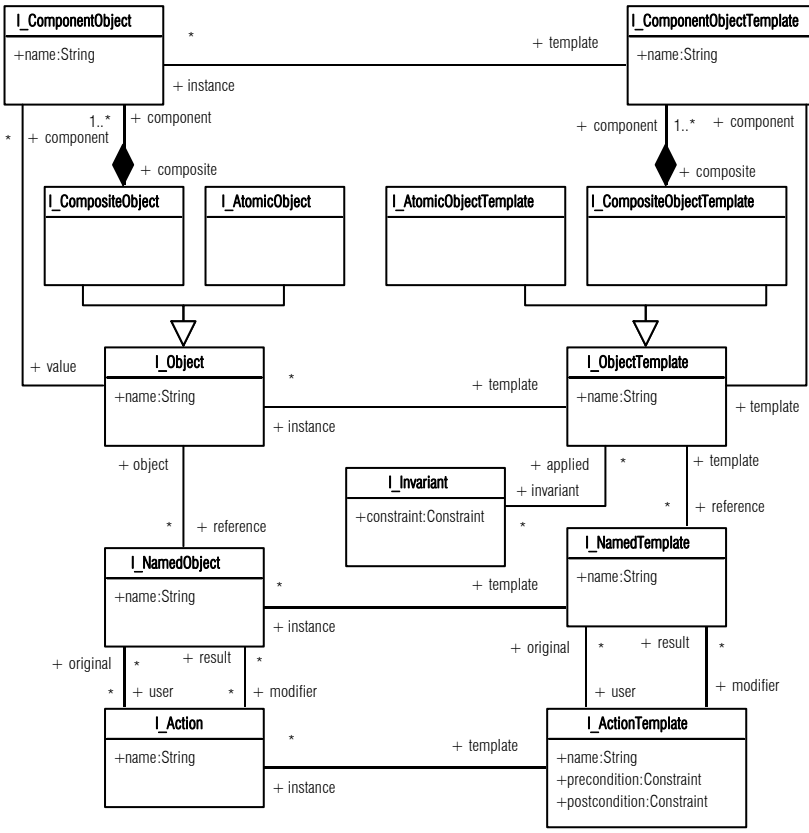


Figure 6-31 Conceptual Model for Information Concepts

ISO/IEC, 2005), because the representation proposed there is based on explicit representation of states.

Table 6-6 Representation of Information Viewpoint Concepts in UML 2.0

Information Viewpoint Concept	UML 2.0 Modelling Element
I_ObjectTemplate	Class stereotyped I_ObjectTemplate
I_ComponentObjectTemplate	Attribute of Class
I_AtomicObjectTemplate	Classes with names: String, Boolean, Integer or UnlimitedNatural
I_Object	Object stereotyped I_Object
I_ComponentObject	Attribute of Object
I_AtomicObject	PrimitiveType
I_Invariant	Constraint represented in OCL stereotyped I_Invariant
I_ActionTemplate	UML CallBehaviourAction
I_NamedTemplate	Pin of the CallBehaviourAction

Table 6-6 describes the representation relation between our information viewpoint concepts and UML 2.0 modelling elements. We stereotype the UML modelling elements, as indicated in Table 6-6, to make them identifiable as information concepts.

We represent object templates by UML classes. We represent the components of a composite object template by attributes of the UML class that represents that template. The name and type of such an attribute represent the name and type of the component, respectively. We represent invariants as OCL constraints. We consider the following atomic object templates (which we derived from the primitive types that UML considers):

- A template with the name 'String' that represents a sequence of characters.
- A template with the name 'Integer' that represents an integer number.
- A template with the name 'UnlimitedNatural' that represents a natural number and infinity. We graphically represent infinity by an asterisk (*).
- A template with the name 'Boolean' that represents either the value 'true' or the value 'false'.

We represent an object by a UML object and we represent the components of a composite object by the attributes of the representing UML object. We represent an atomic object by the value that it represents (e.g.: true, 128, *, …), not by its name (we consider its value to be its name).

We only represent information action templates. Information actions are implicitly defined as the actions that can be constructed from the templates. We represent an action template as a UML call behaviour action. This UML action must have input pins that correspond to the object templates referenced as original object templates. A UML action must have output pins that correspond to the object templates referenced as result

object templates. We represent pre- and postconditions as UML constraints. A constraint can use the name of a named template to represent an instance of the UML class that represents the corresponding object template.

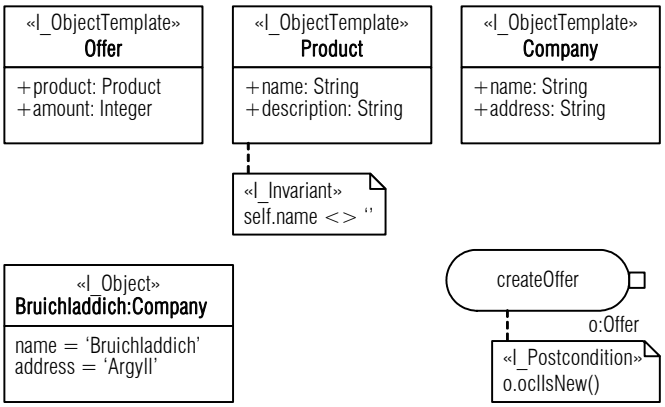


Figure 6-32 Information View Represented in UML

6.5.3 Example of an Information View

Figure 6-32 represents an information view in UML. The view shows three information object templates: 'Offer', 'Product' and 'Company'. The 'Offer' object template consists of a 'product' of template 'Product' and an amount of template 'Integer', representing that it describes a product and an amount of money that is to be paid for this product. The template references the 'Product' template (alternatively this could be represented as a UML association). The 'Product' template has an invariant that prescribes that it must have a name.

Figure 6-32 also represents an information object that is created according to the 'Company' template and an action template that represents the creation of an object from the template 'Offer'. As a postcondition, the instance of the template 'Offer' that it references, must be newly created according to the 'oclIsNew()' predicate.

6.5.4 Relation of Information Viewpoint Concepts to Basic Concepts

We define the relation between the information viewpoint concepts and the basic viewpoint concepts in terms of a transformation. The transformation defines how a view in terms of information viewpoint concepts can be transformed into a view in terms of basic concepts. For the transformation, we assume that the UML binding to the basic information concepts, as described in section 4.3, is used.

We transform an information object template into a basic information type with the same name, such that an atomic information object template is a basic primitive information type and a composite information object template is a basic composite information type. We transform the component of a composite information object template into a basic information block with the same name and a basic information type that corresponds to the object template of the component.

We transform an action template into a basic action type and we transform the named templates of an action template into basic attributes of the corresponding basic action type. We transform a pre- or postcondition into a basic attribute constraint on one of the basic attributes that it affects. We transform information objects into basic information values and information actions into basic actions.

Since the basic concepts focus on behaviour, different ways of transforming invariants exist, depending on how we want the system under design to behave with respect to information invariants.

1. We can transform an information object template into a basic information type that always meets the invariants, because an information value that does not satisfy the constraint is not part of the values identified by the type.
2. We can transform an invariant by adding it as an attribute constraint to all basic actions that affect the corresponding basic information type.
3. We can transform an invariant by adding additional basic behaviour that specifies what must be done if some invariant is not met.

We leave the inclusion of invariants for future work.

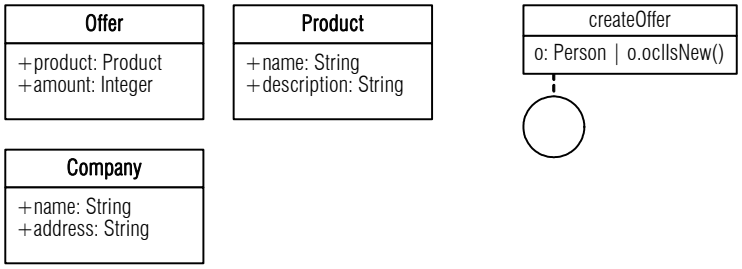
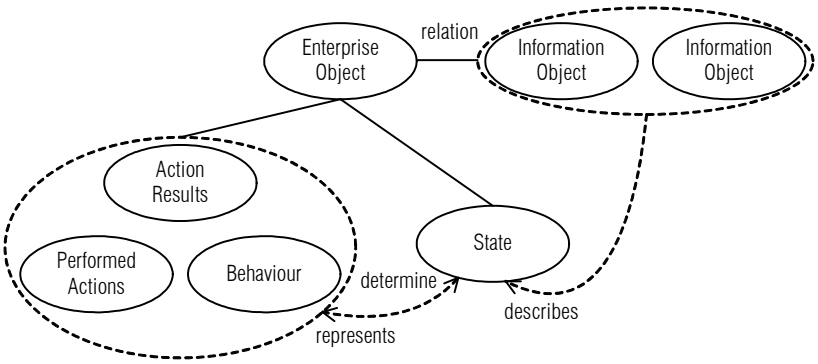


Figure 6-33 Transformation of Information Concepts

Figure 6-33 represents the transformed information view from Figure 6-32, using the UML binding to the basic information concepts. It illustrates that a straightforward relation exists between the information viewpoint concepts and the basic information concepts.

Figure 6-34 Relation between Enterprise and Information Objects Illustrated



6.5.5 Relations between Enterprise and Information Views

In the relations between an enterprise and an information view, an enterprise object can be related to information objects. Such a relation represents that the information objects (partly) describe the state of the related enterprise object. Figure 6-34 illustrates this relation. Since the state of an enterprise object represents the factors that determine which actions the enterprise object can perform next, the information objects also (partly) describe those factors. Since the behaviour of an object is fixed, information objects can represent which actions an enterprise object has performed and the results that were established in those actions. For that reason, an information view refines an enterprise view with respect to the information concern, by representing the results of enterprise actions (represented by artefacts) in more detail.

Example 6-7 Relation between Enterprise and Information Objects

For example, the information object 'Application Form' can describe the state of the enterprise object 'Application'. The information values of 'Application Form' can represent the result of the 'Fill Out Application' enterprise action and an information value 'accepted' can represent that the enterprise action 'Accept Application' was performed on the 'Application' object.

An enterprise object template can be related to information object templates, representing that the state of an instance of that the enterprise object template is described by instances of the related information object templates. Similarly, an enterprise role can be related to information object templates, representing that details about an enterprise object that fulfils the role are described by instances of the related information object templates.

In the relations between an enterprise and an information view, an enterprise action template can be related to information action templates. Such a relation represents that the information action templates are used in the realization of the related enterprise action template (RM-ODP is un-

clear about this relation, claiming that enterprise actions 'correspond to' information actions, but not explaining what that means). In this way, information action templates describe a part of the refinement of an enterprise action. For example, the 'start database transaction', 'add database entry' and 'commit database transaction' information action templates correspond to the 'enter client information' enterprise action template.

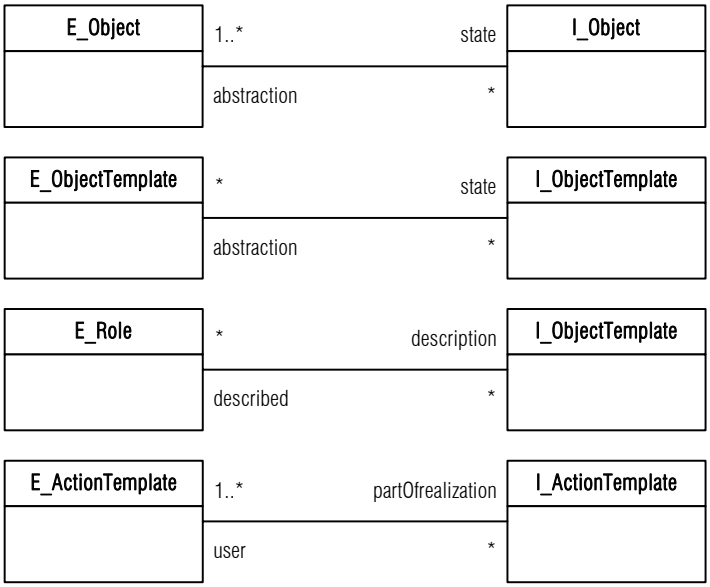


Figure 6-35 Relations between Enterprise and Information Views

Figure 6-35 represents the potential relations between enterprise and information views at a viewpoint level. View relations are instances of the relations represented in this figure.

The viewpoint relations imply consistency rules. One consistency rule is that information objects are related to an enterprise object, they must refine the results of the enterprise actions of that enterprise object. These results are represented as artefacts. This refinement relation is a refinement relation on information. However, since we have not defined what we mean by refinement of information, further research is necessary before we can specify this consistency rule. Other consistency rules are the following.

If an information object template describes the state of an enterprise object template, then the state of each enterprise object of that enterprise object template must be described by an information object of the information object template. In OCL, the consistency rule is:

```
context E_Object inv:
    self.template.state->forAll(iot: I_ObjectTemplate|
        I_Object.allInstances()->exists(io: I_Object|
```

```
            io.template = iot and self.state = io
        )
    )
```

If an information object template describes an enterprise role, then the state of each enterprise object that fulfils the role must be described by an information object of the information object template. In OCL, the consistency rule is:

```
context E_Object inv:
    self.fulfilled.state->forAll(iot: I_ObjectTemplate|
        I_Object.allInstances()->exists(io: I_Object|
            io.template = iot and self.state = io
        )
    )
```

If an enterprise object template is referenced as an artefact in an enterprise action template, then the information object templates that describe the state of the enterprise object template must be referenced by the information action templates that realize the enterprise action template. However, this rule only applies if such information object templates and information action templates exist. In OCL, the consistency rule is:

```
context I_ObjectTemplate inv:
    self.abstraction.mentioner.partOfRealization->
        forAll(iat: I_ActionTemplate|
            iat.result.template->includes(self)
            or
            iat.original.template->includes(self)
        )
```

Example 6-8 Enterprise and Information View Relations

Figure 6-36 illustrates related enterprise and information views. The figure represents that the state of the 'CompanyA' enterprise object is described by an information object that represents details about a company. The states of objects that fulfil the 'Offer' and 'Product' enterprise roles, must be described by information objects of the 'Offer' and 'Product' templates, respectively. The 'RequestPrice' enterprise action template is partly realized by an information action template that creates a new offer.

The 'RequestPrice' enterprise action template references the 'Offer' enterprise object template, of which the state is described by the 'Offer' information object template. Therefore, the third consistency rule requires that the 'createOffer' information action template references the 'Offer' information object template. Since this is the case, the third consistency rule is satisfied.

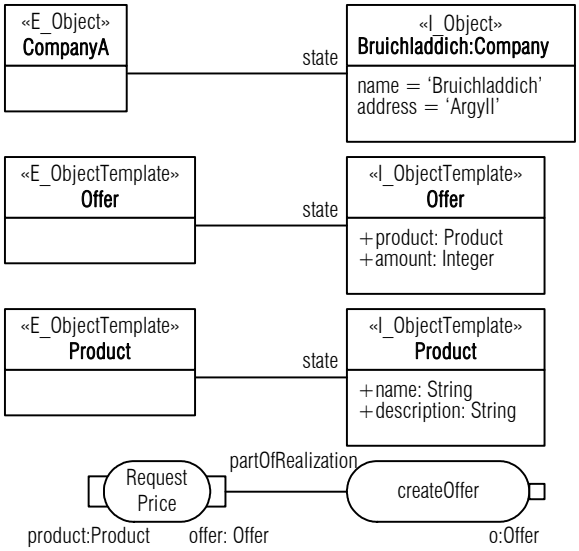


Figure 6-36 Example of Enterprise and Information View Relations

6.5.6 Relations between Computational and Information Views

Similar to the relation between information and enterprise objects, information objects (partly) represent the state of a computational object. However, where the enterprise viewpoint has its own concept (artefact) for representing the information concern, the computational viewpoint relies on the information viewpoint for that. Therefore, an information view complements, rather than refines, a computational view with respect to the information concern.

Like information objects are related to computational objects, information object templates are related to computational object templates, representing that their instances partly describe the state of computational objects.

Example 6-9 Relation between Computational and Information Objects

An example of a relation between a computational object and information objects is a 'Database' computational object that stores information about client accounts. In the information view these accounts can be represented as 'Account' objects. The 'Account' objects are the result of computational actions that store information about client accounts in the 'Database'. As another example, the value of the 'connected' component object of the 'Connection' information object can correspond to possible states of the 'Switchboard' computational object. It represents whether a connection has been established by the 'Switchboard' computational object.

In RM-ODP a change in the state of an information object, represented by an information action being performed on that object, corresponds to some actions being performed in the computational view. Therefore, we relate the information action template that causes a state change to the computational action templates that cause the same state change in the computa-

tional view. For that reason, the computational view refines the information view with respect to the behaviour concern. An example of such a relation is that a 'createAccount' information action template is implemented by some computational interaction templates with the 'Database' computational object template.

We relate each parameter of a computational action to an information object template. This relation represents that the type of the parameter is described by the information object template, such that the information object template determines the possible values and structure of the parameter.

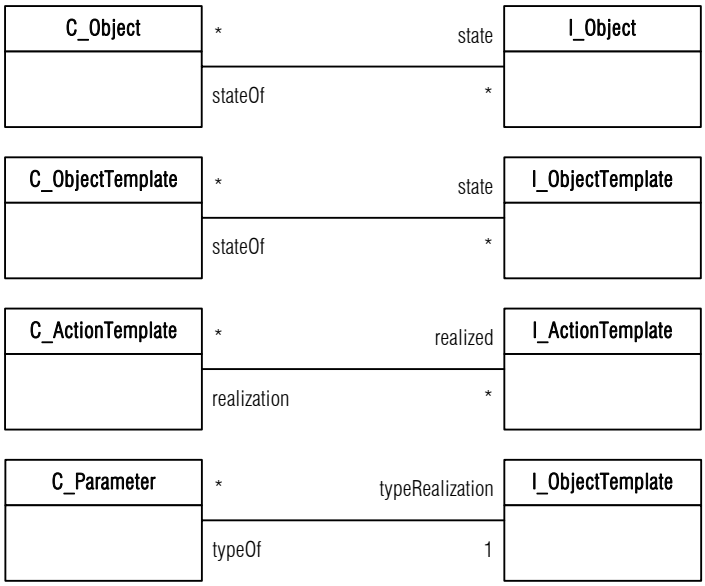


Figure 6-37 Relations between Computational and Information Views

Figure 6-37 represents the relations between the computational viewpoint concepts and information viewpoint concepts described above. These relations imply the following consistency rules.

The relations between computational and information objects (and their templates) imply that each value of an information object corresponds to particular states in the behaviour of a related computational object. Therefore, impossible values of information objects, as defined by invariants, must correspond to impossible states in the related computational objects. Similarly, information actions that cause a change in the values of some information objects, as defined by pre- and postconditions, must be related to computational actions that cause corresponding changes in the states of computational objects. However, since we neither consider state nor are able to compare information values to execution states, we cannot verify consistency with respect to these properties.

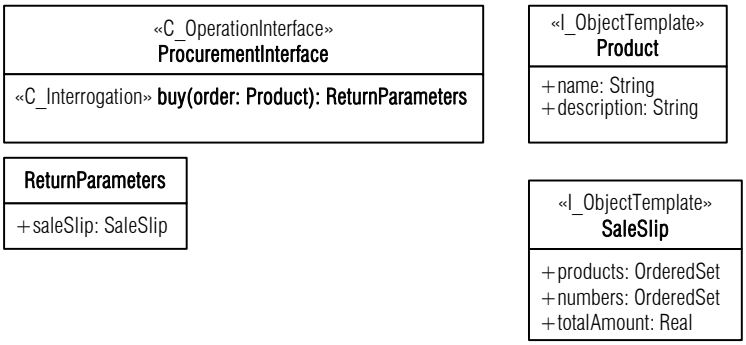
A more trivial consistency rule is that, if a computational object template is related to an information object template, then instances of the computational object template must be related to instances of the information object template. In OCL, the consistency rule is:

```
context C_Object inv:
    self.template.state->forAll(iot: I_ObjectTemplate|
        I_Object.allInstances()->exists(io: I_Object|
            io.template = iot and self.state = io
        )
    )
```

A consistency rule, stating that each parameter must be related to an object template, is specified as a cardinality constraint in Figure 6-37 and by the following OCL constraint. This constraint states that the type of a computational parameter is realized by an information object template with the same name.

```
context C_Parameter inv: self.type = self.typeRealization.name
```

Figure 6-38 Example of Computational and Information View Relations



Example 6-10 Computational and Information View Relations.

Figure 6-38 illustrates a relation between the computational and information viewpoint. The views show how the parameters of the 'buy' computational action template are realized by information object templates 'Product' and 'SaleSlip'. The relations between the parameters and information object templates is represented by the parameter type and the information object template having the same name.

Chapter 7

Conclusions and Future Work

This chapter presents our conclusions and considerations for applying the work described in this thesis. Also, it lists the contributions of this work and it provides suggestions for future work.

7.1 Main Conclusions

This thesis proposes a framework to help maintain consistency in designs that incorporate viewpoints from different stakeholders, focusing on viewpoints that address behavioural, structural and information concerns. Our framework is based on the hypothesis that the use of a common set of basic design concepts aids in defining rules to check the consistency between views. We claim in particular that the set of design concepts that we defined earlier in (Quartel, 1998; Quartel et al., 1997; van Sinderen, 1995; Ferreira Pires, 1994), aids in defining such consistency rules. A common and basic set of concepts represents properties that all stakeholders consider relevant (common) and that are elementary (basic), as opposed to composite properties that can be represented by a composition of elementary properties. We present our framework in chapter 3 and our common and basic concepts in chapter 4.

In a case study, presented in chapter 6, we show that our framework and our set of basic concepts can be applied to check consistency between views. We show this by applying the framework to define consistency rules between designs from RM-ODP based enterprise, computational and information viewpoints.

We show that our framework and our set of basic concepts also aid in managing the consistency between views in the following respects:

– A common set of design concepts provides a common vocabulary that stakeholders can use to understand each other's design concepts.

– Frequently occurring consistency rules can be defined on a common set of design concepts. Such consistency rules can be re-used to define consistency rules between views. In chapter 5 we define some re-usable consistency rules, motivated by the fact that they are commonly used between views. The rules that we define can be used to check the consistency between views that have some form of refinement or overlap relation.

7.2 Considerations for Applying the Framework

Some considerations must be taken into account when applying the framework.

Limitations of the framework. Since the use of the basic design concepts is essential in the framework, the framework only aids in managing the consistency of design properties that can be represented by the basic design concepts. However, in case the basic concepts cannot be exploited, the framework still allows a designer to define consistency rules directly on the viewpoint concepts.

The designer must be careful not to violate the semantics of viewpoint concepts when defining a relation between viewpoint concepts and basic concepts. A viewpoint concept must have exactly the same semantics as the basic concepts to which it is related. Otherwise, the result of a consistency check via the basic concepts is unreliable.

Consistency rules and consistency. That a consistency rule is satisfied is a necessary, but not sufficient conditions for the consistency of views. If a consistency rule is not satisfied we know that the views to which it applies *are not* consistent. However, if the consistency rule is satisfied, we cannot claim that the views *are* consistent. We can only claim that the views are consistent with respect to that particular rule.

For this reason and because chapter 6 shows that specifying a consistency rule can cost significant effort, a designer should not try to be exhaustive, when specifying consistency rules using our framework. The designer can use alternative means to check consistency. For example, the designer can manually compare two views. It is up to the designer to find a suitable means for checking consistency.

Using basic concepts and using formal semantics. We can check consistency between views using basic concepts, using formal semantics or using a combination of both. Each of these options has benefits and drawbacks.

A formalism is developed with the aim of reducing ambiguity, increasing conciseness and aiding in analysis and verification. Our basic concepts on the other hand are developed with the aim of representing the application domain in such a way that designs are easy to understand and develop (more detailed criteria are defined by (Vissers, van Sinderen, & Ferreira Pires, 1993)). Observing these differences with respect to aim, we can conclude that formalisms on the one hand are better suited for verifying and analyzing models. Our basic concepts on the other hand are easier to use by people that are less mathematically skilled.

We can exploit the benefits of both the basic concepts and formal semantics if we provide the basic concepts with a formal semantics. In that way, on the one hand, the designer can focus on the relation between viewpoint concepts and basic concepts, which are relatively easy to understand. On the other hand, the basic concepts inherit means for analysis and verification from the formalisms.

Using basic concepts and using direct view relations. We can check consistency between views using basic concepts or using relations and consistency rules that are defined directly between views. Using basic concepts has both benefits and drawbacks with respect to using direct relations. The drawbacks of using basic concepts are that:

- it requires designers to familiarize themselves with another set of concepts;
- it requires designers to specify the relation between each set of viewpoint concepts and the basic concepts.

In addition to the benefits that basic concepts aid in the definition of reusable view relations and provide a common basis that helps stakeholders to understand each others concepts, benefits of using basic concepts are that:

- basic concepts help to understand more complex (viewpoint) concepts;
- basic concepts help to add precision to more complex (viewpoint) concepts.

For example, the remote procedure call concept can have many forms; to understand which form of remote procedure call we mean and to define it more precisely, we can define it in terms of a composition of basic interactions.

To show that using basic concepts aids in verifying consistency, we have to show that the benefits of using basic concepts outweigh the drawbacks. We have not shown this conclusively in this thesis. However, the observations above do provide a motivation.

Moreover, designers use a 'basic' set of concepts anyway to define and check consistency rules. For example, to define and check consistency rules between views that represent behaviour using different notations, designers typically use concepts from some behaviour formalism (e.g. Petri nets). In

these cases the designers already have the overhead of specifying a relation between the viewpoint concepts and the 'basic' concepts. Hence, there is no additional overhead compared to using the basic concepts that we propose.

7.3 Contributions

The research described in this thesis contributes to research in the area of the design of distributed systems that support administrative business processes. Specifically, it contributes to:
- multi-viewpoint design;
- basic concepts;
- basic view relations;
- concepts for enterprise, information and computational viewpoint design.

Contributions to multi-viewpoint design. Our research contributes to the research on multi-viewpoint design, because it uses basic concepts to define view relations. Although frameworks exist that define viewpoint specific concepts as compositions of more basic concepts (Object Management Group, 2003b; Naumenko, 2002; ITU-T, & ISO/IEC, 1995), the use of these basic concepts to define view relations is new. We also contributed to the techniques to define viewpoint specific concepts (Quartel, Dijkman, & van Sinderen, 2005).

Contributions to basic concepts. This thesis evaluates the expressiveness of the basic concepts that were defined earlier in (Quartel, 1998; Quartel et al., 1997; van Sinderen, 1995; Ferreira Pires, 1994). It evaluates these concepts in their ability to represent the design properties considered by three representative viewpoints: the enterprise, information and computational viewpoints. Based on this evaluation, it proposes improvements to the basic concepts. Specifically, we propose:
- concepts that address the information concern of distributed systems design;
- a more strict definition of concepts that address the structural concern of distributed systems, including some new concepts;
- a more strict separation between concepts that represent types and concepts that represent instances that conform to those types. For example, we defined a strict separation between the 'behaviour type', the 'behaviour instance' and the 'behaviour instantiation' concept, helping to define multiple instances that represent similar behaviour; and

– an abstract syntax for our basic concepts, using MOF meta-modelling techniques. This improves the precision of (the syntax of) our language and helps the re-use of MOF-compliant tools and techniques.

Contributions to basic view relations. Our research contributes to the basic view relations that we defined earlier in (Quartel, 1998; Quartel et al., 1997; van Sinderen, 1995; Ferreira Pires, 1994), in that we add:
– a behaviour decomposition relation, which is a specific form of refinement relation that represents that some interacting behaviours are a decomposition of another behaviour;
– an overlap relation, which represents that some concept instances in one view and some concept instances in another view represent overlapping properties;
– consistency rules that help us to verify the consistency between views that have a decomposition or overlap relation;
– an algorithm to verify causality preserving equivalence, also called strong equivalence, between behaviours.

Contributions to enterprise, information and computational concepts. The definition of the RM-ODP enterprise, information and computational viewpoint concepts as compositions of basic concepts leads to:
– a more precise definition of the relations between views from those viewpoints;
– a more precise definition of the relation between the viewpoint concepts and the basic (RM-ODP) concepts;
– a more general RM-ODP interaction concept, which is being considered in the revision of RM-ODP (Linington, Vallecillo, & Wood, 2004).

7.4 Future Work

We suggest the following topics for future work.

Exploit a formal semantics to compute consistency. We explained above how the strengths of basic concepts and formalisms can be combined. Therefore, we propose that, in addition to the formal semantics that are already defined on the basic concepts by Quartel (1998) to improve precision and perform behaviour simulation, formal semantics are added that help to compute consistency. For example, we can use Z (ISO/IEC, 2002) to compute consistency with respect to the information concern and Petri nets to compute equivalence of one behaviour to another. The basic concepts then serve to relate the different formalisms and use them in a

unified way. They also serve as an intermediate level to define concepts and consistency rules that are more easy to understand for a designer.

Create more informative consistency statements. Currently, the framework is based on specifying OCL constraints to represent consistency rules. These OCL constraints notify a designer *when* a consistency rule is violated. However, they do not provide information about *why* a consistency rule is violated. Such information is useful to solve inconsistencies. We propose that consistency rules that provide such information are developed. The problem of providing such information is named the traceability problem by Boiten, Derrick, Bowman, and Steen (1999; 1997).

Define criteria to distinguish design concepts and language concepts. We explained the distinction between design concepts and language concepts in chapter 3. However, while developing the conceptual model and modelling language for the basic concepts and the enterprise, information and computational viewpoint, we discovered that the distinction still leaves room for interpretation. Therefore, we claim that criteria should be defined to clearly separate design concepts from abstract syntax. To define these criteria, we can use the work of others (e.g. (Guizzardi, 2005; Guizzardi, Ferreira Pires, & van Sinderen, 2005)).

Define concepts to represent structural dynamics. In chapter 4 we defined structural dynamics as the design concern that represents when the system structure changes and how. However, we did not define concepts to represent structural dynamics. The definition of such concepts is left for future work.

Elaborate on basic consistency rules. For the class of monolithic behaviours, the basic consistency rules are defined, up to a level of detail at which they can be implemented. However, for the class of structured behaviours, consistency rules are only defined informally. The definition and implementation of these rules is left for future work. This work could be combined with the definition of a formal semantics to aid in computing consistency. Such a semantics can also be used to formally prove the correctness of the consistency rules.

Appendix A

Consistency Rules in OCL

A.1 Enterprise Roles and Computational Behaviour

```
-- The constraint assumes that the following operations are defined.

-- Operation of completion condition that yields the causality target
-- instantiations that track to the enterprise action instantiation or
-- template for which the completion condition is a completion condition.
-- e2b_cti2eat tracks a basic causality target instantiation to the
-- enterprise action template from which it was transformed.
-- e2b_i2eat tracks a basic interaction to the enterprise action template
-- from which it was transformed.

context CompletionCondition
   def: ctisForAbstractAction(): Set =
      allCTI->select(cti: CausalityTargetInstantiation |
         E_ActionInstantiation.allInstances()->
         exists(eai: E_ActionInstantiation |
            eai.instantiator=
            cti.basicrules_compose_target2target.bi.e2b_bi2eo
            and
            eai.template=
            cti.basicrules_compose_target2target.cti.e2b_cti2eat
            and
            eai.condition=self
         )
      ).union(
         allCTI->select(cti: CausalityTargetInstantiation |
            cti.basicrules_compose_action2interaction.
            e2b_i2eat.condition=self
         )
      )

-- Operation of completion condition that yields a textual version of the
-- completion condition.
-- The elements of the completion condition are (the names of) basic
-- action or interaction contribution instantiations that correspond to
-- computational actions in the completion condition.

   def: leftPartCompletionCondition():String =
      let terms = self.disjunction->asSequence() in
         terms->subSequence(2, terms->size())->
         iterate(t: Term; result: String = terms->first().conditionFor()|
            result.concat(' OR '.concat(term.conditionFor()))
         )

-- Operation that yields a textual version of this term in the completion
-- condition.
```

```
context Term
   def: conditionForTerm(): String =
      let cais = self.conjunctionI->asSequence() in
      let cait = self.conjunctoinT->asSequence() in
      '('.concat(
         if cais->isEmpty() then '' else
            cais->subSequence(2,cais->size())->
            iterate(cai: C_ActionInstantiation;
               result: String =
               Term.conditionForElements(
                  Term.ctisForInstantiation(cais->first())) |
                     result.concat(' AND '.concat(
                        Term.conditionForElements(
                           Term.ctisForInstantiation(cai))))
            )
         endif
      ).concat(
         if cais->isEmpty() or cats->isEmpty() then '' else ' AND ' endif
      ).concat(
         if cats->isEmpty() then '' else
            cats->subSequence(2,cats->size())->
               iterate(cat: C_ActionTemplate; result: String =
                  Term.conditionForElements(
                     Term.ctisForTemplate(cats->first())) |
                        result.concat(' AND '.concat(
                           Term.conditionForElements(
                              Term.ctisForTemplate(cat))))
               )
         endif
      ).concat(')')

   def: conditionForElements(ctis: Sequence): String =
      '('.concat(
         ctis->subSequence(2,ctis->size())->
            iterate(cti: CausalityTargetInstantiation;
               result: String = ctis->first().name |
               result.concat(' OR '.concat(cti.name))
            )
      ).concat(')')

   def: ctisForInstantiation(cai: C_ActionInstantiation): Sequence =
      allCTI->select(cti: CausalityTargetInstantiation |
         cai.instantiator.c2b_co2bi=
         cti.basicrules_abstract_target2target.
            basicrules_compose_target2target.bi
         and
         cai.template=
         cti.basicrules_abstract_target2target.
            basicrules_compose_target2target.cti.c2b_cti2cat
      )

   def: ctisForTemplate(cat: C_ActionTemplate): Sequence =
      allCTI->select(cti: CausalityTargetInstantiation |
         cti.basicrules_abstract_target2target.
            basicrules_compose_action2interaction.c2b_i2cat=cat
      )

-- The actual constraint is the following.
context E_Community inv:

-- Basic behaviour instantiations that correspond to enterprise behaviour
-- templates performed by enterprise objects in the community.
-- Only behaviours of objects that are part of the system are selected.
-- These are objects that are refined by computational objects
-- (according to the 'refinement' relations between the views).
-- track e2b:bi2eo tracks basic behaviour instantiations to the enterprise
-- objects that perform them.
let e_behinst =
   BehaviourInstantiation.allInstances()->select(
      bi: BehaviourInstantiation |
         self.member->includes((bi.e2b_bi2eo))
         and
```

```
          not (bi.e2b_bi2eo).refinement.isEmpty()
    )->asSequence()
in

-- Basic behaviour instantiations that correspond to computational
-- behaviours performed by computational objects.
-- Computational objects must implement an enterprise object in the
-- community.
-- track c2b:co2bi tracks computational objects to the basic behaviour
-- instantiations that they perform
let c_behinst =
    C_Object.allInstances()->select(co: Object |
        co.refined.configuration->includes(self)
    )->collect(co: Object |
        co.c2b_co2bi
    )->asSequence()
in

-- The behaviour instantiation that is the composition of all
-- instantiations in e_behinst.
let e_composed =
    e_behinst->subSequence(2,e_behinst->size())->
    iterate(bi: BehaviourInstantiation;
        result: BehaviourInstantiation = e_behinst->first() |
            bi.compose(result)
    )
in

-- The behaviour instantiation that is the composition of all
-- instantiations in c_behinst.
let c_composed =
    c_behinst->subSequence(2,c_behinst->size())->
    iterate(bi: BehaviourInstantiation;
        result: BehaviourInstantiation = c_behinst->first() |
            bi.compose(result)
    )
in

-- All causality target instantiations
let allCTI = CausalityTargetInstantiation->allInstances()
in

-- All causality target instantiations that can not be abstracted, because
-- they correspond to final actions or interactions.
-- These are:
-- 1. All causality target instantiations in the composed behaviour that
--    track to an action or interaction contribution in the non-composed
--    behaviour. These non-composed actions and interaction contributions
--    must track to a computational action instantiation that is a final
--    action for an enterprise action (template or instantiation).
--    c2b_cti2cat tracks a basic causality target instantiation to the
--    computational action template from which it was transformed.
let finalActions =
    allCTI->select(cti : CausalityTargetInstantiation |
        C_ActionInstantiation.allInstances()->
              exists(cai: C_ActionInstantiation |
                  cai.instantiator.c2b_co2bi=
                  cti.basicrules_compose_target2target.bi
                  and
                  cai.template=
                  cti.basicrules_compose_target2target.cti.
                      c2b_cti2cat
              )
    )
in

-- 2. All causality target instantiations in the composed behaviour that
--    track to an interaction in the non-comopsed behaviour. These
--    interactions must track to a computational action template that is a
--    final action for an enterprise action (template or instantiation).
--    c2b_i2cat tracks a basic interaction to the computational action
--    template from which it was transformed.
let finalInteractions =
```

```
    allCTI->select(cti : CausalityTargetInstantiation |
           not cti.basicrules_compose_action2interaction.
               c2b_i2cat.term.oclIsUndefined()
       )
    )
in

-- The causality targets, in the transformed and composed computational
-- behaviour, from which we can abstract.
-- These are causality targets that do not represent final actions,
-- interaction contributions or interactions.
let ctiToAbstract =
    (allCTI - finalActions) - finalInteractions
in

-- The composed computational behaviour in which we abstracted from all
-- inserted actions
let c_composed_abstracted =
    c_composed.abstract(ctiToAbstract)
in

-- Textual versions of all completion conditions
let ccs =
    CompletionCondition.allInstances()->collect(cc: CompletionCondition |
        cc.ctisForAbstractAction().name.collect(name: String |
            ' TO '.concat(name))->
            product(cc.leftPartCompletionCondition())->
                collect(tuple : Tuple(first: String, second: String) |
                    tuple.second.concat(tuple.first)
                )
            )
        )
    ).flatten()
in

-- The integrated, abstracted and composed behaviour that corresponds to
-- the behaviour of the computational objects.
let c_integrated_abstracted_composed =
    c_composed_abstracted.integrate(ccs)
in

-- Is the integrated, abstracted and composed computational behaviour
-- equivalent to the composed enterprise (system) behaviour?
e_composed.equivalent(c_integrated_abstracted_composed)
```

A.2 Enterprise Processes and Computational Behaviour

```
-- The constraint uses the same operations as the previous section and
-- adds the following.

-- Operation of completion condition that yields the causality target
-- instantiations that track to the step in the process (argument) for
-- which the completion condition is a completion condition.
-- e2b_cti2st tracks a basic causality target instantiation to the
-- enterprise step template from which it was transformed.
context CompletionCondition
    def: ctisForAbstractAction(p: E_ProcessTemplate): Set =
        allCTI->select(cti: CausalityTargetInstantiation |
            cti.e2b_cti2st.in = p
            and(
                cti.e2b_cti2st.action.condition = self
                or
                cti.e2b_cti2st.action.instance.condition->includes(self)
            )
        )

-- The actual constraint is the following.
context E_ProcessTemplate inv:
```

```
-- Basic behaviour instantiations that correspond to computational
-- behaviours performed by computational objects.
-- track c2b'_co2bi tracks computational objects to the basic behaviour
-- instantiations that they perform, c2b' is the transformation that does
-- not transform computational action and constraint templates that
-- represent parts of concurrency mechanisms.
let c_behinst =
   C_Object.allInstances()->collect(co: Object |
      co.c2b'_co2bi
   )->asSequence()
in

-- The basic behaviour representing process that we are comparing to.
-- e2b_p2bi tracks processes to behaviour instantiations that represent
-- them.
let e_process = self.e2b_p2bi
in

-- The behaviour instantiation that is the composition of all
-- instantiations in c_behinst.
let c_composed =
   c_behinst->subSequence(2,c_behinst->size())->
   iterate(bi: BehaviourInstantiation; result:
      BehaviourInstantiation = c_behinst->first() |
      bi.compose(result)
   )
in

-- All causality target instantiations
let allCTI = CausalityTargetInstantiation->allInstances()
in

-- All causality target instantiations that can not be abstracted, because
-- they correspond to final actions or interactions.
-- These are:
-- 1. All causality target instantiations in the composed behaviour that
--    track to an action or interaction contribution in the non-composed
--    behaviour. These non-composed actions and interaction contributions
--    must track to a computational action instantiation that is a final
--    action for a step in the process.
--    c2b'_cti2cat tracks a basic causality target instantiation to the
--    computational action template from which it was transformed.
let finalActions =
   allCTI->select(cti : CausalityTargetInstantiation |
      C_ActionInstantiation.allInstances()->
      exists(cai: C_ActionInstantiation |
         cai.instantiator.c2b'_co2bi=
         cti.basicrules_compose_target2target.bi
         and
         cai.template=
         cti.basicrules_compose_target2target.cti.c2b'_cti2cat
         and
         self.contained.action.condition.disjunction.conjunctionI->
            includes(cai)
      )
   )
in

-- 2. All causality target instantiations in the composed behaviour that
--    track to an interaction in the non-comopsed behaviour. These
--    interactions must track to a computational action template that
--    is a final action for a step in the process.
--    c2b'_i2cat tracks a basic interaction to the computational action
--    template from which it was transformed.
let finalInteractions =
   allCTI->select(cti : CausalityTargetInstantiation |
      self.contained.action.condition.disjunction.conjunctionT->includes(
         cti.basicrules_compose_action2interaction.c2b'_i2cat
      )
   )
in

-- The causality targets, in the transformed and composed computational
```

```
-- behaviour, from which we can abstract.
-- These are causality targets that do not represent final actions,
-- interaction contributions or interactions.
let ctiToAbstract =
    (allCTI - finalActions) - finalInteractions
in

-- The composed computational behaviour in which we abstracted from all
-- inserted actions
let c_composed_abstracted =
    c_composed.abstract(ctiToAbstract)
in

-- Textual versions of all completion conditions that apply to steps in
-- this process
let ccs =
    CompletionCondition.allInstances()->collect(cc: CompletionCondition |
        cc.ctisForAbstractAction(self).name.collect(name: String |
            ' TO '.concat(name))->
                product(cc.leftPartCompletionCondition())->
                    collect(tuple : Tuple(first: String, second: String) |
                        tuple.second.concat(tuple.first)
                    )
                )
        )
    ).flatten()
in

-- The integrated, abstracted and composed behaviour that corresponds to
-- the behaviour of the computational objects.
let c_integrated_abstracted_composed =
    c_abstracted_composed.integrate(ccs)
in

-- Is the integrated, abstracted and composed computational behaviour
-- equivalent to the composed enterprise (system) behaviour?
e_process.equivalent(c_integrated_abstracted_composed)
```

References

Aagedal, J. Ø., & Milosevic, Z. (1999). ODP enterprise language: UML perspective, *Proceedings of the 3rd IEEE International Enterprise Distributed Object Computing Conference (EDOC)* (pp. 60-71).

van der Aalst, W.M.P., & ter Hofstede, A.H.M. (2003). YAWL: Yet Another Workflow Language. *Information Systems, 30*(4), 245-275.

van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., & Barros, A.P. (2003). Workflow Patterns. *Distributed and Parallel Databases, 14*(3), 5-51.

Akehurst, D. H., Derrick, J., & Waters, A. G. (2003a). Addressing computational viewpoint design, *Proceedings of the 7th IEEE Enterprise Distributed Object Computing Conference (EDOC)* (pp. 147-158).

Akehurst, D. H., Kent, S., & Patrascoiu, O. (2003b). A relational approach to defining and implementing transformations between metamodels. *Software and Systems Modeling, 2*(4), 215-239.

Akehurst, D. H., Linington, P. F., & Patrascoiu, O. (2003c). *OCL 2.0: Implementing the standard* (Technical Report No. 1746): University of Kent at Canterbury.

Allen, R., & Garlan, D. (1994). Formalizing architectural connection, *Proceedings of the 16th ACM/IEEE International Conference on Software Engineering (ICSE)* (pp. 71-80).

Allen, R., & Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM), 6*(3), 213-249.

Almeida, J. P. A., Dijkman, R. M., Ferreira Pires, L., Quartel, D. A. C., & Sinderen, M. J. van (2005). Abstract interactions and interaction refinement in model-driven design, *Proceedings of the 9th IEEE EDOC Enterprise Computing Conference (EDOC)* (pp. 273-286).

Balabko, P., & Wegmann, A. (2003). From RM-ODP to the formal behavior representation. In H. Kilov & K. Baclawski (Eds.), *Practical*

foundations of business and system specifications (pp. 41-66): Kluwer Academic Publishers.

Blair, G., & Stefani, J.-B. (1998). *Open distributed processing and multimedia*: Addison Wesley Longman.

Boiten, E., Bowman, H., Derrick, J., & Steen, M. W. A. (1997). Managing inconsistency and promoting consistency (unpublished): University of Kent, Canterbury, United Kingdom.

Boiten, E., Derrick, J., Bowman, H., & Steen, M. W. A. (1999). Constructive consistency checking for partial specification in Z. *Science of Computeer Programming, 35*(1), 29-75.

Boiten, E. A., Bowman, H., Derrick, J., Linington, P. F., & Steen, M. W. A. (2000). Viewpoint consistency in ODP. *Computer Networks, 34*(3), 503-537.

Bordbar, B., Derrick, J., & Waters, A. G. (2002). A UML approach to the design of open distributed systems, *Proceedings of the 4th International Conference on Formal Engineering Methods (ICFEM)* (Vol. 2495 in Lecture Notes in Computer Science, pp. 561-572).

Derrick, J., Boiten, E. A., Bowman, H., & Steen, M. W. A. (1999). Viewpoints and consistency: Translating LOTOS to object-Z. *Computer Standards and Interfaces, 21*, 251-272.

Deursen, A. van, Klint, P., & Visser, J. (2000). Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices, 35*(6), 26-36.

Dijkman, R. M., Quartel, D. A. C., Ferreira Pires, L., & Sinderen, M. J. van (2003). An approach to relate viewpoints and modeling languages, *Proceedings of the 7th IEEE Enterprise Distributed Object Computing Conference (EDOC)* (pp. 14-27).

Dijkman, R. M., Quartel, D. A. C., Ferreira Pires, L., & Sinderen, M. J. van (2004). A rigorous approach to relate enterprise and computational viewpoints, *Proceedings of the 8th IEEE Enterprise Distributed Object Computing Conference (EDOC)* (pp. 187-200).

Eck, P. A. T. van, Blanken, H. M., & Wieringa, R. J. (2004). Project GRAAL: Towards operational architecture alignment. *International Journal of Cooperative Information Systems, 13*(3), 235-255.

Eijk, P. van, Vissers, C., & Diaz, M. (1989). *The formal description technique LOTOS*: North-Holland.

Ferreira Pires, L. (1994). *A framework for distributed systems development.* Ph.D. Thesis. University of Twente, Enschede, The Netherlands.

Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., & Nuseibeh, B. (1994). Inconsistency handling in multi-perspective specifications. *IEEE Transactions on Software Engineering, 20*(8), 569-578.

Glabbeek, R. J. van, & Goltz, U. (2001). Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica, 37*(4/5), 227-327.

Guizzardi, G. (2005). *On the ontological foundations of structural conceptual models.* Ph.D. Thesis. University of Twente, Enschede, The Netherlands.

Guizzardi, G., Ferreira Pires, L., & Sinderen, M. J. van (2005). An ontology-based approach for evaluating the domain appropriateness and comprehensibility appropriateness of modeling languages, *Proceedings of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)*.

Hoare, C. A. R. (1985). *Communicating sequential processes*: Prentice-Hall.

IEEE. (2000). *IEEE recommended practice for architectural description of software-intensive systems* (IEEE Std No. 1471-2000).

ISO/IEC. (2002). *Information technology - Z formal specification notation - syntax, type system and semantics* (Specification 13568).

ISO/IEC/JTC1/SC7. (2004). *Announcement - SC7 study group on the revision of RM-ODP* (Announcement No. N3054).

ITU-T. (2002). *CCITT specification and description language* (Specification Z.100).

ITU-T, & ISO/IEC. (1995). *Open distributed processing reference model (ODP-RM)* (ITU-T Specification 901.4 and ISO/IEC Sprecification 10746-1.4).

ITU-T, & ISO/IEC. (1999). *Information technology - open distributed processing reference model - enterprise language* (ITU-T Specification 911 and ISO/IEC Specification 16414).

ITU-T, & ISO/IEC. (2005). *Information technology - open distributed processing - use of UML for ODP system specification* (Committee Draft Version 1.00 of ITU-T Specification X.906 and ISO/IEC Specification 19793).

Jackson, D. (2002). Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM), 11*(2), 256-290.

Jonkers, H., Lankhorst, M. M., Buuren, R. van, Hoppenbrouwers, S., Bonsangue, M., & Torre, L. van der (2004). Concepts for modelling enterprise architectures. *International Journal of Cooperative Information Systems, 13*(3), 257-287.

Katoen, J.-P. (1995). Causal Behaviours and Nets, *Proceedings of the 16th International Conference on Application and Theory of Petri Nets (ATPN)* (Vol. 935 in Lecture Notes in Computer Science, pp. 258-277).

Lankhorst, M. M. (2005). *Enterprise architecture at work: Modelling, communication and analysis*: Springer.

Lankhorst, M. M., Buuren, R. van, Leeuwen, D. van, Jonkers, H., & Doest, H. ter (2004). Enterprise architecture modelling - the issue of integration. *Advanced Engineering Informatics, 18*(4), 205-216.

Linington, P. F., Vallecillo, A., & Wood, B. (2004). Report of the 2004 workshop on ODP for enterprise computing (WODPEC). Retrieved 4 August 2005, from http://www.lcc.uma.es/~av/wodpec2004/WODPEC2004-Report.doc

Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D., & Mann, W. (1995). Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering, 21*(4), 336-355.

Luckham, D. C., & Vera, J. (1995). An event-based architecture definition language. *IEEE Transactions on Software Engineering, 21*(9), 717-734.

Magee, J., Dulay, N., Eisenbach, S., & Kramer, J. (1995). Specifying distributed software architectures, *Proceedings of the 5th European Software Engineering Conference (ESEC)* (Vol. 989 in Lecture Notes in Computer Science, pp. 137-153).

Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering, 26*(1), 70-93.

Merriam-Webster. (2005). Merriam-webster online. Retrieved 2 October 2005, from http://www.m-w.com/

Milner, R. (1999). *Communicating with mobile agents: The Pi-calculus*: Cambridge University Press.

Naumenko, A. (2002). *Triune continuum paradigm: A paradigm for general system modeling and its applications for UML and RM-ODP.* Ph.D. Thesis. École Politechnique Fédérale de Lausanne, Switzerland.

Nuseibeh, B., Kramer, J., & Finkelstein, A. (1993). Expressing the relationships between multiple views in requirements specification, *Proceedings of the 15th ACM/IEEE International Conference on Software Engineering (ICSE)* (pp. 187-196).

Nuseibeh, B., Kramer, J., & Finkelstein, A. (1994). A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering, 20*(10), 760-773.

Object Management Group. (2002a). *Meta object facility (MOF) specification* (Available Specification No. formal/02-04-03).

Object Management Group. (2002b). *OMG XML metadata interchange (XMI) specification* (Available Specification No. formal/02-01-01).

Object Management Group. (2002c). *Request for proposal: MOF 2.0 query / views / transformations RFP* (Request for Proposal No. ad/02-04-10).

Object Management Group. (2002d). *UML profile for EDOC* (Final Adopted Specification No. ptc/02-02-05).

Object Management Group. (2003a). *MDA guide version 1.0.1* (No. omg/2003-06-01).

Object Management Group. (2003b). *UML 2.0 infrastructure specification* (Final Adopted Specification No. ptc/03-09-15).

Object Management Group. (2003c). *UML 2.0 OCL specification* (Final Adopted Specification No. ptc/03-10-14).

Object Management Group. (2004a). *UML 2.0 superstructure specification* (Final Adopted Specification No. ptc/04-10-02).

Object Management Group. (2004b). *UML profile for meta object facility (MOF) specification* (Available Specification No. formal/04-02-06).

Object Management Group. (2005). *Revised submission for MOF 2.0 query / view / transformation RFP* (Revised Submission No. ad/05-03-02).

Patrascoiu, O. (2004a). Mapping EDOC to web services using YATL, *Proceedings of the 8th IEEE Enterprise Distributed Object Computing Conference (EDOC)* (pp. 286-297).

Patrascoiu, O. (2004b). YATL: Yet another transformation language, *Proceedings of the 1st European Workshop on Model Driven Architecture for Industrial Applications (MDA-IA)* (pp. 83-90).

Pressman, R. S. (2003). *Software engineering: A practitioner's approach* (5th ed.): McGraw-Hill.

Quartel, D. (1998). *Action relations - basic design concepts for behaviour modelling and refinement.* Ph.D. Thesis. University of Twente, Enschede, The Netherlands.

Quartel, D., Ferreira Pires, L., Franken, H., & Vissers, C. (1995). An engineering approach towards action refinement, *Proceedings of the 5th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS)* (pp. 266-273).

Quartel, D., Ferreira Pires, L., Sinderen, M. J. van, Franken, H. M., & Vissers, C. A. (1997). On the role of basic design concepts in behaviour structuring. *Computer Networks and ISDN Systems, 29*(4), 413-436.

Quartel, D., Ferreira Pires, L., & Sinderen, M. van (2002). On architectural support for behavior refinement in distributed systems design. *Journal of Integrated Design and Process Science, 6*(1).

Quartel, D. A. C., Dijkman, R. M., & Sinderen, M. J. van (2005). Extending profiles with stereotypes for composite concepts using model transformation, *Proceedings of the 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS)*.

Romero, J. R., & Vallecillo, A. (2005). Modeling the ODP computational viewpoint with UML 2.0, *Proceedings of the 9th IEEE EDOC Enterprise Computing Conference (EDOC)* (pp. 169-180).

Russell, N., van der Aalst, W.M.P., ter Hofstede, A.H.M., & Edmond, D. (2005). Workflow Resource Patterns: Identification, Representa-

tion and Tool Support, *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE)* (Vol. 3520 in Lecture Notes in Computer Science, pp. 216-232).

Russell, N., ter Hofstede, A.H.M., Edmond, D., & van der Aalst, W.M.P. (2005). Workflow Data Patterns: Identification, Representation and Tool Support, *Proceedings of the 24th International Conference on Conceptual Modeling (ER)* (pp. 353-368).

Sinderen, M. J. van (1995). *On the design of application protocols.* Ph.D. Thesis, University of Twente, Enschede, The Netherlands.

Smith, G. (2000). *The object-Z specification language. Advances in formal methods*: Kluwer Academic Publishers.

Sowa, J. F., & Zachman, J. A. (1992). Extending and formalizing the framework for information systems architecture. *IBM Systems Journal, 31*(3), 590-616.

Spanoudakis, G., Finkelstein, A., & Till, D. (1999). Overlaps in requirements engineering. *Automated Software Engineering, 6*(2), 171-198.

Steen, M. W. A., & Derrick, J. (2000). ODP enterprise viewpoint specification. *Computer Standards and Interfaces, 22*(3), 165-189.

The Open Group. (2005). *The open group architecture framework (TOGAF -- 'the book') version 8.1*: The Open Group.

Vissers, C. A., Ferreira Pires, L., & Lagemaat, J. van de (1995). LOTOSphere, an attempt towards a design culture. In T. Bolognesi, J. van de Lagemaat & C. A. Vissers (Eds.), *LOTOSphere: Software development with LOTOS*. Dordrecht: Kluwer Academic Publishers.

Vissers, C. A., Sinderen, M. J. van, & Ferreira Pires, L. (1993). What makes industries believe in formal methods. In A. A. S. Danthine, G. Leduc & P. Wolper (Eds.), *Proceedings of the 13th IFIP International Symposium on Protocol Specification, Testing and Verification (PSTV)* (pp. 3-26).

Wegmann, A. (2003). On the systemic enterprise architecture methodology (seam), *Proceedings of the 5th International Conference on Enterprise Information Systems - Volume III* (pp. 483-490).

Wieringa, R. J., Blanken, H. M., Fokkinga, M. M., & Grefen, P. W. P. J. (2003). Aligning application architecture to the business context. In J. Eder & M. Missikoff (Eds.), *Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE)* (Vol. 2681 in Lecture Notes in Computer Science, pp. 209-225).

Workflow Management Coalition. (1999). *Workflow management coalition terminology and glossary version 3.0* (Specification No. WFMC-TC-1011).

Zachman, J. A. (1987). A framework for information systems architecture. *IBM Systems Journal, 26*(3), 276-295.

Index

Samenvatting

Dit proefschrift beschrijft een raamwerk dat helpt bij het bewaken van de consistentie in een ontwerp vanuit meerder ontwerpperspectieven. In een dergelijk ontwerp construeert iedere belanghebbende zijn eigen ontwerpdeel. We noemen dat deel de '*view*' van de belanghebbende. Om zijn gedeelte van het ontwerp te construeren, heeft een belanghebbende een '*viewpoint*' dat de ontwerpconcepten, de notatie en de ondersteunende software beschrijft die de belanghebbende gebruikt.

Het raamwerk dat in dit proefschrift beschreven wordt, richt zich op *architectuurontwerp* van *gedistribueerde systemen*.

Een gedistribueerd systeem is een systeem waarin het gedrag wordt uitgevoerd door fysiek gescheiden systeemdelen. Interactie tussen de systeemdelen speelt dan een belangrijke rol. Een voorbeeld van een gedistribueerd systeem is een mobiel communicatienetwerk. In een dergelijk netwerk wordt het gedrag uitgevoerd door bijvoorbeeld mobiele telefoons, antennes waarmee die mobiele telefoons verbinden en computers bij de netwerkaanbieder.

Architectuurontwerp is het gebied van ontwerp dat zich richt op de hogere abstractieniveaus in een ontwerpproces. Het laagste abstractieniveau dat onder de architectuurontwerp valt, is het niveau waarin het gedrag wordt uitgevoerd door delen die communiceren met behulp van een communicatie middleware.

In het raamwerk wordt consistentie bewaakt door relaties en consistentieregels die gedefinieerd worden door de belanghebbenden. Relaties specificeren hoe een 'view' gerelateerd kan zijn aan een andere 'view' en consistentieregels specificeren regels die moeten gelden in een consistent ontwerp.

Om te helpen bij het bewaken van de consistentie definieert het raamwerk:

– een verzameling basisconcepten;
– voorgedefinieerde relaties;

– voorgedefinieerde consistentieregels; en
– een taal om de relaties en consistentieregels te representeren.

De basisconcepten hebben we overgenomen uit eerder werk. Deze concepten zijn ontwikkeld door het gebied van gedistribueerd systeemontwerp te bestuderen. In het raamwerk moeten 'viewpoint'-specifieke concepten worden gedefinieerd als composities of specialisaties van de basisconcepten. Daarmee vormen de basisconcepten een gemeenschappelijke vocabulaire, die de verschillende belanghebbenden kunnen gebruiken om elkaars ontwerpen te begrijpen.

Het raamwerk definieert relaties die kunnen worden hergebruikt om te specificeren hoe een 'view' is gerelateerd aan een andere 'view'. De twee hoofdtypen van relaties die worden voorgedefinieerd zijn: de *verfijningsrelatie* en de *overlap relatie*. Een verfijningsrelatie bestaat tussen 'views' die (deels) dezelfde ontwerpaspecten beschouwen, maar die deze beschouwen op verschillende abstratieniveaus. Een overlap relatie bestaat tussen 'views' die (deels) dezelfde ontwerpaspecten beschouwen op hetzelfde abstractieniveau. De voorgedefinieerde relaties zijn ontwikkeld door bestaande raamwerken voor ontwerp vanuit verschillende perspectieven te analyseren en er de veel gebruikte relaties uit af te leiden.

De voorgedefinieerde relaties impliceren bepaalde consistentieregels. Daarom definieert het raamwerk de volgende herbruikbare consistentieregels die de voorgedefinieerde relaties complementeren. Als twee 'views' een verfijningsrelatie hebben, dan moet de ene 'view' de eigenschappen die de andere view voorschrijft handhaven. Als twee views een overlap relatie hebben, dan moeten ze equivalent zijn met betrekking tot hun overlap.

Het raamwerk definieert ook een architectuur voor softwareondersteuning. Die ondersteuning is erop gericht om de relaties en consistentieregels tussen 'views' te specificeren en om de consistentieregels te evalueren. De architectuur bevat de voorgedefinieerde basisconcepten, relaties en consistentieregels, zodat die kunnen worden hergebruikt.

Als casestudie definiëren we, met behulp van het raamwerk, aangepaste versies van de RM-ODP 'enterprise', 'computational' en 'information viewpoints'. We definiëren de concepten uit deze 'viewpoints' als composities of specialisaties van de basisconcepten. Ook definiëren we de relaties en consistentieregels tussen de 'viewpoints', gebruikmakend van de voorgedefinieerde relaties en consistentieregels. De resultaten van de casestudie steunen de hypothese dat het raamwerk helpt bij het bewaken van de consistentie in een ontwerp met meerder ontwerpperspectieven.

CONSISTENCY IN MULTI-VIEWPOINT ARCHITECTURAL DESIGN

Remco Dijkman

The design of large-scale distributed applications involves the viewpoints of many different stakeholders, such as business analysts, software architects and end-users. Each of these stakeholders uses his own design languages and tools to construct a part of the design from his viewpoint. This presents us with major challenges in maintaining the consistency between the designs of the different stakeholders. The framework presented in this thesis helps to maintain this consistency.

Using our framework, consistency is preserved through inter-viewpoint relations and consistency rules that must be specified by the stakeholders. The framework supports the specification of such relations and rules by providing: (i) a set of basic concepts that stakeholders can use as a common basis to understand each others concepts; (ii) pre-defined relations and consistency rules that stakeholders can re-use to specify inter-viewpoint relations and consistency rules; and (iii) an architecture for a tool-suite that supports enforcing these relations and rules.

We demonstrate the applicability of our framework by applying it to the viewpoints that are defined by the Reference Model for Open Distributed Processing.

ISBN 90-75176- 80-5

CTIT

Telematica Instituut