MODEL-DRIVEN DESIGN OF DISTRIBUTED APPLICATIONS

JOÃO PAULO ANDRADE ALMEIDA

Model-Driven Design of Distributed Applications

Model-Driven Design of Distributed Applications

João Paulo Andrade Almeida

Enschede, The Netherlands, 2006

CTIT Ph.D.-Thesis Series, No. 06-85
Telematica Instituut Fundamental Research Series , No. 018 (TI/FRS/018)

Cover Design:	Studio Oude Vrielink, Losser and Jos Hendrix, Groningen
Cover Photo:	Alexander Calder, *Red Lily Pads*, 1956
	Painted sheet metal, metal rods and wire
	42 x 201 x 109 inches (106. 7 x 510.6 x 276.9 cm)
	Solomon R. Guggenheim Museum, New York
	65.1737
	Photograph by David Heald
	© The Solomon R. Guggenheim Foundation, New York
Book Design:	Lidwien van de Wijngaert and Henri ter Hofte
Printing:	Universal Press, Veenendaal, The Netherlands

CTIT Ph.D.-Thesis Series, No. 06-85
ISSN 1381-3617; No. 06-85
Centre for Telematics and Information Technology, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

Telematica Instituut Fundamental Research Series, No. 018
ISSN 1388-1795; No. 018
ISBN 90-75176-422
Telematica Instituut, P.O. Box 589, 7500AN Enschede, The Netherlands

MODEL-DRIVEN DESIGN OF DISTRIBUTED APPLICATIONS

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 donderdag 1 juni 2006 om 15.00 uur

door
João Paulo Andrade Almeida
geboren op 18 juli 1977
te Vila Velha, Espírito Santo, Brazilië

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

Abstract

A recent trend in the design of distributed applications is to systematically separate their platform-independent and platform-specific aspects, by describing them in separate models. The main benefits of this approach stem from the possibility to derive different platform-specific models (PSMs) from the same platform-independent model (PIM), and to partially automate the model transformation process and the realization of the distributed application on specific target (middleware) platforms. This may reduce initial development costs and improve software quality, but also forms the basis for facilitating evolution and migration of software solutions, hence contributing to the limitation of maintenance costs for distributed applications.

A prominent development in this trend is the Model-Driven Architecture (MDA) approach. In the context of MDA, much effort has been invested in enabling technologies and techniques for model-driven design, which include metamodelling (MOF), language definition and extension mechanisms (e.g., UML and UML profiles), model transformation specification languages (MOF Query/View/Transformation), tool support and tool chain interoperability. In contrast, the methodological and architectural foundations of platform-independent design have received little attention.

In particular, the state-of-the-art in model-driven design can be criticized on a number of points:

- there is a lack of guidelines to select abstraction criteria and modelling concepts for platform-independent design;
- there is little methodological support to distinguish between platform-independent and platform-specific concerns, which is detrimental to the beneficial exploitation of the separation between PIMs and PSMs;
- the distinction between PIM-PSM is coarse and insufficient to cope with the diversity of application requirements and platform characteristics;
- little attention is given to the role of platform characteristics throughout the development trajectory, possibly leading to models with unaccept-

able levels of platform-independence and applications with unacceptable quality attributes;
- the behavioural aspects of designs are largely ignored, and;
- design operations between PIMs and PSMs are not clearly defined, thus inhibiting their effective application in model transformation.

This thesis aims at proposing a design approach for the development of distributed applications that addresses the problems mentioned above, focusing particularly on middleware-platform-independence. This approach consists of:
- a *design process*, which results in application designs at different levels of abstraction and platform-independence;
- the *notion of an abstract platform*, which defines the platform characteristics that are relevant for an application design at a certain level of platform-independence;
- a *set of design quality criteria* for abstract platform definition; and,
- a *design framework*, which aims at supporting a designer in defining abstract platforms and platform-independent designs. This design framework consists of two parts: *a set of basic design concepts*, which are used at different levels of platform-independence to describe both abstract platforms and the platform-independent designs that rely on them, and *design operations*, which can be used in transformations to bridge between different levels of platform-independence. The use of the design framework enables designers to make statements about the conformance of models at different levels of platform-independence.

The design process is structured into a *preparation* and an *execution phase*. In the preparation phase, designers identify (and, when necessary, define) the required levels of models, their abstract platforms and the modelling language(s) to be used. In addition, a designer may also identify or define transformations between related levels of models. The results of the preparation phase are used in the execution phase, which entails the creation of models of an application using specific modelling languages and abstract platforms.

The main aspects of the approach are illustrated with a case study involving the design of context-aware mobile services. We define three levels of models: a platform-independent service specification level, a platform-independent service design level and a platform-specific service design level. Particular attention is given to the representation and transformation of behavioural aspects of service designs.

Acknowledgements

From all the things I have come to appreciate in the process that led to this book, *gratitude* just stands out! Therefore, this section on acknowledgements has special significance to me.

I would like to start by thanking Marten, Luís and Chris. They have supported me unconditionally and gave me freedom to explore! Together, they have created a stimulating environment in the Architecture group. Their previous work (and that of Dick and other “Visserians” [45]) has been constant inspiration to me. We share a passion (or obsession?) for architectural design and I hope we will have the chance to work together for a long time to come.

I would like to thank the members of my defense committee: Prof. dr. Mehmet Akşit, Prof. dr. Colin Atkinson, Prof. dr. Jos van Hillegersberg, Prof. dr. Peter Linington and Prof. dr. ir. Bart Nieuwenhuis. It is an honour to have you in this committee. Bart in particular should be acknowledged for supervising me at the beginning of my Ph.D. trajectory.

Giancarlo Guizzardi and Remco Dijkman have been my closest peers throughout this period. They are extremely intelligent and we have had such great discussions about all kinds of thinkable things. It was a pleasure to share these years with them, and I would like to thank them for that.

Maarten Wegdam has been a constant factor of life in the Netherlands; first as my M.Sc. supervisor at Lucent, then as a colleague at the University of Twente, but mostly as a friend. In fact, with Susan and now “Polycarpus” they form part of what I could call my “extended Dutch family”.

The “Macandra crowd” (Maarten Schokker, Diana, Sander, Azita, Ronald, Aleks, Ivo, Daan, Femke, Arjan, Maartje, and so many others) and the “extended Brazilian family” (Gian, Renatinha, Pablo, Flávia, Cléver, Kellen, Ciro, Léo, Diego, Sonia, Jaque, Luiz Olavo, Ricardo, Tiago) have spared no efforts to make life so *gezellig* in Enschede! The same applies to other (former) members of the ASNA group (Remco van de Meent, Tom, Annelies, etc.). I would like to thank you all for that.

I would also like to thank José Gonçalves. I would not be here in Twente if it weren't for our visit to the UT in 1998. He was also part of the "extended Brazilian family", when he was a visiting professor in our group in 2002 and 2003.

Remco Dijkman should also be acknowledged for providing the ISDL metamodels used in the case study. Bert and Dick provided prompt support for the modelling and simulation tools (Grizzle and Sizzle) used in the case study. Marten provided the Dutch translation of the abstract.

I would like to thank my colleagues at the Telematica Instituut for the fruitful cooperation in the scope of the A-MUSE project, and for welcoming me as part of the team so quickly. In this respect, Maria deserves special mention.

My parents have sparked my interest in research a long time ago, most likely without noticing. They used to travel to conferences to meet interesting people and to have fun discussing and learning. Together with my brother and my sister, they are an essential part of who I am.

My wife Patrícia has left the country she loves so much to be here by my side "under the low sky". *This book is dedicated to her, to her courage and love.*

João Paulo Andrade Almeida,
Enschede, May 2006.

Contents

Chapter 1

Introduction

This thesis proposes a model-driven design methodology for distributed applications. The main characteristic of this methodology is that it strives for obtaining application models that are independent of the technology platforms upon which the application is built. In this way, these models can be reused for realization on different technology platforms and they are more resilient to impact when platforms change. This chapter presents the motivation of this thesis and outlines the main research objectives as well as the approach adopted.

This chapter is organised as follows: section 1.1 provides some background for our work; section 1.2 outlines the main issues in the state-of-the-art in model-driven design; section 1.3 presents our research objectives; section 1.4 discusses briefly the approach proposed in this thesis; section 1.5 defines the scope of the work; finally, section 1.6 presents the structure of this thesis.

1.1 Background

The wide spread of Information and Communications Technologies (ICT) and the establishment of the Internet have popularized innumerable distributed applications beyond most predictions. Distributed applications have increasingly come to occupy a central place in business, science, engineering, and everyday life.

Important characteristics of distributed applications include *remoteness*, *concurrency*, *lack of global state*, *independent failures*, *asynchrony*, *heterogeneity*, and *autonomy*. These and other characteristics pose many challenges for the development of distributed applications. As a result, the timely development of high quality distributed applications is expensive.

Since a great amount of effort is invested in the development of distributed applications, an important quality of these applications is their ability to survive the impact of change, both with respect to changes in application requirements and with respect to changes in the technologies used to build the application.

In the last decades, the development of distributed applications has been facilitated to some extent by the introduction of distribution infrastructures such as middleware platforms. These infrastructures offer generic distribution support for distributed applications, masking from application designers some details and differences in the support offered by programming languages, operating systems and network protocols. Since a significant amount of development effort is spent on overcoming problems related to distribution and in exploiting distribution beneficially (e.g., to achieve performance and dependability), the reuse of middleware platforms significantly increases the efficiency of application development.

Different middleware platforms have been developed in the last decades, e.g., CORBA/CCM [73], J2EE [102] (including EJB [103] and JMS [104]), DCE [109], and Web Services [120, 121]. Currently, designers of distributed applications are exposed to a multitude of platform standards, implementations of standards from different vendors, proprietary platforms and ad hoc infrastructures, standard and proprietary extensions to platforms, etc. Despite standardization efforts, different parts of a distributed application may be built using various middleware platforms, and the set of platforms used may change over time. In addition, middleware platforms may evolve during the lifetime of applications. The use of a single immutable distribution infrastructure is therefore not envisioned as a long term solution for the support of distributed applications.

Since different middleware platforms provide different constructs from which applications can be built, the design of an application in terms of platform constructs is platform-specific. This means that application designs may be affected by changes in technology platforms, with the consequence that applications have to be redesigned. Furthermore, application designers must be knowledgeable about the peculiarities of specific target middleware platforms.

1.2 Motivation

A recent trend in the design of distributed applications is to systematically separate their middleware-platform-independent and middleware-platform-specific aspects, by describing them in separate models. A prominent development setting this trend is the Model-Driven Architecture (MDA) [72, 76] approach.

The notion of platform-independence is central to MDA development. Platform-independence is a quality of a model that relates to the extent to which the model abstracts from the characteristics of particular technology platforms. A common pattern of MDA development is to define a platform-independent model (PIM) , and to apply (parameterised) transformations to this PIM to obtain one or more platform-specific models (PSMs). The main benefits of this approach stem from the possibility to derive different PSMs from the same PIM, and to partially automate the model transformation process and the realization of the distributed application on specific target platforms. This may reduce development costs and improve software quality, but also forms the basis for facilitating evolution and migration of software solutions, hence contributing to limiting the maintenance costs for distributed applications.

In the context of MDA, much effort has been invested in enabling technologies and techniques for model-driven development, which include metamodelling (MOF) [77, 78], language definition and extension mechanisms (e.g., UML and UML profiles) [81, 83, 84], model transformation specification languages and approaches [64, 79], tool support and tool chain interoperability [20]. In contrast, the methodological and architectural foundations of platform-independent design have received little attention.

In particular, the following research questions remain open:

– Which abstraction criteria should be used for platform-independent design? Which concepts should be used to describe platform-independent models of an application?
– How should designers distinguish platform-independent and platform-specific concerns, in order to effectively exploit the PIM-PSM separation of concerns?
– Is the distinction between PIMs and PSMs sufficient to cope with the diversity of application requirements and infrastructure characteristics? Should there be more levels of models (or levels of platform-independence)?
– What are the implications of the separation of platform-independent and platform-specific concerns for the design process? How should the design process be organized?
– Is there a trade-off between platform-independence and other relevant design quality characteristics?
– How should designers cope with platform characteristics along the design trajectory? When and how should restrictions imposed by platforms be incorporated in designs?
– What are the relations between the various models in the design trajectory?
– How to represent behaviour in a platform-independent way?

– Does the focus on a particular design language (e.g., UML) constrain the designer? If so, how?

1.3 Research objectives

In order to obtain the potential benefits of the model-driven approach to the development of distributed applications, we aim at coping with the issues above in an effective model-driven design methodology. The objective of our work is to propose such a methodology for the design of distributed applications so that:

– available and future distribution infrastructures can be (re-)used, improving the efficiency of the design process;
– the knowledge used to perform various design operations can be captured and re-used to improve the overall efficiency of the design process, and;
– the design of applications can be to a certain extent platform-independent, so that these designs can be reused to target different middleware platforms and applications can outlive platforms upon which they are built.

The methodology is defined so as to be generic with respect to application domains and platform characteristics. We propose generic guidelines, which can be applied by designers in specific application domains and with particular requirements on target platforms.

We regard platform-independence as a quality characteristic of strategic importance for distributed application models. Similarly to many other quality characteristics, such as, e.g., adaptability and tailorability, achieving platform-independence is not trivial and requires proper methodological support.

We believe that platform-independence can only be defined once general capabilities of potential target platforms can be established. This leads to the observation that there can be platform-independent models at different abstraction levels, depending on whether one wants to consider different sets of target platforms. Another observation is that different application domain characteristics or different sets of target platforms generally lead to different types of (intermediate) models, design structures or patterns, and model transformations. We have investigated these types of models and design structures and formulated proper design criteria and architectural concepts to support the design trajectory.

1.4 Approach

An architectural concept that plays an important role in our approach is that of an *abstract platform*. An abstract platform defines an acceptable platform from an application developer's point of view, representing the platform support that is assumed by the application developer at some point in the design trajectory. Alternatively, an abstract platform defines characteristics that must have proper mappings onto the set of target platforms that are considered for a design process, thereby defining the level of platform-independence for this particular process.

Because of the variety of application domain characteristics and middleware platform characteristics, different abstract platforms may be required. Therefore, we do not provide a comprehensive catalogue of abstract platforms. Instead, we provide methodological support to design abstract platforms.

The design methodology proposed in this thesis can be decomposed into the following main elements:

- a *design process*, which results in application designs at different levels of abstraction and platform-independence;
- the *notion of an abstract platform*, which defines the platform characteristics that are relevant for an application design at a certain level of platform-independence;
- a *set of design quality criteria* for abstract platform definition; and,
- a *design framework*, whose purpose is to support a designer in defining abstract platforms and platform-independent designs. This design framework consists of two parts: *a set of basic design concepts*, which are used at different levels of platform-independence to describe both abstract platforms and the platform-independent designs that rely on them, and *design operations*, which can be used in transformations to bridge between different levels of platform-independence. The use of the design framework enables designers to make statements about the conformance of models at different levels of platform-independence.

The design process is structured into a *preparation* and an *execution phase*. In the preparation phase, designers identify (and, when necessary, define) the required levels of models, their abstract platforms and the modelling language(s) to be used. In addition, a designer may also identify or define transformations between related levels of models. The results of the preparation phase are used in the execution phase, which entails the creation of models of an application using specific modelling languages and abstract platforms.

1.5 Scope and non-objectives

The scope of this work is the architectural design of distributed ICT applications. We do not address application requirements engineering and we do not propose specific implementation techniques. Furthermore, testing, deployment, operation and retirement activities are outside the scope of this research.

We focus on the methodological aspects of model-driven design, in particular with respect to achieving middleware-platform-independence. Therefore, it is not our intention to propose model transformation approaches or specification languages. It is not our intention either to propose metamodelling, language definition and extension mechanisms or modelling languages, tools and tool architectures. Nevertheless, some developments in these areas provide support for the practical application of our approach. This is illustrated in chapters 6 and 7 of this thesis.

1.6 Thesis structure

This thesis is further structured as follows:

- Chapter 2 (Model-driven design process) identifies the elementary concepts of our approach (such as design process, design, abstraction, model); introduces the notions of platform, platform-independence and abstract platform; and presents an overview of our approach to the design of distributed applications.
- Chapter 3 (Methodological guidelines for the preparation phase) presents methodological guidelines for platform-independent design, specifically addressing the definition of abstract platforms and discussing the role of transformations in the design process.
- Chapter 4 (Separation of concerns and dependencies between models) discusses the implications of the dimensions of separation of concerns (as proposed in chapter 3) to the design process. The dependencies between the various models is visualised and analysed using Design Structure Matrices (DSMs). This results in guidelines for the design process.
- Chapter 5 (Design framework) defines a design framework, whose purpose is to support a designer in defining abstract platforms and platform-independent designs.
- Chapter 6 (Support for abstract platforms in MDA) discusses how abstract platforms can be represented in the modelling infrastructure provided in the MDA, which includes extensions to the Unified Modelling Language (UML) and the use of the Meta-Object Facility (MOF).

- Chapter 7 (Case study: the design of Freeband Services) presents a case study to illustrate the main aspects of our approach. This case study involves the design of context-aware mobile services.
- Chapter 8 (Conclusions) concludes by outlining the main contributions of this thesis and by proposing topics for further investigation.

Figure 1-1 shows the chapters in this thesis and how they can be related to each other. The main body of this thesis consists of two parts: (i) the description of the design approach (chapters 2 to 5) and (ii) the application of our approach (chapters 6 and 7). Related work is discussed throughout the thesis (but with specific considerations about related work in sections 2.5, 5.5, 5.7 and 6.7).

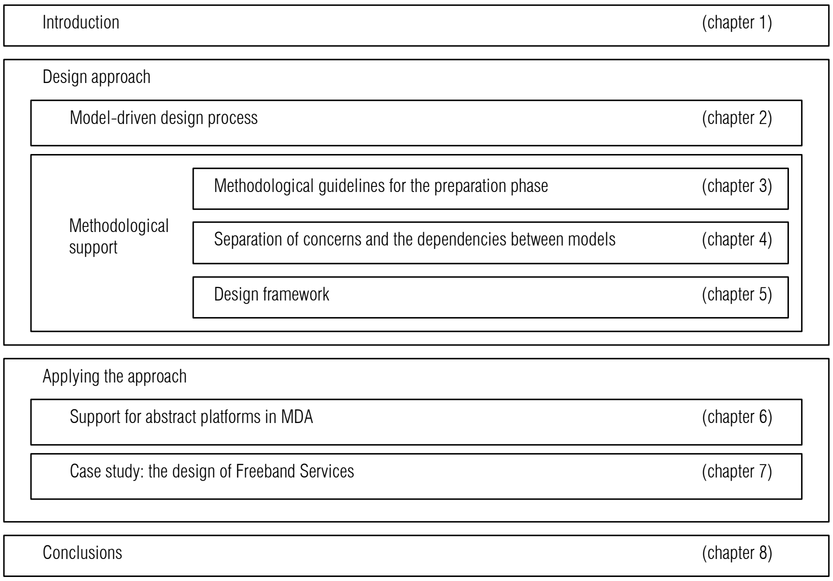

Introduction (chapter 1)
Design approach
Model-driven design process (chapter 2)
Methodological support
Methodological guidelines for the preparation phase (chapter 3)
Separation of concerns and the dependencies between models (chapter 4)
Design framework (chapter 5)
Applying the approach
Support for abstract platforms in MDA (chapter 6)
Case study: the design of Freeband Services (chapter 7)
Conclusions (chapter 8)

Figure 1-1 Structure of this thesis

Chapter 2

Model-driven design process

This chapter presents an overview of our approach to the design of distributed applications. We discuss the role of the separation of platform-independent and platform-specific concerns in the design process, and provide a general methodological framework for platform-independent design, based on the notion of an abstract platform. We outline the design activities and define the scope of our design methodology.

This chapter is organised as follows: section 2.1 introduces some basic concepts; section 2.2 discusses the notions of platform and platform-independence; section 2.3 introduces the concept of abstract platform; section 2.4 provides an overview of the proposed design process; section 2.5 discusses related work; and, section 2.6 presents concluding remarks.

2.1 Basic concepts

2.1.1 Design process

The design of a distributed application can be regarded as the process of building a *realization* of the application that satisfies user requirements while applying a design methodology. This simplistic view of a design process is depicted in *Figure 2-1*.

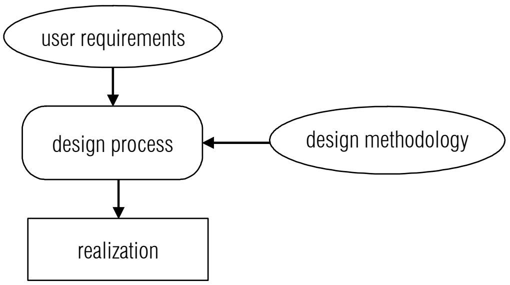


Figure 2-1 Simplistic view of a design process

Since the gap between user requirements and realization is wide, the design of a distributed application is a complex task. Therefore, it is difficult to perform it in a single step. In order to deal with the complexity of this task, a designer should address only a limited set of design concerns in each of a series of *design steps*. This constitutes a basic design principle of effective design methodologies called *separation of concerns*.

A means to achieve separation of concerns in a design process is to use abstraction. *Abstraction* is the process of addressing only the characteristics of an entity that are relevant from a particular point of view. Characteristics that are considered irrelevant are ignored or suppressed. The term abstraction is also used to refer to the result of the process of abstraction. We call an abstraction of a technical object of concern a *design* [40].

In the *stepwise design approach*, concerns are addressed sequentially in design steps, leading to designs of the system at different levels of abstraction. The application of stepwise design in the design process is depicted in *Figure 2-2*.

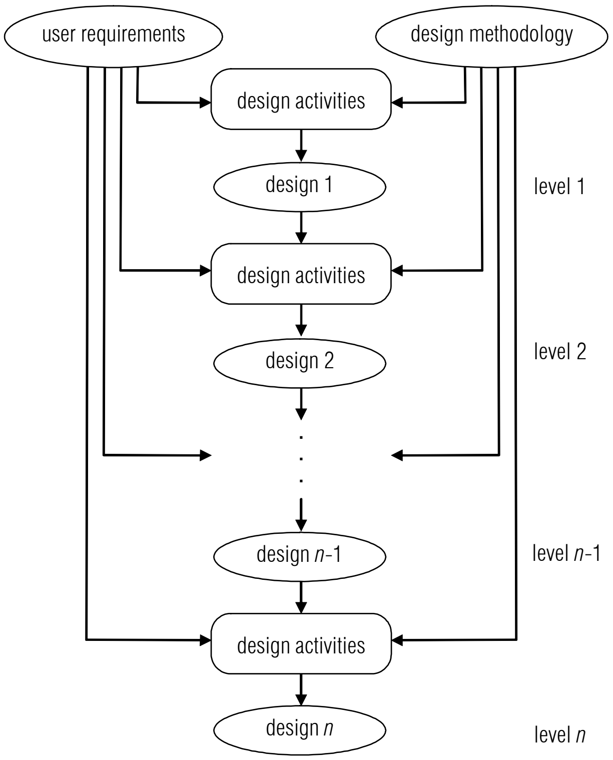


Figure 2-2 Stepwise design

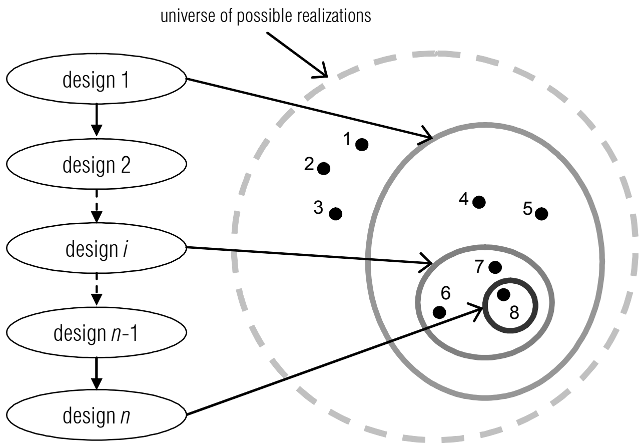
For each design step, *design activities* are executed, which consist of transformation and assessment activities [94]. A *transformation activity* is a generic design activity that entails the production of a target design on basis of some input, and, an *assessment activity* is a generic design activity that comprises the evaluation of the target design as outcome of the transformation activity. In principle, assessment activities should include *conformance assessment*, in order to check whether the target design *conforms* to the original design.

Transformation activities in a design step typically entail selection from a virtually infinite number of potential alternative realizations. The characteristics of these potential realizations are constrained by a design, which defines relevant characteristics of realizations at a particular level of abstraction.

2.1.2 Design decisions

During the design process, design activities result in a number of design decisions, which add characteristics that will eventually be assigned to the realization of a design. We define a *design decision* as a modification of a design that reduces the number of elements of the class of conformant realizations of that design. The reduction of the realization space imposed by successive design decisions is depicted in *Figure 2-3* (inspired by [94]).

Figure 2-3 Reduction of realization space for designs at different levels of abstraction



1..8 – examples of elements from the universe of possible realizations
4..8 – examples of elements from the set of possible realizations defined by design 1
6..8 – examples of elements from the set of possible realizations defined by design *i*
8 – example of element from the set of possible realizations defined by design *n*

Design decisions taken in a design step should meet two requirements in order for the design process to make progress [63]:

- they must preserve the characteristics present in the design that is input to the design step, i.e., the resulting design should conform to the original design; and,
- they must contribute to satisfying requirements that have not yet being fulfilled.

The former requirement reveals the importance of conformance assessment in a design step. The latter requirement reveals the importance of user requirements throughout the design process. This is because user requirements can be stated in terms of characteristics of a realization that are only addressed at a lower level of abstraction. For example, a user may require an application to be deployed in a particular hardware architecture, e.g., because of the availability of this architecture. This requirement is not considered in a design that defines the functions of the application; it is only addressed when hardware characteristics become relevant in the design process. The importance of user requirements throughout the design process justifies the arrows from user requirements to design activities at different levels of abstraction in *Figure 2-2*.

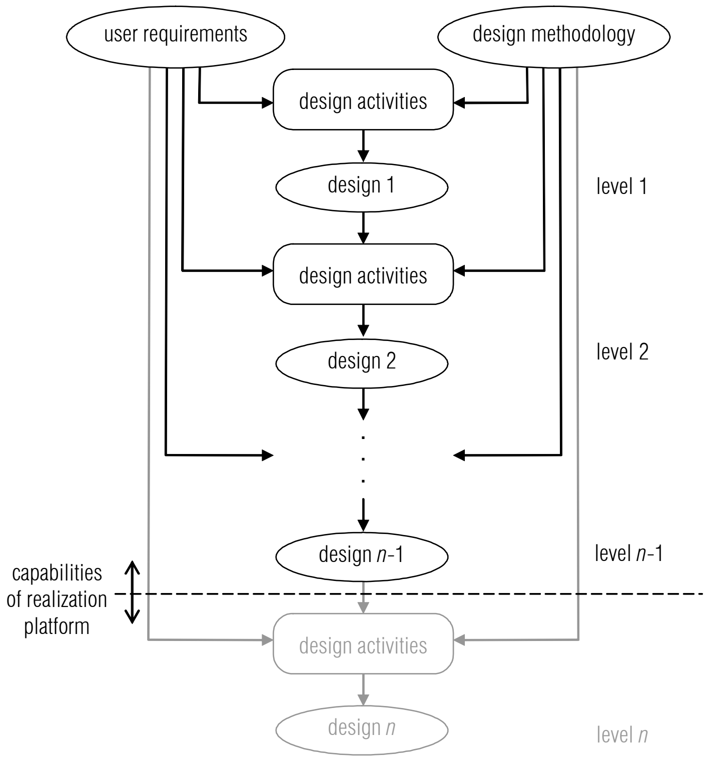
Design decisions should eventually lead to a design that defines all relevant characteristics of an acceptable realization of the system. This design is such that its correspondence with the eventual realization is straightforward.

2.1.3 Realization platform

The realization is defined in terms of realization resources such as, e.g., programming languages and their interpreters or compilers, operating systems and hardware. We call the resources from which a realization can be constructed a *realization platform*.

When capabilities of the realization platform are enhanced, the correspondence between the design and the realization may be established at a higher level of abstraction. In that case, a designer may stop performing design steps earlier, using higher-level constructs entailed by the realization platform. The resulting design process is depicted in *Figure 2-4*. The position of the dashed line that defines the lower boundary of the design process can be adjusted according to the resources available in the realization platform.

Figure 2-4 Enhancement of realization platform



When correspondence between design and realization can be established at a higher level of abstraction, the realization platform embodies design decisions taken at a lower level of abstraction. These design decisions must be consistent with user requirements yet to be satisfied.

2.1.4 Reuse

A design methodology supports a designer in satisfying user requirements, prescribing *design goals* to be accomplished. An important high level design goal is increasing the efficiency of the design process, which contributes to improving the cost-effectiveness of the design process.

The efficiency of the design process can be increased by reusing designs and design knowledge. A design approach based on reuse of existing designs and design knowledge is called *design with reuse*. A design approach that aims at creating reusable designs and capturing reusable design knowledge is called *design for reuse* [106].

While design by reuse increases the efficiency of the design process, creating reusable designs and capturing reusable design knowledge may actually incur additional costs for the design process. Therefore, design for reuse must be justified by potential opportunities for reuse. These opportunities can be considered in the long term, whenever the development of several with similar characteristics is foreseen. This motivates approaches such as software product lines [27], in which a family of products with similar functionality is designed and maintained, and domain analysis [11], which identifies and captures reusable artefacts for particular classes of applications.

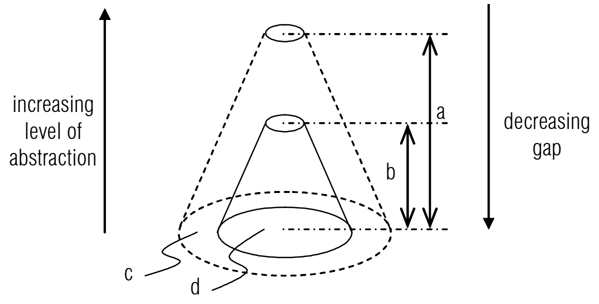
2.1.5 Design reuse

The reusability of designs is determined by a number of properties of the design, such as its generality, open-endedness and level of abstraction.

Generality defines that aspects covered by a design should be defined in their most general form [99]. Generality may come from generalizing user requirements and design goals, or anticipating future user requirements and design goals. *Open-endedness* is the property of a design of allowing future extensions [99].

The higher the level of abstraction of a design, the higher its reusability. Nevertheless, the higher the level of abstraction of a design, the wider the gap between the design and its realizations. The trade-off between the level of abstraction of a design and the gap between the design and its realizations can be visualized as a spotlight (*Figure 2-5*, inspired by [106]): higher levels of abstraction illuminate a wider target area; the penalty, however, is widening the gap between the design and its realizations.

Figure 2-5 Abstraction spotlight



a, b – gap between design and its realizations
c, d – potential target realizations

Examples of categories of reusable (implementation-oriented) designs are frameworks [16], software components [107], protocol stacks [48], and middleware [18].

2.1.6 Design knowledge reuse

A design step can be regarded as a problem solving activity (as suggested in [108]). In this view, the *source design* and user requirements define a problem that is solved by adequate solutions in a *target design*.

Since a design step typically requires a selection from a virtually infinite number of design solutions and alternatives, the selection of design solutions and the exploration of design alternatives are guided by design goals and *design (or solution) knowledge* provided by a design methodology. Design knowledge includes experience, techniques, design patterns, heuristics, technical engineering constraints, etc. The application of design goals and design knowledge in a design methodology is depicted in *Figure 2-6*.

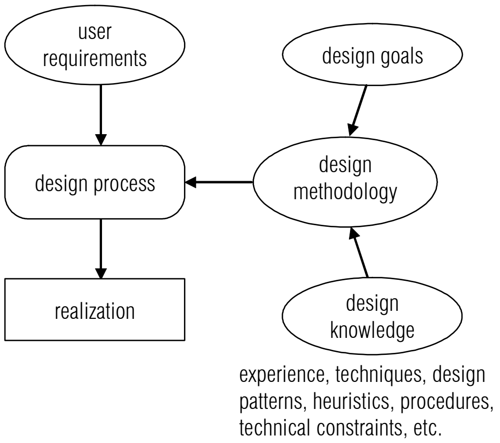

user requirements
design goals
design process
design methodology
realization
design knowledge
experience, techniques, design patterns, heuristics, procedures, technical constraints, etc.

Figure 2-6 Application of a design methodology with design goals and design knowledge

Design knowledge is used at each design step and may be specific to a particular abstraction level. This is captured in *Figure 2-2* in the relations between the design methodology and the design activities at different levels of abstraction.

Design knowledge can be captured with different degrees of rigour, ranging from experience to algorithmic procedures based on mathematical models.

Design experience is an example of design knowledge that is individual and lacks any (rigorous) description. Design experience is the practical knowledge, skill, or practice, which is derived from direct observation of or participation in design activities.

Design experience can be documented in *design techniques* that can be learned and transmitted to other designers.

Inspired from architecture design [2], *design patterns* [43] capture some design knowledge explicitly. Design patterns have been introduced as a way to reuse knowledge in the solution of recurring design problems. A design pattern describes the core of a design solution, and often includes a general description of the design problem addressed. A design pattern also describes the results and trade-offs of applying the pattern, which are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern [43].

Often, design techniques and patterns include *heuristics*, which are procedures whose effectiveness has not been formally proven but are generally accepted based on experience or common sense [16]. Heuristics may be used to explore a design space, as is suggested in [108].

On the extreme end of rigour there are algorithmic procedures based on mathematical models of the design space, which employ optimization techniques to explore a well-defined space of possible designs.

An important kind of design knowledge is *bottom-up knowledge* [40]. Bottom-up knowledge entails information about availability, quality and cost of resources that can be used to construct realizations, e.g., reusable designs, operating systems, programming languages, etc.

2.1.7 Models

We have defined designs as abstractions. Designs are, therefore, conceptual entities that only exist in the mind of a designer or a community of designers. Designs are constructed in terms of *design concepts*, which are abstract constructs of certain aspects of the objects in a given (design) domain. A set of design concepts and their combination rules is a *design conceptualization*.

For designs to be documented, communicated and analysed, they must be captured, i.e. represented in terms of some (symbolic) artefact. This implies that a language is necessary for representing designs in a concise, complete and unambiguous way. We call the representation of a design a *model*, and the language used in the creation of a model a *modelling language*. The relation between design conceptualizations, designs and modelling languages and models is depicted in *Figure 2-7* (adapted from [22, 40]). In *Figure 2-7*, we have represented symbolic artefacts with the icon used to denote UML packages [83]. This convention is also adopted in the remainder of this thesis.

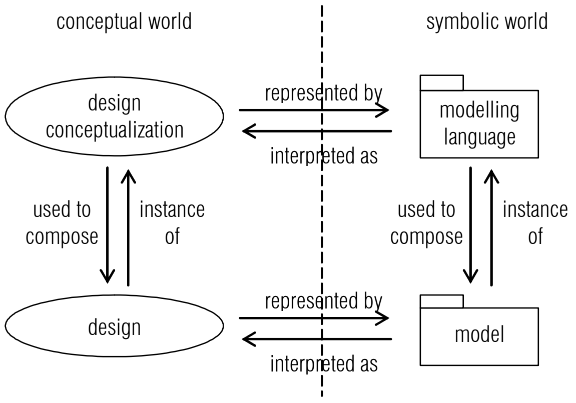


Figure 2-7 Relation between design conceptualization, design, modelling language and model

A design conceptualization should entail design concepts that allow the expression of relevant characteristics of a design. Since different characteristics are relevant at different levels of abstractions, different design conceptualizations may be necessary for designs at different levels of abstraction. Correspondingly, different modelling languages may be necessary for models representing designs at different levels of abstraction.

2.1.8 Model transformation

When designs are captured in models, transformation activities can be regarded as *model transformations*. Therefore, *model transformation specifications* can be used to constrain or (partially) determine the output of transformation activities.

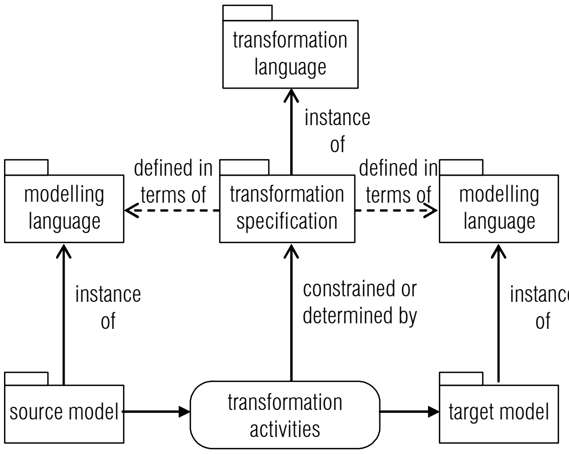
Model transformation specifications determine how the elements of different models relate to each other. When we use model transformation specifications to constrain transformation activities, a model transformation specification relates:

1. *source models* that represent a source design;
2. *target models* that represent a target design, and;
3. *additional information* that captures design decisions and requirements not satisfied in the source design and that are to be satisfied in the target design.

For our purposes, model transformation specifications are intended to capture generalized design knowledge used to perform transformation activities. Therefore, these specifications define the correspondences between types of model elements or particular combinations of types of model elements of source and target modelling languages. Model transformation specifications are defined in transformation languages, such as the ones compared in [28] and the one specified by the OMG in [79].

Figure 2-8 depicts schematically the relations between a transformation specification, source and target modelling languages, transformation activities and source and target models. For the sake of clarity, only one source model and one target model are shown in *Figure 2-8*.

Figure 2-8
Transformation specifications constrain transformation activities



Model transformation specifications may be used to constrain transformation activities with different degrees. For example, transformation specifications may contain enough information to determine how target models can be created given a source model or a number of source models. If such a model transformations specification can be executed, it can be used to automate transformations activities. Model transformation specifications can also be used to preserve a certain relation between source and target models automatically in face of modification of these models. This includes automating the modification of a source model to accommodate the modification of a target model and vice-versa (see "bidirectional transformations" and "execution scenarios" in [79]).

Transformation specifications can be used for purposes other than capturing generalized design decisions. For example, they can be used to automate assessment activities, by definining the acceptable relations between source and target models; or they can be used to support analysis of a design [61]. These uses of transformation specifications are not precluded by our approach, but are considered outside the scope of this thesis.

The model transformation pattern shown in *Figure 2-8* can be applied successively. In this case the notions of source and target models are relative, and an intermediary model is considered a target model from the perspective of the transformation from the source model, and the same intermediary model is considered a source model from the perspective of the transformation to the final target model.

2.2 Platform-independence

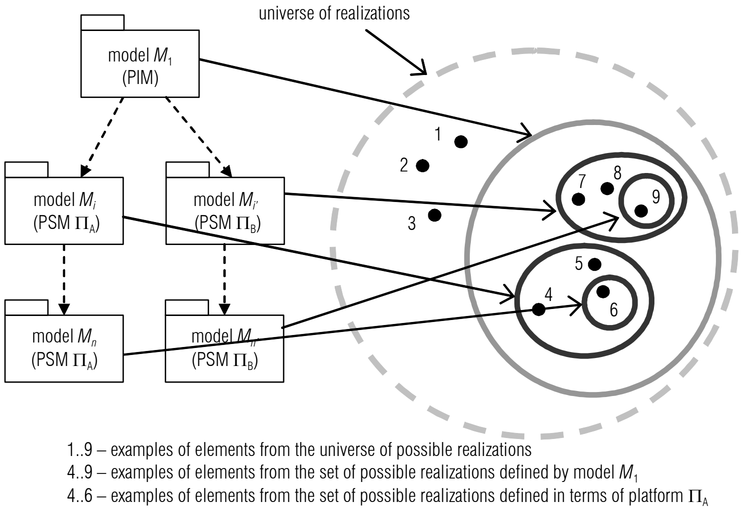
In most traditional development cultures, the ultimate product of the design process is the realization, deployed on available realization platforms. The various models at different levels of abstraction that are produced during the design process are mainly regarded as a means to obtain a realization of the system, and are not considered as final products of the design process.

In our approach, however, intermediate models are reusable and, are therefore, also considered final products of the design process. These models are carefully defined so as to abstract from details in platform technologies, and are therefore called *platform-independent models* (PIMs) (in line with the MDA [76]). A platform-independent model can be used as input to transformation activities that lead to different alternative realizations that use different platform technologies. In addition, platform-independent models remain relatively stable in face of changes in platform technologies.

The design process from platform-independent models to platform-specific realizations may entail the use of intermediate platform-specific models (PSMs).

The reduction of the realization space imposed by PIMs and PSMs is illustrated in *Figure 2-9*. In this figure, a PIM of an application is transformed into models M_i and M_i', which are PSMs that depend on platforms Π_A and Π_B, respectively.

Figure 2-9 Realization space for a PIM and PSMs


universe of realizations
model M_1 (PIM)
model M_i (PSM Π_A)
model $M_{i'}$ (PSM Π_B)
model M_n (PSM Π_A)
model $M_{n'}$ (PSM Π_B)
1
2
3
4
5
6
7
8
9

1..9 – examples of elements from the universe of possible realizations
4..9 – examples of elements from the set of possible realizations defined by model M_1
4..6 – examples of elements from the set of possible realizations defined in terms of platform Π_A
7..9 – examples of elements from the set of possible realizations defined in terms of platform Π_B
6 – example of element from the set of possible realizations defined by model M_n
9 – example of element from the set of possible realizations defined by model $M_{n'}$

2.2.1 Platforms

Before we further refine the notion of platform-independence, we should define more precisely the notion of a platform. The MDA guide [76] defines a *platform* as "a set of subsystems/technologies that provide a coherent set of functionality through interfaces and specified usage patterns that any subsystem that depends on the platform can use without concern for the details of how the functionality provided by the platform is implemented."

This definition is rather general, and lends itself to further refinement into different types of platforms that entail different types of subsystems and technologies defined with different purposes.

For example, in a certain context, an operating system may be considered a platform, including a kernel and a set of operating system libraries. In this case, the functionality provided includes memory allocation and protection, process concurrency, system file access, low-level input-and-output access, etc. This functionality is accessed through interfaces offered by the operating system in interactions called interrupts or kernel traps.

In another context, the term platform may refer to a programming language, its standard libraries and a compiler or an interpreter for the language. In this case, the functionality provided is program execution. Systems that rely on this platform are described using programming language constructs, which include interactions with the programming language's libraries.

The term platform is also used to denote (standardized) middleware technologies, such as CORBA/CCM [73], J2EE [102] (including EJB [103] and JMS [104]), DCE [109], and Web Services [120, 121], or particular implementations of these technologies, such as, e.g., .NET's implementation of Web Services [68], and IONA's Orbix implementation of CORBA [50]. In this work, we are particularly interested in this notion of platform.

2.2.2 Middleware platforms

Middleware is software that provides a supporting infrastructure for distributed applications. Middleware facilitates distributed application development by implementing reusable functionality that is commonly required, and by masking from applications some details and differences in the support offered by network technologies, programming languages, operating systems and hardware architectures. A middleware platform is positioned between parts of a distributed application and a distributed resources platform [18], as depicted in *Figure 2-10*.

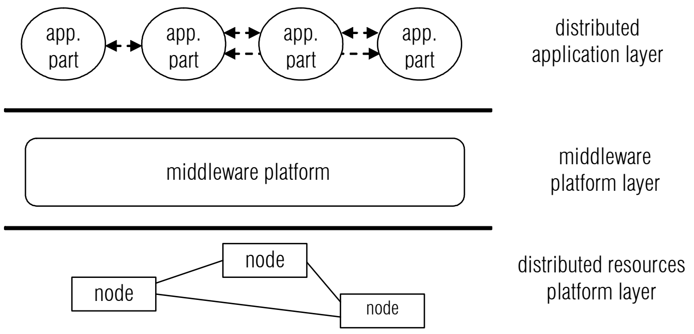

app. part
app. part
app. part
app. part
distributed application layer
middleware platform
middleware platform layer
node
node
node
distributed resources platform layer

Figure 2-10 Middleware platform is positioned between parts of a distributed application and a distributed resources platform

Middleware provides support for the distribution of application parts, offering programming abstractions that are closer to application requirements than the low-level programming abstractions that would have to be manipulated without middleware. For example, middleware may relieve the application designer from explicitly addressing some common tasks distributed applications perform, such as the handling of the reliability of communication, the correlation of requests and responses, the registration, location and activation of application parts, the encoding and decoding of messages, the use of a transport protocol, and the replication of application parts.

By hiding part of the complexity of the distribution support, middleware is said to offer *distribution transparencies* [59], or, in short, *transparencies*. An example of a specific transparency provided by a middleware platform is *replication transparency* [59], in which the complexity of maintaining the consistency between replicated application parts is hidden from the application developer. In recent years, the evolution of middleware has led to an increase in the level of transparency and in the number of generic services provided by middleware platforms.

Since a significant amount of development effort is spent on overcoming problems related to distribution (e.g., remoteness, partial failures, heterogeneity) and in exploiting distribution beneficially (e.g., to achieve performance and dependability), the reuse of middleware platforms significantly increases the efficiency of the development of distributed applications. In addition, the use of a middleware platform contributes to a reduction of long-term maintenance costs by improving the portability of application parts and facilitating the interoperability between (legacy) application parts. This is particularly important when existing applications (or application parts) are to be integrated.

Currently, there are several (competing) middleware standards, middleware implementations from different vendors, proprietary platform extensions and proprietary middleware implementations. In addition, different parts of a distributed application may be built using different middleware platforms, as depicted schematically in *Figure 2-11*.

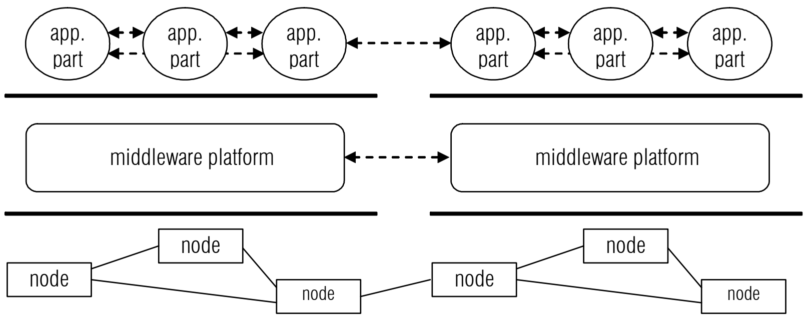


Figure 2-11 Different middleware platforms used to support distributed applications

During the lifetime of applications, these platforms may become obsolete, requiring upgrade or replacement. Therefore, the use of a single immutable distribution infrastructure does not provide an appropriate long term solution for the support of distributed applications.

Different middleware platforms provide different constructs from which applications can be built. For example, a number of popular middleware platforms offer location-transparent operation invocation, following a request-response interaction pattern. Examples of these platforms are CORBA/CCM, Web Services, DCE, Java RMI, and .NET remoting. Other popular middleware platforms offer support for interaction patterns other than operation invocation. Examples of such platforms are event-based and message-oriented middleware platforms such as JMS and MQSeries, respectively. Often, middleware platforms provide languages in which applications must be described so that appropriate support can be provided.

Different middleware platforms also offer different services and distribution transparencies. Examples of services that are typically offered by middleware platforms are directory services, trading services and security services. Examples of distribution transparencies often provided are location transparency, replacement transparency, replication transparency and migration transparency.

In addition to differences in the supported interaction patterns, services and transparencies, different middleware platforms also exhibit different quality characteristics, such as, e.g., time performance, scalability, reliability, availability and security. These characteristics should be considered in order to satisfy application quality requirements.

Since middleware platforms provide generic support for distribution, their usage patterns may include several alternatives from which a developer can choose. For example, CORBA offers both request-response invocations and an Event Service [74] that supports interaction based on event queues. If a designer determines that the Event Service must be used for all the interactions between application parts, we consider that this additional restriction in CORBA's usage patterns actually defines a new platform. All applications in this platform are also CORBA applications.

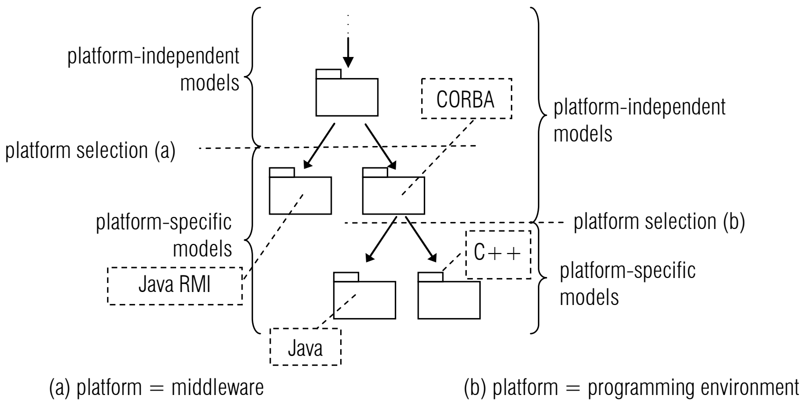
2.2.3 Relative notion of platform-independence

Since the concept of platform may refer to many different technical systems, the notion of platform-independence is relative to the particular definition of platform.

For example, suppose that our platform comprehends alternative middleware technologies, such as, e.g., CORBA or Java RMI. In this case, a CORBA IDL [73] specification is a platform-specific model, since it relies on a specific instance of middleware technology. In this context, a platform-independent model should be constructed by using only generic concepts that allow one to define the application (components and their behaviour) without being bothered by the idiosyncrasies of the middleware platforms that could be used for deploying the application. *Figure 2-12*(a) depicts the consequences of this definition of platform for the distinction between PIMs and PSMs.

In contrast, suppose that our platform comprehends the C++ programming language and a C++ CORBA ORB implementation (such as, e.g., Orbacus [51]). In this case, a CORBA IDL specification is a platform-independent model with respect to our platform, since this specification abstracts from the different programming languages and ORB implementations. For example, the same IDL specification could be realized in Java. *Figure 2-12*(b) depicts the consequences of this definition of platform for the distinction between PIMs and PSMs.

Figure 2-12 Abstraction levels of PIMs and PSMs


platform-independent models
platform selection (a)
platform-specific models
Java RMI
CORBA
platform-independent models
platform selection (b)
C++
platform-specific models
Java
(a) platform = middleware
(b) platform = programming environment

These two examples illustrate the importance of agreeing upon the abstraction criteria for PIMs and PSMs and agreeing upon what the platform is. Since we are particularly interested in the development of distributed applications that are supported by middleware platforms, we defined platform-independence with respect to middleware technologies and their implementations.

2.2.4 Levels of platform-independence

When pursuing platform-independence, one could strive for PIMs that are absolutely neutral with respect to all different classes of middleware platforms. This is possible at levels of abstraction in which the characteristics of a supporting infrastructure are irrelevant. For example, conceptual domain models [11] and RM-ODP Enterprise Viewpoint specifications [57] do not commit to characteristics of a middleware platform.

However, when the application is described as a decomposition of interacting application parts, one may use different sets of design concepts, combinations of concepts or patterns, each of which is better suited for specific classes of target middleware platforms. For example, a designer may choose to describe the interaction between application parts using event queues, favouring a realization on a platform that provides such an interaction pattern.

A consequence of this observation is that models of an application can be defined with different degrees of platform-independence, with respect to the extent to which these models constrain the designer in selecting a target platform. The various models of an application with different degrees of platform-independence may be organized into different *levels of platform-independence*. A model at a particular level of platform-independence can be realized onto a number of platforms. A model defined at a lower level of platform-independence further constrains platform selection when compared to a model at a higher level of platform-independence. We define platform-specific models (again, with respect to a particular definition of platform), as models that constrain platform selection so that only a single platform is acceptable for realization.

The reduction of the realization space imposed by models at different levels of platform-independence is illustrated in *Figure 2-13*. Model M_1 describes the application without constraining its internal structure. This model is used as input to produce model M_i, which excludes all realizations on top of platform Π_A, and, is therefore, defined at a lower level of platform-independence with respect to M_1.

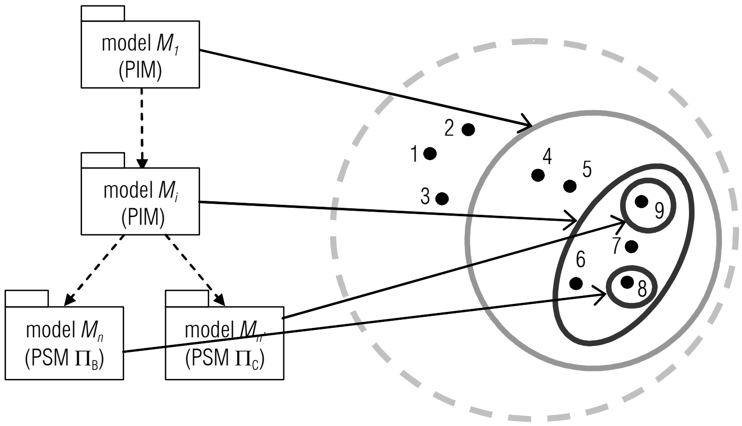


Figure 2-13 Realization space for models at different levels of platform-independence

1..9 – elements from the universe of realizations
4..9 – elements from the realizations defined by model M_1, at high level of platform-independence
6..9 – elements from the realizations defined by model M_i, at lower level of platform-independence
4, 5 – realizations defined by model M_1, platform Π_A
8 – realization defined by model M_n, platform Π_B
9 – realization defined by model $M_{n'}$, platform Π_C

Figure 2-14 illustrates a design trajectory that considers a number of particular target platforms, namely, CORBA, Java RMI, MQSeries and JMS. A model of the application at a high-level of platform-independence is depicted as the starting point of the trajectory. This model is used as input to produce two alternative models of the application: a model based on interaction through object invocation (M_{OI}), which facilitates the transformation to CORBA and Java RMI; and a model based on interaction through event queues (M_{EQ}), which facilitates the transformation to JMS and MQSeries.

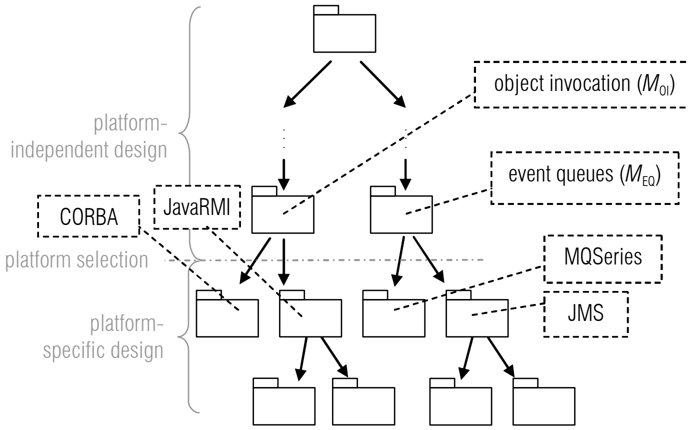


Figure 2-14 A design trajectory with models at different levels of platform-independence

In *Figure 2-14*, M_{OI} must not rely on specific characteristics and assumptions of either CORBA or Java RMI. Likewise, M_{EQ} must not rely on specific characteristics and assumptions of either JMS or MQSeries. These platform-independent models should, instead, rely on generic infrastructure characteristics that can be accommodated when transformation activities are executed and platform-specific models are created.

The implicit assumption of infrastructure characteristics in models may result in models that cannot be reused for different platforms. Furthermore, it may lead to models of different applications that cannot be directly compared and integrated. Infrastructure characteristics assumed in platform-independent models are better understood and controlled by designers if they are explicitly represented. In our design approach, these characteristics are embodied in what we call an abstract platform.

2.3 Abstract platforms

2.3.1 Definition

The concept of *abstract platform*[1] is an important architectural concept of our approach. An abstract platform is an abstraction of infrastructure characteristics which are assumed in the construction of platform-independent models of an application at some point of the design process.

An abstract platform defines an acceptable or, to some extent, ideal platform from an application developer's point of view. Alternatively, an abstract platform defines characteristics that must have proper mappings onto the set of target platforms that are considered for a design. In this way, the notion of an abstract platform allows a designer to explicitly define levels of platform-independence.

An abstract platform is determined by the platform characteristics that are relevant for applications at a certain platform-independent level. For example, if a platform-independent design contains application parts that interact through operation invocations, then operation invocation is a characteristic of the abstract platform. Capabilities of a realization platform are used during platform-specific realization to support this characteristic of the abstract platform. For example, if CORBA is selected as target platform, this characteristic might be mapped directly onto CORBA operation invocations. If JMS is selected as target platform this characteristic may be mapped onto a pattern of JMS asynchronous message exchanges.

Some abstract platform characteristics may depend on application requirements. For example, if a video-on-demand application requires the manipulation of streams of audio and video, this need should be reflected in models of the application at some point in the design process.

[1] proposed initially in [9] and elaborated in [6].

An abstract platform should be clearly defined, for at least two reasons: (1) application designers need to know the characteristics of the abstract platform when defining platform-independent models of an application; and (2) abstract platforms are a starting point for platform-specific realization.

Identifying and defining an abstract platform forces a designer to address two conflicting goals: (i) to achieve platform-independence (by preserving freedom of implementation), and (ii) to reduce the size of the design space explored for platform-specific realization.

2.3.2 Abstract platforms in the design process

In order to illustrate the use of the abstract platform concept along a design trajectory, let us consider the design of a conferencing application that facilitates the interaction of users residing in different geographical locations. Initially, the application designer describes the conference application solely from its external perspective, revealing the interactions that occur between the application and its environment.

Figure 2-15 shows a snapshot of the conference application with three users fulfilling the role of conference participant and a user fulfilling the role of conference manager. Since characteristics of the internal structure of the conference entity are not revealed, this decomposition of the system is specified at a relatively high level of abstraction. The abstract platform at this level of abstraction supports the interaction between users and the conference entity. The interfaces are described in terms of the shared actions between users and the conference entity[2].

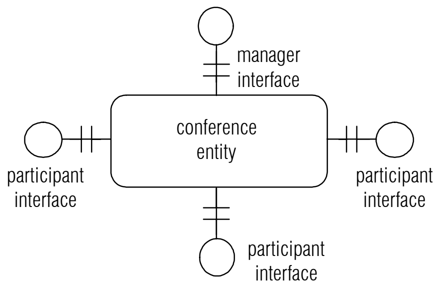


Figure 2-15 Snapshot of conference application (❶ in *Figure 2-18*)

The conference entity may be further decomposed into a centralized or distributed, symmetric or asymmetric design, and different abstract platforms may be used to support the interactions of the entities that imple-

[2] We provide a set of design concepts and design operations for defining and transforming designs in chapter 5. In this chapter, we assume an intuitive understanding of the notions of entities, shared actions (or interactions) and decomposition of entities.

ment it. Any number of recursive decompositions of this entity may be applied as necessary. This example shows that a high level of freedom of implementation is preserved at this level of platform-independence.

One possible way to proceed with design is shown in *Figure 2-16*. In this design, the internal structure of the conference entity is revealed. The conference entity is refined into a multicast entity and parts that are interconnected through the multicast entity. The abstract platform at this level of abstraction supports multicast interconnection as prescribed in the definition of the external behaviour of the multicast entity.

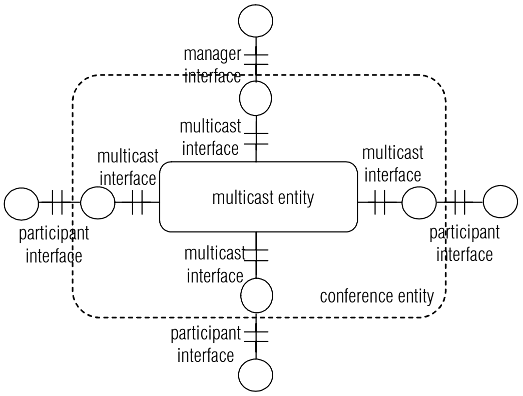


Figure 2-16 Revealing binding decomposition (❷in *Figure 2-18*)

At this point in the design trajectory, it is possible to realize this design on top of a target platform that offers a multicast scheme corresponding to that provided by the abstract platform (❸ in *Figure 2-18*). The implementation structures required to provide an adequate level of support are provided by the target platform. An alternative realization could implement the multicast entity as a centralized object (in a distributed object middleware), realizing the interactions between the objects and the multicast entity as distributed interactions (❹ in *Figure 2-18*). However, this alternative mapping may prove to be inadequate with respect to its quality-of-service characteristics, e.g., since a centralized implementation may fail to satisfy performance and scalability requirements. This mapping flexibility is possible because the refinement of the conference entity does not commit to a particular distribution in terms of nodes.

When the target platform does not provide the required level of support, the design can be further detailed in an abstract platform at a lower level of platform-independence. The refinement depicted in *Figure 2-17* assumes an abstract platform that only supports operation invocations. This realization differs from the previous design steps in that it does not consist solely of decompositions of entities; the internal structure of the conference entity depicted in *Figure 2-16* has been replaced by the structure depicted in *Figure 2-17*.

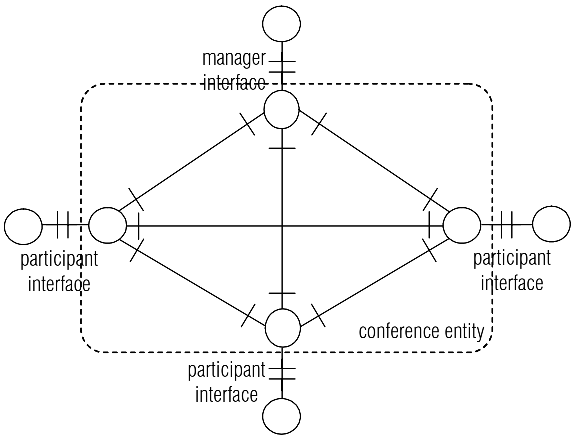


Figure 2-17 Revealing binding decomposition (❺ in *Figure 2-18*)

Figure 2-18 summarizes the application development trajectory that results from the application of the abstract platform concept to the conference application example. A few different middleware platforms are depicted as target platforms.

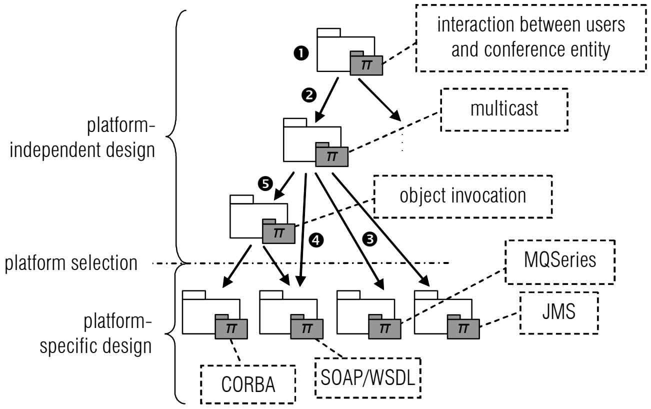


Figure 2-18 Models at related levels of platform-independence

2.3.3 Abstract platforms and modelling languages

Abstract platform characteristics and the characteristics of modelling languages adopted for a design are interrelated. For example, let us suppose a designer chooses to use SDL [54, 55] to represent platform-independent designs. This language provides the "agent" structuring construct: an "agent" is an entity can that exhibit reactive behaviour and communicates with other "agents" by exchanging "signals" asynchronously. If a designer models application parts as "agents" that interact with other application parts through "signals", this use of SDL implicitly defines an abstract platform that supports reliable asynchronous message exchange.

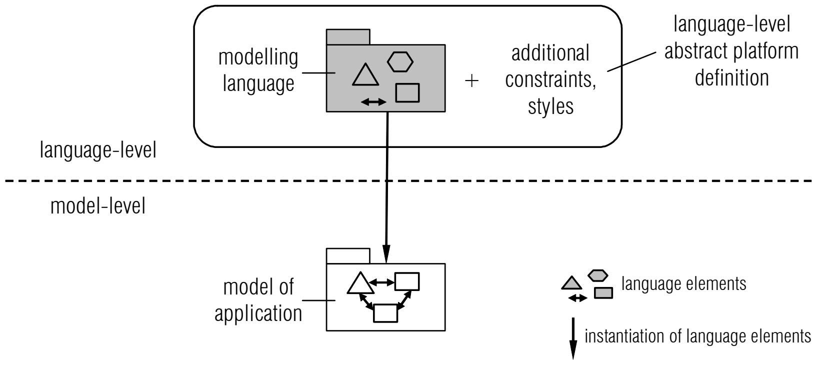
One might be tempted to believe that all relevant characteristics of a design's abstract platform can be derived from the concepts underlying the modelling language adopted for the design. However, abstract platform characteristics may depend on restrictions on the use of particular constructs in a modelling language or the use of certain modelling styles or patterns. In the example above, several modelling choices have been made by the designer with respect to which constructs to use for modelling application parts and their interaction. An alternative and equally valid usage of SDL might define that all application parts modelled using "agents" must interact by broadcasting "signals" to all other application parts.

Furthermore, in the general case, it is not possible to derive the set of modelling constructs that can actually be used by observing specific models of applications. Specific models only reveal the set of constructs that are used in particular combinations. For example, in the SDL examples above, it is impossible to know what "types" of "signals" may be exchanged between application parts. A model of a specific application will reveal the "types" actually used in the application. Without further definition of the abstract platform, one may have to assume that all "types" allowed by the language can be used (without restriction), and, hence, that all "types" are supported by the abstract platform.

We conclude that even using the same modelling language, with the same set of underlying concepts, a designer might implicitly define different abstract platforms. Therefore, it is necessary that the designer clearly document the styles and restrictions applied on the language, so that the intended abstract platform can be determined explicitly and unambiguously. We call this approach *language-level abstract platform definition*[3].

This approach is illustrated schematically in *Figure 2-19*, where concepts are represented as geometric forms.

Figure 2-19 Language-level abstract platform definition

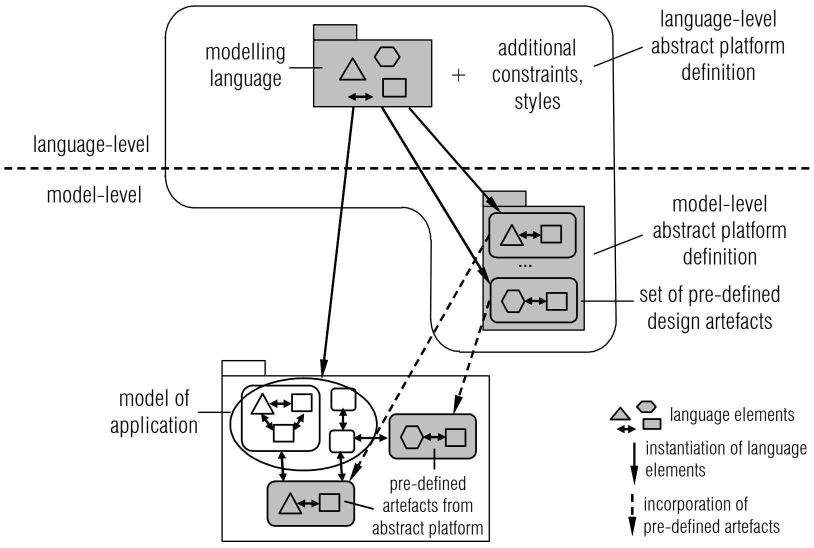

modelling language
+
additional constraints, styles
language-level abstract platform definition
language-level
model-level
model of application
language elements
instantiation of language elements

[3] This corresponds to what we have called "implicit abstract platform definition" in some of our earlier works [5, 6, 9].

When the modelling language supports the definition and reuse of pre-defined design artefacts, it is also possible to define some characteristics of an abstract platform by defining design artefacts that are to be reused. We call this approach to establishing the relation between the abstract platform and the modelling language *model-level abstract platform definition*[4]. In this approach, an application designer builds the application by composing application parts with the pre-defined artefacts that comprise the abstract platform.

The model-level abstract platform definition approach is necessary when intended characteristics of the abstract platform cannot be supported by language-level abstract platform definition. For example, let us suppose a designer requires an abstract platform that supports group communication between application parts and that the adopted modelling language is UML 2.0. While this language does not support group communication directly as a primitive design concept, it is possible to specify the behaviour of the required group communication scheme as a generic reusable sub-system in UML. This sub-system can then be re-used in the design of the distributed application.

The combination of the language-level and model-level approaches is illustrated schematically in *Figure 2-20*.



modelling language
+
additional constraints, styles
language-level abstract platform definition
language-level
model-level
...
model-level abstract platform definition
set of pre-defined design artefacts
model of application
pre-defined artefacts from abstract platform
language elements
instantiation of language elements
incorporation of pre-defined artefacts

Figure 2-20 Language- and model-level approaches to abstract platform definition

In the model-level abstract platform definition approach, the modelling language is used to describe: (i) the application, (ii) any necessary pre-defined design artefacts, and (iii) the composition of application and pre-

[4] This corresponds to "explicit abstract platform definition" in [5, 6, 9].

defined artefacts. Therefore, similarly to the case of the language-level abstract platform definition approach, the set of design concepts is relevant to derive some abstract platform characteristics. For example, in the group communication example above, the characteristics of the interaction between application parts and the group communication sub-system are relevant in the abstract platform definition. For instance, the reliability and time performance of this interaction has consequences for the reliability and time performance of the group communication scheme supported by the abstract platform.

Since in both the language- and model-level abstract platform definition approaches there is some overlap between language characteristics and abstract platform characteristics, we formulate two important requirements for a modelling language to support platform-independent design:

- the language should allow the designer to properly express the intended abstract platform characteristics; and,
- the language should be well-defined so that it is possible to derive the characteristics of the abstract platform unambiguously.

2.4 Overview of the design process

2.4.1 Preparation phase

Defining the organization of models into various levels of platform-independence and the characteristics of the models at each level requires careful consideration of application domain requirements and of a number of design goals. We propose this activity should be addressed explicitly in the *preparation phase* of the design process.

In the preparation phase, designers identify (and, when necessary, define) the required levels of models, their abstract platforms and the modelling language(s) to be used. In addition, a designer may also identify or define transformations between related levels of models. The results of the preparation phase are used in the *execution phase*, which entails the creation of models of an application using specific modelling languages and abstract platforms.

The role of the designer performing the preparation phase is to capture design knowledge that is later reused in the execution phase. The preparation phase should be executed by designers that are knowledgeable in the application domain, in the platforms that are used and in modelling language definition. In the preparation phase, the generalization of application requirements on the distribution infrastructure drives the consolidation of reusable design knowledge for potential target platforms. The role of application domain requirements, application requirements and target platform characteristics with respect to the preparation and execution phases is depicted in *Figure 2-21*.

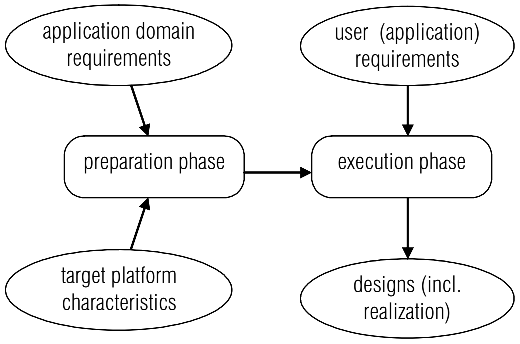

application domain requirements
user (application) requirements
preparation phase
execution phase
target platform characteristics
designs (incl. realization)

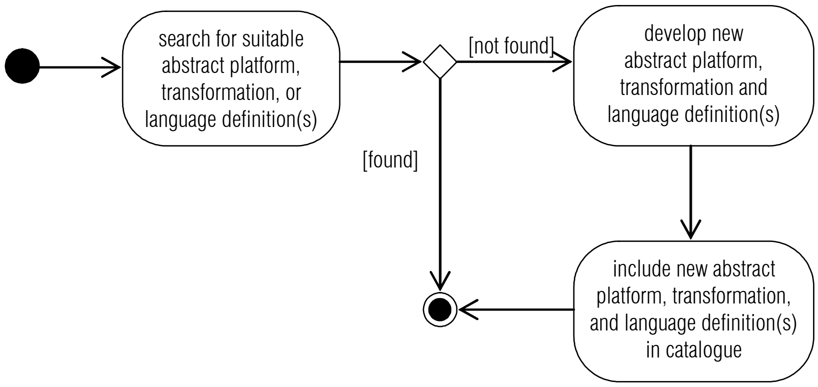
Figure 2-21 Role of application domain requirements, application requirements and target platform characteristics in preparation and execution phases

The following activities are performed in the preparation phase:

- *Platform definition*: in this activity, potential target realization platform(s) are identified and necessary abstract platforms are defined. This step is discussed further in chapters 3 and 4 of this thesis, which focus on the methodological aspects of abstract platform definition.
- *Modelling language definition*: models must be specified in a modelling language that is suitable for its application domain. Since models can be used for various different purposes, such as data representation, business process specification, user requirements capturing, etc., different modelling languages may be necessary in a development project. In this activity, the specific needs for modelling languages are identified, and suitable modelling languages are selected or defined. Modelling language definition is further discussed in chapter 6 of this thesis. We focus on modelling language support for the architectural design of distributed applications addressing modelling needs arising from abstract platform definition.
- *Transformation definition*: model transformation specifications capture generalized (implementation) solutions for models, consolidating design knowledge that is later reused in the execution phase. This activity identifies the possible or necessary transformation trajectories, including transformations from models that rely on the abstract platform to the models that rely on specific target platforms. Design operations for transformation definition are discussed in chapter 5.

In a long term development strategy, the preparation phase can be considered a long running phase, and abstract platform, modelling language and transformation definitions may be consolidated in a catalogue. Designers would consult the catalogue in the search of abstract platforms and transformations that are suitable for the design problem at hand. *Figure 2-22* summarizes this approach using a UML activity diagram.

Figure 2-22 Abstract platform and transformation definitions may be consolidated in a catalogue


search for suitable abstract platform, transformation, or language definition(s)
[not found]
develop new abstract platform, transformation and language definition(s)
[found]
include new abstract platform, transformation, and language definition(s) in catalogue

Identifying reusable abstract platforms and transformations that should be consolidated in a catalogue can be considered as an application of *design for reuse* at the preparation phase. Searching the catalogue can be considered as an application of *design with reuse* at the preparation phase. The costs and benefits of identifying and maintaining reusable designs are discussed in chapter 3 of this thesis.

2.4.2 Execution phase

The *execution phase* entails the creation of models of a specific application (or family of applications [27]) using specific modelling languages and abstract platforms and applying (manual and automated) transformations to models. The execution phase leads ultimately to a realization (or alternative realizations) of the application that satisfies user requirements, while capturing reusable platform-independent models of the application. This phase also entails analysis, testing and validation of models and realizations. The execution phase can be considered as a long-running phase, including activities for the maintenance and evolution of an application.

Modelling and applying transformations

When using an abstract platform with automated transformations to target platforms, the correspondence between some parts of the design and the realization may be established at a higher level of abstraction. In this case, a designer can simply use these parts of the design without performing any additional design steps for these parts. The resulting design process is depicted in *Figure 2-23*. The position of the top dashed line can be adjusted according to the abstract platform definition.

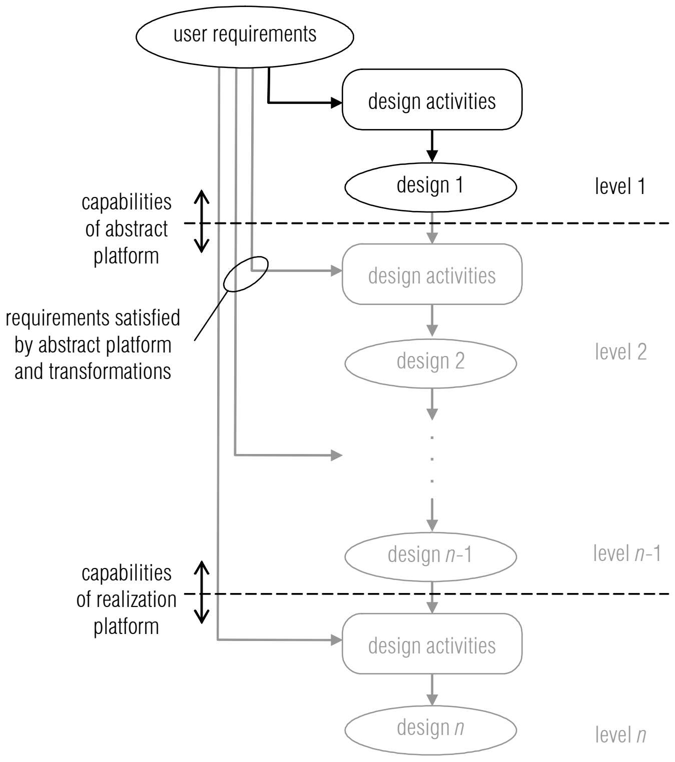

user requirements
design activities
design 1
level 1
capabilities of abstract platform
design activities
requirements satisfied by abstract platform and transformations
design 2
level 2
design n-1
level n-1
capabilities of realization platform
design activities
design n
level n

Figure 2-23
Enhancement of abstract platform

When correspondence between design and realization can be established at a higher level of abstraction, the abstract platform and the transformations embody design decisions taken at lower levels of abstraction. These design decisions must be consistent with user requirements yet to be satisfied (these requirements are circled in *Figure 2-23*). A designer may influence these design decisions by customizing transformations at the execution phase (e.g., through configuration of parameters), which requires that mechanisms for this customization be included in transformation specifications during the preparation phase.

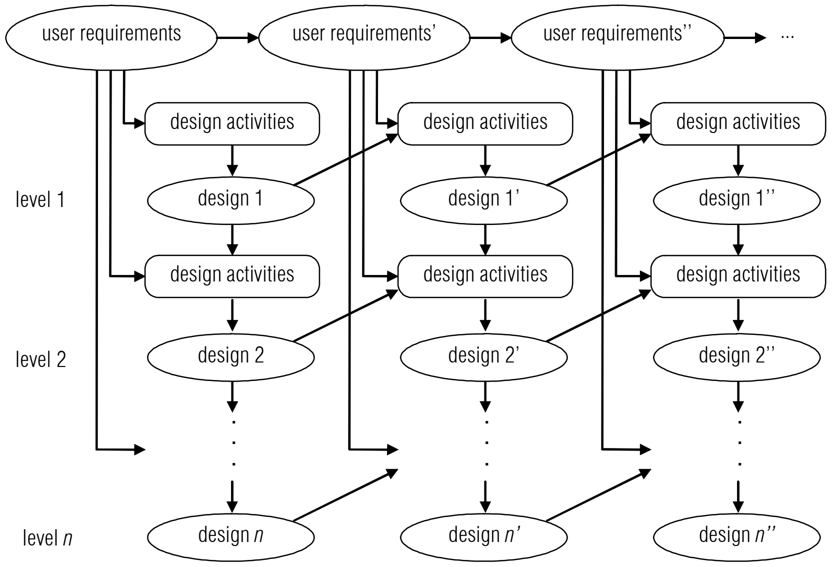
While transformation specifications and abstract platforms definitions may have been analysed, tested and validated during the preparation phase, analysis, testing and validation of transformation results may still be necessary. This is particularly required when properties to be considered for analysis are platform-dependent and only emerge when the realization is obtained and embedded in its environment.

Iterative design approach

While so far we have only shown the use of stepwise design in a top-down design approach, the use of designs in stepwise design does not constrain the designer in applying practical or more realistic design strategies, such as, e.g., iterative design. In this approach, the design process is performed in increments, iterations or cycles of manageable size. Examples of this approach are the spiral model [21] and the incremental model [69].

Figure 2-24 shows the iterative design approach. Since the understanding of user requirements changes during the design process, user requirements may change at each cycle. We have not depicted the influence of the design activities at each cycle and the user requirements for sake of legibility.

Figure 2-24 Iterative design approach



Iterative design approaches put additional requirements onto levels of designs that are visited repeatedly, namely, that they can be altered or extended to accommodate the requirements considered in the subsequent cycles [94]. The separation of platform-independent and platform-specific designs requires iteration to be considered carefully in our design approach. This is because platform-specific design decisions in a cycle should not influence platform-independent design decisions in a subsequent cycle. How to cope with this aspect of iterative design is discussed further in chapter 4 of this thesis.

Preparation and execution phases

Since the preparation phase defines generalized design knowledge that is reused in the execution phase, experience in the execution phase may imply that the preparation phase should be revisited. Therefore, the iterative approach may also be applied for the preparation and execution phases, as depicted in *Figure 2-25*.

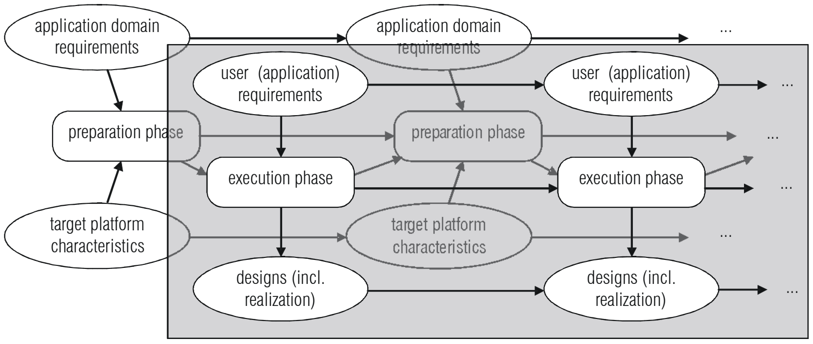

application domain requirements
application domain requirements
...
user (application) requirements
user (application) requirements
...
preparation phase
preparation phase
...
execution phase
execution phase
...
target platform characteristics
target platform characteristics
...
designs (incl. realization)
designs (incl. realization)
...

Figure 2-25 Iterative approach for preparation and execution phases

Conditions that justify revisiting the preparation phase include:

- when the support from abstract platforms does not satisfy specific application requirements. In this case, abstract platforms should be adjusted or extended to address these requirements;
- when the defined modelling languages lack required expressiveness, precision, or other desirable qualities. In this case, language definition should be adjusted;
- when new target platforms are introduced, requiring the development of new transformation specifications; and,
- when improved understanding of design steps performed manually creates opportunities for the automation of these steps in terms of transformation specifications.

The consequences of the use of the iterative design approach for the definition of abstract platforms and transformation specifications are discussed in chapter 4 of this thesis.

2.5 Related work on model-driven design methods

In this section we discuss two specific efforts on model-driven design methods which are closely related to our approach. Considerations about other related work can be found in sections 5.5, 5.7 and 6.7.

2.5.1 Stratified frameworks

Similarly to our approach, the authors of [13] propose a technique in which design concerns can be introduced at subsequent levels of models, which they call *strata*. An abstract platform can be considered a stratum, possibly with an associated framework. However, since we consider platform-independence explicitly in the design criteria for abstract platform definition (chapter 3), our notion of abstract platform is more specific than that of stratum.

This allows us to provide more guidance for the design process than [13]. In particular, by discussing the activity of abstract platform definition, we provide further guidelines on the elaboration of strata (in our preparation phase). In addition, we discuss the implications of the various relations between strata to the design process (chapter 4). We provide design concepts that could be used at the different (platform-independent) strata and the conformance relations that can exist between them (chapter 5).

2.5.2 Enterprise Fondue method

A number of UML profiles for model-driven development are defined in the context of the Enterprise Fondue method [96]. These profiles can be regarded as specific abstract platform models (as discussed in [97]).

The focus on UML makes their approach less suitable for designs at a high level of platform-independence, as argued in chapter 6 of this thesis. In particular, interaction between application parts cannot be described at a high level of abstraction. As a consequence of concentrating on models at a lower level of abstraction, their work addresses code generation with automated tool support.

As far as we are aware, this approach lacks a notion of conformance between models at the different levels. The UML profiles and model transformations proposed in the Enterprise Fondue method have limited support for the behavioural aspects of designs.

2.6 Concluding remarks

Platform-independence has strategic importance as a quality requirement for models of a distributed application. Considering platform-independence as an explicit quality requirement in a design process justifies the development of specific design methods that support the designer in obtaining models with the appropriate level of platform-independence while preserving the cost-effectiveness of the design process.

Separation of concerns in the design process leads to the construction of different models of an application. The different concepts, structures or patterns used and defined in models constrain the choice of technology platforms differently. Organizing models at different levels of platform-independence allows one to separate aspects of designs that remain stable in face of technology changes.

Our approach is based on capturing and reusing design knowledge. This is done in the preparation phase of the design process. This phase is driven by the generalization of application infrastructure requirements and platform support, thereby incorporating respectively top-down and bottom-up knowledge in the design process.

The notion of an abstract platform supports a designer in explicitly considering assumptions on infrastructure characteristics. In this chapter, we have illustrated the role of this concept by appealing to the intuition of the reader. In chapter 3, we define additional criteria that guide the definition of abstract platforms. We have argued that abstract platforms and modelling languages are somewhat interrelated, and, therefore, the definition of abstract platforms should not ignore modelling language characteristics. This is considered further in chapter 6 of this thesis.

Chapter 3

Methodological guidelines for the preparation phase

This chapter presents some methodological guidelines for the preparation phase of our approach. We discuss what qualities of platform-independent designs, abstract platforms and transformation specifications are desirable. We discuss how these qualities are related, forming a basis to enable trade-off analysis in the preparation phase. Because our approach aims at increasing the overall cost-effectiveness of the design process, we discuss the conditions under which the automation of transformation activities is beneficial, as well as the separation of models in different levels of platform-independence.

This chapter is organised as follows: section 3.1 presents the design quality criteria considered in our methodology, section 3.2 discusses how the automation of transformations between two levels of models can be justified, section 3.3 considers the costs and benefits of maintaining different levels of models; finally, section 3.4 presents some concluding remarks.

3.1 Design quality criteria

The quality of a design refers to the extent to which the design is appropriate for some intended purpose. An approach to achieving desirable qualities is to incorporate design quality criteria in the design process. These criteria should be used by designers when evaluating or engineering designs.

This section discusses the most relevant design quality criteria in our design process. We do not discuss design quality criteria in general, but focus on the criteria relevant to guide some activities in the preparation phase. In particular, we focus on how these criteria impact abstract platform definition and transformation specification.

3.1.1 Generality

Generality defines that a design should be defined in its most general form [99]. Generality supports the reusability of designs in different contexts. We are particularly interested in two aspects of generality:
(i) generality *with respect to a class of applications*, and;
(ii) generality *with respect to technology platforms*.

Generality with respect to a class of applications applies to (abstract) platforms and transformations. Platforms and transformations should be general-purpose, as opposed to specific for a particular application within an application domain. This allows reuse of platforms, abstract platform definitions and transformation specifications. Generality with respect to a class of applications is the basic distinction between preparation and execution activities[5]. The reuse of (abstract) platforms and transformations has an important role in improving the ratio between costs and benefits of the design process as discussed in section 3.3.

Generality with respect to technology platforms facilitates the realization of a design in a number of specific platforms. Therefore, it constitutes an important criterion for the composition of a platform-independent design and its abstract platform.

3.1.2 Stability

Stability of a design is the quality of a design of enduring without fundamental or significant change. Stability implies tolerance to some factors that are subject to variation or change in time. A means to cope with instability is to separate stable and unstable aspects of a design that are, to a large extent, independent of each other. Stability is a prerequisite for reusability of designs in time.

In our research, we are particularly interested in:
(i) the stability of a design *despite changes in the set of potential target platforms*, and;
(ii) the stability of abstract platforms *despite changes in application domain infrastructure requirements*.

The stability of a design despite changes in the set of potential target platforms is a pre-requisite for platform-independence. Ideally, a change in target platform should neither lead to a change in the abstract platform nor platform-independent designs that depend on the abstract platform. If possible, platform changes should be accommodated in transformation specifications.

[5] this distinction is explored in chapter 4

The stability of abstract platforms despite changes in application domain infrastructure requirements contributes to the reuse of (abstract) platforms and transformations.

Defining stable abstract platforms is challenging because it involves considering uncertain factors whose impact should be anticipated, both in the application domain and in the support from technology platforms. For example, business requirements may affect the set of potential target platforms, introducing a target platform that cannot be accommodated by transformations from the original abstract platform.

3.1.3 Buildability

Buildability of a design is inversely proportional to the amount of time, effort and resources required to build a conformant realization of the design *in a particular platform*. We say that a design is buildable if, and only if, a realization of acceptable overall quality can be obtained at acceptable costs. A necessary condition for acceptable buildability is that the class of conformant realizations of the design is non-empty.

Buildability and level of abstraction of a design are related. However, determining the relationship between the buildability and the level of abstraction of a design is not straightforward. On the one hand, lowering the level of abstraction of a design decreases the size of the design space to be explored in transformation activities, which tends to affect buildability positively. On the other hand, lowering the level of abstraction of a design may lead to design decisions that conflict with design decisions on the target platform, which also tends to affect buildability negatively. Therefore, not only the level of abstraction of the design should be considered in evaluating buildability, but also the similarity and differences in the concepts, patterns and structures used in the design and those used in conformant designs that are built on top of a target platform.

Buildability depends on the contents of a design. The actual contents of a platform-independent design depend on the abstract platform, which is defined in the preparation phase. Therefore, in the preparation phase, buildability can only be estimated indirectly, by analysing the impact of the use of an abstract platform in the buildability of the class of application designs supported by the abstract platform. We propose this is done by examining the differences and similarities in the abstract and target platforms.

Differences in the characteristics of an abstract platform and a target platform may result in the use of intricate combinations of constructs in conformant designs that rely on the target platform, which affects the complexity of transformation activities (and hence lowers buildability) and may affect the quality of the conformant realizations.

It is questionable whether transformations between disparate abstract and target platforms would provide platform-specific designs with appropriate quality properties, such as, e.g., traceability from platform-independent design, time performance, and maintainability.

3.1.4 Ease of use

Ease of use of a design [99] denotes the quality of a design to be used in a straightforward way. Since a design may have different types of users, ease of use concerns a particular type of user.

If we consider the ease of use of an abstract platform, these users are:

1. *abstract platform designers* who conceive and maintain the abstract platform;
2. *transformation designers* who define transformation specifications that relate platform-independent design and platform-specific design; these designers use abstract platforms as a starting point for transformation definition;
3. *application designers* who define platform-independent designs that use the abstract platforms.

Figure 3-1 illustrates the different types of users of an abstract platform.

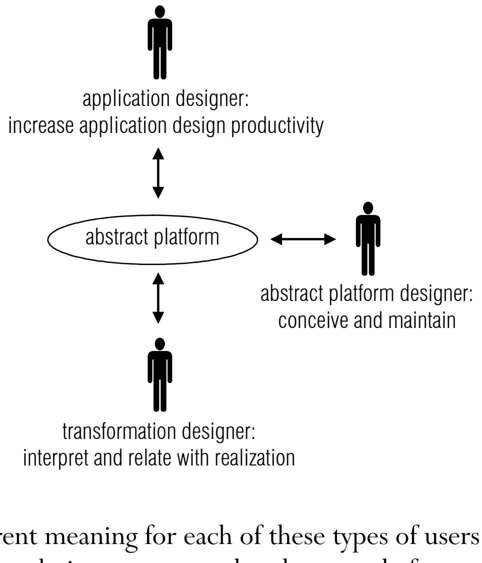

application designer:
increase application design productivity
abstract platform
abstract platform designer:
conceive and maintain
transformation designer:
interpret and relate with realization

Figure 3-1 Different types of users of an abstract platform

Ease of use has a different meaning for each of these types of users:

1. an abstract platform designer expects the abstract platform to require little or no maintenance. The ideal abstract platform for an abstract platform designer is stable with respect to changes in target platforms and general enough to cope with (unanticipated) application requirements;
2. a transformation designer expects the abstract platform to be defined in a precise and unambiguous way, without unnecessarily constraining the

freedom of implementation; for the transformation designer an ideal abstract platform facilitates buildability;

3. an application designer expects the abstract platform to provide facilities that improve productivity in application design. The ideal abstract platform for an application designer provides all infrastructural services required by the application. In this way, the application designer is able to focus on the problem at hand, i.e., the design of a specific application. For example, if an application requires transaction management, an ideal abstract platform should provide services for transaction management to match the requirements.

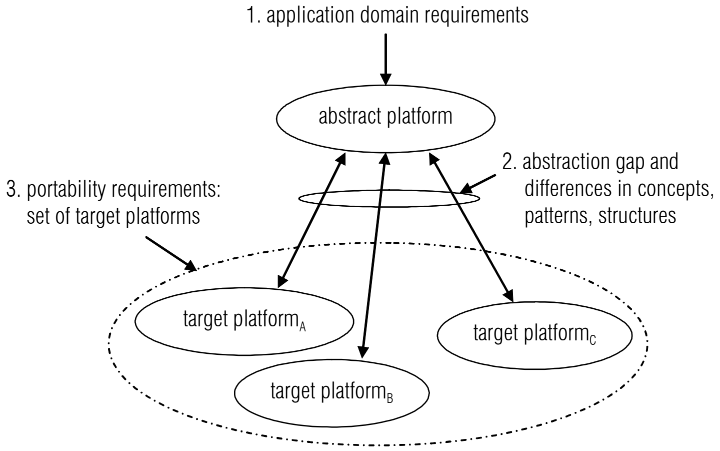
3.1.5 Balancing design quality criteria

Designers should strive to obtain overall design quality rather than focussing solely on a specific quality characteristic. If quality criteria are conflicting, the designers should balance compliance to the different quality criteria in order to obtain the most preferable design.

In the previous sections, we have discussed a number of design quality criteria that influence abstract platform design. These criteria are affected by the following factors (depicted schematically in *Figure 3-2*):

1. *application domain requirements;*
2. *the abstraction gap and the differences in concepts, patterns and structures in the abstract platform and a target platform; and,*
3. *portability requirements;*

Figure 3-2 Factors in the choice of abstract platform


1. application domain requirements
abstract platform
3. portability requirements: set of target platforms
2. abstraction gap and differences in concepts, patterns, structures
target platform$_A$
target platform$_B$
target platform$_C$

Application domain requirements (1) primarily affect *ease of use* of the abstract platform for the application designer. Ease of use for the application designer calls for both matching between application domain requirements and abstract platform characteristics, and alleviating the task of the designer by providing generic functionality in the abstract platform.

The abstraction gap and the differences in concepts, patterns and structures in the abstract platform and a target platform (2) primarily affect *buildability* of designs with respect to each of the target platforms. Considering these factors, an abstract platform should be established by analysing the set of potential target platforms and their common and diverging characteristics.

Factors (1) and (2) are often conflicting:

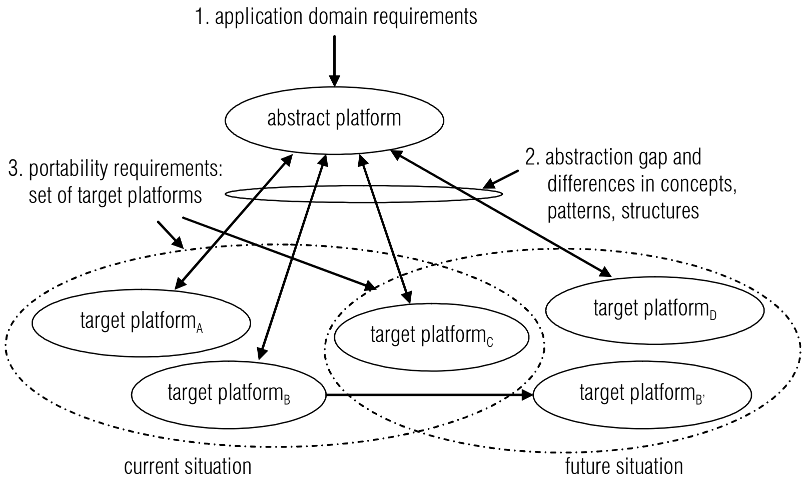
– Raising the provided support to match application domain requirements may increase the gap between the abstract platform and target platforms. This is the case, for example, for the support of multicast message exchange in the abstract platform, when a target platform supports only the request/response interaction pattern.
– Reducing the gap between support provided by the abstract platform and target platforms may lead to an abstract platform that handicaps the designer. This is the case, for example, for a "minimalist" abstract platform that supports a common denominator of a broad class of middleware platforms such as point-to-point one-way message exchange. Patterns such as request/response and multicast message exchange are expected to be included in the application design.

Portability requirements, buildability and platform-independence

Having introduced the notion of buildability, we are able to reformulate the definition of platform-independence of a design. We say that a design is *platform-independent* if, and only if, it is buildable on a number of target platforms. The set of target platforms is determined by *portability requirements* (factor 3 in *Figure 3-2*) for the design, which are themselves determined by technical, business and strategic arguments.

Increasing the buildability of designs with respect to a number of target platforms is a challenging activity. This is partly because increasing the buildability with respect to a particular platform may enlarge the gap between the abstract platform and other platforms, and hence lowers buildability with respect to these other platforms. Therefore, when defining an abstract platform, buildability should be evaluated with respect to each of the platforms implied by the portability requirements.

The set of target platforms may change in the course of time, e.g., due to business arguments. This is depicted in *Figure 3-3*. The modified set of platforms is a result of the inclusion of target platform$_D$, the exclusion of target platform$_A$ and a change in target platform$_B$.



1. application domain requirements
abstract platform
2. abstraction gap and differences in concepts, patterns, structures
3. portability requirements: set of target platforms
target platform$_A$
target platform$_B$
target platform$_C$
target platform$_D$
target platform$_{B'}$
current situation
future situation

Figure 3-3 Change in the set of target platforms

Since platform-independence requires preserving buildability even in future usage scenarios, is it difficult to evaluate platform-independence *a priori*. This evaluation requires defining the possible future usage scenarios, i.e., possible future target platforms, and estimating buildability for each of these scenarios. Actual use of an abstract platform reveals actual buildability in time, which may improve confidence in the level of platform-independence or lead to narrowing portability requirements if acceptable.

3.1.6 Concluding remarks

Defining an abstract platform explicitly brings attention to balancing between ease of use (from the perspective of application designers) and buildability, while observing generality and stability.

On the one hand, an abstract platform indicates directly the support available to designers during platform-independent modelling, and therefore, reflects the needs of application designers, including the needs to handle complexity in application design. On the other hand, an abstract platform is established by considering the set of potential target platforms and their (common and diverging) characteristics; this bottom-up knowledge is useful to reduce the design space to be explored for platform-specific realization, increasing the efficiency of the design process.

The factors we have discussed in the previous section vary in different projects, according to different application domains and specific application requirements, possibly resulting in different abstract platforms. A comprehensive model-driven design approach should, therefore, allow a designer to select or define suitable abstract platforms for their platform-independent designs.

3.2 Automated transformation

During the execution phase, an application developer derives models at a lower-level of platform independence from models at a higher-level of platform independence. In order to increase the efficiency of the design process, it may be possible to automate transformation activities required to bridge between different levels of models.

A requirement to the automation of transformation activities is the specification of transformation in the preparation phase. Full automation of transformation between two levels of models requires the transformation specifier to define rules to transform all possible source models into appropriate target models. The transformation specifier must fully understand the relation between source and target (abstract) platform definitions, and express these rules in a suitable transformation language, supported by a transformation tool. For these reasons, transformation specifications should be produced by a knowledgeable expert.

When transformation is automated, the creative design activities that would otherwise be executed manually by a designer are generalized and moved to the specification of the transformation itself and to the parameterization of transformations. This distribution of design activities is depicted in *Figure 3-4*. In this figure, a star denotes the corresponding design activities. *Figure 3-4(a)* shows the situation with manual transformation. *Figure 3-4(b)* shows the situation with automated transformation. The thick arrow labelled with *T* represents the execution of a transformation specification *T*, and a_T represents *transformation arguments*, i.e., transformation parameters values. Transformation arguments are also called *markings* when these are associated to elements in a source model, in which case parameters of the transformation are called *marks*. Combining markings and the source model in the same model results in a *marked model*.

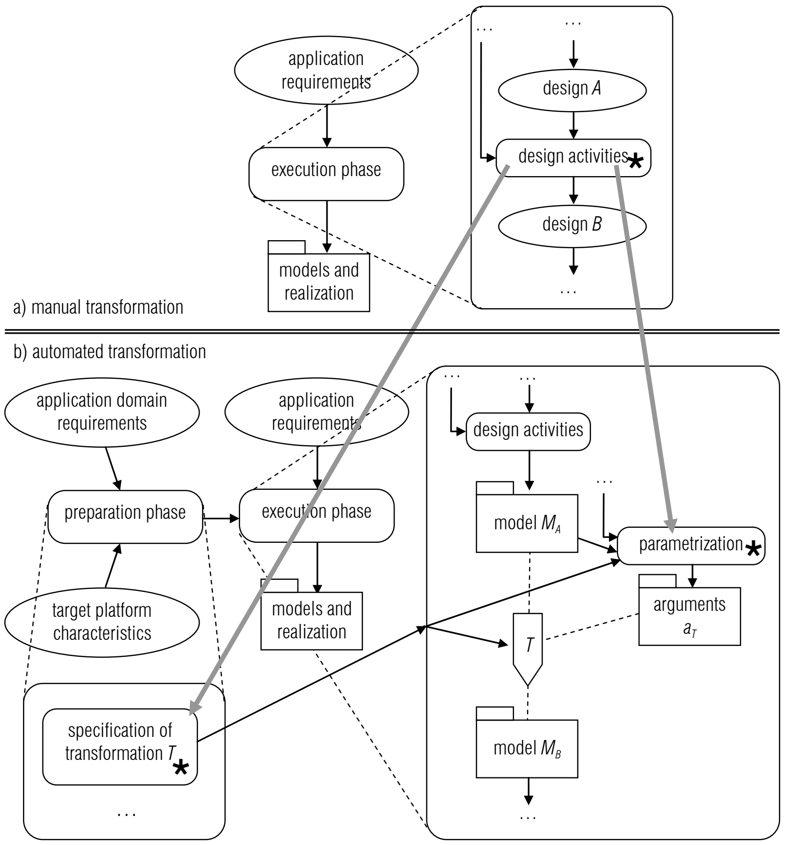


Figure 3-4 Design activities with manual and automated transformation

The costs of defining an automated transformation between two related levels of models *A* and *B* must be compensated by reusing the transformation specification. The following conditions contribute to the reuse of a transformation specification:

- *the number of applications built using models at level A and targeting B is high*, i.e., the (abstract) platform at level *B* is popular for targeting applications that can be expressed in terms of (abstract) platform at level *A*;
- *changes in application requirements are frequent*, but these changes do not affect the stability of the (abstract) platform at level *A*;
- *the development process is cyclic, and the number of design iterations is high*, i.e., the model of the application in *A* is modified several times during the development. In this case, manual manipulation of models would have required manual propagation of changes applied at level *A*.

The bottom-line is that the cost of building an automated transformation between levels *A* and *B* must be lower than the costs of manually deriving models at level *B* (from designs at level *A*) over (a long period of) time. Therefore, the stability of the (abstract) platforms at level *A* and *B* should be considered. The stability of the (abstract) platform at level *A* allows more applications to be developed in terms of this platform and the stability of (abstract) platform at level *B* is required to reuse the transformation, since transformation from *A* to *B* is specific to the platform at level *B*.

It is possible that models obtained manually and automatically differ significantly with respect to relevant qualities. These qualities should be considered when justifying automation. For example, automated code generation may result in implementations of lower time performance. This can be reflected in cost estimates by lowering the cost of manual coding to account for the benefits of obtaining implementations that perform better. In contrast, automated code generation may lead to improvements in the correctness of implementations. In this case, cost estimates should include the costs incurred by testing, both for testing the transformation and testing the code obtained manually.

3.3 Levels of models

We envision two different extreme approaches to organizing the development process with respect to platform-independence levels:

1. *an approach with one level of platform-independent models and one level of platform-specific models related through a (fully- or partially automated) transformation), and;*
2. *an approach with exhaustive use of intermediate levels of models and several (fully- or partially automated) transformations between these models.*

We argue that a combination of these extreme approaches is the most effective way to handle the problem. In the sequence, we examine the costs and benefits of introducing an intermediate level of models between two arbitrary levels, a source level and a target level. This allows us to consider the full range of combinations of the extreme approaches (1) and (2), since the recursive introduction of intermediate levels eventually leads to an exhaustive use of intermediate levels. In the discussion, we distinguish between fully or partially automated transformations.

3.3.1 Fully automated transformations

Figure 3-5 depicts the alternative situations which we contrast for fully automated transformations:

(a) a situation in which a transformation *T* produces models at level *B* from models at level *A*; and,

(b) a situation in which a transformation T_1 produces models at level *X* from models at level *A*, and a transformation T_2 produces models at level *B* from models at the intermediate level *X*.

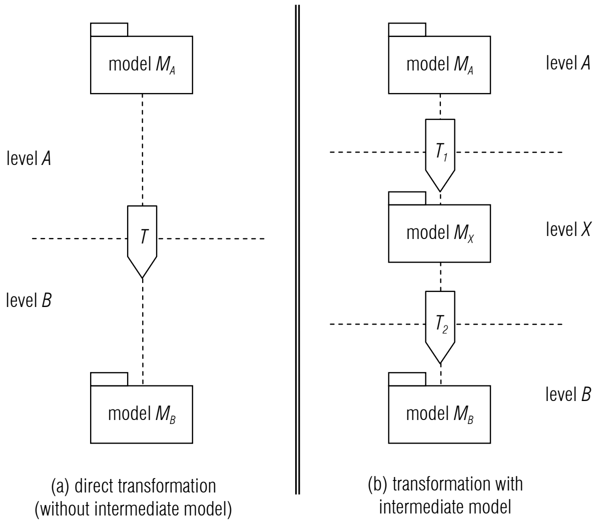


Figure 3-5 Direct transformation and transformation with intermediate model

Considering solely the effort spent in the preparation phase to specify the transformations in situations (a) and (b), we cannot formulate a general rule to decide whether an intermediate step should be introduced. In some cases, it may be easier to define two transformations using an intermediate model, and, in some other cases, direct transformations may be easier to define.

Nevertheless, it is possible to draw some general conclusions on the consequences of introducing intermediate levels of models for the reuse of transformations. In this respect, an intermediate level of models may be beneficial since:

1. it may be possible to *reuse the transformation from source models to intermediate models*, even if the original transformation from intermediate models to target models cannot be reused (e.g., because of platform change); and,
2. it may be possible to *reuse the transformation from intermediate models to target models* in new projects, since there may be transformations from different source levels to the intermediate level.

In both cases (1) and (2) above, situation (a) in *Figure 3-5* would imply no reuse for the transformation from level *A* to level *B*.

A transformation between levels *A* and *B* is specific to the (abstract) platform at level *B*. Therefore, the stability of the (abstract) platform at level *B* is required to reuse the transformation. Introducing an intermediate level of models may serve to factor out parts of the transformation that are less platform-specific, capturing unstable transformation *X* to *B* separately from stable transformation *A* to *X*. For example, consider that the level *A* consists of models in an application-domain-specific language [29], and that level *B* consists of middleware platforms, such as CORBA/CCM [73, 75] and Web Services [120, 121]. Instead of defining a transformation directly from *A* to *B*, one may consider the introduction of EDOC CCA models [82] as intermediate models at level *X*, capturing a transformation from the domain-specific language to a solution that is more stable than middleware platforms. Additional transformations that do not have to consider the specificities of the domain-specific language can be used to transform the EDOC CCA models to CORBA/CCM or Web Services PSMs. Clearly, this solution requires the stability of the intermediate level *X*, in the example, EDOC CCA models. This solution is depicted in *Figure 3-6*(a).

A transformation between levels *A* and *B* is also specific to the (abstract) platform of the source level *A*. Introducing an intermediate level of models may also lead to the reuse of the transformation from the intermediate model to the target model. For example, consider that the level *A* consists of models in different application-domain-specific languages, and level *B* consists of Web Services. Introducing an intermediate level *X*, e.g., populated with EDOC CCA models allows us to reuse the general-purpose EDOC to Web Services transformation. This transformation is not "contaminated" with application-domain-specific issues. Again, this solution requires the stability of the intermediate level *X*. This solution is depicted in *Figure 3-6*(b). Although we have presented both solutions separately, they could be combined, as depicted in *Figure 3-6* (c).

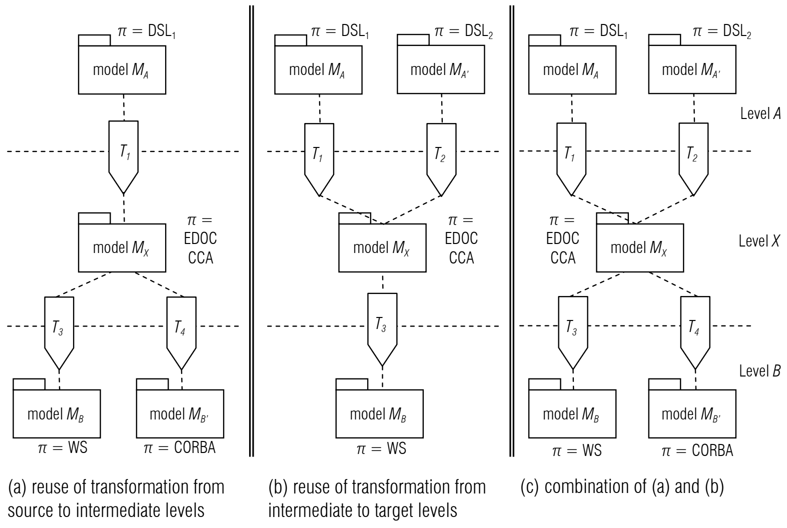

π = DSL1
model MA
T1
model MX
π = EDOC CCA
T3
T4
model MB
model MB'
π = WS
π = CORBA
π = DSL2
model MA'
T2
Level A
Level X
Level B
(a) reuse of transformation from source to intermediate levels
(b) reuse of transformation from intermediate to target levels
(c) combination of (a) and (b)

Figure 3-6 Reuse of transformations due to introduction of intermediate level of models

In order to justify the introduction of the intermediate levels of models *X*, the abstract platform of the level *X* must be suitable for a large number of applications that can be described at level *A* and realized on platforms at level *B*. In our example, the consequence of this observation is that the abstract platform at level *X* should be independent of application domains at level *A* and independent of technology platforms at level *B*. In addition, standardization of this abstract platform may be necessary to increase number of the opportunities for the reuse of transformations to and from the intermediate level. The EDOC CCA is an example of such an abstract platform, allowing the description of distributed application in terms of components and their interconnection in terms of messages exchanged through ports.

The same pattern of transformation reusability can be observed when considering the transformation of EDOC CCA models at level *X* to models at the level of programming languages such as Java. In this case, level *B* in *Figure 3-6* can be regarded as an intermediate level in the transformation, consisting of CORBA and Web Services-specific models. These models are transformed into Java interfaces, stubs and skeletons through standardized transformations [86, 92]. These transformations are executed through tools such as the one available in [93] and the ones listed in [87].

3.3.2 Partially automated transformations

It may be necessary to introduce an intermediate level of models between a source and a target level when no automated transformation can be defined directly, or when automated transformations produce results that do not satisfy non-functional requirements. By introducing an intermediate level of models, intermediate models can be elaborated upon, e.g., incremented, modified, combined with additional models and marked. The intermediate level can be regarded as a means to systematically lowering the degree of automation, and introducing opportunities to insert design decisions in the transformation from source to target models.

For example, let us consider again level *A* consisting of models in application-domain-specific languages, level *X* consisting of EDOC CCA models and level *B* consisting of CORBA/CCM and Web Services-specific models. This situation is depicted in *Figure 3-7*. In this example, marking EDOC CCA models manually is a means to specify properties that are not stated in source nor intermediate models and that may be required for the realization of the application on a target middleware platform. Examples of these properties are requirements on the replication of components to satisfy availability requirements, requirements on the potential location of components in the distributed environment to satisfy time performance requirements, requirements on the persistency mechanisms required, etc. These requirements refer to specific components in the EDOC CCA models and cannot be specified meaningfully at level *A* or derived directly from EDOC CCA models.

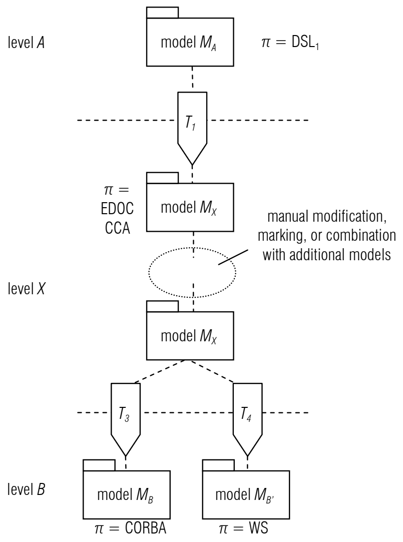


Figure 3-7 Intermediate models as means to introduce design decisions

Changes in models at a high-level of platform-independence may lead to changes in all intermediate models and their associated markings. In the case of partially automated transformations, intermediate models affected by changes may have been modified or marked manually. In this case, propagation of changes may lead to high costs, since manual modifications and markings may have to be adjusted. In contrast, in fully automated transformation chains, changes are automatically propagated through transformation. Since the state-of-the-art still requires significant developer intervention along transformation chains, the propagation of changes contributes to a large portion of the costs incurred by introducing separate levels of models. These costs should ideally be contained by appropriate traceability mechanisms in (MDA) tools.

With the introduction of an intermediate level of models, it may be necessary to design a specific abstract platform for that level. This incurs some additional effort for the preparation activities. For the case of partially automated transformation, application designers using the intermediate level of models must learn how to use the abstract platforms and transformations required at that level, which usually incurs training costs and increases the threshold for developers to apply this approach. In order to reduce these costs, ease of use of the abstract platform for the application designer should be prioritized in the preparation phase.

3.4 Concluding remarks

A conclusive study of the costs and benefits of introducing different levels of models requires empirical verification. Such a study should consider a multitude of application requirements, as well as the opportunities for reuse across different instances of model-driven development projects.

In the absence of such an empirical study, we have discussed, in general terms, the benefits and costs of introducing different levels of models and transformations. We believe this discussion forms a basis to enable trade-off analysis between the different factors in the preparation phase of our design approach. Evaluating these trade-offs at early stages of development remains nevertheless a challenging activity, since the benefits of the separation PIM/PSM must be considered on the long run, particularly due to the role of reuse of models and transformations.

Opportunities for reusing transformations play an important role in deciding the organization of the execution phase in terms of levels of models and transformations. A single transformation from high-level models to implementations may be costly to develop and is rendered useless in the face of technology platform changes. Given that technology platforms are generally regarded as unstable, it is important to attempt to recognize (intermediate) stable abstract platforms that can be used for a large number of applications. This makes transformations to and from this intermediate abstract platform more general and stable, and hence, reusable.

The proliferation of different (incompatible) intermediate levels of models reduces the opportunities for large-scale reuse of intermediate models and transformations to and from intermediate models. This calls for the agreement on a small number of abstract platforms that are, to a great extent, application-domain-neutral and platform-independent.

Chapter 4

Separation of concerns and the dependencies between models

In chapter 3, we have proposed design criteria that lead to the separation of stable and unstable aspects of designs, and the separation of generic and specific aspects of designs. This chapter discusses the implications of these dimensions of separation of concerns to our design approach.

The application of separation of concerns in our approach results in different aspects of a design being captured in different models. Ideally, models should be independent of each other, i.e., it would be possible to create models independently, and a modification in one model should not impact other models. Nevertheless, as we elaborate in this chapter, not all models are independent of each other. For this reason, we examine the relations between the different kinds of models. This provides further insight into what distinguishes these models. Moreover, we discuss the consequences of the separation of models for the design process.

This chapter is organised as follows: section 4.1 sets out the research questions addressed in this chapter; section 4.2 analyses the (inter)dependencies between the various types of models, which results in requirements and guidelines for the separation of models; section 4.3 discusses how the dependencies between models affect the design process; finally, section 4.4 presents some concluding remarks.

4.1 Separation of concerns

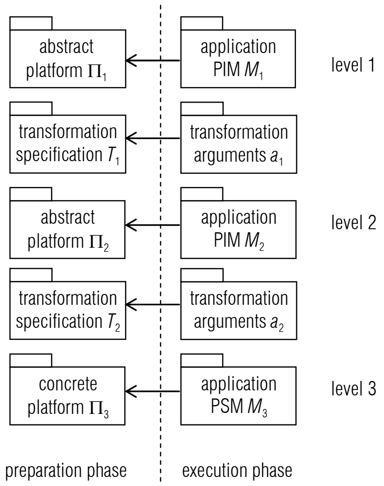
Our design approach explores two main dimensions of separation of concerns: the separation of platform-independent and platform-specific concerns; and the separation of preparation and execution concerns.

The separation of platform-independent and platform-specific concerns leads to the organization of the models of an application in different levels of platform-independence.

The separation of preparation and execution concerns is reflected in the organization of the design process in the phases of preparation and execution. In the execution phase, a specific application is developed using the generalized designs and design knowledge captured during the preparation phase. Separation of concerns in this dimension leads to the definition of (abstract) platforms and transformation specifications that are generally applicable for the class of applications considered.

Figure 4-1 shows the various models in our approach. Three levels of platform-independence are depicted, and the results are classified according to the phase in which they are produced. In this figure, an arrow indicates that a model is dependent on the existence of another model. Abstract platforms have also been depicted as models, indicating that abstract platforms definitions can be captured in *abstract platform models*.

Figure 4-1 Models in our design approach


abstract platform Π_1
application PIM M_1
level 1
transformation specification T_1
transformation arguments a_1
abstract platform Π_2
application PIM M_2
level 2
transformation specification T_2
transformation arguments a_2
concrete platform Π_3
application PSM M_3
level 3
preparation phase
execution phase

Once we propose the use of different types of models in the design process, we must determine what distinguishes the various types, defining the design concerns that are addressed in each of the different types of models.

In order to exploit the separation of models beneficially, we must also understand how the various models relate to each other. The benefits of separation of models are reduced when models are related in such a way that modifications in a model affect other models. Ideally, the impact of change should be limited to the model affected. We should, therefore, analyse the dependencies between models and strive to find techniques to avoid undesirable dependencies between models.

Dependencies between models also restrict the possibility for division of labour and concurrent design. Interdependencies reduce the efficiency of the design process and often have to be addressed in the design process by introducing iteration cycles [14]. As we elaborate on the following sections, some interdependencies can be avoided by following a number of rules with respect to the content of the various models and with respect to the modifications that may be applied to the various models.

In the remainder of this chapter, we address the following questions with respect to the separation of models in our approach:

- Which design decisions should be captured in PIMs, PSMs, abstract platforms, target platforms, transformation specifications, and transformation arguments?
- Can target platforms be modified without affecting PIMs and abstract platforms?
- Can transformation specifications be modified without affecting PIMs and abstract platforms?
- Does a modification in a PIM affect a corresponding PSM?
- Does a modification in a PSM affect a corresponding PIM?
- Are there interdependencies between the various models that require iterations in the design process? Can these be avoided?
- What are the criteria to group design decisions in a certain level of platform-independence?

4.2 Dependencies between models

4.2.1 Models as modules

In order to examine the relations between the various models, we consider models as *modules*. Typically, a module is a set of elements of a design that are grouped together according to an architecture or plan, with three main purposes [14, 15]:

- to make complexity manageable;
- to enable parallel work; and
- to accommodate future uncertainty.

While modularization is often used as a technique to split up and assign different functions of a complex system to different system parts, we split up and assign different design decisions to different models. A number of basic principles of modularity apply both to the functional decomposition of system parts (within a model) and to the separation of models in our design approach.

As is noted in [14]: "a complex engineering system is modular-in-design if (an only if) the *process of designing it can be split up and distributed across different separate modules*, that are coordinated by design rules, not by ongoing consultations amongst the designers." This definition reveals two important features of systems that are modular-in-design:

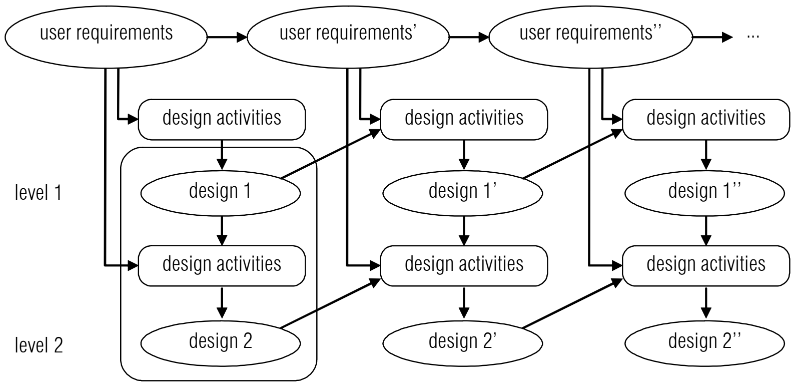
- *Independence*: The absence of ongoing consultations amongst the designers of different modules reveals that modules should be largely independent of each other. Modules correspond to independent activities in the design process; and
- *Dependence*: The relations between the different modules are defined by a set of design rules[6] to be respected. These design rules reflect the need for coordination of design choices. Separating strongly related modules forces the number of design rules to increase, constraining the freedom of designers of the different modules.

In the following sections, we examine independence and dependence of models in our design approach. We employ a technique to visualize modularity-in-design which uses Design Structure Matrices (DSMs) [101, 116]. DSMs have been employed extensively in the field of Engineering Design, both for products and production processes and design processes [14]. In this technique, modules are arrayed along the rows and columns of a square matrix. The matrix is filled in by determining, for each module, which other modules affect it and which are affected by it. The result is a map of the dependencies between the various modules.

4.2.2 Two levels of models

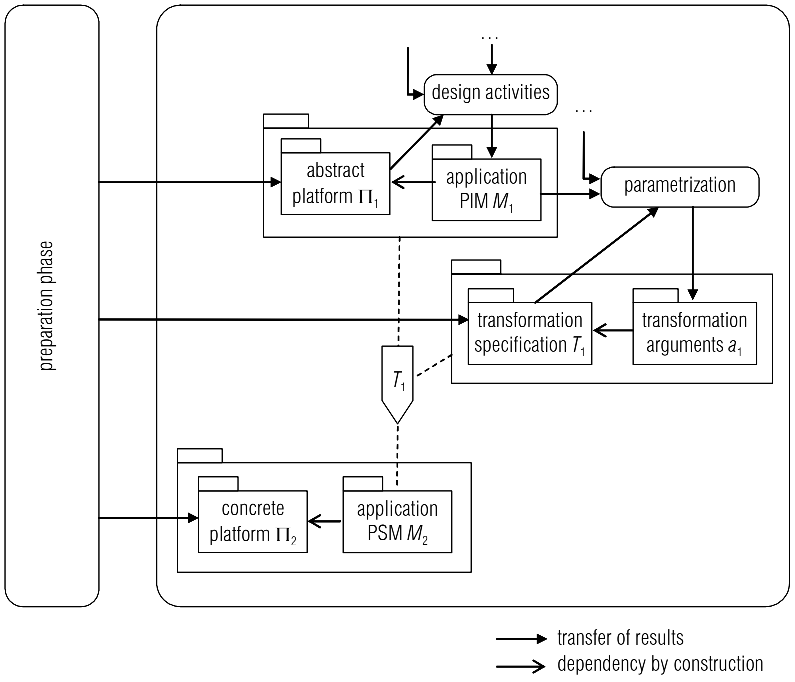
We start our analysis by assuming two levels of design within a single iteration cycle as depicted within the rounded rectangle in *Figure 4-2*.

Figure 4-2 Initial analysis assumes a design step



[6] In functional decomposition, interfaces between components are considered design rules.

We assume further that the preparation phase results in an abstract platform Π_1 for designs at level 1, a concrete (or realization) platform Π_2 for designs at level 2. The design activities are constrained by a transformation specification T_1 that relates models that rely on Π_1 to models that rely on Π_2. This situation is depicted in *Figure 4-3*. This figure reveals the various models of the execution phase that are considered at this point of our analysis, namely, an application PIM, transformation arguments, and an application PSM.


...
design activities
...
abstract platform Π_1
application PIM M_1
parametrization
preparation phase
transformation specification T_1
transformation arguments a_1
T_1
concrete platform Π_2
application PSM M_2
transfer of results
dependency by construction
Figure 4-3 Two levels of models related by transformation

We discuss the dependencies between each of the models depicted in *Figure 4-3* in the following sections. In each section, we discuss how the various models are affected as a result of a modification of one of the other models. After the relations between all models are examined, a DSM is built to visualize the dependencies between the various models.

Application PIM

Table 4-1 shows the dependencies between the various models and an application PIM. The '⊠' symbol marks the existence of some dependency. The absence of the symbol indicates there is no dependency. We justify the existence or absence of a dependency for each pair of models.

Table 4-1 Dependencies between the various models and an application PIM

	Application PIM	**Explanation**
Application PIM	N/A	trivial
Abstract platform		An abstract platform is designed so that is can be used to design a class of applications; the modified application PIM is still a member of this class of applications. This constitutes a generality requirement for abstract platform, but also sets the constraints on possible modifications of an application PIM for a given abstract platform.
Application PSM	⊠ through transformation	The relations between application PIMs and PSMs are determined by transformation specifications and transformation arguments; if the application PIM is modified, it is possible that the modified PIM and the original PSM no longer respect this relation; in this case, the PSM or transformation arguments may be affected by change.
Concrete platform		The concrete platform is a member of the set of platforms implied by portability requirements; all application PIMs that rely on the abstract platform must be buildable in the concrete platform, thus requiring no modifications in the concrete platform. This constitutes requirements for the abstract platform and transformation specification.
Transf. arguments	⊠	Transformation arguments are used to introduce variation in transformation specifications, in order to capture particular design decisions; these decisions may be application-specific or may refer to elements of the application PIM; e.g., transformation parameters can be used to specify the physical allocation of each application component in the application PIM.
Transf. specification		Transformation specifications are designed so that they can be applied to the class of applications that can be built on top of an abstract platform; the modified PIM is still a member of this class of applications. This constitutes a generality requirement for transformation specification.

Abstract platform

Table 4-2 shows the dependencies between the various models and an abstract platform.

Table 4-2 Dependencies between the various models and an abstract platform

	Abstract platform	**Explanation**
Application PIM	☒	By definition: "an abstract platform is an abstraction of infrastructure characteristics assumed in the construction of PIMs of an application" (see chapter 2). If these characteristics change, the application PIM may be affected.
Abstract platform	N/A	trivial
Application PSM		Modifying an abstract platform may affect PIMs, transformation specifications (see respective cells in this table), which in turn may affect application PSMs (see other tables); however, only direct dependencies are represented in a DSM.
Concrete platform		The set of target platforms is determined by portability requirements (see chapter 3); during abstract platform definition, buildability with respect to the target platform must be observed. This constitutes a requirement for abstract platform definition.
Transf. arguments		Transformation arguments depend on transformation specification, which depends on abstract platforms (see cell below); however, only direct dependencies are represented in a DSM.
Transf. specification	☒	The abstract platform defines the common characteristics of a class of platform-independent designs for which there should be generalized implementation relations to different platforms; these implementation relations are captured in transformation specifications; a change in abstract platform characteristics changes the class of applications, invalidating assumptions on common concepts, patterns and structures that were made to define transformations.

The separation between an abstract platform and a transformation specification is analogous to the separation between an interface definition and a realization of the interface in component-based design: an abstract platform defines requirements which are satisfied by one or several transformation specifications.

Application PSM

Table 4-3 shows the dependencies between the various models and an application PSM.

Table 4-3 Dependencies between the various models and an application PSM

	Application PSM	**Explanation**
Application PIM	☒ through transformation	The relations between application PIMs and application PSMs are determined by transformation specifications and transformation arguments; if the application PSM is modified, it is possible that the modified PSM and the original PIM no longer respect this relation; in this case, the PIM or transformation arguments may be affected by change[7].
Abstract platform		A modification in an application PSM may result in a modification in the application PIM (see cell *application PIM* above); the modified PIM is still a member of this class of applications for which the abstract platform is defined. This constitutes a generality requirement for abstract platform, but also sets the constraints on possible modifications of an application PSM for a given abstract platform.
Application PSM	N/A	trivial
Concrete platform		A concrete platform is designed so that is can be used to design a class of applications; the modified PSM is still a member of this class of applications. This constitutes a generality requirement for concrete platforms.
Transf. arguments	☒ through transformation	(see cell *application PIM* above)
Transf. specification		Transformation specifications define generalized implementation relations; transformation specifications define a class of PSMs that conform with PIMs; the modified PSM is still a member of this class of applications. This constitutes a generality requirement for transformation specifications, but also sets the constraints on possible modifications of an application PSM for a given transformation specification and a PIM).

[7] Our analysis of dependencies is valid regardless of whether transformation specifications are "unidirectional", "bidirectional" or "multidirectional" (in the sense of [79]). In this particular cell of the matrix, the only difference is that, in the case of a unidirectional transformation from PIM to PSM, changes to an application PSM cannot be propagated automatically to an application PIM or transformation arguments.

Concrete platform

Table 4-4 shows the dependencies between the various models and a concrete platform.

Table 4-4 Dependencies between the various models and a concrete platform

	Concrete platform	**Explanation**
Application PIM	independence is engineered	Independence is engineered in the definition of abstract platforms (see design criteria for abstract platform in chapter 3).
Abstract platform	independence is engineered	Independence is engineered in the definition of abstract platforms (see design criteria for abstract platform in chapter 3).
Application PSM	☒	Application PSM depends on sets of concepts, patterns and structures provided by a concrete platform; the instability of concrete platforms, and hence application PSMs, motivates separation of platform-independent and platform-specific concerns in our approach.
Concrete platform	N/A	trivial
Transf. arguments	☒	Transformation arguments may be platform-specific, e.g., markings may define that particular components should be transformed into Session or Message-Driven Enterprise Java Beans [103].
Transf. specification	☒	Transformation specifications define generalized implementation relations for a particular target platform; change the target platform and these relations may be invalidated.

Ideally, the dependency between concrete platforms and transformation specifications could be reduced by using concrete platform models as transformation arguments. However, this solution requires highly general transformation specifications, which define generalized implementation relations for a class of target platforms (resulting in a platform-independent transformation specification). For this solution to reduce the dependency between concrete platforms and transformation specifications, a modified target platform must still be a member of the class of target platforms.

Transformation arguments

Table 4-5 shows the dependencies between the various models and transformation arguments.

Table 4-5 Dependencies between the various models and transformation arguments

	Transf. arguments	**Explanation**
Application PIM		Abstract platforms are defined to preserve freedom of implementation, so that different implementations of application PIMs built on top of it are possible; since transformation arguments are used to introduce variations in generalized implementation relations, changes in transformation arguments should not affect application PIMs or abstract platforms. This constitutes a requirement for abstract platforms and transformations, and sets the constraints on possible modifications of transformation arguments for a given combination of abstract platform and transformation specification.
Abstract platform		(see cell *application PIM* above)
Application PSM	☒ through transformation	The relations between PIMs, transformation arguments and PSMs are determined by transformation specifications; if transformation arguments are modified, it is possible that the original PIM, the modified arguments and the original PSM no longer respect this relation; in this case, the PSM may be affected by change in transformation arguments.
Concrete platform		A concrete platform is designed so that is can support a class of applications; a PSM that is affected by a change in transformation arguments is still a member of this class of supported applications, therefore, requiring no modification of the concrete platform. This constitutes a requirement for transformation specification, namely that the results of transformations are always PSMs that use the concrete platform.
Transf. arguments	N/A	trivial
Transf. specification		Transformation specifications have transformation parameters, which are assigned values when the transformation specification is instantiated.

From the perspective of model transformation, the distinction between PIMs and transformation arguments is unnecessary: both PIMs and transformation arguments may be considered as input information for an unparameterized transformation. However, the distinction is relevant from the perspective of the design process: PIMs are *platform-* and *transformation independent*, while transformation arguments may be *platform-* and *transformation specific*. Transformation arguments may be defined after PIMs have been conceived. As a consequence, designers of PIMs may not be aware of whatever transformation parameters may be chosen by a designer using the PIM as a starting point to derive a PSM.

Transformation specification

Finally, *Table 4-6* shows the dependencies between the various models and a transformation specification.

Table 4-6 Dependencies between the various models and a transformation specification

	Transf. specification	**Explanation**
Application PIM		Abstract platforms are defined to preserve freedom of implementation, so that different implementations of application PIMs built on top of it are possible; these different implementations are captured in transformation specifications. This constitutes a requirement for abstract platform, but also sets the constraints on possible modifications of transformation specifications for a given abstract platform.
Abstract platform		(see cell *application PIM* above)
Application PSM	☒	The relation between application PIM and application PSM is determined by transformation specifications and transformation arguments; since a change in transformation specification should not affect PIMs (see cell *application PIM* above), modifications to transformation specifications must be accommodated in the PSM or in transformation arguments.
Concrete platform		PSMs related by transformation specifications must be realizable on top of a concrete platform. This constitutes a requirement for transformation specifications.
Transf. arguments	☒	Transformation parameters are used to introduce variations in generalized implementation specifications; if a transformation specification is modified, parameters may be modified and new parameters may be introduced, affecting transformation arguments.
Transf. specification	N/A	trivial

Since transformation arguments may be transformation-specific, transformation arguments must be captured separately from PIMs so that PIMs do not become transformation-specific. Therefore, in case of parameterization by marking, the unmarked PIM must be kept separately from markings. The unmarked PIM and markings can be combined into a separate marked model for the purposes of transformation if necessary.

Design Structure Matrix

Table 4-7 provides an overview of the dependencies between each of the models considered in our analysis so far. The columns of this table correspond to the columns of tables *Table 4-1* to *Table 4-6*. When the table is read row-wise, the '⊠' mark indicates that the model that names to the row is affected by the models that name each of the columns. When the table is read column-wise, the mark shows the models that may be affected directly as a result of a modification in the model that names the column.

Table 4-7 Dependencies between models: Design Structure Matrix

	Application PIM	Abstract platform	Application PSM	Concrete platform	Transf. arguments	Transf. specification
Application PIM	N/A	⊠	⊠ through transformation	independence is engineered		
Abstract platform		N/A		independence is engineered		
Application PSM	⊠ through transformation		N/A	⊠	⊠ through transformation	⊠
Concrete platform				N/A		
Transf. arguments	⊠		⊠ through transformation	⊠	N/A	⊠
Transf. specification		⊠		⊠		N/A

DSMs exhibit an interesting property for our analysis: if we consider that there is a time sequence associated with the position of the elements in the matrix, then all marks above the diagonal are considered feedback marks [122]. Feedback marks require iterations in the sequence of tasks executed. DSMs can be manipulated to eliminate or reduce feedback marks, e.g., by reordering the sequence of elements in the matrix. It is also possible to group elements of the matrix into clusters, a technique which allows us to consider the set of elements of a cluster as a single module .

In the following section, we manipulate the DSM represented in *Table 4-7* to show how the dependencies between models affect the design process.

4.3 Dependencies between models and the design process

4.3.1 Preparation and execution phase concerns

Table 4-8 shows a reordered DSM. The models that result from the preparation activities, namely, concrete and abstract platforms and transformation specifications are placed in the first three positions of the matrix, respectively. These models are grouped into a cluster, which represents the preparation phase. A second cluster represents the execution phase, grouping application PIM, transformation arguments and application PSM.

Table 4-8 Clustering dependencies with respect to preparation and execution activities

	Concrete platform	**Abstract platform**	**Transf. specification**	**Application PIM**	**Transf. arguments**	**Application PSM**
Concrete platform	N/A					
Abstract platform	independence is engineered	N/A				
Transf. specification	☒	☒	N/A			
Application PIM	independence is engineered	☒		N/A		☒ through transformation
Transf. arguments	☒		☒	☒	N/A	☒ through transformation
Application PSM	☒		☒	☒ through transformation	☒ through transformation	N/A

The absence of feedback marks above the diagonal formed by the preparation and execution phase clusters in *Table 4-8* shows that the preparation phase does not depend on the execution phase. This result is made possible by requirements imposed on the preparation phase. These requirements are described in the cells of tables *Table 4-1* to *Table 4-6* that correspond to the cells positioned above the diagonal formed by the two clusters. Failure to satisfy these requirements would imply the presence of feedback dependencies, which would require revisiting the preparation phase. The absence of feedback marks above the diagonal formed by the preparation and execution phase clusters can be summarized by the following design rule:

Changes in PIM, PSM or transformation arguments must be accommodated in PIM, PSM or transformation arguments, but not in the abstract platform, concrete platform or transformation specification.

Table 4-8 also reveals the absence of feedback dependencies within the preparation phase, since, within the cluster, no feedback marks appear above the diagonal. The same, however, cannot be said of the execution phase: modifications in the application PSM may affect the PIM and transformation arguments.

4.3.2 Platform-independent and platform-specific concerns

The presence of feedback dependencies in the execution phase is addressed through iteration in the execution phase. An iteration in the execution phase allows a designer to gain insight into the implications of design decisions at the PIM-level for the application PSM, which may result in adjusting the PIM in a subsequent iteration.

However, for the design process to advance towards a stable application PIM, it is necessary that the dependencies between PSM and PIM should eventually decrease, as indicated in *Figure 4-4*. Eventually, the application PIM must be such that it does not depend on design decisions that constrain the choice of target platform. This constitutes an important requirement for the iterative approach in the execution phase.

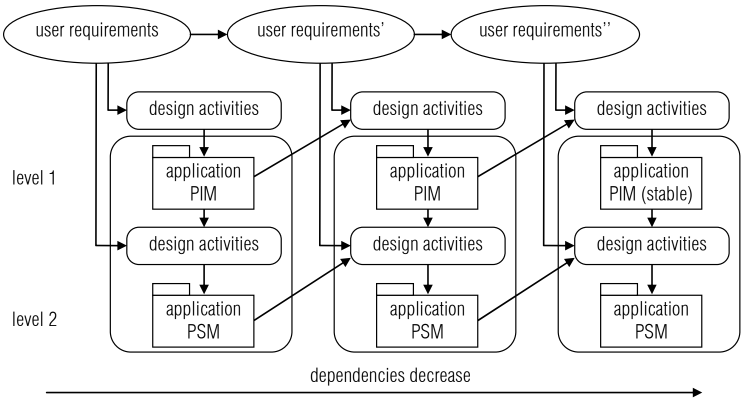


Figure 4-4 Dependency between PIM and PSM is addressed through iteration

In order to respect this requirement, a designer must be able to distinguish between platform-independent and platform-specific design decisions. Nevertheless, the distinction between platform-independent and platform-specific design decisions is not always obvious, particularly because platform characteristics may impact designs at different levels of abstraction. In order to illustrate this, let us consider the design of a groupware service that facilitates the interaction of users residing in different geographical locations.

Initially, the service designer describes the service solely from its external perspective, possibly stating quality-of-service requirements on the service, e.g., that the service should have high availability. At subsequent stages of development, the designer is confronted with design decisions. In this example, we consider the following alternatives: (i) a centralized (server-based) design, and (ii) a distributed (peer-to-peer) design.

Figure 4-5 depicts these two solutions. In solution (i), a server facilitates the interaction between users. In solution (ii), symmetric components facilitate the interaction without the support of a centralized application-level component.

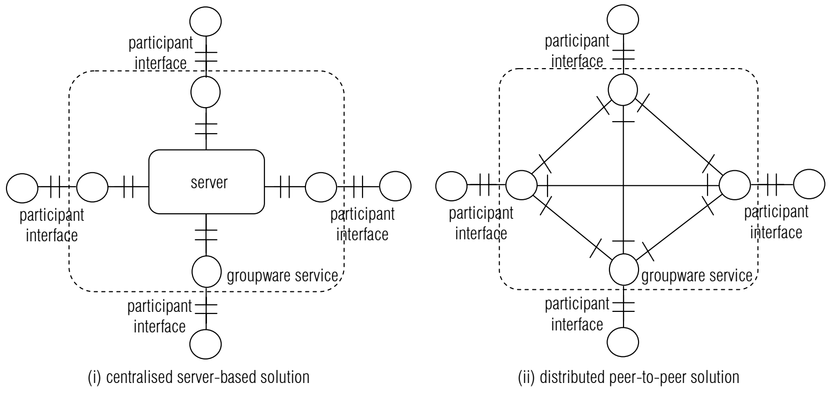

participant interface
server
groupware service
(i) centralised server-based solution
(ii) distributed peer-to-peer solution

Figure 4-5 Alternative designs for the groupware service

Solution (i) introduces a single point of failure, unless the platform provides support for replication transparency (as defined in the Reference Model for Open Distributed Processing (RM-ODP) standards [58, 59]). Solution (ii), in contrast, facilitates interaction without the support of a centralized application-level component and, hence, does not require replication transparency.

If one of alternative solutions is to be chosen and captured in a PIM, this PIM would break the requirement we have stated for stable PIMs, since platform selection would affect platform-independent design.

In order to solve this, a designer should be able to express, at a platform-independent level, requirements on platform-specific realizations that would allow all design decisions that are relevant for platform-independent modelling to be captured. In our groupware service example, this would mean that requirements on the reliability of individual components should be stated at the platform-independent level, justifying the selection of a centralized or a distributed design.

Requirements expressed at a platform-independent level should justify design decisions for the design at that level, and provide input for platform-specific realization. If these requirements invalidate portability requirements for platform-independent designs, then it is impossible to consider the design at the current level of platform-independence. In this case, we envision two different contrasting solutions:

(a) to consider the design at a higher level of abstraction, at which the platform characteristics are no longer relevant for design decisions taken at that level; or,

(b) to relax portability requirements, lowering the degree of platform-independence for the design.

For our groupware service example, possible applications of these solutions would be:

(a) to describe the groupware service solely from its external perspective. At this level of abstraction, the reliability characteristics of the supporting infrastructure are irrelevant. Details on the service's internal design are only addressed at platform-specific modelling, and hence cannot be re-used for different target platforms; and,

(b) to restrict the set of potential target platforms, e.g., to include only platforms that provide support for highly available components. In this case, it is possible to describe the groupware service's internal design at the newly defined level of platform-independence, while still guaranteeing the satisfaction of the service requirements. The abstract platform considered provides support for highly available components.

4.3.3 Multiple levels of models

We continue our analysis by considering the dependencies between the models at three different levels related by transformation. This situation is depicted in *Figure 4-6*.

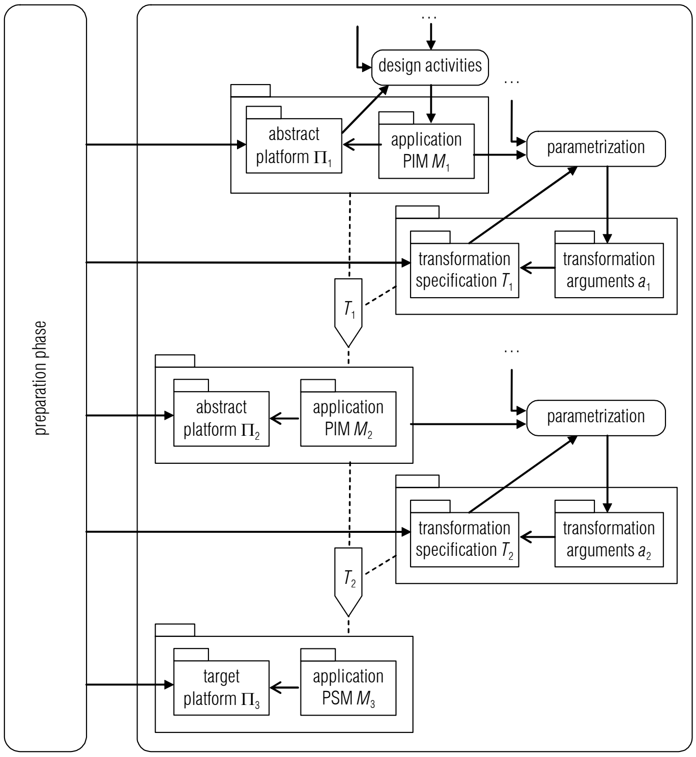

preparation phase
design activities
abstract platform Π_1
application PIM M_1
parametrization
transformation specification T_1
transformation arguments a_1
T_1
abstract platform Π_2
application PIM M_2
parametrization
transformation specification T_2
transformation arguments a_2
T_2
target platform Π_3
application PSM M_3

Figure 4-6 Three levels of models related by transformations

Table 4-9 shows the dependencies between the models depicted in *Figure 4-6*. These dependencies are clustered for each pair of consecutive levels of models, i.e., a cluster for models of levels 1 and 2 and a cluster for models of levels 2 and 3. This DSM is build by reapplying the transformation pattern, which explains the isomorphic nature of the dependencies in the two clusters.

Table 4-9 Clustering dependencies with respect to levels of models

	Abstract platform Π_1	Application PIM M_1	Transf. specification T_1	Transf. arguments a_1	Abstract platform Π_2	Application PIM M_2	Transf. specification T_2	Transf. arguments a_2	Concrete platform Π_3	Application PSM M_3
Abstract platform Π_1	N/A									
Application PIM M_1	☒	N/A				☒				
Transf. specification T_1	☒		N/A		☒					
Transf. arguments a_1		☒	☒	N/A	☒	☒				
Abstract platform Π_2					N/A					
Application PIM M_2		☒	☒	☒	☒	N/A				☒
Transf. specification T_2					☒		N/A		☒	
Transf. arguments a_2						☒	☒	N/A	☒	☒
Concrete platform Π_3									N/A	
Application PSM M_3						☒	☒	☒	☒	N/A

The table shows an overlap between the two clusters. This overlap indicates that the design activities in the different levels are not completely independent, and that the intermediate model PIM forms the 'interface' between the two clusters, as could be expected.

Preparation and execution activities with multiple levels of models

We modify the sequence of models in the matrix and cluster the preparation and execution activities separately. The result is presented in *Table 4-10*. Again, the absence of feedback marks above the diagonal formed by the preparation and execution phase clusters shows that the preparation phase does not depend on the execution phase. Feedback dependencies in the execution phase are shown by marks above the diagonal in the execution phase cluster.

Table 4-10 Clustering dependencies with respect to preparation and execution activities

	Abstract platform Π_1	Abstract platform Π_2	Transf. specification T_1	Concrete platform Π_3	Transf. specification T_2	Application PIM M_1	Transf. arguments a_1	Application PIM M_2	Transf. arguments a_2	Application PSM M_3
Abstract platform Π_1	N/A									
Abstract platform Π_2		N/A								
Transf. specification T_1	☒	☒	N/A							
Concrete platform Π_3				N/A						
Transf. specification T_2		☒		☒	N/A					
Application PIM M_1	☒					N/A		☒		
Transf. arguments a_1		☒	☒			☒	N/A	☒		
Application PIM M_2		☒	☒			☒	☒	N/A		☒
Transf. arguments a_2				☒	☒			☒	N/A	☒
Application PSM M_3				☒	☒			☒	☒	N/A

Let us consider the execution phase in isolation. The DSM for the execution phase is shown in *Table 4-11*. An attempt to create independent clusters within the execution phase will for each pair of consecutive levels of models, results in the overlap between the different clusters, confirming our observations with respect to *Table 4-9*, namely that the activities in the different levels of models are not independent of each other.

Table 4-11 Clustering dependencies with respect to levels of models in the execution phase

	Application PIM M_1	Transf. arguments a_1	Application PIM M_2	Transf. arguments a_2	Application PSM M_3
Application PIM M_1	N/A		☒ through transformation		
Transf. arguments a_1	☒	N/A	☒ through transformation		
Application PIM M_2	☒ through transformation	☒ through transformation	N/A		☒ through transformation
Transf. arguments a_2			☒	N/A	☒ through transformation
Application PSM M_3			☒ through transformation	☒ through transformation	N/A

As we have discussed in 4.2.2, these feedback dependencies are addressed with iterations in the execution phase.

4.4 Concluding remarks

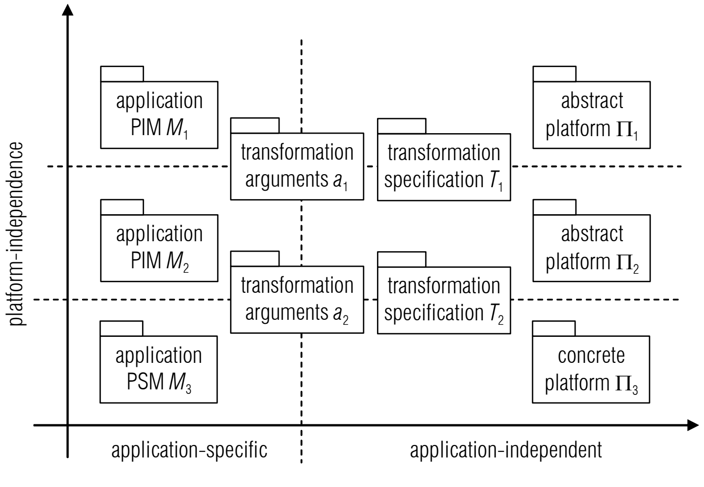
4.4.1 Classification of models

This section concludes our analysis by classifying the various models and design decisions according to the following dimensions of separation of separation of concerns:

- platform-independent and platform-specific concerns;
- application-independent and application-specific concerns, which correspond to preparation and execution phases concerns, respectively; and,
- transformation-independent and transformation-specific concerns.

Figure 4-7 places the different models according to the first two dimensions. Three levels of models are depicted.

Figure 4-7 Dimensions of separation of concerns and models


platform-independence
application PIM M_1
transformation arguments a_1
transformation specification T_1
abstract platform Π_1
application PIM M_2
transformation arguments a_2
transformation specification T_2
abstract platform Π_2
application PSM M_3
concrete platform Π_3
application-specific
application-independent

In *Figure 4-7*, transformation specifications are placed in the boundary between two levels of platform-independence. This is to denote that transformation specifications rely on the (abstract) platforms of both source and target levels of models (see *Table 4-2* and *Table 4-4*). In addition, transformation specifications may also capture some transformation rules which are independent of the target platform.

Similarly to transformation specifications, transformation arguments are also placed in the boundary between two levels of platform-independence. In addition, transformation arguments are placed in the boundary between the application-specific and application-independent concerns area. This is to denote that arguments may be application-specific (see *Table 4-1* row "transformation arguments"), but may also capture application-independent design decisions. Application-independent transformation parameterization is used to improve flexibility of transformation specifications in general, e.g., to cope with to variation in user requirements that are not captured in the source models but that are to be addressed during transformation. An example of an application-independent transformation argument determines that, irrespective of the application model, all application parts should be allocated to the same unit of deployment of the target platform.

Although not apparent in *Figure 4-7*, the separation of application-specific and application-independent concerns is not the same at the different levels of models. Each level of models is defined in the preparation phase, and consists of a different balance of the quality criteria we have discussed in chapter 3.

In addition to the dimensions considered in *Figure 4-7*, we can also classify models related in a transformation step as *transformation-independent* or *transformation-specific*. This classification is relative to a transformation specification. In a transformation step, the source application model is transformation-independent (with respect to a transformation specification from that level of models), since it relies on an abstract platform, which is itself transformation-independent (see *Table 4-6*). In constrast, the target application model and the transformation arguments can be classified as transformation-specific. This dimension helps to determine whether design decisions should be captured at the source application model level (which may only capture transformation-independent design decisions) or at transformation arguments (which may capture transformation-specific design decisions).

4.4.2 Main conclusions and directives

From the analysis of the relations between the various models, we can conclude that:

– *Feedback dependencies between execution and preparation phases can be avoided by addressing generality requirements at the preparation phase*. Failure to address these requirements results in cycles between the execution and preparation phases;
– *Platform-independent and platform-specific models are interrelated, their dependencies defined by transformation*. The interrelation between PIMs and PSMs is addressed through iteration in the execution phase. An iteration in the execution phase allows a designer to gain insight into the implications of design decisions at the PIM-level.
– *The distinction between platform-independent and platform-specific concerns is not obvious and is constrained by the interdependencies between design decisions*. This is apparent in the groupware service example we have presented, in

which some platform characteristics impact the definition of a distributed application's architecture.

Our analysis leads to the following directives for the design process:

– *Changes in PIM, PSM or transformation arguments must be accommodated in PIM, PSM or transformation arguments, but neither in the abstract platform, concrete platform nor transformation specification.*
– *Dependencies between PIM and PSM are handled by iterations in the execution phase, leading to a stable application PIM that does not depend on platform-specific design decisions.*
– *Interdependent design decisions must be captured at the same level of platform-independence. Since some design decisions are platform-specific, this imposes constraints on the organization of models at different levels of platform-independence*[8]*.*
– *The classification of models according to the various dimensions of concerns*[9] *serves as a guideline to determine in which models design decisions should be captured.*

[8] see section 4.3.2 for approaches to coping with interdependent design decisions
[9] see section 4.4.1 for the classification

Chapter 5

Design framework

We have discussed in chapter 3 that a number of design goals and design criteria influence the definition of abstract platforms in the preparation phase of the design process. We have concluded that different design goals in different projects and different stages of the design process may lead to different abstract platforms. It is, therefore, necessary to design abstract platforms in the preparation phase of our design process. In this chapter, we define a design framework, whose purpose is to support a designer in defining suitable abstract platforms.

This design framework consists of two parts: *a set of basic design concepts*, which are used at different levels of platform-independence to describe both abstract platforms and the platform-independent designs that rely on them, and *design operations*, which can be used in transformations to bridge between different levels of platform-independence. An important principle underlying the proposed design framework is that it should enable a designer to make statements about the conformance of designs at different levels of platform-independence.

This chapter is organised as follows. Section 5.1 provides an overview of our design framework. Section 5.2 introduces the basic design concepts in the framework, focussing on the role of the service concept. Two types of design operations are introduced: service decomposition and interaction refinement. Service decomposition and interaction refinement are discussed in further detail in sections 5.3 and 5.4. Section 5.5 relates our framework to the RM-ODP. Section 5.6 presents an evaluation of the design framework, according to the quality criteria defined in chapter 3. Finally, section 5.7 discusses related work and presents some concluding remarks.

5.1 Overview

In the previous chapters, we have argued that the design of a system can be considered at various levels of platform-independence in a model-driven design process. An initial design in a model-driven design process is given at a high level of platform-independence, meaning that it considers little or none of the constraints that a platform imposes on the way in which that design can be implemented. During the design process, a designer must gradually consider these constraints, and the means to incorporate them into designs. Eventually, this should lead to a design at a sufficiently low level of platform-independence such that the realization of the design becomes straightforward.

For these reasons, a model-driven design process requires design concepts and supporting modelling languages that are abstract enough to construct designs in which no specific platform constraints are imposed. At the same time, this design process requires design concepts that allow the construction of designs at a sufficiently detailed level to describe how the design can (eventually) be realized.

In our design framework, we adopt a basic set of design concepts that can be used to support design at various levels of platform independence. We use the concept of *service* [115] to describe application parts from an external perspective, which allows us to abstract from characteristics of middleware platforms that are eventually used to realize the internal design of an application part. This technique is particularly useful when interaction aspects of applications parts are captured as separate objects of design, which are called *interaction systems* [112].

The service of an application-level interaction system is used as a starting point for *service decomposition*, which should result in a design of the interaction system into a structure of interaction system parts interconnected by an underlying abstract platform. This technique can be applied recursively, in which case the abstract platform is itself described as a service, until a transformation into a realization platform can be established.

We use the concept of *abstract interaction* in order to abstract from particular interaction mechanisms that may be used for the interaction between application parts. Designers relate abstract interactions to their realizations in middleware platforms by applying *interaction refinement*, possibly using intermediate abstract platforms.

Figure 5-1 depicts service decomposition and interaction refinement schematically. It also shows that these techniques can be applied in combination. Rounded rectangles represent the behaviour of system parts and arrows between rounded rectangles represent abstract interactions.

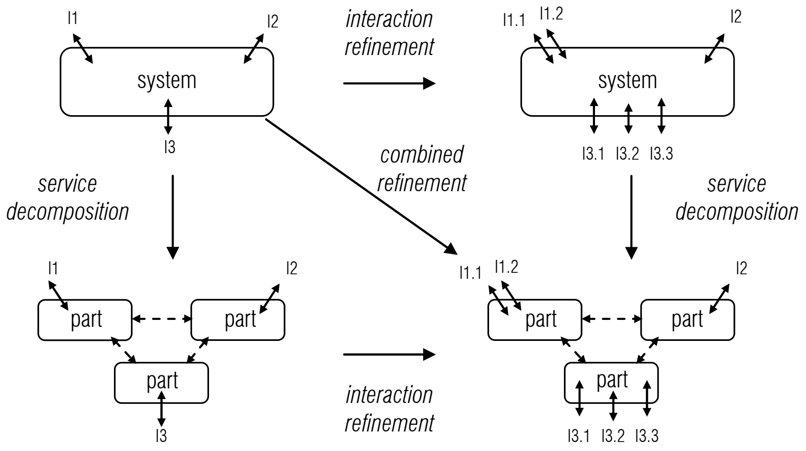


Figure 5-1 Approaches to system refinement [39]

In service decomposition (which is called *interaction allocation and flowdown* in [119]), the designer decomposes the application parts into smaller parts and allocates the existing interactions to these parts. In this case, the interactions remain unchanged, except for the introduction of new (internal) interactions between the smaller parts. In interaction refinement, the designer refines the interactions between the application parts and their environment without changing the granularity of the parts, i.e., without decomposing the parts into smaller parts [39].

Service decomposition and interaction refinement are applied in design steps, either incorporated into automated transformations or performed manually by a designer. In either case, the design step comprises design decisions, which modify a source design. As we have discussed in chapter 2, these decisions must preserve the characteristics of the system that are defined by the source design. This is reflected in the proposed design framework in that service decomposition and interaction refinement must result in conformant refinements of designs.

5.2 Design concepts

5.2.1 The service concept

According to the Webster's dictionary: "A system is a regularly interacting or interdependent group of items forming a unified whole". This definition indicates two different perspectives of a system: an integrated and a distributed perspective [91]. The integrated perspective considers a system as a whole or black box, defining only what function a system performs for its environment. The distributed perspective defines how this function is performed by an internal structure in terms of system parts (which are also systems) and their relationships. *Figure 5-2* depicts both system perspectives.

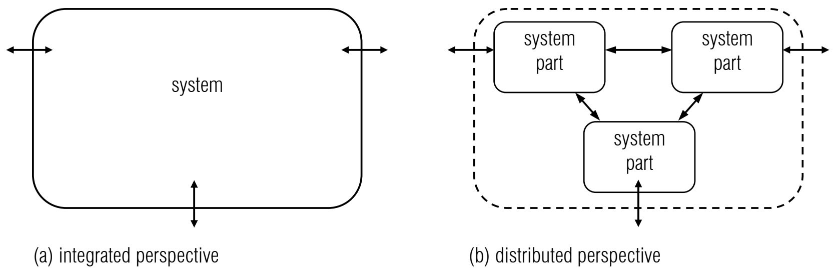


Figure 5-2 Integrated and distributed perspective of a system [91]

When the behaviour of a system is considered according to the integrated perspective, we call the description of this behaviour a *service* [115]. A service is a design that defines the observable behaviour of a system in terms of the interactions that may occur at the interfaces between the system and the environment and the relationships between these interactions[10]. A service does not disclose details of an internal organization that may be given to implementations of the system [114].

Since the concept of system is recursive, in the sense that a system part is a system in itself, the service concept can be applied recursively in a system. The recursive application of the service concept allows a designer to consider the behaviour of a system at different related decomposition levels. In general, the number of decomposition levels and the particular choices for decomposition depend on particular system requirements and objectives of a designer.

5.2.2 Interaction systems

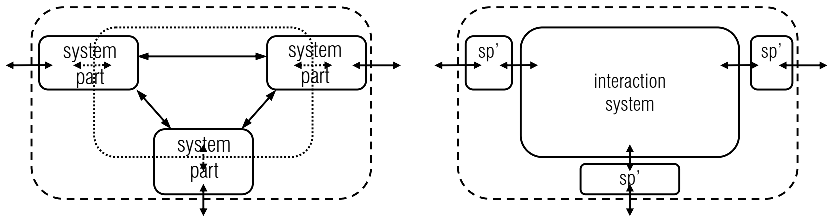
The distributed perspective of a system (depicted in *Figure 5-2*(b)) shows that a system consists of interacting system parts. In this perspective, a designer focuses on system part design and the interactions between system parts are defined implicitly in the composition of system parts. An alternative to this perspective identifies *interaction systems*, which are systems that support the set of related interactions between two or more systems parts [99, 100].

Figure 5-3 depicts two views of an interaction system: (a) an interaction system as consisting of parts of the system parts that were identified in the distributed perspective (in the previous section), and (b) an interaction

[10] The notion of abstract interaction is introduced in detail in section 5.4. At this point, we assume an intuitive notion of interaction, as a shared action between two or more system parts that results in the establishment of information.

system as a separate system. In the former view, the boundaries of the interaction system (show in dotted lines) divide each system part into two. The latter view redefines the original system parts to exclude the functionality that is attributed exclusively to the interaction system.

Figure 5-3 Introducing an interaction system



(a) in the distributed perspective, interaction system consists of parts of system parts

(b) interaction system is considered as separate system, system parts are redefined

The complexity of interaction systems varies, depending on the interactions that need to be considered. For example, when interactions concern multiparty agreement or business negotiations, the interaction system will be more complex than when datagram transfer is considered.

The benefits of explicitly designing the interaction mechanisms between distributed system parts has been acknowledged in the past in seminal work in the area of systems and protocol design [115]. A systematic design method for protocols [112] consists of: (i) defining the service to be supported by a service provider in terms of the service primitives that occur at service access points and the relationships between service primitives; and, (ii) decomposing this service in terms of a structure of protocol entities and a lower level service. This resulting structure, which is called a *protocol*, has to be a correct implementation of the service.

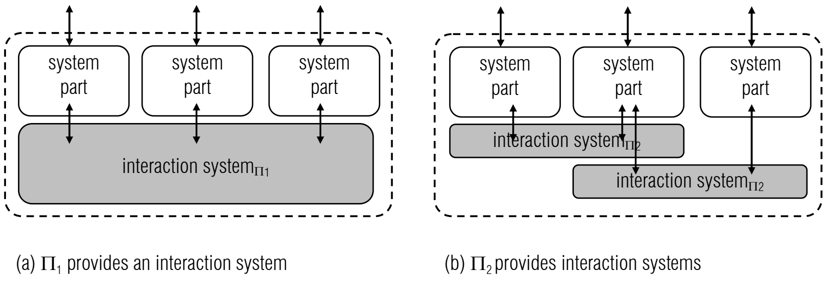
The importance of interaction mechanisms for distributed applications has been recognized also in standardization efforts. In particular, the RM-ODP [56] has introduced the notion of a binding object, which is responsible for facilitating the interaction between objects in the Computational Viewpoint.

More recently, efforts in the area of Software Architecture (e.g., [3]) have identified the "connector" construct. Connectors satisfy basic communication needs between software components, thus emphasizing the importance of describing and analysing interaction aspects of software components in software architectures.

5.2.3 Middleware platforms and interaction systems

Middleware platforms can be seen as providing interaction systems for the interconnection of application parts, as depicted in *Figure 5-4*.

Figure 5-4 Middleware can be regarded as providing interaction system(s)


system part
system part
system part
interaction system$_{\Pi 1}$
(a) Π_1 provides an interaction system
system part
system part
system part
interaction system$_{\Pi 2}$
interaction system$_{\Pi 2}$
(b) Π_2 provides interaction systems

Different middleware platform offer different types of interaction systems, for example, CORBA/CCM [73, 75], .NET [68], Java RMI [102] and Web Services [120, 121] offer interaction systems based on a request response pattern, which is realized by a generic interaction system, as depicted in *Figure 5-4*(a). CORBA (with the Event Service) [75], the Java Messaging Service (JMS) [104] and many other so-called Message-Oriented Middleware (MOM) platforms, offer interaction systems based on *event channels*, or *message queues*. Each of these channels of queues can be regarded as a separate interaction system, as depicted in *Figure 5-4*(b).

The interaction systems provided by the various middleware platforms have different characteristics. In particular, the way in which a design that uses these interaction systems can be structured is often subject to platform-imposed restrictions or constraints. For example, in the CORBA platform, operation invocation between objects is supported, however, only a single interface per object is supported. A consequence of the differences in the various interaction systems provided by middleware platforms is that designs of application parts that rely on the service of these interaction systems are platform-specific.

5.2.4 Application interaction systems

Instead of defining the interconnection of application parts directly in terms of the interaction systems provided by a middleware platform, it is possible to identify *application interaction systems* that support application-level interactions between application parts. *Figure 5-5* illustrates the identification of an application interaction system as a separate system.

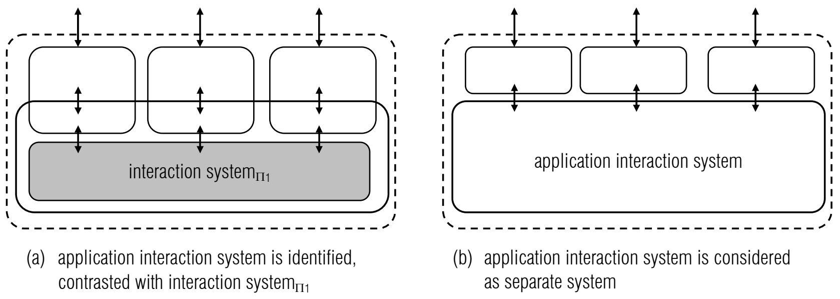

interaction system$_{\Pi 1}$
application interaction system
(a) application interaction system is identified, contrasted with interaction system$_{\Pi 1}$
(b) application interaction system is considered as separate system

Figure 5-5 Introducing an application interaction system

Whether or not the design of application interaction systems should be considered explicitly depends on the application requirements and on the objectives of the designer [4, 99]. In the following situations, application interaction system design should be considered:

- *The relation between system parts is complex.* In this case, designers should pay attention to the design of the relation between system parts. Designers can make this relation a separate design object, i.e., considering the system parts' interaction system separately. Designers can consider the interaction system at different abstraction levels to cope with the relation's complexity. The middleware-provided interaction system plays an important role at lower abstraction levels.
- *Alternative internal designs for the interaction system are expected.* This occurs, e.g., when the designer anticipates the use of different middleware platforms as alternatives to support the interactions. A designer can only replace an interaction mechanism by another equivalent interaction mechanism if the design clearly indicates the mechanism's relevant characteristics. Interaction system design naturally supports this.
- *The interaction system is general-purpose, offering opportunity for reuse.* Interaction systems provided by middleware platforms are an example of general-purpose interaction systems.
- *Different design authorities are responsible for the process of designing the interaction system and system parts.* Specifying the interaction system's service serves as a contract for the communication between system part and interaction system designers.

An interaction system is a system in itself, and therefore the behaviour of an interaction system can be defined as a service. A starting point in the design of an application interaction system is the specification of its service. The design of the application interaction system may, in principle, have any internal structure as long as it provides the required service. For example, it may make use of a data transport service via an application protocol as in a protocol approach [98]. Nevertheless, we observe that the middleware leverages the reuse of a large building block that provides an interoperability architecture across programming languages, operating systems and network technologies. Furthermore, middleware often supports information exchange at the application level, e.g., with the definition of information structures in an interface definition language (such as CORBA IDL [73]). A consequence of that is that designers do not have to be concerned with encoding and decoding pieces of information in protocol data units, which is necessary in a protocol-based approach. For these reasons, we argue that interaction systems provided by the middleware should be considered for building application interaction systems.

Nevertheless, if we structure the design of an application interaction system in terms of the constructs provided by a particular middleware platform, the design of the application interaction system would not be suitable for realizing this design on multiple platforms. Therefore, we define a platform-independent service design in terms of an abstract platform. Later in the design process, platform-independent design is realized on top of a concrete-platform.

5.2.5 Interaction systems provided by abstract platforms

Abstract platforms can be seen as providing interaction systems for the interconnection of application parts described in a platform-independent manner. These interaction systems can also be described by using the service concept, in which case the model-level approach to abstract platform definition is used (see chapter 2).

In this case, the interaction system provided by an abstract platform and an application-level interaction system are similar to each other. The main distinction lies in the criteria used for their definition. The application-level interaction system is defined according to the criteria defined in section 5.2.4 and the abstract platform is defined according to the criteria discussed in chapter 3. The latter include the requirement of buildability in different target platforms, and thus, the interaction systems provided by an abstract platform are defined by considering the characteristics of potential target platforms. Furthermore, we treat application-level interaction systems and abstract platforms as distinct because different realization techniques apply for application-level interaction systems and abstract platforms, as we discuss in section 5.3. The similarity between abstract platforms and application-level interaction systems does not exist when the language-level approach to abstract platform definition is used.

In addition to interaction systems, abstract platforms may also provide other parts to be composed with application parts. For example, an abstract platform may include a service discovery or service trader component, such as the RM-ODP trader. Interaction systems and other parts provided by an abstract platform should respect the criteria for abstract platform definition as defined in chapter 3 of this thesis.

5.3 Service decomposition

The starting point for service decomposition is either a design of the application that consists of application parts and an application interaction system (section 5.3.1) or a design of the application that consists of application parts and interaction systems provided by an abstract platform (section 5.3.2).

5.3.1 Application interaction system decomposition

When the design of the application consists of application parts and an application interaction system, the service of the application interaction system can be decomposed if necessary into a number of service components and an underlying service. This underlying service may represent a simpler application interaction system, in case the criteria defined in section 5.2.4 apply to this underlying service, or it may represent an abstract platform, as proposed in section 5.2.5. The latter alternative is depicted in *Figure 5-6*.

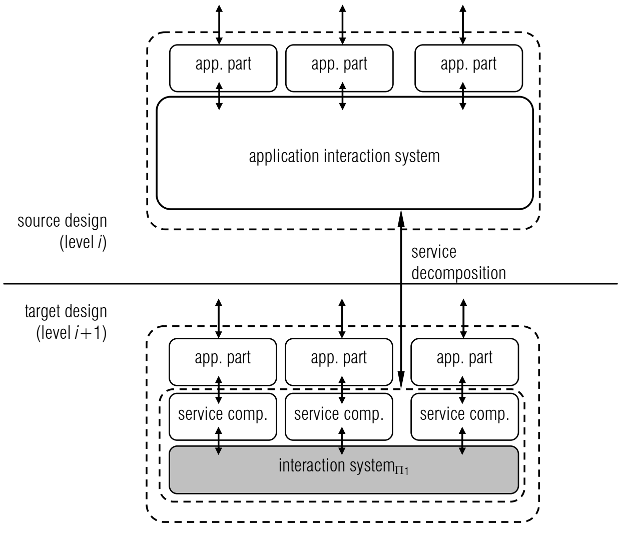
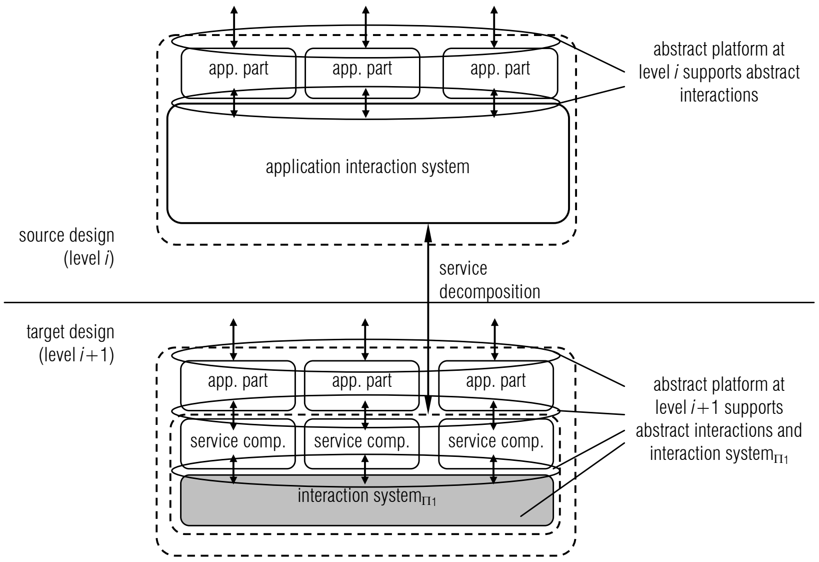


Figure 5-6 Service decomposition, underlying interaction system provided by abstract platform

A consequence of the application of this structuring technique is that application parts that rely on the application interaction system are potentially defined at a high-level of platform-independence. The structure of these application parts is not directly dependent on the interaction systems provided by the abstract platform. The platform-independent level at which application parts are defined is also "paradigm"-independent (as in [23]), in the sense that it does not imply characteristics of a target platform, and, therefore, a broad set of middleware platforms that support different interaction patterns can potentially be used to support the interaction between application parts.

While the design of the application parts does not depend on the interaction systems that can be used in the internal design of the application interaction system, this design depends on the support for abstract interactions between application parts and the application interaction system. *Figure 5-7* identifies the abstract platforms at the two levels of design in a service decomposition design step. In the source design, the abstract platform supports abstract interactions, and in the target design, the abstract platform both supports abstract interactions and provides an interaction system.

Figure 5-7 Abstract platforms at the different levels with service decomposition


app. part
app. part
app. part
abstract platform at level i supports abstract interactions
application interaction system
source design (level i)
service decomposition
target design (level $i+1$)
app. part
app. part
app. part
service comp.
service comp.
service comp.
abstract platform at level $i+1$ supports abstract interactions and interaction system$_{\Pi 1}$
interaction system$_{\Pi 1}$

We discuss the refinement of abstract interactions in section 5.4. In the next section, however, we focus on the decomposition of interaction systems that abstract platforms provide (such as the one shown in grey in *Figure 5-7*, level $i+1$). This abstract platform supports the interaction between service components, which are, therefore, defined at the level of platform-independence that is provided by this abstract platform.

5.3.2 Abstract platform decomposition

Whether or not an application interaction system is used, the platform-independent design of an application can be defined as a composition of application parts and the abstract platform. The platform-independent design is used as input to a design step, which results in a design at a lower level of platform-independence. The resulting design is structured in terms of the target (abstract) platform for the design step and parts that depend on this platform (the platform-specific application design from the perspective of the target platform).

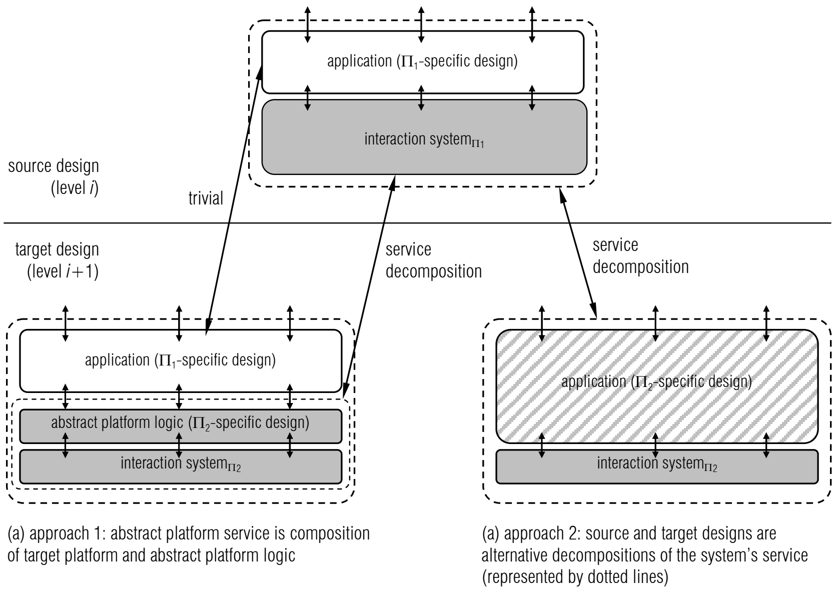
In general, we distinguish two contrasting extreme approaches for this step, namely to:

1. *Adjust the target platform*, so that it corresponds directly to the abstract platform of the source design, or;
2. *Adjust the target application design* so that the application design can be composed with the target platform.

In approach 1, the boundary between abstract platform and source application design is preserved during the design step (see *Figure 5-8*(a)). This implies the introduction of some target-platform-specific *abstract platform logic* to be composed with the target platform. The abstract platform service is a composition of target platform and abstract platform logic. The correspondence between source application design and target application design is trivial in this case.

In approach 2, the boundary between abstract platform and source application design is lost during the design step (see *Figure 5-8* (b)). In order to establish a correspondence between source and target designs one must compare the external behaviour of both source and target designs.

Figure 5-8 Alternative approaches to proceeding with design


application (Π_1-specific design)
interaction system$_{\Pi_1}$
source design (level i)
trivial
target design (level $i+1$)
service decomposition
service decomposition
application (Π_1-specific design)
abstract platform logic (Π_2-specific design)
interaction system$_{\Pi_2}$
application (Π_2-specific design)
interaction system$_{\Pi_2}$
(a) approach 1: abstract platform service is composition of target platform and abstract platform logic
(a) approach 2: source and target designs are alternative decompositions of the system's service (represented by dotted lines)

Approach 1 provides clear correspondence between source and target designs. Abstract platform logic is application-independent and can be directly reused for other platform-independent designs that rely on the same abstract platform. Approach 1 is explicitly enabled by the identification and definition of the service of interaction systems provided by an abstract platform, and allows us to obtain application software components that can be reused on top of different platforms. Approach 1 can be generalized as a recursive application of service definition (external perspective) and service decomposition (service's internal design), resulting in a hierarchy of abstract platforms and (ultimately) a realization platform.

Approach 2 cannot be seen as decomposition of the service of the abstract platform. Therefore, the service of the abstract platform is not used as a starting point in this design step. We consider the target design to be an alternative decomposition of the service provided by the source design as a whole, i.e., the composition of the source application and abstract platform. If a description of this service is not available, a correspondence cannot be established in terms of service decomposition.

The choice for approach 2 is not justified by top-down rationale. Instead, it is justified by bottom-up arguments. For example, a realization of target designs obtained through approach 1 may not satisfy time performance requirements, e.g., due to the use of a layered software architecture that preserves the structure of target platform and abstract platform logic. Another bottom-up argument for choosing for approach 2 may be that it is not possible to adjust the target platform by introducing some abstract platform logic, e.g., due to the lack of extension mechanisms of the target platform or due to the cost of development of these extensions.

Both approaches allow us to target different target platforms from the same platform-independent model. In approach 1, the gap in buildability is reflected in the complexity of abstract platform logic. In approach 2, the gap in buildability is reflected in the complexity of adjustments to the application design during transformation.

At each design step from a source level i to a target level $i+1$, both approaches to realization can be chosen. This leads to innumerable possible structures for designs at different levels of platform-independence. Service decomposition stops when the target platform provides the service of the interaction system.

5.3.3 Example: the service of a floor-control interaction system

In order to illustrate the use of an application service in a design trajectory, we introduce a running example, namely, the *floor-control* application. In this example, several application parts share a set of named resources. Each of these resources can only be used by a single application part at a time, and hence application parts have to coordinate their behaviours in order to ensure that there is no concurrent use of a resource. Application parts are assumed to be cooperative, i.e., they do not use the resources indefinitely. In addition, no pre-emption of control over a resource is necessary.

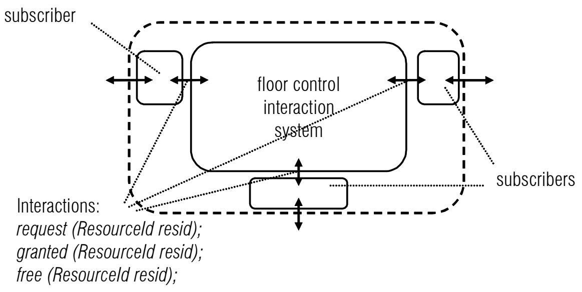
The service of the floor-control interaction system relates the following interactions: *request*, *granted* and *free*. These interactions occur at the boundary between the floor-control service and each of the application parts, which we call *subscribers*. A result of the occurrence of each of these interactions is the establishment of the resource identification and the identification of the subscriber. The latter is implied by the location where the interaction occurs. The occurrence of the *request* interaction means that a subscriber needs to use a resource. The occurrence of the *granted* interaction means that a subscriber is allowed to use the resource. The occurrence of the *free* interaction means that a subscriber no longer intends to use the resource.

The following relations between interactions are informally identified:

– The occurrence of *granted* follows the occurrence of *request* (at the same location, and for a given resource identification);
– The occurrence of *free* follows the occurrence of *granted* (at the same location, and for a given resource identification); and,
– A resource is only granted to one subscriber at a time, i.e., the occurrence of *granted* cannot be followed by another occurrence of *granted*, before the occurrence of *free* (for a given resource identification).

The floor-control service is illustrated in *Figure 5-9*.

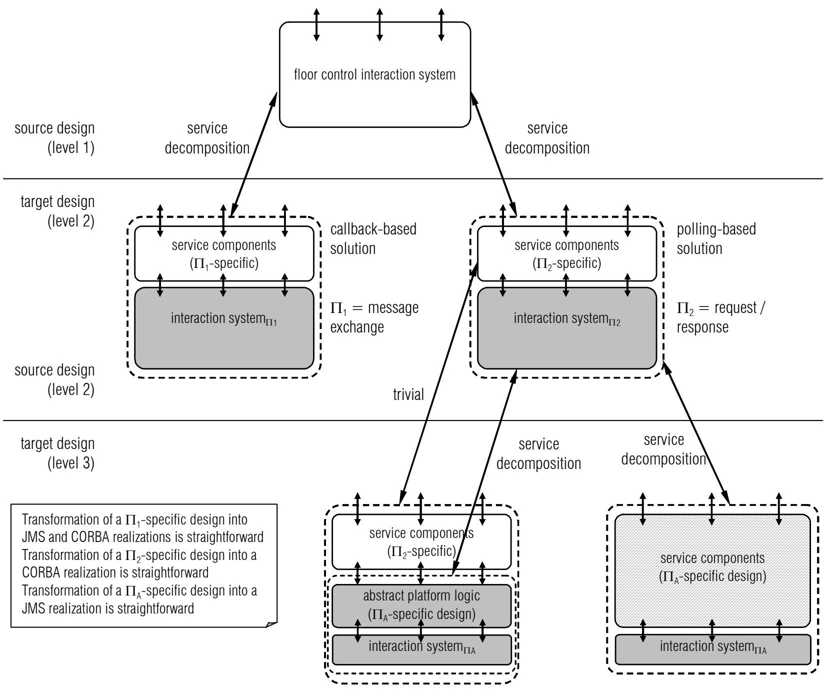
Figure 5-9 The service of the floor control interaction system


subscriber
floor control interaction system
subscribers
Interactions:
request (ResourceId resid);
granted (ResourceId resid);
free (ResourceId resid);

5.3.4 Example: service decomposition and platform-specific realization

Using the service of the floor-control interaction system as a starting point, we follow the design trajectory for two different abstract platforms: an abstract platform that supports message exchange and an abstract platform that supports the request/response pattern. We consider different design solutions for the floor-control service, illustrating that the service definition is to a large extent implementation-independent. For each platform-independent design obtained, we consider realizations in two target platforms: CORBA [73] and the Java Message Service (JMS) point-to-point domain [104]. *Figure 5-10* illustrates the design trajectories followed in our examples.

Figure 5-10 Example trajectories


floor control interaction system
source design (level 1)
service decomposition
service decomposition
target design (level 2)
service components (Π_1-specific)
interaction system$_{\Pi1}$
callback-based solution
Π_1 = message exchange
service components (Π_2-specific)
interaction system$_{\Pi2}$
polling-based solution
Π_2 = request / response
source design (level 2)
trivial
target design (level 3)
service decomposition
service decomposition
service components (Π_2-specific)
abstract platform logic (Π_A-specific design)
interaction system$_{\Pi A}$
service components (Π_A-specific design)
interaction system$_{\Pi A}$

Transformation of a Π_1-specific design into JMS and CORBA realizations is straightforward
Transformation of a Π_2-specific design into a CORBA realization is straightforward
Transformation of a Π_A-specific design into a JMS realization is straightforward

Callback-based solution with message exchange abstract platform

Abstract platform: message exchange. Initially, let us consider an abstract platform that supports message exchange (Π_1 in *Figure 5-10*). We identify two interactions that are related by the abstract platform:

- *send*, which results in the establishment of a *destination* and some *payload*. The occurrence of *send* means that the payload data should be delivered to a certain destination; and
- *receive*, which results in the establishment of some *payload*. The occurrence of *receive* means that the payload data has been delivered.

An occurrence of *receive* follows an occurrence of *send*. The interaction *receive* is executed at the location specified by the information attribute *destination* of *send*. The attribute *payload* represents the information to be sent. The value of the attribute *payload* for an occurrence of *receive* is the value of the attribute *payload* for the related occurrence of *send*.

Platform-independent design. The message exchange abstract platform is used in our *callback-based solution* to exchange messages between subscriber service components and the controller service component. The structure of the platform-independent design is depicted in *Figure 5-11*.

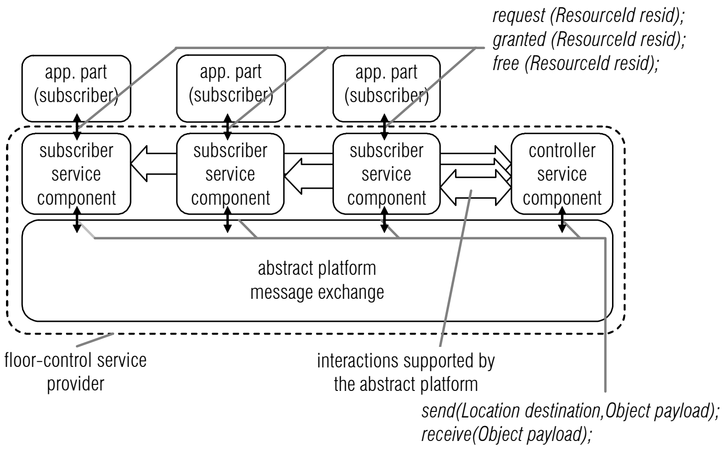

request (ResourceId resid);
granted (ResourceId resid);
free (ResourceId resid);
app. part (subscriber)
app. part (subscriber)
app. part (subscriber)
subscriber service component
subscriber service component
subscriber service component
controller service component
abstract platform message exchange
floor-control service provider
interactions supported by the abstract platform
send(Location destination,Object payload);
receive(Object payload);

Figure 5-11 Structure of the callback-based floor-control service provider

The controller service component centralizes the control of the access to the resources. When a subscriber requests for access to a resource, by executing the interaction *request*, the subscriber service component sends a request message to the controller with the identification of the resource. This is done in interaction with the abstract platform through the *send* interaction, which is followed by the occurrence of the *receive* interaction at the boundary between the controller service component and the abstract platform. Eventually, when the resource is to be granted to the subscriber, the controller sends a *grant message* to the subscriber service component. When the subscriber wants to release the resource, a *free* interaction is executed, resulting in the sending of a *free message* to the controller. A successful execution of a request for a resource is illustrated in *Figure 5-12*.

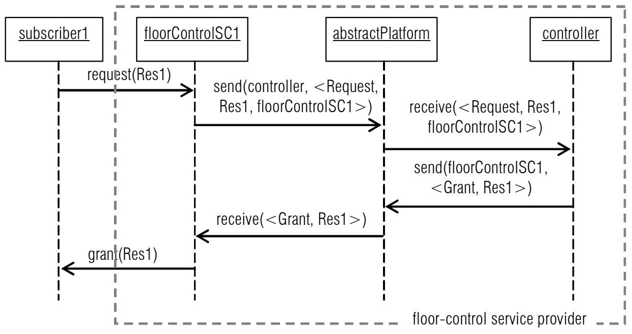
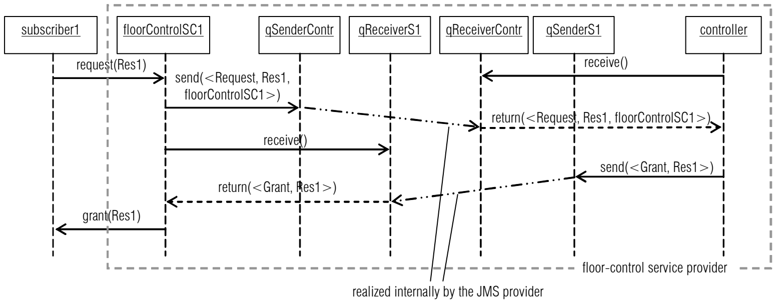

subscriber1
floorControlSC1
abstractPlatform
controller
request(Res1)
send(controller, <Request, Res1, floorControlSC1>)
receive(<Request, Res1, floorControlSC1>)
send(floorControlSC1, <Grant, Res1>)
receive(<Grant, Res1>)
grant(Res1)
floor-control service provider

Figure 5-12 A resource is requested and granted (Platform-independent design)

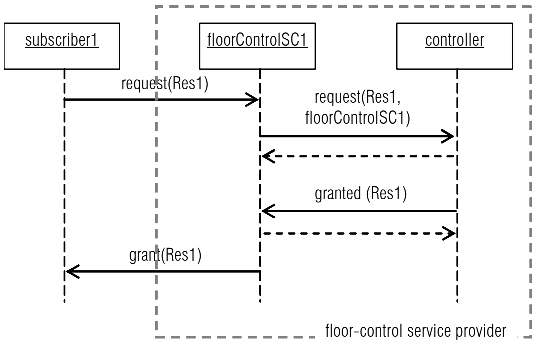
Realization. A realization of the platform-independent design in the JMS platform is straightforward. The service provided by JMS corresponds directly to the service provided by the defined abstract platform. A successful execution of a request for a resource in our realization in JMS is illustrated in *Figure 5-13*. In the JMS platform, the destination of a message is addressed by a queue identifier. In this solution, there is a queue for messages destined to the controller and a queue for messages destined to each subscriber. The addressing of the destination for a message is done through selection of a queue, and the instantiation of a message producer for the queue (*qSenderContr* for the queue directed to *controller* and *qSenderS1* for the queue directed to *subscriber1*).

Figure 5-13 A resource is requested and granted (JMS-specific realization)



The realization in the CORBA platform can be obtained through a simple transformation: message exchange is realized through an operation invocation with no return parameters. A successful execution of a request for a resource in our realization in the CORBA platform is illustrated in *Figure 5-14*.

Figure 5-14 A resource is requested and granted (CORBA-specific realization)



For the CORBA realization, we could have also considered the use of the CORBA Notification Service [85] in a similar way as we have used JMS to accomplish message exchange. This illustrates our observation that there are many possible ways to realize a platform-independent design even for a particular target platform.

Polling-based solution with request-response abstract platform

Abstract platform: request-response. Let us consider an abstract platform that supports the request-response pattern (Π_2 in *Figure 5-10*). We identify four interactions that are related to each other through the abstract platform:

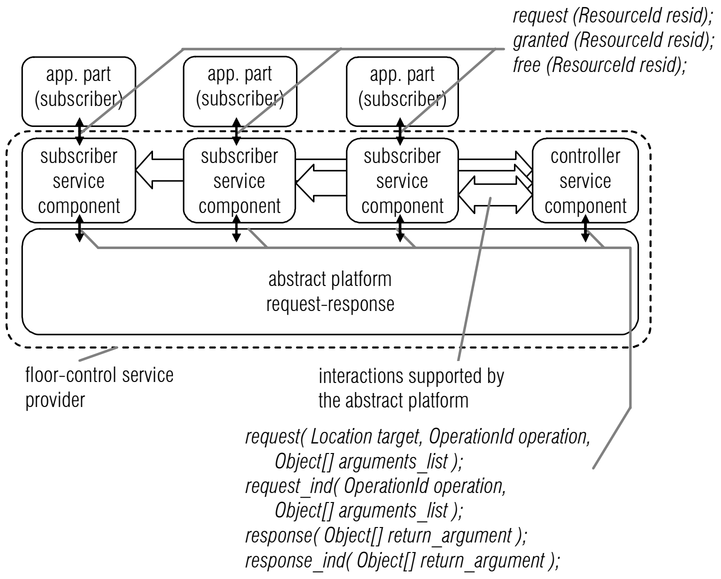
– *request*, with attributes: *target*, *operation* and *argument_list*. The attributes represent, respectively, the identifier of the target object, the identifier of the requested operation and the argument list for the request. The occurrence of request means that *operation* should be invoked on the *target* with a number of arguments (defined in *argument_list*);
– *request_ind*, with attributes: *operation* and *argument_list.* The occurrence of *request_ind* means that the target of the invocation is requested to execute the *operation* with a number of arguments (defined in *argument_list*);
– *response*, with attribute *return_parameters*, which represents the list of return parameters. The occurrence of *response* means that the target informs it has executed the operation, resulting in *return_parameters*; and,
– *response_ind*, with attribute *return_parameters*. The occurrence of *response_ind* informs the requester that the target has executed the operation.

The occurrence of *request_ind* follows the occurrence of *request*, the occurrence of *response* follows the occurrence of *request_ind*, and the occurrence of *response_ind* follows the occurrence of *response*.

This is a generalization of the service provided by request/response platforms. These platforms provide some software infrastructure to generate customized stubs that in conjunction with the middleware core provide specializations of the service as presented in this section.

Platform-independent design. The abstract platform is used in our *polling-based solution* to enable the subscriber service components to issue invocations to the controller. The structure of the platform-independent design is depicted in *Figure 5-15*.

Figure 5-15 Structure of the callback-based floor-control service provider


request (ResourceId resid);
granted (ResourceId resid);
free (ResourceId resid);
app. part (subscriber)
app. part (subscriber)
app. part (subscriber)
subscriber service component
subscriber service component
subscriber service component
controller service component
abstract platform request-response
floor-control service provider
interactions supported by the abstract platform
request(Location target, OperationId operation, Object[] arguments_list);
request_ind(OperationId operation, Object[] arguments_list);
response(Object[] return_argument);
response_ind(Object[] return_argument);

The subscriber service components poll the controller for a certain resource by invoking its operation *request_permission*, which returns the Boolean value *true* when the resource is available and *false* otherwise. When the subscriber wants to release the resource, the operation *free* of the controller's interface is invoked. A successful execution of a request for a resource is illustrated at the top of *Figure 5-16*.

Realization. A realization of the platform-independent design in terms of the CORBA platform is straightforward. The realization in terms of the JMS platform deserves more attention, since this platform does not support the request/response pattern directly.

We have applied the approach 1 to realization as presented in section 5.3.2: the abstract platform service specification is used as a starting point for a recursive application of service design. The diagram at the bottom of *Figure 5-16* illustrates a successful execution of a request for a resource, in a realization with the abstract platform realized in terms of the JMS platform. The occurrence of a *request* interaction results in the sending of a *request* message to the controller, containing the identification of the request, the name of the operation to be invoked, and the parameters for the operation. The identification of the request is used by the abstract platform service components to correlate request and response messages.

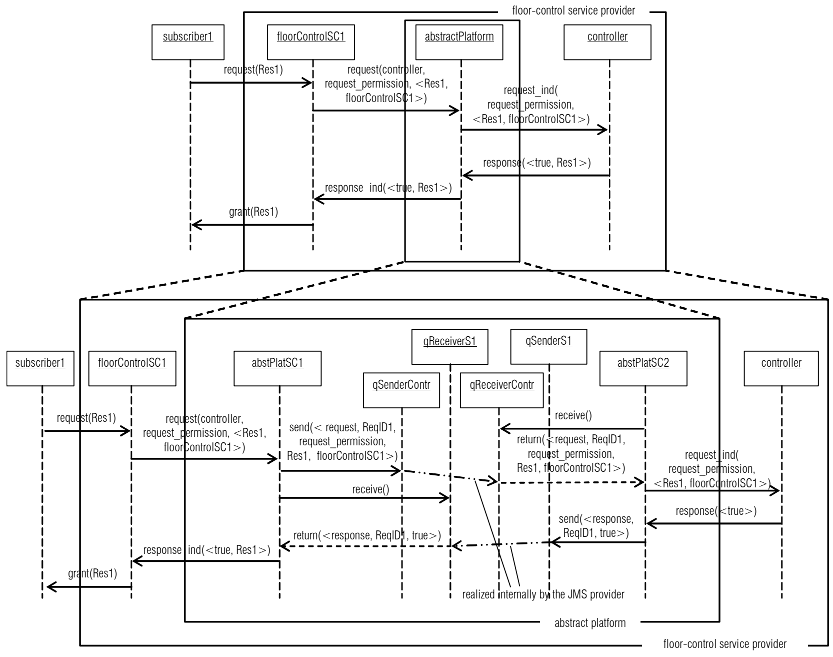

floor-control service provider
subscriber1
floorControlSC1
abstractPlatform
controller
request(Res1)
request(controller, request_permission, <Res1, floorControlSC1>)
request_ind(request_permission, <Res1, floorControlSC1>)
response(<true, Res1>)
response_ind(<true, Res1>)
grant(Res1)
qReceiverS1
qSenderS1
abstPlatSC1
qSenderContr
qReceiverContr
abstPlatSC2
send(< request, ReqID1, request_permission, Res1, floorControlSC1>)
receive()
return(<request, ReqID1, request_permission, Res1, floorControlSC1>)
request_ind(request_permission, <Res1, floorControlSC1>)
response(<true>)
send(<response, ReqID1, true>)
return(<response, ReqID1, true>)
response_ind(<true, Res1>)
realized internally by the JMS provider
abstract platform

Figure 5-16 A resource is requested and granted

A solution based on approach 2 (section 5.3.2) would also be possible, embedding functionality to correlate request and response in the floor control service components. In this case, the structure of the platform-independent design would not be directly recognizable in the platform-specific design.

Symmetric solutions

Both platform-independent solutions we have explored are asymmetric implementations of the floor-control service. Asymmetric solutions are characterized by separate controller and subscriber roles. The controller centralizes the coordination of access to shared resources, while subscribers must request the controller for access to a resource.

In addition to the asymmetric solutions we have presented, we identify a class of *symmetric* solutions to the floor-control service. In symmetric solutions, there is no controller, and all application parts have identical roles in the coordination. An example of a symmetric solution is based on token passing. In this solution, a list with the set of available resources circulates among the subscribers. Each subscriber service component examines the list with the set of identifiers of available resources, removes the identifier of the resource desired and forwards the list by invoking an operation on the interface of the following subscriber. When a subscriber wants to release a resource, the subscriber service component inserts the identifier of the resource to be released in the list. These solutions have been investigated [4] and can be approached in the same way as the asymmetric solutions presented here. They are not further discussed in this thesis.

Conclusions

Among the solutions discussed for the floor-control problem, the floor-control service is a stable abstraction, and shields the design of subscribers from the particular way in which the service is implemented. The floor-control service is neutral, both with respect to *commitments to particular design solutions* (callback-, polling-, or token-based) and with respect to *commitments to a particular middleware interaction pattern* (as provided by CORBA and JMS). It is irrelevant for the design of subscriber application parts whether the design of the floor-control solution is symmetric or asymmetric, callback-, polling-, or token-based, or whether the platform is CORBA or JMS.

For the design of the application interaction system itself, we have relied on abstract platform definitions. This allowed us to target CORBA and JMS from the same platform-independent design. Moreover, by using the abstract platform service specification as a starting point for a recursive application of service design (approach 1 for platform-specific realization), we have obtained software components that can be reused on top of different platforms.

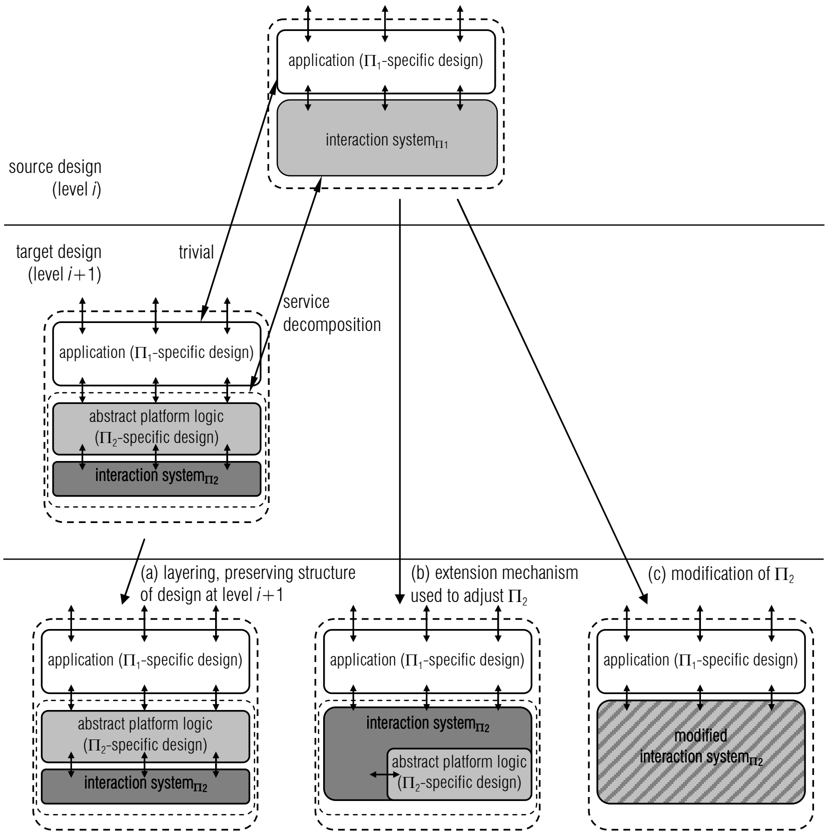
5.3.5 Realization with platform extension mechanisms

In section 5.3.2, we have discussed two approaches for a design step between two levels of platform-independence: an approach based on the adjustment of the target platform (approach 1), and an approach based on the adjustment of the application design (approach 2). An assumption underlying these two approaches is that both source and target designs are defined in the scope of the proposed design framework. The target design is not a realization, but a design which can be transformed into a realization in a straightforward manner. Having defined the design approaches in this way, it is possible to regard both approach 1 and 2 as service decomposition, and hence, make statements about the conformance of source and target designs.

If we relax this assumption, however, we are able to benefit from *platform extension mechanisms* to enable approach 1. The abstract platform logic is then incorporated into the platform at the realization level. Examples of extension mechanisms that can be used for this purpose are CORBA portable interceptors [73], composition filters [17] and specific mechanisms of aspect-oriented programming [37]. Since most of these mechanisms are platform-specific, the choice of mechanism depends on their availability in a particular platform. The availability of the source code of the platform also impacts the choice of extension mechanism, e.g., while CORBA interceptors can be added to a deployed CORBA ORB, many aspect-oriented programming mechanisms require the availability of source code, since new executable code should be compiled. The capabilities of the different extension mechanisms also vary, and must therefore be considered in the choice of a suitable extension mechanism.

The use of a platform extension mechanism to enable adjustment of a platform is depicted schematically in *Figure 5-17*(b). *Figure 5-17*(a) depicts a realization based on layering, which preserves the structure of a design at level $i+1$ which is obtained through the decomposition of the service of the abstract platform defined at level i. *Figure 5-17*(c) depicts the modification of a target platform. This latter technique may be necessary due to the lack of extension mechanisms or due to limited capability of these mechanisms. One can argue, however, that certain platform modifications result in a different platform, and hence, this technique may not satisfy requirements on the portability of platform-independent designs. Any modifications to platforms must, therefore, be considered in the light of these requirements.

Figure 5-17 Realization approaches


application (Π1-specific design)
interaction systemΠ1
source design (level i)
target design (level i+1)
trivial
service decomposition
application (Π1-specific design)
abstract platform logic (Π2-specific design)
interaction systemΠ2
(a) layering, preserving structure of design at level i+1
(b) extension mechanism used to adjust Π2
(c) modification of Π2
application (Π1-specific design)
abstract platform logic (Π2-specific design)
interaction systemΠ2
application (Π1-specific design)
interaction systemΠ2
abstract platform logic (Π2-specific design)
application (Π1-specific design)
modified interaction systemΠ2

Extension mechanisms are particularly useful to introduce required quality-of-service (QoS) mechanisms in the middleware, as is shown in [10, 117]. In [7], we have discussed the role of extension mechanisms for the realization of an abstract platform that support the dynamic reconfiguration of application parts.

5.4 Interaction refinement

The service concept can be used to abstract from the internal design of a system or system part at a particular point in the design process (as we have shown in the previous section). In order to abstract from internal structure, a service focuses on the interactions between application parts. These interactions should be described in a platform-independent way in platform-independent designs, it they are to be potentially realized in different middleware platforms.

In this section, we discuss a concept of abstract interactions that can be refined into interactions that can be realized by a target middleware platform. For that, abstract interactions must not commit to interaction mechanisms provided by a particular middleware platform. We define a number of design operations that can be applied to designs that use abstract interactions.

5.4.1 Interaction refinement in the design process

Before we discuss the concept of abstract interaction in further detail, let us consider the role of interaction refinement in a design process with platform-independent designs. For that, let us consider the design of a conferencing application. This application facilitates the interaction of users residing in different hosts. Let us suppose that, initially, the designer describes the application as a composition of conference participants, a conference manager and a conference service provider. In addition, we assume that the interfaces are described in terms of abstract interactions and interaction relations, which do not prescribe any particular interaction mechanism. The abstract platform at this level of abstraction supports the interactions between application parts and the conference service provider. *Figure 5-18* shows how a snapshot of this design (D_0) could be visualized. It distinguishes three conference participants and one conference manager.

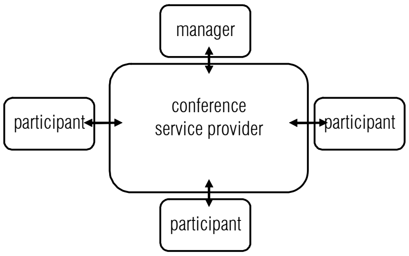


Figure 5-18 A initial design of the conference application (D_0)

We consider several alternative transformations of design D_0 according to the *interaction refinement* approach. The following alternatives show how different platform characteristics influence the refinement process:

1. We refine D_0 into a design D_1 that uses an abstract platform that supports *operation invocation between objects* and supports *multiple operation interfaces per object*. The conference service provider is not decomposed, and is directly implemented as a single object in the realization.
2. We refine D_1 into a design D_2, and as in design step (1) described above, we use an abstract platform that supports operation invocation. In this case, however, we add the platform-imposed constraint that *the abstract platform supports only a single operation interface per object*.
3. We refine D_0 into a design D_3, and as in design step (1) described above, we use an abstract platform that supports operation invocation between objects. The abstract platform supports a single operation interface per object. In this case, however, we add a platform-imposed constraint that *participants and managers are located in so-called 'thin clients', which cannot be used as targets for operation invocation*.
4. We refine D_0 into a design D_4 that uses an abstract platform that supports *asynchronous messaging* between objects. *The abstract platform supports multiple messaging queues*. The conference service provider is not further decomposed.

The abstract platform used in design D_2 facilitates the realization of this design in a CORBA platform (which offers only a single operation interface per CORBA object). The abstract platform used in design D_3 facilitates the realization of this design in a Web Services platform, e.g. with the conference service provider hosted in a J2EE platform, with 'thin clients' running in Mobile Information Device Profile (MIDP) devices [105]. The abstract platform used in D_4 facilitates the realization of this design using the Java Message Service (JMS) [104] or the CORBA Event Service.

Figure 5-19 depicts these alternative transformations steps and the resulting designs capture in models.

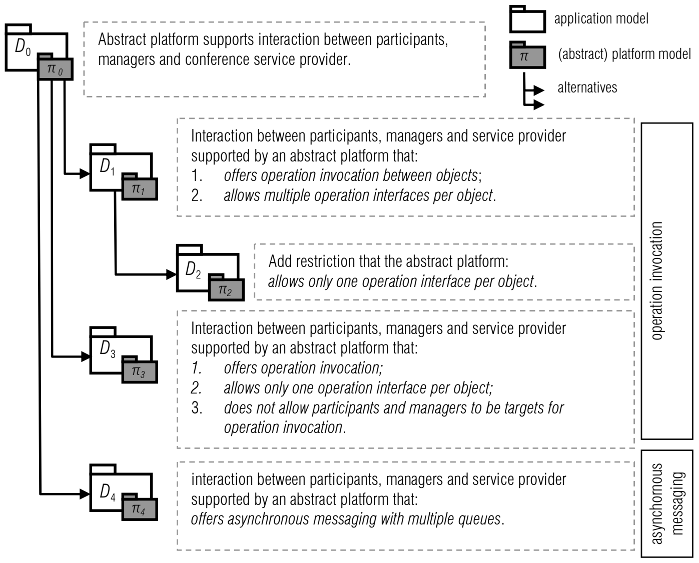

application model
(abstract) platform model
alternatives
D_0 π_0
Abstract platform supports interaction between participants, managers and conference service provider.
D_1 π_1
Interaction between participants, managers and service provider supported by an abstract platform that:
1. offers operation invocation between objects;
2. allows multiple operation interfaces per object.
D_2 π_2
Add restriction that the abstract platform:
allows only one operation interface per object.
D_3 π_3
Interaction between participants, managers and service provider supported by an abstract platform that:
1. offers operation invocation;
2. allows only one operation interface per object;
3. does not allow participants and managers to be targets for operation invocation.
D_4 π_4
interaction between participants, managers and service provider supported by an abstract platform that:
offers asynchronous messaging with multiple queues.
operation invocation
asynchronous messaging

Figure 5-19 Alternative design steps

5.4.2 Abstract interactions

The example in the previous section motivates requirements for design concepts that are not considered in current state of the art modelling languages. These concepts refer to both the behavioural and structural aspects of interaction between application parts.

With respect to behavioural aspects, an *interaction concept* is required that abstracts from the behaviour of a particular interaction mechanism. This is because at the highest level of platform-independence no interaction mechanism should be committed to. In the example both an operation invocation and an asynchronous messaging mechanism are considered as alternatives for the eventual realization of interactions described in D_0. An abstract interaction concept should abstract from these interaction mechanisms and allow the designer to use mechanisms available in middleware platforms for the realization of the design.

We adopt an interaction concept that captures:

- the identity of the interaction;
- the successful occurrence of the interaction;
- the information that is available to the interacting parties as a result of the interaction (the information attribute of the interaction) and the location at which this information is available (the location attribute of the interaction); and
- optionally, the direction in which the information flows.

Such an interaction concept has been proposed in [112]. An interaction is defined in as a unit of common activity of two or more functional entities, in which a value of information is established. This interaction concept abstracts from:

– roles that the interacting parties play in the interaction (e.g. initiator or responder);
– aspects of interaction mechanisms that have yet to be decide upon (e.g. whether an interaction corresponds to an operation invocation or a message being passed, whether queues are used to temporarily store messages, or whether an operation is blocking or non-blocking).

With respect to structural aspects, an *interaction point concept* is required that abstract from a particular interaction mechanism through which interaction takes place. We adopt an interaction point concept that captures:

– the identity of the interaction point,
– optionally, the interactions that may occur at the interaction point.

This concept is based on the interaction point concept that has been proposed in [112]. It is defined as the logical or physical location at which interactions occur, and is shared by two or more functional entities.

The interaction point concept abstracts from:

– any constraints on the interaction mechanisms that are available at the interaction point (e.g. only remote procedure calls can occur at this interaction point);
– the addressing scheme that is used to identify the interaction point (e.g. whether it is identified by a URI or a CORBA object reference).

We distinguish an *integrated* and a *partitioned perspective* for the interaction concept. In the integrated perspective, an interaction is seen as a shared action executed by all interacting parts in conjunction. For example, in the integrated perspective, an interaction *send_mail* does not identify parts with roles receiver and sender. In the partitioned perspective, each part that participates in an interaction can define its own constraint on the occurrence of that interaction. We call that constraint an *interaction contribution*. For example, the part fulfilling the role of receiver in a *send_mail* interaction, accepts any value for the information attribute message. The part fulfilling the role of sender constraints the value of this attribute so that it equals the message it wants to send. A counterpart to interaction contributions in the structural domain is the concept of *interaction point part*, which is an abstraction of part of the mechanism that supports interaction. Interaction point parts can be *bound* together forming an interaction point.

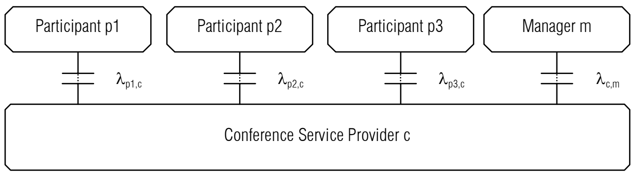
If an interaction occurs, its results are available to all its participants. If an interaction does not occur, no result is established. Hence, none of the participants can refer to any (intermediate) result. The possible results of an interaction are represented by information attributes. If an interaction occurs, the values of its information attributes represent the result of the interaction. An interaction can also be associated with a location attribute that represents the possible locations at which it can occur. If an interaction occurs, the value of its location attribute represents the location at which its results are available. This location identifies an interaction point.

Each interacting entity constrains the attributes established as result of an interaction: a party may offer a set of values, accept a set of values, or both. These constraints on values supply different ways of cooperation [91], namely, *value passing*, *value checking* and *value generation*. Value passing occurs when an interacting party offers a value and the other parties accept this value. Value checking occurs when all interacting parties offer the same value. In value generation, the interacting parties offer a range of acceptable values and the interaction happens if it is possible to establish a value that matches all requirements.

Application of design concepts to D_0

Figure 5-20 presents a snapshot of the structural aspects of D_0 in terms of the basic concepts described above. An entity is represented by a rectangle with cut-off corners that contains entity's name. An interaction point part is represented as a line that is connected to the owner of the interaction point part by another line. An interaction point is represented by a dashed line that connects the bound interaction point parts. Interaction points are annotated with their identifiers.

Figure 5-20 D_0 Snapshot



We identify the following (value passing) interactions:

- *sendmsg* interactions, which occur at the interaction points between participants and the conference service provider ($\lambda_{pn,c}$ in *Figure 5-20*). These interactions result in the establishment of a message to be sent (the information attribute i_{msg}). In this interaction, information flows from participants to the conference service provider;
- *receivemsg* interactions, which occur at the interaction points between participants and the conference service provider ($\lambda_{pn,c}$). These interactions result in the establishment of the message received. In the *receivemsg* interaction, information flows from the conference service provider to a participant;
- the *include* interaction, which occurs at the interaction point between the manager and the conference service provider ($\lambda_{c,m}$). This interaction establishes the identification of a participant (the information attribute

$i_{particip}$) that is to be included in the conference. In this interaction, information flows from the manager to the conference service provider;
- the *exclude* interaction, which occurs at the interaction point between the manager and the conference service provider ($\lambda_{c,m}$). This interaction establishes the identification of a participant (the information attribute i_{particip}) that is to be excluded from the conference. In this interaction, information flows from the manager to the conference service provider.

The following constraints apply to the interactions:
1. the occurrence of *receivemsg* interactions follows the occurrence of a *sendmsg* interaction; *receivemsg* interactions occur at the interaction points between participants currently included in the conference and the conference service provider;
2. the occurrence of *include* eventually leads to a participant being included in the conference (constraint 1 depends on the participants included in the conference), and;
3. the occurrence of *exclude* eventually leads to a participant being excluded from the conference (constraint 1 depends on the participants included in the conference).

In this thesis we do not present the precise way to represent constraints (we refer to [91] for more information about this aspect of design).

5.4.3 Design operations

A design that does not correspond directly to a realization in a selected target platform can be further transformed using the following design operations: (inter)action refinement, interaction point (and interaction point part) decomposition, interaction point (and interaction point part) merging, and entity merging. We present each of these operations in the following sub-sections, by motivating and illustrating them with the conference application and using the concepts presented in section 5.4.2.

Action refinement
If an action (i.e., either an interaction of internal action) cannot be supported by a construct from the realization platform, we must refine that action into multiple actions that can be directly supported by the realization platform.

An action cannot be refined into an arbitrary set of actions and constraints, because the refined behaviour must preserve the characteristics that the original behaviour prescribed. [89] explains how designs, constructed with an extension of the concepts from section 5.4.2, can be refined correctly. Basically, each action is refined into a group of *final actions* that correspond to the completion of that action and *inserted actions* that do not. Since the final actions correspond to the original action, they must together enforce the same constraints and deliver the same results as the original action.

Table 5-1 presents the rule for refining an action into multiple actions, making certain design decisions.

Table 5-1 Action refinement: definition

Input	Any action *a*.
Design decisions	Any (as long as constraints imposed by conformance relation are respected, see below).
Output	A group of actions that capture design decisions made. This group of actions consists of *final actions* that correspond to the completion of the original action *a* and *inserted actions* that do not. Final actions must together enforce the same constraints and deliver the same results as the original action *a* [89].

Action refinement example

In our conference example, none of the realization platforms support the abstract interaction concept directly through the supported interaction mechanisms. All the mechanisms in the considered platforms require additional design decisions, such as, defining the party responsible for initiating interaction. Therefore, the behaviour of a platform's interaction mechanisms is often defined at a level of abstraction at which multiple lower level actions are executed by the interacting parties. For example, asynchronous messaging mechanisms identify an interaction for a party to send a message and an interaction for a party to receive a message. A remote procedure invocation mechanism identifies an interaction for a client to issue a request, an interaction for a server to receive a request, an interaction for a service to respond to a request and an interaction for a client to receive the response to the request. *Table 5-2* illustrates how action refinement can be applied to refine an interaction into multiple interactions that form a remote invocation.

Table 5-2 Action refinement: transformation

Input	Any interaction *i* in which a value is passed from one party to another.
Design decisions	Operation invocation is used to realize interaction. The entity that passes value in the interaction initiates communication.
Output	The interaction *i* is refined into: a invocation_req interaction, a invocation_ind interaction, a invocation_rsp interaction and a invocation_cnf interaction. invocation_ind is a final interaction, all others are inserted interactions.

Interaction point decomposition

The consideration of platform characteristics in a design may require interaction points and interaction point parts to be decomposed into multiple interaction points and interaction point parts. This operation must be applied to an interaction point and its parts in a source design, if the interaction mechanisms that a realization platform provides cannot directly support the interaction point.

Table 5-3 presents the rule for interaction point decomposition. The entities and interaction points by which an interaction point is replaced in the refined design must connect the entities that correspond to the original entities of the abstract design. Otherwise, the refinement does not preserve the connectivity of the original design.

Table 5-3 Interaction point decomposition: definition

Input	Any interaction point λ (and interaction point parts associated with it).
Design decisions	Any (as long as constraints imposed by conformance relation are respected, see below).
Output	Entities that are connected through the original interaction point λ are connected through a configuration of interaction points and entities that replace λ.
Implications for behaviour domain	Interactions that occur at interaction point λ should occur at locations introduced by interaction points or entities that replace λ.

Interaction point decomposition and action refinement are often coupled, because, if an interaction point is refined, interactions that occurred at that interaction point must be refined into actions that can be assigned to the refinement of that interaction point.

Interaction point decomposition example

We obtain design D_1 from D_0 in two steps. *Table 5-4* shows the transformation used in the first step, in which the interaction points from D_0 are decomposed into multiple entities.

Table 5-4 Interaction point decomposition: transformation

Input	Any interaction point λ (and interaction point parts associated with it) between two entities e_1 and e_2.
Design decisions	Operation invocation is used.
Output	An entity e_π that supports operation invocation is introduced. This entity is connected to e_1 through an interaction point $\lambda_{\pi 1}$ and connected to e_2 through a $\lambda_{\pi 2}$.
Implications for behaviour domain	(Inter)actions that replace original interactions that occur at interaction point λ should occur at $\lambda_{\pi 1}$ or $\lambda_{\pi 2}$ or e_π.

Figure 5-18 illustrates this decomposition step graphically.

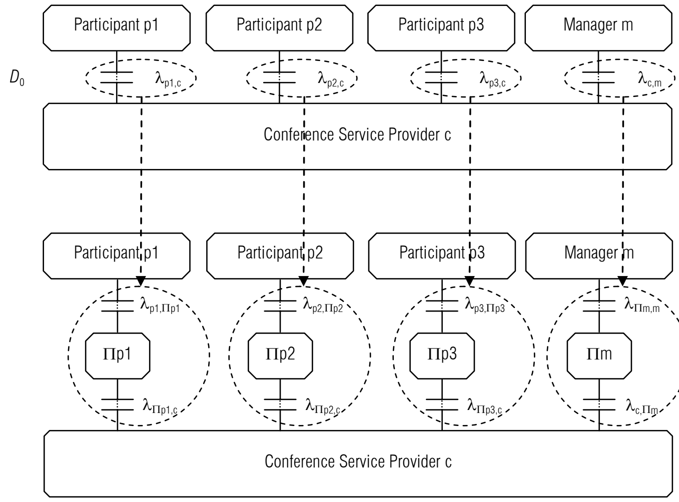


Figure 5-21 Action refinement and interaction point decomposition applied to D_0

The interactions that occurred at the original interaction point are refined according to the rule from *Table 5-2*. The *sendmsg* interactions which occur at interaction points $\lambda_{pn,c}$ are refined into:

– an *invocation_req* interaction, which occurs at interaction point $\lambda_{pn,\Pi pn}$ between a participant and an entity that is part of the abstract platform (see *Figure 5-21*). This interaction results in the establishment of the name of an operation to be invoked, arguments for the invocation, and an identifier for the invocation i_{id}. This identifier is unique in the context of the interaction point and is used to distinguish between multiple simultaneous invocations[11]. In this refinement, the name of the operation is *sendmsg* (not to be confused with the *sendmsg* interaction in D_0) and the argument is the value of information attribute i_{arg}. In our case this argument will carry a more concrete representation of the message that is sent.

– an *invocation_ind* interaction, which follows the occurrence of *invocation_req*. The *invocation_ind* interaction occurs at interaction point $\lambda_{\Pi pn,c}$ between an entity that is part of the abstract platform and the conference service provider (see *Figure 5-21*). The results of this interaction are the same as the results of the *invocation_req* interaction;

– an *invocation_rsp* interaction, which occurs at the same interaction point at which the *invocation_ind* interaction occurs. Since the *sendmsg* interac-

[11] This identifier is either implicit or explicit in realization platforms. For example, a CORBA client using the Dynamic Invocation Interface (DII) manipulates the identifier of a request explicitly. In contrast, for a client using compiled stubs the identifier of a request is implicit and corresponds to the thread in which the local stub method is invoked.

tion only consists of an information flow from a participant to the conference service provider, the response does not have to carry any information;
- an *invocation_cnf* interaction, which occurs at the same interaction point at which the *invocation_req* interaction occurs. This interaction follows the occurrence of the *invocation_rsp* interaction.

The *include* and *exclude* interactions are refined in a similar way. The *receivemsg* operation differs in that it is targeted at participants. For the sake of conciseness, we omit a detailed discussion of this refinement.

The final action for the *sendmsg* interaction from D_0 is *invocation_ind* with a value of *sendmsg* for i_{op}. Similarly, *invocation_ind* with a value of *receivemsg* for i_{op} is a final action for the *receivemsg* interaction from D_0. After abstracting from inserted actions *invocation_req*, *invocation_rsp* and *invocation_cnf*, the final actions enforce the same constraints as the actions for which they are final actions. The constraint that *receivemsg* is caused by *sendmsg* (in D_0) must also be enforced by the final actions for *receivemsg* and *sendmsg*.

The targets of operation invocation are implied by interaction points in which an *invocation_req* occur. For example, if an *invocation_req* occurs at interaction point $\lambda_{p1,\Pi p1}$, the invocation is targeted at the conference service provider. We can further transform this design by generalizing the behaviour of the entities that make up the abstract platform so that they support operation invocations between two arbitrary entities. This results in a better matching between this behaviour and the behaviour of realization platforms (such as, CORBA, Web Services, Java RMI). This generalization is accomplished by adding an information attribute (i_{dst}) to *invocation_req*, which identifies the interaction point at which a corresponding *invocation_ind* should occur. This attribute is defined by the entity that initiates an invocation.

Entity merging

The consideration of platform characteristics in a design may require entities to be merged into a single entity. This operation must be applied, if a realization platform supports multiple entities in a design as a single entity. *Table 5-5* presents the rule for entity merging. The resulting entity has all the interaction points that the original entities had. Similarly, the resulting entity carries all the behaviours of the original entities.

Table 5-5 Entity merging: definition

Input	Any set of entities e_i.
Design decisions	None.
Output	A merged entity *e* replaces the original entities e_i.
Implications for behaviour domain	Merged entity carries behaviour of entities e_i.

Figure 5-22 shows the application of entity merging in our example. Entities Π_{p1}, Π_{p2}, Π_{p3} and Π_{p4} are merged into an entity Π_1'. Entity merging does not affect the behaviour domain. The behaviour of the original entities is carried by the merged entity.

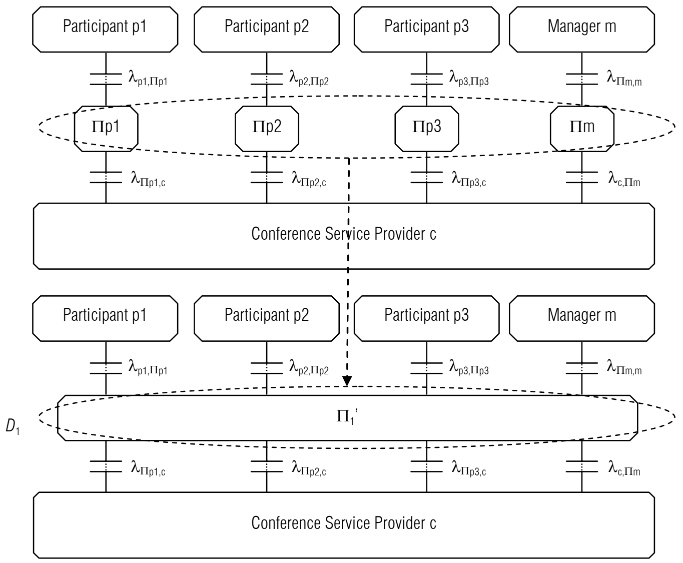


Figure 5-22 Entity merging to obtain D_1

Interaction point merging

The consideration of platform characteristics to a design may require interaction points to be merged into a single interaction point. This operation must be applied to some interaction point parts and their interaction points, if a realization platform imposes constraints on the number of interaction points that can be attached to an entity and the design violates these constraints. Merging of interaction points may require the interactions that occur at these interaction points to be refined, because interactions with the same name could originally be distinguished by the interaction point names. However, if the interaction points are merged, they can not be distinguished anymore. *Table 5-6* presents the rule for interaction point merging.

Table 5-6 Interaction point merging: definition

Input	Any set of interaction points λ_i between the same set of entities.
Design decisions	None.
Output	A interaction point λ replaces the interaction points λ_i.
Implications for behaviour domain	Behaviour preserves distinction between interactions. For example, information attributes can be used to distinguish interactions that occur at different original interaction points λ_i.

Interaction point merging example

We use interaction point merging to obtain D_2 from D_1. In platform Π_2, an entity is not allowed to have more than one interaction point part through which it plays the responding role in invocations. Therefore, multiple interaction point parts through which an entity plays a responding role must be merged into a single interaction point part (the corresponding interaction points are also merged). This step is depicted in *Figure 5-23*.

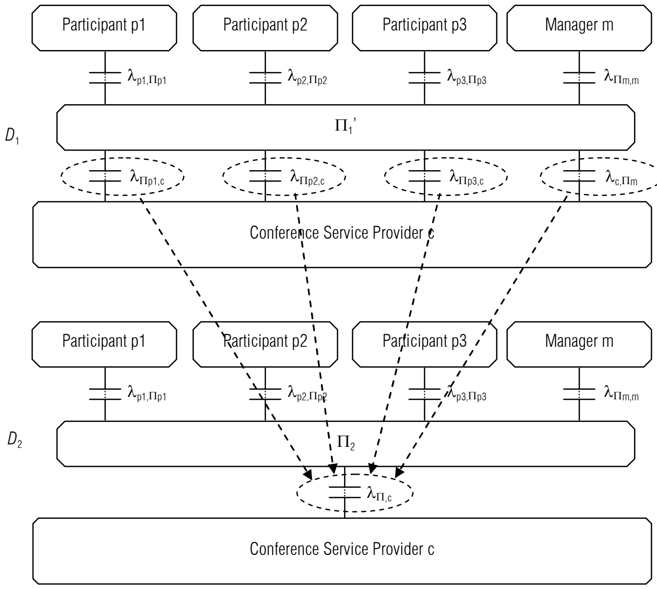


Figure 5-23 Interaction point and entity merging applied to D_1, resulting in D_2

The application of the interaction point merging operation consists of replacing interaction points $\lambda_{\Pi p1,c}$, $\lambda_{\Pi p2,c}$, $\lambda_{\Pi p3,c}$, and $\lambda_{c,\Pi m}$ by $\lambda_{\Pi,c}$ and should be reflected in the behaviour of entity Π_1' by replacing the interaction points being merged by $\lambda_{\Pi,c}$. In addition, *invocation_req* interactions that occur at interaction points $\lambda_{\Pi p1,c}$, $\lambda_{\Pi p2,c}$, $\lambda_{\Pi p3,c}$, and $\lambda_{c,\Pi m}$ (in D_1) are replaced by interactions at interaction point $\lambda_{\Pi,c}$ which have an additional information attribute i_{dst} that can have the values $\lambda_{p1,\Pi p1}$, $\lambda_{p2,\Pi p2}$, $\lambda_{p3,\Pi p3}$, and $\lambda_{\Pi m,m}$. respectively. This ensures that the interactions can still be distinguished as belonging to different original interaction points. For example, an *invocation_req* interaction that originally occurred at interaction point $\lambda_{\Pi p1,c}$ is replaced by an *invocation_req* interaction that occurs at interaction point $\lambda_{\Pi,c}$ and has the value $\lambda_{p1,\Pi p1}$ for i_{dst}. We say that in this way the topology of the original structure is preserved.

Realization of abstract platforms
By applying the design operations we have presented, a designer gradually refines a design into a design whose implementation onto a realization platform is straightforward. For example, the implementation of platform D_2 on a CORBA platform is straightforward, because we can apply the following transformation: each abstract platform entity from D_2 is implemented as a remote procedure invocation mechanism that is supported by CORBA; each interaction point is implemented as a CORBA operation interface on the client or on the server side, as it is specified in CORBA IDL; and each interaction is implemented as an interaction in the remote procedure invocation mechanism (invocation request, indication, response or confirmation).

5.4.4 The example revisited

In the previous section, we have discussed how the design operations can be applied to obtain designs D_1 and D_2. In this section, we show how designs D_3 and D_4 can be obtained from the same platform-independent design D_0.

For D_3, we use an abstract platform that supports operation invocations between objects to realize the interactions between participants, managers and the conference service provider. In this design *participants and managers are located in so-called 'thin clients', which cannot be used as targets for operation invocation*.

The refinements of interactions *sendmsg*, *include* and *exclude* are identical to the refinement we have presented earlier for D_2. The refinement of *receivemsg* differs significantly, since this interaction is realized through a polling scheme. The *receivemsg* interaction is refined into the following interactions:

- an *invocation_req* interaction, which occurs at interaction point $\lambda_{pn,\Pi pn}$ between a participant and an entity that represents the abstract platform. This interaction results in the establishment of the name of an operation to be invoked, in this case *receivemsg_poll*, and an identifier for the invocation, with the same role as the identifier used in the interaction point decomposition example shown in *Figure 5-21*;
- an *invocation_ind* interaction, which follows the occurrence of *invocation_req*. The *invocation_ind* interaction occurs at interaction point $\lambda_{\Pi pn,c}$ between an entity that represents the abstract platform and the conference service provider;

- An *invocation_resp* interaction, which occurs at the same interaction point at which the *invocation_ind* interaction occurs. The information attribute consists of a Boolean value ($i_{isavailable}$), which indicates whether a message is available, and the message (i_{arg}), if available;
- An *invocation_cnf* interaction, which occurs at the same interaction point at which the *invocation_req* interaction occur. This interaction follows the occurrence of the *invocation_resp* interaction.

A recursion in the refined behaviour is necessary, when the value of the $i_{\text{isavailable}}$ information attribute of *invocation_cnf* is false. The final action that corresponds to the original interaction is *invocation_cnf* with $i_{\text{isavailable}}$ equals true. Similarly to the case of design D_2, we can further transform this design by generalizing the behaviour of the entities representing the abstract platform so that they support operation invocations between two arbitrary entities.

For D_4, we use an abstract platform that supports *asynchronous messaging* between objects. *The abstract platform supports multiple messaging queues*. The *sendmsg* interaction is refined into the following interactions:

- a *data_req* interaction, which occurs at interaction point $\lambda_{pn,\Pi pn}$ between a participant and an entity that represents the abstract platform. This interaction results in the establishment of the message to be sent;
- a *data_ind* interaction, which follows the occurrence of *data_req*. The *data_ind* interaction occurs at interaction point $\lambda_{\Pi pn,c}$ between an entity that represents the abstract platform and the conference service provider.

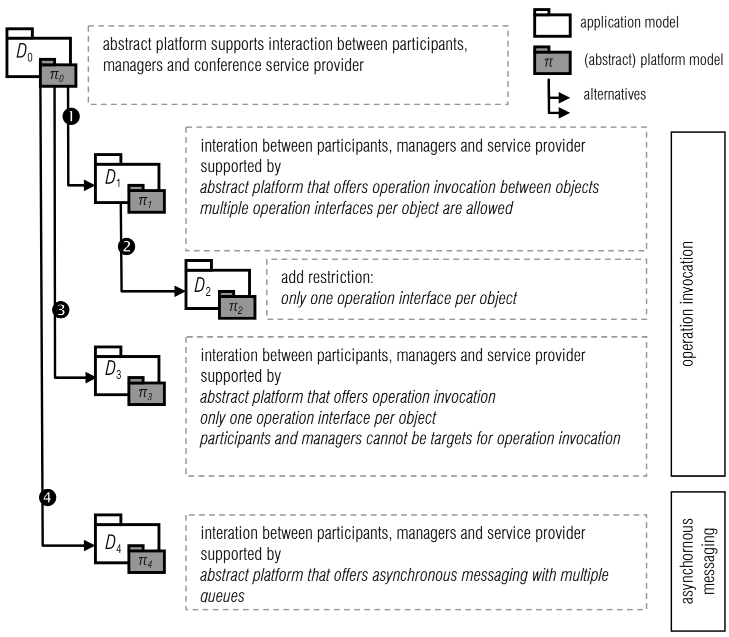
Similar refinements apply to the other interactions, with the exception of *receivemsg*, in which case the *data_req* is directed from the conference service provider to the abstract platform and the *data_ind* is directed from the abstract platform to a conference participant. Each pair of participant and service provider shares a message queue.

The *data_ind* interaction is the final interaction in the refinements. Depending on the constraints on the original interaction, it may be necessary to insert additional interactions to preserve the constraints in the source design. For example, if a participant performs an action that follows the occurrence of the *sendmsg* interaction, it is necessary to insert interactions in the target design to inform the participant that *data_ind* has occurred. This can actually be seen in the refinement framework as a refinement of the causality relation between *sendmsg* and the actions that depend on its occurrence [89].

We summarize the operations we have shown in *Figure 5-24*:

- the transformation marked by ❶ consists of interaction refinement (with a request/response pattern), generalization and entity merging;
- the transformation marked by ❷ consists of interaction point merging;

- the transformation marked by ❸ consists of interaction refinement (with a polling scheme), and generalization;
- the transformation marked by ❹ consists of interaction refinement (asynchronous messaging).



application model
(abstract) platform model
alternatives
D_0 π_0
abstract platform supports interaction between participants, managers and conference service provider
D_1 π_1
interation between participants, managers and service provider supported by
abstract platform that offers operation invocation between objects
multiple operation interfaces per object are allowed
D_2 π_2
add restriction:
only one operation interface per object
D_3 π_3
interation between participants, managers and service provider supported by
abstract platform that offers operation invocation
only one operation interface per object
participants and managers cannot be targets for operation invocation
D_4 π_4
interation between participants, managers and service provider supported by
abstract platform that offers asynchronous messaging with multiple queues
operation invocation
asynchornous messaging

Figure 5-24 Design operations and the designs

5.4.5 Remaining issues

In this section, we discuss some issues in the use of the design concepts proposed in this section.

Describing failure

In our approach, an interaction represents the successful completion of a shared activity. When the activity being modelled fails to complete, we say that the abstract interaction does not occur. If it is necessary to represent the failure of an activity explicitly, the failure should be modelled as an interaction, which can only occur if the interaction that models the successful completion of the activity does not occur.

A consequence of this modelling choice is that failure is perceived by all interacting entities. Therefore, it is not possible to model partial failures of a shared activity in this way. If it is necessary to model partial failure explicitly, the designer must model the shared activity at a lower level of abstraction, e.g., by modelling an entity between interacting entities and describing partial failure through the behaviour of this entity.

Value generation
As discussed in section 5.4.2, the notion of interaction we adopt can be used to model value generation. Value generation can be used to describe complex shared activities at a high-level of abstraction. For example, it is possible to model the negotiation of quality-of-service contracts between parties with their own requirements using a single interaction. However, value generation should not be used indiscriminately, since it may require sophisticated mechanisms for its reliable realization when distribution must be considered.

Dynamic configuration
The design concepts we have described in this chapter represent the behaviour of the system given a certain system configuration of entities and interaction points, i.e., ignoring the actions necessary to modify the system structure during execution. [31] describes design concepts that can be used to describe some of these actions, like the dynamic creation and destruction of entities and interaction points. The application of the interaction refinement operations presented in this chapter when considering these dynamic modifications in the system configuration still remains to be investigated.

Information value types
Our framework focuses on the behavioural aspects of interaction. We have not explored issues related to the exchange of information value types. Nevertheless, we acknowledge that these issues are an important aspect of model-driven design. Techniques used traditionally in the modelling of the information viewpoint in RM-ODP can be useful, as well as abstract data types and metamodelling frameworks and languages to describe abstract syntax (and mappings to concrete syntaxes).

5.5 Relation to RM-ODP

The ISO/ITU-T Reference Model for Open Distributed Processing (RM-ODP) [56] provides a specification framework for distributed systems development based on the concept of *viewpoints*. For each viewpoint, concepts and structuring rules are provided, defining a conceptual framework for specifications from that viewpoint. The use of different viewpoints in the design of complex systems is an accepted technique to achieve separation of concerns. This also has been reflected in standards such as, e.g., IEEE 1471 [49].

The RM-ODP computational and engineering viewpoints are relevant to the purpose of our work since they focus on application and infrastructure concerns, respectively. In this section, we argue that the separation of application and infrastructure in RM-ODP should be interpreted in the same way as the separation between applications and abstract platforms in our approach. In light of this interpretation to the separation of concerns proposed in RM-ODP, we discuss how our design framework compares to RM-ODP.

5.5.1 Concepts in the computational viewpoint

The computational viewpoint is concerned with the decomposition of a distributed application into a set of interacting objects, abstracting from the supporting distribution *infrastructure*. In contrast, the engineering viewpoint focuses on the infrastructure required to support distributed applications. It is concerned with properties and mechanisms required to overcome problems related to distribution (e.g., remoteness, partial failures, heterogeneity) and to exploit distribution capabilities (e.g., to achieve performance and dependability), but that are abstracted from in computational viewpoint specifications.

The RM-ODP relies on the concept of (distribution) transparency, which is defined as the property of hiding from a particular user (or developer) the potential behaviour of some parts of a system [56]. In the context of the computational and engineering viewpoints, transparency is used to hide mechanisms that deal with some aspect of distribution. An example of distribution transparency is replication transparency, which hides the possible replication of an object at several locations in a distributed system. In the computational viewpoint, a single computational object would be represented, while this computational object may possibly correspond to several replica objects in the engineering viewpoint. The mechanisms necessary to ensure replica consistency and management are addressed in the engineering viewpoint, shielding the (computational viewpoint) designers from the burden of developing these mechanisms. Distribution transparency is selective in ODP; the Reference Model includes rules for selecting transparencies. Transparencies are constraints on the mapping from a computational specification to a specification that uses specific ODP functions and engineering structures to provide the required transparency.

In the computational viewpoint, applications consist of configurations of interacting *computational objects*. A computational object is a unit of distribution characterized by its behaviour. A computational object is encapsulated, i.e., any change in its state can only occur as a result of an internal action or as a result of an interaction with its environment. An object is said to have *interfaces*, each of which expose a subset of the interactions of that object. Interaction between objects is only possible if a *binding* can been established between interfaces of these objects. The computational viewpoint supports arbitrarily complex bindings, through the concept of *binding object*, which represents the binding itself as a computational object. The behaviour of a binding object determines the interaction semantics they support. As with any other object, binding objects can be qualified by quality of service assertions that constrain their behaviour. The computational model does not restrict the types of binding objects, allowing various possible communication structures between objects to be defined [59].

The concepts of entity, interaction and interaction point parts as described in this chapter can be used to describe snapshots of applications in the computational viewpoint. The concept of an entity corresponds to that of a computational object. The concept of interaction point part corresponds to the structural aspect of an interface, and the concept of interaction point corresponds to the structural aspect of a binding. Binding objects are considered interaction systems, whose behaviours can be described as services.

The most significant divergence in the design framework presented in this chapter and the RM-ODP is the notion of interaction. In the computational viewpoint, objects interact via special kinds of interactions, namely, operations, signals and flows. The notion of interaction in our framework corresponds to the more general notion of interaction in the basic modelling concepts of RM-ODP, without the restriction that one of the kinds of interaction should be chosen (as in the computational viewpoint). As we have discussed in section 5.4, this more general notion of interaction is necessary to obtain designs at a high-level of platform-independence, since it can be refined into the more specific types of interactions, such as operations and signals in the computational viewpoint.

Another significant diversion in the adopted framework and the RM-ODP refers to quality-of-service (QoS) constraints on interfaces. While these are defined for the RM-ODP, we have not explored them in this thesis. Nevertheless, the framework we have presented can be extended to accommodate these constraints, as has been shown in [89] with the use of timing and probability constraints for the relations between interactions. We do not explore this further in this thesis, but we acknowledge the importance of QoS constraints in the model-driven design trajectory (section 4.3.2 presents an example in which QoS constraints are required).

5.5.2 The RM-ODP notion of infrastructure

In [19], Blair and Stefani have equated the boundary between the computational and the engineering viewpoints to the distinction between application and infrastructure: "It is important to realize that the boundary between the two viewpoints is fluid, depending on the level of the virtual machine offered by the system's infrastructure. Some systems will provide a rich and abstract set of engineering objects whereas others will provide a more minimal set of objects leaving more responsibility to the applications developer." Specifications in the computational viewpoint are, according to this interpretation, influenced by the level of support provided by the infrastructure. By setting the level of support provided by the infrastructure, one can refer to computational concerns and engineering concerns.

Equating infrastructure to predefined middleware platforms would lead us to the conclusion that computational specifications are directly influenced by the level of support provided by a selected middleware platform. Computational specifications would therefore be, to some extent, platform-specific. In this case, the separation of computational and engineering concerns would be identical to the separation between application and middleware platform concerns. The reusability of a computational viewpoint specification would be restricted by its dependence on platform characteristics. Furthermore, from the perspective of application developers, the separation of computational and engineering concerns would be implied by the availability of a software infrastructure. Therefore, we conclude that the motivation for the separation of computational and engineering concerns is predominantly bottom-up.

Another interpretation for the infrastructure assumed by the computational viewpoint is that of an 'ideal infrastructure'. In this interpretation, the motivation for the separation of computational and engineering concerns is predominantly based on the needs of the developer to handle the complexity of application and infrastructure separately, regardless of the availability of a software infrastructure. The engineering viewpoint offers the possibility for a designer to engineer the infrastructure explicitly. While this interpretation is ideal from the perspective of separation of concerns for the application developer, it does not leverage the reuse of middleware platforms, which would significantly improve the efficiency of the development process.

Table 5-7 summarizes the implications of these contrasting interpretations of infrastructure. We conclude that both interpretations considered have limitations when applied in conjunction with our design approach, which inspired us to investigate an alternative.

Table 5-7 Interpretations of infrastructure compared

Interpretation (infrastructure equals to)	**Reuse of middleware**	**Separation of concerns**	**Platform-independence**
Available middleware platform	Yes	Based on target platform	Low
Required middleware platform (ideal from application point of view)	No explicit consideration	Defined by designer's needs; motivated by complexity in application design	High

5.5.3 RM-ODP infrastructure notion revisited

Committing to one of the previously discussed interpretations of infrastructure is undesirable for the adoption of computational viewpoint concepts for our design process. It may lead to models at a low level of platform-independence, or it may lead to models which cannot be realized on existing middleware platforms. We propose to equate the term infrastructure, as used in RM-ODP, to our notion of abstract platform. This approach can be beneficial for the development of distributed applications, so that a proper balance can be obtained between the following design goals:

- designers can use the separation of application and infrastructure concerns to cope with the complexity of distributed application design;
- middleware platforms can be reused to improve significantly the efficiency of distributed application development; and
- platform-independence can be obtained as a means to preserve investments in application development and withstand changes in technology.

A consequence of equating infrastructure to abstract platform is that computational viewpoint concepts can be applied recursively at different levels of platform-independence.

In the computational viewpoint, an abstract platform may be defined in terms of the bindings (and binding objects) supported, the transparencies supported, and the types of QoS constraints that may be applied to interfaces.

The use of binding objects in an abstract platform provides considerable flexibility to implementations of platform-independent designs, since it is possible to provide countless different implementations of a binding object.

One could argue that binding objects should be decomposed exclusively in the engineering viewpoint, since they are part of the infrastructure and should not be considered in the computational viewpoint. However, there are two main reasons to refute this argument. First, the definition of infrastructure (and hence abstract platform) varies during the design process. Second, the engineering viewpoint forces a designer to commit to a particular distribution in terms of nodes, capsules and clusters. Committing to a particular distribution may not be necessary at some point in the design process, in which case, it could unnecessarily constrain the choice of target platform or realization approach.

The use of transparencies and QoS constraints also provides flexibility in implementations, since there is considerable freedom in choosing mechanisms for obtaining a required transparency and satisfying QoS constraints. An example that reveals this flexibility is shown in *Figure 5-25*. In this example, a client and a server object interact through an operation interface. A replication transparency schema is used to specify constraints on the availability and performance of the server object. Two different mappings of the source model (a) are depicted below. In *Figure 5-25*(b), a realization is obtained by mapping the source model directly to a platform that supports replication transparency, namely, Fault Tolerant CORBA. The infrastructure depicted is provided with this platform [73]. In *Figure 5-25*(c), a realization is obtained by mapping the source model into a target model that explicitly addresses the replication of the server object. A replication object is introduced to execute the replication function, delegating requests to the different replicas. For simplicity, we consider stateless server objects, and therefore we can omit extra interfaces required for checkpointing. This step can be considered as service decomposition applied to the server object. A possible realization of this client server application in Web Services [120, 121] is depicted schematically in *Figure 5-25*(d).

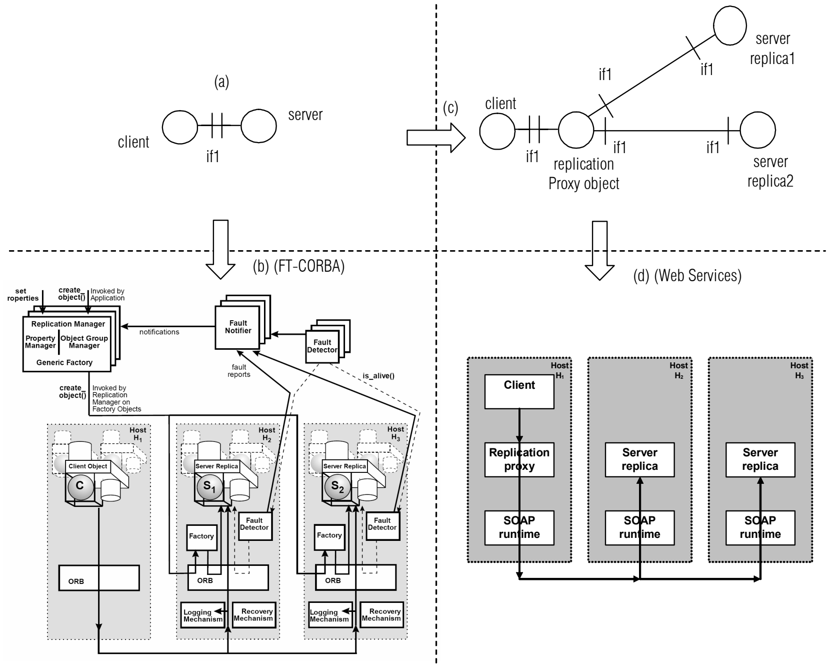


Figure 5-25 Example of flexibility in realization approach

5.6 Evaluation

This section presents an evaluation of the design framework, according to the quality criteria defined in chapter 3 of this thesis, namely: generality, stability, buildability and ease of use. We discuss how the design framework enables designers to produce abstract platform that satisfy quality requirements.

Generality

The design concepts we have adopted in our design framework can be used to define a number of interaction systems of varying generality. In this chapter, we have used the design concepts for the design of a conference interaction system, which can be considered fairly application-specific. At the same time, we have used the design concepts for abstract platforms that support request/response and asynchronous messaging patterns, which can be considered general-purpose. The generality of abstract platforms and interaction systems is, therefore, not guaranteed by the framework, but is enabled by the framework.

In addition, the notion of interaction we have adopted can accommodate a number of interaction mechanisms in the realization as we have shown in the examples presented in this chapter. Therefore, we can conclude that this notion of interaction satisfies the generality requirement.

Buildability

Not unlike generality, buildability is not directly guaranteed by the use of the framework. Designers can obtain designs which are more or less buildable in particular platforms. We have shown that buildability can be increased through application of service decomposition and interaction refinement, through which platform constraints can be progressively incorporated in designs. Platform-independent designs can be transformed by applying the design operations and realized into a number of different platforms with different characteristics. While we have shown realizations of designs in terms of middleware platforms, the concepts we have employed have also been used in protocol design serving as starting point for obtaining protocol implementations [16].

Stability

The concepts we have adopted in the design framework allow description of application designs and abstract platforms at high-level of abstraction. This allows changes in target platforms to be accommodated in the path to realization (e.g., through transformations that apply design operations with different design decisions), preserving the stability of designs at a high-level of abstraction.

Ease of use

The explicit definition of the service of interaction systems that make up an abstract platform is beneficial to transformation designers, which can use abstract platform service definitions as a starting point for transformation definition. Approach 1 to realization can be followed (section 5.3.2), in order to allow transformation designers to gain confidence in the correctness of transformation. Transformation designers also expect the abstract platform to be defined in such a way that does not unnecessarily constraining the freedom of implementation; we have shown in this chapter, that this framework can be used to define platform-independent designs so as to preserve freedom of implementation.

5.7 Related work and concluding remarks

Design transformations in which implementation constraints are incorporated have been proposed earlier, for example, in the LOTOSphere [22] project. Some of the design operations we have presented here have been inspired by the transformations described in [94]. These transformations have been developed to bridge the abstraction gap between formal languages and implementation environments, which is in some aspects similar to the gaps between platform-independent models and platform-specific models that have to be bridged by transformations in model-driven design. The main difference between the transformations in [94] and the design operations proposed here is that the former transformations do not consider middleware technologies as implementation environments (platforms) and therefore they cannot be directly applied to our situation.

Most efforts related to transformations in model-driven design and MDA focus on the languages, methods and tools for the specification of model transformation. These efforts are complementary to the work presented in this chapter, since the design operations we have defined can be used to derive model transformation specifications that could be implemented by tools. This chapter contributes to the understanding of the design operations that are applied by transformation in a model-driven design approach. Furthermore, we argue that suitable notions of conformance between source and target designs are necessary if we want to reach a mature model-driven design process. This chapter explores how these notions of conformance can be defined and enforced, both with service decomposition and interaction refinement.

With respect to service decomposition, the design framework implies an approach based on service definition and service design. While this suggests a top-down design trajectory it does not exclude the use of bottom-up knowledge. Bottom-up knowledge is what allows designers to re-use middleware infrastructures, by defining an abstract platform that can be realized in terms of these concrete middleware platforms, and to find appropriate service designs that implement the required service. This is similar to finding stable solution domains in a synthesis-based design method [108]. The use of approach 2 described in section 5.3.2 allows bottom-up rationale to justify partially ‘breaking’ the structure of a design in a design step. In addition, (bottom-up) extension mechanisms available in platforms can be used to provide suitable realizations of abstract platforms with software composition techniques that are outside the scope of the framework, such as, e.g., aspect-oriented programming [37].

We have shown that the interaction concept and interaction refinement design operations can be used to realize a platform-independent design in multiple realization platforms. This is possible because interaction can be modelled at a high level of abstraction.

We approach interaction refinement from the perspective of architectural design. Several authors approach interaction refinement from a pure formal perspective (e.g., [24, 25]). We believe that, in many cases, these approaches make simplifications at the cost of the usefulness of the formal model for pragmatic engineering purposes (as argued in [113]).

The use of a uniform set of concepts in different levels of models facilitates the establishment of conformance relations between the levels. While this applies to application design at different levels of platform-independence, other authors have shown that this set of concepts can also be applied successfully in describing business process [118], and defining the relations between business process and applications that support these processes [32]. In [30], an approach is shown that uses the set of basic design concepts to relate the RM-ODP Engineering and Computational viewpoints. That work could be used in a design approach that also encompasses business environments and business processes, which are outside the scope of this thesis.

Due to the large variety of target platform characteristics, and hence, the variety of abstract platform characteristics, we do not claim that the concepts of system, interaction system, service and the abstraction interaction are sufficient to define all possible abstract platforms. We anticipate that this set of concepts may have to be extended in the context of a modelling language with concepts that facilitate the structuring and maintenance of designs. For example, the notion of inheritance between different services can be added to allow for the reuse of service definitions. Patterns formed from basic design concepts can also be defined to form coarse grained building blocks that facilitate the definition of abstract platforms and application designs.

Chapter 6

Support for abstract platforms in MDA

In chapter 2 of this thesis, we have discussed in general how abstract platforms and modelling languages can be related. This resulted in two approaches for abstract platform definition: the language-level approach and the model-level approach. In this chapter we discuss how these approaches can be used to define abstract platforms with the Unified Modelling Language (UML) [81] and the Meta Object Facility (MOF) [77].

We have chosen the UML and the MOF for their relevance in the context of OMG's MDA [76] standardization. We assume that the use of widely adopted modelling languages and language definition architectures can promote the reusability of platform-independent models, abstract platform models and transformation specifications. However, we are aware that this argument is only significant if these standards provide proper support for our design approach. This motivates our investigation in this chapter.

This chapter is organised as follows: section 6.1 recapitulates the language-level and the model-level approaches to abstract platform definition; section 6.2 briefly introduces the UML and MOF; sections 6.3 and 6.4 focus on the how to support the language-level and model-level abstract platform definition approaches with UML and MOF; section 6.5 illustrates both approaches with an example; section 6.6 discusses the strengths and limitations of the UML for abstract platform definition; finally, section 6.7 presents some concluding remarks.

6.1 Abstract platform definition approaches

In chapter 2, we have defined the following general approaches to abstract platform definition: the language-level approach and the model-level approach.

In the language-level approach, the abstract platform designer defines styles and restrictions that are to be applied to elements of a particular modelling language. These styles and restrictions combined with the concepts underlying the modelling language allow one to unambiguously determine the abstract platform.

In the model-level abstract platform definition approach, the abstract platform designer defines a set of pre-defined design artefacts which are to be composed with the application by the application designer. Similarly to the case of the language-level abstract platform definition approach, the set of design concepts underlying the language is relevant to derive some abstract platform characteristics, since the modelling language is used to describe: (i) the application, (ii) any necessary pre-defined design artefacts, and (iii) the composition of application and pre-defined artefacts.

Since in both the language- and model-level abstract platform definition approaches there is some overlap between language characteristics and abstract platform characteristics, a modelling language can be evaluated based on its suitability to represent intended abstract platform characteristics.

In the sequel, we discuss how UML, its Profiles and MOF can be used to represent abstract platforms in both the language- and model-level abstract platform definition approaches. We assume the reader is acquainted with OMG standards, but when necessary, we introduce specific UML concepts. We conclude by discussing some limitations of UML with respect to describing abstract platforms at the various levels of platform-independence.

6.2 UML, Profiling and MOF

The UML has been developed initially as a methodology-independent technique for the modelling of the structure and the behaviour of object-oriented systems. It provides a large number of diagrams and notations, is supported widely by modelling tools, and is defined in OMG specifications [81, 84].

Since UML's inception, the language has been used for a number of other purposes, in part thanks to a language extension mechanism called profiling. More recently, the profiling mechanism has been incorporated in

OMG's general metamodelling framework, which includes the MOF. The MOF serves as an infrastructure for defining the *abstract syntax* of the UML and other OMG languages, resulting in a language definition architecture for the MDA. *Figure 6-1* shows a possible usage of this language definition architecture. It shows that the MOF (*metametamodel*) is used to define the UML *metamodel* and that the profiling mechanism is used to extend the UML with the EDOC profile. *Figure 6-1* also shows that the UML metamodel defines the abstract syntax of UML models.

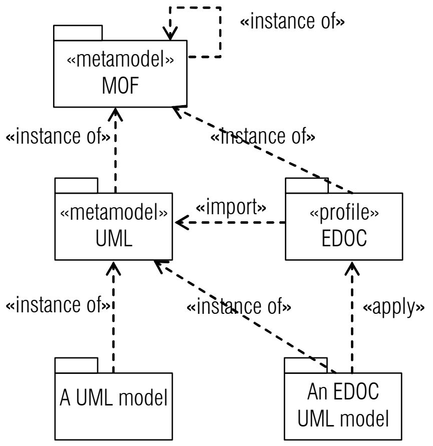


Figure 6-1 Example: usage of OMG's MDA language definition architecture

Interestingly enough, we use a notation for depicting the metamodels and the profile which is based on the profiling mechanism itself. This can be seen in *Figure 6-1*. We use *stereotypes*, depicted as labels within a pair of guillemets, such as «instance of», «profile», «metamodel» and «apply». The stereotypes provide specializations of UML concepts, in this example, the concepts of dependencies (depicted as open arrows with dashed lines) and packages (depicted with a "package" icon).

Metamodels are usually accompanied by natural language descriptions of concepts that correspond to elements of the metamodel, defining informally the semantics of the modelling elements. This approach has been adopted by the OMG in the UML specifications. More rigorous approaches define the semantics of modelling elements in terms of a mathematical or formal domain, such as the formal semantics of the Specification and Description Language (SDL) in [55], or in terms of explicit representations of domain conceptualizations, such as an ontology, as proposed in [46].

6.3 Language-level abstract platform definition

We start by considering how the MDA language definition architecture can be used to define abstract platforms in the language-level approach.

6.3.1 UML constructs for modelling application parts and their interaction

In order to adopt the language-level approach for abstract platform definition in UML, we must first consider the constructs provided by UML for modelling application parts and their interaction.

In the UML 2.0 metamodel, the constructs for interaction are *operations* and *receptions,* which are offered by *BehavioredClassifiers*. *Operations* represent the capability of a classifier to receive and to respond to requests. Requests are sent when objects or components (instances of classifiers) execute *CallOperationActions*. *Receptions* represent the capability of a classifier to receive *Signal* instances, which are sent asynchronously by other objects or components when these execute *SendSignalActions* and *BroadcastSignalActions*.

Given these constructs, we can conclude that the language-level approach can in principle be used in UML for abstract platforms based on request-response invocations and point-to-point message passing.

As we have discussed in section 6.2, UML is currently regarded as a general purpose language that is expected to be customized for a wide variety of domains, platforms and methods [83]. A certain degree of customization may be obtained in UML through semantic variation points and profiles. This choice in the definition of UML has two implications for language-level abstract platform definition. First, the UML specification ("plain" UML) is not definitive with respect to the abstract platform implied. Second, customization mechanisms must be applied in order to precisely define specific abstract platforms.

Semantic variation points provide an intentional degree of freedom for the interpretation of the UML's metamodel semantics. Some semantic variation points defined in the UML specification should be resolved for plain UML to be conclusive with respect to the abstract platform implied by the language. An example of such a semantic variation point is described in the UML 2.0 specification [81] (page 381): "*The means by which requests are transported to their target depend on the type of requesting action, the target, the properties of the communication medium, and numerous other factors. In some cases, this is instantaneous and completely reliable while in others it may involve transmission delays of variable duration, loss of requests, reordering, or duplication.*"

Without resolving this semantic variation point, a designer would be forced to assume worst-case interpretations, e.g., that the implied abstract platform provides an unreliable request/response mechanism. If this is undesirable, e.g., because the abstract platform should provide a reliable request/response mechanism, a designer should resolve the semantic variation point, by defining that requests and response signals are transported reliably. Semantic variation points may be partially resolved, i.e., only for the relevant aspects. For example, a designer may consider the reliability characteristics of requests relevant, but may consider the timing characteristics irrelevant. In this case, any interpretation of the timing characteristics of requests would be acceptable. One could resolve these semantic variation points by relating the UML metamodel with a formal semantics, or to a basic set of design concepts with a formal semantics. Examples of efforts towards a formal semantics for UML are [38, 62, 111].

6.3.2 Profiles and MOF

The specialization of UML for defining abstract platform characteristics can be made more manageable and clearly defined through the use of UML profiles. Profiles are language extensions consisting of metamodel elements that specialise elements of a reference metamodel. The specialized elements can be given specific semantics, in this way resolving semantic variation points. Furthermore, constraints expressed in the Object Constraint Language (OCL) [80] can be added to profiles to restrict the use of specific concepts or combinations of concepts. This use of profiling for language-level abstract platform definition is restricted to constraining or specialising the abstract platform implicitly defined by plain UML. In this approach, the referenced metamodel (UML 2.0's metamodel) in combination with the UML profile assumes the role of abstract platform model. This approach is illustrated in 6.5.1.

In case the relevant abstract platform characteristics cannot be represented by resolving semantic variation points through the definition of profiles, one should define new languages in terms of MOF metamodels. This approach is illustrated in chapter 7 of this thesis. The design concepts of languages defined in MOF are not constrained by UML, and can be arbitrarily defined through mappings from the metamodel elements to any suitable semantic domain. In this case, the metamodel (defined using the MOF) assumes the role of an abstract platform model.

UML Profiling is more suited to the abstract platforms that require concepts that can be represented as specialisations of UML concepts. MOF metamodelling is suitable in case the required concepts differ too much from the UML concepts, so that a new independent metamodel has to be defined. When used systematically, profiling has the advantage that UML tools can be used for model validation and verification, since the resulting models still comply with the UML rules and constraints. MOF metamodelling has a potential drawback that available validation and verification tools may be impossible to reuse, so that new tools may have to be built for the new metamodel.

6.4 Model-level abstract platform definition

In addition to changing the design concepts of plain UML in the language-level abstract platform definition approach, we can define the abstract platform at the model-level. The abstract platform is then modelled in UML and is composed with the application model. This can be accommodated in UML 2.0 by using model library packages [81] to define the abstract platform model. Model library packages are packages stereotyped with the standard *«modelLibrary»* stereotype. The abstract platform model library package can be imported by the application PIM. This is represented by creating a dependency between the package where the PIM is defined and the model library package where the abstract platform is defined.

An abstract platform can have an arbitrarily complex behaviour and structure, varying from a simple one-way message passing mechanism to a communication system that maintains transactional integrity and time order of messages. To make the design of complex abstract platforms manageable, we can use UML 2.0's composite structures to break up a complex design into smaller pieces. State machine and activity diagrams may be associated with encapsulated classifiers to define their behaviour.

Since the behaviour of the abstract platform is also described in UML, it is often necessary to combine the model-level and the language-level abstract platform definition approaches, e.g., by resolving semantic variation points that are relevant for the composition of the abstract platform (explicitly defined) and the platform-independent model of the application.

6.5 Example

In order to illustrate both approaches to abstract platform definition in UML, we specify the platform-independent model of a simple chat application. This application allows users residing in different hosts to exchange text messages.

Initially, the application is described in terms of an abstract platform that supports the interaction of objects through a conference interaction system. We call this abstract platform the *ConferenceAbstractPlatform*. In order to define the composition of the conference interaction system with the application, we use reliable exchange of asynchronous signals. For this purpose, we define an abstract platform that supports reliable signal exchange with the implicit approach, by defining a UML profile. Later, we consider two possible realizations of the *ConferenceAbstractPlatform*, one of them relying on an event-based platform we define at the model-level, and the other relying solely on the exchange of reliable signals. The relations between the different models are depicted in *Figure 6-2* (the *EventAbstract-Platform* is only necessary for the realization presented in section 6.5.4).

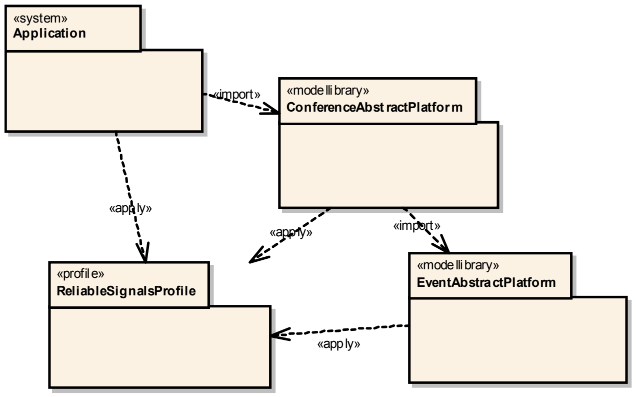


Figure 6-2 Relations between the application PIM and the abstract platforms defined with the implicit and explicit approaches

6.5.1 Reliable signal exchange

Figure 6-3 depicts the *ReliableSignalsProfile* that specializes the exchange of asynchronous messages in UML 2.0. A stereotype *«reliable»* is defined that can be applied to instances of *SendSignalAction* (defined in the package *IntermediateActions* of the UML 2.0 metamodel). Signals created by executing a *SendSignalAction* with this stereotype are exchanged reliably, in that they cannot be lost or duplicated. The *SendSignalAction* meta-class is the only meta-class specialized in the profile. It is not necessary to specialise the meta-classes *Signal* and *Reception*, since these represent respectively, the type of signal instances exchanged and the ability to receive signal instances. The semantics of these meta-classes are independent of the manner of transmitting signal instances.

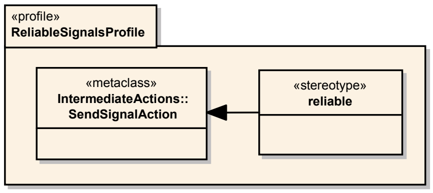


Figure 6-3 A UML profile specializing the exchange of asynchronous messages

6.5.2 The *ConferenceAbstractPlatform*

The *ConferenceBinding* component provides the *ConferenceInterface* and requires the *ParticipantInterface*. An application part that uses the *Conference-Binding* should provide the *ParticipantInterface*. The signals exchanged between application parts and the abstract platform are defined explicitly. A class diagram showing the *ConferenceAbstractPlatform*'s component, signals and interfaces is depicted in *Figure 6-4*.

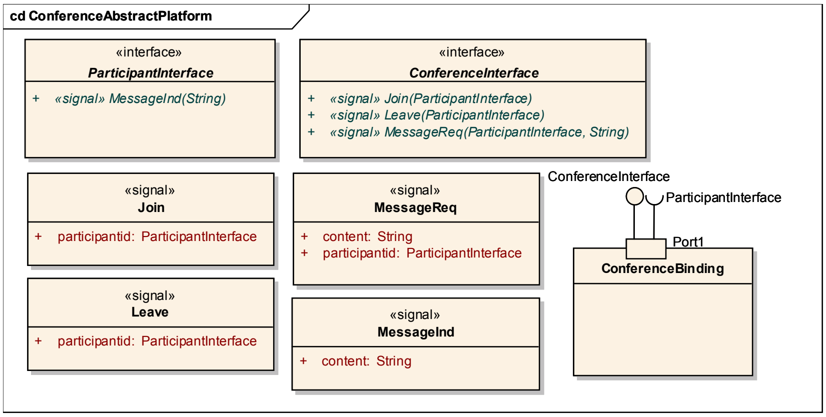

cd ConferenceAbstractPlatform
«interface»
ParticipantInterface
+ «signal» MessageInd(String)
«interface»
ConferenceInterface
+ «signal» Join(ParticipantInterface)
+ «signal» Leave(ParticipantInterface)
+ «signal» MessageReq(ParticipantInterface, String)
«signal»
Join
+ participantid: ParticipantInterface
«signal»
MessageReq
+ content: String
+ participantid: ParticipantInterface
«signal»
Leave
+ participantid: ParticipantInterface
«signal»
MessageInd
+ content: String
ConferenceInterface
ParticipantInterface
Port1
ConferenceBinding

Figure 6-4 The *ConferenceAbstract-Platform*

Figure 6-5 shows the behaviour of the *ConferenceBinding* component specified as a state machine. *ComponentBinding* keeps a list of conference participants, which is updated whenever a *Join* or *Leave* signal is handled. Upon reception of a *MessageReq* signal, the *ConferenceBinding* sends out *MessageInd* signals to all participants of the conference. In order to simplify the behaviour we have assumed that the *MessageInd* signals are sent sequentially based on the order imposed by the list of participants (result of *i.next()*). This illustrates the use of the *«reliable»* stereotype.

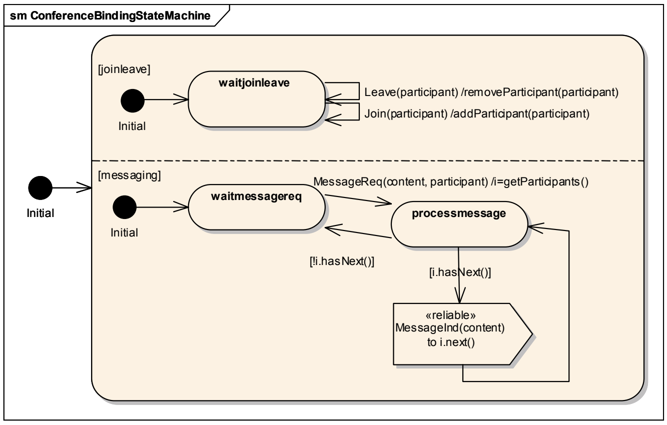

sm ConferenceBindingStateMachine
[joinleave]
waitjoinleave
Leave(participant) /removeParticipant(participant)
Join(participant) /addParticipant(participant)
Initial
[messaging]
Initial
Initial
waitmessagereq
MessageReq(content, participant) /i=getParticipants()
processmessage
[!i.hasNext()]
[i.hasNext()]
«reliable»
MessageInd(content)
to i.next()

Figure 6-5 The *ConferenceBinding* state machine

The application that uses the *ConferenceAbstractPlatform* may be defined at a high-level of platform-independence, communicating with the conference binding through signal exchange. Many alternative implementations for signal exchange are possible, depending on the target platform. Further, there is a large freedom of implementation for the conference abstract platform itself. Since the application is shielded from the internal design of the conference abstract platform, it does not depend on the interaction support eventually used by the conference interaction system.

6.5.3 Realization of the ConferenceAbstractPlatform

Figure 6-6 depicts a realization of the *ConferenceBinding*. This realization relies on the abstract platform that provides reliable signals.

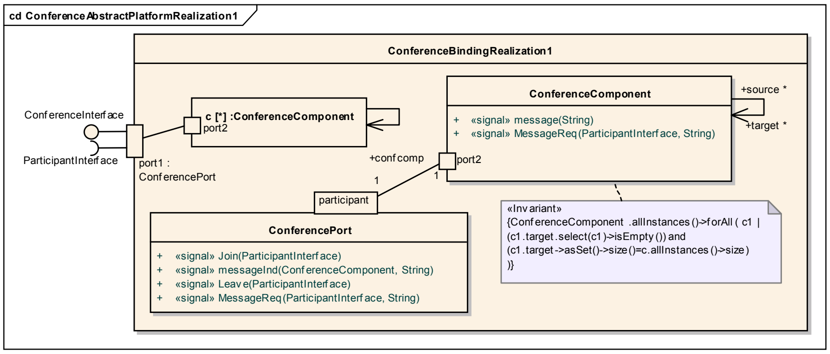

cd ConferenceAbstractPlatformRealization1
ConferenceBindingRealization1
ConferenceInterface
ParticipantInterface
c [*] :ConferenceComponent
port2
port1 : ConferencePort
+confcomp
1
1
participant
ConferenceComponent
+ «signal» message(String)
+ «signal» MessageReq(ParticipantInterface, String)
+source *
+target *
port2
ConferencePort
+ «signal» Join(ParticipantInterface)
+ «signal» messageInd(ConferenceComponent, String)
+ «signal» Leave(ParticipantInterface)
+ «signal» MessageReq(ParticipantInterface, String)
«Invariant»
{ConferenceComponent .allInstances()->forAll(c1 |
(c1.target.select(c1)->isEmpty()) and
(c1.target->asSet()->size()=c.allInstances()->size)
)}

Figure 6-6 A realization of the *ConferenceAbstract-Platform*

The interaction point that corresponds to *port1* is of type *ConferencePort*. The *ConferencePort* handles the signals *Join* and *Leave* and delegates the handling of signals *MessageReq* to the appropriate *ConferenceComponent*. There is a *ConferenceComponent* instance for each participant in the conference. *Conference-Component* instances exchange *message* signals among each other and *message-Ind* with the interaction point of *port1*. The definition of these signals is omitted. An OCL [80] constraint is used to define that *ConferenceComponent* instances are fully connected, and that there are no links between an instance and itself. *Figure 6-7* shows the behaviour associated with the *ConferenceComponent*. The behaviour of *ConferencePort* is omitted for conciseness. The signals are exchanged reliably, and therefore, the stereotype *«reliable»* is applied to all *SendSignalAction* instances.

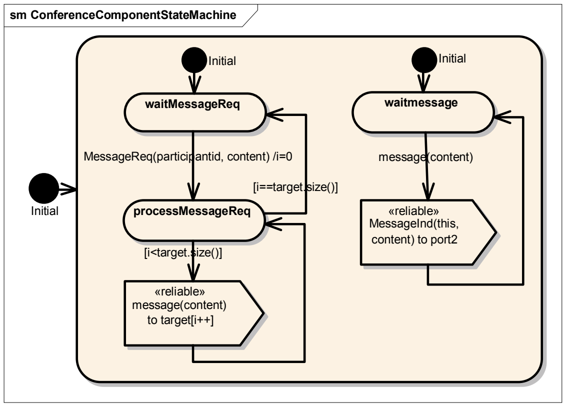

sm ConferenceComponentStateMachine
Initial
waitMessageReq
MessageReq(participantid, content) /i=0
[i==target.size()]
Initial
processMessageReq
[i<target.size()]
«reliable» message(content) to target[i++]
Initial
waitmessage
message(content)
«reliable» MessageInd(this, content) to port2

Figure 6-7 Behaviour of the *ConferenceComponent* represented as a state machine

6.5.4 ConferenceAbstractPlatform realized in terms of EventAbstractPlatform

Figure 6-8 depicts an alternative realization of the *ConferenceBinding*. This realization illustrates the recursive use of an explicitly defined abstract platform. The *EventAbstractPlatform* is used as part *eap* in *ConferenceBindingRealization2*. The dashed line around part *eap* is used to denote that this part is contained by reference. The multiplicity of *eap* is one, i.e., only one instance of the *EventAbstractPlatform* is used in this decomposition of the *ConferenceBinding*.

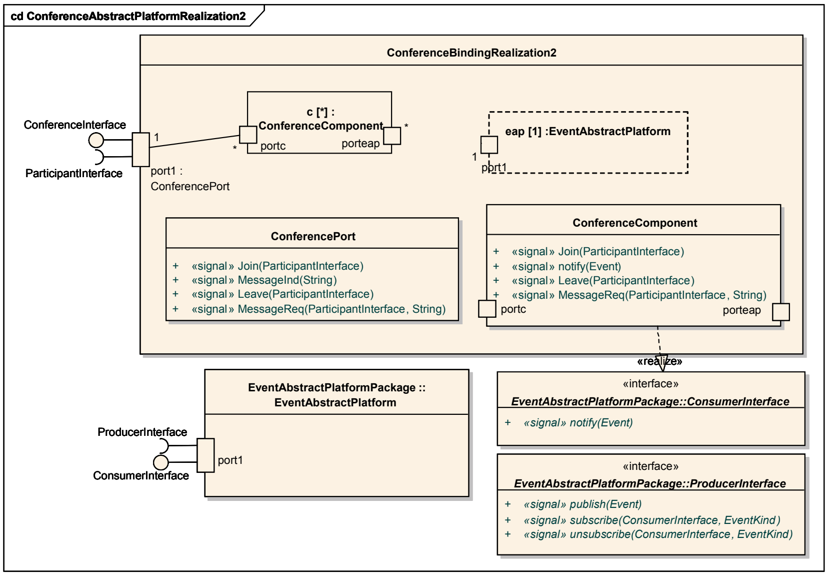

cd ConferenceAbstractPlatformRealization2
ConferenceBindingRealization2
ConferenceInterface
ParticipantInterface
1
port1 : ConferencePort
c [*] : ConferenceComponent
*
portc
porteap
*
eap [1] :EventAbstractPlatform
1
port1
ConferencePort
+ «signal» Join(ParticipantInterface)
+ «signal» MessageInd(String)
+ «signal» Leave(ParticipantInterface)
+ «signal» MessageReq(ParticipantInterface, String)
ConferenceComponent
+ «signal» Join(ParticipantInterface)
+ «signal» notify(Event)
+ «signal» Leave(ParticipantInterface)
+ «signal» MessageReq(ParticipantInterface, String)
portc
porteap
«realize»
EventAbstractPlatformPackage :: EventAbstractPlatform
ProducerInterface
ConsumerInterface
port1
«interface» EventAbstractPlatformPackage::ConsumerInterface
+ «signal» notify(Event)
«interface» EventAbstractPlatformPackage::ProducerInterface
+ «signal» publish(Event)
+ «signal» subscribe(ConsumerInterface, EventKind)
+ «signal» unsubscribe(ConsumerInterface, EventKind)

Figure 6-8 Alternative realization of the *ConferenceAbstract-Platform*

The *EventAbstractPlatform* accepts events and subsequently forwards these events to objects that have subscribed to the particular event type. There is a *ConferenceComponent* for each participant in the conference. The definition of the behaviour of the *EventAbstractPlatform* is omitted here, as well as the classes *Event* and *EventKind*.

The *EventAbstractPlatform* can be realized on a number of event-based platforms, such as, e.g., JMS [104] and CORBA (with the Event Service) [73]. Alternatively, a recursive decomposition of the *EventAbstractPlatform* can be done, resulting, e.g., in a design of the *EventAbstractPlatform* that relies on a request-response abstract platform.

6.6 Discussion

This section discusses some lessons learned by applying the MDA language definition architecture for abstract platform representation and provides some remarks with respect to the interaction concepts provided in UML.

6.6.1 Lessons learned

The example presented in section 6.5 illustrates two kinds of problems that can arise when defining abstract platforms with a particular modelling language.

Firstly, a language's design concepts may force decisions about desired platform properties to be taken too early in the design process, because they do not permit abstraction of these properties. The example in section 6.5 illustrates this for the case of UML state machines. The state machine in *Figure 6-5* determines that message requests are processed one at a time. Therefore, a strict interpretation of this model would exclude realizations of this abstract platform that accept multiple message requests simultaneously. Alternatively, we could have specified that a number of concurrent threads process multiple message requests at the same time. However, this alternative commits to a particular concurrency model. Ideally, we would have stated only that message requests are independent of each other, which is appropriate at the level of abstraction considered. The decision on a particular concurrency model would be delayed, and different alternative implementations would be deemed acceptable. A designer may try to mitigate the limitation of the UML representation by interpreting the behavioural specification loosely, e.g., informally defining that message requests can also be treated simultaneously despite the state machine model. However, this limits the usability of models for model transformation, automated testing, validation and simulation.

Secondly, a language's design concepts may indirectly favour some platforms over others, due to similarities in the structure of models and realizations in a particular platform. Although an implementer could try to ignore the structure and choose to adhere only to the model's semantics, he or she will be inclined to use the platform with the matching structure. The example from section 6.5 illustrates this for UML composite structures. In composite structures, interaction points that correspond to ports can only be created and destroyed along with the component to which they are attached. This implies that, if we want to model that an unbound number of distinct users may use the component through ports, we have to use a multiplexing scheme like the one used in *Figure 6-6* and *Figure 6-8*. Although the specification gives the impression that the multiplexing scheme has to be implemented, it is wiser for the implementer to ignore this scheme in case the target platform allows the dynamic creation and destruction of a component's interaction points. This raises issues with respect to suitable conformance relations for model-driven design with UML.

6.6.2 UML interaction concepts

The basic interaction concepts of UML are derived from operation invocation and message passing mechanisms. Operation invocation and message passing concepts represent both the direction in which information flows and roles of components (or objects) in an interaction (initiator or responder). This forces a designer to commit to a direction of an interaction and roles in an interaction at all levels of platform-independence. This, for example, forces a designer to decide at a high level of platform-independence, whether information is obtained by an entity using a callback, event-based or polling mechanism. For all these mechanisms, information may flow in the same direction, but different parties may initiate interaction. The decision on which mechanism to use often depends on characteristics of the realization platform, and therefore, a designer should not be forced to consider this decision at a high level of platform-independence. For example, a designer may choose between a callback and a polling mechanism for performance reasons. If CORBA is used as a realization platform, using a callback mechanism requires the server-side part of an ORB to be installed on the side of the recipient of the information. This may be problematic, e.g., for mobile devices with few resources. Installing the server-side part of an ORB is not required in the case where the designer chooses for a polling mechanism.

In addition, languages that use operation invocation and message passing concepts often define some details of the mechanisms that realize operation invocation and message passing. In the example we have considered in UML, interacting parties exchange messages through queues of infinite length. Messages exchanged are always delivered unaltered and in sequence. These assumptions may not match the characteristics of a target realization platform, forcing a designer to bridge a large gap between the design and its realization. This significantly decreases the benefit of a model-driven design approach. As we have discussed in section 6.3, UML leaves some of these aspects for the designer to decide ("*The means by which requests are transported to their target depend on the type of requesting action, the target, the properties of the communication medium, and numerous other factors. In some cases, this is instantaneous and completely reliable while in others it may involve transmission delays of variable duration, loss of requests, reordering, or duplication.*" [81]) Such aspects must be decided upon by the application designer (or tool designer), even at a high-level of platform-independence. This is because different decisions for these aspects would result in different application models. We can conclude that semantic variation points allow designers to select between alternative semantics for some of its constructs, but they do not allow designers to abstract from the alternatives, e.g., at a high-level of platform-independence (choice is different from abstraction).

6.7 Concluding remarks

Since modelling language concepts and characteristics of abstract platforms are interrelated, careful selection of a modelling language is indispensable for the beneficial exploitation of the PIM/PSM separation and the definition of abstract platforms.

The MDA Guide [76] provides some examples of "generic platform types" and mentions briefly the need for a "generic platform model", which "can amount to a specification of a particular architectural style." Nevertheless, the introduction of these concepts is superficial: for example, the term "generic platform" is not even defined explicitly, and no further information is given on what a "generic platform model" is. In our interpretation of that documentation, we position our notion of abstract platform as subsuming that of generic platform. This is because abstract platforms can have other relevant characteristics in addition to defining a "particular architectural style". In this chapter, we have identified models that may serve as abstract platform models, in two different approaches to abstract platform definition that can be incorporated in MDA.

In the model-level approach, we have proposed that pre-defined instances of language elements should be part of the abstract platform model. This is corroborated by [12] where a "generalized notion" of platforms is proposed which includes pre-defined instances of language elements.

We have presented an example in UML in which a number of abstract platforms can be combined, both in the language-level and the model-level abstract platform definition approaches.

We have discussed how to support the concept of abstract platform in standard UML, through both the language-level and the model-level abstract platform definition approaches. In the language-level definition approach, the semantic variation points of UML should either be resolved or should be considered irrelevant for deriving intended abstract platform characteristics. UML Profiles can be useful in this approach to specialise design concepts, and manage and package abstract platforms. In the model-level definition approach, UML 2.0's composite structures are useful for defining abstract platforms both from an external and from an internal perspective. Composite structures have been a useful addition to UML 2.0. Nevertheless, we have identified some limitations with respect to the level of abstraction that can be obtained in the representation of abstract platforms with composite structures. In addition, UML 2.0 still lacks some notion of behaviour conformance in order to relate behaviours defined at a high-level of abstraction and the refined realizations of these behaviours.

In chapter 7, we present an example of abstract platform definition with the MOF. In the example, we use a modelling language called Interaction Systems Design Language (ISDL) [52, 89]. The concepts in ISDL are not constrained by the UML, and provide better support for the design framework presented in chapter 5, in particular with respect to the notion of abstract interaction.

Chapter 7

Case study: the design of Freeband Services

In this chapter, we report on the case study that has been conducted in order to show the applicability of the design approach proposed in this thesis. We illustrate all the steps defined in the proposed design process. We start with the definition of abstract platforms and transformations in the preparation phase, and continue to describe the application models produced in the execution phase.

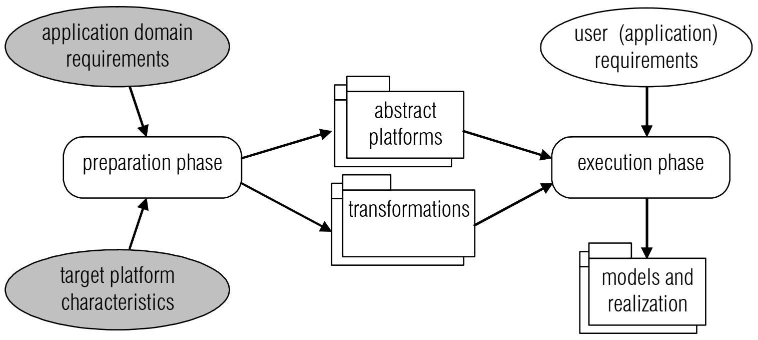
The application domain of this case study is context-aware services. Context-aware services exhibit behaviour that depends on the situation or environment of the user. The target platforms considered include middleware platforms and part of the mobile telecommunications infrastructure, which is used in this case study to send messages to mobile terminal users, and to determine the current location and availability (or presence) of mobile terminal users. We use the term Freeband Services to denote context-aware services that are deployed on the mobile telecommunications infrastructure. This terminology is in-line with that employed in the Freeband A-MUSE project [42], in the context of which this case study has been developed.

This chapter is organised as follows. Section 7.1 defines Freeband Services and the infrastructure upon which they are realized. Section 7.2 gives an overview of the preparation phase activities. Section 7.3 and 7.4 describe the abstract platforms of the service specification and service design levels, respectively. Section 7.5 discusses model transformations. Section 7.6 presents the design of a specific Freeband Service, namely, the Telemonitoring Service. This is intended to illustrate the activities in the execution phase of the design process. Finally, section 7.7 evaluates the results of the case study in terms of the design quality criteria discussed in section 3.1.

7.1 Freeband Services

This section defines Freeband Services, by describing both the requirements of the application domain and the characteristics of the infrastructures that are necessary to realize these services. This is an input for the preparation phase of the design process, as shown in the shaded ovals in *Figure 7-1*.

Figure 7-1 Input for the preparation phase

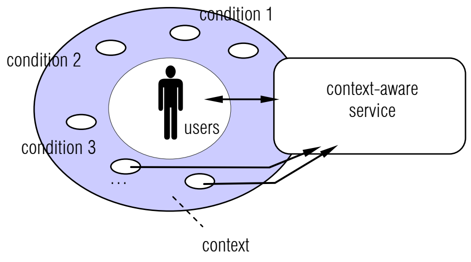

application domain requirements
user (application) requirements
abstract platforms
preparation phase
execution phase
transformations
target platform characteristics
models and realization

7.1.1 Context-awareness

Context-awareness has emerged as an important and desirable feature in distributed mobile applications [35]. Context-awareness refers to the capabilities of applications to provide relevant services to their users by sensing and exploring the users' context [34]. *Context* is defined as a "collection of interrelated conditions in which something exists or occurs". The users' context often consists of a collection of conditions, such as, e.g., the users' location, environmental aspects (temperature, light intensity, etc.), and users' activities [26]. For example, a context-aware service may inform the user when he or she is located within walking range of certain points-of-interest, such as a restaurant or a train station. COMPASS [95] is an example of this kind of application.

Figure 7-2 represents the relation between users, their context and context-aware services.

Figure 7-2 Context-aware service



The users' context may change dynamically, and, therefore, a basic requirement for a context-aware system is its ability to sense context without intervention of the user. Changes in context can be considered external stimuli, which require a reaction from the context-aware system. In section 7.3, we describe a level of models in which a Freeband Service can be described in terms of *events*, which represent contextual changes, and *actions*, which represent actions to be performed in order to provide the service to the user.

7.1.2 Mobility

Two aspects of mobility are relevant to Freeband Services. Firstly, users should be able to access Freeband Services anywhere. A consequence for the realization of a service is that mobile phones and personal digital assistants (PDA) can be used to access the service. Secondly, sensing the users' context may require users to carry or wear devices that are parts of the system. Therefore, the interaction between these devices and other parts of the context-aware system must be supported by a mobile telecommunications platform.

7.1.3 A-MUSE Service Platform

Further decomposition of a context-aware service reveals the architecture shown in *Figure 7-3*. This architecture consists of *context sources*, which are able to sense context and represent it as *context information* in the scope of the system. The service provided by context sources is used by a *coordination component*, which requests actions to be executed by *action providers* depending on situations that can be inferred from context information. For example, two users may require a service to establish a call between them when they are located within a certain range of each other. An example of an action provider suitable for this service is a Parlay gateway [110], which can be requested to establish a telephone call between two users. Each user accesses the service through a user component, which provides the user interface and interacts with the coordination component.

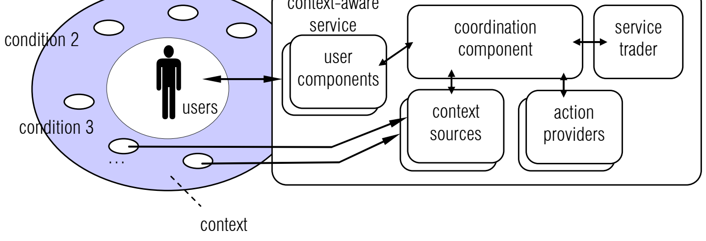


Figure 7-3 Further decomposition of context-aware service

The user component and the coordination component exhibit service-specific behaviour, and are called service components. In contrast, context sources and action providers are general-purpose and, therefore, can be reused in several different Freeband Services. For this reason, we consider context sources and action providers as part of the A-MUSE Service Platform. This platform also supports the interaction between the user component and the coordination component, and the interactions between the coordination component and context sources and action providers. The services provided by context sources and action providers to the coordination component are registered in a *service trader*. This allows the coordination component to select context sources and action providers dynamically according to service offers that are registered in the service trader. Service offers have properties that can be used to select a particular service offer. For example, an action provider can be selected according to its geographical proximity to a user.

7.2 Preparation phase overview

In the preparation phase, we define the required levels of models, identify their abstract platforms and the modelling languages to be used. In addition, we define transformations between related levels of models. The shaded model icons in *Figure 7-4* represent the results of the preparation phase.

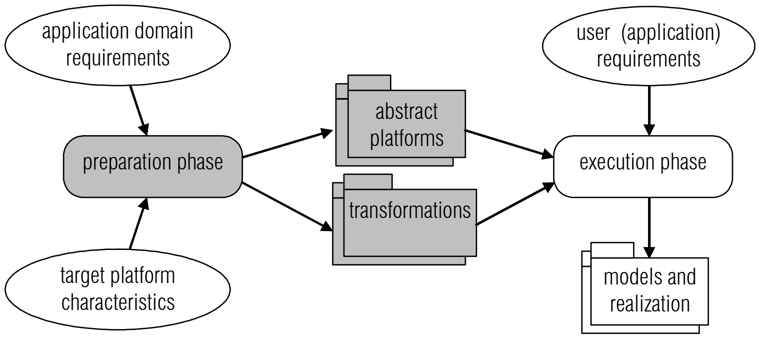

application domain requirements
user (application) requirements
abstract platforms
preparation phase
execution phase
transformations
models and realization
target platform characteristics

Figure 7-4 The preparation phase and its results

Our objective is to capture design knowledge that is applicable to a large number of different Freeband Services and that can be later reused in the execution phase in the design of a specific service that addresses specific service requirements. These requirements correspond to the oval "user (application) requirements" in *Figure 7-4*.

Levels of models

We define the scope of the Freeband Services design trajectory to include the design activities from the specification of a service at a high-level of abstraction to the realization of this service. Given this scope, one extreme approach to organizing the design trajectory would be to have one level of service specification and one level of service realization and one transformation that relates these two levels. However, the gap between these two levels of models is very large.

This means that a lot of effort should be invested in defining the transformation. This effort is rendered useless when changes in the target platform invalidate the transformation. Therefore, the opportunities for reuse can be increased if an intermediate level of models is introduced. This level of models uses an abstract platform to achieve platform independence, and, hence, models at this level can be reused for different target platforms. The organization of the design trajectory is depicted in *Figure 7-5*.

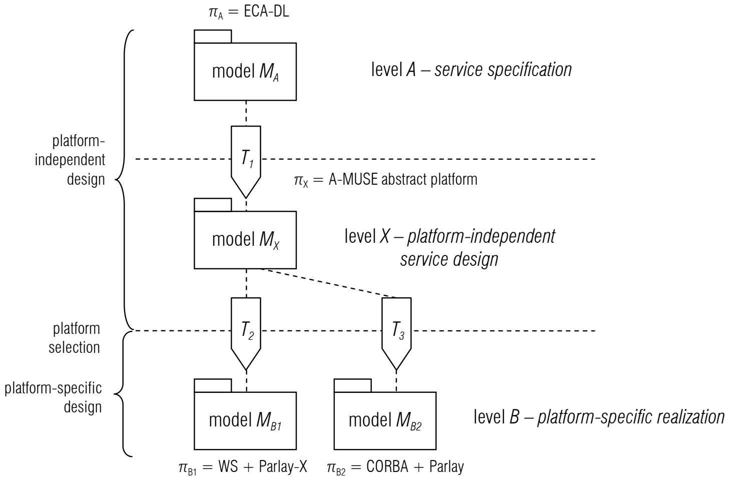

π_A = ECA-DL
model M_A
level *A – service specification*
platform-independent design
T_1
π_X = A-MUSE abstract platform
model M_X
level *X – platform-independent service design*
platform selection
T_2
T_3
platform-specific design
model M_{B1}
model M_{B2}
level *B – platform-specific realization*
π_{B1} = WS + Parlay-X
π_{B2} = CORBA + Parlay

Figure 7-5 An intermediate level of models between service specification and platform-specific realization

The three levels of models are defined as follows:

– *Service specification* level. This level of models describes the behaviour of a Freeband Service from an integrated perspective, i.e., we do not distinguish the environment (including service users) and the service provider. The concept of abstract action (described in chapter 5) is used to model both the occurrence of events originated from context sources and the execution of actions. The language we use to represent service specifications is called Events-Conditions-Actions Domain Language (ECA-DL) and it is presented in section 7.3.
– *Platform-independent service design* level. This level of models describes the behaviour of a Freeband Service revealing the A-MUSE Service Platform (as illustrated in *Figure 7-3*). The A-MUSE Service Platform is described

as an abstract platform, and is decomposed into a hierarchy of abstract platforms (see section 7.4). It relies on a Service-Oriented Architecture (SOA) abstract platform. This SOA abstract platform uses abstract interactions to support the communication of application parts in this design, and provides a service trader with support for dynamic service properties.

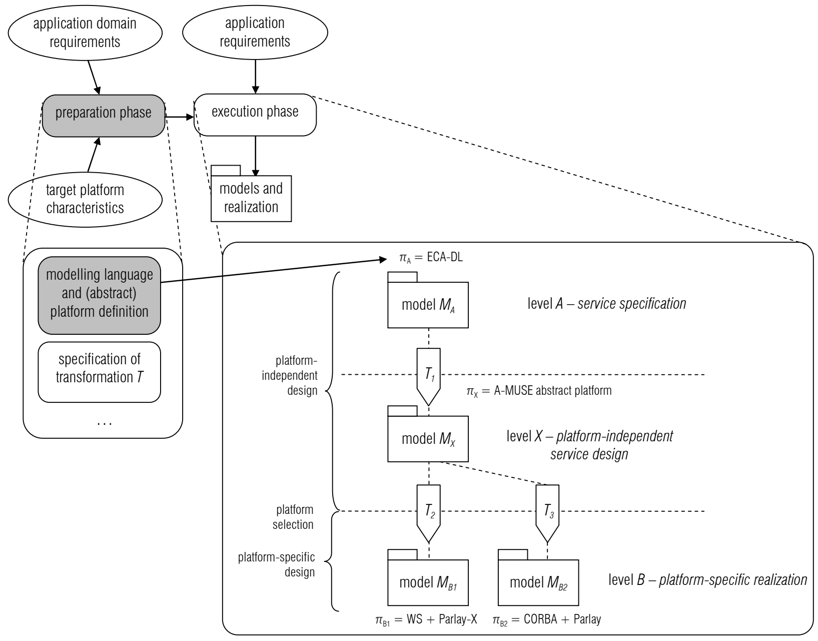
- *Platform-specific service design* level. This level of models describes the realization of the service for a particular middleware platform. In order to show the flexibility of the relation between the platform-independent service design level and the platform-specific service design level two different middleware platforms are used, namely, Web Services and CORBA. These platforms offer support Parlay-X and Parlay services respectively.

Simplifications

In order to limit the size of the case study reported in this chapter, a number of simplifications have been made. First, we assume that context sources and action services are available and can be reused by service components. The implications of this assumption are explained in section 7.4.4. Second, we assume that the communication with users is done via action services. Therefore, the coordination component does not interact with user components but uses a suitable action service for communication with users.

7.3 Service specification level

In this section, we show the definition of the service specification level. The shaded icons in *Figure 7-6* denote the phase and activity performed.



application domain requirements
application requirements
preparation phase
execution phase
target platform characteristics
models and realization
modelling language and (abstract) platform definition
specification of transformation T
...
π_A = ECA-DL
model M_A
level A – service specification
platform-independent design
T_1
π_X = A-MUSE abstract platform
model M_X
level X – platform-independent service design
platform selection
T_2
T_3
platform-specific design
model M_{B1}
model M_{B2}
level B – platform-specific realization
π_{B1} = WS + Parlay-X
π_{B2} = CORBA + Parlay

Figure 7-6 Defining the service specification level during the preparation phase

7.3.1 Abstract platform definition

At the level of service specification, a Freeband Service is described in terms of *events*, which represent contextual changes, *queries* to context sources, and *actions*, which represent actions to be performed in order to provide the service to the user. The abstract platform supports the execution of these events, queries and actions according to the behaviour defined in the service specification.

We use the language-level approach to the definition of the abstract platform at this level. This leads to a domain-specific language for the domain of Freeband Services specification. We specialize elements of a general-purpose design language, namely the Interaction System Design Language (ISDL) [52, 89] thus defining a dialect of it, which we call Events-Conditions-Actions Domain Language (ECA-DL). This language provides a means to specify behaviours in terms of actions and causality relations between these actions. The concept of an action in ISDL is identical to that of an abstract action as defined in the design framework presented in chapter 5 of this thesis, since they are both based on the same basic concepts, as defined in [40, 90]. The definition of this abstract platform is illustrated schematically in *Figure* 7-7.

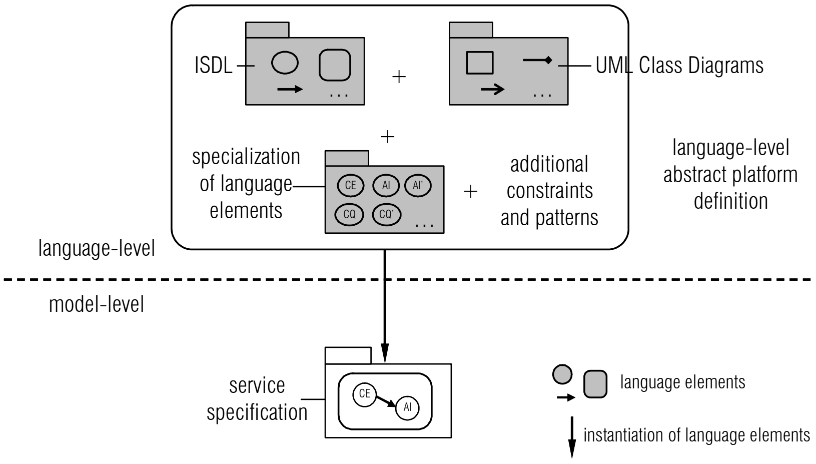

ISDL
UML Class Diagrams
specialization of language elements
CE
AI
AI'
CQ
CQ'
additional constraints and patterns
language-level abstract platform definition
language-level
model-level
service specification
CE
AI
language elements
instantiation of language elements

Figure 7-7 Language-level abstract platform definition for service specification

The ISDL specialization consists of defining special types of actions, namely, *context events* (CE in *Figure 7-7*), *context query requests* (CQ), *context query responses* (CQ'), *action invocation requests* (AI) and *action invocation responses* (AI'). Context query request and context query responses are always related by causality, forming a pattern.

We use the notation supported by the ISDL modelling tool Grizzle [52]. A behaviour is represented by a rounded rectangle. An action is represented by an oval. Action attributes are drawn inside a box and attached to an action by a line. A causality condition is represented by an arrow. The action pointed to by an arrow can only occur after the action at the origin of the arrow has occurred. *Figure 7-8* shows an example of a simple service specification, using the specialized types of actions and some causality conditions between the actions. We use a simple naming convention with suffixes to denote specialized actions: the suffix *_indC* (or shortly *_ind*) denotes a context event, *_reqC* denotes a context query request, *_rspC* denotes a context query response, *_reqA* denotes an action invocation request and, finally, *_rspA* denotes an action invocation response. This allows us to reuse tool support without modifications. In the example, the occurrence of a context event *seizureAlert_ind* is followed by the occurrence of an *alertTeam_reqA* action invocation request. This example is inspired by the Telemonitoring service that is described in detail in section 7.6. The context event *seizureAlert_ind* occurs when an (imminent) epileptic seizure is detected on a patient being monitored. The result of *alertTeam_reqA* is that a health care team is alerted. This team can provide proper care for the patient suffering a seizure.

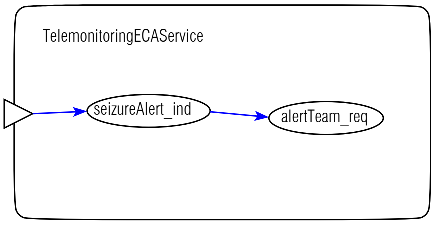


Figure 7-8 Example of a service specification

ISDL allows designers to use a modelling language of their choice to define the attributes of actions and constraints on these attributes. For ECA-DL, we have chosen to use UML class diagrams for the information attributes. Further, we use a simple constraint language to express constraints on information attributes. The constraint language is defined in section 7.3.2. A straightforward transformation of expressions in this language to expressions in Object Constraint Language (OCL) [80] is also provided in that section.

Figure 7-9 shows the example presented in *Figure 7-8* augmented with information attributes. The context event *seizureAlert_ind* has an information attribute *pat* of type *Patient*. The action invocation request *alertTeam_reqA* has an information attribute *pat* of type *Patient* and an information attribute *alert* of type *String*. The attribute *pat* of *alertTeam_reqA* is constrained so that it is identical to *pat* of *seizureAlert_ind*. The attribute *alert* is constrained so that its value equals the string "Epileptic seizure". The type *Patient* is defined with a UML class. In this specification, the value for the attribute *pat* of *seizureAlert_ind* is established in a non-deterministic way, since it is left unconstrained. This allows us to capture the behaviour of the service for any patient, abstracting from the conditions which cause a seizure.

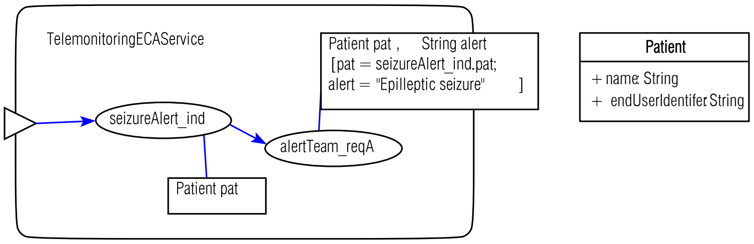

TelemonitoringECAService
Patient pat , String alert
[pat = seizureAlert_ind.pat;
alert = "Epilleptic seizure"]
seizureAlert_ind
alertTeam_reqA
Patient pat
Patient
+ name: String
+ endUserIdentifer String

Figure 7-9 Example of a service specification with information attributes

Constraints on information attributes can be used to specify not only the required results of action services, but also required *properties* of these services. This is illustrated in *Figure 7-10*. In this example, the attribute *isAvailable* is required to be *true*. We can say that the attribute *isAvailable* represents a property of the health team alert service. Properties can be used to select action services based on context information.

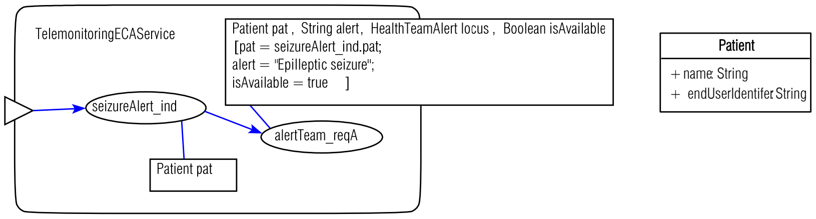

TelemonitoringECAService
Patient pat , String alert, HealthTeamAlert locus , Boolean isAvailable
[pat = seizureAlert_ind.pat;
alert = "Epilleptic seizure";
isAvailable = true]
seizureAlert_ind
alertTeam_reqA
Patient pat
Patient
+ name: String
+ endUserIdentifer String

Figure 7-10 Example of a service specification with an information attribute playing the role of a property

Location attribute

The specialized actions in ECA-DL are said to occur at different locations. The notion of location in ISDL and ECA-DL does not necessarily correspond to the geographical location of users, but rather the (logical) location at which an action occurs in the system being modelled. For example, a location may represent a context source at which a context event occurs, or an action provider at which an action invocation request occurs.

Similar to the case of information attributes, it is possible to model ISDL location attribute types with UML. We have created a pre-defined location attribute type called *Locus*. Service specifications may specialize *Locus*. *Figure 7-11* shows an example of specification with a location attribute. The *alertTeam_reqA* action invocation request has been augmented with a location attribute *locus* of type *HealthTeamAlert*. The predefined location attribute type and its specialization are depicted in a UML class diagram.

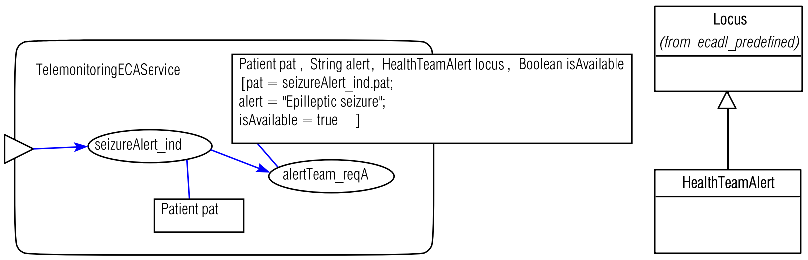

TelemonitoringECAService
Patient pat , String alert, HealthTeamAlert locus , Boolean isAvailable
[pat = seizureAlert_ind.pat;
alert = "Epilleptic seizure";
isAvailable = true]
seizureAlert_ind
alertTeam_reqA
Patient pat
Locus
(from ecadl_predefined)
HealthTeamAlert

Figure 7-11 Example of a service specification with information and location attributes

Locus is defined at the model-level, so, actually, a combination of the language-level and model-level approaches is used for the service specification level. This is shown in *Figure 7-12*.

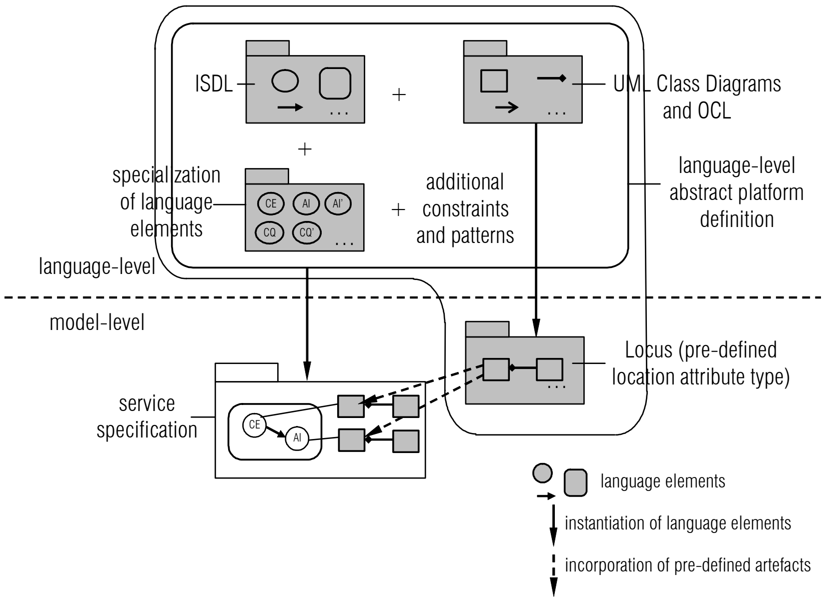

ISDL
+
UML Class Diagrams and OCL
+
specialization of language elements
CE
AI
AI'
CQ
CQ'
+
additional constraints and patterns
language-level abstract platform definition
language-level
model-level
Locus (pre-defined location attribute type)
service specification
CE
AI
language elements
instantiation of language elements
incorporation of pre-defined artefacts

Figure 7-12 Language-level and model-level abstract platform definition for service specification

7.3.2 ECA-DL metamodel

In this section, we present the ECA-DL metamodel. It consists of: (i) a specialization of the ISDL metamodel[12], (ii) part of the UML metamodel (which is used to represent information and location attributes) and (iii) metaclasses for modelling constraints. First, we present the ISDL metamodel, then specialize it with ECA-DL metaclasses to obtain (i). After that, we discuss (ii) and (iii).

ISDL metamodel

We start with the part of the ISDL metamodel that supports behaviour description. The metamodel in *Figure 7-13* shows that a monolithic behaviour (an instance of *MonolithicBehaviourType*) consists of causality relations (instances of *CausalityRelation*). Causality relations describe the conditions (an instance of *CausalityCondition*) for the occurrence of a causality target instantiation (an instance of *CausalityTargetInstantiation*). At this level of models, causality relations are applied to actions (an instance of *ActionInstantiation*, which is a subclass of *CausalityTargetInstantiation*).

[12] We use the ISDL metamodel described in chapter 4 of [33]

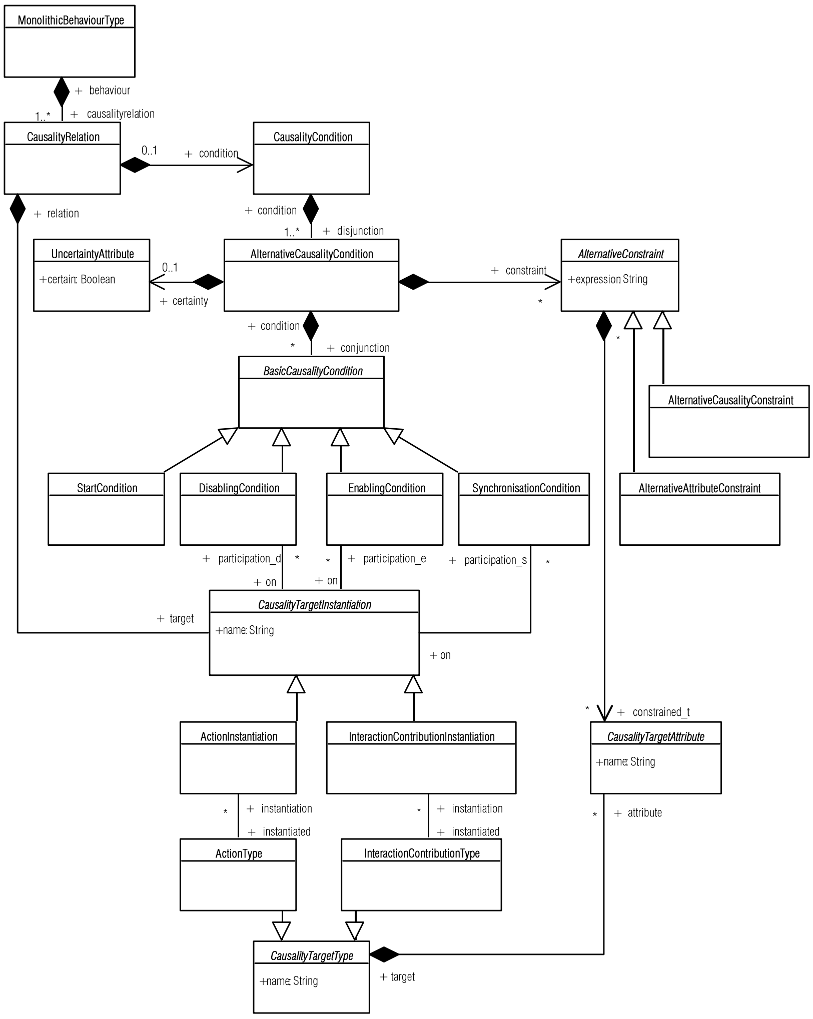
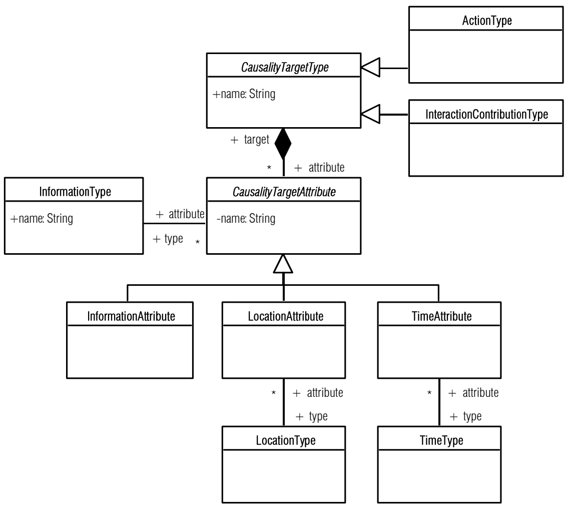


Figure 7-13 Monolithic Behaviours in ISDL

Interaction contributions are not used at the service specification level (and hence *InteractionContributionType* should not be instantiated in ECA-DL), since at this level we define the integrated behaviour of the Freeband Service and its environment. In addition, we do not use synchronization conditions (*SynchronizationCondition*) nor uncertainty attributes (*UncertaintyAttribute*). The disabling condition (*DisablingCondition*) is only used in a choice pattern, which is a composition of mutual disabling conditions [89]. These restrictions in the use of concepts limit the expressiveness of ECA-DL, but facilitate the transformations of service specifications to service designs.

Figure 7-14 shows the ISDL metaclasses that represent the attributes of actions, namely, information attributes (*InformationAttribute*, with a corresponding *InformationType*), location attributes (*LocationAttribute*, with a corresponding *LocationType*) and time attributes (*TimeAttribute,* with a corresponding *TimeType*). Time attributes are not used in our version of the ECA-DL.

Figure 7-14 Attributes in ISDL



Specializing the ISDL metamodel

Figure 7-15 shows the specialization of the *ActionType* and *ActionInstantiation* metaclasses, introducing metaclasses for context events (*ContextEventType*, *ContextEventInstantiation*), context query requests (*ContextRequestType*, *ContextRequestInstantiation*), context query responses (*ContextResponseType*, *ContextResponseInstatiation*), action invocation requests (*ActionInvocation-RequestType*, *ActionInvocationRequestInstantiation*) and action invocation responses (*ActionInvocationResponseType*, *ActionInvocationResponseInstantiation*).

The specialization shown in *Figure 7-15* could have been implemented by using the MOF profiling mechanism on the ISDL metamodel. In this case, the ECA-DL profile would consist of stereotypes for *ActionType* and *ActionInstantiation*.

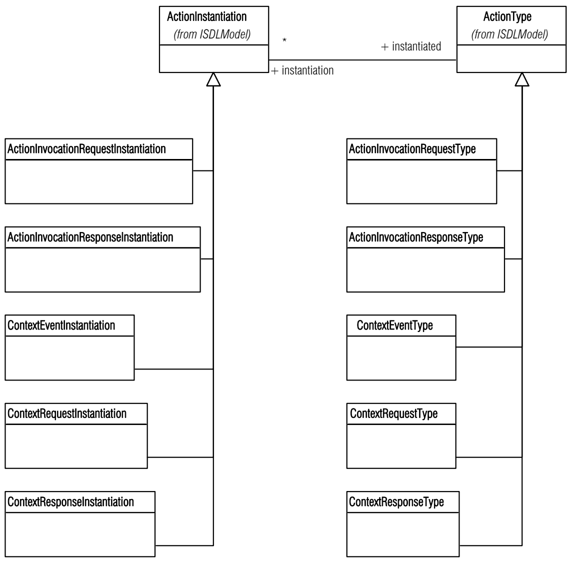

ActionInstantiation
(from ISDLModel)
ActionType
(from ISDLModel)
*
+ instantiated
+ instantiation
ActionInvocationRequestInstantiation
ActionInvocationResponseInstantiation
ContextEventInstantiation
ContextRequestInstantiation
ContextResponseInstantiation
ActionInvocationRequestType
ActionInvocationResponseType
ContextEventType
ContextRequestType
ContextResponseType

Figure 7-15 Specialization of actions in ECA-DL

Information and location attributes

Figure 7-16 shows the ISDL language elements used to model the types of the information, location and time attributes. A composite information type (an instance of *CompositeInformationType*) consists of several information blocks (instances of *InformationBlock*), which are themselves typed by an information type (an instance of *InformationType*).

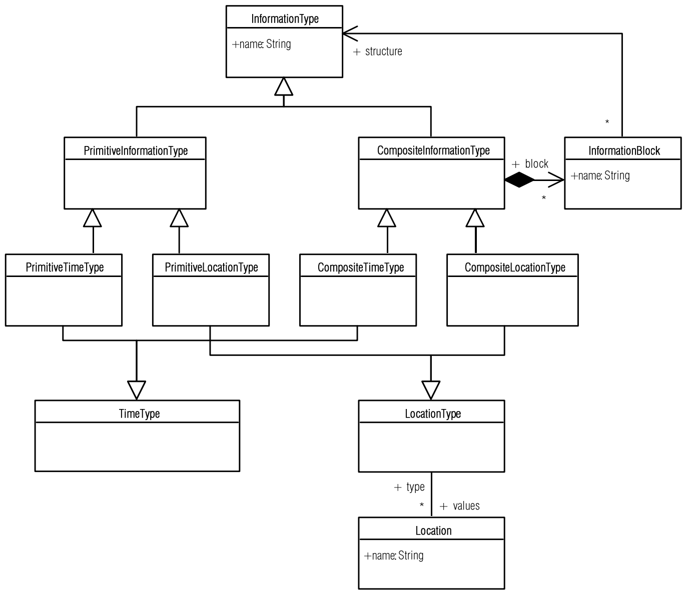


Figure 7-16 Attribute types in ISDL

In order to use UML as a language for information and location attributes, we define a correspondence between ISDL language elements (depicted in *Figure 7-16*) and UML language elements. *Figure 7-17* shows the part of the UML metamodel used to describe the types of information and location attributes in ECA-DL. Primitive types that can be used are *Integer*, *Boolean* and *String*.

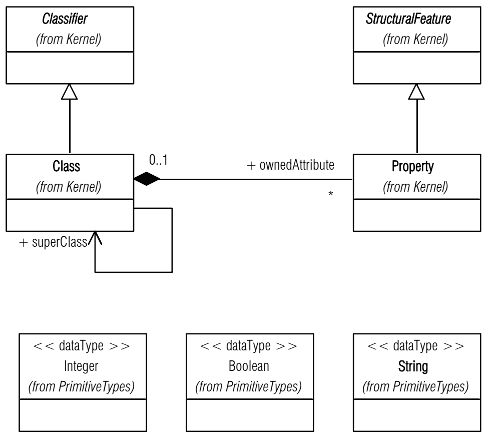


Figure 7-17 Part of the UML metamodel used to define information and location attributes

For each *CausalityTargetType* instance (thus including instances of the subclass *ActionType*), there is a corresponding instance of *Class*. For each *CausalityTargetAttribute* instance, there is a corresponding instance of *Property*. This instance of *Property* represents either an information attribute or a location attribute, and is typed by a *Class* which corresponds to a *LocationType* or an *InformationType*. For each instance of *InformationBlock* associated with a *CompositeInformationType*, there is a corresponding instance of *Property*. The correspondence is shown in an example in *Figure 7-18*, which revisits the example presented in *Figure 7-11*.

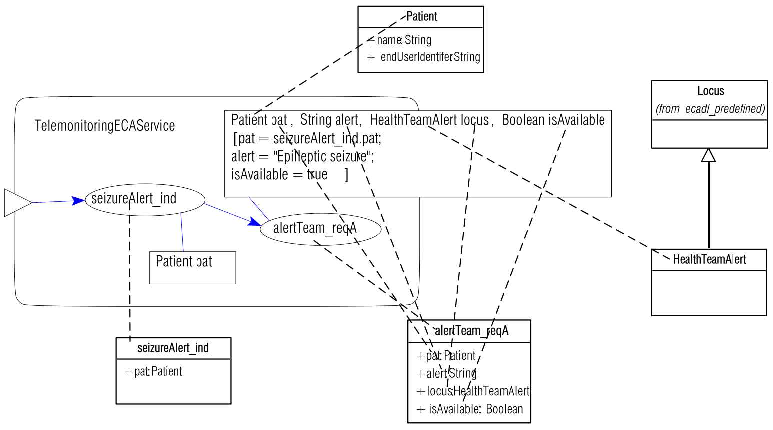


Figure 7-18 Example of a service in ECA-DL, revealing the binding with UML to represent information and location attributes

Table 7-1 summarizes the relation between ISDL language elements and UML language elements (adapted from chapter 4 of [33]).

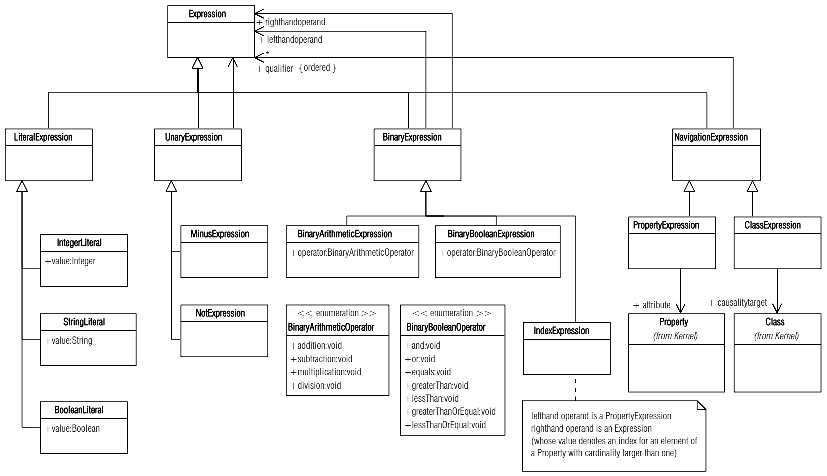
Table 7-1 Relation between ISDL language elements and UML language elements

ISDL language element	**UML 2.0 language element**
InformationType	*Classifier (Class* or *DataType)*
PrimitiveInformationType	*DataType* (primitive data types: *Integer*, *Boolean* and *String*)
CompositeInformationType	*Class*
InformationBlock	*Property* of *Class* that represents *CompositeInformationType*
CausalityTargetInstantiation	*Class*
Attribute	*Property* of *Class* that represents the result of an Action or Interaction
AlternativeConstraint	*OclExpression* (that can be derived from the ECA-DL Constraint language, see below)

Constraints

Figure 7-19 shows the metamodel of the ECA-DL constraint language. An expression in ECA-DL constraint language allows us to represent constraints on attributes (instances of *AlternativeAttributeConstraint*, a subclass of *AlternativeConstraint*). Integer, Boolean and String literals are supported, as well as Boolean operators (*and*, *or* and *not*), arithmetic operators (*unary minus* and *binary addition*, *subtraction*, *multiplication* and *division*) and comparison operators (*equals*, *greater than*, *great than or equal*, *less than* and *less than or equal*). A property expression (instance of *PropertyExpression*) refers to an information or location attribute, or an information block of an attribute (which are represented as instances of *Property* in UML). Since the *Property* metaclass in UML is a subclass of *MultiplicityElement*, properties may have cardinalities larger than one, e.g., to represent sequences. In the constraint language, we provide an index expression to select an element of such sequences. A class expression (instance of *ClassExpression*) is used to test the occurrence of an action or interaction, and is evaluated to *true* in case the action or interaction has occurred and *false* otherwise.

Figure 7-19 Metamodel of the ECA-DL constraint language



For convenience, we define a concrete textual syntax for the ECA-DL constraint language. The EBNF (Extended Backus-Naur Form) is given in *Figure 7-20*[13].

Figure 7-20 EBNF specification of concrete textual syntax for ECA-DL constraint language

```
<Expression> :=
      <LiteralExpression> |
      <UnaryExpression> |
      <BinaryExpression> |
      <NavigationExpression> |
      ( <Expression> )

<LiteralExpression> :=
      <BooleanLiteral> |
      <IntegerLiteral> |
      <StringLiteral>

<BooleanLiteral> :=
      true |
      false

<IntegerLiteral> := <Digits>

<Digits> :=
      <Digits> <Digit> |
      <Digit>

<StringLiteral> := " <TextChars> "

<TextChars>:=
      /* <empty> */ |
      <TextChars> <TextChar>

<TextChar>:=
      <Alpha> |
      <Digit> |
      <Other> |
      <Special>

<Special>:=
      \\ |
      \"

<UnaryExpression> :=
      - <Expression> |
      not <Expression>
```

```
<BinaryExpression> :=
      <BinaryArithmeticExpression> |
      <BinaryBooleanExpression>

<BinaryArithmeticExpression> :=
      <Expression> + <Expression> |
      <Expression> - <Expression> |
      <Expression> * <Expression> |
      <Expression> / <Expression>

<BinaryBooleanExpression> :=
      <Expression> and <Expression> |
      <Expression> or <Expression> |
      <Expression> < <Expression> |
      <Expression> <= <Expression> |
      <Expression> > <Expression> |
      <Expression> >= <Expression> |
      <Expression> <> <Expression>

<NavigationExpression> :=
      <Ident> |
      <Expression> . <Ident>

<Ident>:=
      <Leader> <FollowSeq>

<FollowSeq>:=
      /* <empty> */ |
      <FollowSeq> <Follow>

<Leader>:=<Alpha>

<Follow>:=
      <Alpha>|
      <Digit>|
      _
```

Textual expressions in ECA-DL are also OCL expressions. The textual expression defines in this way an implicit mapping between ECA-DL and (a subset of) OCL.

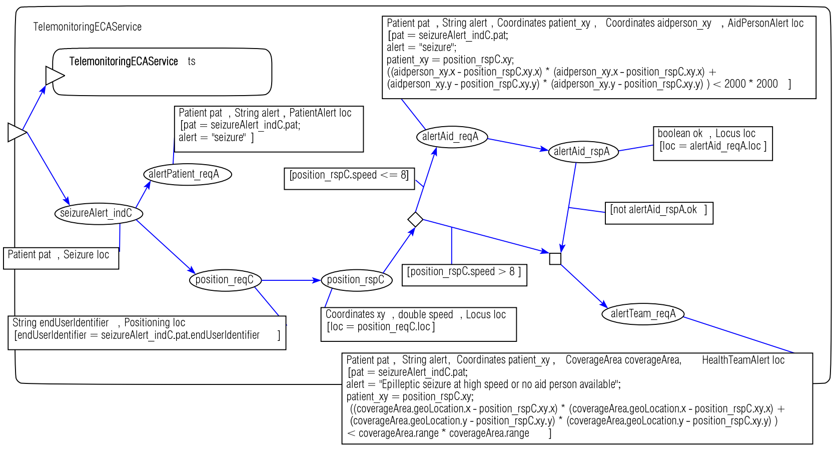
[13] <*Alpha*> is the set of alphabetic characters (from "A" to "Z", and "a" to "z"), <*Digit*> is the set of digits (from "0" to "9"), and <*Other*> is the set of ASCII characters that are not <*Alpha*>, <*Digit*>, nor <*Special*>.

7.3.3 Service specification example

The design trajectory described in this chapter aims at providing support for the design of Freeband Services in general. In order to illustrate how a designer can make use of the abstract platform at a certain level of models, we must show the design of a specific service. We have chosen to use the Telemonitoring Service as example. This service has been used as a case study in both the Freeband A-MUSE [42] and AWARENESS [41] projects. We describe a simplified version of the scenario considered.

We assume that patients are monitored with a wearable 24-hour epilepsy seizure monitoring system. During a couple of minutes around the onset of a seizure, the monitoring system detects its signs. The patient is warned of a (imminent) seizure and based on location information a voluntary aid person or a health team can be dispatched for assistance.

The Telemonitoring Service specification is depicted in *Figure 7-21*. We use a shorthand notation to denote a *choice* between two actions (a white diamond). Choice can be described in terms of enabling and disabling causality conditions, as discussed in [52, 89].


TelemonitoringECAService
TelemonitoringECAService ts
Patient pat , String alert , Coordinates patient_xy , Coordinates aidperson_xy , AidPersonAlert loc
[pat = seizureAlert_indC.pat;
alert = "seizure";
patient_xy = position_rspC.xy;
((aidperson_xy.x - position_rspC.xy.x) * (aidperson_xy.x - position_rspC.xy.x) +
(aidperson_xy.y - position_rspC.xy.y) * (aidperson_xy.y - position_rspC.xy.y)) < 2000 * 2000]
Patient pat , String alert , PatientAlert loc
[pat = seizureAlert_indC.pat;
alert = "seizure"]
alertAid_reqA
alertAid_rspA
boolean ok , Locus loc
[loc = alertAid_reqA.loc]
alertPatient_reqA
[position_rspC.speed <= 8]
[not alertAid_rspA.ok]
seizureAlert_indC
Patient pat , Seizure loc
position_reqC
position_rspC
[position_rspC.speed > 8]
alertTeam_reqA
String endUserIdentifier , Positioning loc
[endUserIdentifier = seizureAlert_indC.pat.endUserIdentifier]
Coordinates xy , double speed , Locus loc
[loc = position_reqC.loc]
Patient pat , String alert, Coordinates patient_xy , CoverageArea coverageArea, HealthTeamAlert loc
[pat = seizureAlert_indC.pat;
alert = "Epilleptic seizure at high speed or no aid person available";
patient_xy = position_rspC.xy;
((coverageArea.geoLocation.x - position_rspC.xy.x) * (coverageArea.geoLocation.x - position_rspC.xy.x) +
(coverageArea.geoLocation.y - position_rspC.xy.y) * (coverageArea.geoLocation.y - position_rspC.xy.y))
< coverageArea.range * coverageArea.range]
Figure 7-21 The Telemonitoring Service specification

The specification shows a number of specialized actions. A simple naming convention has been used to indicate the type of action. The event *seizureAlert_ind* represents that an (imminent) epileptic seizure has been detected in a patient being monitored. The action *alertPatient_reqA* requests the patient to be informed about the seizure. Following a seizure alert, the patient's current location and speed is requested (*position_reqC* followed by *position_rspC*). An aid person within range of the patient is informed of the seizure and the current location of the patient (*alertAid_reqA*). When no aid persons are available or the speed of the patient exceeds a certain value (which could indicate a hazardous situation) a health team capable of handling epileptic seizures is dispatched to the location of the patient. *Figure 7-22* show the attribute types for the Telemonitoring Service specification.

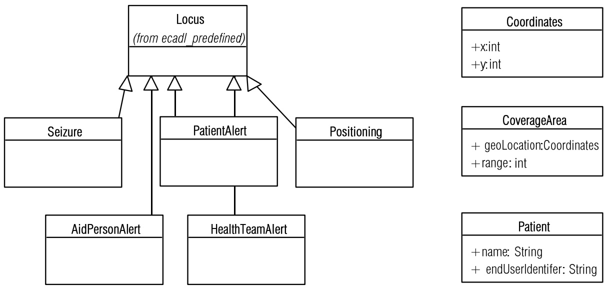


Figure 7-22 Attribute types for the Telemonitoring Service specification

In section 7.6, we show the transformation of the Telemonitoring service specification into a platform-independent service design. Before that, we present both the platform-independent service design level (section 7.4) and the transformation from service specification level to the service design level (section 7.5).

7.4 Platform-independent service design level

In this section, we show the definition of the platform-independent service design level. The shaded icons in *Figure 7-23* denote the phase and activity performed.

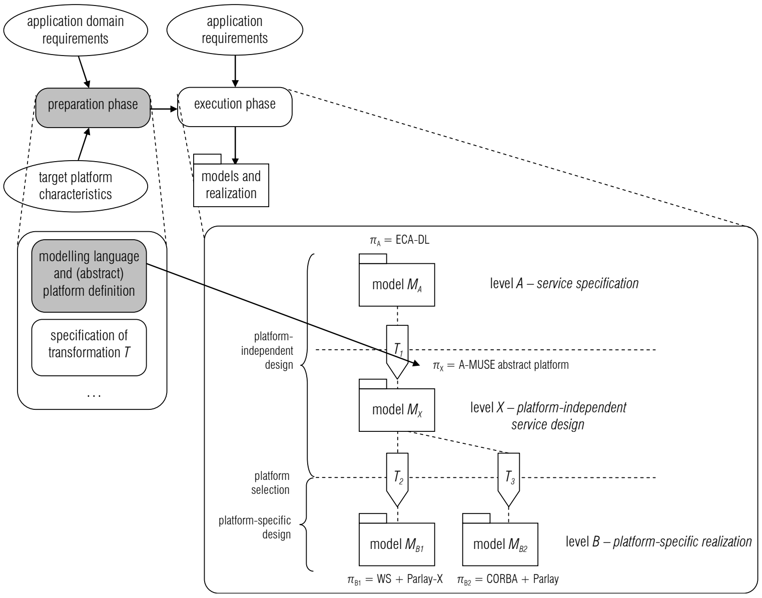
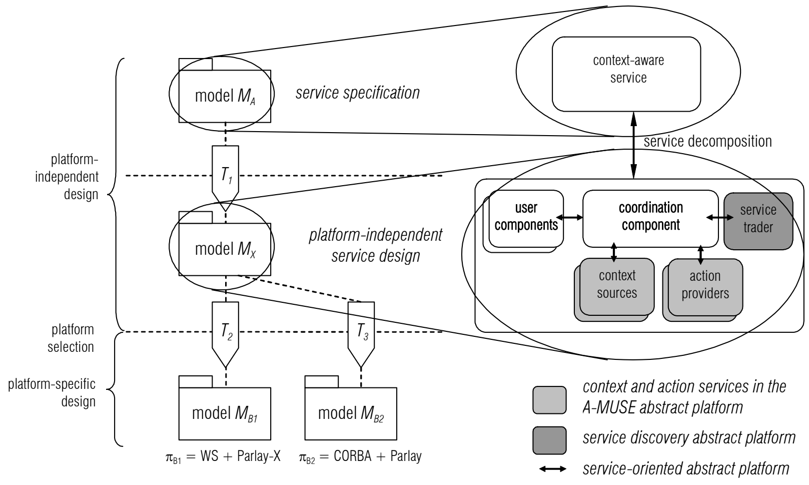

application domain requirements
application requirements
preparation phase
execution phase
target platform characteristics
models and realization
modelling language and (abstract) platform definition
specification of transformation T
...
π_A = ECA-DL
model M_A
level A – service specification
platform-independent design
T_1
π_X = A-MUSE abstract platform
model M_X
level X – platform-independent service design
platform selection
T_2
T_3
platform-specific design
model M_{B1}
model M_{B2}
level B – platform-specific realization
π_{B1} = WS + Parlay-X
π_{B2} = CORBA + Parlay

Figure 7-23 Defining the platform-independent service design level during the preparation phase

7.4.1 Abstract platform definition

In the platform-independent service design level, action services and context sources interact with a coordination component to provide the service specified at the service specification level. Context sources, action services and a service trader are parts of the A-MUSE abstract platform. The A-MUSE abstract platform relies on an underlying service discovery abstract platform and an underlying service-oriented abstract platform. The abstract platforms at the platform-independent service level are depicted schematically in *Figure 7-24*. This figure also shows the relation between the service specification level and the platform-independent service design level. We discuss this relation in further detail in section 7.5.

Figure 7-24 Abstract platforms at the platform-independent service design level



The decomposition of the A-MUSE abstract platform into a hierarchy of abstract platforms facilitates its definition. We use a combination of the language-level and model-level abstract platform definition approaches to define this hierarchy.

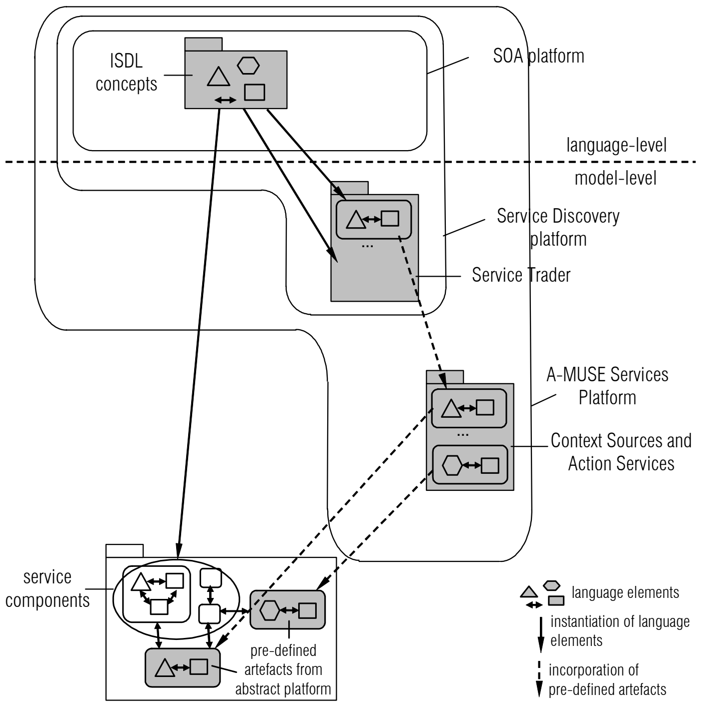
We start with the definition of the underlying service-oriented abstract platform. The service-oriented abstract platform is defined using a pure language-level approach. Similarly to the case of the service specification level, the language adopted for this level is ISDL; however, at this level, no specialization of the language is necessary. The information and location attributes are described with UML.

The service discovery abstract platform is built on top of the underlying service-oriented abstract platform and is defined with a model-level approach. This abstract platform provides a trader service, which can be composed with an application (in this case, service components). The trader service is defined in ISDL. Information attributes (e.g., service offers) are described with UML. This use of a trader service is a well established pattern of service discovery in service-oriented architectures. Examples of service traders in middleware platforms are the OMG CORBA trader [88] and the UDDI registry [71] (a Web Services technology).

The A-MUSE Abstract Platform is built on top of the service discovery abstract platform (and the service-oriented abstract platform). It is defined with the model-level approach. The A-MUSE Abstract Platform offers context sources and action services, which can be composed with service components. Service components discover context services and action services through the trader service.

A schematic overview of the approach for the definition of this hierarchy of abstract platform is shown in *Figure 7-25*.

Figure 7-25 Defining the hierarchy of abstract platforms definition


ISDL concepts
SOA platform
language-level
model-level
Service Discovery platform
Service Trader
A-MUSE Services Platform
Context Sources and Action Services
service components
pre-defined artefacts from abstract platform
language elements
instantiation of language elements
incorporation of pre-defined artefacts

In the following sections, we define each of the abstract platforms.

7.4.2 Service-oriented abstract platform

The service-oriented abstract platform supports the composition of various (potentially distributed) components which operate through services. The concept of abstract interaction is used, as well as some supporting structuring concepts. The modelling language we use at this level is ISDL.

Figure 7-26 shows part of the ISDL metamodel revealing interaction contribution instantiation and interaction contribution types, which are special kinds of causality target instantiation and causality target types. A behaviour can therefore relate different interaction contributions with causality conditions.

Figure 7-26 Monolithic behaviour types including interactions

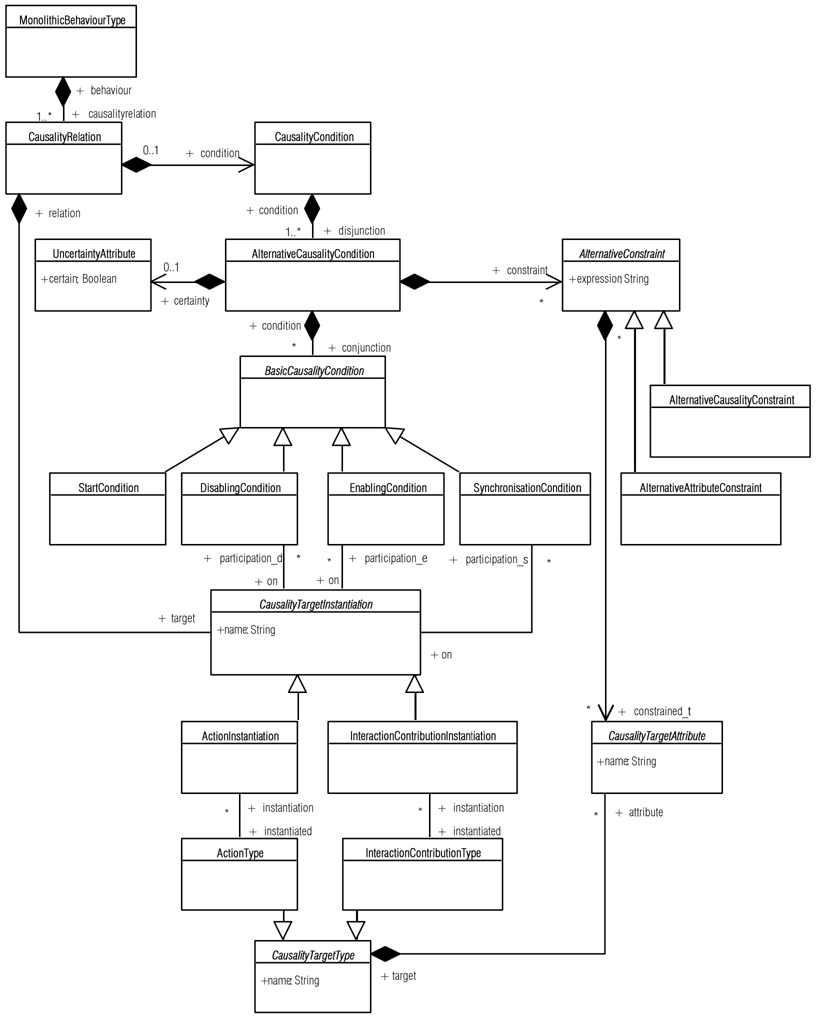
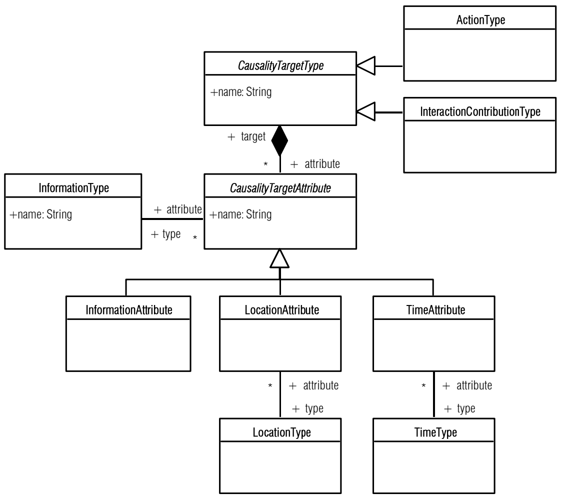


Figure 7-27 reveals that interaction contribution types may have attributes in the same way as action types. The constraints can be used by each interacting party in an interaction to constrain the results of an interaction (information and location attributes): each party may offer a set of values, accept a set of values, or both. This results in *value passing*, *value checking* and *value generation* as discussed in chapter 5 of this thesis.

Figure 7-27 Attributes of interactions



So far, we have only discussed how interaction contributions can be described within the context of a single monolithic behaviour. However, since at the platform-independent service design level we describe the composition of different services, we require structured behaviour definitions. *Figure 7-28* shows metaclasses for structured behaviour definition. An interaction type (instance of *InteractionType*) consists of two or more interaction participations (instance of *InteractionParticipation*). An interaction participation represents the participation of a behaviour (identified by *InteractionParticipation.participant*) and an interaction contribution of that behaviour in an interaction type (identified by *InteractionParticipation.contribution*).

Figure 7-28 Structure behaviour

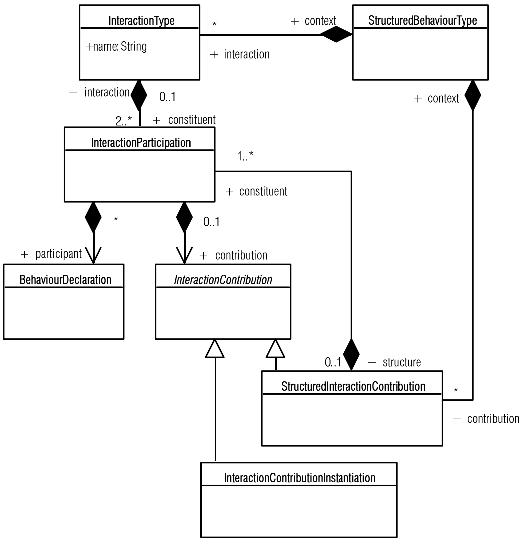


Figure 7-29 shows an example of a structured behaviour (of name *Composition*), which consists of five behaviour instantiations (of names *c1*, *c2*, *c3*, *s1* and *s2*) of two behaviour types (of names *ClientBehaviour* and *ServerBehaviour*). An interaction contribution is represented by a semi-circle drawn on the border of the behaviour in the context of which it is defined. An interaction is represented as lines that connect the interaction contributions that form the interaction. We have encircled with dashed lines three pairs of interaction contributions which form three interactions (between *c1* and *s1*, *c2* and *s1*, and, *c3* and *s1*).

Figure 7-29 Example of structure behaviour

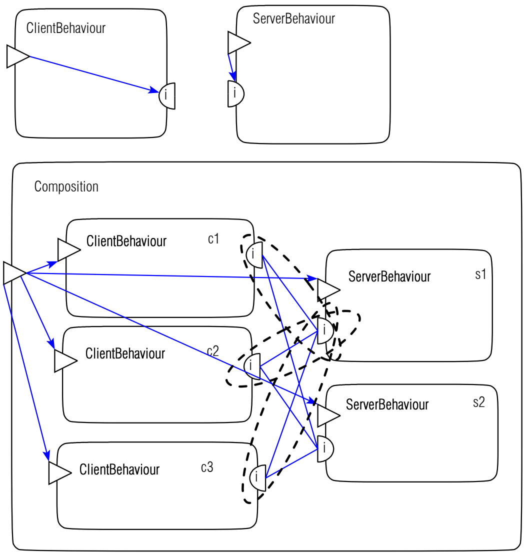


Figure 7-30 shows the role of constraints on location attributes to establish which behaviours are allowed to interact with each other. A constraint of an interaction contribution is drawn on a box attached to the interaction contribution. We use a composite location type (*Location*), which consists of two service endpoints (*ServiceEndpoint*). For describing constraints, we use OCL, which is more expressive than the constraint language we have defined for the service specification level. In this example, *c1* only interacts with *s1*, *c2* only interacts with *s1* and *c3* only interacts with *s2*.

Figure 7-30 Example of use of location attributes

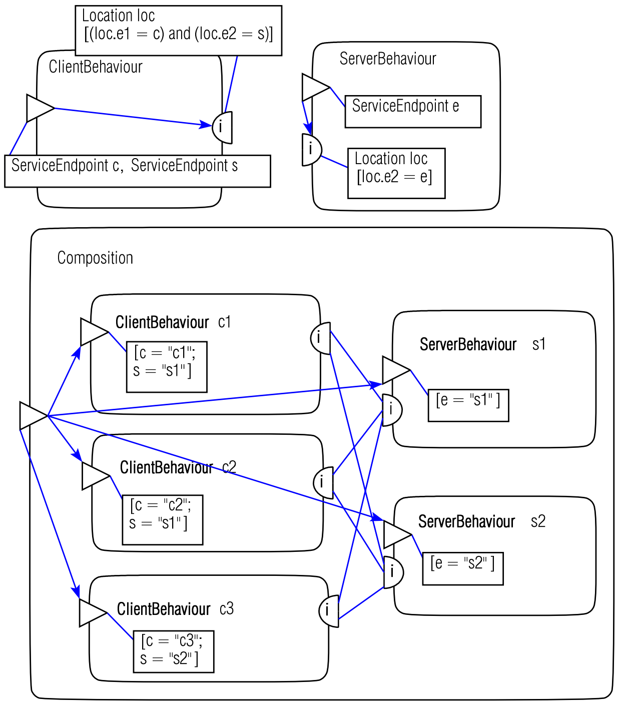
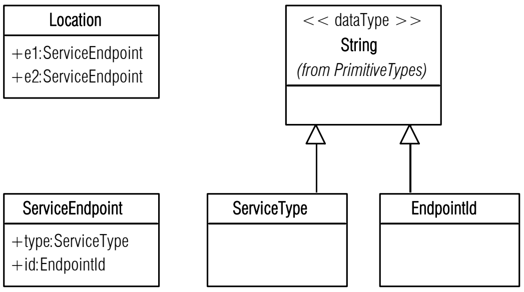


Figure 7-31 shows the UML class diagrams that define the location attribute type *Location*, which is used at the platform-independent service design level.

Figure 7-31 Location attribute type class diagram



In the next section, we use service endpoints to identify service offers in the service trader.

7.4.3 Service discovery abstract platform

In order to allow for service discovery, the service discovery abstract platform introduces a trader service. The trader registers a number of service offers. Service offers (instances of *ServiceOffer*) are information attributes, exchanged with the trader in an *export* interaction. Service offers include a service endpoint (*ServiceOffer.serviceEndpoint*) and a number of service properties (*ServiceProperty*). Service properties may be either static or dynamic. Static properties have immutable values, while dynamic properties have values that change at runtime. Each static service property consists of a name-value pair. Each dynamic service property consists of a service endpoint (*DynamicServiceProperty.serviceEndpoint*) and a service property type (*DynamicServiceProperty.datatype*). The service endpoint associated to a dynamic service property is used by the trader to request the current value of the dynamic property. The service property type identifies the type of the dynamic property. The classes relevant to service offers are depicted in *Figure 7-32*.

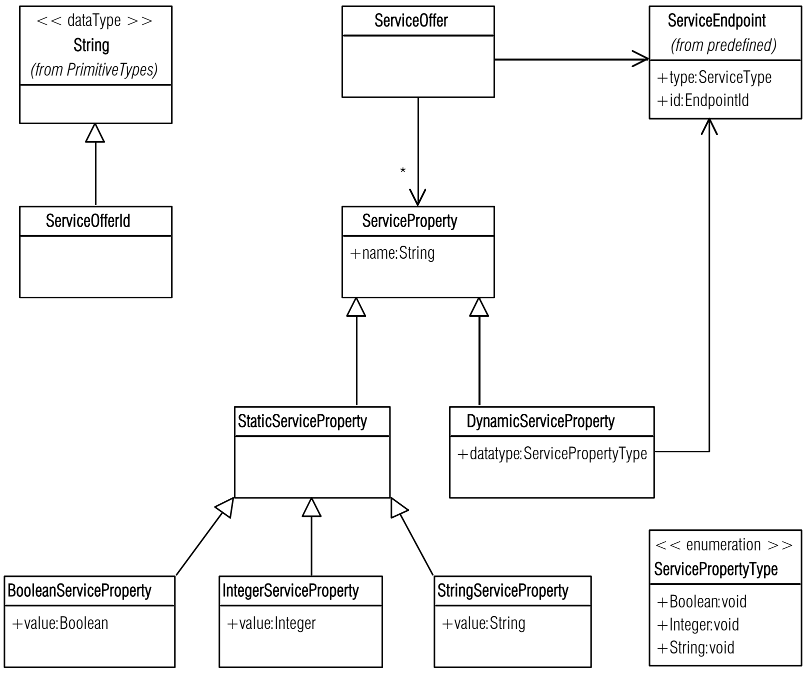


Figure 7-32 Service offers

A client of the trader service specifies a service query by providing a service type (*ServiceType*) and an expression (*ServiceQueryExpression*) involving service properties (*ServiceProperty*), which are referred to by their names (in a "leaf" expression *ServicePropertyExpression*). *Figure 7-33* shows the model that defines the *Expression* information attribute type. This model is similar to the metamodel of the ECA-DL constraint language, but should not to be mistaken for a metamodel. This model is part of the abstract platform at the model level.

Figure 7-33 Expressions for service queries

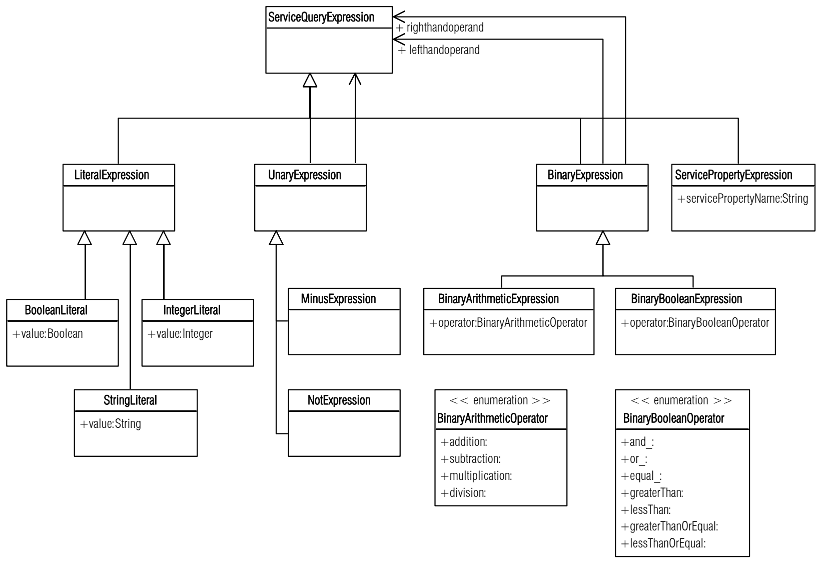


Figure 7-34 shows the ISDL specification of the service trader. The details of the relations between the interactions are omitted. A complete specification of the service trader is provided in Appendix B.

Figure 7-34 Trader service

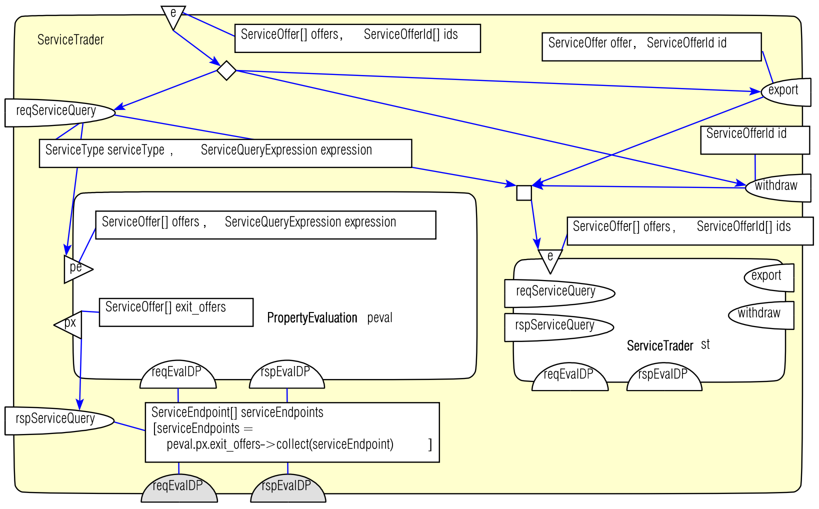

ServiceTrader
e
ServiceOffer[] offers, ServiceOfferId[] ids
ServiceOffer offer, ServiceOfferId id
export
reqServiceQuery
ServiceType serviceType , ServiceQueryExpression expression
ServiceOfferId id
withdraw
ServiceOffer[] offers , ServiceQueryExpression expression
ServiceOffer[] offers , ServiceOfferId[] ids
pe
px
ServiceOffer[] exit_offers
PropertyEvaluation peval
reqEvalDP
rspEvalDP
e
reqServiceQuery
rspServiceQuery
export
withdraw
ServiceTrader st
reqEvalDP
rspEvalDP
rspServiceQuery
ServiceEndpoint[] serviceEndpoints
[serviceEndpoints =
peval.px.exit_offers->collect(serviceEndpoint)]
reqEvalDP
rspEvalDP

Table 7-2 describes the interactions and the information attributes established.

Table 7-2 Interactions and information attributes for the trader service

Interaction	Information attributes
reqServiceQuery – the trader service is queried for a service	*ServiceType* – the type of service being requested *ServiceQueryExpression* – an expression involving service properties
rspServiceQuery – the trader service responds to a service query	*ServiceEndpoint[]* – a sequence of service endpoints, which is a result of the query
export – a service offer is published in the trader service	*ServiceOffer* – a service offer, *ServiceOfferId* – the identification of the offer
withdraw – a service offer is removed from the trader service	*ServiceOfferId* – the identification of the offer for its removal
reqEvalDP – the service trader requests a dynamic property to be evaluated	none
rspEvalDP – the current value of the property is sent to the trader	*Boolean, Integer* or *String* – the value of the dynamic property, depending on its type

7.4.4 A-MUSE abstract platform

The A-MUSE Service Platform offers context and action services which can be composed with service components. Since they are part of the abstract platform, these context and action services should be general-purpose within the application domain considered. Context services in the domain of mobile applications include (device) positioning and availability services. These services can be provided by the mobile telecommunications network. Action services in this domain include messaging services (such as Short-Message Services or SMS). In addition to these general-purpose services, we have included a number of domain-specific services required for the telemonitoring health application that is considered in this case study: a seizure detection service, which informs when a patient is about to suffer an epileptic seizure, and a number of alert services (for the patient, health care team and aid persons). The decision to include these domain-specific services contributes to limiting the size of this case study. However, we acknowledge that a general solution to coping with domain-specific services would be to allow service designers to define their own service-specific context and action services.

Figure 7-35 shows the ISDL specification of the context services in the A-MUSE platform: *EventBasedSeizureService*, *PositionService* and *AvailabilityService*. The specification abstracts from the environment, i.e., users and their devices.

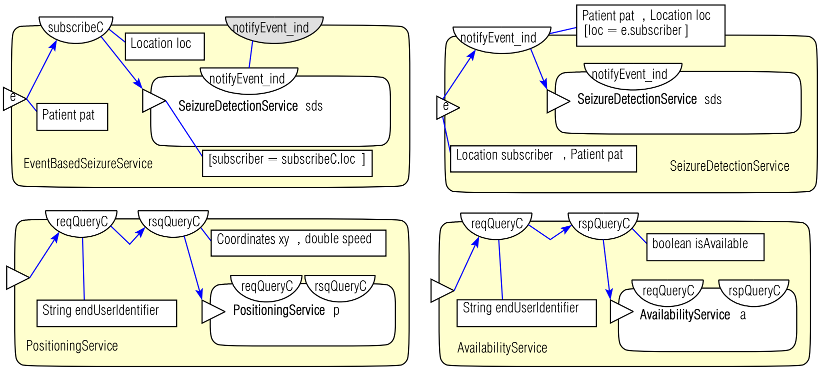


Figure 7-35 Context services

These services are offered by entities which are part of the abstract platform. *PositioningService* and *AvailabilityService* are defined as singletons, i.e., are offered by a single entity in the system. They can provide the location and availability of any user relevant to the Freeband service. Modelling these services as singletons simplifies the management of the entities that provide these services. In constrast with *PositioningService* and *AvailabilityService*, *EventBasedSeizureService* is not a singleton, and is offered by several entities in the system, namely, one entity for each patient being monitored for seizures. This simplifies the specification of the "subscription" scheme between an entity that offers this service and its users.

Figure 7-36 shows the ISDL specification of the action services in the A-MUSE platform: *PatientAlertService*, *HealthTeamAlertSeizure*, *AidPersonAlertService* and *SendSMSService*.

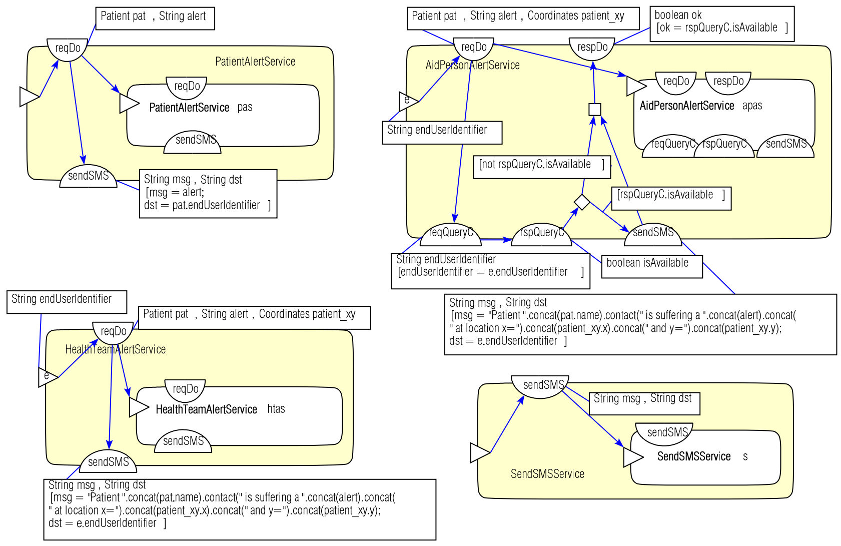

Patient pat , String alert
reqDo
PatientAlertService
reqDo
PatientAlertService pas
sendSMS
sendSMS
String msg , String dst
[msg = alert;
dst = pat.endUserIdentifier]
Patient pat , String alert , Coordinates patient_xy
boolean ok
[ok = rspQueryC.isAvailable]
reqDo
respDo
AidPersonAlertService
reqDo
respDo
AidPersonAlertService apas
e
String endUserIdentifier
reqQueryC
rspQueryC
sendSMS
[not rspQueryC.isAvailable]
[rspQueryC.isAvailable]
reqQueryC
rspQueryC
sendSMS
String endUserIdentifier
[endUserIdentifier = e.endUserIdentifier]
boolean isAvailable
String msg , String dst
[msg = "Patient ".concat(pat.name).contact(" is suffering a ".concat(alert).concat(
" at location x=").concat(patient_xy.x).concat(" and y=").concat(patient_xy.y);
dst = e.endUserIdentifier]
String endUserIdentifier
Patient pat , String alert , Coordinates patient_xy
reqDo
HealthTeamAlertService
e
reqDo
HealthTeamAlertService htas
sendSMS
sendSMS
String msg , String dst
[msg = "Patient ".concat(pat.name).contact(" is suffering a ".concat(alert).concat(
" at location x=").concat(patient_xy.x).concat(" and y=").concat(patient_xy.y);
dst = e.endUserIdentifier]
sendSMS
String msg , String dst
sendSMS
SendSMSService s
SendSMSService

Figure 7-36 Action services

SendSMSService and *PatientAlertService* are singletons and can send messages and alerts to any terminal user and patient, respectively. In constrast, an entity that provides the *HealthTeamAlertService* is available for each health team, and an entity that provides the *AidPersonAlertService* is available for each aid person. These services are not singletons so that each different entity providing these services can be registered as a service offer in the service trader with different properties.

All endpoints that offer context and action services are registered in the service trader, so that service components can find them and interact with them. Service components use service properties in queries that are sent to the trader to select a suitable service offer. *Table 7-1* shows the services registered in the A-MUSE abstract platform and their service properties.

Table 7-3 Services in the A-MUSE abstract platform

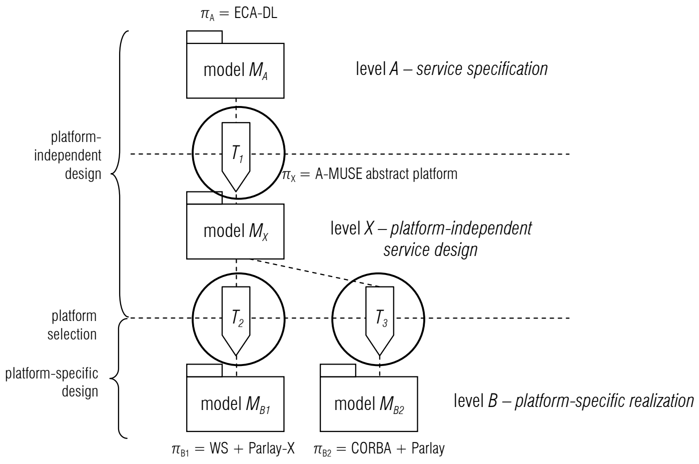
Services	**Singleton?**	**Service properties**
PositioningService	Yes	-
AvailabilityService	Yes	-
EventBasedSeizureService	No, one entity for each patient monitored	-
SendSMSService	Yes	-
PatientAlertService	Yes	-
AidPersonAlertService	No, one entity for each aid person	Dynamic properties: geoLocation_x : Integer geoLocation_y : Integer
HealthTeamAlertService	No, one entity for each health team	Static properties: coverageArea_geoLocation_x : Integer coverageArea_geoLocation_y : Integer range : Integer

An offer of *AidPersonAlertService* allows a user of the service to contact a particular aid person. The dynamic properties of an offer of *AidPersonAlertService* refer to the current geographical location of the aid person. These coordinates change when aid persons move and are, therefore, dynamic properties. These coordinates can be used to select a service offer based on the aid person's location. The properties of an offer of *HealthTeamAlertService* are static, and refer to the coordinates of the location from which a health team is dispatched (e.g., a hospital), and the operating range, i.e., the maximum distance a health team may travel to support a patient.

7.5 Transformations

In this section, we demonstrate buildability by defining transformations for the service specification- and platform-independent service design levels. *Figure 7-37* depicts these transformations. Transformation T_1 is discussed in section 7.5.1 and T_2 and T_3 are discussed in section 7.5.2.

Figure 7-37
Transformations


π_A = ECA-DL
model M_A
level *A – service specification*
platform-independent design
T_1
π_X = A-MUSE abstract platform
model M_X
level *X – platform-independent service design*
platform selection
T_2
T_3
platform-specific design
model M_{B1}
model M_{B2}
level *B – platform-specific realization*
π_{B1} = WS + Parlay-X
π_{B2} = CORBA + Parlay

7.5.1 From service specification to platform-independent service design

The transformation from the service specification level to the platform-independent service design level results in a composition of a coordination component and the A-MUSE abstract platform.

ECA-DL actions at the service specification level are refined into sequences of interactions in the service design. While at the service specification level an action represents an activity performed by the Telemonitoring system as a whole (including any context sources and action services), at the service design level the same action has to be performed by cooperation of different services, revealing the trader service, and the various context and action services. *Figure 7-38* illustrates schematically the refinement of actions in the service specification level.

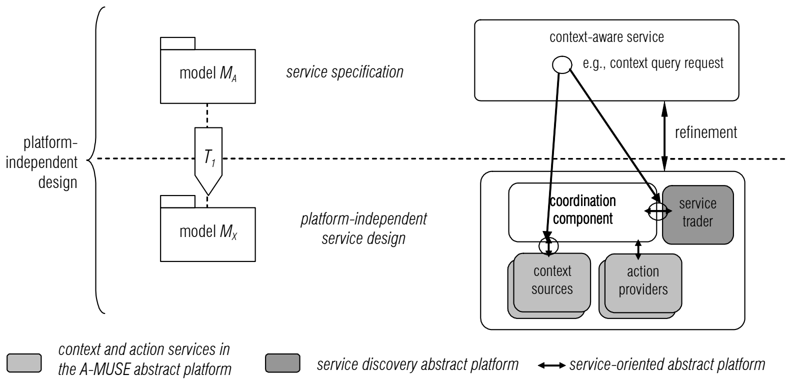

model M_A
service specification
context-aware service
e.g., context query request
refinement
platform-independent design
T_1
model M_X
platform-independent service design
coordination component
service trader
context sources
action providers
context and action services in the A-MUSE abstract platform
service discovery abstract platform
service-oriented abstract platform

Figure 7-38 From service specification to platform-independent service design

Action invocation request and response

Each ECA-DL action of type *ActionInvocationRequestType* (denoted with suffix *_reqA*) corresponds to a sequence of three interactions in the service design: a request to the service trader, a response from the service trader and the invocation of the appropriate action service according to the response issued by the service trader.

Figure 7-39 shows, informally, the transformation of an action invocation request. The corresponding behaviours on the source and target levels are depicted at the top and bottom sides of the picture, respectively. Behaviour blocks are used to help in the visualization, and are not actually part of the transformation.

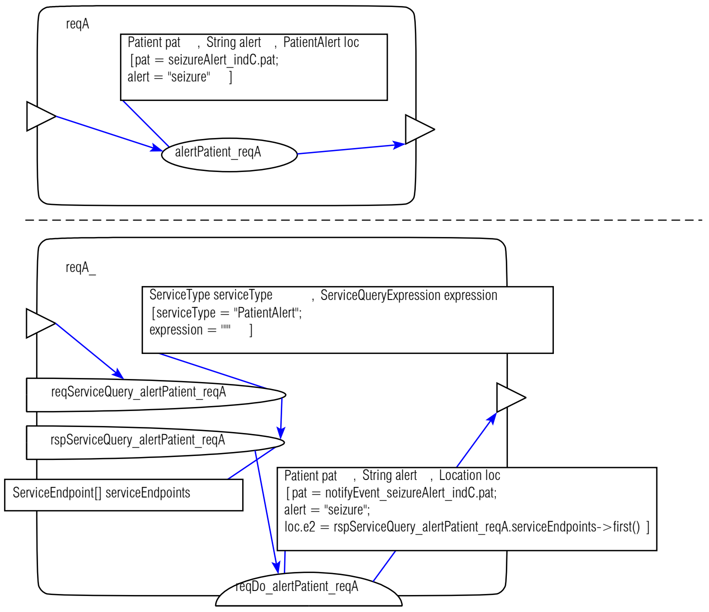

reqA
Patient pat , String alert , PatientAlert loc
[pat = seizureAlert_indC.pat;
alert = "seizure"]
alertPatient_reqA
reqA_
ServiceType serviceType , ServiceQueryExpression expression
[serviceType = "PatientAlert";
expression = ""]
reqServiceQuery_alertPatient_reqA
rspServiceQuery_alertPatient_reqA
ServiceEndpoint[] serviceEndpoints
Patient pat , String alert , Location loc
[pat = notifyEvent_seizureAlert_indC.pat;
alert = "seizure";
loc.e2 = rspServiceQuery_alertPatient_reqA.serviceEndpoints->first()]
reqDo_alertPatient_reqA

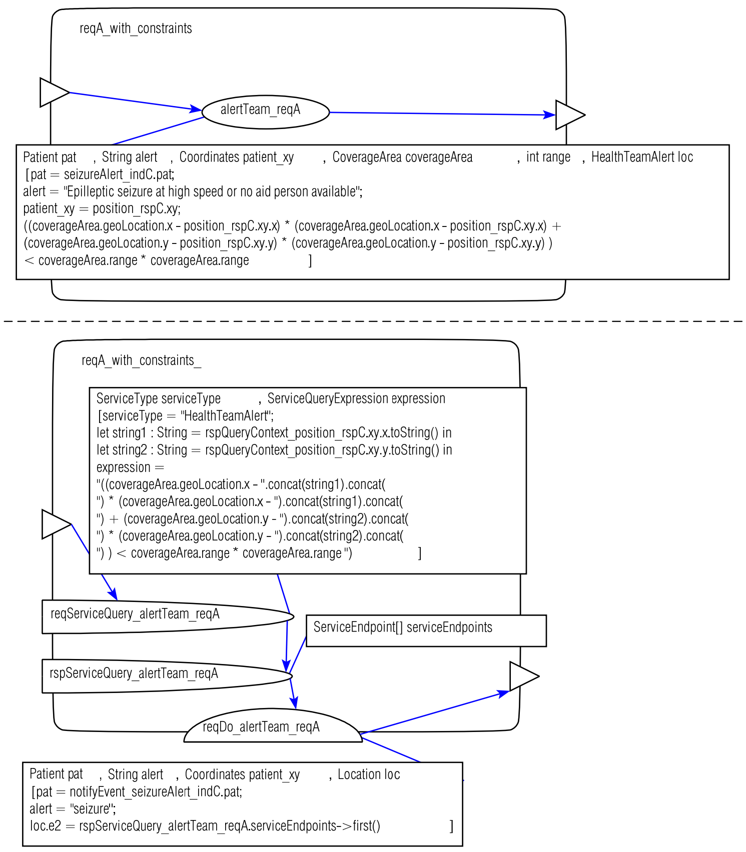
Figure 7-39 Transformation of an action invocation request

The reference to *seizureAlert_indC* in the constraints must be replaced by a reference to the final action that corresponds to *seizureAlert_indC* (*notifyEvent_seizureAlert_indC*), since this original action must also be refined in the transformation.

The service type in the service query is derived from the specific type of *Locus* for the action. In the example in *Figure 7-39*, the required service type is *"PatientAlert"*. Expressions on service properties in the query to the service trader are derived from information attributes and their constraints at the service specification level. This derivation requires marking of the service specification to indicate which information attributes should be used in the service query. In *Figure 7-39*, no information attributes are marked, and hence the query is empty. In this transformation, we define that the first service offer returned by the trader is used for the action invocation request. The information attributes for the request are the same as those for the original action.

Figure 7-40 shows informally the transformation of an action invocation which includes information attributes to be used as service properties.

Figure 7-40
Transformation of an action invocation request with constraints


reqA_with_constraints
alertTeam_reqA
reqA_with_constraints_
reqServiceQuery_alertTeam_reqA
rspServiceQuery_alertTeam_reqA
ServiceEndpoint[] serviceEndpoints
reqDo_alertTeam_reqA

```
Patient pat   , String alert   , Coordinates patient_xy       , CoverageArea coverageArea          , int range   , HealthTeamAlert loc
 [pat = seizureAlert_indC.pat;
alert = "Epilleptic seizure at high speed or no aid person available";
patient_xy = position_rspC.xy;
((coverageArea.geoLocation.x - position_rspC.xy.x) * (coverageArea.geoLocation.x - position_rspC.xy.x) +
(coverageArea.geoLocation.y - position_rspC.xy.y) * (coverageArea.geoLocation.y - position_rspC.xy.y) )
< coverageArea.range * coverageArea.range                ]
```

```
ServiceType serviceType          , ServiceQueryExpression expression
 [serviceType = "HealthTeamAlert";
let string1 : String = rspQueryContext_position_rspC.xy.x.toString() in
let string2 : String = rspQueryContext_position_rspC.xy.y.toString() in
expression =
"((coverageArea.geoLocation.x - ".concat(string1).concat(
") * (coverageArea.geoLocation.x - ").concat(string1).concat(
") + (coverageArea.geoLocation.y - ").concat(string2).concat(
") * (coverageArea.geoLocation.y - ").concat(string2).concat(
") ) < coverageArea.range * coverageArea.range ")                 ]
```

```
Patient pat   , String alert   , Coordinates patient_xy       , Location loc
 [pat = notifyEvent_seizureAlert_indC.pat;
alert = "seizure";
loc.e2 = rspServiceQuery_alertTeam_reqA.serviceEndpoints->first()             ]
```

This transformation requires marking of *coverageArea* as input for query expression. The query expression is given in its text format, for the sake of readability. Appendix B shows how these expressions are defined in OCL. The reference to *position_rspC* is replaced by a reference to the final action that corresponds to *position_rspC* (*rspQueryContext_position_rspC*).
Figure 7-41 shows informally the transformation of the pattern of action invocation request and action invocation response. This transformation requires marking of *aidperson_xy* as input for query expression.

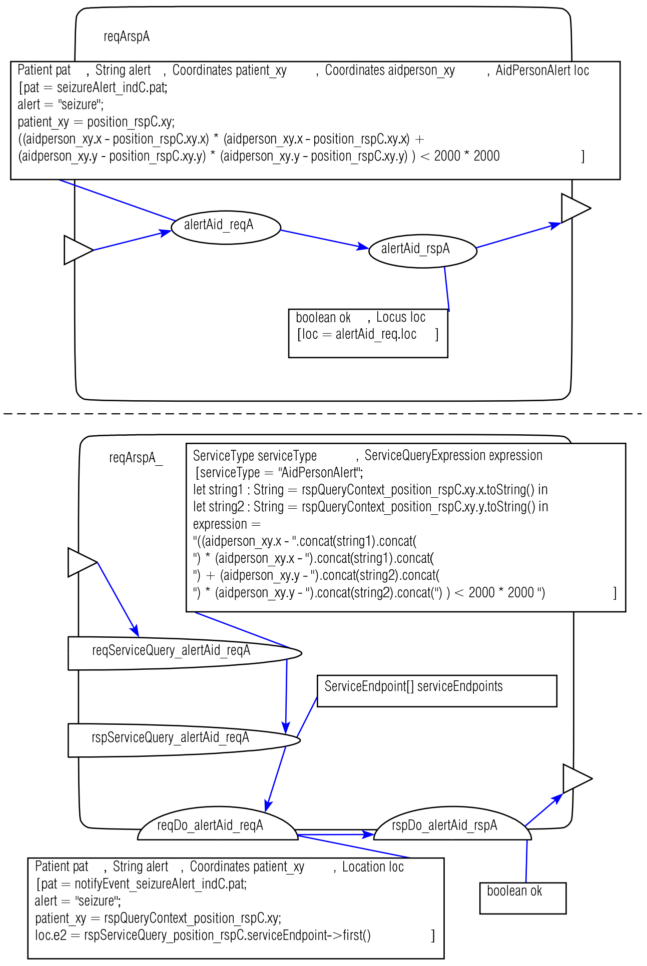

reqArspA
Patient pat , String alert , Coordinates patient_xy , Coordinates aidperson_xy , AidPersonAlert loc
[pat = seizureAlert_indC.pat;
alert = "seizure";
patient_xy = position_rspC.xy;
((aidperson_xy.x - position_rspC.xy.x) * (aidperson_xy.x - position_rspC.xy.x) +
(aidperson_xy.y - position_rspC.xy.y) * (aidperson_xy.y - position_rspC.xy.y)) < 2000 * 2000]
alertAid_reqA
alertAid_rspA
boolean ok , Locus loc
[loc = alertAid_req.loc]
reqArspA_
ServiceType serviceType , ServiceQueryExpression expression
[serviceType = "AidPersonAlert";
let string1 : String = rspQueryContext_position_rspC.xy.x.toString() in
let string2 : String = rspQueryContext_position_rspC.xy.y.toString() in
expression =
"((aidperson_xy.x - ".concat(string1).concat(
") * (aidperson_xy.x - ").concat(string1).concat(
") + (aidperson_xy.y - ").concat(string2).concat(
") * (aidperson_xy.y - ").concat(string2).concat(")) < 2000 * 2000 ")]
reqServiceQuery_alertAid_reqA
ServiceEndpoint[] serviceEndpoints
rspServiceQuery_alertAid_reqA
reqDo_alertAid_reqA
rspDo_alertAid_rspA
Patient pat , String alert , Coordinates patient_xy , Location loc
[pat = notifyEvent_seizureAlert_indC.pat;
alert = "seizure";
patient_xy = rspQueryContext_position_rspC.xy;
loc.e2 = rspServiceQuery_position_rspC.serviceEndpoint->first()]
boolean ok

Figure 7-41
Transformation of pattern of action invocation request and action invocation response

Context query request and response

The transformation of the pattern of context query request (*ContextRequest-Type*) and context query response (*ContextResponseType*) is similar to that of action invocation request and responses. *Figure 7-42* shows this transformation informally.

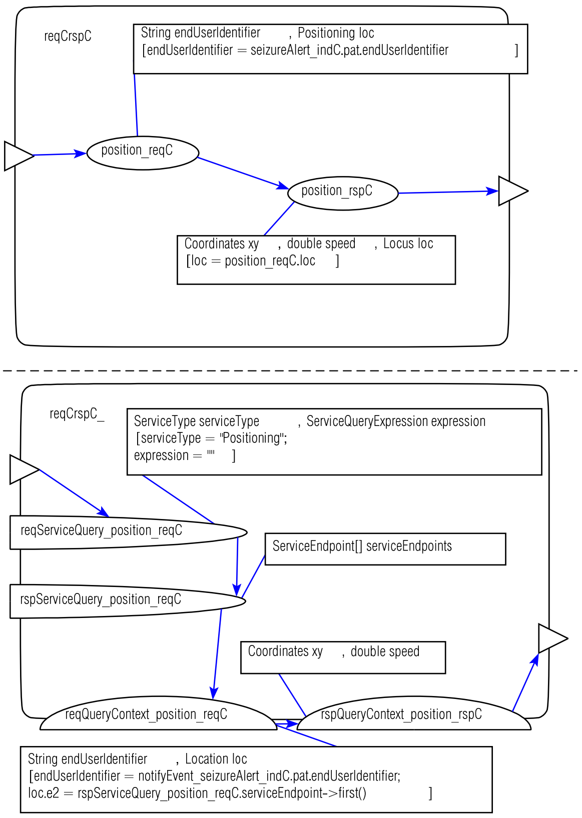

reqCrspC
String endUserIdentifier , Positioning loc
[endUserIdentifier = seizureAlert_indC.pat.endUserIdentifier]
position_reqC
position_rspC
Coordinates xy , double speed , Locus loc
[loc = position_reqC.loc]
reqCrspC_
ServiceType serviceType , ServiceQueryExpression expression
[serviceType = "Positioning";
expression = ""]
reqServiceQuery_position_reqC
ServiceEndpoint[] serviceEndpoints
rspServiceQuery_position_reqC
Coordinates xy , double speed
reqQueryContext_position_reqC
rspQueryContext_position_rspC
String endUserIdentifier , Location loc
[endUserIdentifier = notifyEvent_seizureAlert_indC.pat.endUserIdentifier;
loc.e2 = rspServiceQuery_position_reqC.serviceEndpoint->first()]

Figure 7-42
Transformation of pattern of context query request and response

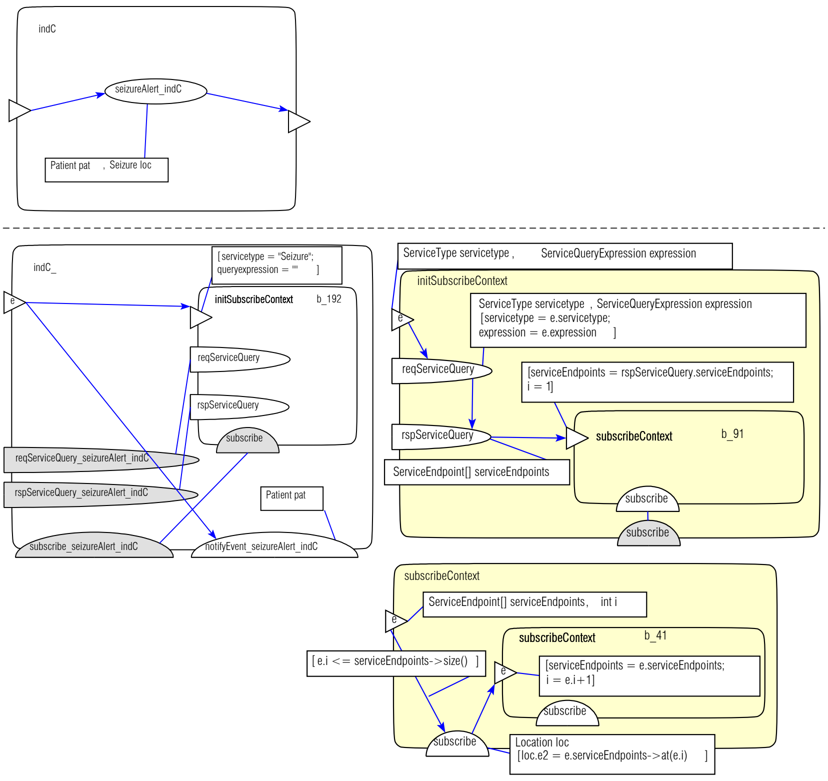
Causality constraints

The transformation of causality constraints is rather intuitive when described in terms of the notation: arrows pointing to an action in the source design should point to the first inserted interaction in the target design, and arrows pointing from an action in the source design should point from the final action in the target design. The constraints are enforced by the coordination component.

Context event

Figure 7-43 shows informally the transformation of a context event. Each context event is transformed into an initialization behaviour and a final interaction between the coordination component and a context source. The initialization behaviour consists of subscribing to all context service offers returned by the trader service. This differs from the transformations we have seen so far that use only the first result of the service query. This is necessary because we are interested in any context sources whose service type matches the location type for the original action. Subscribing to context sources is required for future notification of the occurrence of context events. The initialization behaviour is not subject to the same causality conditions as the original action, and maybe performed in advance, as long as results required for the service query are available. The behaviours *initSubscribeContext* and *subscribeContext* are generic and can be reused for other context events.

Figure 7-43 Transformation of a context event


indC
seizureAlert_indC
Patient pat , Seizure loc
indC_
[servicetype = "Seizure"; queryexpression = ""]
initSubscribeContext b_192
e
reqServiceQuery
rspServiceQuery
subscribe
reqServiceQuery_seizureAlert_indC
rspServiceQuery_seizureAlert_indC
Patient pat
subscribe_seizureAlert_indC
notifyEvent_seizureAlert_indC
ServiceType servicetype , ServiceQueryExpression expression
initSubscribeContext
ServiceType servicetype , ServiceQueryExpression expression [servicetype = e.servicetype; expression = e.expression]
[serviceEndpoints = rspServiceQuery.serviceEndpoints; i = 1]
subscribeContext b_91
ServiceEndpoint[] serviceEndpoints
subscribe
subscribeContext
ServiceEndpoint[] serviceEndpoints, int i
subscribeContext b_41
[serviceEndpoints = e.serviceEndpoints; i = e.i+1]
[e.i <= serviceEndpoints->size()]
Location loc [loc.e2 = e.serviceEndpoints->at(e.i)]

The entry points in the service specification are replicated in the coordination component, as well as any recursive behaviour instantiation.

Conformance

The transformation described above results in service designs which conform to the source service specification under the following assumptions: (i) the service trader is always able to produce a service offer for a service query, (ii) context sources always reply to context query requests, and (iii) action services always reply to action invocation requests (in case action invocation request and action invocation response is used in a pattern). Assumption (i) can be guaranteed by availability of service offers in the service trader. Assumptions (ii) and (iii) can be verified in the design of context sources and action services.

These assumptions are necessary to integrate the interaction contributions in the target design into actions and then apply the conformance assessment method described in [89]. This assessment method requires the identification of inserted and final actions for the refinement (as we have discussed in chapter 5). *Table 7-4* shows, for each original action type, the inserted and final interactions in the target design.

Table 7-4 Original actions and the corresponding inserted and final actions

Original action type	Inserted interactions	Final interaction
ActionInvocationRequestType (<name>_reqA)	*reqServiceQuery_<name>_reqA* *rspServiceQuery_<name>_reqA*	*reqDo_<name>_reqA*
ActionInvocationResponseType (<name>_rspA)	none	*rspDo_<name>_reqA*
ContextRequestType (<name>_reqC)	*reqServiceQuery_<name>_reqC* *rspServiceQuery_<name>_reqC*	*reqDo_<name>_reqC*
ContextResponseType (<name>_rspC)	none	*rspDo_<name>_reqC*
ContextEventType (<name>_indC)	initialization behaviour, including: *reqServiceQuery_<name>_indC* *rspServiceQuery_<name>_indC* *subscribe_<name>_indC* (possibly many occurrences)	*notifyEvent_<name>_indC*

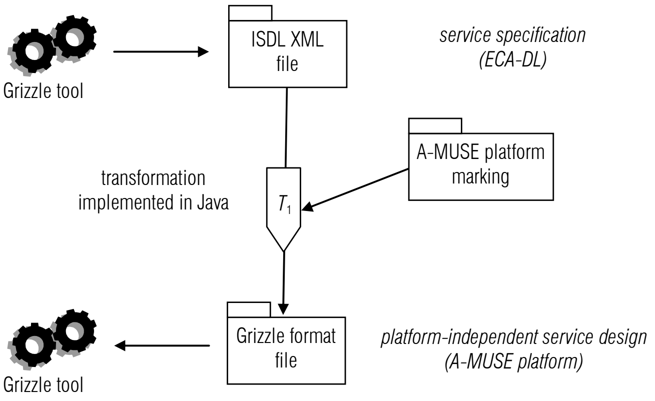
The information attributes of the final interactions correspond to information attributes of the original action by construction, since these attributes (and constraints on them) are copied during transformation. The information attributes in the source design that are marked to be used in the service query have no corresponding information attribute at the target design; however, the constraints on these information attributes are captured in the service query, and hence the location (service endpoint) where the interaction occurs respects these constraints.

Implementation

Currently, ISDL models in Grizzle are not stored in a model repository. They can be exported in an XML format defined in an ISDL XML schema [53] or stored in a proprietary Grizzle textual file format. Models can only be imported in the tool in the Grizzle textual file format.

In order to use generic model transformation tools with Grizzle, it would be necessary to populate a model repository based on the contents of the exported ISDL XML, apply the transformation based on a model transformation tool and then generate Grizzle file format from the target models. Instead of using a model transformation tool, we have opted for implementing transformation T_1 in the Java programming language. A Java program reads ISDL XML and generates an output file in the Grizzle textual file format, as shown in *Figure 5-2*. The parameterization of the transformation is done through a simple XML file format, which lists the names of the information attributes which are used as service properties in the target model. The information attributes are qualified through their action names.

Figure 7-44
Implementation of transformation T_1


Grizzle tool
ISDL XML file
service specification (ECA-DL)
A-MUSE platform marking
transformation implemented in Java
T_1
Grizzle format file
platform-independent service design (A-MUSE platform)
Grizzle tool

The Grizzle tool is currently being redesigned and its implementation will be based on the Eclipse Modelling Framework (EMF) [36]. This will facilitate the use of generic transformation tools, since the models will be directly stored in a model repository which model transformation tools can access, avoiding format conversions.

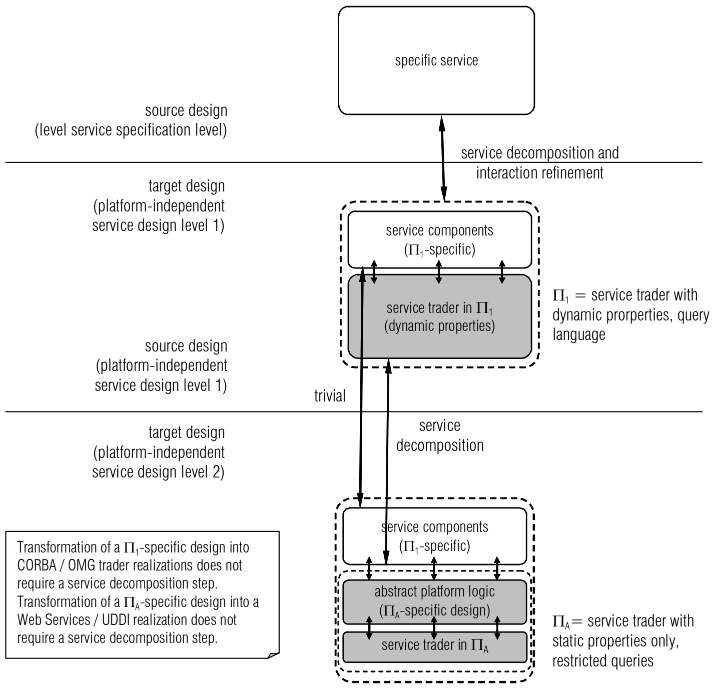
7.5.2 From platform-independent to platform-specific service design

In order to show the flexibility of the relation between the platform-independent service design level and the platform-specific service design we describe in this section a possible transformation of platform-independent service designs into two different middleware platforms, namely, Web Services and CORBA. These platforms differ significantly with respect to their support for service discovery.

CORBA provides a trader that supplies a constraint language which can accommodate the constraints that can be defined with the *ServiceQueryExpression* information attribute type defined in *Figure 7-33*. In addition, it supports dynamic service properties.

In the case of Web Services technologies, service discovery is provided by UDDI. UDDI does not support dynamic service properties and supports no query language, being able only to provide the values of static service properties (*tModels* [71]) to its clients.

A realization of the trader service in CORBA is rather straightforward and does not require service decomposition. A realization of the trader service in UDDI is more complex due to the differences in the support provided by UDDI and the trader service as specified in the abstract platform. We approach this by introducing a service decomposition step prior to realization. The two approaches to platform-specific realization are shown in *Figure 5-10*. In the case of the CORBA realization, only platform-independent service design level 1 is used. In the case of the Web Services/UDDI realization, both platform-independent service design levels 1 and 2 are used.



Transformation of a Π_1-specific design into CORBA / OMG trader realizations does not require a service decomposition step. Transformation of a Π_A-specific design into a Web Services / UDDI realization does not require a service decomposition step.

Figure 7-45 Realization of the service discovery platform into two different platforms

The abstract platform logic must bridge the gap between the trader service at the abstract platform and the service provided by a UDDI registry. Each service offer is registered as an entry in the UDDI registry. Given a query, the abstract platform logic uses the UDDI registry to retrieve all entries for a particular service type, evaluates the expressions (which may include dynamic property evaluation) and returns the list of service offers for which expressions evaluate to *true*. In order to support dynamic service properties, Web service endpoints that are used to evaluate dynamic properties must be registered as an additional *tModel*, which is present only for dynamic service properties.

7.6 Execution phase

In this section, we show the results of the execution phase. The shaded icons in *Figure 7-46* indicates the phase and activities performed and the results obtained.

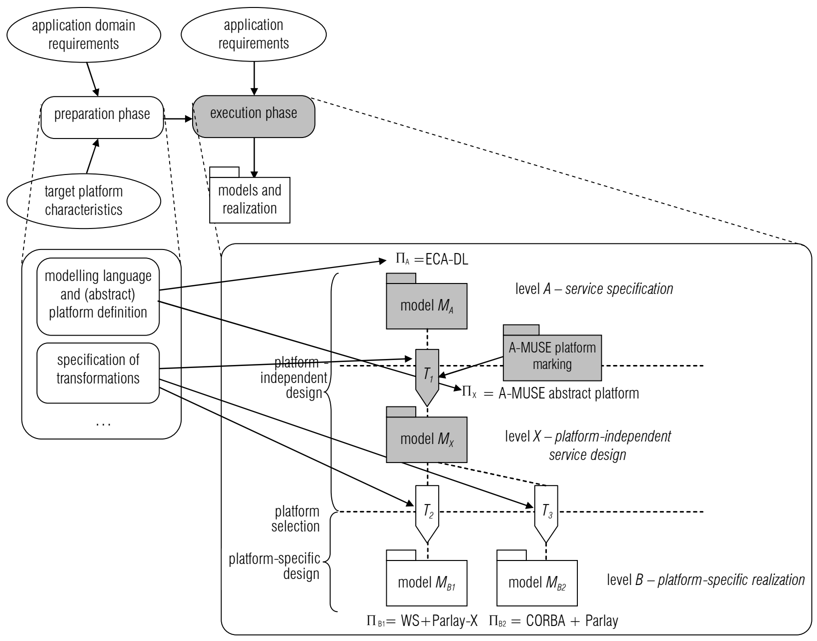

application domain requirements
application requirements
preparation phase
execution phase
target platform characteristics
models and realization
modelling language and (abstract) platform definition
specification of transformations
...
Π_A =ECA-DL
model M_A
level A – service specification
A-MUSE platform marking
T_1
Π_X = A-MUSE abstract platform
platform-independent design
model M_X
level X – platform-independent service design
platform selection
T_2
T_3
platform-specific design
model M_B1
model M_B2
level B – platform-specific realization
Π_B1= WS+Parlay-X Π_B2 = CORBA + Parlay

Figure 7-46 The execution phase and its results

7.6.1 Service specification

The Telemonitoring Service specification as defined in section 7.3.3 is depicted in *Figure 7-47*. The information attributes that have to be transformed into trader service properties are marked with dashed boxes.

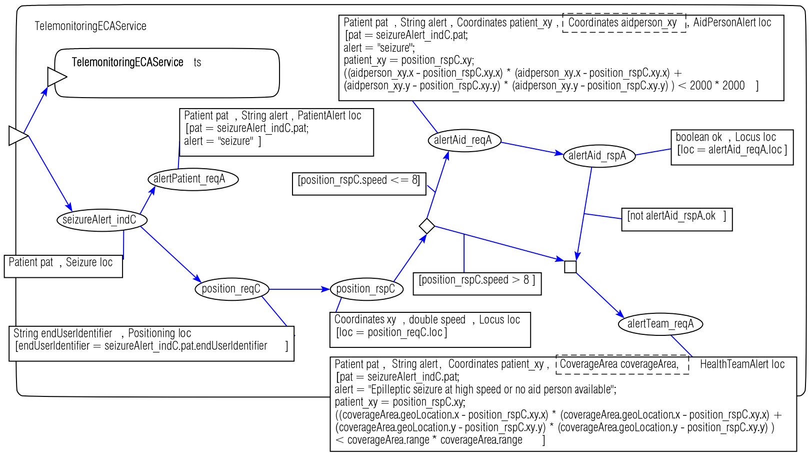

TelemonitoringECAService
TelemonitoringECAService ts
Patient pat , String alert , Coordinates patient_xy , Coordinates aidperson_xy , AidPersonAlert loc
[pat = seizureAlert_indC.pat;
alert = "seizure";
patient_xy = position_rspC.xy;
((aidperson_xy.x - position_rspC.xy.x) * (aidperson_xy.x - position_rspC.xy.x) +
(aidperson_xy.y - position_rspC.xy.y) * (aidperson_xy.y - position_rspC.xy.y)) < 2000 * 2000]
Patient pat , String alert , PatientAlert loc
[pat = seizureAlert_indC.pat;
alert = "seizure"]
alertAid_reqA
alertAid_rspA
boolean ok , Locus loc
[loc = alertAid_reqA.loc]
alertPatient_reqA
[position_rspC.speed <= 8]
[not alertAid_rspA.ok]
seizureAlert_indC
Patient pat , Seizure loc
position_reqC
position_rspC
[position_rspC.speed > 8]
alertTeam_reqA
Coordinates xy , double speed , Locus loc
[loc = position_reqC.loc]
String endUserIdentifier , Positioning loc
[endUserIdentifier = seizureAlert_indC.pat.endUserIdentifier]
Patient pat , String alert , Coordinates patient_xy , CoverageArea coverageArea, HealthTeamAlert loc
[pat = seizureAlert_indC.pat;
alert = "Epileptic seizure at high speed or no aid person available";
patient_xy = position_rspC.xy;
((coverageArea.geoLocation.x - position_rspC.xy.x) * (coverageArea.geoLocation.x - position_rspC.xy.x) +
(coverageArea.geoLocation.y - position_rspC.xy.y) * (coverageArea.geoLocation.y - position_rspC.xy.y))
< coverageArea.range * coverageArea.range]

Figure 7-47 The Telemonitoring Service specification, with markings

7.6.2 Platform-independent service design

The platform-independent service design is the result of the application of T_1 to the service specification with its markings. The generated *TelemonitoringECAServiceCoordination* enforces the behaviour defined in the service specification level. This behaviour is illustrated in *Figure 7-48*. The dashed lines represent causality relations in the service specifications.

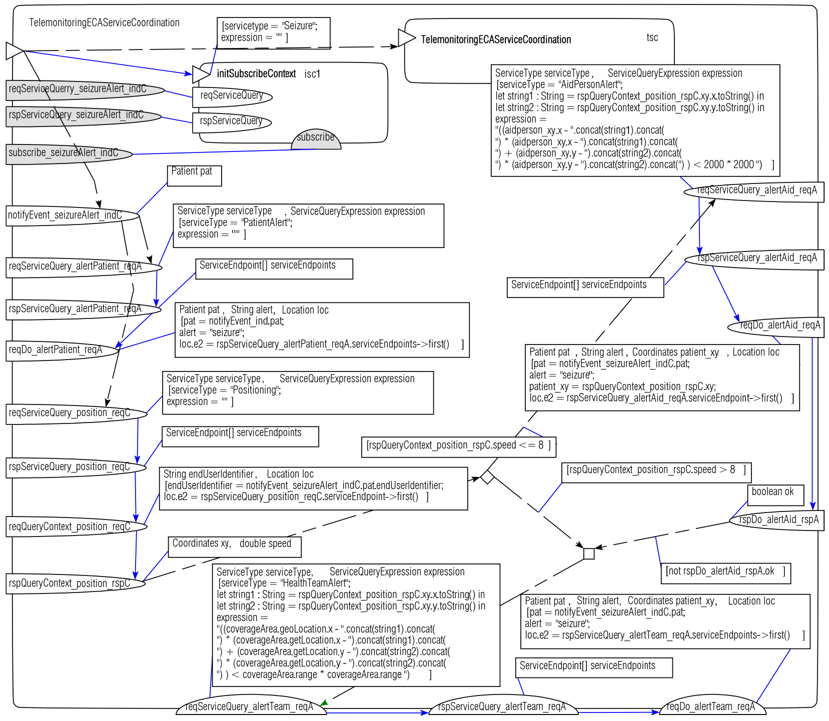

TelemonitoringECAServiceCoordination
[servicetype = "Seizure";
expression = ""]
TelemonitoringECAServiceCoordination tsc
initSubscribeContext isc1
reqServiceQuery
rspServiceQuery
subscribe
reqServiceQuerry_seizureAlert_indC
rspServiceQuery_seizureAlert_indC
subscribe_seizureAlert_indC
ServiceType serviceType , ServiceQueryExpression expression
[serviceType = "AidPersonAlert";
let string1 : String = rspQueryContext_position_rspC.xy.x.toString() in
let string2 : String = rspQueryContext_position_rspC.xy.y.toString() in
expression =
"((aidperson_xy.x - ".concat(string1).concat(
") * (aidperson_xy.x - ").concat(string1).concat(
") + (aidperson_xy.y - ").concat(string2).concat(
") * (aidperson_xy.y - ").concat(string2).concat(")) < 2000 * 2000")]
Patient pat
notifyEvent_seizureAlert_indC
ServiceType serviceType , ServiceQueryExpression expression
[serviceType = "PatientAlert";
expression = ""]
reqServiceQuery_alertAid_reqA
rspServiceQuery_alertAid_reqA
reqServiceQuery_alertPatient_reqA
ServiceEndpoint[] serviceEndpoints
rspServiceQuery_alertPatient_reqA
Patient pat , String alert, Location loc
[pat = notifyEvent_ind.pat;
alert = "seizure";
loc.e2 = rspServiceQuery_alertPatient_reqA.serviceEndpoints->first()]
ServiceEndpoint[] serviceEndpoints
reqDo_alertAid_reqA
reqDo_alertPatient_reqA
Patient pat , String alert , Coordinates patient_xy , Location loc
[pat = notifyEvent_seizureAlert_indC.pat;
alert = "seizure";
patient_xy = rspQueryContext_position_rspC.xy;
loc.e2 = rspServiceQuery_alertAid_reqA.serviceEndpoint->first()]
ServiceType serviceType , ServiceQueryExpression expression
[serviceType = "Positioning";
expression = ""]
reqServiceQuery_position_reqC
ServiceEndpoint[] serviceEndpoints
[rspQueryContext_position_rspC.speed <= 8]
rspServiceQuery_position_reqC
String endUserIdentifier , Location loc
[endUserIdentifier = notifyEvent_seizureAlert_indC.pat.endUserIdentifier;
loc.e2 = rspServiceQuery_position_reqC.serviceEndpoint->first()]
[rspQueryContext_position_rspC.speed > 8]
boolean ok
reqQueryContext_position_reqC
rspDo_alertAid_rspA
Coordinates xy , double speed
rspQueryContext_position_rspC
ServiceType serviceType , ServiceQueryExpression expression
[serviceType = "HealthTeamAlert";
let string1 : String = rspQueryContext_position_rspC.xy.x.toString() in
let string2 : String = rspQueryContext_position_rspC.xy.y.toString() in
expression =
"((coverageArea.geoLocation.x - ".concat(string1).concat(
") * (coverageArea.getLocation.x - ").concat(string1).concat(
") + (coverageArea.getLocation.y - ").concat(string2).concat(
") * (coverageArea.getLocation.y - ").concat(string2).concat(
")) < coverageArea.range * coverageArea.range ")]
[not rspDo_alertAid_rspA.ok]
Patient pat , String alert, Coordinates patient_xy , Location loc
[pat = notifyEvent_seizureAlert_indC.pat;
alert = "seizure";
loc.e2 = rspServiceQuery_alertTeam_reqA.serviceEndpoints->first()]
ServiceEndpoint[] serviceEndpoints
reqServiceQuery_alertTeam_reqA
rspServiceQuery_alertTeam_reqA
reqDo_alertTeam_reqA

Figure 7-48 Behaviour of the coordination component

Figure 7-49 shows the behaviour responsible for the initialization of the coordination component, which subscribes to the relevant context sources.

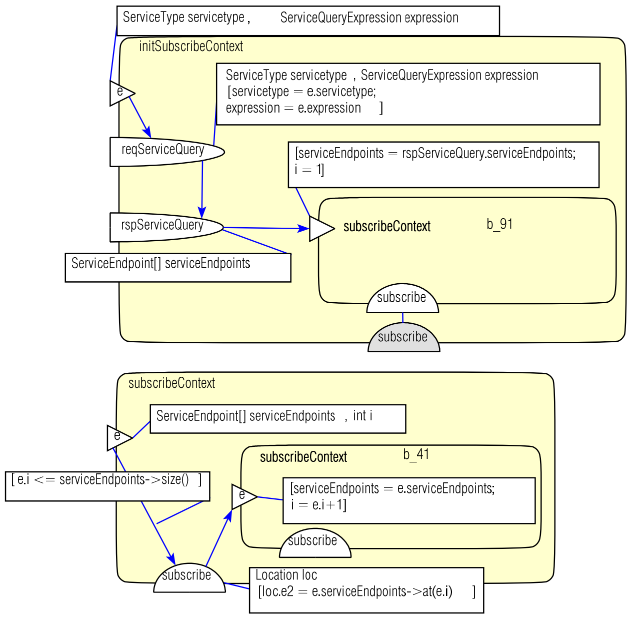

ServiceType servicetype , ServiceQueryExpression expression
initSubscribeContext
e
ServiceType servicetype , ServiceQueryExpression expression [servicetype = e.servicetype; expression = e.expression]
reqServiceQuery
[serviceEndpoints = rspServiceQuery.serviceEndpoints; i = 1]
rspServiceQuery
subscribeContext
b_91
ServiceEndpoint[] serviceEndpoints
subscribe
subscribe
subscribeContext
ServiceEndpoint[] serviceEndpoints , int i
e
subscribeContext
b_41
[e.i <= serviceEndpoints->size()]
e
[serviceEndpoints = e.serviceEndpoints; i = e.i+1]
subscribe
subscribe
Location loc [loc.e2 = e.serviceEndpoints->at(e.i)]

Figure 7-49 Initialization of the coordination component

7.7 Evaluation

The service specification level emphasizes ease of use for the service specifier and platform-independence for service specifications. A Freeband Service is defined from its integrated perspective, abstracting from any components that may support the execution of the service in terms of technology platforms such as Parlay (which provides context and action services in the telecommunications domain) and Web Services or CORBA (which provide service-oriented middleware architectures, including some service discovery functionality).

We have used a simple constraint language at the service specification level. An alternative to that could have been to adopt a standard language such as OCL for this level. However, this would compromise buildability of service specifications on top of the A-MUSE Platform. In the realization in the A-MUSE platform, constraints on information attributes become constraints on service properties of the trader service. Support for full OCL at the service specification level would introduce a large gap between the expressiveness of constraints on information attributes and the capabilities of the trader service (and its realizations). In addition, a simple constraint language improves ease of use for service designer. The constraint language adopted is a strict subset of OCL, which greatly simplifies the transformation of constraints (in T_1).

We have sacrificed generality at the service specification level to limit the size of the case study. The most important decision in this respect is that we assume that general-purpose reusable context sources and action services are available and can accommodate the needs of different services. A general solution would allow service designers to define their own service-specific context and action services. This is, however, not considered in this thesis.

The abstract platform at the platform-independent service design level has been chosen based on the pattern of service discovery found in a number of middleware platforms (e.g., OMG CORBA trader [88] and the UDDI registry [71] and in the ODP trader [60]). The trader service in the A-MUSE abstract platform is capable of supporting a simple constraint language and is capable of supporting dynamic service properties. These capabilities of the service trader do not have to be implemented in the coordination component, therefore simplifying the design of transformations that use the A-MUSE platform as target.

We have defined the abstract platform at the platform-independent service design level with a combination of the language-level and the model-level approaches. The decomposition of the A-MUSE abstract platform into a hierarchy of abstract platforms has facilitated its definition.

We have outlined how one can implement the trading service by service decomposition on top of a service discovery platform that does not support constraint languages or dynamic service properties (UDDI), in order to provide some indication of the buildability of the platform-independent service design level on middleware platforms of divergent characteristics. However, we have not included transformations T_2 and T_3 in the scope of this example.

Other quality characteristics of the A-MUSE abstract platform are a result of the set of design concepts adopted, and their direct support in the modelling language ISDL. An evaluation of the impact of these design concepts in quality characteristics of the abstract platforms is provided in chapter 5 of this thesis.

Chapter 8

Conclusions

This chapter summarizes the conclusions of this thesis and identifies some areas for further investigation. This chapter is organised as follows: section 8.1 presents some general considerations of our work; section 8.2 summarizes the main contributions and section 8.3 provides recommendations for future work.

8.1 General considerations

Understanding the methodological and architectural foundations of platform-independent design is paramount to reaping the benefits of platform-independence in model-driven design. We believe this thesis contributes to a better understanding of middleware-platform-independence and its consequences for the model-driven design process.

The methodology proposed in this thesis explores two main dimensions of separation of concerns: the separation of platform-independent and platform-specific concerns; and the separation of preparation and execution concerns. The former dimension results in the organization of application designs in several levels of platform-independence and the accompanying notion of abstract platform. The latter dimension results in the structuring of the design process into preparation and execution phases, and is necessary to deal with the vast diversity of application domain requirements and target platform characteristics.

The work we have presented in this thesis is complementary to metamodelling and model transformation engineering techniques. These techniques are neutral with respect to the abstraction criteria and design concepts used for platform-independent design. Therefore, these techniques do not clarify the relation between source and target designs, although they allow designers that know these relations to capture them in transformation specifications.

We have shown that service decomposition and interaction refinement can serve as design operations that progressively introduce middleware-platform-specific restrictions to designs, while preserving the conformance between source and target application designs. Conformance rules ensure that design decisions captured at a high level of platform-independence are preserved throughout the design trajectory.

We believe that the methodology proposed in this thesis enables more cost-effective development of distributed applications in the long term, especially due to the reuse of platform-independent designs. Inevitably, however, evidence for that can only be obtained with long-term cost-effectiveness studies, which fall outside the scope of this thesis.

8.2 Main contributions

We categorize the contributions of our work by their relation to:
– the *notion of an abstract platform*;
– the proposed *design process*, including the *design quality criteria* for abstract platform definition;
– the relation between *abstract platforms and modelling languages*; and,
– the adopted *design framework*.

8.2.1 Platform-independence and the notion of an abstract platform

Separation of concerns in the design process leads to the construction of different models of an application. The different concepts, structures or patterns used to construct application models constrain the choice of platforms differently, i.e., one can refer to many degrees of platform-independence. Organizing models at different levels of platform-independence allows designers to separately capture aspects of designs that remain stable in face of technology changes, leading to reusable platform-independent models.

Platform characteristics may affect designs at various levels of platform-independence, which may lead to subtle relations between designs at a low-level of platform-independence and designs at a higher-level of platform-independence. Platform characteristics assumed in platform-independent designs are better understood and controlled by designers if explicitly captured in abstract platform definitions as proposed in chapter 2 of this thesis.

We have shown the suitability of the abstract platform concept in several design examples throughout this thesis. We have also shown that the abstract platform concept can be used in the context of the RM-ODP, leading to a recursive application of the Computational Viewpoint. This is a first step towards reconciling the RM-ODP and the MDA in a comprehensive design framework for distributed application design.

8.2.2 Design quality criteria and the design process

The definition of abstract platforms should be guided by design quality criteria, as we have shown in chapter 3 of this thesis. Compliance to the criteria ensures that an abstract platform is influenced by a combination of top-down (generality, stability and ease of use) and bottom-up forces (buildability and platform portability requirements). The proposed design criteria have been justified by the Design Structure Matrices (DSM) [101, 116] analysis we have conducted in chapter 4. In the analysis, we have regarded models at different levels of platform-independence as modules in order to analyse their dependencies and interdependencies. We believe this is a useful application of the DSM technique, and perhaps can be explored in other design trajectories to analyse the dependencies of models at different levels of abstraction.

Since we have not restricted ourselves to an analysis of the design process based on the general model transformation pattern, we have been able to provide guidelines for separation of concerns that are grounded in design goals, including that of achieving platform-independence. For example, from the sole perspective of the general model transformation pattern, the distinction between a source model and transformation parameters is arbitrary, since both can be treated as inputs for the transformation. However, from a methodological perspective, it is possible to establish a meaningful distinction between a source model and transformation parameters: transformation parameters can be transformation-specific and platform-specific, whereas source models should be transformation-independent and platform-independent (see chapter 4 of this thesis).

8.2.3 Abstract platforms and modelling languages

We have shown that modelling language concepts and characteristics of abstract platforms are interrelated. Therefore, careful selection of a modelling language is indispensable for the definition of suitable abstract platforms, and, hence, beneficial exploitation of platform-independence.

Nevertheless, not all relevant characteristics of a design's abstract platform can be derived from the concepts underlying the modelling language adopted for the design. In particular, abstract platform characteristics may depend on restrictions on the use of particular constructs in a modelling language or the use of certain modelling styles or patterns. This is reflected in our methodology in the *language-level abstract platform definition* approach (see chapter 2 section 2.3.3 and chapter 6 section 6.3).

We have also shown that it may be necessary to define an abstract platform by defining reusable design artefacts that are composed with the application in the execution phase of the design process. This approach is called *model-level abstract platform definition* (see chapter 2 section 2.3.3 and chapter 6 section 6.4).

8.2.4 Design framework for platform-independent design

We have argued the case for a more prominent role of interaction system design in the model-driven design of distributed applications. In particular, by using service definitions for application interaction systems, a designer is able to obtain a high-level of platform-independence, in the sense that a broad set of middleware platforms that support different interaction patterns can potentially be used to support the interaction between application parts.

We have shown that the abstract interaction concept and interaction refinement design operations can be used to realize a platform-independent design in multiple platforms. This is possible because interaction can be modelled at a high level of abstraction with the abstract interaction concept. This level of abstraction is higher than the level of abstraction that can be obtained with concepts that correspond closely to operation invocation and asynchronous messaging mechanisms, such as those underlying UML and SDL.

We have shown that conformance can be defined and enforced by using service decomposition and interaction refinement design operations. The use of a uniform set of concepts in different levels of models facilitates the establishment of conformance relations between the levels.

8.3 Directions for further research

8.3.1 Reusable elements for abstract platform definition

The proliferation of different abstract platforms conflicts with the economies of scale that can be obtained by large-scale reuse of abstract platforms and transformations. The term abstract *platform* is meant to expose that, not unlike middleware platforms, abstract platforms can become themselves sources of heterogeneity.

One approach to cope with this is to define a small number of (reference) abstract platforms that are, to a great extent, application-domain-neutral and platform-independent. The event-based abstract platform, the service-oriented abstract platform and the service discovery abstract platform (chapters 6 and 7) are examples of abstract platforms that are general enough to qualify for inclusion in a reference architecture for abstract platform definition. However, since abstract platforms can be considered as coarse-grained architectural elements, this approach may lead to a reference architecture that is not flexible enough to deal with the variety of requirements for abstract platforms.

An alternative to this approach is to define a number of finer-grained *abstract platform elements* that can be composed to form abstract platforms that suit the needs of particular projects. While, in principle, this alternative would address the issue of flexibility, it would not directly address the issue of reuse of transformations, since specific transformations may be required for each valid combination of abstract platform elements. A solution to that would be to require transformation to be compositional, i.e., to require some correspondence between abstract platform elements and *transformation elements* to be established.

We believe the set of design concepts discussed in chapter 5 of this thesis can serve as a foundation for either of these two approaches, with the basic design concepts serving as the finest-grained abstract platform elements.

8.3.2 Conformance and transformation

Conformance rules and (non-parameterized) transformation specifications can be regarded as two extremes in relating source and target designs from the perspective of flexibility in the target design. Conformance rules determine the minimum to be preserved in a design step (hence maximum flexibility for target design without losing design decisions in the source design) and transformation specifications determine the maximum that can be prescribed in a design step (hence minimum flexibility for the target design). Future work should investigate techniques to assert whether a transformation specification complies with a set of conformance rules.

In addition, it would be interesting to investigate both conformance and transformation within the same transformation framework, possibly using the same techniques and tools for model transformation and for capturing and enforcing conformance rules. An application of that would be to allow designers to manually modify results of a transformation step when necessary, without breaking the relation between source and target design as defined by the conformance rules. We believe this is feasible by regarding both transformation and conformance as relations ([1] and [79]).

8.3.3 Platform-independent transformations

We have considered that transformations are specific to a target platform. In order to improve the opportunities for transformation reuse, the dependency between transformation specifications and target platforms could be reduced by using target platform models as transformation arguments. However, this solution would require general transformation specifications to define generalized implementation relations for a class of target platforms. Effectively, this would result in platform-independent transformation specifications. The level of generality that can be obtained with this technique is unclear and the feasibility of such an approach is issue for further investigation.

8.3.4 Beyond the scope of the design framework

Composition mechanisms

In section 5.3.5, we have discussed an approach to platform-specific realization based on the extension of middleware platforms with a number of mechanisms, such as middleware-level interceptors (with message reflection) [73, 117], composition filters [17] and aspect-oriented programming [37]. These mechanisms provide composition operators that can be used at middleware-platform level to separate extensions from a "base" platform. There is no direct correspondence between these mechanisms and service composition in the design framework. Further investigation is necessary to indicate whether similar approaches would facilitate composition at the design level, e.g., for the composition of abstract platform elements. Any composition operators introduced would have to be accounted for in an adequate conformance framework.

Conformance for the realization step

In the scope of the design framework presented in chapter 5, the use of a uniform set of concepts in different levels of models facilitates the establishment of conformance relations between the levels. However, since realizations fall outside the scope of the design framework, the notion of conformance we have explored cannot be directly applied to determine the relation between detailed designs and the realization of the application. A natural extension of our work would be to investigate practical conformance relations for the realization step.

Quality-of-service concepts

While we have considered the impact of platform quality-of-service (QoS) characteristics in our methodology (e.g., see chapter 4 section 4.3.2), we have not explored QoS concepts in the design framework. Further investigation should aim at establishing the relation between timing and probability constraints [89] and platform QoS support. Furthermore, the usage of specific transparency schemas referring to specific distribution transparencies should be investigated in the context of the recursive application of the computational viewpoint in RM-ODP.

Appendix A

Methodology quick guide

This appendix can be used as a quick reference guide for designers applying the methodology described in this thesis. Section A.1 provides an overview of the proposed design process; section A.2 outlines the activities of the preparation phase; section A.3 outlines the activities in the execution phase; finally, section A.4 provides some overall directives for the design process.

A.1 Overview of the design process

The design process is structured into a *preparation* and an *execution phase*. In the preparation phase, designers identify (and, when necessary, define) the required levels of models, their *abstract platforms* and the modelling language(s) to be used. A designer may also identify or define *transformation specifications* between related levels of models in the preparation phase. The results of the preparation phase are used in the execution phase, which entails the creation of models of an application using specific modelling languages and abstract platforms, and the (possibly automated) execution of transformation activities.

Iterations between the preparation and execution phases may be necessary when new target platforms are introduced, thus requiring the development of new transformations, or when improved understanding of design steps performed manually creates opportunities for the automation of these steps in terms of transformation specifications. The preparation phase may also have to be revisited in case it becomes evident during the execution phase that requirements for abstract platforms, modelling languages and transformations are not satisfied.

A.2 Preparation phase

In the preparation phase, designers should:

1. define the organization of the execution phase, i.e., define required levels of models, their abstract platforms, and transformations;
2. define the modelling language(s) used for representing models at each level;
3. define abstract platforms using the language-level or model-level abstract platform definition approaches; and,
4. define (parameterized) transformation specifications between the various levels of models.

Figure A-1 shows the activities in the preparation phase schematically. Activity (1) precedes (2), (3) and (4). Activities (2) and (3) are interrelated and are depicted in the same block.

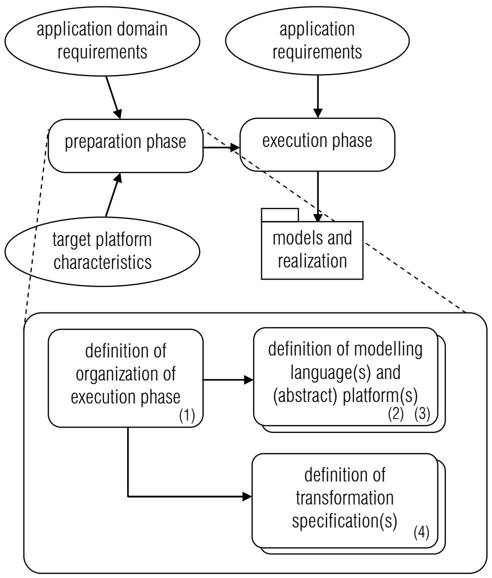

application domain requirements
application requirements
preparation phase
execution phase
target platform characteristics
models and realization
definition of organization of execution phase (1)
definition of modelling language(s) and (abstract) platform(s) (2) (3)
definition of transformation specification(s) (4)

Figure A-1 Preparation and execution phases and their results

Criteria for activity (1) are defined in chapters 3 and 4 of this thesis. An example of organization of the execution phase is provided in chapter 7. Design concepts relevant for activities (2) and (3) are defined in chapter 5, and include the concepts of interaction systems, abstract interaction and service. The language-level and model-level abstract platform definition approaches are defined in chapter 2 (section 2.3) and illustrated in chapter 6 (with UML and UML Profiling) and chapter 7 (for ISDL and MOF). Activity (4) consists of capturing service decomposition and interaction refinement design operations (chapter 5) in transformation specifications.

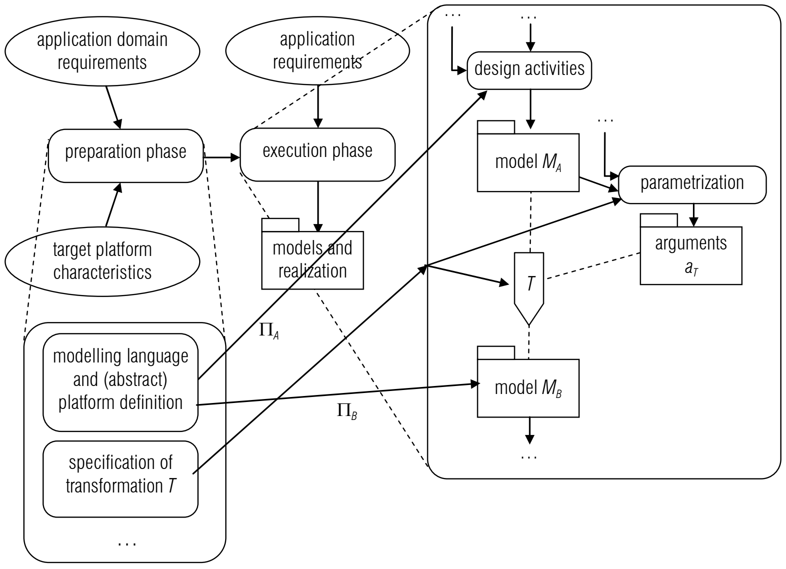
The results of the preparation phase (abstract platform, modelling language and transformation definitions) may be consolidated in a catalogue, which a designer consults in the preparation phase of each new project.

A.3 Execution phase

The execution phase entails the creation of models of a specific application using modelling languages, abstract platforms and transformations defined in the preparation phase. The execution phase leads ultimately to a realization (or alternative realizations) of the application and reusable platform-independent models of the application (at different levels of platform-independence). This phase also entails analysis, testing and validation of models and realizations (outside the scope of this thesis). The execution phase can be considered as a long-running phase, including activities for the maintenance and evolution of an application.

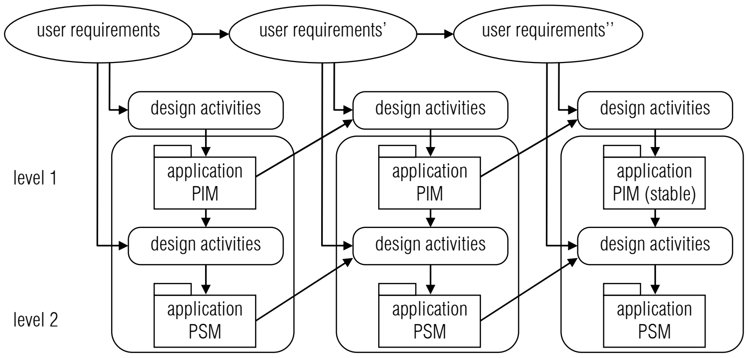
Figure A-2 shows how the preparation phase relates to the execution phase, considering only two levels of models *A* and *B* related by an automated transformation *T*. An abstract platform model Π_A is used in the elaboration of a model M_A. When the transformation defined in the preparation phase is parameterized, a designer may provide transformation arguments a_T to influence design decisions for the transformation activities. The result of transformation activities is a model M_B, which relies on a (abstract) platform model Π_B.

Figure A-2 Preparation and execution phases and their results



A designer may apply an iterative design approach in the execution phase, as illustrated in *Figure A-3* for the case of two levels of models, a platform-independent level and a platform-specific level. Implications of the iterative design approach for platform-independence are discussed in chapter 4 of this thesis.

Figure A-3 Iterative design approach in the execution phase



A.4 Some overall directives

The following directives apply to the design process (for motivation see chapter 4):

- Interdependent design decisions must be captured at the same level of platform-independence. Since some design decisions are platform-specific, this imposes constraints on the organization of models at different levels of platform-independence (see section 4.3.2 for approaches to coping with interdependent design decisions).
- Platform-independent models must be transformation-specification-independent and transformation-arguments-independent;
- Transformation arguments can be transformation-specific as well as platform-specific;
- Changes in source and target models or transformation arguments should be accommodated in source, target models or transformation arguments, but neither in the (abstract) platforms nor transformation specification;
- A designer may identify application-specific interaction systems to define application parts at a high-level of platform-independence. Criteria for justifying this technique are presented in section 5.2.4 and techniques to apply service decomposition, designing application interaction systems in terms of an abstract platform are presented in section 5.3.1.

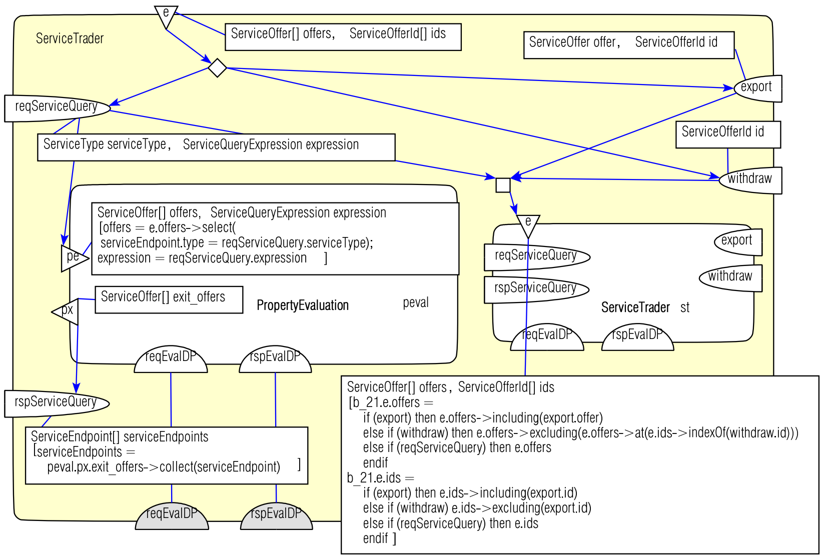
Appendix B

Specification of the trader service

In this appendix, we show the complete specification of the trader service which is used in our case study (see chapter 7 of this thesis, section 7.4.3).

Figure B-1 depicts the behaviour definition of the trader service in ISDL. A *reqServiceQuery* interaction is followed by the execution of the *PropertyEvaluation* behaviour which evaluates the service query expression. Its *exit_offers* exit parameter represents a sequence of offers which comply with the service query. The *rspServiceQuery* interaction returns the list of endpoints for the service offers in *exit_offers*. The list of current offers (*offers*) is updated in a recursive instantiation of the *ServiceTrader* behaviour: the occurrence of *export* results in the inclusion of the exported offer (*export.offer*) in *offers* and the occurrence of *withdraw* results in the exclusion of the offer.

Figure B-1 Trader behaviour



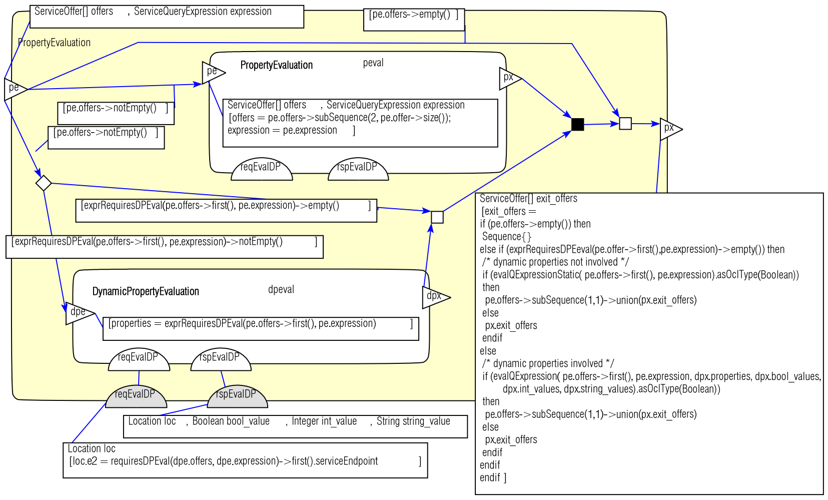
```
ServiceOffer[] offers,   ServiceQueryExpression expression
[offers = e.offers->select(
 serviceEndpoint.type = reqServiceQuery.serviceType);
expression = reqServiceQuery.expression    ]
```

```
ServiceEndpoint[] serviceEndpoints
[serviceEndpoints =
   peval.px.exit_offers->collect(serviceEndpoint)    ]
```

```
ServiceOffer[] offers , ServiceOfferId[] ids
[b_21.e.offers =
    if (export) then e.offers->including(export.offer)
    else if (withdraw) then e.offers->excluding(e.offers->at(e.ids->indexOf(withdraw.id)))
    else if (reqServiceQuery) then e.offers
    endif
b_21.e.ids =
    if (export) then e.ids->including(export.id)
    else if (withdraw) e.ids->excluding(export.id)
    else if (reqServiceQuery) then e.ids
    endif ]
```

Figure B-2 shows the *PropertyEvaluation* behaviour definition. This behaviour evaluates the service query expression for each service offer. It is defined by recursive instantiation. A service offer is only included in *exit_offers* when the service query evaluates to *true* to that particular offer. Evaluating the service query may require the evaluation of dynamic service properties, which is the role of the *DynamicPropertyEvaluation* behaviour.

Figure B-2 PropertyEvaluation behaviour definition

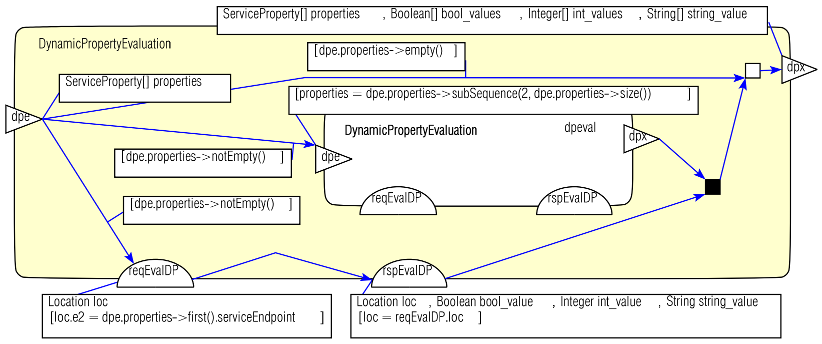

ServiceOffer[] offers , ServiceQueryExpression expression
[pe.offers->empty()]
PropertyEvaluation
pe
[pe.offers->notEmpty()]
[pe.offers->notEmpty()]
PropertyEvaluation
peval
pe
px
ServiceOffer[] offers , ServiceQueryExpression expression
[offers = pe.offers->subSequence(2, pe.offer->size());
expression = pe.expression]
reqEvalDP
rspEvalDP
px
[exprRequiresDPEval(pe.offers->first(), pe.expression)->empty()]
[exprRequiresDPEval(pe.offers->first(), pe.expression)->notEmpty()]
DynamicPropertyEvaluation
dpeval
dpe
dpx
[properties = exprRequiresDPEval(pe.offers->first(), pe.expression)]
reqEvalDP
rspEvalDP
reqEvalDP
rspEvalDP
Location loc , Boolean bool_value , Integer int_value , String string_value
Location loc
[loc.e2 = requiresDPEval(dpe.offers, dpe.expression)->first().serviceEndpoint]

```
ServiceOffer[] exit_offers
 [exit_offers =
if (pe.offers->empty()) then
 Sequence{}
else if (exprRequiresDPEeval(pe.offer->first(),pe.expression)->empty()) then
 /* dynamic properties not involved */
 if (evalQExpressionStatic( pe.offers->first(), pe.expression).asOclType(Boolean))
 then
  pe.offers->subSequence(1,1)->union(px.exit_offers)
 else
  px.exit_offers
 endif
else
 /* dynamic properties involved */
 if (evalQExpression( pe.offers->first(), pe.expression, dpx.properties, dpx.bool_values,
       dpx.int_values, dpx.string_values).asOclType(Boolean))
 then
  pe.offers->subSequence(1,1)->union(px.exit_offers)
 else
  px.exit_offers
 endif
endif
endif ]
```

Recursive instantiation of *PropertyEvaluation* does not force a particular order for service property evaluation: all service properties are evaluated independently, and the results are merged.

Figure B-3 shows the *DynamicPropertyEvaluation* behaviour definition. This behaviour is also defined by recursive instantiation, using the same instantiation pattern that was used for *PropertyEvaluation*. For each dynamic property, two interactions occur: *reqEvalDP* and *rspEvalDP*. These interactions occur at the endpoint registered in the service offer as a dynamic property evaluator.

Figure B-3 DynamicPropertyEvaluation behaviour definition


ServiceProperty[] properties , Boolean[] bool_values , Integer[] int_values , String[] string_value
DynamicPropertyEvaluation
[dpe.properties->empty()]
ServiceProperty[] properties
dpx
[properties = dpe.properties->subSequence(2, dpe.properties->size())]
dpe
DynamicPropertyEvaluation
dpeval
dpx
[dpe.properties->notEmpty()]
dpe
[dpe.properties->notEmpty()]
reqEvalDP
rspEvalDP
reqEvalDP
rspEvalDP
Location loc
[loc.e2 = dpe.properties->first().serviceEndpoint]
Location loc , Boolean bool_value , Integer int_value , String string_value
[loc = reqEvalDP.loc]

The constraints attached to exit point *dpx* are shown in *Figure B-4*. Depending on the type of the dynamic property its value is added to either *bool_values*, *int_values* or *string_values*.

Figure B-4 Constraints attached to exit point *dpx*

```
properties = dpe.properties

bool_values =
        if (dpe.properties->empty()) then
                Sequence{}
        else
                if (dpe.properties->first().serviceProperty.datatype == ServicePropertyType::Boolean)
                        dpx.bool_values->prepend(rspEvalDP.bool_value)
                else
                        dpx.bool_values->prepend(false)   /* this value is a placeholder */
                endif
        endif
int_values =
        if (dpe.properties->empty()) then
                Sequence{}
        else
                if (dpe.properties->first().serviceProperty.datatype == ServicePropertyType::Integer)
                        dpx.bool_values->prepend(rspEvalDP.int_value)
                else
                        dpx.bool_values->prepend(0)       /* this value is a placeholder */
                endif
        endif
string_values =
        if (dpe.properties->empty()) then
                Sequence{}
        else
                if (dpe.properties->first().serviceProperty.datatype == ServicePropertyType::String)
                        dpx.bool_values->prepend(rspEvalDP.string_value)
                else
                        dpx.bool_values->prepend("")      /* this value is a placeholder */
                endif
        endif
```

In the design of the trader service, dynamic properties are evaluated by invoking a dynamic property evaluator. To accommodate potential differences between the services that provide property values and the behaviour which is expected by the trader, we introduce wrappers when necessary. *Figure B-5* shows the wrappers that expose context information as service properties, in this case the coordinates for the location of an aid person.

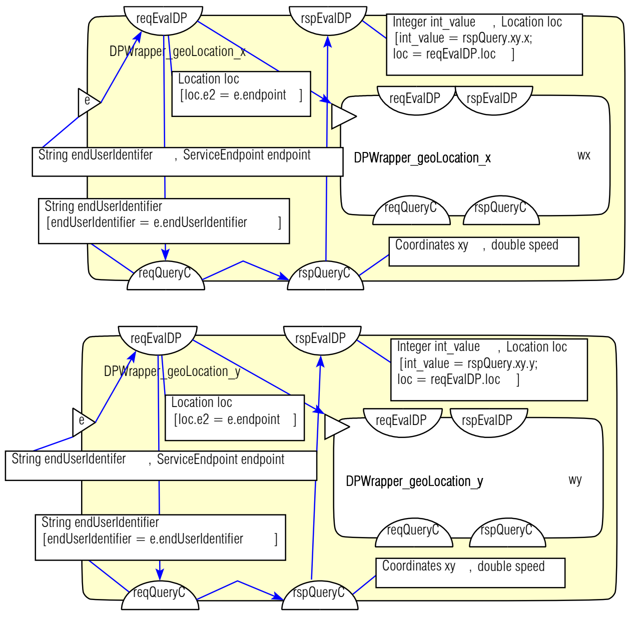


Figure B-5 Dynamic property wrappers

Figure B-6 shows the OCL definition of the *evalQExpression* helper operation. This operation evaluates the service query expression for a particular offer. It is defined recursively, navigating the *ServiceQueryExpression* tree.

Figure B-6 Helper *evalQExpression* used in constraints of behaviour *PropertyEvaluation*

```
/*
evalQExpression is a helper in behaviour PropertyEvaluation.
It evaluates the service query expression.  It is used in PropertyEvaluation to determine whether an
offer complies with the service query expression.

Parameters dproperties, bool_values, int_value, string_values represent dynamic properties and their
values.
pre: expression is valid expression
*/
context px
def: evalQExpression( offer : ServiceOffer, expression : ServiceQueryExpression,
        dproperties : Sequence(ServiceProperty), bool_values : Sequence(Boolean),
        int_values : Sequence(Integer), string_values : Sequence(String)  ) : oclAny
=
/* defined recursively */
if (expression.oclIsKindOf(LiteralExpression)) then
                expression.value
else
if (expression.oclIsKindOf(UnaryExpression)) then
        if (expression.oclIsKindOf(MinusExpression)) then
                -evalQExpression(offer, expression.serviceQueryExpression).oclAsType(Integer)
        else
        /* inv: expression.oclIsKindOf(NotExpression) */
                not evalQExpression(offer, expression.serviceQueryExpression).oclAsType(Boolean)
        endif
else
if (expression.oclIsKindOf(BinaryExpression)) then
        if (expression.oclIsKindOf(BinaryArithmeticExpression)) then
                let exp = expression.oclAsType(BinaryArithmeticExpression) in
                if (exp.operator = BinaryArithmeticOperator::addition) then
                        (evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer)+
                         evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer))
                else
                        if (exp.operator = BinaryArithmeticOperator::subtraction) then
                                (evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer)-
                                 evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer))
                        else
                        if (exp.operator = BinaryArithmeticOperator::multiplication) then
                                (evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer)*
                                 evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer))
                        else
                        if (exp.operator = BinaryArithmeticOperator::division) then
                                (evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer)/
                                 evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer))
                        endif
                endif
        else
        /* inv: expression.oclIsKindOf(BinaryBooleanExpression) */
                let exp = expression.oclAsType(BinaryBooleanExpression) in
                if (exp.operator = BinaryBooleanOperator::or_) then
                        (evalQExpression(offer, expression.lefthandoperand).oclAsType(Boolean) or
                         evalQExpression(offer, expression.lefthandoperand).oclAsType(Boolean))
                else if (exp.operator = BinaryBooleanOperator::and_) then
                        (evalQExpression(offer, expression.lefthandoperand).oclAsType(Boolean) and
                         evalQExpression(offer, expression.lefthandoperand).oclAsType(Boolean))
                else if (exp.operator = BinaryBooleanOperator::equal_) then
                        (evalQExpression(offer, expression.lefthandoperand) =
                         evalQExpression(offer, expression.lefthandoperand))
                else if (exp.operator = BinaryBooleanOperator::notequal_) then
                        (evalQExpression(offer, expression.lefthandoperand) <>
                         evalQExpression(offer, expression.lefthandoperand))
                else if (exp.operator = BinaryBooleanOperator::greaterThan) then
                        (evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer) >
                         evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer))
                else if (exp.operator = BinaryBooleanOperator::lessThan) then
                        (evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer) <
                         evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer))
                else if (exp.operator = BinaryBooleanOperator::greaterThanOrEqual) then
                        (evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer) >=
                         evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer))
                else if (exp.operator = BinaryBooleanOperator::lessThanOrEqual) then
                        (evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer) <=
                         evalQExpression(offer, expression.lefthandoperand).oclAsType(Integer))
                endif
        endif
else
if (expression.OclisTypeOf(ServicePropertyExpression)) then
        let property : ServiceProperty =
                offer.serviceProperty->select( name = expression.servicePropertyName )->first() in
        if (property.OclisTypeOf(StaticServiceProperty)) then
                if (property.OclisKindOf(BooleanServiceProperty))
                        property.asOclType(BooleanServiceProperty).value
                else if (property.OclisTypeOf(IntegerServiceProperty))
                        property.asOclType(IntegerServiceProperty).value
                else if property.OclisTypeOf(StringServiceProperty))
                        property.asOclType(StringServiceProperty).value
                endif
        else
                /* property is a dynamic service property of the offer */
                let datatype : ServicePropertyType =
                        property.asOclType(DynamicServiceProperty).datatype in

                /* read value of dynamic service property */
                if (datatype = ServicePropertyType::Boolean) then
                        bool_values->at(dproperties->indexOf(property))
                else if (datatype = ServicePropertyType::Integer) then
                        int_values->at(dproperties->indexOf(property))
                else if (datatype = ServicePropertyType::String) then
                        string_values->at(dproperties->indexOf(property))
        endif
endif

context px
def: evalQExpressionStatic( offer : ServiceOffer, expression : ServiceQueryExpression ) : oclAny
=
evalQExpression(offer, expression, Sequence{}, Sequence{}, Sequence{}, Sequence{})
```

Figure B-7 shows the OCL definition of a helper operation used in the *PropertyEvaluation* behaviour. This operation returns to a list of dynamic service properties which must be evaluated for the service query expression to be evaluated.

Figure B-7 Helper *exprRequiresEval* used in constraints of *PropertyEvaluation*

```
/*
exprRequiresDPEval is a helper in behaviour PropertyEvaluation.
This operation returns to a list of dynamic service properties that have to be
evaluated in order to evaluate the expression.
*/
context pe
def: exprRequiresDPEval ( offer : ServiceOffer, expression : ServiceQueryExpression ) :
      Sequence(ServiceProperty)
=
/* defined recursively, until leaf nodes of expression tree are found */
if (not expression.oclAsType(ServicePropertyExpression).oclIsUndefined()) then
      /* either a static or a dynamic property */

      let property : ServiceProperty =
      offer.serviceProperty->select( name = expression.servicePropertyName )->first() in

      if (not property.oclAsType(StaticServiceProperty).oclIsUndefined()) then
            /* static property */
            Sequence {}
      else
            /* dynamic property */
            (Sequence{})->append(property)
else
if (not expression.oclAsType(UnaryExpression).oclIsUndefined()) then
      /* recursively evaluate */
      requiresDPEval ( expression.serviceQueryExpression )
else
if (not expression.oclAsType(BinaryExpression).oclIsUndefined()) then
      /* recursively evaluate both left- and right-hand sides of binary expression */
      requiresDPEval ( offer, expression.righthandoperand ).union(
      requiresDPEval ( offer, expression.lefthandoperand ))
else
      /* a leaf that is not a property */
      Sequence {}
endif
```

The *expression* parameter expected by the trader in the *reqServiceQuery* interaction has type *ServiceQueryExpresssion.* This means that during the transformation from service specification to the platform-independent service design level, constraints on information attributes that have been marked as service properties must be translated into OCL statements that define an equivalent expression as an instance of *ServiceQueryExpresssion.* The instance of a *ServiceQueryExpression* can be seen as a parsed tree that corresponds to the expression in textual format. *Figure B-8* shows an example of the transformation of a textual expression at the service specification level to a set of OCL statements at the service design level.

Figure B-8 Textual expression at the service specification level and OCL statements at the service design level

constraint on information attribute at service specification level

```
((coverageArea.geoLocation.x - position_rspC.xy.x) *
 (coverageArea.geoLocation.x - position_rspC.xy.x) +
 (coverageArea.geoLocation.y - position_rspC.xy.y) *
 (coverageArea.geoLocation.y - position_rspC.xy.y))
 < coverageArea.range * coverageArea.range
```

---

service query expression in textual format at service design level

```
expression =
"((coverageArea.geoLocation.x - ".concat(
rspQueryContext_position_rspC.xy.x.toString()).concat(
") * (coverageArea.getLocation.x - ").concat(
rspQueryContext_position_rspC.xy.x.toString()).concat(
") + (coverageArea.getLocation.y - ").concat(
rspQueryContext_position_rspC.xy.y.toString()).concat(
") * (coverageArea.getLocation.y - ").concat(
rspQueryContext_position_rspC.xy.y.toString()).concat(
") ) < coverageArea.range * coverageArea.range ")
```

---

instance of ServiceQueryExpression at service design level

```
expression.isOclType(BinaryBooleanExpression) and
expression.operator = BinaryBooleanOperator::lessThan and
  expression.lefthand.isOclType(BinaryArithmeticExpression) and
  expression.lefthand.operator = BinaryArithmeticOperator::addition   and
    expression.lefthand.lefthand.isOclType(BinaryArithmeticExpression) and
    expression.lefthand.lefthand.operator = BinaryArithmeticOperator::multiplication   and
      expression.lefthand.lefthand.lefthand.isOclType(BinaryArithmeticExpression) and
      expression.lefthand.lefthand.lefthand.operator = BinaryArithmeticOperator::subtraction and
        expression.lefthand.lefthand.lefthand.lefthand.isOclType(ServicePropertyExpression) and
        expression.lefthand.lefthand.lefthand.lefthand.servicePropertyName = "coverageArea.geoLocation.x" and
        expression.lefthand.lefthand.lefthand.righthand.isOclType(IntegerLiteral) and
        expression.lefthand.lefthand.lefthand.righthand.value = rspQueryContext_position_rspC.xy.x and
      expression.lefthand.lefthand.righthand.isOclType(BinaryArithmeticExpression) and
      expression.lefthand.lefthand.righthand.operator = BinaryArithmeticOperator::subtraction and
        expression.lefthand.lefthand.righthand.lefthand.isOclType(ServicePropertyExpression) and
        expression.lefthand.lefthand.righthand.lefthand.servicePropertyName = "coverageArea.geoLocation.x" and
        expression.lefthand.lefthand.righthand.righthand.isOclType(IntegerLiteral) and
        expression.lefthand.lefthand.righthand.righthand.value = rspQueryContext_position_rspC.xy.x and
    expression.lefthand.righthand.isOclType(BinaryArithmeticExpression) and
    expression.lefthand.righthand.operator = BinaryArithmeticOperator::multiplication   and
      expression.lefthand.righthand.lefthand.isOclType(BinaryArithmeticExpression) and
      expression.lefthand.righthand.lefthand.operator = BinaryArithmeticOperator::subtraction and
        expression.lefthand.righthand.lefthand.lefthand.isOclType(ServicePropertyExpression) and
        expression.lefthand.righthand.lefthand.lefthand.servicePropertyName = "coverageArea.geoLocation.y" and
        expression.lefthand.righthand.lefthand.righthand.isOclType(IntegerLiteral) and
        expression.lefthand.righthand.lefthand.righthand.value = rspQueryContext_position_rspC.xy.y and
      expression.lefthand.righthand.righthand.isOclType(BinaryArithmeticExpression) and
      expression.lefthand.righthand.righthand.operator = BinaryArithmeticOperator::subtraction and
        expression.lefthand.righthand.righthand.lefthand.isOclType(ServicePropertyExpression) and
        expression.lefthand.righthand.righthand.lefthand.servicePropertyName = "coverageArea.geoLocation.y" and
        expression.lefthand.righthand.righthand.righthand.isOclType(IntegerLiteral) and
        expression.lefthand.righthand.righthand.righthand.value = rspQueryContext_position_rspC.xy.y
  expression.righthand.isOclType(BinaryArithmeticExpression) and
  expression.righthand.operation = BinaryArithmeticOperator::multiplication and
    expression.righthand.lefthand.isOclType(ServicePropertyExpression) and
    expression.righthand.lefthand.name = "coverageArea.range" and
    expression.righthand.righthand.isOclType(ServicePropertyExpression) and
    expression.righthand.lefthand.name = "coverageArea.range"
```

References

1. D. Akehurst, S. Kent, O. Patrascoiu, “A relational approach to defining and implementing transformations between metamodels”, *Software and Systems Modeling*, vol. 2. no. 4, Springer-Verlag, 2003, pp. 215-239.

2. C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction,* Center for Environmental Structure Series, Oxford University Press, 1977.

3. R. Allen and D. Garlan, “A Formal Basis for Architectural Connection,” *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, ACM Press, July 1997, pp. 213-219.

4. J.P.A. Almeida, M. van Sinderen, D. Quartel, and L. Ferreira Pires, “Designing Interaction Systems for Distributed Applications”, *IEEE Distributed Systems Online*, vol. 6, no. 3, IEEE Computer Society, March 2005.

5. J.P.A. Almeida, R. Dijkman, M. van Sinderen, and L. Ferreira Pires, “Platform-independent modelling in MDA: supporting abstract platforms”, *Proceedings Model-Driven Architecture: Foundations and Applications 2004 (MDAFA 2004),* Linköping University, Linköping, Sweden, June 2004, pp. 219-233. Revised version appeared in Lecture Notes in Computer Science, vol. 3599, Springer-Verlag, June 2005, pp. 174-188.

6. J.P.A. Almeida, R. Dijkman, M. van Sinderen, and L. Ferreira Pires, “On the Notion of Abstract Platform in MDA Development,” *Proceedings 8th IEEE International Conference on Enterprise Distributed Object Computing (EDOC 2004)*, IEEE Computer Society Press, Sept. 2004.

7. J.P.A. Almeida, M. van Sinderen, L. Ferreira Pires, and M. Wegdam, “Platform-independent Dynamic Reconfiguration of Distributed Applications,” *Proceedings 10th IEEE International Workshop on Future Trends in Distributed Computing Systems (FTDCS 2004)*, IEEE Computer Society Press, May 2004, pp. 286-291.

8. J.P.A. Almeida, M. van Sinderen, and L. Ferreira Pires, “The role of the RM-ODP Computational Viewpoint Concepts in the MDA approach,” *Proceedings*

of the 1st European Workshop on Model-Driven Architecture with Emphasis on Industrial Applications (MDA-IA 2004), technical report TR-CTIT-04-12, ISSN 1381-3625, Centre for Telematics and Information Technology, University of Twente, The Netherlands, March 2004, pp. 43-51.

9. J.P.A. Almeida, M. van Sinderen, L. Ferreira Pires, and D. Quartel, "A systematic approach to platform-independent design based on the service concept," *Proceedings 7th IEEE Intl. Enterprise Distributed Object Computing Conference (EDOC 2003)*, IEEE Computer Society Press, Sept. 2003, pp. 112-123.
10. J.P.A. Almeida, *Dynamic Reconfiguration of Object-Middleware-based Distributed Systems*, M.Sc. thesis, University of Twente, The Netherlands, June 2001.
11. G. Arango, "Domain Analysis: from Art Form to Engineering Discipline," *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 3, ACM Press, May 1989, pp. 152-159.
12. C. Atkinson and T. Kühne, "A Generalized Notion of Platforms for Model-Driven Development", *Model-driven Software Development*, S. Beydeda, M. Book, V. Gruhn, eds., Springer, 2005.
13. C. Atkinson and T. Kuhne, "Aspect-Oriented Development with Stratified Frameworks", *IEEE Software*, vol. 20, no. 1, IEEE Computer Society Press, 2003, pp. 81-89.
14. C.Y. Baldwin and K.B. Clark, *Modularity in the Design of Complex Engineering Systems*, Harvard Business School Working Paper Series, no. 04-055, Jan. 2004.
15. C.Y. Baldwin and K.B. Clark, *Design Rules, Volume 1, The Power of Modularity*, MIT Press, Cambridge, MA, 2000.
16. C.B. Barbosa, *Frameworks for Implementing Protocols: a Model Based Approach*, Ph.D. thesis, University of Twente, The Netherlands, Feb. 2001; http://purl.org/utwente/36595
17. L. Bergmans and M. Aksit, "Composing Crosscutting Concerns Using Composition Filters", *Communications of the ACM*, vol. 44, no. 10, pp. 51-57, Oct. 2001.
18. P.A. Bernstein, "Middleware: a model for distributed system services", *Communications of the ACM*, vol. 39, no.2, ACM Press, Feb. 1996, pp. 86-98.
19. G. Blair and J.B. Stefani, *Open Distributed Processing and Multimedia*, Addison Wesley, 1997.
20. X. Blanc, *ModelBus: A MODELWARE White Paper*, April 2005; http://www.modelware-ist.org/public_documents/ModelBusWhitePaper_MDDI.pdf
21. B. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer,* vol. 21, no. 5, IEEE Computer Society Press, May 1988, pp. 61-72.
22. T. Bolognesi, J. van de Lagemaat, and C. Vissers, eds., *LOTOSphere: Software Development with LOTOS*, Kluwer Academic Publishers, 1995.

23. C. Burt et al., "Quality of Service Issues Related to Transforming Platform Independent Models to Platform Specific Models," *Proceedings Sixth International Conference on Enterprise Distributed Object Computing (EDOC 2002)*, IEEE Computer Society Press, Sept. 2002, pp. 212-223.

24. E. Brinksma, B. Jonsson, and F. Orava, "Refining Interfaces of Communicating Systems", *Proceedings of the International Joint Conf. on Theory and Practice of Software Development (TAPSOFT'91)*, Lecture Notes in Computer Science, vol. 494, Springer-Verlag, 1991, pp. 297-312.

25. M. Broy, "(Inter-)action refinement: The easy way", *Program Design Calculi*, Springer NATO ASI Series, Series F : Computer and System Sciences, vol. 118, Springer-Verlag, 1993, pp. 121-158.

26. H. Chen, T. Finin, and A. Joshi, "An ontology for context-aware pervasive computing environments", *Knowledge Engineering Review*, Special Issue on Ontologies for Distributed Systems, vol. 18, no. 3, pp. 197-207, 2003.

27. P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns, Series: The SEI Series in Software Engineering*, Addison Wesley Professional, 2001.

28. K. Czarnecki and S. Helsen, "Classification of Model Transformation Approaches," *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, Anaheim, CA, USA, Oct. 2003.

29. A. van Deursen, P. Klint, and J. Visser, "Domain-Specific Languages: An Annotated Bibliography," *ACM SIGPLAN Notices,* vol. 35, no. 6, June 2000, pp. 26-36.

30. R.M. Dijkman, D. Quartel, L. Ferreira Pires, and M. van Sinderen, "A Rigorous Approach to Relate the RM-ODP Enterprise and Computational Viewpoint," *Proceedings 8th IEEE International Conference on Enterprise Distributed Object Computing (EDOC 2004)*, IEEE Computer Science Press, Sept. 2004, pp. 187-200.

31. R.M. Dijkman, "A Basic Design Model for Service-Oriented Design," *ArCo Project Deliverable ArCo/WP1/T1/D2/V1.00*, University of Twente, The Netherlands, November 2003.

32. R.M. Dijkman, J.P.A. Almeida, and D. Quartel, "Verifying the Correctness of Component-Based Applications that Support Business Processes", *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering (CBSE) - Automated Reasoning and Predication*, Portland, OR, USA, May 2003, pp. 43-48.

33. R.M. Dijkman, *Consistency in Multi-Viewpoint Architectural Design*, Ph.D. thesis, University of Twente, The Netherlands, Feb. 2006.

34. P. Dockhorn Costa, L. Ferreira Pires, and M. van Sinderen, "Architectural Support for Mobile Context-Aware Applications" (Chapter XXXI), *Handbook of Research on Mobile Multimedia*, Idea Group, 2006, pp. 456-475.

35. P. Dockhorn Costa, L. Ferreira Pires, and M. van Sinderen, "Designing a Configurable Services Platform for Mobile Context-Aware Applications", *International Journal of Pervasive Computing and Communications (JPCC)*, vol. 1, no. 1, Troubador Publishing, March 2005.

36. Eclipse Foundation, *Eclipse Modeling Framework*; http://www.eclipse.org/emf

37. T. Elrad, R.E. Filman, and A. Bader, eds., *Communications of the ACM, Special Section on Aspect-Oriented Programming*, vol. 44, no.10, ACM Press, Oct. 2001, pp. 29-97.

38. R. Eshuis, *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*, Ph.D. thesis, University of Twente, The Netherlands, Oct. 2002.

39. C.R.G. de Farias, *Architectural design of groupware systems: a component-based approach*, Ph.D. thesis, University of Twente, The Netherlands, May 2002; http://purl.org/utwente/37999

40. L. Ferreira Pires, *Architectural Notes: a framework for distributed systems development*, Ph.D. Thesis, University of Twente, The Netherlands, Sept. 1994.

41. L. Ferreira Pires, J. de Heer, J. Brok, M. Hutschemaekers, M. van Sinderen, K. Sheikh, *High level requirements - Version 2*, Freeband/AWARENESS/D1.2v2, Oct. 2005; https://doc.freeband.nl/dscgi/ds.py/Get/File-60593

42. *Freeband A-MUSE*, 2004; http://a-muse.freeband.nl

43. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

44. J. Gray, T. Bapty, S. Neema, D.C. Schmidt, A. Gokhale, and B. Natarajan, “An Approach for Supporting Aspect-Oriented Domain Modeling,” *Proceedings Generative Programming and Component Engineering (GPCE 2003),* Lecture Notes in Computer Science, vol. 2830, Springer-Verlag, Sept. 2003, pp. 151-168.

45. G. Guizzardi, R. Dijkman, J.P.A. Almeida, and P. Dockhorn Costa, “Visserian Metaphysics”, *Architectural Design of Open Distributed Systems: From Interface to Telematics, Liber Amicorum, dedicated to Chris Vissers,* M. van Sinderen and L. Ferreira Pires (eds.), Telematica Instituut, The Netherlands, 2006, pp. 27-40.

46. G. Guizzardi, L. Ferreira Pires, M. van Sinderen, “An Ontology-Based Approach for Evaluating the Domain Appropriateness and Comprehensibility Appropriateness of Modeling Languages”, *Proceedings ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005)*, Lecture Notes in Computer Science, vol. 3713, Springer-Verlag, pp. 235-244, Oct. 2005.

47. D. Harel and B. Rumpe, *Modelling Languages: Syntax, Semantics and All That Stuff*, technical report MCS00-16, The Weizmann Institute of Science, Rehovot, Israel, 2000.

48. G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

49. The Institute of Electrical and Electronics Engineers (IEEE) Standards Board, *Recommended Practice for Architectural Description of Software-Intensive Systems (IEEE-Std-1471- 2000)*, Sept 2000.

50. IONA Technologies, *IONA Products: Orbix*; http://www.iona.com/products/orbix.htm

51. IONA Technologies, *Orbacus*; http://www.orbacus.com

52. Centre for Telematics and Information Technology, *ISDL home*; http://isdl.ctit.utwente.nl/

53. Centre for Telematics and Information Technology, *ISDL schema*; http://isdl.ctit.utwente.nl/ISDLSchema/BasicISDL.xsd

54. ITU-T, *Recommendation Z.100 – CCITT Specification and Description Language*, International Telecommunications Union (ITU), 2002.

55. ITU-T, *Recommendation Z.100, Annex F: SDL Formal Semantics Definition*, International Telecommunications Union (ITU), Geneva, 2000.

56. ITU-T / ISO, *Open Distributed Processing - Reference Model – All Parts*, ITU-T Recommendations X.901, X902, X903, X.904 | ISO/IEC 10746-1, 2, 3, 4, 1995.

57. ITU-T / ISO, *Open Distributed Processing - Reference Model - Enterprise Language*, ITU-T Recommendation X.901 | ISO/IEC 15414:2002, Oct. 2001.

58. ITU-T / ISO, *Open Distributed Processing - Reference Model - Part 2: Foundations*, ITU-T Recommendation X.902 | ISO/IEC 10746-2, Nov. 1995.

59. ITU-T / ISO, *Open Distributed Processing - Reference Model - Part 3: Architecture*, ITU-T Recommendation X.903 | ISO/IEC 10746-3, Nov. 1995.

60. ITU-T / ISO, *ODP Trading Function: Specification*, ITU-T Recommendation X.950 | IS 13235-1, 1997.

61. H. Jonkers, M.E. Iacob, M. Lankhorst, P. Strating, "Integration and Analysis of Functional and Non-Functional Aspects in Model-Driven E-Service Development", *Proceedings 9th IEEE International Conference on Enterprise Distributed Object Computing (EDOC 2005)*, IEEE Computer Society Press, Sept. 2005, pp. 229-238.

62. J. Jürjens, "A UML statecharts semantics with message-passing," *Proceedings of the 2002 ACM Symposium on Applied Computing*, ACM Press, 2002, pp. 1009-1013.

63. H. Kremer, *Protocol Implementation: Bridging the gap between Architecture and Realization*, Ph.D. thesis, University of Twente, The Netherlands, Oct.1995.

64. I. Kurtev, *Adaptability of Model Transformations*, Ph.D. thesis, University of Twente, The Netherlands, May 2005; http://purl.org/utwente/50761

65. I. Kurtev and K. van den Berg, "A Synthesis-Based Approach to Transformations in an MDA Software Development Process", *Model Driven Architecture: Foundations and Applications*, Technical Report TR-CTIT-03-27, Centre for Telematics and Information Technology, University of Twente, June 2003.

66. D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann, "Specification and Analysis of System Architecture Using Rapide," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, IEEE Computer Society Press, Apr. 1995, pp. 336-355.

67. D. Luckham and J. Vera, "An Event-Based Architecture Definition Language," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, IEEE Computer Society Press, Sept. 1995, pp. 717-734.

68. Microsoft Corporation, *Microsoft .NET Remoting: A Technical Overview*, July 2001; http://msdn.microsoft.com/library/en-us/dndotnet/html/hawkremoting.asp

69. H.D. Mills, D. O'Neill, R.C. Linger, M. Dyer, and R.E. Quinnan, "The Management of Software Engineering," *IBM Systems Journal*, vol. 19, no. 4, 1980, pp. 414-477.

70. E. Di Nitto and D. Rosenblum, "Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures," *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, IEEE Computer Society Press, May 1999, pp. 13-22.

71. OASIS, *OASIS - Committees - OASIS UDDI Specifications TC*; http://oasis-open.org/committees/uddi-spec/doc/tcspecs.htm

72. Object Management Group, *Model driven architecture (MDA)*, ormsc/01-07-01, July 2001.

73. Object Management Group, *Common Object Request Broker Architecture: Core Specification, Version 3.0*, formal/02-12-06, Dec. 2002.

74. Object Management Group, *Event Service Specification*, *Version 1.2*, formal/04-10-02, Oct. 2004.

75. Object Management Group, *CORBA Component Model, v3.0*, formal/02-06-65, July 2002.

76. Object Management Group, *MDA-Guide, V1.0.1*, omg/03-06-01, June 2003.

77. Object Management Group, *Meta Object Facility (MOF) 2.0 Core Specification*, ptc/03-10-04, Oct. 2003.

78. Object Management Group, *Meta Object Facility (MOF) Specification Version 1.4*, formal/02-04-03, April 2002.

79. Object Management Group, *MOF QVT Final Adopted Specification*, ptc/05-11-01, Nov. 2005.

80. Object Management Group, *Unified Modelling Language: Object Constraint Language version 2.0,* ptc/03-10-04, Oct. 2003.

81. Object Management Group, *UML 2.0 Superstructure*, ptc/03-08-02, Aug. 2003.

82. Object Management Group, *UML Profile for Enterprise Distributed Object Computing Specification*, ptc/02-02-05, Feb. 2002.

83. Object Management Group, *Unified Modelling Language (UML) Specification: Infrastructure, Version 2.0*, ptc/03-09-15, Sept. 2003.

84. Object Management Group, *Unified Modelling Language (UML) Specification, Version 1.5*, formal/03-03-01, March 2001.

85. Object Management Group, *Notification Service Specification*, v1.0.1, OMG document formal/02-08-04, Aug. 2002.

86. Object Management Group, *IDL to Java Language Mapping, v1.2*, formal/02-08-05, Aug. 2002.

87. Object Management Group, *Getting Specs and Products*; http://www.omg.org/gettingstarted/specsandprods.htm#GetProds

88. Object Management Group, *Trading Object Service Specification*, V1.0, formal/00-06-27, May 2000.

89. D. Quartel, L. Ferreira Pires, and M. van Sinderen, "On Architectural Support for Behaviour Refinement in Distributed Systems Design," *Journal of Integrated Design and Process Science*, vol. 6, no. 1, Society for Design and Process Science, 2002.

90. D. Quartel, *Action relations Basic design concepts for behaviour modelling and refinement*, Ph.D. thesis, University of Twente, The Netherlands, Feb. 1998.

91. D.A.C. Quartel, L. Ferreira Pires, M. van Sinderen, H.M. Franken, and C.A. Vissers, "On the role of basic design concepts in behaviour structuring", *Computer Networks and ISDN Systems*, vol. 29, no. 4, 1997, pp. 413-436.

92. Sun Microsystems, Inc., *JSR-000224 Java API for XML-Based RPC 2.0*, June 2003; http://www.jcp.org/aboutJava/communityprocess/edr/jsr224/

93. Sun Microsystems, Inc., *Java Web Services Developer Pack (Java WSDP)*; http://java.sun.com/webservices/jwsdp/index.jsp

94. J. Schot, *The role of Architectural Semantics in the formal approach of Distributed Systems design*, Ph.D. thesis, University of Twente, The Netherlands, Feb. 1992; http://purl.org/utwente/17886

95. M. van Setten, S. Pokraev, and J. Koolwaaij, "Context-Aware Recommendations in the Mobile Tourist Application COMPASS", *Adaptive Hypermedia and Adaptive Web-Based Systems: Third International Conference (AH 2004),* Lecture Notes in Computer Science, vol. 3137, Springer-Verlag, pp. 235-244, Aug. 2004.

96. R. Silaghi, F. Fondement, and A. Strohmeier, "Towards an MDA-Oriented UML Profile for Distribution", *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004)*, IEEE Computer Society, Sept. 2004, pp. 227-239.

97. R. Silaghi, "MDA Refinements along Middleware-Specific Concern-Dimensions", *Proc. of the 1st Doctoral Symposium at the 5th ACM/IFIP/USENIX International Middleware Conference, Middleware 2004 Companion*, ACM Press, Oct. 2004, pp. 309-313.

98. M. van Sinderen and L. Ferreira Pires, "Protocols versus objects: can models for telecommunications and distributed processing coexist?", *Proceedings Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, IEEE Computer Society Press, October 1997, pp. 8-13.

99. M. van Sinderen, *On the Design of Application Protocols*, Ph.D. thesis, University of Twente, The Netherlands, March 1995.

100. M. van Sinderen, L. Ferreira Pires, C.A. Vissers, and J.-P. Katoen, “A design model for open distributed processing systems”, *Computer Networks and ISDN Systems,* vol. 27, no. 8, 1995, pp. 1263-1285.

101. D.V. Steward, “The Design Structure System: A Method for Managing the Design of Complex Systems”, *IEEE Transactions on Engineering Management*, vol. 28, 1981, pp. 71-74.

102. Sun Microsystems, Inc., *Java™ 2 Platform Enterprise Edition Specification, v1.4*, Nov. 2003; http://java.sun.com/j2ee/1.4/docs/index.html

103. Sun Microsystems, Inc., *Enterprise JavaBeans Specification 2.1*, July 2002; http://java.sun.com/products/ejb/docs.html

104. Sun Microsystems, Inc., *Java(TM) Message Service Specification Final Release 1.1*, 2002; http://java.sun.com/products/jms/docs.html

105. Sun Microsystems, Inc., *J2ME Mobile Information Device Profile (MIDP)*; http://java.sun.com/products/midp/

106. A. Sutcliffe, *The Domain Theory: Patterns for Knowledge and Software Reuse*, Lawrence Erlbaum Associates, 2002.

107. C. Szyperski, *Component software: beyond object-oriented programming,* 2nd ed., Addison-Wesley, 2002.

108. B. Tekinerdogan, *Synthesis-Based Software Architecture Design*, Ph.D. thesis, University of Twente, The Netherlands, March 2000; http://purl.org/utwente/17903

109. The Open Group, *DCE 1.1: Remote Procedure Call*, Catalog number C706, Aug. 1997; http://www.opengroup.org/dce/

110. The Parlay Group, *The Parlay Group – Specifications*; http://www.parlay.org/en/specifications

111. D. Varró, “A Formal Semantics of UML Statecharts by Model Transition Systems”, *Proceedings ICGT 2002: International Conference on Graph Transformation,* Lecture Notes in Computer Science, vol. 2505, Springer-Verlag, 2002, pp. 378-392.

112. C.A. Vissers, L. Ferreira Pires, D. A. Quartel, and M. van Sinderen, *The Architectural Design of Distributed Systems*, Lecture Notes, University of Twente, The Netherlands, Nov. 2002.

113. C.A. Vissers, M. van Sinderen, and L. Ferreira Pires, “What makes industries believe in formal methods”, *Proceedings of the 13th International Symposium on Protocol Specification, Testing, and Verification (PSTV XIII)*, Elsevier Science Publishers, 1993, pp. 3-26.

114. C.A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma, “Specification styles in distributed systems design and verification”, *Theoretical Computer Science*, vol. 89, 1991, pp. 179-206.

115. C.A. Vissers and L. Logrippo, "The importance of the service concept in the design of data communications protocols", *Proceedings Fifth IFIP WG6.1 International Conference on Protocol Specification, Testing and Verification (PSTV V)*, June 1985, pp. 3-17.

116. J.N. Warfield, "Binary Matrices in System Modeling", *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 3, 1973, pp. 441-449.

117. M. Wegdam, *Dynamic Reconfiguration and Load Distribution in Component Middleware*, Ph.D. thesis, University of Twente, The Netherlands, June 2003; http://purl.org/utwente/41469

118. M. de Weger, *Structuring of Business Processes: An architectural approach to distributed systems development and its application to business processes*, Ph.D. thesis, University of Twente, The Netherlands, 1998; http://purl.org/utwente/17905

119. R. Wieringa, "A survey of structured and object-oriented software specification methods and techniques," *ACM Computing Surveys*, vol. 30, no. 4, ACM Press, 1998.

120. World Wide Web Consortium, *SOAP Version 1.2 Part 1: Messaging Framework*, W3C Recommendation, June 2003; http://www.w3.org/TR/soap12-part1

121. World Wide Web Consortium, *Web Services Description Language (WSDL) 1.1*, W3C Note, March 2001; http://www.w3.org/TR/wsdl

122. A. Yassine and D. Braha, "Complex Concurrent Engineering and the Design Structure Matrix Method", *Concurrent Engineering*, vol. 11, no. 3, 2003, pp 165-176.

Index

Samenvatting

Een recente trend inzake het ontwerpen van gedistribueerde applicaties is het scheiden platformonafhankelijke en platformspecifieke aspecten in verschillende modellen volgens een systematische aanpak. De voornaamste voordelen van deze benadering zijn gelegen in de mogelijkheid om verschillende platformspecifieke modellen (PSMs) af te leiden van hetzelfde platformonafhankelijke model (PIM), en om het modeltransformatieproces en de realisatie van de gedistribueerde applicatie op bepaalde (*middleware*) doelplatformen deels te automatiseren. Hiermee kunnen de initiële ontwikkelkosten gereduceerd en kwaliteit van de resulterende software verbeterd worden. Daarnaast wordt met deze benadering een basis gelegd voor het faciliteren van de evolutie en migratie van softwareoplossingen, en dus voor het beheersbaar maken van de kosten van het onderhoud van gedistribueerde applicaties.

Een gerelateerde prominente ontwikkeling is Model-Driven Architecture (MDA). In het kader van MDA wordt hard gewerkt aan *enabling* technologieën en technieken voor model-gedreven ontwerp, zoals metamodellering (MOF), taaldefinitie- en uitbreidingsmechanismes (bijv. UML en UML *profiles*), modeltransformatietalen (MOF *Query/View/Transformation*), ondersteuning met softwaregereedschappen en interoperabiliteit van gereedschappen. Weinig aandacht is er echter tot nu toe voor de methodologische en architecturale onderbouwing van platformonafhankelijke ontwerpen.

In het bijzonder kan de *state-of-the-art* in model-gedreven ontwerpen op de volgende punten bekritiseerd worden:

- er is een gebrek aan richtlijnen voor het kiezen van abstractiecriteria en modelleerconcepten die toegepast kunnen worden in platformonafhankelijke ontwerpen;
- er is weinig methodologische ondersteuning voor het scheiden van platformonafhankelijke en platformspecifieke zaken, waardoor de effectieve toepassing van PIMs en PSMs wordt beperkt;

- het onderscheid tussen PIM en PSM is grof en niet voldoende om de diversiteit van applicatieeisen en platformeigenschappen te addresseren;
- er is weinig aandacht voor platformeigenschappen gedurende het ontwikkeltraject als geheel, wat tot gevolg kan hebben dat modellen onvoldoende platformonafhankelijk zijn en applicaties geen acceptabele kwaliteitsattributen hebben;
- de gedragsaspecten van ontwerpen worden grotendeels buiten beschouwing gelaten; en
- ontwerpoperaties voor het PIM-PSM traject zijn niet precies gedefinieerd, hetgeen de effectieve toepassing van deze operaties in modeltransformaties in de weg staat.

Dit proefschrift beschrijft een ontwerpbenadering voor het ontwikkelen van gedistribueerde applicaties, met inachtneming van bovengenoemde problemen en vooral gericht op *middleware* platformonafhankelijkheid. De voorgestelde benadering bestaat uit:

- een *ontwerpproces*, dat leidt tot applicatieontwerpen op verschillende niveaus van abstractie en platformonafhankelijkheid;
- een *notie van abstract platform*, waarmee de platformeigenschappen die relevant zijn voor een applicatieontwerp op een gegeven niveau van platformonafhankelijkheid expliciet gemaakt worden;
- een *verzameling ontwerpkwaliteitscriteria* voor het definiëren van een abstract platform; en
- een *ontwerpraamwerk*, waarmee de ontwerper ondersteund wordt bij het definiëren van abstracte platformen en platformonafhankelijke ontwerpen. Dit ontwerpraamwerk bestaat uit twee delen: (1) een *verzameling elementaire ontwerpconcepten* die gebruikt wordt om zowel abstracte platformen als corresponderende platformonafhankelijke ontwerpen te beschrijven op verschillende niveaus van platform-onafhankelijkheid, en (2) *ontwerpoperaties* die gebruikt worden in transformaties om de verschillende niveaus van platform-onafhankelijkheid te overbruggen. Het ontwerpraamwerk stelt ontwerpers in staat om uitspraken te doen over de *conformance* van modellen op verschillende niveaus van platformonafhankelijkheid.

Het ontwerpproces is in onze ontwerpbenadering gestructureerd in een *voorbereidingsfase* en een *uitvoeringsfase*. In de voorbereidingsfase identificeren (en zonodig definiëren) ontwerpers de gewenste abstractieniveaus voor modellen, hun abstracte platformen en de modelleertalen die gebruikt gaan worden. Ontwerpers kunnen bovendien transformaties identificeren of definiëren tussen gewenste modelniveaus. De resultaten van de voorbereidingsfase worden gebruikt in de uitvoeringsfase, waarin modellen van een applicatie worden gecreëerd, gebruik makend van de gekozen modelleertalen en abstracte platformen.

De hoofdonderdelen van onze benadering worden geïllustreerd met een case study waarin contextbewuste mobiele diensten worden ontworpen. We definiëren drie modelniveaus: een niveau voor het specificeren van platformonafhankelijke diensten, een niveau voor het ontwerpen van platformonafhankelijke diensten en een niveau voor ontwerpen van platformspecifieke diensten. Speciale aandacht is er voor het representeren en transformeren van gedragsaspecten van diensten.

Resumo

Nos últimos anos, o desenvolvimento de aplicações distribuídas tem sido marcado pela separação da descrição dos aspectos dependentes de plataformas dos aspectos independentes de plataformas em diferentes modelos. Os principais benefícios desta abordagem devem-se: (i) à possibilidade de produzir diferentes modelos dependentes de plataformas (PSMs) a partir de um mesmo modelo independente de plataformas (PIM) e (ii) à possibilidade de automação parcial do processo de transformação e realização de uma aplicação distribuída. Desta forma, os custos iniciais de desenvolvimento podem ser reduzidos, assim como a qualidade das realizações pode ser melhorada. Além disto, esta abordagem forma uma base para facilitar a evolução e a migração de soluções de software, contribuindo então para reduzir os custos de manutenção para aplicações distribuídas.

Uma importante iniciativa que adota esta abordagem é a Arquitetura Baseada em Modelos (*Model-Driven Architecture*) (MDA). No contexto da iniciativa MDA, muitos trabalhos tratam das tecnologias e técnicas básicas para o desenvolvimento baseado em modelos, incluindo técnicas para metamodelagem (MOF), mecanismos de definição e extensão de linguagens (como, por exemplo, UML e seus *profiles*), linguagens de especificação de transformação de modelos (MOF *Query/View/Transformation*), e suporte para construção e integração de ferramentas de desenvolvimento. Os fundamentos metodológicos e arquiteturais do desenvolvimento de aplicações distribuídas de forma independente de plataformas têm recebido pouca atenção.

Mais especificamente, as atuais abordagens para desenvolvimento baseado em modelos podem ser criticadas nos seguintes pontos:

– há poucas diretivas para a seleção de critérios de abstração e conceitos para a modelagem de aplicações de forma independente de plataformas;

- há pouco suporte metodológico para a distinção entre aspectos dependentes de plataformas e aspectos independentes de plataformas, o que é prejudicial à exploração benéfica da separação entre PIMs e PSMs;
- a distinção PIM-PSM é insuficiente para lidar com a diversidade de requisitos de aplicação e características de plataformas;
- pouca atenção é dada para o papel de características de plataformas na trajetória de desenvolvimento, resultando em modelos com níveis de independência de plataforma inaceitavelmente baixos ou software com outras qualidades indesejáveis;
- os aspectos comportamentais de aplicações são frequentemente ignorados, e;
- manipulações de modelos que levam de PIMs a PSMs não são bem definidas, o que prejudica o desenvolvimento de transformações entre estes modelos.

Esta tese propõe uma abordagem para o desenvolvimento de aplicações distribuídas que ataca os problemas apresentados acima, concentrando-se na independência de aplicações com relação a plataformas de middleware. Esta abordagem consiste em:
- um *processo de desenvolvimento*, que resulta em modelos de uma aplicação em diferentes níveis de abstração e independência de platafomas;
- o conceito de *plataforma abstrata*, que define características de plataformas que são relevantes para a descrição de aplicações em um certo nível de independência de plataformas;
- *critérios de qualidade* para a definição de plataformas abstratas; e,
- um *framework*, que auxilia projetistas na definição de plataformas abstratas e modelos independentes de plataformas. Este *framework* é divido em duas partes: um *conjunto de conceitos básicos*, que são usados em diferentes níveis de independência de plataforma para descrever tanto plataformas abstratas quanto os modelos que dependem destas, e *manipulações de modelos*, que são usadas em transformações para relacionar diferentes níveis de independência de plataforma.

O processo de desenvolvimento é estruturado em uma fase de *preparação* e uma fase de *execução*. Na fase de preparação, os projetistas identificam (e, quando necessário, definem) os níveis de modelos necessários, assim como as plataformas abstratas e as linguagens de modelagem a serem usadas. Além disto, nesta fase, projetistas também podem identificar ou definir transformações entre níveis de modelos. Os resultados da fase de preparação são usados na fase de execução, na qual modelos de uma aplicação são criados usando-se linguages de modelagem específicas e plataformas abstratas.

Os principais aspectos da abordagem proposta nesta tese são ilustrados com um estudo de caso que trata o desenvolvimento de serviços móveis

sensíveis ao contexto do usuário. Três níveis de modelos são definidos: um nível de especificação de serviços independente de plataformas, um nível de projeto de serviços independente de plataformas, e um nível de projeto de serviços dependente de plataformas. A representação e a transformação de aspectos comportamentais dos serviços são enfatizadas neste estudo de caso.

Publications by the author

During the development of this thesis, the author has published various parts of his work in the following papers (listed in reverse chronological order):

- J.P.A. Almeida, R. Dijkman, L. Ferreira Pires, D. Quartel, and M. van Sinderen, "Model Driven Design, Refinement and Transformation of Abstract Interactions", *International Journal of Cooperative Information Systems (IJCIS)*, World Scientific, to appear.
- J.P.A. Almeida, M.-E. Iacob, H. Jonkers, and D. Quartel, "Model-Driven Development of Context-Aware Services", *6th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, Lecture Notes in Computer Science, vol. 4025, Springer, to appear.
- J.P.A. Almeida, M.-E. Iacob, H. Jonkers, M. Lankhorst, and D. van Leeuwen, "An Integrated Model-Driven Service Engineering Environment", *2nd International Conference Interoperability for Enterprise Software and Applications (I-ESA 2006)*, Springer, to appear.
- J.P.A. Almeida, L. Ferreira Pires, and M. van Sinderen, "Abstract Platform and Transformations for Model-Driven Service-Oriented Development", *Proceedings of the 2nd International Workshop on Model-Driven Enterprise Information Systems (MDEIS 2006) at ICEIS 2006,* INSTICC Press, Portugal, to appear.
- M. van Sinderen, J.P.A. Almeida, L. Ferreira Pires, D. Quartel, "Designing Enterprise Applications using Model-Driven Service-Oriented Architectures", *Enterprise Service Computing: From Concept to Deployment*, Robin G. Qui, ed., Idea Group, to appear.
- J.P.A. Almeida, R. Dijkman, L. Ferreira Pires, D. Quartel, and M. van Sinderen, "Abstract Interactions and Interaction Refinement in Model-Driven Design", *Proceedings of the 9th IEEE EDOC Conference (EDOC 2005)*, IEEE Computer Society Press, Sept. 2005, pp. 273-286.
- J.P.A. Almeida, L. Ferreira Pires, M. van Sinderen, "Dependencies between Models in the Model-driven Design of Distributed Applications", *Proceedings of the Joint Workshop on Web Services and Model-Driven Enterprise Information Systems (WSMDEIS 2005)*, INSTICC Press, Portugal, May 2005, pp. 95-109.

- J.P.A. Almeida, M. van Sinderen, D. Quartel, and L. Ferreira Pires, "Designing Interaction Systems for Distributed Applications", *IEEE Distributed Systems Online*, vol. 6, no. 3, Mar. 2005.
- J.P.A. Almeida, "Model-driven Design of Distributed Applications", *Workshop Proceedings of the 2004 International On The Move to Meaningful Internet Systems 2004: OTM 2004 Workshops (OTM 2004 Ph.D. Symposium)*, Lecture Notes in Computer Science, vol. 3292, Springer, Oct. 2004, pp. 854-865.
- J.P.A. Almeida, R. Dijkman, M. van Sinderen, and L. Ferreira Pires, "On the Notion of Abstract Platform in MDA Development", *Proceedings Eighth IEEE International Conference on Enterprise Distributed Object Computing (EDOC 2004)*, IEEE Computer Society Press, Sept. 2004, pp. 253-263.
- J.P.A. Almeida, L. Ferreira Pires, and M. van Sinderen, "Costs and Benefits of Multiple Levels of Models in MDA Development", *Proceedings of the 2nd European Workshop on Model-Driven Architecture with Emphasis on Methodologies and Transformations*, technical report no. 17-04, Computing Laboratory, University of Kent, Canterbury, UK, Sept. 2004, pp. 12-20.
- J.P.A. Almeida, R. Dijkman, M. van Sinderen, and L. Ferreira Pires, "Platform-Independent Modelling in MDA: Supporting Abstract Platforms", *Proceedings Model-Driven Architecture: Foundations and Applications 2004 (MDAFA 2004),* Linköping University, Linköping, Sweden, June 2004, pp. 219-233. Revised version appeared in Lecture Notes in Computer Science, vol. 3599, Springer, June 2005, pp. 174-188.
- J.P.A. Almeida, M. van Sinderen, L. Ferreira Pires and M. Wegdam, "Platform-independent Dynamic Reconfiguration of Distributed Applications", *Proceedings IEEE 10th International Workshop on Future Trends in Distributed Computing Systems (FTDCS 2004)*, IEEE Computer Society Press, May 26-28, 2004, pp. 286-291.
- A. Gavras, M. Belaunde, L. Ferreira Pires and J.P.A. Almeida, "Towards an MDA-based Development Methodology for Distributed Applications", *Proceedings of the 1st European Workshop on Model-Driven Architecture with Emphasis on Industrial Applications (MDA-IA 2004)*, CTIT Technical Report TR-CTIT-04-12, University of Twente, The Netherlands, March 2004, pp. 71-81. Also appeared in *Software Architecture: First European Workshop (EWSA2004)*, Lecture Notes in Computer Science , vol. 3047, Springer, May 2004, pp. 230-240.
- J.P.A. Almeida, M. van Sinderen and L. Ferreira Pires, "The Role of the RM-ODP Computational Viewpoint Concepts in the MDA Approach", *Proceedings of the 1st European Workshop on Model-Driven Architecture with Emphasis on Industrial Applications (MDA-IA 2004)*, CTIT Technical Report TR-CTIT-04-12, University of Twente, The Netherlands, March 2004, pp. 43-51. Revised version appeared in *Workshop on ODP for Enterprise Computing (WODPEC 2004) Proceedings*, technical report no. ITI-04-07, University of Málaga, Spain, 2004, pp. 28-35.
- L. Ferreira Pires, M. van Sinderen, C.R.G. de Farias and J.P.A. Almeida, "Use of Models and Modelling Techniques for Service Development", *Digital Communities in a Networked Society: eCommerce, eGovernment and eBusiness,* M.J. Mendes, R. Suomi, and C. Passos, eds., Kluwer Academic Publishers, 2004, pp. 441-456.

- J.P.A. Almeida, M. van Sinderen, L. Ferreira Pires, D. Quartel, "A Systematic Approach to Platform-Independent Design based on the Service Concept", *Proceedings Seventh IEEE International Conference on Enterprise Distributed Object Computing (EDOC 2003)*, IEEE Computer Society Press, Sept. 2003, pp. 112-123.
- J.P.A. Almeida, L. Ferreira Pires, and M. van Sinderen, "Web Services and Seamless Interoperability", *Proceedings of the First European Workshop on Object-Orientation and Web Services (held at ECOOP 2003, Darmstadt, Germany)*, IBM Research Report, RA220 Computer Science, IBM Research Division, July 2003, pp. 4-9.
- J.P.A. Almeida, M. van Sinderen, L. Ferreira Pires, and D. Quartel, "The Role of the Service Concept in Model-driven Applications Development", *Middleware 2003 Companion: Proceedings of the Workshop on Model-driven Approaches to Middleware Applications Development (MAMAD 2003) at the ACM/IFIP/USENIX International MIDDLEWARE Conference 2003*, Pontifícia Universidade Católica do Rio de Janeiro (PUC-RJ), Brazil, June 2003, pp. 288-296.
- J.P.A. Almeida, M. van Sinderen, L. Ferreira Pires, and M. Wegdam, "Handling QoS in MDA: A Discussion on Availability and Dynamic Reconfiguration", *Proceedings of the Workshop on Model Driven Architecture: Foundations and Application (MDAFA 2003)*, CTIT Technical Report TR–CTIT–03–27, University of Twente, The Netherlands, June 2003, pp. 91-96.

Model-Driven Design of Distributed Applications

João Paulo Andrade Almeida

The model-driven design approach described in this thesis aims at supporting designers in managing the complexity of distributed application design and evolution.

In this approach, different aspects of a distributed application are described throughout the design process using models. This thesis proposes a technique that allows designers to build application models that are – to a certain extent – independent of the technologies with which applications can be implemented. These technologies include the so-called middleware platforms, which are used to cope with distribution and to exploit distribution beneficially.

A cornerstone of the approach is the notion of abstract platform. An abstract platform is an abstraction of the characteristics of potential technology platforms which are assumed by application designers at a certain point of the design trajectory. By choosing abstract platforms carefully, a designer is able to obtain application models that do not have to be modified as a consequence of the evolution of technology platforms, and that can be used as a starting point for realizations on different platforms.

We define criteria for abstract platform definition and propose a design framework for abstract platforms and platform-independent application models. This framework is based on the concepts of service and abstract interaction, and includes design operations to transform application models through the various levels of abstraction and platform-independence.

The main aspects of the approach are illustrated with a case study involving the design of context-aware mobile services.

ISBN: 90-75176-42-2

CTIT
Centre for Telematics and Information Technology

Telematica Instituut