Architectural Style-based Modeling and Simulation of Middleware for Mobile Systems

Ping Guo

Dissertation
in Computer Science

submitted to the

Faculty of Computer Science,
Electrical Engineering and Mathematics
University of Paderborn

in partial fulfillment of the requirements for the degree of

doctor rerum naturalium
(Dr. rer. nat.)

Paderborn, November 2006

Abstract

Today, mobility is one of the most important market and technology trend within information and communication technology. As the demand for rapid deployment of dependable mobile applications increases, middleware for mobile systems is emerging as one of the most active areas of system research in mobility. The middleware is a set of distributed software services that exists between distributed operating systems and mobile applications. The key to the middleware is to provide support across the mobile application domains, help application developers overcome the complexity and problems brought by mobility, and enhance dependability and usability of developed mobile applications. The criticality and pervasiveness of middleware for mobile systems is continually growing. However, the design and development of the middleware are difficult tasks, and it is not easy to ensure the quality of a developed middleware. This is mainly caused by the increasing complexity of the middleware. In addition, the great diversity of this area makes it very difficult for the designers to reuse the already established design knowledge or successful experience when building new systems. All these make the design process quite inefficient and unpredictable, and therefore risking the project.

"One man's magic is another man's engineering". Engineering design is much more routine than innovative. Founding on this fundamental notion in software engineering, we develop an architectural style-based approach to deal with the problems in the thesis. We build architectural styles for a class of related middleware. The style represents a common form of design, which originates from the results that practitioners have achieved in one area. The style is formulated to repeat successes and avoid failures from previous projects. When building a new middleware, the designers and developers do not need to explore all possible alternatives for its supported architecture. Instead, they can use the architectural style that is effective for the middleware. They can define the design as instances of the style, or they can use the style as a reference model for further improvement and development. By structuring the design space for a family of related middleware, the style can drastically simplify the process of building a middleware, reduce costs of implementation through reusable infrastructure, and improve system integrity through style-specific analysis and checks.

We develop the approach based on UML-like meta modeling and graph transformation techniques to support sound methodological principles, powerful modeling, formal analysis and refinement. The approach consists of several main parts: the modeling language that supports specification of the style and mobility, the refinement formalization that ensures that an abstract style is correctly refined to a concrete one, as well as the consistency check framework that validates behavioral consistency between two styles on different abstract layers. With the Fujaba simulation tool support, we also develop a style-based engineering process that helps us to efficiently develop correct and consistent styles. Besides, it allows a seamless integration of our approach into the well-known object oriented design. By providing a concrete example of how to construct the style for a class of related middleware, and how to use the style to help the design and development of a new middleware, we show that the architectural style-based approach is useful and practical.

To Frauke and Edgar,
my German parents

Acknowledgements

Pursuing the doctoral work is like taking a long, long journey for me. I need to climb over the hills, break through the brambles and thorns, before I can finally come to the end. I have been often frustrated and exhausted when I still could not see one corner of the peak after traveling night and day. I have also experienced many happiness and excitement after reaching every small target. Along the way, I have got a lot of guidance, support, encouragement and companionship from many people, to whom I am full of gratitude.

Above all, I am very grateful to my advisor, Prof. Dr. Gregor Engels, for all the dedication and support he has given at every stage of my study from my first arrival in Paderborn. After working several years in industry, I have started with my exploration in academy as a complete beginner at his working group. With many industrial projects in head, I have often seen problems from a different view from the researchers. It has been a quite difficult task to build common understandings between us, which has caused him many headaches. During these unforgettable three and a half years, he has taught me not only about research, but also about life. He has significantly contributed to the dissertation with his comprehensive knowledge, penetrating perspectives, and consistent patience. Thanks to his strict requirements and uncountable suggestions, I have the possibility to make the dissertation such a one that I am satisfied with.

Dr. Reiko Heckel (presently at University of Leicester) also deserves a great deal of thanks, who has supervised me at the early stage of my study. He has guided me into the research area of software modeling and graph transformation systems, starting from basics and going in deep. I have benefited a lot from his generosity and collaboration. He has given me many good ideas and suggestions. Especially, the main modeling and simulation framework proposed in the thesis is based on his original idea.

During the research, I have also got many useful suggestions and hints from other people. I would particularly like to thank Rick Kazman (University of Hawaii and Software Engineering Institute) for letting me understand much better the role of architectural styles in software engineering area. Thanks also to Jaakko Kangasharju (University of Helsinki), the designer and developer of Wireless CORBA,

for the discussions and suggestions.

I would also like to thank the Fujaba team for the support and help. Especially thanks to Leif Geiger for the guides to Dobs, and to Lothar Wendehals for the guides to Fujaba.

I would sincerely like to thank all my colleagues, both former and present, of Database and Information Systems research group. I have learnt a lot from the experienced ones, such as how to write papers, how to make presentations, how to work, etc. I would like to mention Stefan Sauer, Tim Schattkowsky, Alexander Förster, Marc Lohmann, Jan Hendrik Hausmann, Jochen Küster, Ralph Depke, Katharina Mehner, Sebastian Thöne and Alexey Cherchago. It has been also my pleasure to work with Hendrik Voigt (and his American football), Christian Soltenborn (and his music), Baris Güldali (and his Salsa), Martin Assmann, Jan Christopher Bals and Fabian Christ. Besides, I have always got immediate support from our technician Friedhelm Wegener when meeting problems with computers and infrastructures. Thanks also to our secretary Beatrix Wiechers for her help.

My work has been financially supported by the International Graduate School of Dynamic Intelligent Systems. I would like to thank the whole graduate school team, for offering a stimulating and supportive environment for study as well as various opportunities and activities, from which I have benefited a lot. Especially thanks to Astrid Canisius for the many, many help I have gotten, to Eckhard Steffen for his encouragement and understanding.

I would also like to thank all my friends and neighbors in PHW2A for their companionship. Especially thanks to Elina Hotman, ChengYee Low, Madhura Purnaprajna, Su Zhao and Andreas Ziermann for proofreading of this manuscript.

I deeply thank my family for their support and understanding. My grandfather has taught me to be optimistic even in the darkest and hardest time, to be appreciative to every experience in my life even to tortures and difficulties. My parents have always encouraged me to try new things that I dream of. My sisters, my uncle HanShu, and my aunt XiaoQing have also supported me a lot.

Last but not the least, I would like to thank Flüge family, my family in Germany, especially Gregor, Frauke and Edgar, for their endless patience, encouragement, care, support, and all they have done for me.

Ping Guo
Paderborn, November 2006

Contents

Chapter 1

Introduction

1.1 Motivation: middleware for mobile systems

Distributed systems [142] enable us to use the best combination of hardware and software components for an enterprise. However, it is difficult to construct a coherent and operational distributed system that integrates the needed components. Middleware is originally designed to help manage the complexity and heterogeneity caused by the distribution characteristics of distributed systems [50, 142]. It integrates distributed heterogenous components and makes components interoperable. Middleware is layered between network operating systems and application components [19, 27]. It adds mechanisms and services that are much more specialized than those provided by the operating system. It enables application engineers to abstract from the implementation of error-prone and complex low-level details, such as concurrency control, transaction management and network communication, and allows them to focus on application requirements [50, 124]. The construction of a large class of distributed systems can be hence simplified by leveraging middleware. In addition, middleware leads to faster and cheaper system development and enhances the quality of systems. Besides have being rapidly adopted in industry [32], middleware platforms are getting popular in the academy research area [50]. Microsoft's DCOM (Distributed Component Object Models), Sun Microsystems' EJB (Enterprise JavaBeans), OMG's CORBA (Common Object Request Broker Architecture) are the examples of most popular middleware models.

Today, mobility is one of the most important market and technology trend within information and communication technology. With the fast development of high-speed wireless communication technologies and component miniaturization technologies, new types of mobile applications and mCommerce have emerged. At the same time, distributed systems are evolving into mobile systems, which can be seen as a special kind of distributed system designed for mobile, wireless communication environments. Mobility represents a total meltdown [121] of the stability assumptions (e.g. network structure, network connection, power supply, CPU, etc.) associated with traditional distributed computing. The main difference is caused

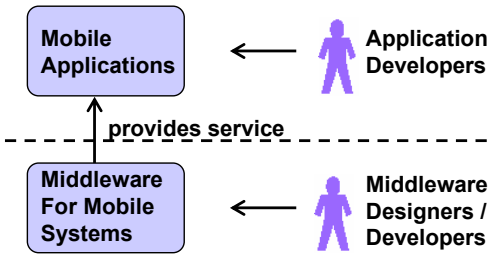

Figure 1.1: Middleware and applications

by the possibility of roaming and wireless connection. Roaming implies that, since devices can move to different locations, their computational context (network access, services, permissions, etc.) may change, and the mobile hosts are resource limited. Wireless connections are generally less reliable, more expensive, and provide smaller bandwidth, and they come in a variety of different technologies and protocols. All these result in a very dynamic software architecture, where configurations and interactions have to be adapted to the changing context and relative location of applications.

Mobility poses new requirements and complexity for application developers (Fig. 1.1) through defining a very challenging target execution environment. As the demand for rapid deployment of dependable new mobile applications increases, middleware is emerging as one of the most active areas of system research in mobility [121]. Many middleware platforms and paradigms for mobile systems [29] have been created. The key to middleware for mobile systems is to provide support (Fig. 1.1) across the mobile application domains, help application developers overcome the complexity and problems brought by mobility, and enhance dependability and usability of developed mobile applications.

The criticality and pervasiveness of middleware for mobile systems is continually growing. However, the design and development of the middleware is a difficult task, and it is not easy to ensure the quality of a developed middleware. Middleware designers and developers (Fig. 1.1) have some serious problems to cope with. The middleware is becoming increasingly complex with the fast development of wireless technologies and the increase of mobile application requirements. The managed components are increasing in scale and in scope. The requirements for the middleware are becoming more and more complex. Increasing system complexity typically brings the following problems:

- longer design and development time;
- more complex assembly due to number of components and number of people involved;
- increased cost and time for testing;
- increased maintenance costs.

Overall, this results in an increased time to market for the developed system. And development and maintenance costs are also increased in order to ensure the quality of the system. However, the current research area of middleware for mobile systems has concentrated on developing new types and patterns of middleware, and there is nearly no work on the design and development process of the middleware.

At the same time, to build large complex systems you must have predictability. We can not afford to be ad hoc in design. However, there is no common agreement or understanding of the middleware for mobile systems. The middleware platforms present a great diversity:

- they have distinct functionalities and provide various services to applications;
- they use very different design concepts and strategies;
- the aimed wireless network could be different (e.g. nomadic network or ad-hoc network);
- they exist because of mobility or in spite of mobility or both;
- the moving units can be logical or physical mobile or both;
- the definition of context and the level of context-awareness can be very different.

On one side, designers have to manage the huge diversity of design techniques, concepts, implementation technologies of the middleware. On another side, it is very difficult for the designers to reuse the already established design knowledge or successful experience when building new systems. This makes the design process quite inefficient and unpredictable, and thereforey risking the project.

1.2 Problems with architecture-centric approaches

Engineering [18] is the systematic application of scientific knowledge in creating and building cost-effective solutions to practical problems in the service of mankind. Software engineering [18, 140, 137] is that form of engineering that applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems. Generally speaking, software engineering researchers seek better ways to develop and evaluate software [135]. They are motivated by practical problems, and key objectives of the research are often quality, cost, and timeliness of software products. “One man’s magic is another man’s engineering” (Robert Heinlein). Engineering design is much more routine than innovative [134, 137]. It depends upon the existence of a mature discipline, such as standard notation for recording and communicating designs, established procedures, published processes, design standards, small number of unit operations, widely disseminated handbooks, etc [137, 88, 116].

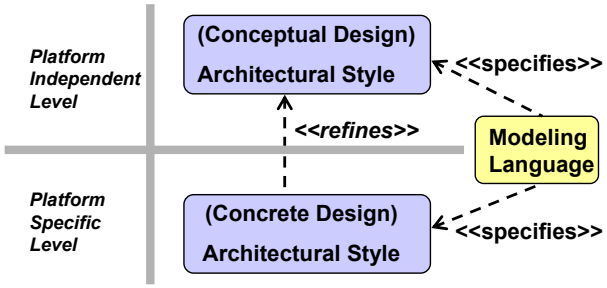

Figure 1.2: The architecture-centric approach

Software architecture has grown in importance to be a key discipline within software engineering [137, 115, 16]. Architecture-centric development directs on developing large, complex systems in a manner that facilitates robustness of products, economy of the development process and rapid time to market. It focuses on the construction of more abstract architectural elements, or, models, but not the program source code. A model is a reduced representation of the system, from a given viewpoint [23]. This enables designers to abstract away unnecessary details and concentrate on the main concerns and problems of the system. The model is coarse-grained enough so that the design of complex systems can be represented in an easier understandable form than the actual system. Besides helping understanding complex problems and solutions, the model communicates understanding between designers and developers. It can be also used as a guide to implementation.

One of the key issues in software development, like in all other engineering areas, is to ensure that the product delivered meets its requirements. Another key purpose of models is to allow the reasoning about important properties of the software before actually building it. Developers can manipulate the models in significantly more sophisticated ways than they can code. Model-based analysis such as model checking and simulation takes place at the early stage within the process of software construction, thus ensuring the quality of the system already at the modeling level. Besides, testing is often used in software engineering for validating properties.

To support architecture-based development, software architecture modeling languages [92, 36] and their accompanying toolsets have been proposed to provide notations for specifying and analyzing the architectural models of software systems. Modeling languages (Fig. 1.2) provide abstractions that are adequate for modeling or specifying a large system, while ensuring sufficient detail for establishing properties of interest.

Refinements (Fig. 1.2) are the basic steps in the architecture centric software development process. Starting from an abstract description of the system, stepwise refinements yield more and more concrete specifications that should finally be directly implementable on a machine. For example, an abstract or a conceptual

architecture is generated from user requirements, which mainly covers core functionality components. Later, a more concrete architecture is created that integrates non-functional requirements like security concepts and design specific aspects. In general, a conceptual architecture is smaller and easier to understand, while a concrete architecture reflects more design or implementation concerns.

"Architecture is design, but not all design is architecture." [36] That is, many design decisions are left unbound by the architecture and are happily left to the discretion and good judgement of downstream designers and implementers. The architecture establishes constraints on downstream activities, and those activities must produce artifacts – finer grained design and code, that are compliant with the architecture, but architecture does not define an implementation.

Architectural style (Fig. 1.2) is an important concept in the software architecture area. A critical challenge for software engineering is to capture system designs and reuse established design knowledge when building new systems. One way to do this is to define an architectural style for a collection of related systems. Indeed, the use of patterns [100] and styles of design is pervasive in many engineering disciplines. An established shared understanding of the common forms of design is one of the hallmarks of a mature engineering field [137]. The architectural style of software engineering determines a coherent vocabulary of system design elements and rules for their composition. Styles are often developed systematically to solve difficult architectural problems at the conceptual level. Most styles originate when novel classes of software systems are being designed or existing systems are being adapted to unforeseen circumstances and/or environments. They are formulated and evolved to consistently repeat successes and avoid failures from previous projects. New architectures can be defined as instances of specific styles. By structuring the design space for a family of related systems a style can, in principle, drastically simplify the process of building a system, reduce costs of implementation through reusable infrastructure, and improve system integrity through style-specific analysis and checks.

Both software architectures and middleware technologies focus on helping the development of large-scale, component-based system, but they address different stages of the development lifecycle. Software architecture is an abstract model of a system that highlights the system's critical conceptual properties, while middleware enables the system's realization and ensures the proper composition and interaction of the implemented components. Some researchers are trying to coupling architecture modeling and analysis approaches with middleware technologies [94, 90, 44]. For example, some architectural styles such as client-server and event publish-subscribe style can be used as a reference on the abstract layer for middleware development, therefore a middleware is a realization of an architecture style.

However, mobility has created additional complexity for computation and coordination. This makes the current architectural approaches and techniques hard to use [10]. The current architectural approaches have been designed for traditional distributed systems without consideration of mobility. They offer only a logical

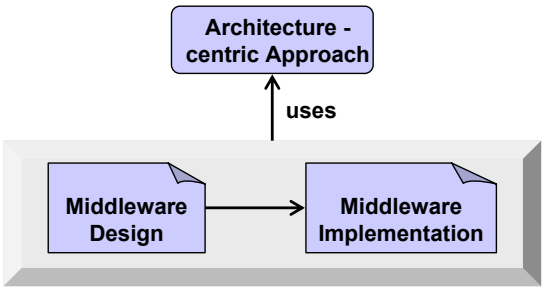

Figure 1.3: Objective of the thesis: design and development of the middleware using the architecture - centric approach

view of change; they do not take into account the properties of the "physical" distribution topology of locations and communications. It relies on the assumption that the computation performed by individual components is irrelative to location of the component, and that coordination mechanisms through connectors can be always transmitted successfully by the underlying communication network. In order to support mobility, the architectural and modeling approach needs to be adjusted.

1.3 Objectives of the thesis

The overall objective of the thesis is to develop an approach to help the design and development of middleware for mobile systems. Following the basic idea of the architecture centric development (Fig. 1.3), we will propose an approach that supports style-based modeling, refinement, and formal analysis (Fig. 1.4).

The modeling language In order to help to understand complex problems and solutions, the modeling language (Fig. 1.4) should capture the main aspects of the middleware. It should consider and model mobility explicitly, which includes two aspects: the architectural elements related to mobility, and how computation and coordination are influenced by mobility. The models should be represented in an easy understandable form, at the same time be clear and unambiguous, in order to truly help to understand the middleware, and enable easy and unambiguous communication about the middleware among different designers, developers and other stakeholders.

The architectural style The approach should support the modeling of architectural styles (Fig. 1.4) for the middleware, in order to solve the problems brought by diversity and unpredictability in the middleware area, enhance reusability and quality of the software, and simplify the development process. We will explore the specification of architectural styles, to see how we can build a common form of design, how to capture the already established design knowledge or successful experience for a class of related middleware.

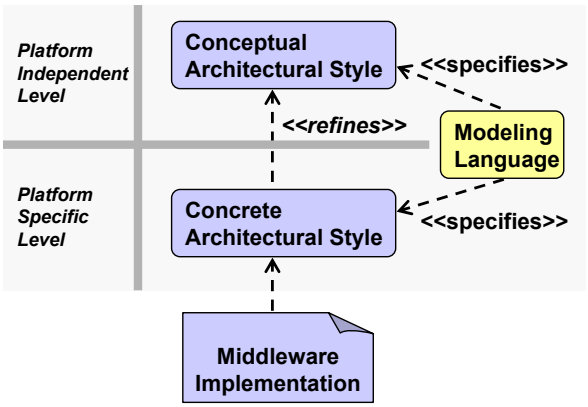

Figure 1.4: Objective of the thesis (in detail): architectural style - based modeling and analysis of the middleware

When building new middleware, the designers and developers can define the middleware design as instances of a specific style, or they can use the style as a reference model for further improvement and development. This will make the design process quite efficient and predictable.

Model-based analysis The approach should support model-based analysis. The approach is intended to model large, complex software system - middleware for mobile systems. The ability to evaluate and analyze the properties of such systems upstream, at an architectural level, can ensure the quality of the developed system in the specification process, and hence substantially lessening the cost of errors. Especially, automation and tool support of the analysis can help a lot since the specification is often complicatedly constructed and thus difficult to be checked manually.

Refinement The approach should support a stepwise refinement, which is generally required by a complex system in order to decrease complexity. The designers can start from a simpler, easier design, come to a more and more concrete, complete design through refinements. In the refinement process, it is difficult to ensure that the concrete specification is a correct refinement of the more abstract one. Especially, the correctness of the newly added architectural elements is difficult to be checked since it is not easy to directly map them to the abstract ones. Therefore, refinement correctness criteria should be provided to guarantee the correctness of a refined concrete specification. Besides, it is very helpful to have tool support and automation of the refinement process, since it is normally a complex task.

Consistency check One important issue of a model-based approach is to correctly bridge the gap between abstract models and concrete models, or even implementations. However, to validate whether a concrete model or an implemen-

tation is a correct refinement of an abstract model is usually difficult and complicated. The approach should provide a method to support the validation process between different abstraction layers. This is also important to prevent architectural decay, which is usually gradual divergence between an abstract model and a concrete model, or an implementation. Again, tool support and automation of the validation process are important.

1.4 Our approach

Correspondingly, we develop an approach based on UML-like meta models and graph transformation techniques to support sound methodological principles, powerful modeling, formal analysis and refinement. In detail, we can divide the approach into five main parts:

Requirements analysis The very first step of an architecture-centric approach is to characterize the problem domain, conclude the main characteristics and aspects to be modeled. Understanding the problem domains and their properties is a key to better understanding the needs of software architectures, architecture-based development, and architectural description. We develop a conceptual framework for understanding and comparing the different approaches, which is abstracted from specific implementation details and summarizes the main characteristics of the middleware. The generalization enables us to compare different approaches from a same point of view, and observe similarities and common aspects from the great diversities of the middleware. This builds a basis for constructing the architectural style of the middleware, which should originate from the results that practitioners have achieved in the area.

The modeling language The modeling language is based on the Graph Transformation Systems (GTSs) and UML-like metamodeling technique. We exploit the GTS for specifying the architectural style of the middleware for mobile systems, which includes the specification of mobility related elements. Visual representation and formal semantics are the main advantages of the modeling language. It has not only a graphic, easy understandable syntax, but also an operational semantics. It supports rich modeling techniques, such as separation of concerns, refinement, adaptability, extensibility, etc. It can describe complex structures and systems, and model concepts and ideas in a direct and intuitive way. At the same time, the formal semantics allows execution, analysis, transformation, automation and tool support.

The architectural style The great diversity of the middleware for mobile systems makes it quite difficult to build architectural styles for the middleware. It is difficult to explore commonalities across the middleware families. We observe that the middleware presents some commonalities among the basic aspects, which are summarized as the characteristics for the middleware during

the requirements analysis process. Therefore, we develop the architectural style through exploring the commonalities among these basic aspects, which include not only the concrete design specific models and patterns, but also the more abstract features brought by different definition of mobility, e.g., how to define the movement space and how components move. This makes our style vary greatly from the traditional architectural style for distributed systems. We also explore how to separate these aspects into the styles on different abstract layers. Such separation can decrease the complexity brought by mobility, help the designers and developers fully understand how mobility influences the middleware, and how to come to a complete design gradually. We illustrate the process of developing the style through constructing the style of the middleware for nomadic networks.

Simulation The operational semantics of the GTS allows us to execute the models and thus analyzing and validating the system through simulation. Simulating the dynamic behavior of the system allows the designers to execute the system and to play with specific scenarios. The designers can concentrate on the key aspects of the architecture. It can also detect errors and improve the confidence of the model. In such a validation process, it happens often that the result is wrongly judged as incorrect, although the specification is proved to be correct later on. This is normally caused by inputting an incomplete initial state, or the reference test result is wrong itself, both of which result in naturally wrong test result or wrong judgement. This makes the validation process itself error-prone and very inefficient. We solve this problem by constructing the minimal initial state and the expected test result with the help of an algorithm. Our approach can greatly enhance the correctness and efficiency of the validation process.

Besides supporting simulation, we also focus on providing a practical and usable process and environment to help the design and development of the style. The tool support of style specification, style refinement and other related concept is very important too. After comparing the available simulation tools, we choose the existing Fujaba [1] environment as our basis. Fujaba is a CASE (Computer-Aided Software Engineering) tool that aims to push graph transformation systems to a broader industrial usage. Fujaba allows a seamless integration of object-oriented software development and graph transformation systems, which facilitates greatly the designers and developers who are not familiar with graph transformation systems. With its support, we develop a style - based engineering process for efficient style development, which includes style specification, style validation, refinement consistency check, behavioral consistency check and code generation.

Refinement When developing the style, we use a stepwise refinement-based approach in order to decrease complexity and enhance reusability. We start from a simple abstract (i.e., conceptual) style that is refined to a more con-

crete one (platform-independent) with more design specific aspects, which is further refined to an even more concrete style (platform-specific). We develop refinement correctness criteria that check whether the concrete style is a correct refinement of the abstract one. The correctness criteria include both structural and behavioral ones. Especially, we formalize the refinement relationship between two abstract layers mainly based on the mapping of rules. We derive a simplified rule for a sequence of rules through a scenario based construction. The derived rule preserves the semantics of the sequence of rules, thus it can be used to substitute the complicated rule sequence. By using the derived simplified rule, we build a bi-directional mapping between a sequence of rule and another rule sequence. We also develop an algorithm to construct the derived rule.

The approach can check the correctness of the sequence of rules. It also enables us to check the correctness of newly added structural and behavioral elements through the scenario construction, which associates the completely different concrete elements with the abstract ones. Besides, our formalization can construct exactly the needed state graphs and transformation sequences for checking, which makes the approach efficient and practical to large systems. It also enables us to use the existing graph transformation simulation tool Fujaba as the basis for automating the refinement consistency check.

Consistency check We develop a framework that supports behavioral consistency check between models on different abstract layers. It checks whether the specified architectures in a same hierarchy belong to the same family of middleware, i.e., they should realize and provide the same functionalities, although they belong to different abstraction layers and use different design strategies. Besides, we automate the consistency check process with the support of Fujaba.

1.5 Structure of the thesis

The rest of the dissertation is organized as follows.

Chapter 2 pursues requirements analysis for the problem domain: the middleware for mobile systems. After developing the conceptual framework for understanding and describing the middleware, we illustrate the framework with some existing middleware examples. Afterwards, we conclude the aspects to be modeled, generalize commonalities for a class of related middleware. The results will be used to construct the architectural style of the middleware in the following chapters.

Chapter 3 at first identifies the detailed requirements for the architecture centric approach, which are classified into two groups: style specification and modeling

languages. We then review related work against these requirements, revealing existent achievements and open problems.

Chapter 4 gives an overview of our style-based approach after introducing the concept of the new style. We generalize the approach as a modeling and simulation framework, which structurizes the approach into several main parts: modeling language, style, style refinement and style simulation. We put the consistency check under simulation since it is developed based on the support of simulation. The next chapters follow this structure and explain each of these parts.

Chapter 5 explains the TGTS based modeling language, which includes the related background knowledge and how to model the required aspects of the style.

Chapter 6 illustrates the process of developing the style through constructing a concrete style as an example: the style of the middleware for nomadic networks. The style captures the commonalities of the class of related middleware, which are already concluded in Chapter 2 during the requirements analysis. In order to decrease complexity and enhance reusability, we separate the style on three different abstract layers, i.e., the conceptual style, the platform-independent concrete style and the platform-specific concrete style. Each of the style captures different architectural aspects or views. They are associated by refinement relationship. We take a concrete middleware, Wireless CORBA, as the example of the platform-specific concrete style.

Chapter 7 is about the refinement. We start with the detailed requirements for the refinement, which contain the properties to be preserved. The existing approaches and open problems are discussed. Afterwards, we explain how to correctly refine a TGTS-based abstract architectural style to a concrete one. We also check whether the requirements are fulfilled.

Chapter 8 clarifies the consistency check and simulation. We explain at first the framework for behavioral consistency check. After evaluating the suitability of existing graph transformation tools, we explain how to use Fujaba environment to specify and simulate the style. We illustrate then three ways to use Fujaba simulation for further style-based analysis and automation: to validate the model efficiently, to automate the refinement consistency check, and to automate behavioral consistency check. Finally, we summarize the style - based engineering process that helps the design and development of the style.

Chapter 9 concludes the thesis. We evaluate the results against the requirements listed in Chapter 3, explain the relevance to practice, summarize the contributions, and give an outlook on future work.

Bibliographical note

Preliminary results of this work have already been published in [63, 64, 72, 65].

Chapter 2

The Problem Domain

2.1 Overview

This chapter focuses on the requirements analysis of the middleware for mobile systems. The very first step of an architecture-centric approach is to characterize the problem domain, conclude the main characteristics and aspects to be modeled. Different problem domains require different software architecture descriptions [36], and it is not good and not sensible to have the same view(s) of software architecture for all the software systems. Understanding the problem domains and their properties is a key to better understanding the needs of software architectures, architecture-based development, and architectural description.

There exist some papers that survey the middleware for mobile systems. However, they focus mainly on technical details and design strategies of the different middleware platforms, which present a great diversity. What is missing is to apprehend the middleware at an abstract level, which should abstract from particular product characteristics and provide a conceptual framework for understanding and comparing the different approaches. In the chapter, we will develop a general framework for describing the middleware, which fills in the blank. The framework is abstracted from specific implementation details and summarizes the main characteristics of the middleware. The generalization based on the framework enables us to compare different approaches from a common point of view, and to observe similarities and common aspects from the great diversities of the middleware. This builds a basis for constructing the architectural style for the middleware that should represent a common form of design for a class of related middleware.

In order to explain the middleware for mobile systems better, we will pursue the chapter in a gradual way. We will start with characterization of the middleware for distributed systems in Section 2.2. We will characterize the middleware for distributed system in such a way that allows a better understanding of the middleware for mobile systems, and that emphasizes the main difference between the two kinds of middleware. Afterwards, we will explain the middleware for mobile systems in detail in Section 2.3, which includes the general framework for describing the middleware and its definition. Section 2.4 illustrates the framework with some

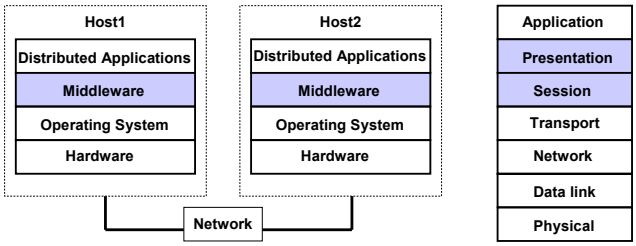

Figure 2.1: Left-hand side: middleware structure; right-hand side: ISO/OSI reference model

existing middleware examples. At the same time, we will compare the different middleware, conclude the important elements that decide or influence the design of the middleware. Besides, we will conclude commonalities of the design strategy of the middleware, which will be used to construct the architectural style of the middleware in the following chapters. Section 2.5 discusses which aspects should be modeled for the middleware.

2.2 Middleware for distributed systems

The term middleware first appeared in the late 1980s to describe network connection management software [124]. It became popular in the mid-nineties with the development and penetration of network technologies and distributed systems [142]. Distributed systems [142] enable us to use the best combination of hardware and software components for an enterprise. However, distributed systems are among the most complex artifacts human beings have ever constructed. It is difficult to construct a coherent and operational distributed system that integrates the needed components, due to the inherent heterogeneity, distribution problems of distributed systems. The developers need to deal with the complexity of networks and communications, different hardware platforms, operating systems and programming languages [19, 27]. Middleware is originally designed to simplify distributed system construction. Middleware is a set of distributed software services that exists between network operating systems and distributed applications [50, 124], as shown in Fig. 2.1. It adds mechanisms and services that are much more specialized than those provided by the operating system. It enables application engineers to abstract from the implementation of error-prone and complex low-level details, such as concurrency control, transaction management and network communication, and allows them to focus on application requirements. The construction of a large class of distributed systems can be hence simplified by leveraging middleware. The quality of the developed systems is also enhanced through using middleware.

The implementation of the middleware can take different forms with different design strategies. Generally, four main categories [50, 29] of middleware

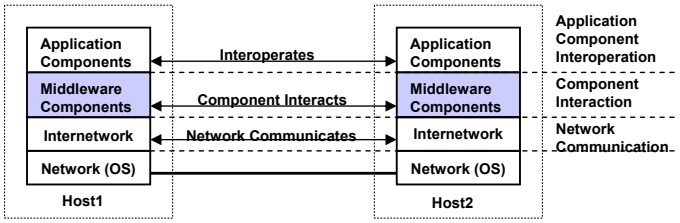

Figure 2.2: Middleware architecture

for distributed systems can be distinguished. They are transactional middleware, message-oriented middleware, Remote Procedural Call (RPC) middleware, and distributed object and component middleware [117, 124]. Transactional middleware supports transactions involving components that run on distributed hosts. The products in this category include BEA's Tuxedo [67]. Message-oriented middleware supports the communication between distributed system components by facilitating message exchange. Products in this category include IBM's MQSeries [59] and Sun's Java Message Queue [68]. RPC middleware is an important early middleware model that is based on Remote Procedure Calls (RPCs). The standard DCE (Distributed Computing Environment), specified by the Open Software Foundation, is completely identified with RPC middleware. Object and Component middleware [117] evolved from RPCs, whose most popular models are Microsoft's Component Object Models (COM, DCOM, COM+) [133], Sun Microsystems' Enterprise JavaBeans (EJB) [11], and OMG's Common Object Request Broker Architecture (CORBA) [110, 146].

In spite of the diversity of design strategies, the middleware can be still characterized at an abstract level that abstracts from particular product characteristics. For instance, some researchers [50, 29] characterize the middleware according to the requirements for distributed system construction, i.e., the difficulties that arise during distributed system construction. Accordingly, the middleware provides services to deal with network communication, component coordination, reliability, scalability and heterogeneity. This classification is quite complete and general enough to describe the functionalities of the middleware for distributed systems.

Nevertheless, we have a different target in the thesis. We want to characterize the middleware for distributed systems in such a way that allows a better understanding of the middleware for mobile systems, and that emphasizes the main difference between the two kinds of middleware. Therefore, we would like to emphasize the very basic requirements like interoperability and distribution transparency, but not the more advanced requirements like reliability and scalability. Consequently, we depict the middleware from four aspects (Table 2.1): A1 component interoperability, A2 component behavior, A3 network communication and A4 distribution transparency. The classification follows a top-to-down structure (Fig. 2.2), where the component interoperability is on the very top layer and it allows the application components to interoperate. It is also the very basic and kernel require-

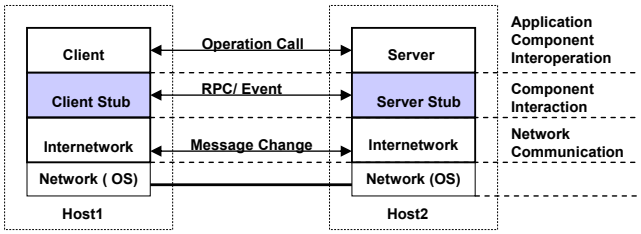

Figure 2.3: RPC-based middleware architecture

ment. The other aspects, i.e., component behavior and network communication, help the realization of the interoperability on the lower layers.

A1 Component interoperability Middleware [142] provides a view of a single interoperable coherent system and is tended to handle a collection of independent components. Middleware is sometimes called a glue technology because it is often used to integrate heterogeneous distributed components, and make components interoperable (Fig. 2.2), which means that a component on one system can access a component on another system. To achieve such an integration and interoperation, there are several main problems need to be addressed. First, components come out of different programming languages, data representations and operating systems. Secondly, components need to define interfaces at appropriate levels of abstraction in order to advertise the services that they provide. Thirdly, the components are located on different hosts, and the network and communications need to be solved.

Middleware solves these problems through providing the definition of services, standard programming interface and standard protocols. For example, object-oriented middleware CORBA [146] and DCOM [24] support the definition of object(component) services. The service provided by a component is encapsulated as an object and the interface of an object describes the provided service, which is a set of method calls defined through an IDL (Interface Definition Language). The interfaces defined in an IDL file serve as a contract between a server and its clients. Clients interact with a server by invoking methods described in the IDL.

IDL is designed to be independent of a particular programming language. The middleware can define binding to different programming languages. For example, CORBA [146] defines bindings to C, C++, Smalltalk, Ada, Java and OO-Cobol. These programming language bindings determine how object types with their attributes, operations and exceptions are implemented in server objects and how clients can make object requests and catch exceptions the server may raise. Through such definition of services and IDL, the different types of components have now a homogeneous definition, which allows a system construction through integrating legacy and commercial off-the-shelf components with newly built components.

A2 Component behavior In order to provide a single interoperable coherent system, one important aspect of middleware is to manage component behaviors, facilitate component interaction, and enable the cooperation of distributed components. More specifically, we can distinguish *inter-component behavior* and *intra-component behavior*. Given that components execute concurrently on distributed hosts, the middleware could support [50] multi-threads of control, single thread of control, or the both. The activation and deactivation of the component execution process need to be supported too. We call such behaviors that happens in the scope of one component *intra-component behavior*. It controls the execution state of the component.

On the contrary, *inter-component behavior* happens in the scope of a group of components. *Component interaction* (Fig. 2.2) is a synonym of inter-component behavior and we will use component-interaction in the rest of the paper. Component interaction covers component communication, collaboration, and coordination. These different aspects are not orthogonal and there is no clear boundary. For example, components need a method to communicate with each other through the network. The method can be message based, transactional based, event based, etc. The communication requires the coordination and synchronization between different actions and components, and the components collaborate with each other in order to perform a task.

Middleware can be distinguished through the supported component interaction patterns and paradigms [124, 50, 29]. For example, in distributed computing, the middleware can be distinguished in RPC (Remote Procedure Call) pattern, message based pattern, event-based pattern, etc. For example, in both DCOM [24] and CORBA [146], the interactions between a client process and an object server process are implemented as object-oriented RPC-style communication. Figure 2.3 shows a typical RPC structure. To invoke a remote function, the client makes a call to the client stub. The stub packs the call parameters into a request message, and invokes a wire protocol to ship the message to the server. At the server side, the wire protocol delivers the message to the server stub, which then unpacks the request message and calls the actual function on the object. In DCOM, the client stub is referred to as the proxy and the server stub is referred to as the stub. In contrast, the client stub in CORBA is called the stub and the server stub is called the skeleton. Sometimes, the term “proxy” is also used to refer to a running instance of the stub in CORBA.

A3 Network communication In a distributed system, components often locate on different hosts. Network communications (Fig. 2.2) are involved when the remote components interact. The interoperability of components on the networking layer is achieved through using standard networking protocols, which are often classified by the ISO/OSI (Fig. 2.1) model [50, 142]. Most middleware platforms are built on top of the transport layer (TCP or UDP are examples). Transport layer protocols generally provide services that handle the flow of data between

systems and provides access to the network for applications via sockets. Application engineers need to implement session, presentation, and application layer when programming at this level of abstraction. This is costly, error prone and time-consuming [50]. Middleware implements session and presentation layer, enables application engineers to request parameterized services from remote components and can execute them as atomic and isolated transactions. The transmitted parameters have often complex data structures. The presentation layer implementation of the middleware provides the ability to transform these complex data structures into a format that can be transmitted using a transport protocol, i.e., a sequence of bytes. This transformation is referred to as marshalling and the reverse is called unmarshalling.

A4 Distribution transparency One main functionality of middleware for distributed systems is to provide distribution transparency to the application. The distribution transparency can be further classified as interaction, network communication and location transparency. Interaction transparency means that the application components do not notice that they are interacting with other remote components, and the remote interaction will be performed by the middleware. Moreover, the application components are not aware of the network communication and the location of other components. For example, in an object-oriented middleware, if a client wants to perform some interaction with a server object, it issues a local call request to the middleware. The middleware locates the server-object by querying a database to discover the location of the requested object and then invokes the server-object. This processes involves the low layer network communication mechanisms from the middleware. The client and server do not need to know where the other is located.

From the description we can observe that the main requirements for middleware for distributed systems is to integrate heterogeneous distributed components and make the components a single interoperable coherent system. There are not so many other major requirements from the application side anymore. Therefore, the main functionality of the middleware is to provide component interoperability, which is the very basic and kernel requirement. Accordingly, we can define the middleware in the following as:

Def. 2.1 Middleware for distributed systems is a set of distributed software services that exists between network operating systems and distributed applications. It adds mechanisms and services that are much more specialized than those provided by the operating system. It integrates heterogeneous components, and makes the components a single interoperable coherent system.

2.3 Middleware for mobile systems

With the fast development of mobile systems and the popularity of mobile applications, a new kind of middleware: the middleware for mobile systems is created. The middleware for mobile systems is very different from the middleware for distributed systems, and it presents new functionalities and characteristics. This is caused by mobility that brings opportunities of new applications as well as difficulties for developers. In order to understand the new requirements for the middleware, we will at first introduce mobile systems and highlight the extents to which they differ from distributed systems. We will also introduce mobile applications and the problems with mobile application development. Afterwards, we will explain the middleware for mobile systems, give a general framework for describing the middleware, and define the middleware.

2.3.1 Mobile systems

With the development and penetration of wireless communication technologies, distributed systems are evolving into mobile systems, which can be seen as a special kind of distributed system designed for mobile, wireless communication environments. Mobility results in a total meltdown [121] of the stability assumptions (e.g. network structure, network connection, power supply, CPU, etc.) associated with traditional distributed systems. This is mainly caused by the possibility of roaming[1] and unstable wireless connections. Mobile systems differentiate from distributed systems mainly in the following aspects:

Unfixed network structure The fixed network structure of distributed systems does not hold for mobile systems anymore. Roaming nodes build an unfixed network structure in mobile systems. The network topology and structure are changing correspondingly when the mobile devices move to other locations. The current existing wireless networks can be divided as nomadic networks and Mobile Adhoc NETworks (MANET), which are defined based on whether there exists fixed infrastructure support or not. A nomadic network utilizes the infrastructure as access points to establish connectivity. Examples of this kind of network are Telecom networks like GPRS/GSM/UMTS with base stations, Wireless LAN (WLAN) with access point support, etc. In contrast, an ad hoc network does not have the infrastructure support. It uses direct neighbors as relays to connect to other nodes. The nomadic network has still a relative structured space where devices are allowed to move inside the scope of areas covered by access points. While the network structure of the MANET [114, 127] is completely unfixed, since the MANET allows unrestricted mobility of the terminals, as long as at least one terminal is within transmission range. Typical examples of the MANET are Wireless LAN

[1] Roaming means that devices can change its location.

without access point support, wireless PAN(Personal Access Network) including Bluetooth, IrDA, RFID.

Unstable network connection Wireless network connections are very unstable compared to fixed network connections. The wireless connection is highly variable in performance and reliability. The connection is rather intermittent. During roaming, devices can reach an area with no coverage or where the wireless signal is weak. The bandwidth may suddenly drop to zero and the connection may be lost. Besides, latencies in wireless networks are generally greater than in fixed networks, throughput will generally be lower and may vary unpredictably, and losses and errors are likely to be more frequent.

Resource-limited device Mobile devices vary from mobile PC, personal digital assistants, mobile phones, to smart cards, etc. For a given cost and level of technology, tradeoffs will be made between weight, power, size and computational resources such as processor speed, memory size, and disk capacity [123]. Although the capability and technologies of mobile devices are always in improvement, they will always be resource-limited relative to static devices.

Dynamic context Mobility defines a very dynamic execution environment. Context [121, 126, 154] represents the peculiar and novel aspect of mobile computing, to the point that some researchers characterize mobility as context-aware computing. There exist different definitions of context. For example, the researchers [125] define context to be the constantly changing execution environment, which includes

- *Computing environment* available processors, devices accessible for user input and display, network capacity, connectivity, and costs of computing.
- *User environment* location, collection of nearby people, and social situation.
- *Physical environment* lighting and noise level.

While some researchers [121] think that the context of a mobile unit is determined by its current location which, in turn, defines the environment where the computation associated with the unit is performed. The context may include resources, services, as well as other components of the system.

2.3.2 Mobile applications

Mobility brings great opportunities, and new types of mobile applications [143, 150] have emerged. It is a trend and of great business value to offer mobile applications to end-users. Here we introduce some important classes of mobile applications.

Nomadic applications In nomadic applications, mobile components make use of wireless networks connect to a fixed network infrastructure, such as the Internet, and they may suffer periods of disconnection while moving between points of connectivity. Such applications typically employ infrastructure networks like WLAN or GSM/GPRS networks.

Location-based services An important element of mobile computing lies in the device's interaction with location-based services [60]. That is, the ability to discover what services are available at a particular location and communicate with them. One popular example is the telecom GSM/GPRS location-based services, with which the users can get surrounding information based on the current location, such as traffic situation, close restaurants, gas stations, hospitals, ATMs, etc.

Context-aware applications One of the most important class of applications within mobile environments is context-aware applications[125, 126, 6], which operate in fluctuating environments and adapt themselves to provide the best level of service to the user. There exit different definitions of the context-aware applications[154, 125, 6], according to the degree of context-awareness presented to users (i.e., how much mobility should the users perceive), and who will activate the adaption, i.e., the predefined behavior in the application or the input from the user.

Mobile e-commerce Mobile e-commerce [143, 150] (also called mobile commerce or m-commerce) is defined as all activities related to a (potential) commercial transaction conducted through communications networks that interface with wireless (or mobile) devices. Mobile e-commerce has presented a wide range of interesting application types [60], which include reservation and ticketing systems that allow the user to book at nearby restaurants and taxi firms, and advertising applications allowing retailers to present their latest offers to customers close by. Automated tolling is another encouraging application area. For example, the ROBIN toll collection system uses the satellite-based GPS (Global Positioning System) to determine when a vehicle is on a toll road and charges the on-board device.

Collaborative applications Collaborative applications [95] mean that mobile nodes use the wireless network to interact with other mobile nodes that have come together at some common location. Although these applications may use infrastructure networks, they will often use ad hoc networks to support communication without the need of a separate infrastructure. Consequently, this collaborative style of application allows loosely coupled components to communicate and collaborate in a spontaneous manner. The hot-pot available in airports and meeting centers are examples of such applications. This kind of application is also very useful in emergency or in military case. For instance, in war fields, or at the scene of accident, the people can connect and communicate each other without the infrastructure support.

2.3.3 Problems with mobile application development

From the preceding introduced different mobile applications we can see that the mobile applications have much more complicated requirements than distributed applications since they are placed in a dynamic changing environment. They need to provide much more services to end-users. For instance, they need to provide continuous connectivity service when the users are roaming (Nomadic Applications), they need to discover what services are available at a particular location and communicate with them (Location-based Services), or do transactions with them (Mobile E-commerce), they need to operate in fluctuating environments and adapt themselves to provide the best level of service to the user (Context-aware Applications), they need to discover the available peers and to interact with them (Collaborative Applications), etc.

Correspondingly, it is very difficult to develop dependable mobile applications. The application developers need to integrate the dynamic changing, heterogeneous components together in order to perform the required tasks. They need to manage the movement of components, unstable wireless network connection, changing context. etc. In addition, they need to deal with different end-user requirements and provide different services and functionalities. For instance, how to present the context and the level of context-awareness is a problem to face. Some end-users want to hide the mobility and have continuous service in the mobile environment. Some users want to be aware of the context information, and the required content of the context varies a lot, too.

2.3.4 Middleware for mobile systems

Obviously, the middleware for distributed systems can not simplify and help the development of mobile applications so much anymore, since it does not provide enough services to help the application developers be released from the complicated mobility scenarios. Accordingly, a type of new middleware - middleware for mobile systems has been created. The key to middleware for mobile systems is to provide support across the mobile application domains, help application developers overcome the complexity and problems of mobility, and enhance the dependability and usability of developed applications.

As the demand for rapid deployment of dependable new mobile applications increases, middleware is emerging as one of the most active areas of system research in mobility [121]. Many middleware platforms for mobile systems [29, 60, 55] have been created. This is partly because that middleware can take advantage of the deployed software infrastructure while providing clean high-level programming abstractions in languages already available today. And it is better than developing a specialized language, which requires too much investment and entail unacceptable risks. Middleware also hides the protocol layer but makes explicit the key concepts involved in the development of mobile applications, e.g., the management of location data, event notification, quality of service assessment, adaptability, etc.

As a quite new research area, there is no common agreement or understanding of the middleware for mobile systems. The middleware platforms [29, 60, 55] present a great diversity:

- they have distinct functionalities and provide various services to applications;
- they use very different design concepts and strategies;
- the aimed wireless network could be different (e.g. nomadic network or adhoc network);
- they exist because of mobility or in spite of mobility or both [102];
- the moving units can be logical or physical mobile or both [102];
- the definition of context and the level of context-awareness can be very different;

There exist some papers [29, 60, 55] that survey the middleware for mobile systems. However, they focus mainly on technical details and design strategies of the different middleware. What is missing is to apprehend the middleware from an abstract layer, which should abstract from particular product characteristics and provide a conceptual framework for understanding and comparing the different approaches. In order to fill in this blank, we will explore a framework for describing the middleware. We will abstract from specific implementation details and focus on the important and general aspects presented by the middleware.

We will start from exploring and comparing the aspects abstracted for the middleware for distributed systems. The four aspects (Table 2.1) given in Section 2.2: A1 component interoperability, A2 component behavior, A3 network communication and A4 distribution transparency are adapted to A1 component interoperability, A2 component interaction, A3 wireless network communication and A4 context awareness correspondingly in the mobile system (Table 2.1). Besides these four items, we will mention other two aspects, which are specific for mobile systems. They are A5 space definition and A6 dynamic change.

A1 Component interoperability One of the main requirement for middleware for mobile systems is still to integrate heterogeneous distributed components, and make the components a single interoperable coherent system. However, the roaming components and unstable wireless connection make the integration much more complicated. Components may come and leave rapidly, and services that are available when we disconnect from the network may not be there anymore when we reconnect. Location is no longer fixed and mobile nodes can roam to other places where new services are available in a new surrounding. Depending on where we are and if we are moving, bandwidth and quality of the network connection may

	Middleware For Distributed Systems	Middleware For Mobile Systems
A1	**Component Interoperability** Integrate heterogeneous distributed components, make components interoperable	**Component Interoperability** 1. Integrate heterogeneous mobile components, make components interoperable 2. Provide other diverse functionalities
A2	**Component Behavior** 1. Inter-component behavior is important: process management, thread management 2. Static component Interaction	**Component Interaction** 1. Component collaboration and coordination in the dynamic environment are important 2. Dynamic component interaction
A3	**Network Communication** Treat disconnections as exceptions	**Wireless Network Communication** Special treatment: 1. Provide efficient and reliable communications 2. Support disconnection operation 3. Explore asynchronous communication paradigms for decoupled and opportunistic wireless connections
A4	**Distribution Transparency** Interaction, network communication, location transparency	**Context Awareness** Present diversities: 1. Distribution transparency 2. Location awareness 3. Network connection awareness 4. Context awareness
A5	**No**	**Space Definition** 1. Decide the allowed roaming style of components 2. Influence interaction and communication models of components
A6	**No** Only important for safety- and mission-critical systems	**Dynamic Change** 1. Changing context, location of components, network connection 2. Dynamic configurations and interactions

Table 2.1: The framework for middleware: comparison between the middleware for distributed systems and the middleware for mobile systems

vary greatly as well. The middleware needs to manage the roaming components, the unstable wireless connection, the dynamic context, etc.

Besides component interoperability, the middleware needs to provide other diverse functionalities to satisfy different application requirements. The middleware for mobile systems exists because of mobility or in spite of mobility or both. Some middleware hides the mobility completely from the application, and they provide continuous service when components roaming. Application components do not realize the wireless environment at all. While some middleware monitors the dynamic environment and reflects the context information to the applications.

A2 Component interaction Managing component behaviors is still one important task of middleware for mobile systems. However, it focuses much more on *component interaction* than intra-component behavior. Component interaction covers component communication, collaboration, and coordination. In a mobile environment, component collaboration and coordination are very critical and dynamic. How [121] to discover who is around for interaction with other components in the dynamic environment is an important characteristic that differentiates among middleware that support mobility. Some researchers [120] argue that coordination is central to understanding mobility, and the key of the middleware is to define the coordination.

In a mobile setting, the components involved in an interaction change dynamically due to their migration or connectivity patterns. This results in dynamic reconfiguration among these components. While the static assumption of network structure, network connection and execution context in distributed systems makes the component interaction quite static. The components involved for interactions are generally fixed. Such fixed configurations will not change till a dynamic change happens. And the messages for communication are always supposed to be successfully transmitted by the network.

A3 Wireless network communication The middleware deals with unstable wireless connections. Some middleware explores asynchronous communication paradigms for wireless communication, since the characteristics of wireless communication media (e.g., low and variable bandwidth, frequent disconnections, etc.) favor a decoupled and opportunistic communication paradigm: decoupled in the sense that computation proceeds even in the presence of disconnections, and opportunistic as it exploits connectivity whenever it becomes available. In this case, the communication happens only when network connection is available, and the component computation proceeds even in presence of disconnections. Some middleware offers efficient and reliable communications over the unstable wireless network connections. Some middleware explores mechanisms that allows continuous execution of application components when disconnection happens.

A4 Context-awareness The middleware for mobile systems presents great diversity according to different application requirements. Some middleware still holds distribution transparency, and they provide interaction, network communication and location transparency to the application. Some middleware provide location related and context information to the applications. The definition of context and the level of context-awareness can be very different.

A5 Space definition The definition of space is a key factor when dealing with mobility, since the space decides the allowed roaming style of components, influences the interaction and communication models of components. Generally, the targeted wireless network of a middleware decides the definition of spaces. For example, the nomadic network has a structured space, which is divided into regular patches with a base station supporting the communication needs of the components in that special area. Roaming among cells entails special handover protocols, i.e., communication behavior is tied to space structure. In the case of ad hoc networks, the space lacks structure but the distance metric is important because communication can take place only when components are within range.

A6 Dynamic change The stability assumptions in distributed systems make the software architecture static. Dynamic changes are generally only important for safety- and mission-critical systems, such as air traffic control or telephone switching systems, where shutting down and restarting the system incurs unacceptable delays and failures. On the contrary, dynamic change is a main characteristic of mobile systems.

The stability assumptions associated with traditional distributed computing is completely broken in mobile systems. When devices can move to different locations, their context may change, and that mobile hosts are resource limited. Wireless connections are very unstable. All these result in a very dynamic software architecture, where configurations and interactions have to be adapted to the changing context, relative location of components, and the network connection. Consequently, supporting for runtime modification is a key aspect of middleware for mobile systems.

Based on the understanding, we now define the middleware for mobile systems as:

Def. 2.2 Middleware for mobile systems is a middleware (see Def. 2.1) designed for mobile systems. It is a set of distributed software services that exists between network operating systems and mobile applications that use wireless networks. It adds mechanisms and services that are much more specialized than those provided by the operating system. It integrates heterogeneous mobile components, and makes the components a single interoperable coherent system. It has special consideration and treatment of locations, wireless network connections, and possibly context-related aspects.

Middleware is a relative concept, as it is closely related to distributed operating systems. Whether a given service is classified as middleware may change over time. A facility that is currently regarded as part of a middleware may become one part of operating system in the future, to simplify mobile system construction and to increase the commercial value of the operating system. For example, some researchers [153, 126] explore the operating system to support context-aware computing. In our context, we assume that the operating system provides simple communication means to connect to other computers, e.g., socket programming, and it does not provide further services to manage roaming components and wireless network connections.

2.4 Middleware for mobile systems: examples

After giving the general framework for describing the middleware for mobile systems in last section, we will illustrate the framework with some existing middleware examples, to see whether the framework is complete and abstract enough to depict the middleware. At the same time, we will compare the different middleware, conclude the important elements that decide or influence the design of the middleware. We will also conclude commonalities for a class of related middleware, which will be used to construct the architectural style for the middleware in the following chapters. This is the main objective of the thesis as outlined in Section 1.3.

In order to get more abstraction, we will not review individual middleware, but a class of middleware. We will classify the middleware according to the major adopted design strategy, which can be in most case identified through the component interaction model. We identify three classes of middleware here: event-based (or publish/subscribe) middleware, tuple space-based middleware and object component middleware. This classification is also often adopted by other researchers [29, 60, 55].

2.4.1 Event-based (or publish / subscribe) middleware

Event-based (or publish/subscribe) middleware belongs to message-oriented middleware that supports the communication between distributed system components by facilitating message exchange. Event-based [53, 157, 76, 95] or publish/subscribe model is often used for supporting message exchange between components in the mobile environment. Many event-based middleware for distributed systems have been adapted to support mobility. JEDI [43], REBECA [157], Siena [28], JECho [33], STEAM [95] are the examples of this kind of middleware.

A1 Component interoperability Client components use the middleware to send a message to a server component across the network. The message can be a notification about an event, or a request for a service execution from a server component.

The content of such a message includes the service parameters. The server responds to a client request with a reply-message containing the result of the service execution.

We can distinguish two categories of the models according to how to support mobility: nomadic roaming and ad-hoc roaming. Nomadic roaming means that the components can roam inside the nomadic network with the infrastructure support, while ad-hoc roaming means the components do not need the infrastructure support, and they come into contact when in proximity. Generally, the main functionality of the model supporting nomadic roaming is to provide continuous notification service when components are moving between points of connectivity. The main functionality of the model supporting ad-hoc roaming is to provide access to other components during a dynamic change of topology. REBECA [157], JEDI [43], Siena [28] and JECho [33] are the examples that support nomadic roaming. STEAM [95] supports ad-hoc roaming.

A2 Component interaction Components interact through pub/sub communication. Components can act both as producers and consumers, they are clients of the underlying event notification service, which brokers notifications between producers of information and subscribers of information. Consumers subscribe the events they are interested in and are informed by the notification service when the event occurs. The notification service is responsible for checking the event against all current subscriptions and delivering it to all users whose subscriptions match the event. This model supports undirected communication where the producers do not need to explicitly define the receiver. The loose coupling of producers and consumers is the prime advantage of this model.

The component interaction models are quite different depending the supported mobility models. The model for nomadic roaming often introduces centralized or intermediate components in order to handle disconnection and provide continuous notification service while the application components move from one access point to another. These additional components are responsible for buffering notifications when disconnected, or rerouting and delivering the notifications to different locations where the roaming client currently locates. Therefore, the component interaction behavior is tied to these components. For example, JEDI [43] includes a dynamic tree of dispatchers (the client can reconnect to any) for ensuring publish-subscribe information is retained as members connect and reconnect. The dispatching servers manage temporary storage for notifications. They coordinate that no duplicates are received and that the notifications are causally ordered. They are also responsible for routing the notifications to a new server, with whom the client is currently connecting. Another example is the REBECA [157] mobility model. It uses intermediate components as brokers that relocate and redirect the notifications to the roaming clients.

In contrast, STEAM [95] contains neither centralized components nor intermediate components, since the very dynamic change of topology in ad-hoc networks

makes such service not suitable anymore. The components maybe distributed over a potentially large geographical area. Thus they are unlikely to be able to maintain a permanent communication link to the servers. STEAM is based upon the concepts of group communication with publishers and subscribers belonging to the same group. Notably, STEAM exploits a proximity-based group communication, where a group is identified by both geographical and functional aspects. To apply for group membership, a component must firstly be located in the geographical area corresponding to the group and secondly be interested in the group in order to join. Therefore, the interaction is scaled by the proximity of publisher to subscriber, and events are only need to be forwarded within the scope of their proximity.

A3 Wireless network communication Event-based model has an asynchronous communication[2] style, which allows persistent communication between components that are not executing at the same time. That means, a message is stored even when its sender and its receiver are not executing after the message has been sent. It will be stored as long as necessary. This fits naturally to the intermittent and opportunistic wireless network connection. The message or event is buffered when the component disconnects, and it will be delivered when the component reconnects.

Especially, the proximity-based group communication of STEAM [95] limits the interaction between components to a certain geographical area. Subsequently, the behavior of communication connections between components becomes more predictable, and it allows the middleware to support reliable event delivery in a dynamic mobile environment.

A4 Context-awareness On one side, the middleware supporting nomadic mobility generally fosters a strategy that enables the seamless integration of the legacy distributed applications without rewriting them. Thus, hiding the mobility to the applications is one requirement of such middleware. In this case, the middleware still holds the network and location transparency. On another side, the middleware provides location-awareness to the application, in order to support the location-based applications.

At the same time, the middleware supporting ad-hoc mobility does not focus on providing location transparency. For example, STEAM includes a location service that uses sensor data to compute the current geographical location of its host machine and subsequently providing this location information to the middleware and to the local event producers and consumers. The sensors are generally equipped on some fixed points locating in specific geographical areas. To suit outdoor applications, STEAM also exploits a version of the location service that uses a GPS satellite receiver to provide latitude and longitude coordinates.

[2] After issuing a request, a client component can continue to perform its operations and synchronize with the server component until the server component responds.

A5 Space definition The middleware supporting nomadic mobility has a relative structured space where devices are allowed to move inside the scope of areas covered by access points. While the middleware supporting ad-hoc mobility does not have a structured space. For example, STEAM has been designed for IEEE 802.11-based, wireless local area networks(WLAN). The application components are allowed to move within the scope of a specific proximity and between proximities. Additionally, STEAM allows entities to define geographical scopes independently of the physical transmission range of the wireless transmitters. It supports a multi-hop event dissemination in which nodes residing within the boundaries of a proximity forward event messages. Members of the corresponding multicast group recognize the identifiers of these event messages and subsequently deliver them.

A6 Dynamic change The interaction and configuration between components is rather dynamic in a nomadic roaming model. The configuration changes dynamically when components move in or move out of area covered by access points, or when the wireless connection disconnects or connects. Especially, an interaction involves generally the central server or the intermediate components, as they provide continuous notification service. A client component can dynamically establish connections to the intermediate component within its current vicinity. The connection is not valid anymore when the client moves to other area. The configuration of the coordinating intermediate components changes too when the client component moves to other area. A new configuration of the coordinating intermediate components will be thus built.

The interaction and configuration in an ad-hoc network is much more dynamic, and it's direct interaction between the moving components. For example, STEAM supports dynamic interaction and configuration between components with the change of proximity. The configuration changes dynamically when components move in or move out of the proximity, or when the wireless connection disconnects or connects. The interaction is accordingly dynamically too since the interacting components are not fixed. A component can dynamically establish connections to other components within its current vicinity. An already established connection between two components can easily expire.

2.4.2 Tuple space-based middleware

Tuple space is an asynchronous communication model, which is effectively a shared distributed memory spread across all participating hosts that processes can concurrently access. As having decoupled nature that provides useful facilities for communication in wireless settings, tuple space has been explored for the coordination of mobile components. The middleware for mobile systems based on tuple space is created. L2imbo[45], Lime [102], TSpaces [156] and JavaSpace [141] are examples.

A1 Component interoperability Tuple space is a globally shared associatively addressed memory space used by distributed processes to communicate. If two processes need to communicate, a sender writes the data (a tuple) into the shared data space (the tuple space), while the receiver retrieves it from the same place. The tuple space acts as a virtual space between service providers (servers) and requesters (clients) through which they can exchange tasks, requests and data that they may use for future computation. We can roughly distinguish two categories of the models regarding how to support mobility: nomadic roaming and ad-hoc roaming. L2imbo[45], Lime [102] are designed for ad-hoc roaming. They focus on how to provide a dynamically changing context for computation in the presence of mobility. TSpaces [156] and JavaSpace [141] are mainly suitable for nomadic roaming with a distinct client-server style. The client is supposed to be running on a resource-limited device, while the server side is a resource-rich device.

A2 Component interaction The tuple space based middleware belongs to coordination based model, which is specialized in describing concurrent processes interactions, abstracting away the details of computation and formally defining constrains and conditions on the interactions, i.e., offering facilities for controlling synchronization, communication, creation and termination of computational activities. The tuple space model is data-oriented model that essentially concerns what happens to the data. The model analyzes how data are shared among agents and the state of the computation is defined in terms of both the values of the data and the actual configuration of the coordinated components. Linda [58] is a famous tuple space based coordination language designed for distributed systems. In Linda, the context for computation, i,e, the tuple space is permanent and fixed. In order to support ad-hoc mobility in the tuple-space based model, the people mainly focus on how to provide a dynamically changing context for computation. For example, LIME [102] breaks up the Linda central global tuple space into many separate tuple spaces, each of it is permanently associated to a mobile unit. It introduces rules for transient sharing of the individual tuple space based on connectivity, i,e, the co-located mobile units share the transient data, which changes dynamically according to connectivity and migration. Accordingly, the context for computation changes dynamically too. The interaction between components is called location-dependent transient interaction, which is conditional on the relative positions of components. More precisely, interaction is limited to the situations in which the components are in the communication range of each other. L2imbo[45] is similar to Lime in this aspect. Multiple tuple spaces can be created and used for the mobile unit. They are shared transiently between the co-located mobile units.

The nomadic roaming support in tuple-space based models follow normally a client-server coordination. There exist centralized tuple spaces, accessible through remote operations by multiple processes. The interaction behavior is tied to the server. For example, the TSpaces [156]from IBM allows the creation of multi tuple spaces, each of which exists only on a TSpaces server. When a client issues

an operation, information is sent to the server, which uses a lookup operation to find out the tuple space on which the operation needs to be performed and passes the operation and tuple operand to process. JavaSpaces [141] also implements a client-server coordination.

A3 Wireless network communication In the tuple space paradigm, the sender does not need to know who the receiver is or where it is working. Hence the communication is decoupled in time and space, i.e., senders and receivers do not need to be available at the same time, and mutual knowledge of their location is not necessary for data exchange. This fits to the decoupled and opportunistic style of wireless network communication. Decoupled in the sense that computation proceeds even in presence of disconnections, and opportunistic as it exploits connectivity whenever it becomes available.

A4 Context-awareness Lime provides both location-transparent and location-aware styles of data access, in order to support applications requiring different styles of programming. Lime fosters a style of coordination that reduces the details of mobility and distribution to changes to what is perceived as the local tuple space. This view simplifies application design in many scenarios. It relieves the designer from explicitly maintaining a view of the context consistent with changes in the configuration of the system.

At the same time, LIME also provides location-awareness through defining tuple location parameters. It supports getting location data from a GPS device. Besides, it also supports the awareness of the system configuration through a read-only, system-maintained tuple space, which contains information about the mobile units present in the community and their relationship.

A5 Space definition The middleware supporting nomadic mobility has a relative structured space where devices are allowed to move inside the scope of areas covered by access points. While the middleware supporting ad-hoc mobility does not have a structured space. For example, STEAM has been designed for IEEE 802.11-based, wireless local area networks(WLAN). The application components are allowed to move within the scope of a specific proximity and between proximities. In addition, STEAM allows entities to define geographical scopes independently of the physical transmission range of the wireless transmitters. It supports a multi-hop event dissemination in which nodes residing within the boundaries of a proximity forward event messages. Members of the corresponding multicast group recognize the identifiers of these event messages and subsequently deliver them.

A6 Dynamic change The context for computation, represented in Linda by the central permanent tuple space, is represented in LIME [102] by transient sharing of the tuple spaces carried by each individual mobile unit. Through this way, the global context for computation is defined by the transient community of mobile

units that are currently present. Since these communities are dynamically changing according to connectivity and migration, the context changes as well. Therefore, the computation is dynamically associated to the connectivity and migration of the components.

2.4.3 Object and component middleware

Object middleware evolved from RPC, which is a classical paradigm that supports remote procedural calling in distributed system. The object and component middleware for distributed systems has been adapted to support mobility too. The synchronous communication paradigm of the object-oriented middleware assumes a permanent connectivity that cannot be guaranteed in the ad-hoc network. Therefore, adaptations of traditional object middleware to mobile scenarios are usually targeted to nomadic roaming. Wireless CORBA [111], ALICE [66], RAPP [132] and DOLMEN [149] are examples of this kind of middleware.

A1 Component interoperability The object middleware supports the definition of object(component) services. The service provided by a component is encapsulated as an object and the interface of an object describes the provided service, which is a set of method calls defined through an IDL. The interfaces defined in an IDL file serve as a contract between a server and its clients. Clients interact with a server by invoking methods described in the IDL. Marshalling operation parameters and results is again by stubs that are generated from an interface definition.

The main functionality of the model supporting nomadic roaming is providing continuous invocation service when objects are moving between points of connectivity.

A2 Component interaction Object middleware [49] supports distributed object interaction, that is, a client object requests the execution of an operation from a server object that may reside on another host. This interaction evolved from Remote Procedure Call (RPC). The basic form of interaction is synchronous, that means the client object issuing a request is blocked until the server object has returned the response.

The object middleware supporting nomadic roaming often introduce centralized or intermediate objects in order to provide continuous invocation service while the application objects move from one access point to another. These additional objects are responsible for redirecting and delivering the invocations to different locations where the roaming object currently locates. Therefore, the interaction behavior is tied to these objects. For example, Wireless CORBA [111] and DOLMEN [149] introduce access bridges as intermediate objects that provide continuous invocation service. Handover protocol is used between access bridges for redirecting the invocations.

A3 Wireless network communication The synchronous communication paradigm of the object middleware does not fit to the unstable wireless connection naturally, unlike other asynchronous communication paradigms. The middleware supporting mobility has to deal with the unstable wireless connections.

There are two main solutions in the current available object middleware. One solution is to emulate the server object as much as possible in order to permit continued execution of application client objects. When disconnection happens, operations originally performed on server objects are performed on replicated or migrated objects in the cache. For example, ALICE [66] provides disconnected operation that lets clients replicate the server data and cache the replica locally. While the client and server are disconnected, client requests to the server are redirected to the replica stored on the client side. This allows the client to operate in spite of not having access to the server. Support for conflict detection and resolution is also provided.

Another solution is to provide reliable transmission over the wireless connection. For example, Dolmen [149] and Wireless CORBA redefine the inter-ORB protocol to enhance reliability and performance of object communication in the mobile environment. Messages over the wireless link are transferred by the adaptation layer that guarantees reliability and ordered delivery of messages. Dolmen [149] and Wireless CORBA maintain the state and forward messages for a disconnected terminal. There is a parameter in the message to indicate if the message should be redirected and handover should be pursued.

A4 Context-awareness The object middleware supporting nomadic roaming focuses on seamless integration of the legacy distributed applications without modifying them. Therefore, hiding mobility to the applications is one of the design goal of the middleware. The network and location information are hidden to the application. The application objects will not notify that they are doing invocation with the mobile objects. On another side, the middleware also provides location-awareness to the application, in order to support the location-based applications.

A5 Space definition The object middleware supporting nomadic mobility has a relative structured space where devices are allowed to move inside the scope of areas covered by access points.

A6 Dynamic change The interaction and configuration between objects is rather dynamic. The configuration changes dynamically when application objects move in or move out of area covered by access points, or when the wireless connection disconnects or connects. Especially, an interaction between application objects involves the intermediate objects, which provide continuous invocation connection service. An object can dynamically establish connections to the intermediate object within its current vicinity. The connection is not valid anymore when the client moves to other area. The configuration of the coordinating intermediate objects

	Middleware for Nomadic Network	Middleware for Ad Hoc Network
Examples	1. Event-based middleware JEDI, REBECA, Siena, JECho 2. Tuple space-based middleware TSpaces, JavaSpace 3. Object and component middleware Wireless CORBA, ALICE , RAPP, DOLMEN	1. Event-based middleware STEAM 2. Tuple space-based middleware L2imbo, Lime
Main Functionality	1. Provide continuous service to moving components 2. Handover protocol is often used	1. Provide access to other components 2. No need for handover
Component Interaction	1. Existence of central servers 2. Communication behavior is tied to central servers 3. Synchronous communication can be still used	1. Central servers are not suitable 2. Prefer geographical proximity group communication 3. Prefer asynchronous communication
Space Definition	1. Structured space 2. With infrastructure support 3. Mobility is restricted by the infrastructure	1. Space has no structure 2. Without infrastructure support 3. Mobility is not restricted by the infrastructure, but by proximity of other nodes

Table 2.2: Commonalities of the middleware

changes too when the application objects move to other area. A new configuration of the coordinating intermediate objects will be thus built.

2.4.4 Generalization of commonalities

We have described three important types of existing middleware for mobile systems using the given framework that includes: A1 component interoperability, A2 component interaction, A3 wireless network communication, A4 context awareness correspondingly in the mobile system, A5 space definition and A6 dynamic change. The framework allows us to abstract from specific implementation details and focus on the important and general aspects presented by the middleware.

Additionally, the generalization enables us to compare different approaches from a same point of view, and explore commonalities from the great diversities of the middleware. This builds a basis for constructing the architectural style for the middleware that should represent a common form of design for a class of related middleware.

For instance, we can see that *space* and *wireless connection* are the two most important aspects for the middleware to deal with mobility. Space decides allowed roaming style of components, influences the interaction and communication models of components, and hence deciding the preferable design strategy. Different treatment and consideration of wireless connection leads to various communication paradigms. At the same time, the component interaction and configuration are

associated to space and wireless connection. They are not static anymore as in traditional distributed systems. The change of space and wireless connection causes dynamic change of the interaction and configuration. Computation is also related to such dynamic change.

In addition, we also observe some commonalities of the reviewed middleware regarding *space definition*, *main functionality* and *component interaction / design pattern* if we consider the supported *wireless network* of the middleware (Table 2.2). Accordingly, we can roughly separate the middleware into two classes: the middleware for nomadic networks, and the middleware for ad hoc networks.

Space The targeted wireless network of the middleware generally decides the definition of space. Different wireless networks can have different definition of space that greatly influences the design and functionality of the middleware. The nomadic network has a structured space where components are allowed to move inside the scope of areas covered by access points. In contrast, an ad hoc network does not have a structured space. The distance metric is important because communication can take place only when components are within range.

Main functionality Generally speaking, the middleware for nomadic network focuses on how to provide continuous connectivity and other services when components move across the structured spaces where handover protocols are often used. In an ad hoc network, the main functionality of the middleware is to provide access to other components during a dynamic change of topology.

Component interaction / design pattern The component interaction behavior and the design strategy of the middleware share some similarities too for the same kind of network. The middleware for nomadic networks allows the existence of central servers to provide service to other components. Such servers can be fixed and other moving components can always communicate with them. Communication behavior is tied to central servers here. In contrast, the very dynamic change of topology in ad-hoc networks makes such servers not suitable anymore, since the components maybe distributed over a potentially large geographical area. Thus they are unlikely to be able to maintain a permanent communication link to the servers. The favorable component interaction model is also influenced by the network. Synchronous communication paradigms like RPC still can be adapted for nomadic networks, but they do not fit to the ad-hoc network anymore. Ad-hoc networks prefer asynchronous (non-blocking), proximity (geographical space) groups communication, and components most likely interact once they are in close proximity.

Inside these three aspects, we can also observe that space definition and functionality are more general than component interaction. That means, several different middleware platforms use different component interaction design

patterns but supports the same space definition and functionality. This is quite natural, since different middleware can be designed for the same class of mobile application, which is normally used in a typical wireless network. Thus, space definition and main functionality are decided by the targeted application and wireless network. At the same time, different middleware platforms can adopt specific component interaction design patterns to deal with the mobility model and realize the functionality.

2.5 Aspects to be modeled

In the preceding sections, we have summarized the characteristics of the middleware for mobile systems, especially by emphasizing the main difference between the middleware for distributed systems and the middleware for mobile systems. Based on the understanding, we will discuss now which aspects should be modeled for the middleware for mobile systems. We will consider the items listed in the general framework for capturing the middleware (Table 2.1). We separate the aspects into two parts: the very general aspects to deal with mobility, and the aspects to model the middleware.

2.5.1 Modeling mobility

We combine the discussion of space and wireless network connection together, since they cause the main difference between distributed systems and mobile systems.

Space and wireless network connection Recall the conclusion given in last section, *space* and *wireless connection* are the two most important aspects to deal with mobility. Space decides allowed roaming style of components, influences the interaction and communication models of components, and hence deciding the preferable design strategy. Different treatment and consideration of wireless connection leads to various communication paradigms. At the same time, the component interaction and configuration are associated dynamically to space and wireless connection. The change of space and wireless connection causes dynamic change of the interaction and configuration. Computation is also related to such dynamic change.

In fact, the main difference between distributed systems and mobile systems is caused by the possibility of roaming and wireless connection. Correspondingly, space and wireless connection are two very basic elements need to be considered in order to model mobility generally. Especially, they need to be treated as first class entities, i.e., they have a name and they are refinable. It is important for a model to be capable of dealing these elements throughout the software development life cycle, starting from the definition of the environment where mobility occurs, through designing and reasoning about a mobile system, and down to the tools provided to

programmers. Making the location and network connection explicit into the software architecture at the specification level enables us to formally reason about it in the same manner as we reason about other components in a distributed system.

More specifically, modeling space includes two basic aspects:

- the definition of space, and the relationship with other components and elements (structural).
- the allowed movement of components inside such spaces, the dynamic change of component interaction, configuration and computation when components move (behavioral).

Modeling wireless connection also includes two basic aspects:

- the definition of wireless connection, and the relationship with other components and elements (structural).
- the connection and disconnection of the wireless connection, and the influence to component interaction, configuration and computation (behavioral).

Dynamic change The stability assumptions in distributed systems make the software architecture static. Dynamic changes are generally only important for safety- and mission-critical systems, such as air traffic control or telephone switching systems, where shutting down and restarting the system incurs unacceptable delays and failures. On the contrary, dynamic change is a main characteristic of mobile systems. The configurations and interactions change dynamically according to the movement of the components, the change of wireless connection, the change of context that may include resources, services, etc. Consequently, modeling dynamic change is also a very basic aspect to deal with mobility. The behavioral part of modeling space and wireless connection in previous section is covered here. The behavioral part of component interoperability and context-awareness in the following sections is also covered here.

2.5.2 Modeling other aspects

Besides the mentioned general aspects to deal with mobility, we consider now the other aspects listed in the framework.

Component interoperability It is important to model the framework for component interoperability in the middleware in order to understand how the middleware makes the components a single interoperable coherent system, and how they handle a collection of independent components. In a mobile setting, the framework needs to manage the roaming application components, the unstable wireless connection, the dynamic context, etc. It also provides other specific functionalities to satisfy application requirements, for instance, continuous service or access to

other components. Thus, the main components that construct the framework need to be modeled. How the components cooperate to perform a task needs to be modeled too. These described aspects can be separated into structural and behavioral part respectively. Actually, component interoperability is related to component interaction very much, which provides underlying support to realize the needed framework, and it is very difficult to separate them. In most cases, modeling component interaction covers already component interoperability. Therefore, we will put component interoperability under *component interaction*.

Component interaction Component interaction is a very important aspect of the middleware. The main goal of middleware is to facilitate component interaction of components. Component interaction covers component communication, collaboration, and coordination. These different aspects are not orthogonal and there is no clear boundary. For example, components need a method to communicate with each other through the network. The method can be message based, transactional based, event based, etc. The communication requires the coordination and synchronization between different actions and components, and the components collaborate with each other in order to perform a task.

How to discover who is around for interaction with other components in the dynamic environment is an important characteristic that differentiates among middleware that supports mobility. In a mobile setting, the components involved in an interaction change dynamically due to their migration or connectivity patterns. This results in dynamic reconfiguration among these components. While the static assumption of network structure, network connection and execution context in distributed systems makes the component interaction quite static. The components involved for interactions are generally fixed. Such fixed configurations will not change till a dynamic change happens. And the messages for communication are always supposed to be successfully transmitted by the network.

Modeling component interaction should include several aspects: the roles played by the components during interaction, the methods used for communication, the coordination mechanisms, the influence on the coordination and communication caused by the movement of components and change of connectivity, etc.

Context-awareness The middleware for mobile systems presents great diversity with how to treat context-awareness. Some middleware still holds distribution transparency. Some middleware provides location related and context information to the applications. In addition, the definition of context and the level of context-awareness are various in different middleware. Hence, modeling context-awareness varies in different middleware platforms. There are two important aspects to model context awareness:

- The structural part of the context. There exist different definitions of context (see Section 2.3). In spite of that, the common aspect of context is: the context [121] of a mobile unit is determined by its current location which,

in turn, defines the environment where the computation associated with the unit is performed. The context may include resources, services, as well as other components of the system. Therefore, it is important to clarify what aspects belong to the context, and their relationship with location and wireless network connection, etc.

- The behavioral part of the context. It includes: how the migration or connectivity patterns changes the context, what will happen when the context changes. Usually, context changes cause dynamic adaption of the middleware. This leads back to the previously explained dynamic change. We think *dynamic change* is the key to model context-awareness. The other aspects are not so important since the are rather implementation related, for example how to present the context information to applications, and which degree of context-awareness should be presented to users (i.e., how much mobility should the users perceive), and who will activate the adaption, i.e., the predefined behavior in the application or the input from the user, etc.

Chapter 3

Related Work

3.1 Overview

In last chapter, we have developed a general framework for describing the middleware for mobile systems. The framework is abstracted from specific implementation details and emphasizes the main difference between the middleware for distributed systems and the middleware for mobile systems. Based on the framework, we have summarized the characteristics of the middleware for mobile systems, concluded the aspects to be modeled. Such abstractions and conclusions allow us to better understand the needs of the architecture centric approach, which will be used to help the design and development of the middleware, as outlined in Section 1.3.

In this chapter, we will at first identify the detailed requirements for the architecture centric approach, based on the understanding of the middleware. Principally, we will give the requirements for the architectural style that can be used to guide the design and development of the middleware. We will then review related work against these requirements, revealing existent achievements and open problems. In particular, we intend to explore a gap between the two research areas: architectural styles and the middleware for mobile systems. We will conclude that mobility has created additional complexity for computation and coordination, which makes the current styles hard to use for the middleware for mobile systems. In order to use the style to help the design and development of the middleware, we need new architectural styles that capture the main architectural aspects of the middleware. Correspondingly, the modeling language needs to be adjusted in order to support the specification of the new style.

The chapter is organized as follows. We give the detailed requirements for the architecture centric approach in Section 3.2, which are further organized into two groups: style specification and the modeling language. In Section 3.3, we at first survey styles that are related to the middleware for mobile systems. We evaluate whether those styles meet the architectural requirements of the middleware. Afterwards, we survey the current modeling languages, to see, what are the main problems of them for supporting the required style specification.

3.2 Requirements

In Section 1.3, we have identified roughly four main parts that the architecture centric approach should include, which are the modeling language, the architectural style, model-based analysis, refinement and consistency check. Based on our understanding of the middleware, we will now explore more detailed requirements for the architecture centric approach.

In an architecture centric approach, it is very difficult to separate the requirements for the style from the requirements for the modeling language, and those for the whole approach. The modeling language decides the specification capability and possible available methods and tools. The researchers generally put all the related requirements under the requirements for the modeling language [3, 5, 92, 93]. In order to provide a better framework to evaluate the related work and our approach later on, we will divide the requirements into two groups: style specification and the modeling language.

3.2.1 Requirements for style specification

As having outlined in Section 1.3, we need architectural styles for the middleware in order to solve the problems brought by diversity and unpredictability in the area, enhance reusability and quality of the software, and simplify the development process. The style stands for a common form of design for a class of related middleware. It is a a mechanism for categorizing architectures and for defining their common characteristics. Very basically, it should capture the main architectural aspects of the middleware. In Section 2.5, we have concluded the aspects should be modeled for the middleware. They are separated into two groups: the very general aspects to deal with mobility, i.e., space, wireless network connection and dynamic change, and other aspects that include component interaction covering component interoperability, the structural part of the context, and the behavioral part of the context (i.e., dynamic changes). Correspondingly, the style should cover these main aspects.

Type definition The style should have type definition of architectural elements, which is very important to achieve software reuse. The style can support reuse by modeling design elements like components as types in order to encapsulate functionalities, behavior definitions into reusable building blocks that can be instantiated multiple times. Besides this, type definition is also necessary in order to constrain the behavior of instances, which is very useful for type check in code generation and programming.

Style vocabulary The style should define the vocabulary of design elements for capturing the main characteristics of the middleware, which include the mobility related elements, e.g., space and wireless network connection. In fact, the main

difference between distributed systems and mobile systems is caused by the possibility of roaming and wireless connection (see Section 2.5). Correspondingly, space and wireless connection are two very basic elements need to be considered in the style. Especially, they need to be treated as first class entities, i.e., they have a name and they are refinable. Besides, the design elements include those elements that construct the component interaction pattern and the middleware framework, for instance, the roles played by the components during interaction like the client and server for RPCs, other middleware construction components used to manage roaming components, etc.

In addition, the context need to be defined too. There exist different definitions of context (see Section 2.3). In spite of that, the common aspect of context is that the context of a mobile unit is determined by its current location which, in turn, defines the environment where the computation associated with the unit is performed. The context may include resources, services, as well as other components of the system. Therefore, it is important to clarify what aspects belong to the context, and their relationship with location and wireless network connection, etc.

Configuration The style should define architectural configuration, or topologies, which are the overall interconnection structure of design elements. This information is needed to determine whether design elements like components, wireless network connection and space are properly associated and connected. An architecture consists of a specific configuration of different components and their relationship to wireless network connection and space.

Configuration constraints Define configuration constraints that determine the permitted compositions of design elements. A constraint [92] is a property of or assertion about a style or one of its parts, the violation of which will render the system unacceptable. In order to ensure adherence to intended component uses, enforce usage boundaries, and establish dependencies among the design elements, constraints on them must be specified. Constraints may be defined in a separate constraint language, or they may be specified using the notation of the given modeling language and its underlying semantic model.

Dynamic change Dynamic change specifies how the architecture evolves at runtime and how to change its current configuration in the presence of mobility [25, 9]. The dynamic change is a main characteristic of mobile systems. The configurations and interactions change dynamically according to the movement of the components, the exchange of wireless connection, the change of context, etc. Modeling dynamic changes is also a very basic aspect to deal with mobility.

Component interaction Component interaction is a very important aspect of the middleware. The main goal of middleware is to facilitate component interaction of

components. Component interaction covers component communication, collaboration, and coordination. These different aspects are not orthogonal and there is no clear boundary. For example, components need a method to communicate with each other through the network. The method can be message based, transactional based, event based, etc. The communication requires the coordination and synchronization between different actions and components, and the components collaborate with each other in order to perform a task.

In a mobile setting, the components involved in an interaction change dynamically due to their migration or connectivity patterns. This results in dynamic reconfiguration among these components. The style should capture several aspects of component interaction: the roles played by the components during interaction, the methods used for communication, the coordination mechanisms, the influence on the coordination and communication because of the movement of components and change of connectivity, etc.

3.2.2 Requirements for the modeling language

The modeling language should support modeling the previously required items for specifying the style. Especially, it should model mobility explicitly, which includes two aspects: the architectural elements related to mobility, and how computation and component interaction are influenced by mobility. Besides, we also require some other important aspects in order to support practical development of consistent and correct styles. We require a refinement based approach with validation process and tool support.

Understandability A good model should be understandable, clear, unambiguous. One of the major roles of software architectures is that they facilitate understanding of systems at a high level of abstraction. To truly enable easy and unambiguous communication about a system among designers, developers and other stakeholders, modeling language must model the system with a simple, clear, and understandable syntax. Generally, visual (or graphical) presentation of the models could help the understanding very much.

Analysis The middleware is a large, complex software system. The ability to evaluate and analyze the properties of such systems upstream, at an architectural level, can ensure the quality of the developed system in the specification process, and hence substantially lessening the cost of errors [138, 81]. Particularly, it should allow us to reason about mobility related aspects, such as location and wireless network connection, dynamic changes and dynamic component interaction, etc., in the same manner as we reason about other components in a distributed system. In addition, automation and tool support of the analysis could help a lot since the specification is often complicatedly constructed and thus difficult to be checked manually.

Formal semantics Semantics have a central role to play in defining the semantically rich language capabilities such as execution, analysis, transformation and tool support. An informal semantics [136, 92, 5] brings with it some significant problems

- Because users have to assign an informal or intuitive meaning to models, there is significant risk of misinterpretation and therefore misuse by the users of the modeling language.
- An informal semantics cannot be interpreted or understood by tools. Tool builders are thus required to implement their own interpretation of the semantics. Unfortunately, this means that the same language is likely to be implemented in different ways. Thus, two different tools may offer contradictory implementations of the same semantics, e.g. the same StateMachine may execute differently depending on the tool being used.

Adaptability and extensibility The style should capture the main architectural aspects of a class of middleware. It is not realistic to have one common style that covers all middleware platforms and fits to every middleware. That means, we need to adapt the model to support other styles that are induced from other classes of middleware with different design techniques and functionalities support. Hence, the modeling language should be adaptable and extensible to add new concepts and delete unneeded elements.

Style refinement Support a stepwise style refinement, which is generally required by a complex system in order to decrease the complexity. The designers can start from a simpler, easier design, come to a more and more concrete, complete design through refinements. In the refinement process, it is difficult to ensure that the concrete specification is a correct refinement of the more abstract one. Especially, the correctness of the newly added architectural elements is difficult to be checked since it is not easy to directly map them to the abstract ones. Therefore, refinement correctness criteria should be provided to guarantee the correctness of a refined concrete specification. Besides, it is very helpful to have tool support and automation of the refinement process, since it is normally a complex task.

In addition, one main requirement for refinement is to preserve the desired properties. Some researchers argued that different domains have different requirements for the properties. Therefore, which properties should be preserved vary in different architecture refinement processes. It is then important to further identify the properties to be preserved when refining the style concretely.

Consistency check One important issue of a model-based approach is to correctly bridge the gap between abstract models and concrete models, or even implementations. However, to validate whether a concrete model or an implementation is a correct refinement of an abstract model is usually difficult and complicated.

The approach should provide a method to support the validation process between different abstraction layers. This is also important to prevent architectural decay, which is usually gradual divergence between an abstract model and a concrete model, or an implementation. Again, tool support and automation of the validation process are important.

Usability The approach should provide the ability to specify the style in a direct and expressive way, in order that the designers can specify the models easily, and use the approach conveniently. To be truly usable and useful, it should provide tool support for architectural design, refinement, analysis, and executable system generation.

3.3 Survey of related work

After giving the detailed requirements for the architecture centric approach, we will now evaluate related approaches against the requirements. Following the similar classification as the requirements, we separate the survey into two parts: architectural styles and modeling languages.

3.3.1 Survey of architectural styles

There are many architectural styles currently in use [137, 5, 3, 36], such as "client-server system", "layered system", "blackboard organization", "pipes and filters", and styles for GUI software [144]. These styles address different architectural aspects, and they are developed to solve specific design problems. When designing a software system, selection of appropriate architectural style(s) becomes an important determinant of the system's success [36, 56]. The style should meet a set of architectural requirements. Since there are no styles developed specifically for the middleware for mobile systems, or for mobile systems, we will focus on those styles that are closest to our style requirements, which are styles used for the middleware for distributed systems, styles induced by middleware, and styles for self-healing systems. We will discuss the problems of the styles when used for the middleware for mobile systems, and what do we need.

3.3.1.1 Styles used for middleware

Both software architectures and middleware technologies focus on helping the development of large-scale, component-based system, but they address different stages of the development lifecycle. Software architecture is an abstract model of a system that highlights the system's critical conceptual properties, while middleware enables the system's realization and ensures the proper composition and interaction of the implemented components. Some researchers are trying to coupling architecture modeling and analysis approaches with middleware technologies [94, 90, 44]. Some researchers [90, 44] suggest to use architecture styles at

the abstraction level as a reference for middleware development, therefore a middleware is a realization of an architectural style. Middleware can be viewed as a connector between components that use the middleware.

These styles are component-connector style that focuses on the system's runtime behavior. In such a style, a component refers to a runtime entity, and it is the system's units of execution. The component can be processes, objects, clients, servers, and data store. Component interaction is embodied in the notion of connectors [94]. A connector is an abstract mechanism that mediates interactions among components. A connector can be communication links and protocols, information flows, and access to shared storage. Examples of connectors include shared representations, remote procedure calls, message-passing protocols, event-based and data streams. The researchers [94] give a quite comprehensive classification framework and taxonomy of the connectors. They classify four kinds of interaction services a connector provides. They are: communication services that support transmission of data among components, coordination services that support transfer of control among components, conversion services that convert the interaction required by one component to that provided by another, and facilitation services mediate and streamline component interaction. In their opinion, every connector provides services that belong to at least one of these categories. It is also possible that one connector provides multi services.

Their definition of component interaction is quite close to our definition for the middleware: "Component interaction covers component communication, collaboration, and coordination" (see Section 3.2). We do not mention conversion services and facilitation services separately since they can be covered by collaboration and coordination in our definition. Actually, these different aspects are not orthogonal and there is no clear boundary.

In order to clarify in detail how to use the component-connector style at an abstraction level as a reference for middleware development, we will explain in the following three typical component-connector styles: client-server style, publish-subscribe style and shared-data style, each of which can be matched respectively to the mentioned three kinds of middleware for distributed systems in Section 2.2: object/component middleware, event-based (or publish/subscribe) middleware, and tuple space-based middleware. In particular, we will focus on component interaction and dynamic changes, which are required by our style specification.

Client-server style The client-server style is often used in distributed systems. The component plays two roles: clients that request services of other components, and servers that provide services to other components. The connector provides request-reply invocation of services. Components may request services of other components through the connector. Components may also provide a set of services through the connector. The connector associates clients with servers through a so-called attachment relation, which associates clients with the request role of the connector, servers with the reply role of the connector, and determines which

services can be requested by which clients. The connector manages the flow of control among components through various invocation techniques (coordination). It also performs transfer of data among the interacting components through the use of parameters and messages (communication).

The pure client-server style provides asymmetric invocation of services: clients initiate actions by requesting services of servers. Thus, the client must know the identity of a service to invoke it, and clients initiate all interactions. A server listens for requests upon those services. One basic form of service invocation is synchronous: the client sends a request to the server and waits, or is blocked, until a requested service completes its actions, possibly providing a return result.

The dynamic changes in this style generally include the addition, removal, or replacement of clients and servers, hence the topology of the architecture is changed. The changes are usually caused by the computation behavior of the components. For instance, some researchers [9] give a typical example of the dynamic change that happens in a fault-tolerant client-server system. The primary server's failures trigger the dynamic change, and the client is attached to the secondary server after the change happens.

The client-server style focuses mainly on following issues: correct coordination of the service invocations, performance issue that determines whether a system's servers can keep up with the volume and rates of anticipated service requests, dependability issue to understand whether a system can recover from a service failure, and security issue that determines whether information provided by servers is limited to clients with the appropriate privileges.

The client-server captures some architectural aspects of the middleware, and it can be used for guiding the design of the middleware in the captured aspects. It provides a guide of how to correctly coordinate service invocations, how to transfer parameters and messages among components, how to activate the backup server when the primary one fails, and how to calculate the server's performance, etc. Therefore, the client-server style can be used for the reviewed object/component middleware (see Section 2.2) when considering these aspects, whose basic component interaction pattern is invocations between object clients and servers.

Publish-subscribe style In this style, the component is event subscriber that subscribes to events of other components, and event publisher that publishes events to other components. The main form of connector in this style is a kind of event bus, which administrates and issues events. Therefore, components interact via announced events. Components may subscribe to a set of events through the connector. Components may also place events to the connector by announcing them, the connector then delivers those events to the appropriate subscribers. The connector manages the flow of control among components (coordination). Upon the occurrence of an event, the connector generates messages for all interested subscribers and yields control to the components for processing these messages (communication).

The style is commonly used to decouple message producers and consumers in a message queuing model that allows only a directed communication, where the producers explicitly define the recipients. The style supports undirected communication where the producer does not need to explicitly define the receiver. Hence, the correctness of the producer does not depend on the receiver's. The decoupling of producers and consumers is the prime advantage of this style. It allows deferring the binding of procedures and consumers of messages until runtime, and therefore supports the modification of these procedures and consumers without affecting other parts of the system.

The dynamic changes in the style generally include the addition, removal, or replacement of subscribers and publishers. The decoupling of producers and consumers allows flexible dynamic topology configurations. The producers and consumers do not need to be active at the same like the client-server style. The changes are usually caused by the computation behavior of the components. For instance, the publisher disconnects to the connector after publishing events, and it reconnects again when it has new events to publish.

The publish-subscribe style focuses mainly on the event dispatch mechanisms, which include: handling the published events, delivering the events, setting priority of the events, synchronization between actions, etc.

Similar to the client-server style, the publish-subscribe style provides a guide of how to correctly dispatch events, and how to coordinate the event publishers and subscribers, etc. It can be used for the reviewed event-based (or publish/subscribe) middleware (see Section 2.2) when considering these aspects, which uses the publish / subscribe model as the basic component interaction pattern.

Shared-data style This style is organized around one or more shared-data stores, which store data that other components may read and write. The component is shared data store that stores the data, and data accessor that accesses the data. The connector is responsible for data access which often requires preparation of the data store before and clean-up after access has been completed. Components may write and read data through the connector (communication). The connector may perform translation of the information being accessed (conversion), when there is a difference in the format of the required data and in the format of provided data.

The data can be stored either persistently or temporarily, in which case the data access mechanisms will vary. Examples of persistent data access include query mechanisms, such as SQL for database access, and accessing information in repositories, such as the CORBA interface repository. Examples of transient data access includes heap and stack memory access, and information caching.

The topology configurations of this style composes of the data accessors attached to connectors, and the attached data stores. The dynamic changes in this style generally include the addition, removal, or replacement of data accessors and data stores. Again, the changes are usually caused by the computation behavior of the components. For instance, the primary data store's failures may trigger the

dynamic change, and the data accessor is attached to the secondary data store after the change happens.

The style focuses mainly on performance, availability, data integrity and security. The reviewed tuple space-based middleware also uses a shared data space. However, the shared-data style can not be used to guide the design of the middleware since the notion of the shared data base is completely different. The tuple space-based middleware uses a shared distributed memory spread across all participating hosts that processes can concurrently access. In this case, the component should be process, while the connector should be the shared data space that associates components through data access. Differently, the shared-data style defines the shared data store as the component, and the connector is responsible for data access.

3.3.1.2 Middleware-induced styles

Some researchers [106] introduce middleware-induced styles that capture the architectural assumptions induced by the middleware. Instead of the general top-down approach adopted by the software architecture community, they take a bottom-up approach that originates from the results that practitioners have achieved in the definition of middleware. They think that a class of related forms of middleware induces the definition of an architectural style, with each specific middleware of the class defining a variation of that style. These styles describe the assumptions and constraints that middleware infrastructures impose on the architecture of systems. They believe that the explicit availability of middleware induced styles is extremely useful in guiding the architect in the definition of the architecture of an application and in selecting the most suitable middleware, independently of any special purpose development environment.

They take two representative middleware of the event-based paradigm: JEDI and C2 as case studies. However, they do not specify the styles really. They only focus on providing an evaluation of ADLs (Architecture Definition Languages) as to their suitability for defining middleware-induced styles. They conclude that the top-down approach adopted by the software architecture community in the development of languages and tools seems in many ways to ignore the results that practitioners have achieved (in a bottom up way) in the definition of middleware. Middleware has demonstrated their usefulness and effectiveness in a number of practical cases. The software architecture community has now the potential to formalize these achievements in expressive and usable ADLs and, more generally, to coordinate the definition of support technology for the development of middleware-based applications.

Other researchers [145] take a close approach to [106] and specify a style that is induced by component-based architectures, which can be refined into a service-oriented style induced by service-oriented architectures. Their styles are quite close to the explained component-connector style in last section, since they also focus on the system runtime behavior specification, especially dynamic changes.

However, they do not provide a complete component interaction specification compared to the component-connector style. They specify only component communications, but not component coordinations. That means, they only specify how to transfer parameters and messages among components, but not the coordinations among components. The component-based style can be used for the reviewed object/component middleware, when we design the part of transferring parameters and messages among components.

3.3.1.3 Styles for self-healing systems

A mobile system is a kind of self-healing system, which is an emerging class of software systems that exhibit the ability to adapt themselves at run-time to handle situations such as environment change, mobility and system faults. Oreizy et. al [113] describe the life cycle of self-healing systems as consisting of four major activities: (1) monitoring the system at run-time, (2) planning the changes, (3) deploying the change descriptions, and (4) enacting the changes.

Some researchers [98] give the architectural style requirements of self-healing systems, which are adaptability, dynamicity, awareness, autonomy, robustness, distributability, mobility, and traceability. They also design a specific architectural style, called PitM, for supporting self-healing systems. Their work is quite helpful for understanding the general characteristics of the style for mobile systems. However, the style is too general to capture a class of mobile systems. Similarly, [69] also presents architectural styles for adaptable self-healing dependable systems, which are again too general to capture a class of mobile systems.

Other researchers [34] provide a technique that explicitly leverages architectural styles to enable system evolution in support of self-healing. They generalize architecture-based adaptation by making the choice of architectural style as an explicit design parameter in the framework. The created style is then a run time artefact, but not a design time artefact. The style does not meet our requirements since we need a style that captures design strategies and system characteristics. Their approach is only helpful if we already have the required style.

3.3.1.4 Evaluation and open problems

We have explained the currently available styles that are closest to our style requirements, which are the general component-connector styles used for the middleware, styles induced by middleware, and styles for self-healing systems, etc. As we can see, nearly all the explained styles capture some architectural aspects of the middleware, and they can be used for guiding the design of the middleware in the captured aspects. Especially, the middleware-induced styles provide more reasonable and suitable architecture abstractions specialized for a specific class of middlewares than the general component-connector style.

Nevertheless, these styles provide only very limited design guides for the middleware for mobile systems, since they do not consider the following required im-

portant aspects:

Space and wireless connection As mentioned in Section 3.2.1, space and wireless connection are the two most important aspects to deal with mobility, which influence greatly component and connector behavior. It is important for a style to make the space and network connection explicit into the component structure.

However, these styles do not take into account the properties of the "physical distribution topology of locations and communications. It is assumed that the physical links that enable communication between hosts in the underlying communication network are fixed and statically determined. Thus, these models rely on the assumption that the computation performed by individual components is irrelative to location of the component, and the coordination mechanisms put in place through connectors can be made effective across the wires that link the components hosts. However, all these assumptions are not valid anymore for mobile systems where the physical locations, communications, and context influence the component and connector behavior.

Mobility and dynamic change Another basic requirement for the style is to modeling dynamic changes in the presence of mobility. In a mobile system, the configurations change dynamically according to the movement of the components, the change of wireless connection, the change of context, etc.

The reviewed styles also model dynamic changes. However, they consider dynamic change mainly for safety or system integrity reasons. Dynamic changes are therefore caused generally by the computation behavior of the component. This is not enough for a middleware for mobile systems. They should consider other basic aspects, like the movement of the components, the connectivity changes, and the change of context, in order to provide a design guide of how to deal with aspects.

Dynamic component interaction The reviewed styles support component interaction specification, which is embodied in the notion of connectors. A connector is an abstract mechanism that mediates interactions among components. As mentioned, their definition of component interaction is quite close to our's. Specifically, they specify component communication, coordination, and collaboration, which can be used to guide the design of these aspects to some degree.

However, the specified component interaction model is rather static. They do not consider the factors of location change and connectivity change, which cause often dynamic changes of coordinated components for a component interaction. Therefore, the interaction changes dynamically according to the movement of the components, the change of wireless connection, the change of context, etc. All these make the connector unsuitable for capturing the required dynamic component interactions.

Actually, we can say that these styles are designed for distributed systems, which capture the main characteristics of distributed system designs. They can be used for helping the design and development of the middleware for distributed systems. In such styles, mobility related part is naturally not considered or abstracted away on purpose in order to decrease the complexity and concentrate on the important aspects of the distributed system. However, mobility has created additional complexity for computation and coordination, this makes these styles hard to use for helping the design and development of the middleware for mobile systems. We need architectural styles that provide design guidance during the creation of an architecture for a specific class of middleware for mobile systems. The style should capture the main design characteristics of a class of middleware for mobile systems, which include space and wireless connection, new dynamic component interaction models and dynamic changes in the presence of mobility, etc.

3.3.2 Survey of modeling languages

To support the definition of software architecture and architectural styles, software architecture modeling languages [92, 36] and their accompanying toolsets have been proposed to provide notations for specifying and analyzing the architecture. There exist many modeling languages that support different modeling notions and techniques [91]. Many researchers have classified and compared the detailed characteristics of the modeling languages [93, 155, 37, 151, 139]. As mentioned, the modeling language decides the specification capability and possible available methods and tools. In our survey, we will only focus on two very basic and important aspects: the specification of architectural styles and the specification of mobility, to see, what are the main problems of the current modeling languages to support the new architectural style specification.

3.3.2.1 Modeling languages for style specification

There are many modeling languages that support the specification of architectural styles, examples are different ADLs that include Rapide [86], MetaH [152], Darwin [87], Wright [8], and C2 [144]. Recently, some researchers also explore graph transformation systems to specify architectural styles [145, 14, 75]. The specified styles are mainly component-connector based styles that focus on the system's runtime behavior. In such a style, a component refers to a runtime entity, and it is the system's units of execution. The component can be processes, objects, clients, servers, and data stores. A connector is an abstract mechanism that mediates interactions among components. Generally, the style includes a coherent collection of component types, connector types, configurations and configuration constraints, similar to our basic requirements for the style. In addition, each of the modeling languages embodies a particular approach to the specification and evolution of an architecture and architectural style, with specialized modeling and analysis techniques that address specific system aspects.

However, these modeling languages are not designed for mobile systems. They do not take into account the properties of the "physical" distribution topology of locations and communications. It is assumed that the physical links that enable communication between hosts in the underlying communication network are fixed and statically determined. Thus, these models rely on the assumption that the computation performed by individual components is irrelative to location of the component, and the coordination mechanisms put in place through connectors can be made effective across the wires that link the components hosts.

Besides, the researchers generally hold the idea that it is important to separate the specification of dynamic reconfiguration behavior from its non reconfiguration functionality. Therefore, the component, connector and dynamic changes are specified separately. This separation works well when modeling a traditional distributed system, which has a stability assumption with the network connection and location of the components. And the component behavior is independent of the below communication mechanisms. Modeling dynamic changes thus does not involve the connector behavior. However, this causes problem when modeling a mobile system where dynamic changes of components often involve the below communication mechanisms. The change of connectivity can cause dynamic change between components. In addition, the location change of the components causes dynamic changes. Context change can causes dynamic changes too. Therefore, it is important to involve these factors when modeling dynamic changes in mobile systems. Besides, dynamic changes needs to involve the connector behavior.

In addition, they specify the style that consists of normally only two kinds of separated elements: components and connectors. Correspondingly, the modeling and analysis techniques with toolsets support only the definition of components and connectors, each of which has its own behavioral definition and semantic domain. It is rather difficult to use them for specifying the required new style: it is difficult to add mobility related elements like space and connection into the style as first class entities, it is even more difficult to deal with these elements throughout the software development life cycle, starting from the definition of mobility, through designing and reasoning about mobility, and down to the tools provided to designers.

All these make the modeling languages hard to use for specifying the new architectural style with mobility support. They need to be adapted to support the new style specification.

3.3.2.2 Modeling languages for mobility specification

As the importance of mobile systems increases, many researchers provide mobility support in specification languages and modeling languages. A number of different process calculi have been proposed to describe the interaction and movement of mobile processes, mostly by extending the π-calculus [99, 73, 31]. In order to express runtime properties of mobile systems, some of these calculi have been complemented by logics [26, 30, 97]. More than a calculus, Mobile Unity [119] provides a programming language and logic for describing mobile systems. These

process calculi with their associated logics have a complementary focus: They provide means for programming mobility in terms of the processes driving individual components or devices, rather than for its high-level conceptual modeling. Although the actual protocol implementing the operation is more easily specified (and verified) in a process calculus, it is difficult to specify architectural styles.

AGILE project [10] extends existing specification languages and methods to support mobility. UML class, sequence, and activity diagrams are extended to describe how mobile objects can migrate from one host to another, and how they can be hosts to other mobile objects [17, 84]. Graph transformation systems are proposed as a means to give an operational semantics to these extensions. Other extensions are based on architectural description languages, like the parallel program design language CommUnity [85, 10] using graph transformation to describe the dynamic reconfiguration; Klaim [105] as a programming language with coordination mechanisms for mobile components, services and resources; the specification language CASL [12] as a means for providing architectural specification and verification mechanisms. While Klaim and CASL are, again, more programming and verification-oriented, the approaches based on UML and CommUnity are at a level of abstraction similar to ours. However, the goals are different: Our focus is to build an architectural style that captures the main characteristics of the middleware for mobile systems, while the focus in the cited approaches is on the modeling of applications within a style more or less determined by the formalisms.

Chapter 4

An Overview of the Approach

4.1 Overview

As concluded in Section 3.3, mobility has created additional complexity for computation and coordination, which makes the current architectural styles hard to use for helping the design and development of the middleware for mobile systems. We need new architectural styles that provide design guidance during the creation of an architecture for a specific class of middleware for mobile systems. Correspondingly, the modeling language needs to be adjusted in order to support the specification of the new style.

Now, we will develop an architectural style-based approach that supports the design and development of the middleware[1]. We will develop a new architectural style for the middleware that should capture the already established design knowledge or successful experience in this area. The style is middleware-induced. That means, instead of the general top-down approach adopted by the software architecture community, we will take a bottom-up approach that originates from the results that practitioners have achieved in the middleware area.

The same as in [106], we think a class of related forms of the middleware induces the definition of the architectural style, with each specific middleware of the class defining a variation of that style. More specifically, we will build the style based on the concluded architectural commonalities of the reviewed middleware (see Section 2.4). In order to decrease complexity and enhance reusability, we will specify the style on different abstract layers, namely, conceptual style (on a more abstract layer) and concrete style (on a more design-specific layer). The concluded common aspects will be specified in different styles. We abstract the more general aspects: space definition and main functionality in the conceptual style, which also includes dynamic changes. The conceptual style specifies a style of mobile systems, or a mobility style, which includes the movement style of the components, and the specific service provided by this kind of mobility, etc. The concrete style models the more design-specific component interaction pattern adapted for

[1]Without further explanation, "the middleware" stands for "the middleware for mobile systems" in the rest of the thesis.

the mobility style. The dynamic change and functionalities specified in the conceptual style will be refined into the concrete style.

This separation can simplify the specification of the style, and help the designers and developers fully understand how mobility influences the component interaction, and how the models evolve. This differentiates our style from the classical architectural styles that do not have such a separation. Besides, in contrast to the classical architectural style (see Section 3.3), we combine the specification of dynamic change and component interaction, since dynamic change is an important characteristic of mobile systems, and it also influences component interaction very much.

We develop the approach based on GTS (Graph Transformation System) and UML-like metamodeling techniques. To the existing contributions, we will add the new role of GTS as modeling mobility and the architectural style for the middleware. We will then investigate appropriate refinement based on the style specification. We formalize the refinement relationship between two abstract layers based on the mapping of rules. In addition, we develop a framework that supports behavioral consistency check between models on different abstract layers. It checks whether the specified architectures in a same hierarchy belong to the same family of middleware. The operational semantics of the GTS allows us to execute the models and thus analyzing the system through simulation. Based on the existing tool Fujaba, we develop three ways to use simulation for further style-based analysis and automation: efficiently validating the style model, style refinement consistency check and behavioral consistency check. Besides, we develop a practical and useful style-based engineering process that helps efficiently developing correct styles.

In order to describe the whole approach in a clear and structured way, we will generalize the approach as a modeling and simulation framework, which is divided into several main parts: GTS based modeling, architectural style, refinement and simulation. The following chapters will follow this structure and explain every part respectively. In this chapter, we will at first explain the concept of the new architectural style. Afterwards, we will conclude the main ideas of the framework.

4.2 The architectural style for the middleware

This section introduces the concept of the architectural style for the middleware. It includes two parts: what common architectural aspects of the middleware can be captured by the style, and how can we structurize the architectural style into different abstract layers, in order to decrease complexity and to enhance reusability of the model.

4.2.1 Middleware-induced style

We will create *the architectural style for the middleware* that provides design guidance during the creation of an architecture for a specific type of the middleware. The style is middleware-induced. That means, we will take a bottom-up approach that originates from the results that practitioners have achieved in the area of the middleware for mobile systems. This is different from the general top-down approach adopted by the software architecture community. We think that a class of related forms of middleware induces the definition of the architectural style, with each specific middleware of the class defining a variation of that style.

The goal of an architectural style is to capture solutions that were already in use, but not to make up solutions. There is a rule for how to build a style: before being eligible for inclusion, the style has to occur in at least two systems, or in two different organizations, and so on [36]. However, the great diversity of the middleware makes it difficult to build a style in this area. There exist so many different middleware prototypes, which have different design concepts and strategies, which support distinct functionalities and provide various services to applications (see Section 2.3). It is difficult to explore commonalities across the middleware families for mobile systems.

Thanks to the generalized framework for describing the middleware (see Section 2.3), we can compare different approaches from a same point of view, and observe similarities and common aspects from the great diversities of the middleware. Recalling the conclusion in Section 2.4, we can see that *space* and *wireless connection* are the two most important aspects when designing a middleware. We have also observed some commonalities of the reviewed middleware regarding *space definition*, *main functionality* and *component interaction / design pattern* if we consider the supported *wireless network* of the middleware. Accordingly, we can roughly separate the middleware into two classes: the middleware for nomadic networks, and the middleware for ad hoc networks.

Correspondingly, we can develop the architectural style that captures these common aspects, and build *the architectural style for the middleware for nomadic networks* and *the architectural style for the middleware for ad hoc networks*. The style defines a family of middleware that shares some common design strategies and characteristics.

4.2.2 Layered structure of the style

In Section 2.4, we have also concluded that space definition and main functionality are more general than component interaction. That means, several different middleware platforms use different component interaction design patterns but support the same space definition and functionality, or a special style of mobility, i.e., the allowed movement for the components, the specific service provided by this kind of mobility, etc. Therefore, we can separate the more general space definition and functionality from the design specific component interaction, and build a model

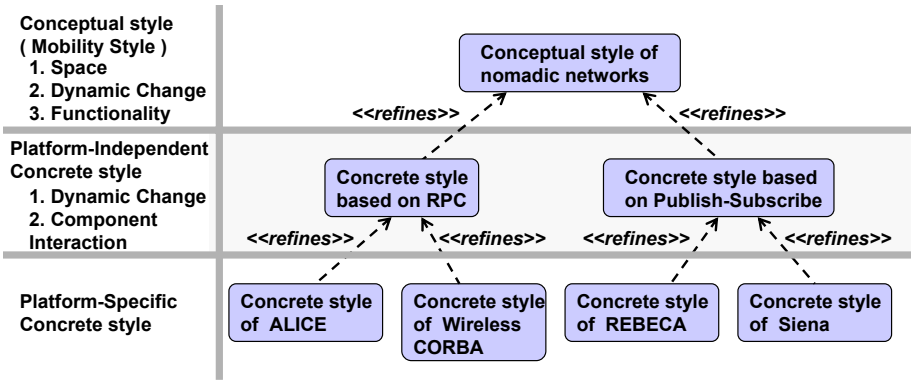

Figure 4.1: The layered structure of the style for middleware for nomadic networks

that captures only space definition and functionality.

This separation is reasonable since space definition and functionality are decided by the targeted application and wireless network, which present the main requirements for the mobile systems. Thus it can be put into a more abstract layer. The abstract style also represents what the middleware should provide in terms of services to the application. On a more concrete layer, the middleware adopts specific component interaction design pattern to deal with the mobility model and realize the functionality. We can further separate the component interaction layer into two abstraction layers: the platform-independent layer is abstracted from design specific details, while the platform-specific layer is a more complete model of the middleware that includes the design specific details.

In conclusion, we separate the style into three abstract layers, i.e., the *conceptual style* on a conceptual layer and the *concrete styles* on concrete layers in Fig. 4.1. The conceptual style also models a style of mobile system, or a mobility style, which includes the space definition of the mobility style and main functionality. It models the movement style for the components, the specific service required by this kind of mobility, etc. On the concrete layer, the conceptual style is refined into a concrete style, where more and more design-specific aspect, i.e., component interaction pattern is integrated into the functionality. The concrete style also specifies dynamic changes of components involved in an interaction due to their migration or connectivity patterns. As a result, the dynamic change specified at the conceptual level will be associated with component interaction at the concrete level. The concrete style can be further refined into a platform-specific style to add more design specific aspects.

For instance, we can build a conceptual style for the middleware for nomadic networks (Fig. 4.1), which is induced from the middleware for nomadic networks. The conceptual style can be refined into platform-independent concrete styles that support different component interaction models, such as RPC and publish-subscribe model. The concrete style can be further refined into platform-specific concrete styles, like concrete style of Wireless CORBA (RPC-based), or concrete style of

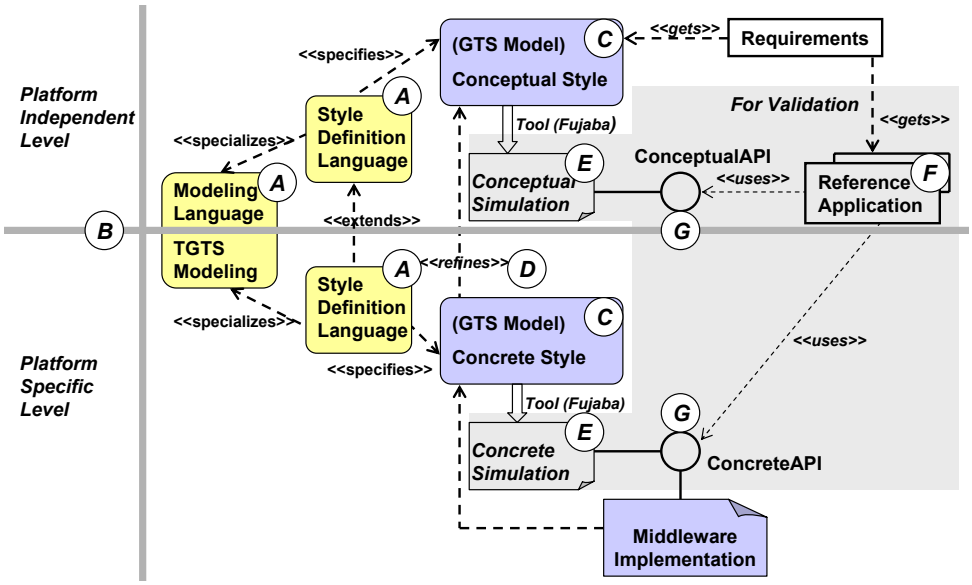

Figure 4.2: GTS - based modeling and simulation framework

REBECA (Publish-Subscribe based), etc.

The definition of the style as platform-independent or platform-specific is relative. In our example, we separate the concrete styles further into two abstraction levels: platform-independent and platform-specific one, where the platform-specific style contains more design specific aspects. However, the platform-independent concrete style is also platform dependent when compared to the conceptual style, since it specifies design-specific component interaction patterns.

We will illustrate the three-layered style through specifying *the architectural style for the middleware for nomadic networks*, which will be explained in Chapter 6.

4.3 The modeling and simulation framework

We develop the architectural style-based approach based on the GTS (Graph Transformation System) and UML-like metamodeling. It supports the complete process of style development, which includes style specification, refinement, simulation, validation and consistency check, etc. We generalize the approach as a modeling and simulation framework (Fig. 4.2), which is divided into four main parts: modeling language, architectural style, refinement and simulation.

4.3.1 Style-based modeling

We model the architectural style (A) using the GTS (Graph Transformation System) and UML-like metamodeling techniques, namely, TGTS (Typed Graph Trans-

formation System) [122]. A GTS (Graph Transformation System) [122] is a system using the techniques of graph transformation. GTSs have been used for modeling distributed systems, mobile systems and other complex software systems. Especially, a TGTS (Typed Graph Transformation System) is used often for modeling functional requirements, software architectures, and architectural styles (for distributed systems).

Visual representation and formal semantics are the main advantages of the modeling language. It has not only a graphic, easy understandable syntax, but also an operational semantics. It supports rich modeling techniques, such as separation of concerns, refinement, adaptability, extensibility, etc. It can describe complex structures and systems, and model concepts and ideas in a direct and intuitive way. The developed model is easy to understand and handle with the well-known, user-friendly UML diagrams. At the same time, the formal semantics allows execution, analysis, transformation, automation and tool support. The designers and developers do not need deep mathematical knowledge in order to use the approach to develop new architectural styles.

We can separate the architectural style specification into structural view and behavioral view. Correspondingly, we use the TGTS [122] $G = \langle TG, C, R \rangle$ to specify the required aspects of the architectural style. TG *type graph* is here a meta model (or, UML class diagrams) that defines structural part of the style. C is a set of constraints restricting their possible compositions. R is a set of *graph transformation rules* (given by pairs of object diagrams) that defines the behavioral part of the style.

4.3.1.1 Type graph and instance graph

The TG *type graph* definition is based on the MOF (meta object facility) [108], which provides the standard OO modeling concepts including MOF packages, classes and associations. They are visualized using class diagrams. The object diagrams are called *instance graphs* that are typed over the type graph. We use the type graph to define the *vocabulary*, allowed *configurations*, *constraints*, and allowed *type definition* of the style. The instance graphs of the type graph represent both the declarative definition of an architectural style and the architectural configuration.

More specifically, the classes define the vocabulary of the architectural style. Associations define the possible links and relationship among these elements. The *type definition* of the architectural elements are defined as a "instanceOf" association. Simple constraints, already included in the class diagrams, are cardinalities that restrict the multiplicity of links between the elements. More complex restrictions can be defined by OCL (Object Constraint Language) constraints. Through this way, the required architectural design elements of the style such as space, wireless network connection and middleware construction components can be specified at type level and instance level. The relationship and constriction among them are also modeled in the same class diagram.

Additionally, we group different architectural concerns using the package concept, in order to decrease complexity of the model. The class diagram allows a uniform representation where elements of different submodels are represented as vertices of the same abstract syntax graph, i.e., an instance of the meta model. Based on this uniform representation, the different submodels can be related by associations between the common elements belonging to different submodels. For instance, we can separate different concerns of the system, i.e., software architecture, distribution, and roaming, while at the same time retaining an integrated representation.

4.3.1.2 Graph transformation rules

The behavioral part of the style is specified using the graph transformation rules. A graph transformation rule $r : L \Rightarrow R$ consists of a pair of TG-typed instance graphs L, R such that the intersection $L \cap R$ is well-defined. The left-hand side L represents the pre-conditions of the rule while the right-hand side R describes the post-conditions.

The rule defines a certain modification of instance graphs. We can differentiate two kinds of rules according to different usages: rules for dynamic changes and rules for component interaction. Rules for dynamic changes specify the reconfiguration of the style in the presence of mobility. Therefore, the instance graphs represent architectural configurations. Rules for component interaction specify the change of states of the elements involved for communication and collaboration. The coordination and synchronization between different actions and components are specified using UML Interaction diagrams. Therefore, rules for interaction have a fixed execution sequence, unlike the rules for dynamic changes. The instance graphs in this case represent the interaction states. Correspondingly, the rules for interaction are assigned to relative components, since the specification of component interaction is design specific, and the tasks of the components can be decided.

In addition, we use the rules to specify IDL semantics. Since IDL is often used to specify a concrete middleware, our semantics specification of the IDL belongs to the platform-specific style. Functional requirements towards middleware are often structured through interface definitions, e.g., using IDL for CORBA [110], Microsoft IDL (MIDL) for DCOM [133], or Java interfaces in the case of RMI [117]. The semantic part of an IDL is only expressed informally in natural language. This is a potential cause of conflicts and ambiguities, especially when components have complicated behavior or many components collaborate to perform a task in a mobile setting. As a result, the gap between a middleware's specification and its implementation is huge, and it is difficult for developers to derive implementations directly from specifications. Several implementations of the same specification may thus show different behaviors. Our semantics specification of an IDL includes two parts: the structural semantics is specified through class diagram, and the behavioral semantics is specified through rules. Our specification focuses on

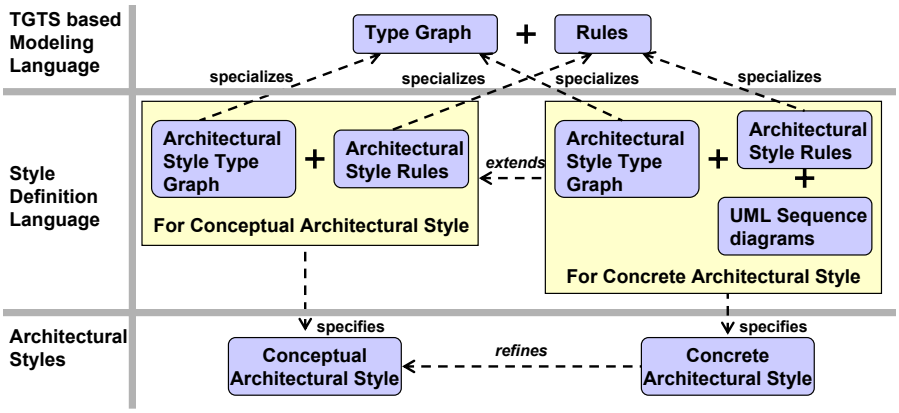

Figure 4.3: GTS - based style modeling

the change of states of the elements involved for communication and collaboration. We can specify how the instances of objects are affected by the execution of an operation, thus the behavior of the involved internal elements for an operation can be defined formally.

For instance, we define graph transformation rules (see Section 6.3) specifying dynamic changes in the presentation of mobility, based on the uniform representation of space, wireless network connection and architectures as meta model instances. The movement style of the components in the specified space, the disconnection of the wireless connection and other actions related to mobility can be now modeled through the graph transformation rules. In the platform-independent concrete style, we define a set of rules with fixed order for the RPC component interaction model (see Section 6.4). The dynamic change rules will influence the component interaction rules since the components involved in an interaction change dynamically due to their migration or connectivity patterns. In the platform-specific concrete style, we define rules for the Wireless CORBA IDL that specifies a concrete middleware.

Our specification of component interaction and dynamic change based on the graph transformation rule can model the influence of mobility to component interaction naturally. The dynamic change and component interaction are integrated smoothly and directly in one specification with the same semantic background. The simplicity is very important to help the designers and developers to understand the model and use the approach easily.

4.3.1.3 Layered modeling structure

Corresponding to the architectural style separated on different abstract layers, our approach is driven by a stepwise refinement between TGTS-based models on different abstract layers (B). As shown in Fig. 4.2, a conceptual style model is smaller and easier to understand, while a concrete style model reflects more design specific

concerns.

The modeling part in Fig. 4.2 can be further extended to Fig. 4.3. More clearly, we use TGTS to specify the style definition language (at the type level) that defines the architectural style (at the instance level). The style definition language includes type graph (TG) and rules (R) that are specialized for the style. That means, the style definition language constraints the allowed architectural elements and behavioral definition for the style through the type relationship. In order to specify a new style, different structural and behavioral elements are needed. In our case, a conceptual style can be refined into a concrete style. Correspondingly, the style definition language specialized for the conceptual style will be extended into the style definition language specialized for the concrete style. The conceptual style specified at the instance level is hence refined into the concrete style.

Our modeling language based on TGTS provides a flexible way to extend or change the models. This allows adaption of a specified model to meet new requirements of a new style. The adaption includes two parts: the type graph and rules. Class diagrams (Type Graph) provide a powerful and flexible way to model architectural elements and their relationships. New architectural elements can be easily added through adding proper associations and classes. While unnecessary elements can be deleted together with the related associations. The relevant rules built on instance graphs can be adapted easily to be typed over the adapted type graph. Rules can be added or deleted without violating the semantics of the TGTS. Our approach is quite similar to the metamodeling based language driven development approach [35] that adapts the language (but not the model) to new application domains and new requirements.

In addition, the number of the abstraction layers is not fixed as only two or three layers. More or less layers can be defined according to the complexity and size of the system and requirements of the models. Although we define three-layer architectural styles, more layers' concrete styles can be defined till a complete specification is reached, or the implementation of the software is accomplished. Such a flexible definition of the models on different abstract layers allows designers and developers to develop a complex, huge system step by step. It can decrease the complexity of the specified models and ease the design and development process.

4.3.2 The style for the middleware

The style for the middleware (C) should capture the common design strategies of a class of middleware. We explain the process of constructing the style through a concrete example: the architectural style of the middleware for nomadic networks (in Chapter 6). This type of middleware is quite mature from technique's point of view, and many existing middleware can be categorized into it. In spite of that, there is no common agreement or understanding of the middleware, not to mention available design standards that help the design and analysis of the middleware.

We develop the style that originates from the results that practitioners have achieved in the area. We construct the style on three different abstract layers, i.e.,

the conceptual style, the platform-independent concrete style and the platform-specific concrete (Wireless CORBA) style, in order to decrease complexity and enhance reusability. Each of the styles captures different architectural aspects or views, and they can be used to help the design and analysis of the middleware at different design stages.

The conceptual style specifies a style of mobile systems for nomadic networks, or a mobility style supported by the nomadic network, which includes the movement of the components, and the specific service provided by this kind of mobile system. It is about the requirements analysis for the middleware and it can be used at the very early stage of designing the middleware. The platform independent concrete style specifies the design-specific component interaction pattern adapted for the mobility style. We choose the adapted RPC as the example. The style can be used when we design a specific component interaction pattern for the middleware at a more abstract level. The platform-specific concrete style is a more refined version of the platform-independent concrete style. It can be used when we define a complete design specification of the middleware, e.g., the Wireless CORBA specification in our case, which is originally specified using the IDL.

4.3.3 Refinement

When developing the style, we use a stepwise refinement-based approach (D) in order to decrease complexity and enhance reusability. In the refinement process, it is difficult to ensure that the concrete specification is a correct refinement of the more abstract one. Especially, the correctness of the newly added architectural elements is difficult to be checked since it is not easy to directly map them to the abstract ones. Besides, one main requirement for refinement is to preserve the desired properties. Some researchers argued that different domains have different requirements for the properties. Therefore, which properties should be preserved vary in different architecture refinement process.

4.3.3.1 Properties to be preserved

In our context, we start from a simple abstract (i.e., conceptual) style that is refined to a more concrete style (platform-independent), which is further refined to an even more concrete style (platform-specific). The conceptual style defines functional requirements for a family of middleware for mobile systems through defining the required structural and behavioral elements. It can be refined to several different concrete styles that contain newly added structural and behavioral elements. The refined concrete styles should still satisfy the functional requirements and they belong to the same family of middleware, although when they use different design strategies to realize the functional requirements.

We require that the refinement is functional-preserved, which means that the functional requirements for the abstract style should be preserved in the refined concrete style and thus it satisfies the functional requirements. At the same time,

we also require that the refinement is functional-constrained, which means that the concrete style is only allowed to contain elements that are required or constrained by the functional requirements, since the concrete elements are constructed in order to realize a specific abstract task or functionality. It should not contain elements that exceed the scope of the functional requirements.

We use TGTSs to specify the style formally, which includes a structural part (UML class diagrams) and a behavioral part (graph transformation rules). Correspondingly, we require structural refinement criteria that ensure the correctness of the structural elements. We also require behavioral refinement criteria that ensure the correctness of the behaviors. Especially, we require that correctness of the rule sequence should be ensured during refinement. This is important for the rules for component interaction that have a fixed sequence, where the rules are only allowed to appear in the specified sequence, but not in other sequences.

4.3.3.2 Rule mapping-based refinement

We provide refinement criteria to ensure that the refined concrete style is a correct refinement of the abstract style. The criteria include both structural ones and behavioral ones. Especially, we develop an approach that can check the correctness of the sequence of rules through a scenario based construction. For a set of refined concrete rules with a fixed sequence $r'_1, r'_2, ..., r'_n$, we derive an initial graph L' that contains the needed precondition of all of the rules in the sequence. The rules are applied then one by one following the sequence. We denote it as $L' \overset{r'_1}{\Longrightarrow} M'_1 \overset{r'_2}{\Longrightarrow} M'_2 \ ... \ M'_{n-1} \overset{r'_n}{\Longrightarrow} R'$, where M'_i is the created intermediate graph, R' is the last graph that contains the result (i.e., postcondition) after applying the last rule. Afterwards, we derive a new concrete rule $r' : L' \Rightarrow R'$, which is a simplified version of the refined sequence of concrete rules. The rule r' transforms the overall precondition L' to the overall postcondition R'. The new rule preserves semantics of the sequence of concrete rules since it is based on a scenario construction, i.e., it checks whether the scenario (defined by the rule sequence) is reachable. Therefore, it can be used to substitute the complicated rule sequence. The similar construction of rules is minimal elements of a class of so-called derived rules, which are concerned with the construction of rules by which a given sequence or rewriting steps can be simulated in a single step [83, 40].

Before applying the approach, we need to construct the important L' and R' at first. In order to keep semantic consistency, we can not just put all the preconditions L'_i and postcondition R'_i of the rules together. The rule defines a certain modification of instance graphs. It happens often that some elements of the postcondition graph are created in the intermediate step, which are deleted again by another rule. Those elements should not be included in the result graph. Or some elements of the precondition are derived after applying another rule, which should not be included in the start graph neither.

We derive L' and R' through a step by step approach, which means, we combine firstly the first two rules, whose result will be used to combine the next

rule, and so on. The rules will be combined according to the specified sequence. We construct an algorithm to get the result graph of L', R', and the intermediate graph M_i'. The algorithm is based on the theory of minimality of derived rules in single pushout graph rewriting, whose theoretical proof is given by the researchers [83, 40]. By corresponding explicit constructions they have proved that there is a sequentially composed rule for each derivation sequence.

The refinement criteria build a bi-directional mapping between an abstract rule and the sequence of concrete rules via the derived simplified concrete rule. Since we formalize the refinement relationship between two abstract layers mainly based on the mapping of rules, we call the approach rule mapping-based refinement. This differentiates our approach from that proposed in [145], which also investigates refinement for the styles specified using graph transformation systems. However, they do not define fixed mappings between rules, but only between structural elements and transformation states. They check then whether all system states of an abstract model are also reachable at the concrete level, no matter by which order of transformation rules, vice versa. Their approach is not suitable for the systems that require correctness check of the fixed sequence of rules. In addition, it is very difficult to check the correctness of the completely different concrete behavior using their approach.

Our refinement criteria enable us to check the correctness of newly added elements through the scenario construction. We build refinement criteria for the newly added elements through a scenario-based construction, i.e., we check whether the abstract scenario (rules with a fixed sequence) is correctly mapped to the equivalent concrete scenario. The criteria associate the completely different concrete elements with the abstract ones. In this way, the correctness of the newly added rules and structural elements is semantically ensured. Besides, we can construct exactly the needed state graphs and transformation sequences for checking, and we do not need to explore and compare all the possible transition states between two graph transformation systems. This makes the approach efficient and practical to large systems. It also enables us to use the existing graph transformation simulation tool Fujaba as the basis for further automation.

4.3.4 Simulation

The operational semantics of the GTS allows us to execute the models and thus analyzing and validating the system through simulation (E). Simulating the dynamic behavior of the system allows the designers to execute the system and to play with specific scenarios. The designers can concentrate on the key aspects of the architecture. It can also detect errors and improve the confidence of the model. Besides supporting simulation, we also focus on providing a practical and usable process and environment to help the design and development of the style. The tool support of style specification, style refinement and other related concept is very important too. After comparing the available simulation tools, we choose the existing Fujaba [1] environment as our basis. Fujaba is a CASE (Computer-Aided

Software Engineering) tool that aims to push graph transformation systems to a broader industrial usage. Fujaba allows a seamless integration of object-oriented software development and graph transformation systems, which facilitates greatly the designers and developers who are not familiar with graph transformation systems. With the support of Fujaba, we develop three ways to use Fujaba simulation for further style-based analysis and automation. We also develop a style - based engineering process for style development.

4.3.4.1 Efficient validation

Given a specified graph transformation system G = ⟨ TG, C, R ⟩, we can validate the correctness of the specification using simulation. We define a start graph S_0 to describe the initial configuration of the system, and an application scenario as a sequence of rules. The resulting trace of the sequence of rules can be validated through Fujaba Dobs simulation. In this way we can test whether the model reacts in the desired way, thus validating functional completeness of the system.

In such a validation process, it happens often that the result is wrongly judged as incorrect, although the specification is proved to be correct later on. This is normally caused by inputting an incomplete initial state, or the reference test result is wrong itself, both of which result in naturally wrong test result or wrong judgement. This makes the validation process itself error-prone and very inefficient. We solve this problem by constructing the minimal initial state and the expected test result with the help of the algorithm developed for the scenario construction in the refinement part. The constructed initial state covers the minimal preconditions of all the involved rules, thus it is reasonable enough to allow the users to pursue the following execution of the sequence of rules. The expected test result is derived from the created intermediate state, which allows a correct judgement of a test result. Through such a construction, our approach can greatly enhance the correctness and efficiency of the validation process.

4.3.4.2 Behavioral consistency check

The correct refinement of an abstract conceptual style into a concrete one is important, and the validation process is usually difficult and complicated. In our refinement-based style development process, it is important to ensure that the architectures in a same hierarchy belong to the same family of middleware, i.e., they should realize and provide the same functionalities, although they belong to different abstraction layers and use different design strategies. In order to solve the problem, we construct a framework that checks whether the specified architectures on different abstraction layers provide the required functionalities, with the help of a standard reference application (F) and the Application Programming Interfaces (APIs) (G). The application covers the requirements for the middleware and it is used to validate the functionality completeness of the architectures, which is further specified as a sequence of operations using UML sequence diagram. It in-

vokes and uses the APIs provided by each style, for example, Concrete API and Conceptual API. From the execution result we can check and compare whether the architectures belong to the same family of middleware.

Since the three styles are specified on different abstract layers that have different focuses, we extend the style models to provide a consistent API. In order to do so, we encapsulate the rules of the style to provide a consistent API that allows the application to invoke. Because we formalize the refinement relationship mainly based on the fixed mapping of rules, we can now define the rule encapsulation based on the refinement mappings. We can map the operations implemented in Concrete API to the operations defined in Conceptual API through the type transformation and semantic match defined in the refinement process. In Fujaba, we can do the encapsulation at the specification stage by using its transformation rule editor, which supports complicated programmed graph transformations, and it allows users to specify advanced control flow with user-defined order.

4.3.4.3 Style - based engineering

With Fujaba's support, we develop a style - based engineering process for efficient style development, which includes style specification, style validation, refinement consistency check, behavioral consistency check and code generation. Taking as the first step, we use Fujaba to edit the full style specification, i.e., the type graph (UML class diagrams) and the graph transformation rules. Fujaba provides an UML class diagram editor that supports rather complete UML class diagram notions. The graph transformation rule editor uses UML activity diagram and UML collaboration diagram notions. During constructing the class diagram and rules, the editors already do consistency checks. For instance, they check consistency of the class diagram, the users are only allowed to create instance elements that are correctly typed over the class diagram, etc. This helps the designers very much to create a consistent and correct specification. Having finished the specification, we can efficiently validate it using the Fujaba simulator Dobs. After getting an abstract style specification and a refined concrete one, we can check the correctness of the mapped rules through checking whether the refined sequence of rules is reachable in Dobs. Afterwards, we use the behavioral consistency check framework to check whether the architectures in a same hierarchy belong to the same family of middleware. Finally, we get the validated styles that should be correct after passing the different validation processes. Taking as the last step, we generate automatically executable Java source code.

Chapter 5

Architectural Style-based Modeling

5.1 Overview

In this chapter, we will explain how to use the TGTS (Typed Graph Transformation System) to model the required aspects of the architectural style for the middleware. We will construct our modeling language based on the existing work of Thöne [145], which provides a formal foundation of specifying the architectural style (for distributed systems) using the TGTS, a combination of graph transformation techniques and meta-modeling approach. More specifically, they define the style through defining the style definition language (i.e., metamodel and graph transformation rules) that is specialized for a specific application domain. The style definition language constraints the allowed architectural elements (through the metamodel) and behavioral definition (through the rules) for a specific style. Therefore, there is no universal language that defines a style fitting to all the software systems, and it is necessary to have a dedicated language for providing appropriate solutions to different requirements of different application fields, which can be achieved mainly through defining specialized metamodels and rules.

Although exploiting the same TGTS to specify the style, we apply the TGTS in a different way from Thöne's since the requirements for the style vary. They focus on formalizing the TGTS to define a general style for distributed systems, whereas we focus on modeling the specific aspects required by the style for the middleware for mobile systems, and we provide a dedicated style definition language that satisfies the style requirements, i.e., we create a specialized metamodel and a set of rules for defining the style for the middleware for mobile systems. In other words, our modeling language is a light-weight extension of the TGTS proposed by Thöne. In addition, we do not add UML profiles to specify the style, and we do not add an additional UML layer to provide concrete syntax of the modeling language. This will enhance the simplicity and usability of the approach.

The chapter is organized as follows: We will start with the introduction of the related TGTS background knowledge and Thöne's work in Section 5.2. Afterwards, we will explain how to use the TGTS to model the different aspects required by the style specification in Section 5.3.

In this chapter, we will focus mainly on how to model the required aspects of the style, but not the process of constructing the style for a specific type of middleware, which will be illustrated in the next chapter through specifying the architectural style for the middleware for nomadic networks.

5.2 Background of the TGTS

We will introduce the related TGTS background knowledge, which includes graphs and graph morphism, graphs and object-oriented modeling, rules and graph transformation, metamodeling, etc. Afterwards, we will explain the general idea of using the TGTS to specify architectural styles [145].

5.2.1 Graphs and graph morphism

Graphs are often used as abstract representation of diagrams, e.g., UML class diagrams. Graphs and diagrams are a very useful means to describe complex structures and systems, and to model concepts and ideas in a direct and intuitive way. In particular, they provide a simple and powerful approach to a variety of problems that are typical to software engineering. For example, bubbles and arrows are often used to analyze a new software project. Moreover, a variety of visual diagram notations have been proposed for almost each activity of the software design and development process, such as state diagrams, UML diagrams, Structured Analysis, control flow graphs, architectural languages, function block diagrams, and several others.

A graph consists of a set of nodes, a set of edges and two mappings assigning a source and a target node to each edge. Formally, a graph is a tuple $G = (N, E, src, tar)$ with a set N of nodes, a set E of edges, and functions $src, tar : E \rightarrow N$ that assign source and target nodes to each edge. Variations include hypergraphs, where edges can be attached to an arbitrary sequence of vertices, attributed graphs, whose vertices and edges are decorated with textual or numerical information, or more complex object-oriented or hierarchical graph models.

Graphs may be related to each others by using graph morphisms. Formally, a graph morphism $f = (f_N, f_E) : G \rightarrow G'$ is a pair of functions $f_N : N \rightarrow N'$ and $f_E : E \rightarrow E'$ preserving source and target ($src' \circ f_E = f_N \circ src$ and $tar' \circ f_E = f_N \circ tar$).

5.2.2 Graphs and object-oriented modeling

In object-oriented modeling graphs occur at two levels: the type level (given by the class diagrams) and the instance level (given by all valid object diagrams). This idea can be described more generally by the concept of typed graphs, where a fixed type graph TG serves as abstract representation of the class diagram. Its object diagrams are graphs equipped with a structure-preserving mapping to the type graph, which is expressed as a *graph homomorphism*. Formally, given a graph

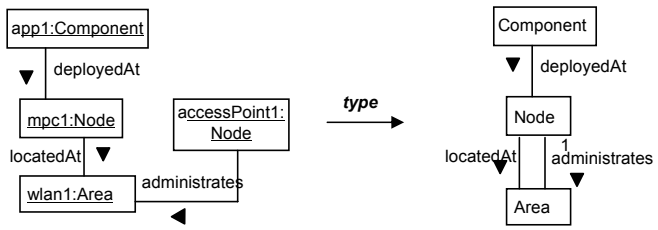

Figure 5.1: Object diagram (left) typed over class diagram(right)

TG, a TG-`typed graph` $\langle G, tp_G \rangle$ is a graph G equipped with a structure-preserving graph morphism $tp_G : G \to TG$ [39]. We call TG *type graph* and $\langle G, tp_G \rangle$ *instance graph over* TG. The category of TG-typed instance graphs is called $\mathbf{Graph}_{TG}$.

Fig. 5.1 shows examples of an object and a class diagram in UML notation. The object diagram on the left can be mapped to the class diagram by defining type(o) = C for each instance o : C in the diagram. Extending this to links, preservation of structure means that, for example, a link between object o1 and o2 must be mapped to an association in the class diagram between type(o1) and type(o2). By the same mechanisms of structured compatibility we ensure that an attribute of an object is declared in the corresponding class, etc.

5.2.3 Rules and graph transformation

Graph transformation is a good means to describe the dynamic aspects of various system structures, i.e. evolving class and object structures or reconfiguration of distributed networks. Graph transformation means the rule-based manipulation of graphical structure. Graph transformation rules describe graph transformation steps in an if then style. If a certain graph namely the left hand side of the rule does exist then it can be transformed into a new one the right hand side of the rule. Both sides may have some common nodes and edges. These nodes and edges remain unchanged during a transformation and act as central points where the transformation takes place. They form an intermediate graph which is part of the left and the right hand side of the rule.

Formally, a graph transformation rule $r : L \Rightarrow R$ consists of a pair of TG-typed instance graphs L, R such that the intersection $L \cap R$ is well-defined (this means that, e.g., edges which appear in both L and R are connected to the same vertices in both graphs, or that vertices with the same name have to have the same type, etc.). The left-hand side L represents the pre-conditions of the rule while the right-hand side R describes the post-conditions.

Graph transformation is defined by the application of graph transformation rules that model the permitted actions on graphs representing system states. Graph transformation defines a relation on state graphs that can be iterated arbitrarily yielding the transformation process. In this way, a graph grammar consisting of a

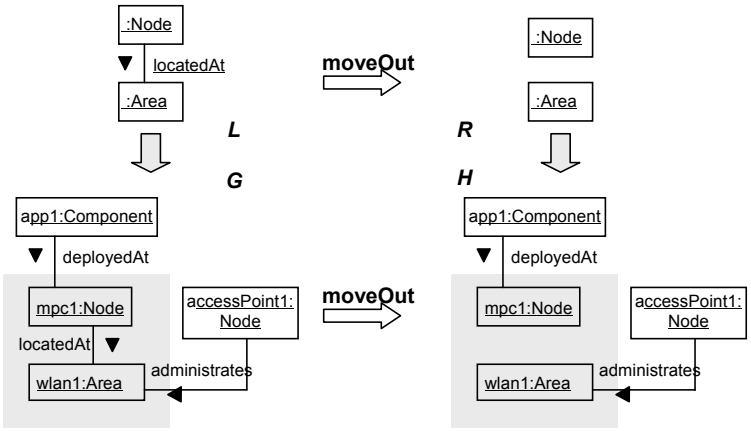

Figure 5.2: A sample transformation step using rule moveOut

start graph and a set of rules gets an operational semantics.

Formally, a graph transformation from a pre-state *G* to a post-state *H*, denoted by $G \overset{p(o)}{\Longrightarrow} H$, is given by a graph homomorphism $o : L \cup R \rightarrow G \cup H$, called *occurrence*, such that

- *o(L)* $\subseteq$ *G* and *o(R)* $\subseteq$ *H* , i.e., the left-hand side of the rule is embedded into the pre-state and the right-hand side into the post-state, and
- *o(L\R) = G\H and o(R\L) = H\G*, i.e., precisely that part of G is deleted which is matched by elements of L not belonging to R and, symmetrically, that part of H is added which is matched by elements new in R.

There are different approaches for the formal definition of graph transformation. They basically differ in the way how the local effect of a graph transformation rule is embedded into the original host graph. Fundamental approaches that are still popular today include the algebraic DPO (Double-PushOut) approach [48, 40], SPO (Single-PushOut) approach, the node-label controlled (NLC) [78, 51], the monadic second-order (MSO) logic of graphs [41], and the Progres approach [129, 128, 131], which represents the first major application of graph transformation to software engineering [52]. We follow the Double-Pushout approach [48, 52], where the application of a transformation rule like moveOut in Fig. 5.2 is performed in three steps:

1. Find an occurrence o_L of the left-hand side L in the current object graph G.
2. Remove all the vertices and edges from G which are matched by $L \setminus R$. We must also be sure that the remaining structure $D := G \setminus o_L(L \setminus R)$ is still a legal graph, i.e., that no edges are left dangling because of the deletion of their source or target vertices.

3. Glue D with a copy of $R \setminus L$ to obtain the derived graph H. We assume that all newly created nodes and edges get fresh identities, so that $G \cap H$ is well-defined and equal to the intermediate graph D.

5.2.4 Metamodeling

Metamodels [35, 89, 42] have been around for many years in a wide variety of different application domains and under various pseudonyms: data model, language schema, data schema, etc. Wherever there is a need to define a language, it is common to find a metamodel. This is particularly the case for standards, which by virtue of being a standard must have a precise definition. The Object Management Group (OMG) has been particularly involved in their use in the standards arena. One of the largest metamodels is contained in the UML specification, whose abstract syntax is specified by a subset of UML class diagrams. This subset is determined by the meta object facility (MOF) [108] specification which is also referred to as a meta-meta model because it has meta models as instances whose instances, in turn, are models.

The benefit of metamodeling is its ability to describe the modeled languages in a unified way. The languages can be uniformly managed and manipulated thus tackling the problem of language diversity. For instance, mappings can be constructed between any number of languages provided that they are described in the same metamodelling language. Another benefit is the ability to define semantically rich languages that abstract from implementation specific technologies and focus on the problem domain at hand. Using metamodels, many different abstractions can be defined and combined to create new languages that are specifically tailored for a particular application domain. Productivity is greatly improved as a result.

A metamodel [35] defines the vocabulary to be used for defining models. Modeling is intended to design systems using a predefined set of concepts. Metamodeling is intended to specify concept to be used for defining models. Generally speaking, a metamodel is a model of a modelling language. The term meta means transcending or above, emphasizing the fact that a metamodel describes a modelling language at a higher level of abstraction than the modelling language itself.

In order to understand what a metamodel is, it is useful to understand the difference between a metamodel and a model. Whilst a metamodel is also a model, a metamodel has two main distinguishing characteristics [35]:

- A metamodel must capture the essential features and properties of the language that is being modeled. Thus, a metamodel should be capable of describing a language's concrete syntax, abstract syntax and semantics.

- A metamodel must be part of a metamodel architecture. Just as we can use metamodels to describe the valid models or programs permitted by a language, a metamodel architecture enables a metamodel to be viewed as a model, which itself is described by another metamodel. This allows all metamodels to be described by a single metamodel. This single metamodel,

sometimes known as a meta-metamodel, is the key to metamodelling as it enables all modelling languages to be described in a unified way.

Two key expressions summarizing the metamodeling approach [2] as:

- an attempt to describe the world.
- for a particular goal.

The second expression illustrates the fact that there will never be a universal metamodel to define all the software systems from embedded to business framework ones. It is important to understand that a (meta) model has to be defined for a specific goal. Thus, there are a multitude of (meta) models. The need for dedicated means is required for providing appropriate solutions to different requirements of different application fields, such as e-business, telecom, or embedded systems.

5.2.5 Typed graph transformation system and style specification

The theory of graph transformation originated from the idea to generalize Chomsky grammars from strings to graphs. In analogy to string grammars, graph grammars consist of a set of node- or edge-replacing production rules that are used to define a certain graph-based language by the set of possible derivations from a dedicated start graph.

Since the declarative meta-modeling approach becomes more and more popular, especially in the area of modeling languages with UML being the best-known example, [145] applies the meta-modeling approach with the graph transformation techniques to specify the architectural style, which is called the TGTS (Typed Graph Transformation System) [122] $G = \langle TG, C, R \rangle$. TG is here UML class diagrams (or meta models). C is a set of constraints. R is a set of graph transformation rules.

The modeling graphs occur at two levels: the type level (the type graph TG) and the instance level (the instance graph typed over TG $\langle G, tp_G \rangle$). The type graph is used at the meta level for the definition of a style definition language. All graphs conforming to the type graph TG belong to the corresponding language. Hence, for defining the set of graphs representing valid architectures of a certain architectural style, they do not provide graph production rules but a declarative graph schema (the type graph). They rather use graph transformation rules, which can describe any local transformations of instance graphs of the schema, in order to specify possible changes of system states, i. e., the effects of architectural operations.

The style definition language constraints the allowed architectural structural and behavioral elements for a specific style. Therefore, there is no universal language that defines a style fitting to all the software systems, and it is necessary to have a dedicated language for providing appropriate solutions to different requirements of different application fields, which can be achieved mainly through defining specialized metamodels and rules. For example, [145] specifies type graph

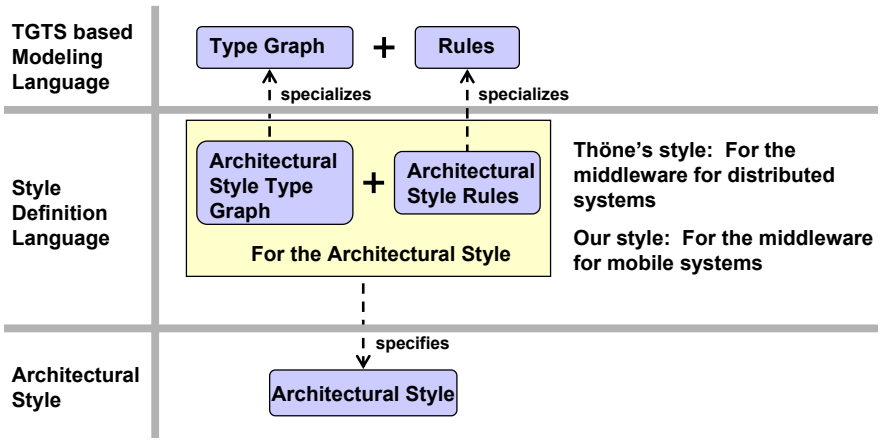

Figure 5.3: TGTS - based style modeling

(TG) and rules (R) that are specialized for the style for the middleware for distributed systems. We need to define a new language for other application domains that have different requirements, i.e., we need to create specialized type graph and rules in order to define the style for the middleware for mobile systems. This adaption is a kind of light-weight extension of the modeling language.

The approach is similar to the metamodeling-based language driven development approach [35] that adapts the language (but not the model) to new application domains and new requirements. The TGTS provides a flexible way to extend or change the models. This allows adaption of a specialized model to meet new requirements of a new style. The adaption includes two parts: the type graph and rules. Class diagrams (Type Graph) provide a powerful and flexible way to model architectural elements and their relationships. New architectural elements can be easily added through adding proper associations and classes. While unnecessary elements can be deleted together with the related associations. The relevant rules built on instance graphs can be adapted easily to be typed over the adapted type graph. Rules can be added or deleted without violating the semantics of the TGTS.

5.3 Specification of the style

As mentioned in last section, there is no universal language that defines a style fitting to all the software systems, and it is necessary to have a dedicated language for providing appropriate solutions to different requirements of different application fields. We will explain now how to use the TGTS $G = \langle TG, C, R \rangle$ to specify the required aspects of the style for the middleware for mobile systems (see Section 3.2). Our style specification is a light-weight extension of the TGTS-based modeling approach proposed in [145]. We define specialized metamodels and rules for the style. In order to present the style in a more structured way, we will separate it into two main parts: the structural part and the behavioral part. We will discuss in

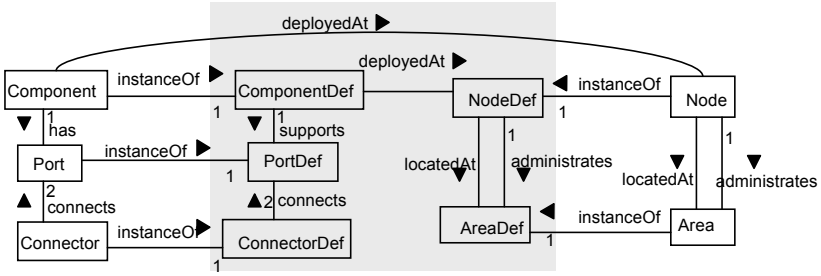

Figure 5.4: Type graph of the style

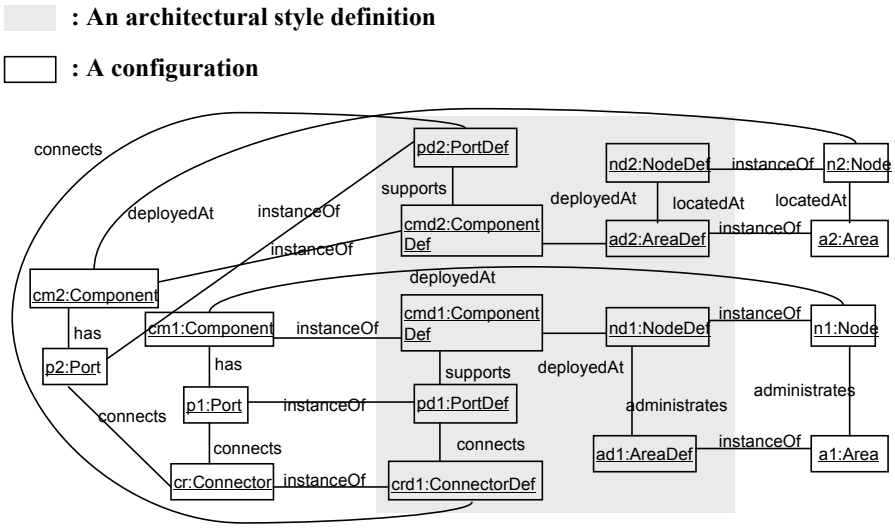

Figure 5.5: An exemplary instance graph of the style

the third section the semantics and syntax of the modeling language.

5.3.1 Structural part

The structural part includes the vocabulary, constraints, and configuration (see Section 3.2). We specify this part using the type graph TG, which is based on the MOF (meta object facility) [108] that provides the standard OO modeling concepts including MOF packages, classes and associations[1]. They are visualized using UML class diagrams. The object diagrams are called *instance graphs* that are typed over the type graph. We use the type graph to define the *vocabulary*, allowed *configurations*, *constraints*, and allowed *type definition* of the style. The instance graphs of the type graph represent both the declarative definition of an architectural style

[1]If unspecified, the use of the terms package, class, and association means MOF package, MOF class, MOF association

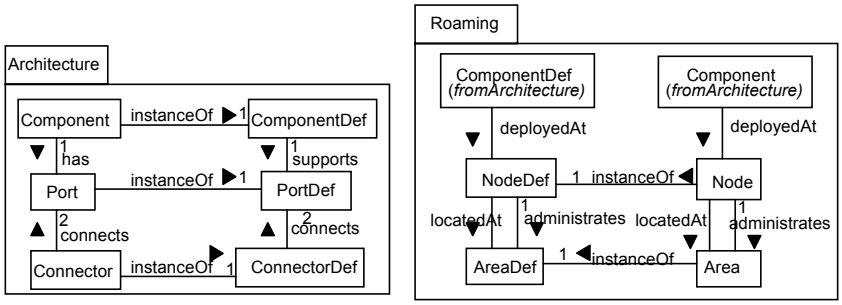

Figure 5.6: The type graph in different packages

and the style configuration.

More specifically, the classes define the vocabulary of the architectural style. Associations define the possible links and relationship among these elements. Simple constraints, already included in the class diagrams, are cardinalities that restrict the multiplicity of links between the elements. More complex restrictions can be defined by OCL (Object Constraint Language) constraints.

For example, we define an architectural style for a component-based middleware that composes components, which are deployed on physical nodes that can be located in specific locations. Fig. 5.4 shows the type graph definition of the style, while Fig. 5.5 is about an instance graph of the style. The classes named after "Def" such as *ComponentDef, ConnectorDef, PortDef, NodeDef, AreaDef* (in Fig. 5.4) represent the allowed vocabulary of the style. The associations define the relationship between the vocabulary. The classes *Component, Connector, Port, Node, Area* with their associations (Fig. 5.4) represent allowed configurations of the style. At the instance level (Fig. 5.5), the architectural style description and configurations are specified. In this way, the required architectural design elements of the style such as space and middleware construction components can be specified at type level and instance level. The relationship and constriction among them are also modeled in the same class diagram.

What's more, we can group different architectural concerns using the package concept, in order to decrease complexity of the model [72, 70, 71]. For instance, we can separate the type graph into two packages (Fig. 5.6): *Architecture* and *Roaming*, each of which represents a specific concern of the system, i.e., software architecture and roaming. The class diagram allows a uniform representation where elements of different submodels are represented as vertices of the same abstract syntax graph, i.e., an instance of the meta model. Based on this uniform representation, the different sub-models can be related by associations between the common elements belonging to different submodels.

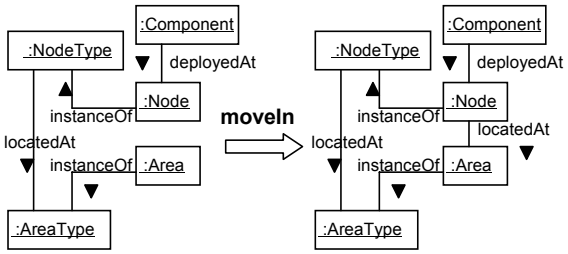

Figure 5.7: Rule: moveIn

5.3.2 Behavioral part

The behavioral part describes the runtime behavior of the system, which includes component interaction and dynamic changes (see Section 3.2). Correspondingly, we use the graph transformation rules R to specify the dynamic change and component interaction. We can differentiate two kinds of rules: rules for dynamic changes and rules for component interaction, which will be explained separately in the following sections. Besides, we will explain how to use the rule to specify IDL semantics in the third section, since it can be seen as the component interaction in a platform-specific concrete style, and it will be used when specify a concrete middleware.

5.3.2.1 Rules for dynamic changes

Dynamic changes are provided by the style to let an architecture evolve at runtime and to change its current configuration in the presence of mobility, i,e, when the component moves to other spaces, and when the wireless connection is not available, etc (see Section 3.2). Based on the uniform representation of space and architectures as meta model instances, we can now specify dynamic changes in the presentation of mobility as graph transformation rules typed over the type graph specified in the structural part (see Section 5.3.1). The movement style of the components in the specified space, the disconnection of the wireless connection and other actions related to mobility can be now modeled through the graph transformation rules. The rule defines a certain modification of instance graphs. Rules for dynamic changes specify the reconfiguration of the style in the presence of mobility. Therefore, the instance graphs represent architectural configurations.

Taking as an example, we will introduce a rule *moveIn*(in Fig. 5.7) that describes the reconfiguration when a node moves. The left side of the rule describes the pre-conditions that must be satisfied when reconfiguration happens, and the left side describes the post-conditions that is the result after the reconfiguration happens. As precondition, there should be a *Node* and an *Area* whose types *NodeType* and *AreaType* should be connected by a *locatedAt* link. That means the node is of a type that is supported by the area, like a cell phone in a GSM cell. In this

```
bool :moveIn (Area: a ) {

if (! is_subtype(a. type(), AREA_TYPE))
     {
      return false;
     }
else
     {
      return moveIn(a);
     }
 }
```

Figure 5.8: Code to check rule MoveIn

case, the rule can be applied with the result of creating a new *locatedAt* link between the two instances. This is expressed in the post-condition of the rule shown on the right-hand side. The rule is typed over the type graph (in Fig. 5.4), which requires the type declaration when new elements is created on the right-hand side.

To see why this is useful (and necessary), consider the java code generation of rule moveIn, which moves a node into a area. In the generic case the nodes may be move into any kind of area. In the case of the architectural style for the middleware, however, we may want to allow a node to move into an area only if it is an instance of one of the area types defined in the style, namely, a WLAN, a GPRS area. It is reasonable, therefore, to cause an invocation of moveIn to fail if the parameter is not one of these two types. On the other hand, if one node moved into an area, then the observable effect should be the same as in the generic case. Fig.5.8 shows the java code for doing this.

Similarly, we can define rules for other dynamic change related activities in a style. For example, we define rules: *moveIn, moveOut, connect, disconnect* and *bind, unbind, handOver* for a conceptual style in Section 6.3. Rules *moveIn, moveOut* model the movement of components, *connect, disconnect* model the connection and disconnection of wireless networks, *bind, unbind* model the creation and deletion of a session between two components when they interact, and *handOver* models the main functionality of the middleware.

The specified rules for dynamic changes are applied non-deterministically simulating ad-hoc reconfigurations. The main benefit of graph transformation for describing software architectures is the ability to model dynamic reconfigurations in an abstract and visual way. Some approaches [42, 63, 96, 93] assume a global point of view when describing reconfiguration steps which, in a real system, would correspond to the perspective of a centralized configuration management. In a mobile system, the existence of such centralized services cannot be taken for granted. We model reconfiguration from the point of view of individual components that synchronize to achieve non-local effects. Here, locality corresponds to context freeness, that is, a rule is local if it accesses only one component (or connector) and their immediate neighborhood. Additional preconditions concerning the current topology configurations can be used to restrict reconfigurations to certain phases

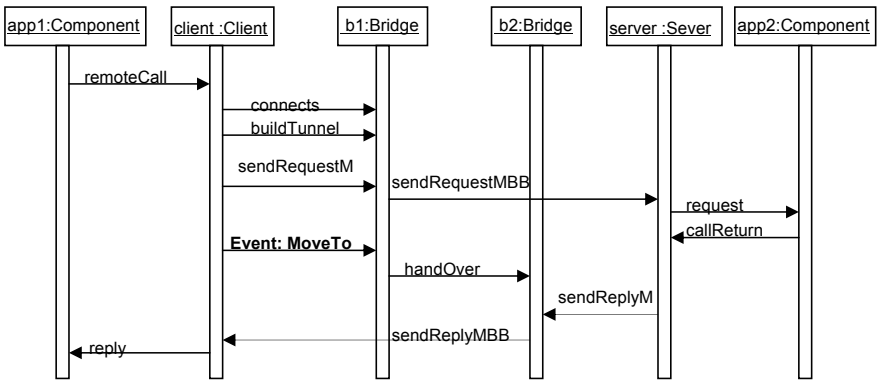

Figure 5.9: Sequence diagram for a remote invocation process

of the component behavior.

5.3.2.2 Rules for Component Interaction

As introduced in Section 3.2, component interaction covers component communication, collaboration, and coordination. We use graph transformation rules to specify the component communication and collaboration, whilst the coordination and synchronization between different actions and components are specified using UML Interaction diagrams. Through this way, we can specify a specific component interaction pattern supported in the middleware for mobile systems.

For example, we will specify the adapted RPC interaction pattern in Section 6.5 to show how mobility influences the component interaction. Figure 5.9 defines the sequence of operations for one remote procedure call using a UML Interaction diagram[2] The semantics of operations is defined using graph transformation rules. The left side of the rule defines the precondition to execute the operation, where the right side of the rule defines the postcondition after executing the operation. In a mobile setting, the components involved in an interaction change dynamically due to their migration or connectivity patterns. Using graph transformation rules, the dynamic reconfiguration among the coordinating components concerned in interactions can be modeled naturally.

The main difference between the rules for dynamic changes and rules for component interaction is that the former has a nondeterministic sequence, while the latter presents a fixed sequence, which is specified using UML Interaction diagrams. In addition, the instance graphs of the former represent architectural configurations, while the instance graphs of the latter represent interaction states.

Our specification of component interaction and dynamic change using graph transformation rules can model the influence of mobility to component interaction

[2]We use UML 2 notations for Sequence diagrams in this paper. Filled arrowhead lines show a synchronous message, while stick arrowhead lines show an asynchronous message.

naturally. The dynamic change and component interaction are integrated smoothly and directly into one specification with the same semantic background. This enhances the simplicity of the specification very much, and it eases the designers and developers to understand the model and to use the approach.

5.3.3 Syntax and semantics of the modeling language

As having mentioned in the precious section, we construct our modeling language based on the existing work of [145]. Our modeling language is a light-weight extension of the TGTS proposed by [145]. In order to provide a user-friendly concrete syntax, [145] adds a UML layer on top of the graph-based representation. The application architects can then design software architectures modularized into different views using various UML diagram types. Internally, CASE tools will translate these models into the corresponding formal graph representation so that the above mentioned graph transition system can still be derived and used for further analysis and refinement checks. In addition, they [145] use UML profile to provide various style-specific ADL variants, and each of which semantically backed up by its own graph transformation system. For defining architectural styles, style architects need expertise in both areas: they assemble syntactical style constructs in a UML profile, setting up a new UML variant, and define a graph transformation system as the underlying operational semantics. By a formal relation, they fix the correspondences between style syntax and semantics.

[145] provides a quite complete method of using the GTGS combined with the popular UML CASE tools theoretically. They separate the style specification into different parts that are further assigned to different responsible architects, namely, style architect, platform architect and application architect. The existence of so many different specifications can easily lead to misunderstanding in a real project. Besides, there are too many ideas and concepts for the approach, which makes it not easy to use. For example, the way to construct the concrete syntax is rather complicated, and it requires that the style architects have deep knowledge in both GTS and UML. In addition, the approach can only ease the work of the application architects with the help of a CASE tool that provides such a combination of UML standard diagrams and the GTS. However, such CASE tools do not exist and it is not a easy job to construct the tool.

We think that simplicity and usability is a very important aspect of a model-based approach. The visual presentation of the graph transformation system with the UML class diagrams are understandable enough for the users. Therefore, we do not add UML profile to specify the style, and we do not add an additional UML layer to provide concrete syntax of the modeling language, although using the same graph transformation system for defining semantics. In our approach, meta modeling defines the concrete syntax, abstract syntax and static semantics, while the GTS specifies the operational semantics of the metamodels. In this way, we do not need to provide different specifications to different architects that have different responsibilities. There exits only one common specification, which will simplify

the process of learning and communication. In addition, we can use directly the existing GTS tools like Fujaba for further simulation and analysis, which makes the approach more usable.

Concrete Syntax Our metamodel definition is based on the MOF [108], which provides the standard OO modeling concepts including packages, classes and associations. These are visualized using class diagrams, which are the *concrete syntax* presented to the end user. The users will use directly class diagrams with rectangles, lines, arrows, etc. Here, each notation element is concretely rendered in terms of the geometrical elements that define its appearance, like other diagrammatic languages. For example, classes are represented through rectangles, or relationships among these elements can be a line connects two rectangles, etc.

Abstract Syntax MOF also provides some well-formedness rules of the language. For example, the well-formedness rules for classes include multiplicity and ordering constraints. Besides, the typical role of a metamodel is to define the static semantics for how model elements in a model gets instantiated. In our style specification, a style-specific vocabulary of design elements is introduced by providing subtypes of the basic architectural classes. The instances of the type graph represent both a declarative definition of an architectural style as well as an individual configuration and their interrelation by the meta association *instanceOf*.

Semantics We use graph transformation system to define operational semantics based on a direct interpretation of meta models at the level of abstract syntax. Consequently, the semantics of the language is fully integrated in the language's definition. The operational semantics is easy to understand and write, since it is expressed in terms of operations on the language itself.

Chapter 6

Style Examples

6.1 Overview

After explaining in Chapter 5 how to use the TGTS to model the required aspects of the style, we will illustrate now the process of developing the style through constructing the style of the middleware for nomadic networks. This type of middleware is quite mature from technique's point of view, and many existing middleware can be categorized into it. In spite of that, there is no common agreement or understanding of the middleware, not to mention available design standards that help the design and analysis of the middleware. As a result, it is very difficult for the designers to reuse the already established design knowledge or successful experience when building new systems. This makes the design process quite inefficient and unpredictable, and therefore risking the project.

The style for the middleware can solve those problems. We will develop the style that represents a common form of design for this class of middleware, and that originates from the results practitioners have achieved in the area. As having outlined in Section 4.2, we will construct the style on three different abstract layers, i.e., the conceptual style, the platform-independent concrete style and the platform-specific concrete style, in order to decrease complexity and enhance reusability. Each of the styles captures different architectural aspects or views, and they can be used to help the design and analysis of the middleware at different design stages.

The conceptual style specifies a style of mobile systems for nomadic networks, or a mobility style supported by the nomadic network, which includes the movement of the components, and the specific service provided by this kind of mobile system. It is about the requirement analysis for the middleware and it can be used at the very early stage of designing the middleware. The platform-independent concrete style specifies the design-specific component interaction pattern adapted for the mobility style. We choose the adapted RPC as the example. The style can be used when we design a specific component interaction pattern for the middleware at a more abstract level.

The platform-specific concrete style is a more refined version of the platform-independent concrete style. It can be used when we define a complete design spec-

ification of the middleware, e.g., the Wireless CORBA specification in our case, which is originally specified using the IDL (Interface Definition Language), whose semantics is informally specified using natural language. This is a potential cause of conflicts and ambiguities. Especially when the components have complicated behavior or many components are involved to perform a task in dynamic mobile settings. When we specify the style for Wireless CORBA using the TGTS based modeling approach, we also provide formal semantics to the IDL specification at the same time. We can specify formally the most important aspect of the component interaction in mobile settings, i.e., the dynamic change of components involved in interactions, the collaboration and synchronization definition between different objects and operations, etc. The errors caused by conflicting or unclear definition in the specification can thus be already avoided at the specification stage.

The first step of the approach is to abstract the aspects for constructing the style. Since we have concluded the architectural commonalities of the middleware in Section 2.4, we will recall and explain in detail these aspects, which will be used to build the conceptual style and the platform-independent concrete style. Afterwards, we will introduce a concrete middleware: Wireless CORBA, which will be taken as the example of the platform-specific concrete style. This part is explained in Section 6.2. The three styles are explained respectively from Section 6.3 to Section 6.5.

The specified three styles will be used later for demonstrating other related parts in the following chapters, i.e., refinement, consistency check and simulation.

6.2 The middleware for nomadic networks

In Section 2.4, we have compared the middleware platforms that belong to the same class, and abstracted the common design aspects to be modeled for constructing the style. We will recall and explain in detail these aspects. We will also introduce a concrete middleware: Wireless CORBA as the example of the platform-specific concrete style.

6.2.1 Architectural commonalities

In Section 2.3, we have generalized a framework for describing the middleware, which allows us to compare different approaches from a common point of view, and observe similarities and common aspects from the great diversities of the middleware. We have also concluded some commonalities of the reviewed middleware regarding space definition, main functionality and component interaction / design pattern if we consider the supported wireless network of the middleware (see Section 2.4). Accordingly, we can roughly separate the middleware into two classes: the middleware for nomadic networks, and the middleware for ad hoc networks.

Many existing middleware can be classified into the middleware for nomadic networks, which has quite mature techniques. The examples are: event-based

	Aspects to be modeled (Structural)	Aspects to be modeled (Behavioral)
Space	Structured space	moveIn, moveOut, moveTo

Table 6.1: Space: aspects to be modeled

	Aspects to be modeled (Structural)	Aspects to be modeled (Behavioral)
Functionality	Session, Component, Wireless connection	bind, unbind, connect, disconnect, handover

Table 6.2: Main functionality: aspects to be modeled

(or publish/subscribe) middleware REBECA [157], JEDI [43], Siena [28] and JECho [33]; tuple space-based middleware TSpaces [156], JavaSpace [141]; object and component middleware Wireless CORBA [111], ALICE [66], RAPP [132] and DOLMEN [149]. These middleware shares some common design strategies and characteristics:

Space The nomadic network utilizes the infrastructure as access points to establish connectivity. Examples of this kind of network are Telecom networks like GPRS/GSM/UMTS with base stations, Wireless LAN (WLAN) with access point support, etc. The nomadic network has a structured space, which is divided into regular patches with an access point (or base station) supporting the communication needs of the components in that special area. Hence, components are allowed to move inside the scope of areas covered by access points. More specifically, we can describe the aspects to be modeled as given in Table 6.1.

Main functionality Generally speaking, the middleware for nomadic network focuses on how to provide continuous connectivity and other services when components move across the structured spaces where handover protocols are often used. Handover originates from the telecom network like GSM/GPRS/UMTS [77, 7] that belongs to nomadic network, where handover protocol is used to provide continuous connectivity service when mobile terminals roam across different cells. Handover in this case is managed by the physical network layer. It is defined as the procedure [149] of changing radio connections between a network and a mobile terminal due to well determined reasons, which include degradation of the radio link quality, requirements on the spectrum, user requirements or management reasons, etc. The type of handover [77] may also differ, mainly depending on the capabilities of the underlying transport network. Generally, the changing of the radio connection is not noticeable to the user.

In the middleware for nomadic networks, centralized or intermediate components are often introduced in order to provide mobility support of the application

	Aspects to be modeled (Structural)	Aspects to be modeled (Behavioral)
Component Interaction (adapted RPC)	Client, Server, Operation, Parameter, Message, Bridge	remoteCall, request, callReturn, reply, dispatchMessage, sendMessage, handoverConcrete

Table 6.3: Component interaction: aspects to be modeled

components. Depending on the supported component interaction models, the functionalities of the introduced components are different. For instance, in the event-based (or publish/subscribe) middleware REBECA [157], JEDI [43], Siena [28] and JECho [33], these additional components are responsible for buffering event notifications when disconnected, or rerouting and delivering the notifications to different locations where the roaming client currently locates (see Section 2.4). Handover is often used for redirecting the notifications. While in the object/component middleware Wireless CORBA [111], ALICE [66] and DOLMEN [149], the intermediate objects are responsible for redirecting and delivering the invocations to different locations where the roaming object currently locates. Handover protocol is used between the intermediate objects for redirecting the invocations.

In spite of the different functionalities of the components, the middleware shares a common target of using handover: to provide continuous service to roaming application components. It is a procedure of changing the control between middleware construction components and mobile application components. The handover in the middleware is pursued on the logical session layer, which can be seen as retaining the session between two components. This differentiates the handover on the middleware layer from that pursued on the physical network layer. Although the handover procedures on the two layers are different, they always associate with each other. The handover in the middleware is often activated by the handover on the physical network layer. That means, the network connectivity change on the physical network layer often causes the handover on the middleware layer.

In conclusion, the *handover* process should be modeled for the main functionality, which includes keeping sessions, connecting (i.e., *connect*) and disconnecting (i.e., *disconnect*) the *wireless connection*, etc. (Table 6.2). The *session* related operations like *bind* and *unbind* need to be modeled too in order to construct a complete system.

Component interaction / design pattern The favorable component interaction model is influenced greatly by the physical network. The middleware for nomadic networks allows the existence of central servers to provide service to other components. Such servers can be fixed and other roaming components can always communicate with them. Communication behavior is tied to central servers here. Many interaction paradigms designed for distributed systems are adapted to sup-

port mobility in the nomadic network. A typical example is the synchronous communication paradigm RPC (Remote Procedural Call), which is often adopted in the middleware area and in the software architecture area.

We will take the RPC model as our object for building the platform-independent conceptual style. It is very helpful if we investigate how such a model is adjusted to support mobility in the middleware for mobile systems. We can understand better what do we need in the software architecture in order to support mobility. There exist different RPC mobility models. We will take the one supported by the OMG standard specification Wireless CORBA [111], which is also adopted by ALICE [66].

RPC supports the definition of *server* components as RPC programs. A RPC program exports a number of parameterized procedures and associated parameter types. *Clients* that reside on other hosts can invoke those procedures across the network. RPC based middleware provides compilers to automatically generate client and server stubs. The client stub is responsible for marshalling the *parameters* into a *message*, and dispatching (i.e., *dispatchMessage*) them to the host where the server component is located. The server component unmarshalls the message and executes the procedure. Upon completion, it transmits marshalled results back to the client, if required. The basic form of interaction is synchronous, i.e., the client component that issues a request is blocked until the server component returns the response. The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In other words, RPC is transparentthe calling procedure should not be aware that the called procedure is executing on a different machine or vice versa.

The RPC-based middleware for distributed systems only needs to define two components: client and server component. An invocation also involves only these two components. In order to support mobility in a nomadic network, more components are needed. For instance, Wireless CORBA [111], ALICE [66] and DOLMEN [149] introduce intermediate objects (called *bridge*) that connect the roaming objects to the fixed network side. The bridges are also responsible for redirecting and delivering the invocations to the roaming objects, which is done through sending messages between bridges (i.e., *sendMessage*). As introduced previously, handover (i.e., *handoverConcrete*) protocol is used between bridge objects for providing continuous invocation service.

Besides, the interaction and configuration between objects in the RPC mobility model is rather dynamic. The configuration changes dynamically when application objects move in or move out of area managed by the intermediate objects, or when the wireless connection disconnects or connects. Especially, an invocation between application objects involves the intermediate objects, which provide continuous invocation connection service. An object can dynamically establish connections to the intermediate object within its current vicinity. The connection is not valid anymore when the client moves to other area. The configuration of the coordinating intermediate objects changes too when the application objects move to other areas. A new configuration of the coordinating intermediate objects will be thus built.

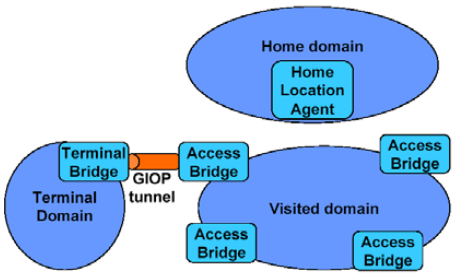

Figure 6.1: Wireless CORBA architecture

We identify the aspects to be modeled for the adapted RPC in Table 6.3.

6.2.2 The concrete middleware: Wireless CORBA

As one of the middleware for nomadic networks, Wireless CORBA has the architectural commonalities introduced in the last section. At the same time, it has its own design specific aspects, which will be explained now. Especially, we will focus on the six aspects listed in the general framework for capturing the middleware (see Section 2.3), which will be used for constructing the platform-specific concrete style. Since Wireless CORBA comprises IDL (Interface Definition Language), we will also explain the shortcomings of the IDL specification. We will show later (see Section 6.5) how we improve the specification using our modeling method.

6.2.2.1 Introduction

As a popular and important middleware, CORBA [110] is adjusted to support mobility by many researchers [112]. There are quite a lot of models developed. Wireless CORBA, ALICE [66], DOLMEN [149], RAPP [132], OpenORB [21], dynamicTAO [82] are examples. Inside these models, only Wireless CORBA is a standard OMG specification that supports mobility in nomadic networks. Wireless CORBA is built by both the industry and academy side. The possible issues of using CORBA in wireless environment are thoroughly discussed [107, 109]. The main design strategy of Wireless CORBA is adapted from the past successful project DOLMEN [149]. It is a middleware that has a sound theoretical foundation and practical basis. Therefore, it will be very helpful to see how wireless CORBA works, and to use Wireless CORBA as the concrete example.

In 2001, the Object Management Group (OMG) has adopted the specification of Wireless Access and Terminal Mobility in CORBA [112], Wireless CORBA, for short. It specifies the architecture and methods how CORBA can be used over wireless links. Wireless CORBA is designed for nomadic networks that need the support of fixed infrastructure to provide mobility management. As shown in Fig. [112], the Wireless CORBA architecture identifies three different domains:

the home domain, the visited domain, and the terminal domain. The home domain is the mobile terminal's administrative home. The Home Location Agent in the home domain is responsible for tracking the location of each terminal owned by its domain. The terminal domain defines the mobile terminal. The visited domain represents the fixed network side, where access bridges provide different access points to the mobile terminal.

In CORBA, a bridge (object) is used to provide protocol interoperability between different domains, and GIOP (General Inter-ORB Protocol) specifies most of the protocol details that are necessary for client and server objects to interact. In wireless CORBA, terminal bridge and access bridge are used between terminal domain and visited domain to provide GIOP message encapsulation and transmission (called GTP: GIOP Tunneling Protocol) over the wireless links.

Besides these main architectural elements shown in Fig. 6.1, the Mobile IOR (Interoperable Object Reference) is also an important concept in Wireless CORBA. The Mobile IOR is a relocatable object reference that identifies the terminal on which the target object resides and the Home Location Agent that keeps track of changes in the Access Bridge to which the terminal is actually attached. This allows CORBA's normal invocation forwarding mechanisms to be used to locate the terminal at invocation time.

6.2.2.2 IDL specification

Functional requirements towards a concrete middleware are often structured through interface definition languages (IDL)s, e.g., IDL for CORBA [110], Microsoft IDL (MIDL) for DCOM [133], or Java interfaces in the case of RMI [117]. Wireless CORBA comprises three IDL files (see Appendix A), which are *MobileTerminal.idl*, *MobilityEventNotification.idl* and *GIOP.idl* (see Appendix A). Each of the Wireless CORBA IDL files specifies one specific part of the system. *MobileTerminal.idl* specifies the elements to construct the architecture of Wireless CORBA. *MobilityEventNotification.idl* specifies mobility service booking and notification. *GTP.idl* specifies the GTP message classification and channels for transmitting the GIOP messages. Besides, Wireless CORBA includes the basic two IDL files of CORBA since it is developed based on CORBA, which are *orb.idl* and *IOP.idl*. *orb.idl* specifies the object invocation related elements. *IOP.idl* specifies protocols for interoperability between different objects.

The semantic part of an IDL is only expressed informally in natural language. This is a potential cause of conflicts and ambiguities, especially when components have complicated behavior or many components collaborate to perform a task in a mobile setting. As a result, the gap between a middleware's specification and its implementation is huge, and it is difficult for developers to derive implementations directly from specifications. Several implementations of the same specification may thus show different behaviors.

For example, there are errors discovered when developing a reference implementation for Wireless CORBA [79, 107]. Some errors are caused because of con-

	Aspects to be modeled (Structural)	Aspects to be modeled (Behavioral)
Space	Structured space, VisitedDomain, TerminalDomain, HomeDomain	moveIn, moveOut, moveTo
Wireless Connection	AccessBridge, TerminalBridge, Wireless connection	connect, disconnect, connectBridge, disconnectBridge
Component Interaction (Interoperability)	Object, Operation, Parameter, Client, Server, ObjectReference, MobileIOR, AccessBridge, TerminalBridge, HomeLocationAgent	call, request, callReturn, reply, dispatchMessage, sendMessageConcrete, buildTunnel, deleteTunnel
Component Interaction (Message & handOver)	GIOPMessage, GIOPRequest, GIOPReply, Operation, GIOPTunnel, GTPMessage, GTPForwarding, EstablishTunnel, HandOffTunnel	establishTunnelRequest establishTunnelReply releaseTunnelRequest releaseTunnelReply updateLocation

Table 6.4: Wireless CORBA: aspects to be modeled

flicting or unclear behavior definition in the specification. For example, the error named as Issue 4422 [107] is "New Access Bridge cannot distinguish between different kinds of handoff"[1]. That means, in the terminal initiated handoff and access recovery process, the new access bridge needs to invoke different methods depending on the kind of handoff. However, it is not possible for it to make this distinction based on the information it has received in previous steps of the handoff. This error is caused because the synchronization between different objects and operations is not defined clearly in the specification. The same reason holds for the Issue 4614: "A possible race condition with location updates during recovery". Such errors are not trivial issues, and they will cause serious problems during implementation.

6.2.2.3 Aspects to be modeled

As we can see, the three IDL files of Wireless CORBA (see Appendix A) are quite complicated, not to mention if we add other two files of CORBA. It is not sensible if we build the style that captures every aspect of the specification. The style should focus on the important aspects. As explained in Section 2.5, there are six aspects need to be considered when modeling a middleware for mobile systems, which are space, wireless network connection, dynamic change, component interoperability, component interaction, and context-awareness. We will conclude the aspects to be modeled in Table 6.4.

Component interoperability Wireless CORBA is an object-oriented middleware that supports the definition of object services. The service provided by a component is encapsulated as an object and the interface of an object describes the provided service, which is a set of method calls defined through an IDL. The interfaces defined in an IDL file serve as a contract between a server and its clients.

[1] handoff is another name for handover. We do not distinguish them in the thesis

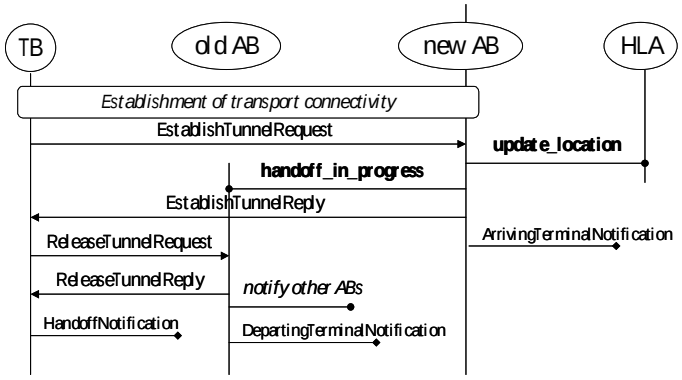

Figure 6.2: Message sequence chart of the terminal initiated handoff

Clients interact with a server by invoking methods described in the IDL. Marshalling operation parameters and results is by stubs that are generated from an interface definition.

As mentioned in last section, this basic object invocation related part is defined in the CORBA specification *orb.idl*. We will not go into details of CORBA, but focus on the aspects that are specific for Wireless CORBA, e.g., how to provide continuous invocation service when mobile terminals move between points of connectivity. Thus, the basic object invocation related part will be kept nearly the same as in the more abstract style (see Table 6.4), which are *call, request, callReturn, reply, dispatchMessage*, etc. The main difference is that Wireless CORBA uses *GIOP tunnels* for transmitting the messages between bridges. The process of sending messages is pursued between bridges (i.e., *sendMessageConcrete*). In addition, the message for invocation is specifically encapsulated as *GIOP messages*, i.e., *GIOPRequest*, *GIOPReply*, etc.

We will classify the component interoperability as one subpart of the component interaction (Table 6.4) since it is realized through component interaction.

Component interaction Wireless CORBA supports distributed object interaction, that is, a client object requests the execution of an operation from a server object that may reside on another host. This interaction evolved from RPC. The basic form of interaction is synchronous, that means the client object issuing a request is blocked until the server object has returned the response.

In order to support mobility, Wireless CORBA [111] introduces intermediate objects (i.e., *access bridges*) that connect the roaming objects to the fixed network side. The access bridges are also responsible for redirecting and delivering the invocations to the roaming objects. Wireless CORBA uses *handover* to ensure that the invocation between clients and servers are kept consistent. Handover will forward the invocation message from an old attached access bridge to a new attached bridge. The *Home Location Agent* will be responsible for tracing the current loca-

tion (i.e. the attached access bridge) of the mobile terminal. Besides, the terminal bridge is also involved in the invocation procedure. The roaming terminal has a *terminal bridge* that connects the terminal with the access bridge over wireless link. The invocation from (or to) the objects located on the terminal will be transferred through the terminal bridge. The *Mobile IOR* is also an important concept in Wireless CORBA, which identifies the terminal on which the target object resides and the Home Location Agent that keeps track of changes in the Access Bridge to which the terminal is actually attached.

In wireless CORBA, many components are involved to perform an invocation procedure, especially when the terminal is moving across several access bridges, and thus handover happens. The handover process involves many *GTP message* changes. GTP messages are very important in Wireless CORBA, which are used to provide GIOP message encapsulation and transmission over wireless links between bridges. The handover is mainly specified as a process of keeping the GIOP tunnel for transmitting the GIOP messages between the terminal bridge and the new access bridge. For instance, the terminal initiated handoff scenario (Fig. 6.2) specified in Wireless CORBA([112]) builds a tunnel between the new *Access Bridge newAB* and the *Terminal Bridge TB*. The old tunnel between the old *Access Bridge oldAB* and the *Terminal Bridge TB* is deleted. The important messages for handover include *EstablishTunnelRequest, EstablishTunnelReply, ReleaseTunnelRequest, ReleaseTunnelReply*, etc. Besides, operations like *updatelocation, handoffinprogress* are also important.

The tunnel related messages are very important for modeling the handoff process. However, it becomes too complicated if we just want to build a GIOP tunnel for normal message transmission, which does not involve mobility related dynamic changes of the components. Therefore, it is not meaningful if we model the intermediate steps of processing the messages for building the tunnel in this case. We prefer having an additional simple version instead, which can be abstracted as *buildTunnel* and *deleteTunnel*.

Other messages like *ArrivingTerminalNotification, DepartingTerminalNotification, HandoffNotification* and *Notify Other ABs* are about event notification, which is not so interesting for us since it is not special for the mobility part of Wireless CORBA.

Wireless network communication In order to provide reliable transmission over wireless connections, Wireless CORBA redefines the inter-ORB protocol to enhance reliability and performance of object communication in the mobile environment. Messages over the wireless link are transferred by the adaptation layer that guarantees reliability and ordered delivery of messages. There are some messages that are sent by either the Terminal Bridge or the Access Bridge to allocate a connection on the remote end of the tunnel, which include *OpenConnectionRequest, OpenConnectionReply, CloseConnectionRequest, CloseConnectionRequest*, etc. Since we are interested in the dynamic changes caused by the wireless

network connection, we will simplify the process of opening a connection between the bridges as an operation *connectBridge*, and the process of closing a connection as *disconnectBridge*. The intermediate steps of message change will not be modeled. Besides, the operations of building and deleting a physical wireless connection *connect*, *disconnect* will be still kept.

Context-awareness Transparency of the mobility mechanism to non-mobile Object Request Broker (ORB) is one of the primary design constraints. Wireless CORBA uses handover process to ensure that the invocation between clients and servers are kept consistent and the mobility is transparent to the applications. On another side, Wireless CORBA also provides location-based event notification service to the application, in order to support the location-based applications. The location of a terminal is defined as the actually attached access bridge. The terminal‘s location changes when it moves and attaches to other access bridge. As having mentioned before, we will model the handover process, but not the event notification part, since it is not special for the mobility part of Wireless CORBA.

Space definition Wireless CORBA has a structured space where mobile terminals are allowed to move inside the scope of areas covered by the access bridges, which is called *visited domain*. The visited domain is logically partitioned by access bridges. It is very helpful to model the relationship between the physical structured area and the logical visited domain. In addition, terminal domain and the home domain are also important. The movement of the objects need to be modeled too.

Dynamic change Wireless CORBA provides the fundamental mechanisms to support terminal mobility in CORBA, i.e., the terminal (acting as client or server) can change its location. Such system is very dynamical reconfigurable since objects can always change its location and therefore the relationship with other components. Besides, the interaction and configuration between objects are also very dynamic. This part is already covered in other previous parts, e.g., the wireless connection change, the movement of the objects, the handover process, etc. Therefore, we will not mention it additionally in (Table 6.4).

6.3 Conceptual style

After introducing the architectural commonalities and Wireless CORBA as a concrete example of the middleware for nomadic networks, we will explain now how to develop the architectural style for the middleware. As having explained in Section 4.2, we will separate these aspects into different abstract layers, in order to decrease complexity and enhance reusability of the model. More specifically, we separate the style into three abstract layers, i.e., the conceptual style, the platform-

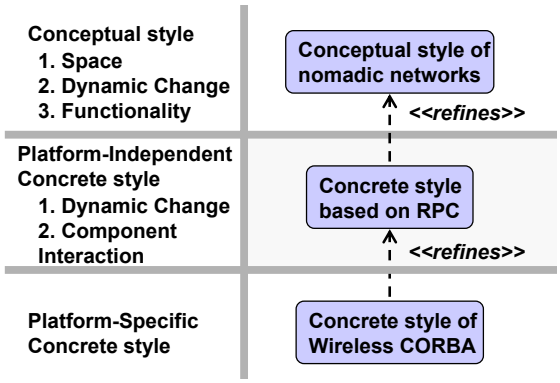

Figure 6.3: The architectural style of the middleware for nomadic networks in layered structure

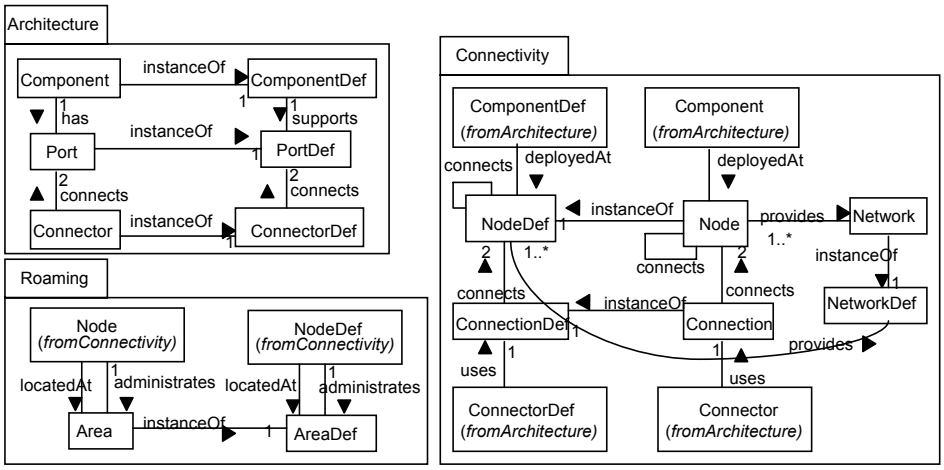

Figure 6.4: The type graph of the conceptual style

independent concrete style, and the platform-specific concrete style (Fig. 6.3), each of which captures different architectural aspects or views.

We will start with the conceptual style, which should specify the structured space of nomadic networks and the main functionality of the middleware, which is presented as the handover protocol that provides continuous connectivity and other services when components roam. The conceptual style also specifies a style of the mobile system for nomadic networks, or a mobility style supported by nomadic networks, which includes the allowed movement of the components, and the specific service provided by this kind of mobile system.

As having explained in Section 5.3, we will specify the style using the TGTS, where $G = \langle TG, C, R \rangle$. We will separate the style into two parts: static structural part and dynamic behavioral part, which will be introduced separately in the following sections.

6.3.1 Structural part

We use the type graph (TG) to define the static structural part of the style, which includes the important architectural components and elements, and the relationship between them. The type graph for the conceptual style is structured into three packages (Fig. 6.4) *Architecture*, *Connectivity* and *Roaming*. This allows us to separate different concerns, i.e., software architecture, distribution, and roaming, while at the same time retaining an integrated representation (see Section 5.3).

The structured space of nomadic networks is modeled as *AreaDef* in *Roaming* package. An area is defined by an administrative domain, like a cell managed by a GSM base station, or a Wireless LAN domain. The wireless connection is modeled as *Connection* in *Connectivity* package. *Connection* is a physical network link which delivers communication services to *Connectors* at the software level. The typing *ConnectionDef* means that we can distinguish, for example, between Ethernet, WLAN, or GSM-based connections.

The *Architecture* package defines the architectural view, containing both a definition of an architectural model (meta classes *ComponentDef, ConnectorDef, PortDef*) and an individual configuration (meta classes *Component, Connector*, and *Port*), related by the meta association *instanceOf*. *ComponentDef* class represents a component, which defines the primary computations of the application. *PortDef* represents the concept of port, which is the connecting point between two interacting components. *ConnectorDef* class represents a connector between two ports of two components. A connector represents the interaction between two components.

The *Connectivity* package presents the distributed view of the system in terms of the concepts *Node*,*Connection*, and *Network* paired as above with corresponding type-level concepts. A *Node* is a (real or virtual) machine, usually having a memory and a processing capability. Components are deployed on nodes. A *Connection* is a physical network link that delivers communication services to *Connectors* at the software level. A Connection at the physical level links two nodes. A Network is a physical network, which is provided by a node. This concept is used to model the case when the node acts as a connectivity provider (or an administrative node), for instance, the WLAN access point, or the GSM/GPRS base station. The typing means that we can distinguish, for example, between Ethernet, WLAN, or GSM-based connections, or between different kinds of machines like PCs, laptops, cell phones, etc. Besides, we specify an association *connects* between *Node*s, which is used to present the connection to a node locating on the fixed network side. The package uses the elements *Component, ComponentDef* from the Architecture package to be able to specify deployment using the *deployedAt* association.

The *Roaming* package defines the location and mobility of *Nodes* in terms of *Areas*, i.e., places where *Nodes* can be located. An area is defined by an administrative domain, like a cell managed by a GSM base station, or a Wireless LAN domain. Thus, there are different types of areas which may be overlapping. An area can have an administrative node that serves the connections in this Area. This node does not need to be located inside the same area. We do not separate mobile

Space	Funct. (1. Interoperability)	Funct. (2. Wireless Connection)	Funct . Component Interaction (3. Handover)	Component Interaction
moveIn moveOut moveTo	bind unbind	connect disconnect	handOver	No

Table 6.5: The rules of the conceptual style

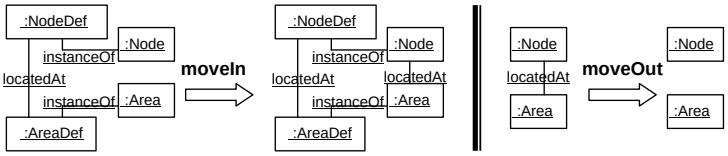

Figure 6.5: Rules: *moveIn*(left) and *moveOut*(right) of the conceptual style

from stationary hosts at this level. A node is mobile if it changes its location to a different area, a fact that is part of the dynamics of the model. This allows the flexibility of considering, e.g., a laptop that does not move as a stationary device.

6.3.2 Behavioral part

We use graph transformation rules (R) to define the dynamic behavioral part of the style, i.e., the movement of components, the handover process, etc. We define eight graph transformation rules (Table 6.5): *moveIn, moveOut, moveTo, connect, disconnect, bind, unbind* and *handOver*. Rules *moveIn, moveOut, moveTo* model the movement of components. Rules *connect, disconnect* model the connection and disconnection of wireless networks. Rules *bind, unbind* model the creation and deletion of a session between two components when they interact. Rule handOver models the handover process, which is the main functionality of the this type of middleware. Rules *moveOut*, *disconnect* and *unbind* are dual to *moveIn*, *connect* and *bind* respectively.

On left-hand side of Fig. 6.5 the *moveIn* rule is shown. According to its precondition, expressed by the pattern on the left-hand side, there should be a *Node* and an *Area* whose types *NodeType* and *AreaType* should be connected by a *locatedAt* link. That means the node is of a type that is supported by the area, like a cell phone in a GSM cell. In this case, the rule can be applied with the result of creating a new *locatedAt* link between the two instances. This is expressed in the post-condition of the rule shown on the right-hand side. The dual operation *moveOut*, specifying the deletion of a *locatedAt* link, is shown on right-hand side of Fig. 6.5.

The rules can be combined to define more complicated rules. For example, we can combine *moveIn* and *moveOut* to become a new rule *moveTo*(Fig. 6.6), which

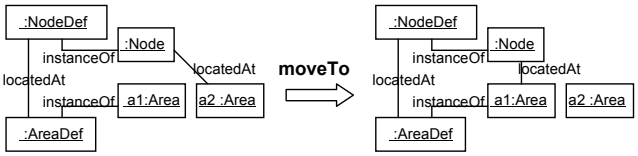

Figure 6.6: Rule *moveTo* of the conceptual style

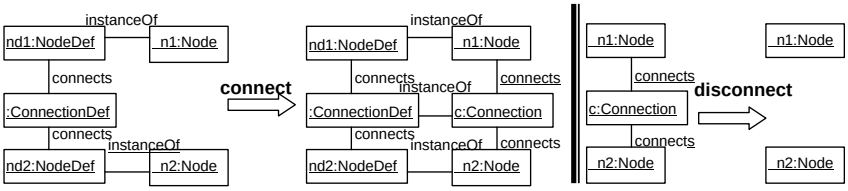

Figure 6.7: Rules: *connect*(left) and *disConnect*(right) of the conceptual style

represents a process of moving out from an old area, and then moving in to a new area.

Connecting is the action of building a network connection between two nodes. The precondition of the rule in Fig. 6.7 requires the existence of a corresponding connection type between the two nodes types. As postcondition, the desired connection c:Connection and its links are created. The dual rule *disconnect* is for deleting a connection.

The rule *handOver* (in Fig. 6.8) explains how to maintain connectivity between administrative domains. It requires that node *n1* is located in two areas served by two administrative nodes *n2* and *n3*. Connector *cr* uses connection *c1* between node *n1* and *n2*. The connection is replaced by *c2* of type *ConnectionDef* which, according to the types declaration, is permitted between nodes of type *NodeDef nd1* and *nd2*. The *uses* relation of the connector is transferred to the new connection. In reality, the handover process is rather complicated. The reason to activate it can be various too. In order to capture the handover from a more general point of view, we define the handover simply as the procedure of changing connectivity

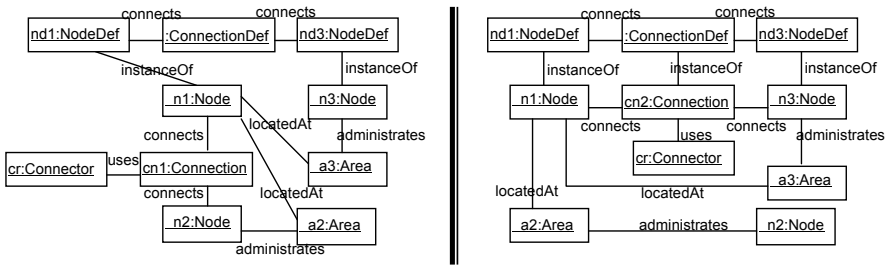

Figure 6.8: Rule: *handOver* of the conceptual style

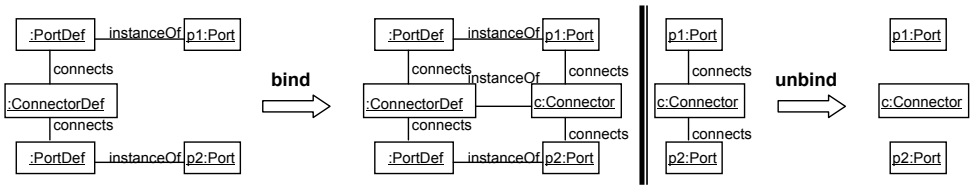

Figure 6.9: Rules: *bind* (left) and *unbind* (right) of the conceptual style

between administrative domains. The session layer connection is hence kept continuously. The reason that activates the handover process is implicit here. It can be quality degradation of the old connection, or the old connection is not available anymore. Such an abstraction allows us to capture the most cases of the handover.

The rule *bind* in (Fig. 6.9) is for building sessions between two components through the ports. The precondition of the rule requires the existence of a corresponding connector type between the two port types. As postcondition, the desired connector c:Connector and its links are created. The dual rule *unbind* is for deleting a session.

6.4 Platform-independent concrete style

After specifying the conceptual style in last section, we will refine it into the platform-independent concrete style to add more design-specific aspects. The component interaction pattern is integrated now into the core functionality specified in the conceptual style. In other words, it specifies the design-specific component interaction pattern adapted for the mobility style of the nomadic network. This includes the dynamic change of components involved in an interaction due to their migration or connectivity patterns. As a result, the dynamic changes specified in the conceptual style will be associated with component interaction in the platform-independent concrete style.

As having explained in Section 6.2, we will take the adapted RPC mobility model as the object for building the style. The same as the conceptual style, we specify the platform-independent concrete style using the TGTS. The style can be separated into two parts: the structural part and the behavioral part. We will further separate the behavioral part into two subparts: interaction diagrams and graph transformation rules, where the interaction diagrams define the synchronization between different rules.

6.4.1 Structural part

The specified type graph in the conceptual style is refined to support component interaction RPC model in the concrete style. The *Architecture* package in Fig. 6.10 adds more architectural elements that support interoperability, interaction and communication. Besides, a *Message* package is created to support message definitions

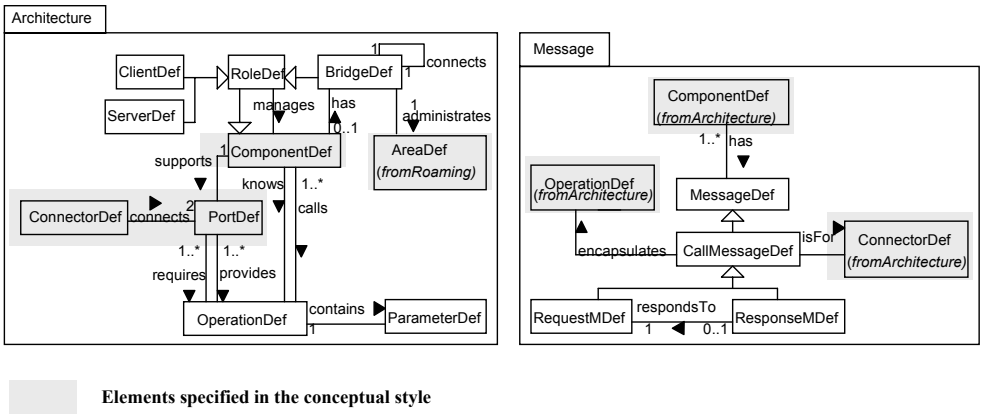

Figure 6.10: Type graph (partial) of the platform-independent concrete style

for calling a remote procedure. We put the message processing part in a separate package since it is important in the middleware, and it can be further refined and evolved to support more complicated message processing and types. The other two packages *Connectivity* and *Roaming* in the conceptual style remain unchanged and are not shown here. Fig. 6.10 only contains the Def part that defines the type meta classes like *ComponentDef*. The individual instance configuration is not included here. We suppose that every "Def" class has a corresponding *instanceOf* associated meta class.

In the specification, a component can play three Roles: *Bridge*, *Client* and *Server*, which are the main middleware construction components. *Client* and *Server* model the client and server stub of the middleware respectively. They will be responsible for the interaction between application components, for example, marshalling and unmarshalling the message for a remote procedure call. *Bridge* is responsible for connecting the mobile components with fixed components, and bridging the sessions at the logical level. The transmission of messages for a procedurral call and the hold of session connectivity will be pursued by bridges, i,e, the handOver is processed between bridges. There exists one bridge for every *Area*, and the bridge administrates the area. The bridge does not need to be located inside the same area.

Port requires or provides *Operations* that contain *Parameters*. The *Connector* in the conceptual style remain the same here. *Connector* defines one logical session between two component ports, and such a session will be refined as procedure calls at the concrete level. The procedure calls will be encapsulated inside messages. The *Message* package defines the *Message* that needs to be created and sent. The most important message is *CallMessage* that can be divided in *RequestMessage* and *ResponseMessage*.

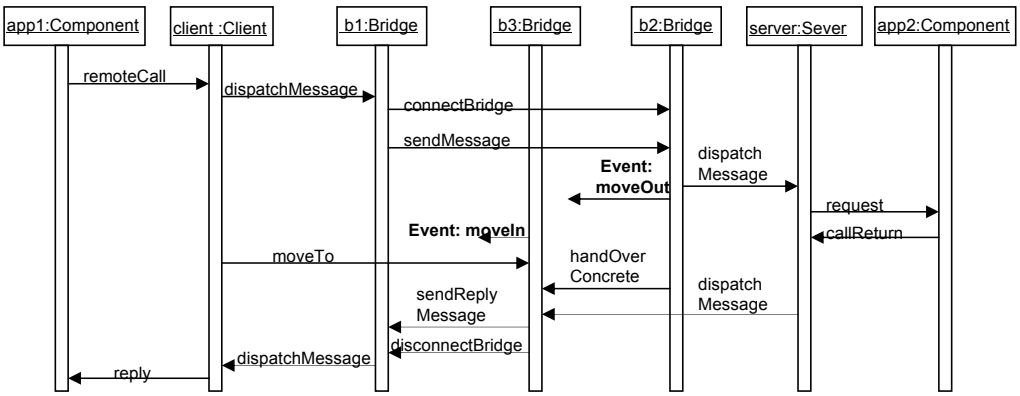

Figure 6.11: Sequence diagram for a remote invocation process (mobile client)

6.4.2 Behavioral part

The dynamic behavioral part models mainly the component interaction, which covers component communication, collaboration, and coordination. We use graph transformation rules to specify the component communication and collaboration, whilst the coordination and synchronization between different actions and components are specified using UML Interaction diagrams (see Section 5.3). We will explain at first the interaction diagram. The rules are explained afterwards.

6.4.2.1 Interaction diagrams

We use the UML Interaction diagrams to specify the sequence of operations for one remote procedure call. There are two cases of the calling process: the client is mobile (Fig. 6.11), or the server is mobile (Fig. 6.12).

We will explain at first the calling process when the client is mobile (Fig. 6.11). The interacting components here are mainly middleware construction components, for instance, *Client client*, *Server server*, *Bridge b1*, *b2*, *b3*. *Component app1 app2* are application components. *App1* and *client* reside on the same host *h1*. *Server* resides on the same host as *app2*. When *app1* wants to call a remote operation through *remoteCall*, *client* generates a request message and dispatches it to Bridge *b1* by calling *dispatchMessage*. At the same time, a session is created between the two ports of the component *app1* and *app2*. Afterwards, the *Bridge b1* calls the *connectBridge* operation to build a connection between the two bridges *b1* and *b2*, since Bridge *b2* administrates the wireless area where the client locates and hence being responsible for bridging the mobile components. The message is then sent between the Bridge *b1* and *b2* through *sendMessage*.

After receiving the message, *b2* sends it to the *server* that unmarshalls the message and *request*s a service (operation) directly from *app2*. *app2* calls the operation locally and generates results. *app2* calls then *callReturn* that sends the results to *app1*. The server generates reply message and sends it to bridge *b3*,

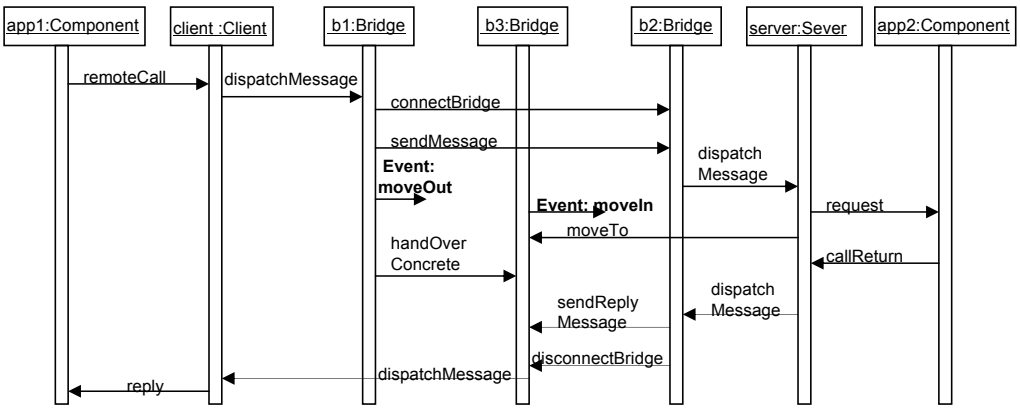

Figure 6.12: Sequence diagram for a remote invocation process (mobile server)

but not *b2* anymore. This is because client moves from the area administrated by *b2* to another area *a2*, which is administrated by *Bridge b3*. This movement is represented as an operation *moveTo* in the diagram. A handOver operation has been invoked between *b2* and *b3* to build a new connection between *b1* and *b3* for the session. The old connection between *b1* and *b2* is deleted afterwards. The bridge b3 now is responsible for transmitting the reply message between *b1* and *b3* through calling *sendReplyMessage*. The connection between the bridges is deleted by calling *disconnectBridge*. The b1 and b2 can be responsible for the issuing of the *moveOut* and *moveIn* events of the mobile client. *Bridge b1* dispatches the message to *client*. The client unmarshalls the message and calls *reply* of the Component *app1*. The related session is deleted and the whole remote calling process is finished.

The interaction in Fig. 6.12 is about the case when the server is mobile. The specification of the operations is the same as the diagram in Fig. 6.11. However, there are some differences regarding the configuration. In Fig. 6.11, the client is mobile, and it is bridged through b1, b2, b3 to the server. B2 models the old connected bridge in the old access area. B3 is the new connected bridge in the new access area after moving of the client. While in 6.12, the server is mobile, and it is bridged through b1, b2, b3 to the client. B1 is the old connected bridge in the old access area. B3 is the new connected bridge in the new access area after movement. Therefore, the handOver operation is pursued between b1 and b3 to build a new connection between *b2* and *b3* for the session. The old connection between *b1* and *b2* is deleted then.

6.4.2.2 Graph transformation rules

After introducing the specification of coordination and synchronization between different actions and components using UML Interaction diagrams, we will use graph transformation rules to specify the operations defined in the interaction dia-

Space	Funct . (1. interoperability)	Funct. (2. Wireless Connection)	Funct./ Component Interaction (3. Handover)	Component Interaction
moveIn moveOut moveTo	No	connect disconnect connectBridge disconnectBridge	handOverConcrete	remoteCall request callReturn reply dispatchMessage sendMessage sendReplyMessage

Table 6.6: The rules of the platform-independent concrete style

grams. The dynamic change happening during the component communication and collaboration processes will be specified.

We specify fifteen graph transformation rules (Table 6.6) for the style (R): *moveIn, moveOut, moveTo connect, disconnect, connectBridge, disconnectBridge, remoteCall, request, callReturn, reply, handOverConcrete, dispatchMessage, sendMessage, sendReplyMessage*. The rules defined in the conceptual style are directly used or adapted here. Rules *moveIn, moveOut, moveTo* model the movement of components, and they are taken from the conceptual style without change. However, they are presented as events in the interaction diagrams, since the sequence diagram represents the interaction of objects that have states.

Rules *connect, disconnect, connectBridge, disconnectBridge* model the connection and disconnection of wireless networks. Rules *connect, disconnect* remain the same as in the conceptual style. Rules *connectBridge, disconnectBridge* are adapted from the *connect, disconnect* respectively. They are used to build and delete a logical connection between two bridges based on the physical network connection.

Rules *remoteCall, request, callReturn, reply* model the procedure of a remote procedure call. They are refined from the conceptual style rules *bind, unbind* that model the creation and deletion of a session respectively. Rule *dispatchMessage* is for dispatching messages. Rules *sendMessage* and *sendReplyMessage* are for sending the message between two bridges.

In order to simplify the figures and help understanding, we will not put the type definition "*Def*" and the association *instanceOf* in the following rules. We suppose that every new created class and association has by default corresponding associated type "Def" class and association.

The rule *remoteCall* (Fig. 6.13) models the processing of a remote calling request issued from an application component. The left side of the rule defines the precondition to execute the operation, and the right side of the rule defines the postcondition after executing the operation. It requires that there exist Client stub *cl* and Server stub *s* for the application component *app1* and *app2* respectively. The app1 that will request the operation (through the port *p1*) *knows* the operation

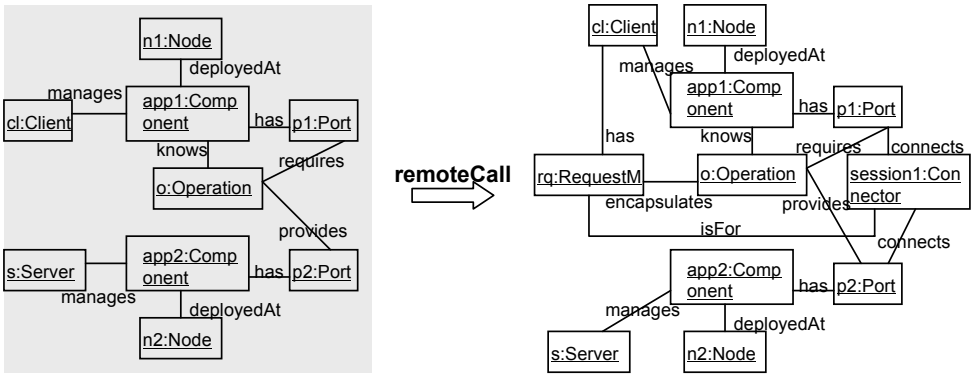

Figure 6.13: Rule for *remoteCall* operation

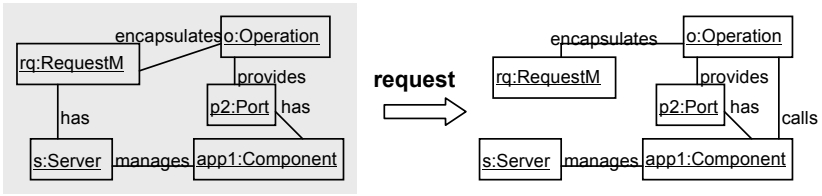

Figure 6.14: Rule for *request* operation

o, which is provided through the port *p2* of component *app2*. *App1* and *app2* are deployed at different node *n1* and *n2* respectively.

The application components *app1* and *app2* do not distinguish a remote call from a local call. The middleware components client and server are responsible for managing the calling process. In this rule, it is judged as a remote call since the components are located on different nodes. It can happen that the components reside on the same host. The client will forward it as a local call then. We are not interested in this case and do not create rules for it.

After the execution of the operation, the request message *RequeseM rq* is created by the *Client cl*. *RequeseM rq* encapsulates the parameters of the required operation *o*. At the same time, *Connector session1* is also created. It is built between the two ports *p1* and *p2*, and will exist till the finish of the calling process, i.e., when the reply is received by the caller *app1*. *RequeseM rq* is used for this session and a *isFor* association is also created.

The rule *request* (Fig. 6.14) models the processing of a calling request message that is received on the server side. The server will analyze the message and issue an operation call request to the application component. The request message *rq:RequestM* still exists in the post-condition of the rule because it encapsulates the information of the caller.

The rule *callReturn* (Fig. 6.15) models the call return process of the application

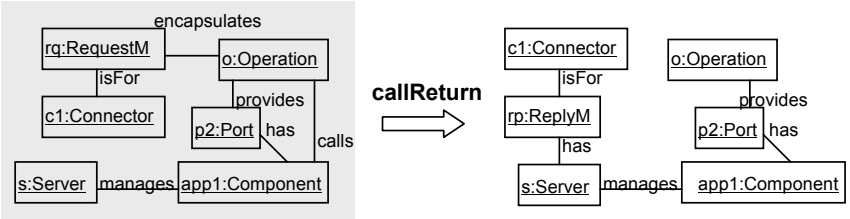

Figure 6.15: Rule for *callReturn* operation

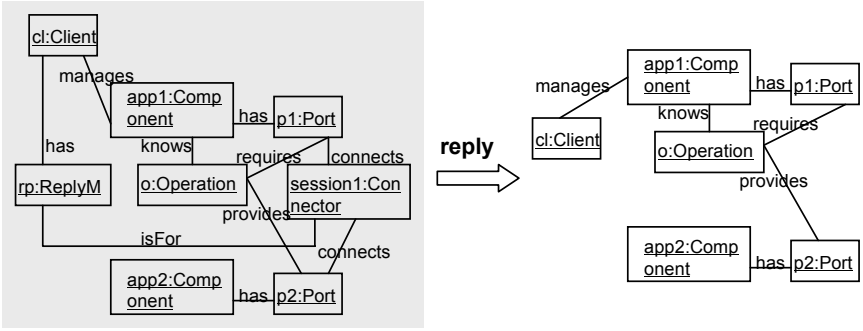

Figure 6.16: Rule for *reply* operation

component. The information encapsulated in the request message *rq:RequestM* is used to create a reply message *rp:ReplyM*. Afterwards, the request message *rq:RequestM* is deleted in the post-condition. The session *c1:Connector* for the calling process is associated with the reply message, but not with the request message anymore.

The rule *reply* (Fig. 6.16) models the processing of the calling reply message that is received on the client side. The client will analyze the message and then delete the corresponding session created for the calling process. As the post condition of the rule, both of the session and the reply message will be deleted. The whole calling process is finished then.

The rule *dispatchMessage* (Fig. 6.17) is for dispatching messages between two roles. It requires that the role *r1* has the message *m1*. The message *m1* is then held

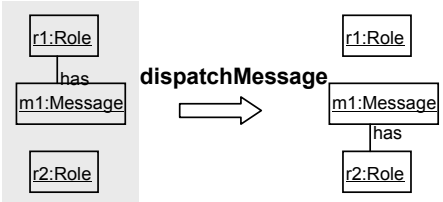

Figure 6.17: Rule for *dispatchMessage* operation

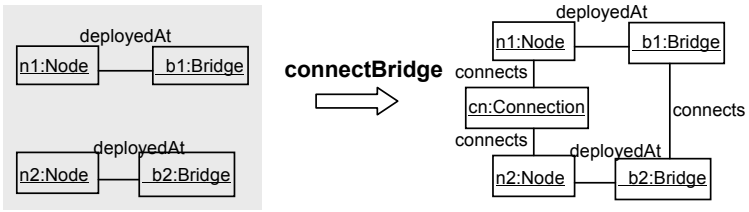

Figure 6.18: Rule for *connectBridge* operation

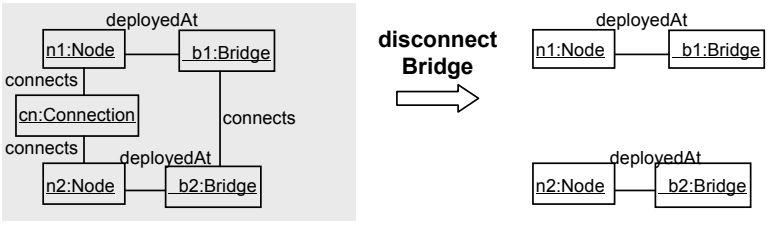

Figure 6.19: Rule for *disconnectBridge* operation

by *r2* after the execution of the rule. We use it when there is no need to build a wireless network connection for directing a message, for instance, when one role is client or server, and another role is bridge. In this case, the roles locate on the same host, or they use stable fixed network connection.

The rule *connectBridge* (Fig. 6.18) is for building not only a network connection but also a logical connection between two bridges. The dual rule *disconnectBridge* (Fig. 6.19) is for deleting a connection between bridges.

The rules *sendMessage* (Fig. 6.20) and *sendReplyMessage* (Fig. 6.21) model how to send messages between two bridges. The precondition of the rule requires that there exists a connection between the nodes where the bridges are deployed. As the only difference between the two rules, *sendReplyMessage* has already a *Connection cn* that is used by the *Connector c1*, which was built for sending the request message, and the connector will use the old connection for sending the reply message.

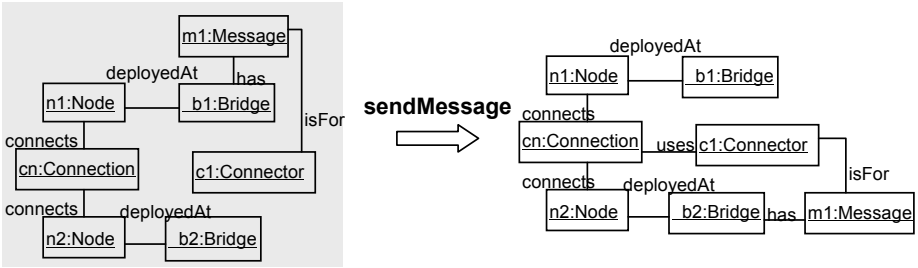

Figure 6.20: Rule for *sendMessage* operation

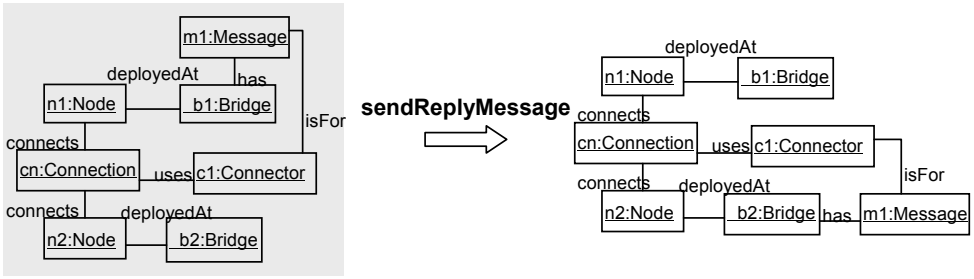

Figure 6.21: Rule for *sendReplyMessage* operation

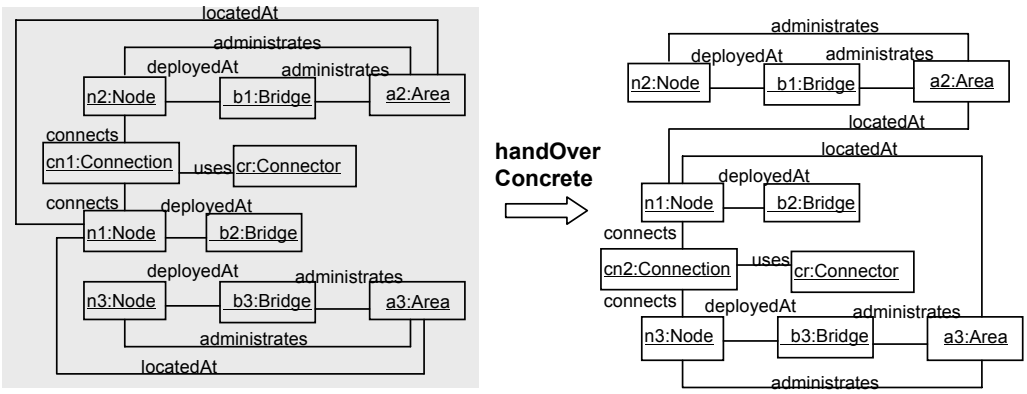

Figure 6.22: Rule for *handOverConcrete* operation

The *handOver* rule specified in the conceptual style has been changed into the rule *handOverConcrete* (Fig. 6.22). The definition of the handOver in the concrete style is more at the session level. The purpose of the handOver here is to provide continuous connectivity service for the session between two bridges that deployed on physical nodes. The main difference is that we add the concept of *bridge* compared to the more abstract *handOver* rule. As precondition, the session *Connector: c1* uses the *Connection: cn1*, which is built between *Node n2* and *Node n1*. As postcondition, the *Connection: cn1* is deleted. A new *Connection: cn2* is created between *Node n2* and *Node n3*. The *Connector: c1* uses the *Connection: cn2* now. Therefore the connectivity of the session is retained.

6.5 Platform-specific concrete style: Wireless CORBA

The platform-independent concrete style specifies the RPC model adapted for nomadic networks. We will further refine it into a platform-specific concrete style that adds more design-specific aspects. We will take Wireless CORBA to illustrate how to define the style for a concrete middleware. The same as in the platform-independent concrete style, we separate the style specification into three parts: the structural part, the behavioral part with interaction diagrams, and the behavioral part with graph transformation rules.

When we specify the style for Wireless CORBA using the TGTS based modeling approach, we also provide formal semantics to the IDLs specification at the same time. As introduced in Section 6.2, Wireless CORBA comprises IDL files, whose semantics is informally specified using natural language. This is a potential cause of conflicts and ambiguities. Especially when the components have complicated behavior or many components are involved to perform a task in dynamic mobile settings. Using our TGTS based modeling approach, we can specify formally the most important aspect of the component interaction in mobile settings, i.e., the dynamic change of components involved in interactions, the collaboration and synchronization definition between different objects and operations, etc. We will also explain this part in a subsection.

6.5.1 Structural part

As introduced in Section 6.2, Wireless CORBA comprises three IDL files (see Appendix A), which are *MobileTerminal.idl*, *MobilityEventNotification.idl* and *GIOP.idl* (see Appendix A). Each of the Wireless CORBA IDL files specifies one specific part of the system. *MobileTerminal.idl* specifies the elements to construct the architecture of Wireless CORBA. *MobilityEventNotification.idl* specifies mobility service booking and notification. *GTP.idl* specifies the GTP message classification and channels for transmitting the GIOP messages. Besides, Wireless CORBA includes the basic two IDL files of CORBA since it is developed based on CORBA, which are *orb.idl* and *IOP.idl*. *orb.idl* specifies the object invocation related ele-

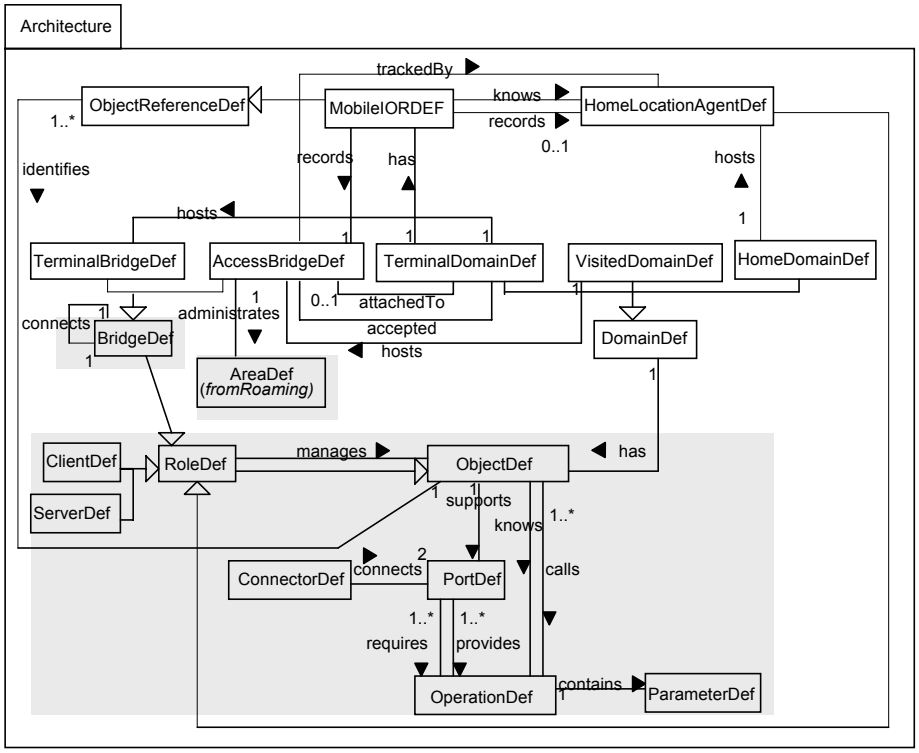

Figure 6.23: *Architecture* package for Wireless CORBA

ments. *IOP.idl* specifies protocols for interoperability between different objects.

As having identified in Section 6.2, we will not go into details of CORBA, but focus on the aspects that are specific for Wireless CORBA, e.g., how to provide continuous invocation service when mobile terminals move between points of connectivity. Thus, the basic object invocation related part will be kept nearly the same as in the more abstract style, which is specified in the *Architecture* package that is now extended to support Wireless CORBA architectural elements. Correspondingly, the structural part of *MobileTerminal.idl* is specified in the *Architecture* package, which includes the main elements, the meaning of attributes and the relationship between the elements. The structural part of *GTP.idl* is specified in the *GTP* package, which is refined from the *Message* package in the platform-independent concrete style. The main structural elements of *IOP.idl* are also specified in the *GTP* package. We will not model the *MobilityEventNotification.idl* since the event notification part is not special for the mobility part of Wireless CORBA, as identified in Section 2.5.

The *Architecture* package in Fig. 6.23 contains classes that form the core of the Wireless CORBA architecture. The *Architecture* package adds more design-specific elements compared to the one in the platform-independent concrete style. The *ComponentDef* is refined into *ObjectDef*. The invocation related elements like *PortDef, OperationDef, ConnectorDef, ParameterDef, ClientDef* and *Serveref* remain the same. *VisitedDomainDef* represents the fixed network side. The *Visited Domain* has several *Access Bridges* that represent the access point. In Wireless CORBA specification, the mobility of objects is defined logically by the association *attachedTo* between *TerminalDomainDef* and *AccessBridgeDef*. When an object moves to the area administrated by the access bridge, it will attach to the bridge. Although the physical mobility space *AreaDef* is not specified in Wireless CORBA, we still keep it in the package in order to present the model clearly. The *BridgeDef* is designed as an object role here.

A *Domain* contains different *Objects*. Each object can play the role of a *HomeLocationAgent, TerminalBridge* or *AccessBridge*, which make these objects different from others since they extend the functionality of the Object Requester Broker(ORB) [110] through redirecting the invocation between the client and server over wireless links. In addition, the ORB provides invocation redirecting and message encapsulation for these objects. A *TerminalDomain* hosts *TerminalBridges*, and a *VisitedDomain* hosts *AccessBridges*. A *TerminalDomain* can change its *attachedTo* association to different access bridges when the terminal moves to different locations. The *Accepted* association is used in the handoff process to indicate that the terminal bridge is accepted by the access bridge, but not responsible for the access bridge yet. The *AccessBridge* is *trackedBy* the *HomeLocationAgent* to record the currently attached access bridge.

The *GTP* package (Fig. 6.24) specifies the relative elements related to the GTP processing. In Wireless CORBA, the GTP is used to encapsulate and transmit the GIOP messages over wireless links. GIOP specifies most of the protocol details that are necessary for clients and servers to communicate. It specifies a set of GIOP

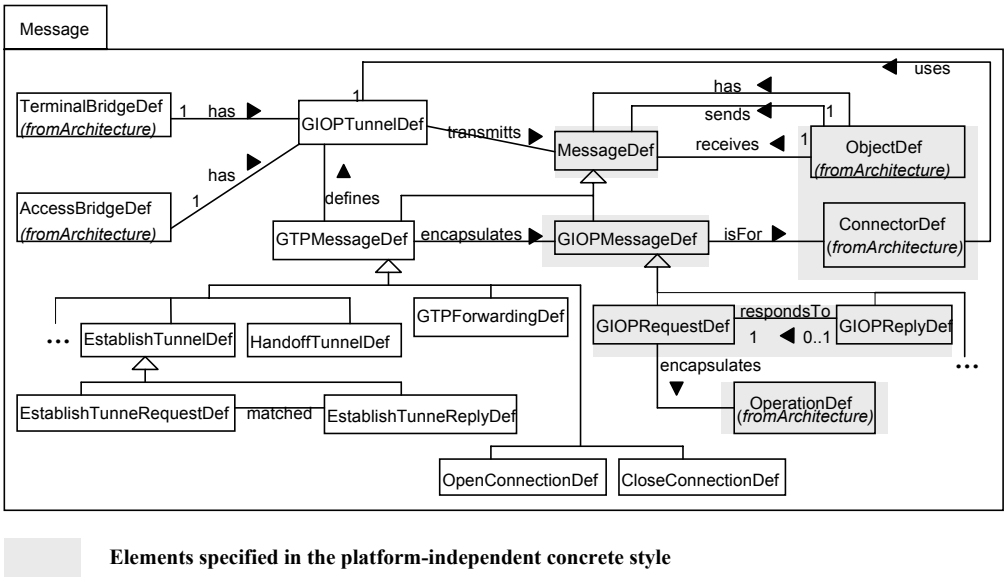

Figure 6.24: *GTP* Package for wireless CORBA

messages that include the request and reply message for invocations. A *GIOPTunnel* is established between terminal bridge and access bridge in order to transmit the GTP messages. GTP messages can be grouped into four categories [79]: *tunnel management* to establish, release, and handoff a tunnel, *GIOP connection* use to open and close a GIOP connection, *GTP forwarding*and *miscellaneous*. GTP messages are mostly Request-Reply pairs, e.g., *EstablishTunnel* has request and reply messages. In Wireless CORBA, the messages for calling are encapsulated as GIOP (General Inter-ORB Protocol) messages. Therefore, the invocation related messages *CallMessageDef, RequestMDef, ResponseMDef* in the more abstract *Message* package are refined to *GIOPMessageDef, GIOPRequestDef, GIOPReplyDef* respectively.

Besides these two packages, the *Roaming* package is kept the same, which models physical mobility of the mobile terminals. The relationship between the physical structured space and the visited domain is already given in the *Architecture* package. The *Connectivity* package is also kept the same.

In order to keep consistency with other two styles, we classify the elements according to their purpose. Therefore some elements are duplicate in different packages.

6.5.2 Behavioral part

Again, we separate the behavioral specification into the interaction diagrams and the graph transformation rules.

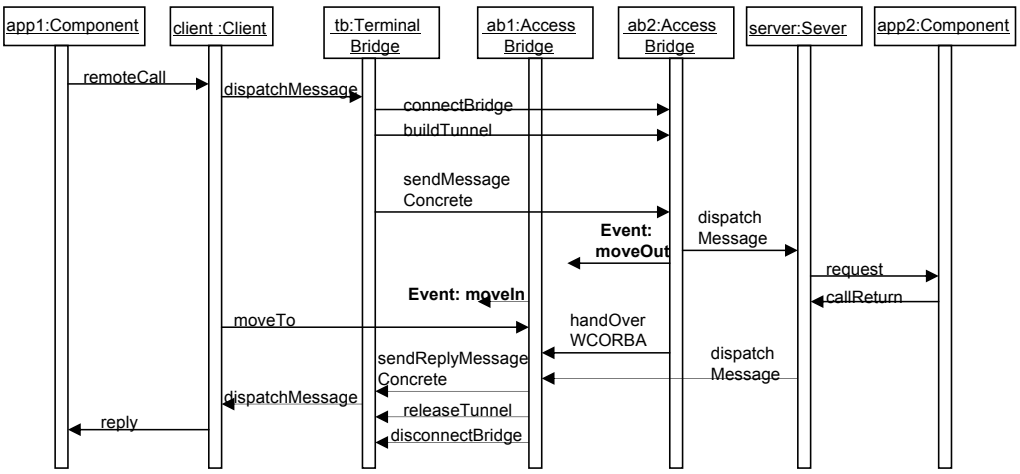

Figure 6.25: Sequence diagram for a remote invocation process (mobile client)

6.5.2.1 Interaction diagrams

We have used two UML Interaction diagrams in the platform-independent concrete style to specify the sequence of operations for remote procedure calls. There are two cases of the calling process: the client is mobile (Fig. 6.11), or the server is mobile (Fig. 6.12).

Wireless CORBA specifies mainly the interaction and communication between the access bridge, terminal bridge and home location agent. The other invocation related parts are specified in CORBA. Since we will not go into details of CORBA, the interaction between application component, client and server are kept the same as in the platform-independent style. The rules for basic invocation, i.e., *remoteCall*, *request*, *callReturn*, *reply*, *dispatchMessage*, *connectBridge* remain nearly the same. The other interactions between bridges have been changed. The rule *sendMessage* is refined to two rules: *sendMessageConcrete* and *buildTunnel*. Similarly, the rule *sendReplyMessage* is refined to *sendMessageReplyConcrete* and *deleteTunnel*.

The sequence diagram of (Fig. 6.11) in the platform-independent concrete style is refined to the sequence diagram in Wireless CORBA (Fig. 6.25). The sequence diagram in Fig. 6.12 has the same adaption of the rules.

The rule *handOverConcrete* in the platform-independent concrete style is refined to *handOverWCORBA* in the sequence diagram of Fig. 6.25, which is further refined to a sequence of rules (Fig. 6.26). There exist three different handover scenarios: terminal-initiated handOver, network initiated handOver and access recovery mechanism. The network initiated handOver starts when an external application invokes the start-handoff operation in the access bridge currently serving the terminal. The terminal initiated handover procedure requires that the terminal can establish connectivity to the new access bridge before releasing the connectivity to

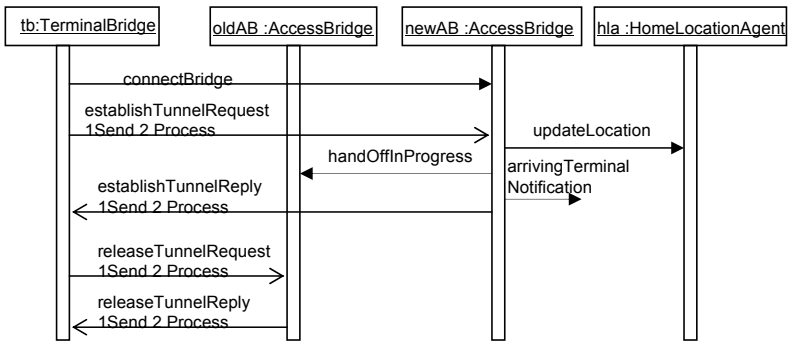

Figure 6.26: Sequence diagram for the terminal-initiated handOver scenario

the old access bridge. If this cannot be done, then the terminal initiated handover must be done using the access recovery mechanism. The terminal bridge closes connectivity to the old access bridge and then carries out the access recovery to the new access bridge.

The rules defined for the three scenarios do not have so much difference. We will take the terminal-initiated handOver scenario (see Fig. 6.2, in Section 6.2) as an example. The handOver scenario in Fig. 6.2 is modeled as a sequence diagram (Fig. 6.26). Correspondingly, the *handOverWCORBA* rule is refined into a sequence of rules as shown in Fig. 6.26. There are four object instances involved in the sequence diagram (Fig. 6.26) for a handover process. The *tb:TerminalBridge*, *oldAB: AccessBridge*, *newAB: AccessBridge*, and *hla: HomeLocationAgent*. The *tb:TerminalBridge* needs to establish transport connectivity to the new access bridge *newAB* before the handover starts. Afterwards, the terminal bridge needs to build a GIOP tunnel with the new access bridge. It sends a *EstablishTunnelRequest* message to the new access bridge and waits for a *EstablishTunnelReply* message from the new access bridge. After creating a GIOP tunnel with the terminal bridge, the new access bridge call the *updateLocation* to update the location of the terminal bridge. It also notifies the event of the arrivingTerminal to other objects. The old GIOP tunnel between the terminal bridge and the old access bridge will be deleted then. This is done through the GTP messages *releaseTunnelRequest* and *releaseTunnelReply*.

6.5.2.2 Graph transformation rules

Since we will not go into details of CORBA, the rules for basic invocation, i.e., *remoteCall, request, callReturn, reply, dispatchMessage, connectBridge* remain nearly the same as in the platform-independent concrete style, except that the instance of request and reply message *RequestMDef* and *ReplyMDef* will be changed to *GIOPRequestDef* and *GIOPReplyDef* respectively. The rule *sendMessage* is refined into two rules *buildTunnel* and *sendMessageConcrete*. The rule *sendReply-*

Space	Funct. (1. interoperability)	Funct. (2. Wireless Connection)	Funct. / Component Interaction (3. Handover)	Component Interaction
moveIn moveOut moveTo	No	connect disconnect connectBridge disconnectBridge	handOverWCORBA sendEstablishTunnelRequest processEstablishTunnelRequest send EstablishTunnelReply processEstablishTunnelReply sendReleaseTunnelRequest processReleaseTunnelRequest sendReleaseTunnelReply processReleaseTunnelReply updateLocation	remotecall request callReturn reply dispatchMessage buildTunnel releaseTunnel sendMessageConcrete sendReplyMessageConcrete

Table 6.7: The rules of the platform-specific concrete style

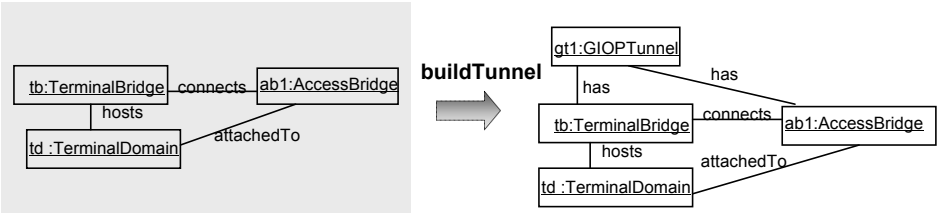

Figure 6.27: Rule for *buildTunnel*

Message is refined into *sendMessageReplyConcrete* and *releaseTunnel*. The rule *handOverConcrete* is changed to *handOverWCORBA*, which models a simplified version of the handover process in Wireless CORBA and can be further refined into a sequence of rules. The rules are shown in (Table 6.7).

The rule *buildTunnel* (Fig. 6.27) is for building a tunnel between a current administrative access bridge and a terminal bridge. This rule simplifies the process of building tunnels, which requires processing of a set of request-reply messages in the Wireless CORBA specification. Rule *releaseTunnel* is the dual rule that deletes the GIOP tunnel between the two bridges.

The rule *sendMessageConcrete* (Fig. 6.28) models the process of sending a GIOP message from a terminal bridge to an access bridge. The *m1:GIOPMessage* is used for a session *:Connector*. It uses the GIOP tunnel between the terminal bridge and the access bridge. The *cn:Connection* is used to by the connector. The message is held on the access bridge side after processing the rule. The similar rule *sendMessageReplyConcrete* (Fig. 6.29) is for sending a reply GIOP message from the access bridge to the terminal bridge.

The rule *handOverWCORBA* (Fig. 6.30) models a simplified version of the handover process of Wireless CORBA. Before execution of the rule, the *c1:Connector* uses *gt1:GIOPTunnel* between the old access bridge and terminal bridge. The physical connection *cn1:Connection* is also used. After execution of the rule, the

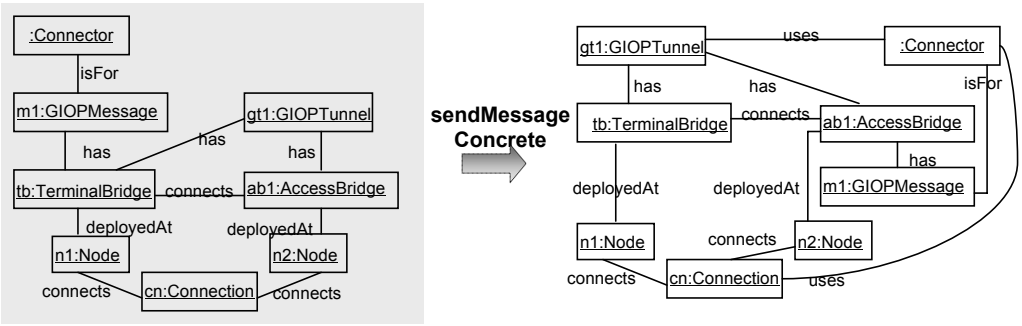

Figure 6.28: Rule for *sendMessageConcrete*

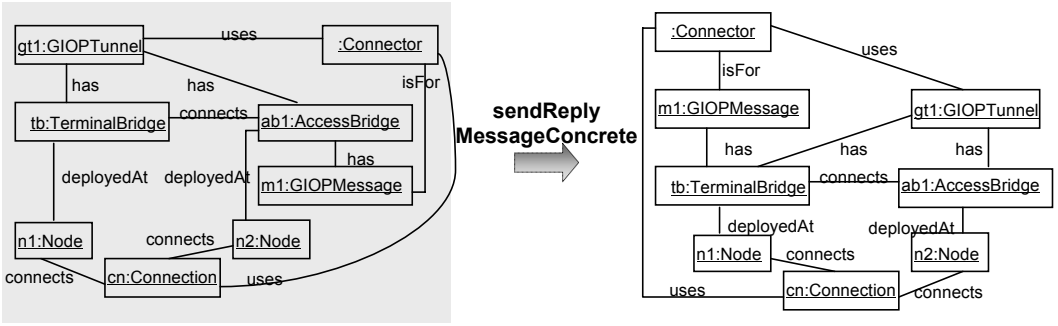

Figure 6.29: Rule for *sendMessageReplyConcrete*

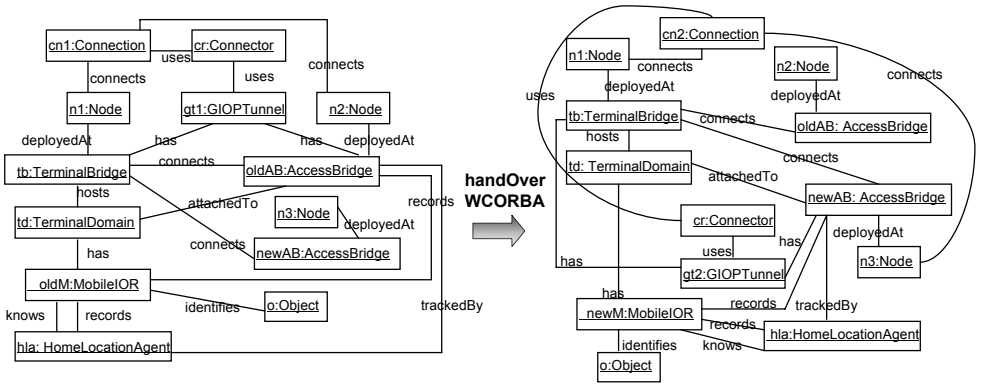

Figure 6.30: Rule for *handOverWCORBA*

connector uses a new tunnel, *gt2:GIOPTunnel*, that is built between the terminal bridge and the new access bridge. The physical connection is also updated to a new one, *cn1:Connection*. A new MobileIOR is created, which records the new access bridge. At the same time, the old MobileIOR is deleted. The Home Location Agent tracks then the new access bridge, but not the old access bridge.

The rule *handOverWCORBA* is realized as a sequence of GTP messages and operations (see Fig. 6.2, in Section 6.2). The GTP messages are used to communicate between a terminal bridge and a access bridge over wireless links. The GTP messages are mostly Request-Reply pairs, which have asynchronous semantics. The operations are normal CORBA operations that can be distinguished as synchronous, asynchronous and one-way invocations [74, 146]. The operations are specified for the objects that do not communicate over wireless links, such as the communication between access bridges, and the communication between access bridges and a home location agent.

We model the GTP messages and operations using graph transformation rules. We use only one rule to specify the operation, since we are only interested in the global pre- and post-condition of each operation. The intermediate procedures of how to generate request and reply message nodes for the invocation are not modeled. The difference between different types of invocations, like synchronous, asynchronous and one-way invocations, are not discussed in this case. Instead, for processing a GTP message, we use two rules expressing the asynchronous request and reply semantics of the message. Therefore, the detailed handover process specified in Wireless CORBA using message charts (see Fig. 6.2) is modeled using interaction diagrams and rules in the style (in Fig. 6.26), which will be explained in the following text.

In Fig. 6.31, the graph transformation rule for operation *updateLocation* is shown, which is called by the Access Bridge to update the location of a mobile terminal after handoff. According to its pre-condition, expressed by the left-hand side object graph, an operation call updateLocation is made by the client *ac2* to the

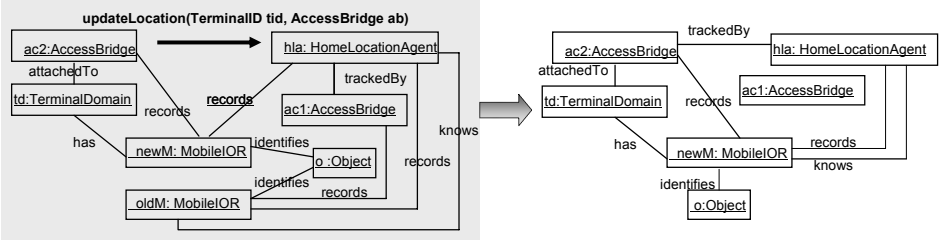

Figure 6.31: Rule for operation *updateLocation*

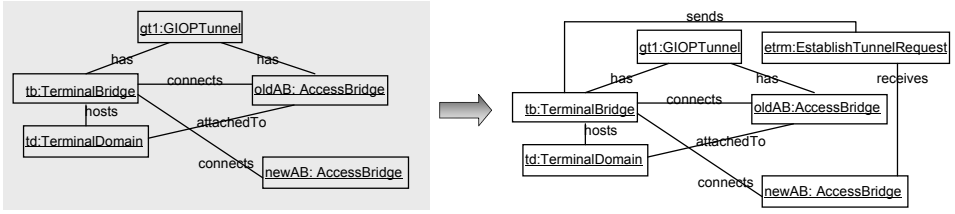

Figure 6.32: Rule for *sendEstablishTunnelRequest*

server *hla*. The Terminal Domain *td* has moved from the old AccessBridge *ac1* to the new AccessBridge *ac2*, which is expressed by the *attachedTo* association between *ac2* and *td*.

The TerminalDomain *td* has a MobileIOR *newM* to record the current attachedTo Access Bridge and Home Location Agent, while MobileIOR *oldM* still records the old AccessBridge and Home Location Agent. Both *newM* and *oldM* identify the same Object *o*, while the Agent *hla* only knows *oldM*, and *ac1* is trackedby *hla*. The new Access Bridge *ac2* needs to inform the Home Location Agent about the current access bridge association in such case. As the effect of the operation (see right side graph), the node *oldM* and the related associations will be deleted, as well as the *trackedBy* association between *ac1* and *hla* will be deleted. A new association *knows* will be created between *hla* and newM, and trackedBy is also created between *hla* and *ac2*. Further, the updateLocation node and the related associations will be removed.

The GTP message *EstablishRequest* is used when the terminal bridge wants to build a GIOP tunnel with the access bridge. It is divided as two steps: sendEstablishTunnelRequest (Fig. 6.32), and processEstablishTunnelRequest (Fig. 6.33). Fig. 6.32 is about the pre- and post-condition for sending a message. As precondition, it requires that the terminal bridge can connect physically to the access bridge. As postcondition after sending the message, a node for message *EstablishTunnelRequest* is created with the associations *sent* from *TerminalBridge: tb* to the receiver side *AccessBridge: newAB* expressed by the *receives* association.

Fig. 6.33 is about the result of processing the message. The post condition of the rule shows the result of processing the message. The *receives* association is

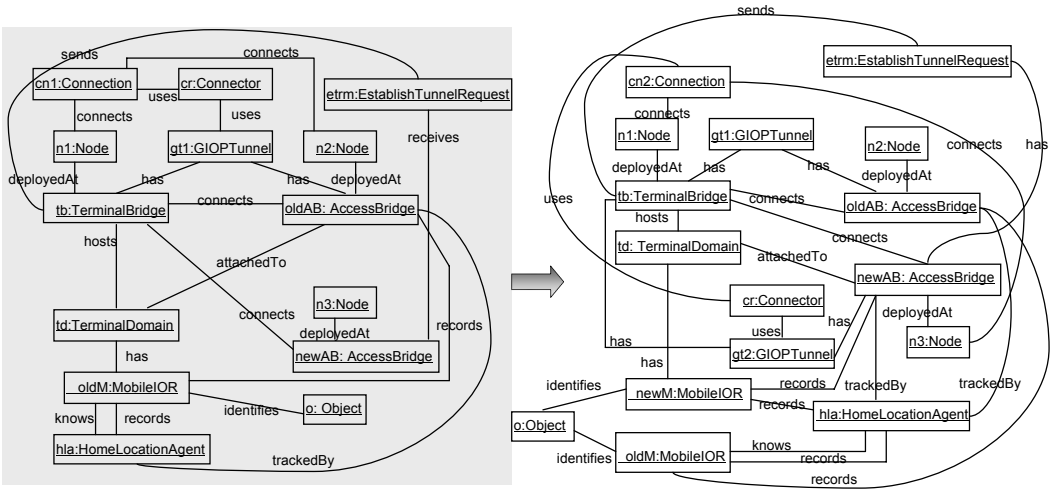

Figure 6.33: Rule for *processEstablishTunnelRequest*

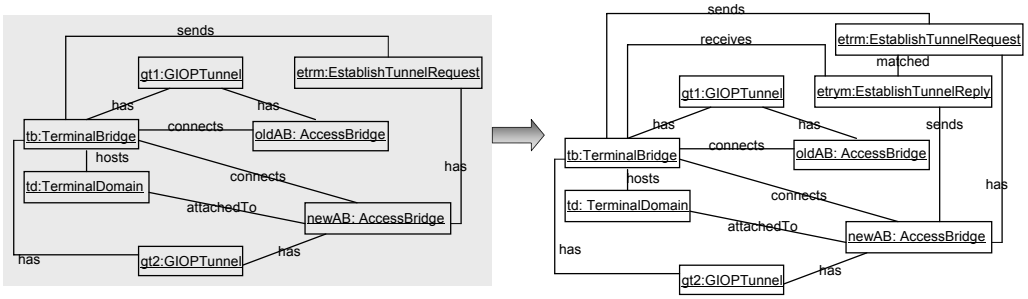

Figure 6.34: Rule for *sendEstablishTunnelReply*

deleted, and the *has* association is created between the message *etrm* and the new access bridge *newAB*, that means, the request message will be kept till the processing of a matched reply message. The terminal domain *td* changes its *attachedto* association from the old access bridge *oldAB* to the new one *newAB*. *GIOPTunnel gt2* is created between the terminal bridge *tb* and *newAB*. The *td* has a new MobileIOR *newM* to identify the *object: o*, the old MobileIOR *oldM* still exists. Besides, the connection *cn1:Connection* between *n1:Node* and *n2:Node* is deleted. A new connection *cn2:Connection* between *n1:Node* and *n3:Node* is created, which is used by the connector *cr:Connector*.

The access bridge sends a GTP message *EstablishTunnelReply* to the terminal bridge when the processing of the request message is finished. It is modeled as two steps: *sendEstablishTunnelReply* (Fig. 6.34), and *processEstablishTunnelReply* (Fig. 6.35). Fig. 6.34 is about the pre- and post-condition for sending the request message. The new access bridge has a *EstablishTunnelRequest* message, which is sent from the terminal bridge. As the post condition, the new access bridge

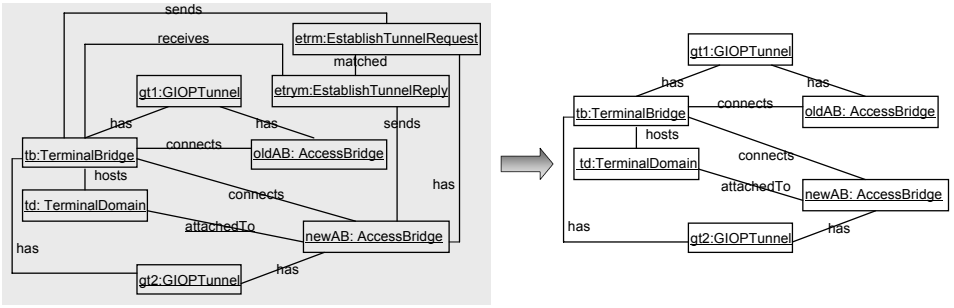

Figure 6.35: Rule for *processEstablishTunnelReply*

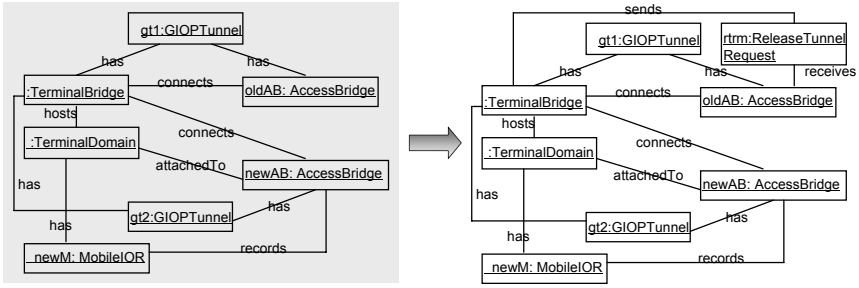

Figure 6.36: Rule for *sendReleaseTunnelRequest*

creates a *EstablishTunnelReply* message matched to the request message. The post condition of the rule *processEstablishTunnelReply* (Fig. 6.35) shows the result of processing the message. Both of the request and reply messages of *EstablishTunnel* are deleted after processing the message.

The terminal bridge sends a GTP message *ReleaseTunnelRequest* to an access bridge when the GIOP tunnel is not needed anymore. It is modeled as two steps: *sendReleaseTunnelRequest* (Fig. 6.36), and *processReleaseTunnelRequest* (Fig. 6.37). Fig. 6.36 is about the pre- and post-condition for sending the request message. As precondition, there exist two GIOP tunnels, where one is between the new access bridge and the terminal bridge, another one is between the old access bridge and the terminal bridge. A node of the tunnel release request message is created as the post condition. As the postcondition of the rule *processReleaseTunnelRequest* (Fig. 6.37), the GIOP tunnel between the old access bridge and the terminal bridge is deleted. The message is kept on the terminal bridge side.

The access bridge sends a GTP message *ReleaseTunnelReply* to the terminal bridge when the processing of the request message is finished. It is modeled as two steps: *sendReleaseTunnelReply* (Fig. 6.38), and *processReleaseTunnelReply* (Fig. 6.39). Fig. 6.38 is about the pre- and post-condition for sending the request message. The new access bridge has a *rtrm: ReleaseTunnelRequest* message, which is sent from the terminal bridge. As the post condition, the new access bridge

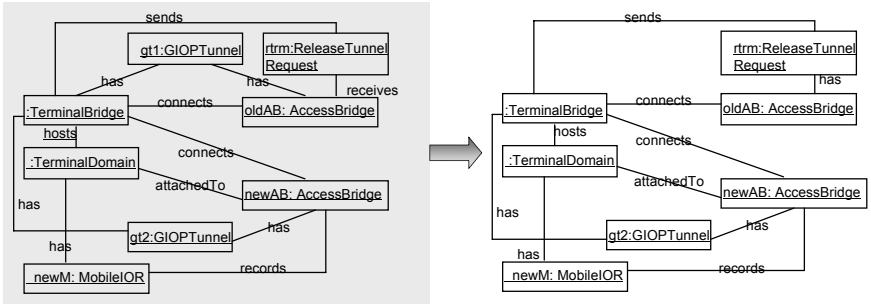

Figure 6.37: Rule for *processReleaseTunnelRequest*

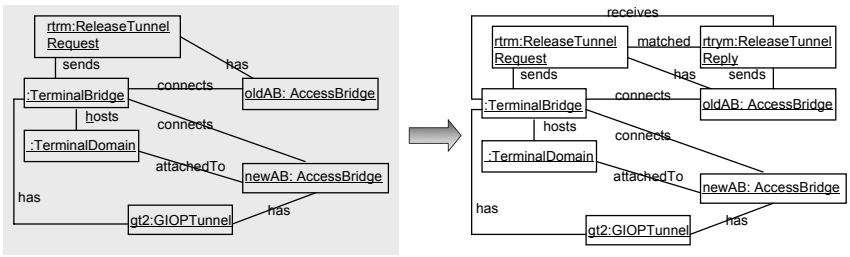

Figure 6.38: Rule for *sendReleaseTunnelReply*

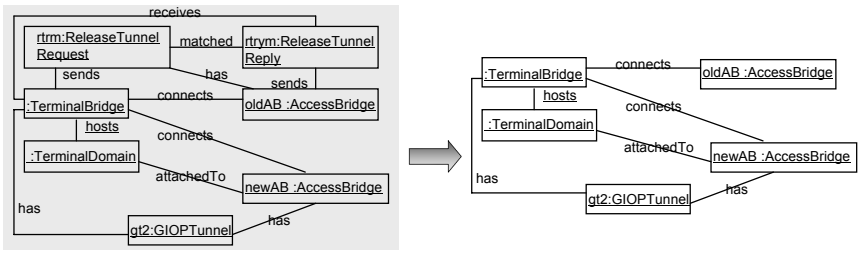

Figure 6.39: Rule for *processReleaseTunnelReply*

creates a *rtrym: ReleaseTunnelReply* message matched to the request message. The post condition of the rule *processReleaseTunnelReply* (Fig. 6.39) shows the result of processing the message. Both the request and reply messages of *ReleaseTunnel* are deleted after the message processing.

6.5.3 IDL semantics specification

When we specify the style for Wireless CORBA using the TGTS, we also provide formal semantics to the IDLs specification at the same time. The semantics specification of an IDL includes two parts: the structural semantics is specified through the type graph, and the behavioral semantics is specified through interaction diagrams and graph transformation rules.

As having explained in the previous sections, we use the type graph to specify the structural semantics of the the IDL file. The type graph can specify formally the architectural elements, the meaning of attributes and the relationship between the elements. We use graph transformation rules to specify the behavioral semantics, which includes the GTP messages and operations. Our specification focuses on the change of states of the elements involved for communication and collaboration. We can specify how the instances of objects are affected by the execution of an operation, thus the behavior of the involved internal elements for an operation can be defined formally.

Using the TGTS based modeling approach, we can specify formally the most important aspect of the component interaction in mobile settings, i.e., the dynamic change of components involved in interactions, the collaboration and synchronization definition between different objects and operations, etc. The errors caused by conflicting or unclear definition in the specification can be thus avoided already in the specification process.

We will illustrate the specification of IDL operation updateLocation as an example. In the wireless CORBA IDL (see Appendix A), the signatures of the operation updateLocation is defined as follows:

```
Void update_Location (in Terminalid terminal-id,
in AccessBridge new-access-bridge)
raises (UnknownTerminalid, IllegalTargetBridge);
```

In Fig. 6.31, the graph transformation rule for operation *updateLocation* is shown, which is called by the *Access Bridge* to update the location of a mobile terminal after handoff. According to its pre-condition, expressed by the left-hand side object graph, an operation call updateLocation is made by the client *ac2* to the server *hla*. As the effect of the operation (see right side graph), the corresponding nodes and relationships will be created or deleted.

The synchronization between different objects can be assured through the definition of the pre and post condition of the rules. The rule is only allowed to execute if the pre-condition is satisfied. When some of the edges or vertices is missing in the pre-condition, the rule can not be activated. For instance (in Fig. 6.31), if the

records edge between the *hla:HomeLocationAgent* and *newM:MobileIOR* is missing, that means, the *newM* is not recorded yet by the *hla*, the operation updateLocation can not be activated, it will wait till the *newM* is recorded by the *hla*, which can be done by activating actions performing this task. Such sub-actions inside one operation can be asynchronous.

The errors caused because of unclear synchronization definition between different objects and operations will be cleared using our model. For instance, if we consider the error mentioned in Issue 4422, it would be possible to reach a terminal-initiated handOff behavior from a scenario that starts as a network-initiated handOff. While the issue like 4614 will be clearly defined too. It would be possible to reach a situation where a terminal is attached to an access bridge, yet no access bridge is tracked by the home location agent, which is a dead end then, since there are no graph transformation rules defined for this situation.

Chapter 7

Style Refinement

7.1 Overview

When developing the style, we use a stepwise refinement-based approach in order to decrease complexity and enhance reusability. We start from a simple abstract (i.e., conceptual) style that is refined to a more concrete style (platform-independent), which is further refined to an even more concrete style (platform-specific). The conceptual style defines functional requirements for a family of middleware for mobile systems through defining the required structural and behavioral elements. It can be refined to several different concrete styles that contain newly added structural and behavioral elements. The refined styles should still satisfy the functional requirements and they belong to the same family of middleware, although when they use different design strategies to realize the requirements.

Refinements are the basic steps in the architecture centric software development process. However, it is difficult to check whether a concrete architecture is a correct refinement of a more abstract architecture. Especially, the correctness of the newly added architectural elements is difficult to be ensured since it is difficult to directly map them to the abstract ones. In this chapter, we will investigate refinement relations between the abstract and concrete style that are specified using Typed Graph Transformation Systems (TGTSs). We will provide refinement criteria to ensure that the refined concrete style is a correct refinement of the abstract style. The criteria include both structural ones and behavioral ones. Especially, we formalize the refinement relationship between two abstract layers mainly based on the mapping of rules. We derive a simplified rule for a sequence of rules through a scenario based construction. The derived rule preserves the semantics of the sequence of rules, thus it can be used to substitute the complicated rule sequence. By using the derived simplified rule, we build a bi-directional mapping between a sequence of rule and another rule sequence. We also develop an algorithm to construct the derived rule.

We call the approach *rule mapping-based refinement* since it is mainly based on the mapping of rules. The approach can check the correctness of the sequence of rules. It also enables us to check the correctness of newly added structural and

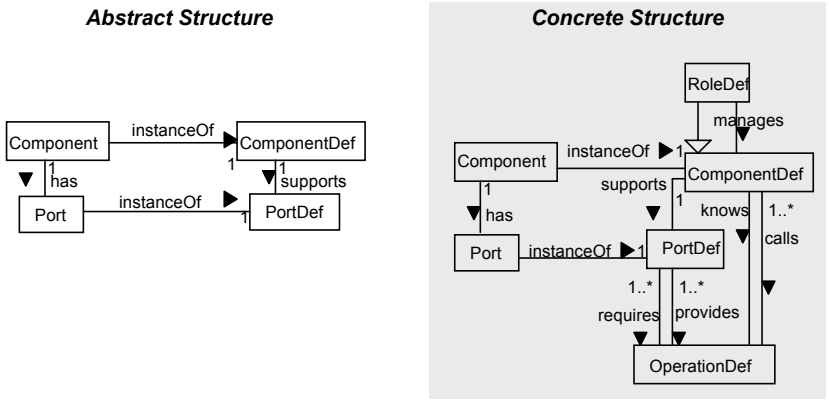

Figure 7.1: Structural refinement

behavioral elements through the scenario construction, which associates the completely different concrete elements with the abstract ones. Besides, our formalization can construct exactly the needed state graphs and transformation sequences for checking, which makes the approach efficient and practical to large systems. It also enables us to use the existing graph transformation simulation tool Fujaba as the basis for automating the refinement consistency check.

We will start the chapter with the introduction of the requirements for the refinement in Section 7.2. The existing approaches and open problems are introduced in Section 7.3. Afterwards, we will explain how to refine a TGTS-based abstract architectural style to a concrete one in Section 7.4. Section 7.5 evaluates our approach against the requirements listed in Section 7.2, and compares the approach to others.

7.2 Requirements for the refinement

One main requirement for refinement is to preserve the desired properties. Some researchers argued that different domains have different requirements for the properties. Therefore, which properties should be preserved vary in different architecture refinement process [4, 101, 46]. The examples are computational behavior [86], performance [4], or something completely different.

In our context, we start from a simple abstract (i.e., conceptual) style that is refined to a more concrete style (platform-independent), which is further refined to an even more concrete style (platform-specific). The conceptual style defines functional requirements for a family of middleware for mobile systems through defining the required structural and behavioral elements. It can be refined to several different concrete styles that contain newly added structural and behavioral elements. The refined concrete styles should still satisfy the functional requirements and they belong to the same family of middleware, although when they use different design strategies to realize the functional requirements.

Therefore, we require that the refinement is *functional-preserved*, which means that the functional requirements of the abstract style should be preserved in the refined concrete style and thus it satisfies the functional requirements. At the same time, we also require that the refinement is *functional-constrained*, which means that the concrete style is only allowed to contain elements that are required or constrained by the functional requirements, since the concrete elements are constructed in order to realize a specific abstract task or functionality. It should not contain elements that exceed the scope of the functional requirements. This requirement is important for checking the correctness of the concrete structural and behavioral elements, especially the newly added ones, which are very different from the abstract ones and whose correctness is difficult to be ensured.

We use Typed Graph Transformation Systems (TGTSs) to specify the style formally, which is separated into a structural part (UML class diagrams) and a behavioral part (graph transformation rules). Accordingly, we require refinement criteria that allow us to check correctness of the structure and behavior of the two architectures. We divide it further into structural and behavioral refinement.

Correspondingly, we require *structural refinement criteria* that ensure that the abstract structural elements are preserved in the concrete style, i.e., every abstract structural element requires a corresponding concrete structural element, which includes not only the classes but also associations and cardinalities. We call it *structural-preservation* that preserves the abstract structural elements. The criteria should also ensure that all the concrete structural elements are constructed in order to realize a specific abstract task or functionality. It is not allowed to have structural elements that are irrelevant to the functionality. That means, all the concrete structural elements should be used in the refined concrete rules that define the functionality. We call it *structural-constraint* that is important to constrain the newly added concrete structural elements, which also include the inherited classes that are presented as inheritance relationship in the class diagram.

Figure 7.1 gives an example of an abstract structure and a refined concrete structure, which are taken from the conceptual style (Fig. 6.4, see Section 6.3) and the platform-independent concrete style (Fig. 6.10, see Section 6.4) respectively. All the abstract elements are preserved in the concrete structure. The newly added elements *RoleDef* and *OperationDef* are added in order to realize the functionality of building and deleting sessions, which is modeled as a sequence of concrete rules *remoteCall, request, callReturn, dispatchMessage, sendMessage, sendReplyMessage, reply* (see Section 6.4). Therefore, *RoleDef* and *OperationDef* are used in these rules.

We also require *behavioral refinement criteria* that ensure that the abstract behavior is preserved in the concrete style. This is *behavioral-preservation* that preserves the abstract behavior. After comparing the specified rules of the three-layers's styles (see Table 7.1, 7.2, 7.3, 7.4), we further differentiate three kinds of behavioral preservation according to our two kinds definition of rules: rules for dynamic changes that have a nondeterministic sequence and rules for component interaction that have a fixed sequence (see Section 5.3). The fixed sequence of the

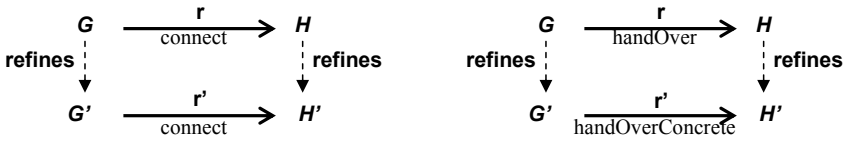

Figure 7.2: Refinement of an abstract rule to a concrete rule

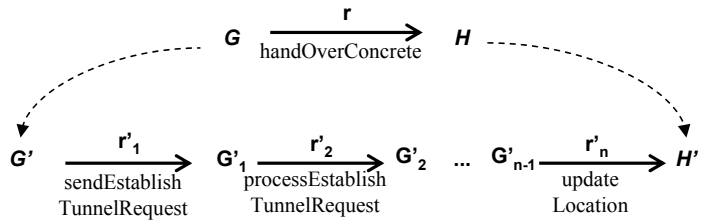

Figure 7.3: Refinement of an abstract rule to a fixed sequence of concrete rules

rule is specified using an UML sequence diagram, which specifies the coordination and synchronization among different actions and components. Therefore, the rules are only allowed to appear in the specified sequence, but not in other sequences.

The first kind of behavioral preservation (in Fig. 7.2) is rather simple, which is about that an abstract rule is refined to a concrete rule. Therefore, the abstract behavior has a corresponding concrete behavior, i.e., an abstract transformation step has a corresponding concrete one. The second kind of behavioral preservation (in Fig. 7.3) is about refining an abstract rule to a fixed sequence of concrete rules, which are specified using UML sequence diagrams, as already mentioned. In such a refinement, it happens often that some concrete rules are completely different from the abstract ones since they are added to present some new concepts. The examples are the rules $r'_1, r'_2, ..., r'_n$ in Fig. 7.3. We require that the correctness of the rule sequence should be ensured. Therefore, the abstract behavior has a corresponding sequence of concrete behavior, i.e., an abstract transformation step has a corresponding fixed sequence of concrete steps.

The third one (in Fig. 7.4) is about refining a fixed sequence of abstract rules to

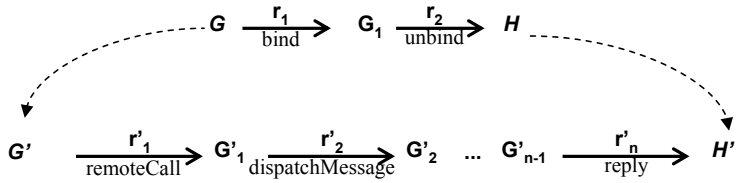

Figure 7.4: Refinement of a fixed sequence of abstract rules to a fixed sequence of concrete rule

a fixed sequence of concrete rules. Again, it happens here often that some concrete rules are completely different from the abstract ones since they are added to present some new concepts. The examples are the rules $r'_2, r'_3, ..., r'_{n-1}$ in Fig. 7.4. We require that the correctness of the rule sequence should be ensured for both of the abstract rules and the concrete rules. Therefore, the sequence of abstract behavior has a corresponding sequence of concrete behavior, i.e., a sequence of abstract transformation steps has a corresponding sequence of concrete steps.

Besides, the behavioral refinement criteria should also ensure that all the concrete behaviors are constructed in order to realize a specific abstract task or functionality. It is not allowed to have behaviors that are irrelevant to the functionality. We call it *behavioral-constraint*, whose similar terms are *observational substitutability* or *behavior reflection* in some literatures [145]. We further differentiate two kinds of behavior constraints that are opposite to the behavioral-preservation ones. For the concrete rule specifying dynamic changes (in Fig. 7.2), we require that the concrete behavior has a corresponding abstract behavior, i.e., a concrete transformation step has a corresponding abstract one. For a fixed sequence of concrete rules that is for component interaction, (in Fig. 7.3, Fig. 7.4), we require that the sequence of concrete behavior has a corresponding abstract behavior or a corresponding sequence of abstract behaviors, i.e., the sequence of concrete steps has a corresponding abstract step, or a corresponding sequence of abstract steps.

In addition, we require *reusability* of the refinement, that means, a refinement relationship should be defined in terms of the two involved style models rather than in terms of individual architectures so that it can be applied to any two architectures conforming with these style models, respectively, which is called *style-based refinement* [4].

Since refinement is not an easy task and thus error-prone and cost-intensive if done by hand, we are also aiming at *tool support*, which should help us to decide whether a concrete architecture is a valid refinement of an abstract architecture.

7.3 Existing approaches and open problems

There exist many different architecture refinement methods and notions since refinement is related to the specification method and the applied domain. For example, Rapide [86] represents system executions as partially ordered event sets. It provides event pattern mappings that translate concrete architecture executions to the abstract ones. Then, one can check whether the events of the concrete architecture also satisfy the constraints specified for the abstract architecture. Although these pattern mappings are a powerful and flexible concept for validating executions of the concrete architecture against abstract constraints, they do not ensure that all the concrete behaviors are constructed in order to realize a specific abstract task or functionality, as required by the *behavioral-constraint* or *observational substitutability*.

Since we use graph transformation systems as the underlying formalism to

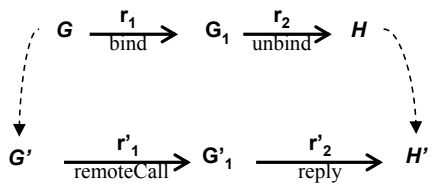

Figure 7.5: An example of a wrong refinement

describe architectures, we will concentrate now on existing work on refinement of graph transformation systems. The general idea is to relate the transformation rules and, thus, the behavior of an abstract graph transformation system to the rules of a more concrete transformation system. We can judge these refinements as syntactical ones or semantical ones.

Grosse-Rhode et. al. [61, 62] propose refinement mappings between abstract and concrete rules that can be checked syntactically. One of the conditions requires that the abstract rule and its refinement must have the same pre- and post-conditions except for retyping. Based on this very restrictive definition they can prove that the application of the concrete rule expression yields the same behavior as the corresponding abstract rule. The draw-back of this approach is that it cannot handle those cases where the refined rule contains concrete architecture-specific elements that do not occur in the abstract rule. In addition, it does not support the refinement of a sequence of rules. Similarly, the work in [38] is based on a syntactical relationship between two graph transformation systems. This approach is less restrictive since it allows additional elements at the concrete level. But it is still difficult to apply if there are no direct correspondences between abstract and concrete rules. Moreover, it does not support the refinement of a sequence of rules. The *behavioral-constraint* or *observational substitutability* is not supported neither.

Some researchers [145, 15] provide more flexible, semantic-based notion of style refinement. They do not define a fixed mapping between the various transformation rules but only between the structural parts of the graph transformation system, since it is difficult to build a direct behavioral mapping between those styles that have completely different behaviors. Then, they check whether all system states of an abstract model are also reachable at the concrete level, no matter by which order of transformation rules. The requirement for *behavioral-preservation* is hence satisfied. In order to ensure *observational substitutability*, they check also whether all system states of a concrete model are also reachable at the abstract level. This is further done through an abstraction function that maps the concrete instance graph to the abstract one. By avoiding direct refinement mappings between transformation rules, they can also relate transformation systems with completely different behavior, and they are flexible enough to cope with alternate refinements.

However, the approach is not suitable for the systems that require correctness check of the fixed sequence of rules. In this case, it can happen that the required abstract state is reached through an incorrect sequence of concrete rules, but not through the required correct sequence of concrete rules, which will be considered as a correct refinement in their approach. This happens often for the refinement of a fixed sequence of dual rules that will not change anything after execution. For example, if we execute at first the rule *bind* that builds a session, and then *unbind* that deletes a session, a correct refinement should be a sequence of rules as shown in Fig. 7.4. If we use the approach proposed by [145, 15], it will only check whether G, and H are reachable in the concrete style, which can be reached through another sequence (see Fig. 7.5) but not the required one, which will be considered as a behavior-preservation refinement. However, it is not enough to ensure that it is a correct refinement since the rule sequence is not correct.

In addition, it is still very difficult to check the correctness of the completely different concrete behavior using their approach. They ensure *observational substitutability* by checking whether every concrete execution has an abstract equivalent or not. However, the complete different rules can create concrete states that are completely different from the abstract ones. The examples of such concrete states are $G'_1, G'_2, ..., G'_{n-1}$ in Fig. 7.3 and $G'_1, G'_2, ..., G'_{n-1}$ in Fig. 7.4. In their approach, such a concrete state is mapped to an empty abstract graph using the abstraction function, since there exist no corresponding abstract classes for the concrete ones in the instance graphs. This is then considered as a correct refinement. Every complete different concrete rule will be considered as a correct refinement in their approach. Obviously, this is not enough in order to ensure *observational substitutability*.

7.4 Rule mapping - based style refinement

As having explained in Chapter 5, we use the TGTS, $\mathcal{G} = \langle TG, C, \mathcal{R} \rangle$, to specify the architectural style for the middleware. TG (Type Graph) is here an UML class diagram that defines the structural part of the style. C is a set of constraints. $\mathcal{R}$ is a set of graph transformation rules (given by pairs of object diagrams) that defines the behavioral part of the style. We will now provide refinement criteria to check whether a concrete style $\mathcal{G}' = \langle TG', C', \mathcal{R}' \rangle$ is a correct refinement of an abstract architectural style $\mathcal{G} = \langle TG, C, \mathcal{R} \rangle$. The criteria include structural refinement ones and behavioral refinement ones that allow us to check correctness of the refined structure and behavior. Correspondingly, we separate the refinement into several parts: structural refinement, behavioral refinement and style refinement. The criteria satisfy the listed refinement requirements (see Section 7.2), which are analyzed in each of the refinement part.

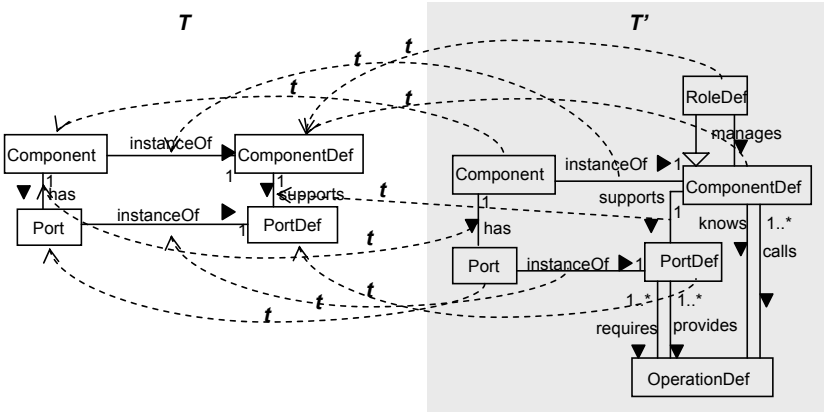

Figure 7.6: An example of the type graph mapping t

7.4.1 Structural refinement

The structural part of the style is defined by graphs, i.e., UML class diagrams. More specifically, the modeling graphs occur at two levels: the type level (the type graph TG) and the instance level (the instance graphs typed over TG $\langle G, tp_G \rangle$). The type graph defines the style vocabulary and the relationships among these elements. The instance graphs represent both the declarative definition of an architectural style and the architectural configuration. When taking a step-wise refinement development process of the style, we start from a small and abstracted set of architectural elements in the abstract style, which will be further refined to more concrete elements in the concrete style. If we compare the type graph of the conceptual style (Fig. 6.4), the platform-independent concrete style (Fig. 6.10), and the platform-specific concrete style (Fig. 6.23, Fig. 6.24) specified in Chapter 6, we distinguish three types of structural refinement:

1. All the structural elements (nodes) and the relationships (edges) in the abstract style can be mapped to their equivalent in the concrete style. In most cases, they will be kept the same in the concrete style. For example, the structural elements in the conceptual style are kept the same in the platform-independent concrete style, whose structural elements can be further mapped to their equivalent in the platform-specific concrete style. We call such elements direct-mapped elements.

2. Some elements in the concrete style are inherited from those in the abstract style, which is presented as inheritance relationship in the class diagrams. The examples are the *AccessBridgeDef, TerminalBridgeDef* in the platform-specific concrete style (Fig. 6.23) that are inherited from the *BridgeDef* in the platform-independent concrete style (Fig. 6.10). We call such elements inherited elements.

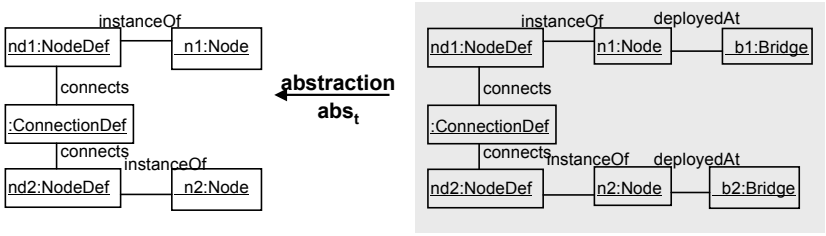

Figure 7.7: Abstraction of an instance graph

3. All other elements in the concrete style are concrete style-specific, that means, they are added to represent some new concepts in the concrete style. For example, the message related elements in the platform-independent concrete style are added in order to realize the functionality of building and deleting sessions, which can not be directly mapped to any abstract elements. We call such elements concrete-specific elements.

For the direct-mapped elements and inherited elements, we define a type mapping $t : TG' \rightarrow TG$, formally a *partial* surjective graph homomorphism, which maps elements of the concrete type graph TG' to the elements of the abstract type graph TG. The definition of t is driven by semantic correspondences between the elements of the two styles. If the concrete type graph TG' is an extension of the abstract one, it should contain equivalent elements for all elements of TG. In our case, TG is even a subgraph of TG'. Thus the abstraction mapping easily becomes surjective. This criterion satisfies the required *structural-preservation*. The type mapping t maps also the inherited elements to the images of their supertypes in the abstract type graph. For example, Figure 7.6 gives part of the type mapping between the conceptual style and the platform-independent concrete style. This mapping is driven by semantic correspondences between the elements of the two styles. The inherited RoleDef is mapped to ComponentDef. The concrete-specific elements will not be mapped since they are completely different from the abstract ones.

Def. 7.1 A type graph TG' is a *structural-preserved* refinement of the abstract graph TG, if it contains equivalent elements for all elements of TG, i.e., there exists a partial surjective type mapping $t : TG' \rightarrow TG$.

Based on the type mapping t, we can derive an abstraction function $abs_t : \mathbf{Graph}_{TG'} \rightarrow \mathbf{Graph}_{TG}$ that abstracts instance graphs typed over TG' to those typed over TG. The abstraction function informally consists of

1. Deleting all objects (with adjacent edges) and links which, due to the partiality of t, have a type in TG' but not in TG.

Aspects \ Style	Conceptual Style	Platform-independent concrete style	Platform-specific concrete style (Wireless CORBA)
Space	moveIn	moveIn	moveIn
	moveOut	moveOut	moveOut
	moveTo	moveTo	moveTo
Functionality (2. Wireless connection)	connect	connect	connect
	disconnect	disconnect	disconnect
		connectBridge	connectBridge
Component Interaction		disconnectBridge	disconnectBridge
		remoteCall	remotecall
		request	request
		callReturn	callReturn
		reply	reply
		dispatchMessage	dispatchMessage

Table 7.1: Direct - mapped rules

2. Renaming the types of the remaining elements according to t. We must also be sure that the modified structure is still a legal graph, i.e., that no edges are left dangling because of deleting their source or target vertices.

3. Extracting the maximal subgraph that satisfies all constraints C.

We illustrate the abstraction function (in Fig. 7.7) through an instance graph that is taken from the left side of *connectBridge* rule in the platform-independent concrete style, which is abstracted to left side of *connect* rule in the conceptual style. Based on the abstraction function, we can define then:

Def. 7.2 A concrete instance graph IG' is a *structural-preserved* refinement of an abstract instance graph IG, if it contains equivalent elements for all elements of IG, i.e., if $abs_t(IG') = IG$.

Till now, we have defined a semantic driven mapping between the structural elements of the two styles at type level and instance level. The mapping can check whether the elements of the abstract style are included or equal to the elements of the concrete style. Therefore, it satisfies the required *structural-preservation*. The mapping also ensures *structural-constraint* for the direct-mapped elements and inherited concrete elements, but not for the concrete-specific elements. We will do such check in the behavioral refinement, which will be introduced in the next section.

7.4.2 Behavioral refinement

The behavioral part describes the runtime behavior of the system, which is specified using the graph transformation rules $\mathcal{R}$ (see Section 5.3). We differentiate two kinds of rules: rules for dynamic changes and rules for component interaction, whose main difference is that the former has a nondeterministic sequence, while

Aspects \ Style	Conceptual Style	Platform-independent concrete style	Platform-specific concrete style (Wireless CORBA)
Functionality (2. Wireless connection)	connect	connectBridge	
	disconnect	disconnectBridge	
Functionality/ Component Interaction (3. Handover)	handOver	handOverConcrete	
Functionality (1. Interoperability) Component Interaction	bind	remoteCall	
	unbind	reply	

Table 7.2: Single - mapped rules

Aspects \ Style	Conceptual Style	Platform-independent concrete style	Platform-specific concrete style (Wireless CORBA)
Functionality/ Component Interaction (3. Handover)		handOverConcrete	Sequence Begin sendEstablishTunnelRequest processEstablishTunnelRequest sendEstablishTunnelReply processEstablishTunnelReply sendReleaseTunnelRequest processReleaseTunnelRequest sendReleaseTunnelReply processReleaseTunnelReply updateLocation Sequence End
Component Interaction		sendMessage	Sequence Begin buildTunnel sendMessageConcrete Sequence End
		sendReplyMessage	Sequence Begin sendReplyMessageConcrete releaseTunnel Sequence End

Table 7.3: Sequence - mapped rules

Style / Aspects	Conceptual Style	Platform-independent concrete style	Platform-specific concrete style (Wireless CORBA)
Functionality (1. Interoperability) Component Interaction	Sequence Begin bind connect moveIn handOver moveOut disconnect unbind Sequence End	Sequence Begin remoteCall dispatchMessage connectBridge sendMessage dispatchMessage request callReturn moveTo handOverConcrete dispatchMessage sendReplyMessage disconnectBridge dispatchMessage reply Sequence End	

Table 7.4: Scenario - mapped rules

the latter presents a fixed sequence, which is specified using UML Interaction diagrams (see Section 5.3).

We can define four types of behavioral refinement after comparing the specified rules of the three-layers's styles (see Table 7.1, 7.2, 7.3, 7.4):

1. The abstract rule that is not changed and it will be directly used in the concrete style. Those rules generally model the very basic behavior of the system, which can not be refined anymore. The examples are rules for space definition: *moveIn, moveOut, moveTo* and rule for wireless connection: *connect, disconnect,* etc. We call such rules *direct-mapped rules* (see Table 7.1).

2. The abstract rule that is refined to another concrete rule. The left-side and the right-side of the rule are correspondingly refined. The example is the *handOver* rule in the conceptual style, which is refined to *handOverConcrete* rule in the platform-independent concrete style (see Table 7.2). Besides, *bind* and *unbind* in the conceptual style can be mapped to *remoteCall* and *reply* in the platform-independent concrete style respectively, if we compare only the precondition and postcondition of the rule. The precondition and postcondition of *bind* and *unbind* are included inside those of *remoteCall* and *reply* (see Section 6.3, Section 6.4). We call such rules *single-mapped rules*.

3. The abstract rule that is refined to a fixed sequence of concrete rules (see Table 7.3). The example is the *handOverConcrete* rule in the platform-independent concrete style, which is refined to a sequence of rules *sendEstablishTunnelRequest, processEstablishTunnelRequest, sendEstablishTunnelReply, processEstablishTunnelReply, sendReleaseTunnelRequest, processRelease-*

TunnelRequest, sendReleaseTunnelReply, processReleaseTunnelReply, updateLocation. In the refinement, it is often that some concrete rules are completely different from the abstract ones since they are added to present some new concepts. For example, the listed intermediate rules are very different, which can not be mapped directly to the abstract style. We can associate them with the abstract rule through a scenario definition, that means, the abstract rule is realized through a scenario. This is because that the sequence of the refined concrete rules is specified using an UML sequence diagram, which also stands for a scenario of the concrete style. We call such rules *sequence-mapped rules*, which maps an abstract rule to a concrete scenario.

4. A fixed sequence of abstract rules that is refined to a fixed sequence of concrete rules (see Table 7.4). In such a refinement, it happens often that some concrete rules are completely different from the abstract ones since they are added to present some new concepts. For instance, the rules *request, callReturn, reply, dispatchMessage, sendMessage* in the platform-independent concrete style are very different, which can not be mapped directly to any rules in the conceptual style. However, they can be associated with the abstract rules through a scenario definition, since they are added in order to realize a specific application scenario in the abstract style. The abstract scenario specified through a sequence of rules *bind, connect, moveIn, handOver, moveOut, disconnect, unbind* is realized as the concrete scenario with a sequence of rules *remoteCall, dispatchMessage, connectBridge, sendMessage, dispatchMessage, request, callReturn, moveTo, handOverConcrete, dispatchMessage, sendReplyMessage, disconnectBridge, dispatchMessage, reply*. We call the rules *scenario-mapped rules*, which maps an abstract scenario to a concrete one.

We define different refinement criteria for these rules, which will be introduced in the following subsections.

7.4.2.1 Refinement of the direct-mapped and single-mapped rule

The refinement for the direct-mapped rule is very simple since they are not changed in the concrete style. The only possible change is renaming of the rules in case that their name is different in the concrete style. We include the refinement for the direct-mapped rule in the refinement of the single-mapped rule. The refinement for the single-mapped rule includes three steps: (1) Refine the left-side of the abstract rule to that of the concrete rule; (2) Refine the right-side of the abstract rule to that of the concrete rule; (3) Rename the rule. Figure 7.8 illustrates the refinement through an example of refining the *handOver* rule in the conceptual style to *handOverConcrete* rule in the platform-independent concrete style.

Therefore, we define the refinement of the rule based on the instance graph refinement:

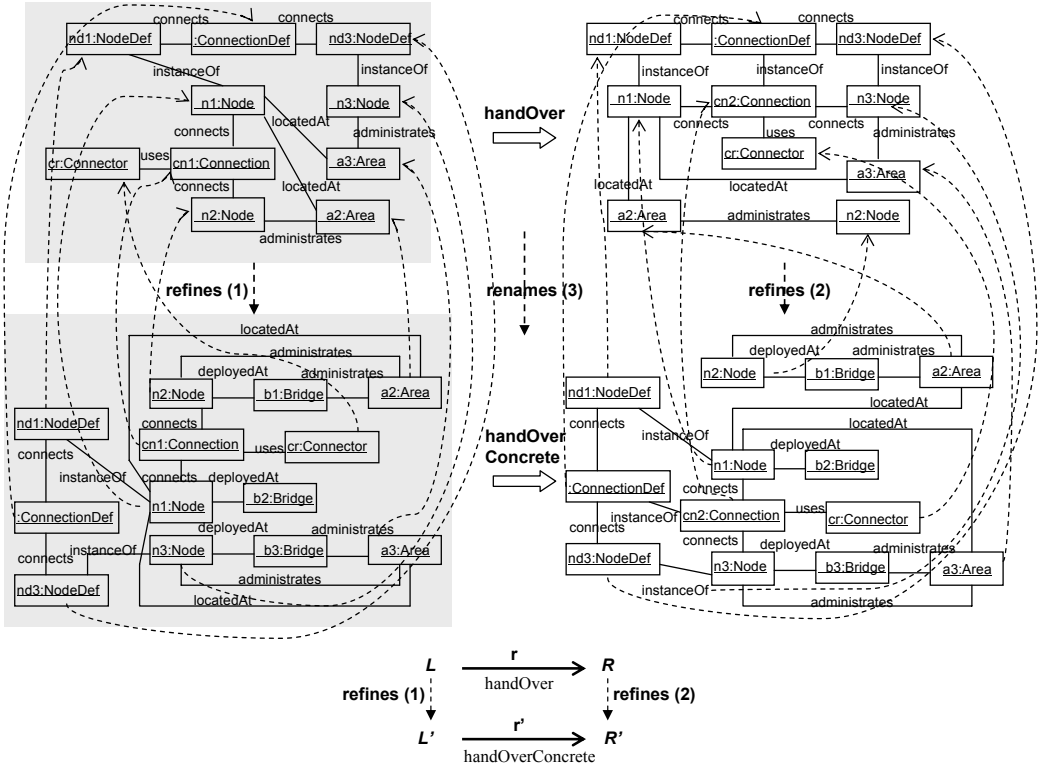

Figure 7.8: Refinement of a single-mapped rule

Def. 7.4 For a direct-mapped and single-mapped graph transformation rule $r : L \Rightarrow R$ in the abstract transformation system $\mathcal{G}$, the concrete rule $r' : L' \Rightarrow R'$ in the concrete transformation system $\mathcal{G}'$ is a correct refinement, if L' refines L and if R' refines R, i.e., $abs_t(L') = L \wedge abs_t(R') = R$.

The refinement criteria build a bi-directional mapping between an abstract rule and a concrete rule. Therefore, the abstract behavior has a corresponding concrete behavior, i.e., an abstract transformation step has a corresponding concrete one. Vice versa, the concrete behavior has a corresponding abstract behavior, i.e., an concrete transformation step has a corresponding abstract one. The required *behavioral-preservation* and *behavioral-constraints* are hence ensured.

7.4.2.2 Refinement of the sequence-mapped rule

A sequence-mapped rule $r : L \Rightarrow R$ is refined to a fixed sequence of rules $r'_1, r'_2, ..., r'_n$, with $r'_1 : L'_1 \Rightarrow R'_1$, $r'_2 : L'_2 \Rightarrow R'_2$, ..., $r'_n : L'_n \Rightarrow R'_n$. The sequence of the refined concrete rules is specified using a UML sequence diagram, which also stands for a scenario of the concrete style. In such a refinement, it happens often that some concrete rules are completely different from the abstract ones since they are added to present some new concepts.

We develop an approach that can check the correctness of the sequence of rules. We derive an initial graph L' for the refined concrete rules, which contains the needed precondition of all of the rules in the sequence (Fig. 7.9). The rules are applied then one by one following the sequence. We denote it as $L' \stackrel{r'_1}{\Longrightarrow} M'_1 \stackrel{r'_2}{\Longrightarrow} M'_2 \; ... \; M'_{n-1} \stackrel{r'_n}{\Longrightarrow} R'$, where M'_i is the created intermediate graph, R' is the last graph that contains the result (i.e., postcondition) after applying the last rule. Afterwards, we derive a new concrete rule $r' : L' \Rightarrow R'$, which is a simplified version of the refined sequence of concrete rules. The rule r' transforms the overall precondition L' to the overall postcondition R'. The new rule preserves semantics of the sequence of concrete rules since it is based on a scenario construction, i.e., it checks whether the scenario (defined by the rule sequence) is reachable. Therefore, it can be used to substitute the complicated rule sequence. The similar construction of rules is minimal elements of a class of so-called derived rules, which are concerned with the construction of rules by which a given sequence or rewriting steps can be simulated in a single step [83, 40].

We use the derived rule $r' : L' \Rightarrow R'$ and the intermediate graph $M'_1, M'_2, \; ... \; M'_{n-1}$ to check whether the refinement is correct. We start with L', and check whether M'_i, R' are reachable by applying the fixed rule sequence $s' = r'_1, r'_2, ..., r'_n$. We define the correctness criteria of the refinement as:

Def. 7.5 For a sequence-mapped graph transformation rule $r : L \Rightarrow R$ in the abstract transformation system $\mathcal{G}$, the concrete rule sequence $s' = r'_1, r'_2, ..., r'_n$ in the concrete transformation system $\mathcal{G}'$ is a correct refinement, if L' refines L and if R' refines R, (i.e., $abs_t(L') = L \wedge abs_t(R') = R$), and if

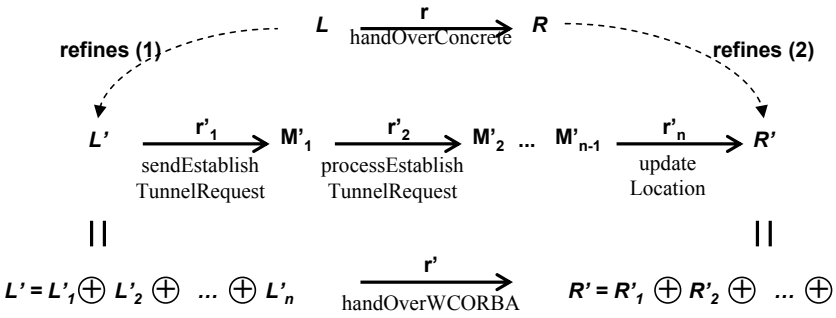

Figure 7.9: Refinement of a sequence-mapped rule

$L' \stackrel{r'_1}{\Longrightarrow} M'_1 \stackrel{r'_2}{\Longrightarrow} M'_2$ ... $M'_{n-1} \stackrel{r'_n}{\Longrightarrow} R'$, where $L' = L'_1 \oplus\ L'_2 \oplus$, ..., $\oplus\ L'_n$, $R' = R'_1 \oplus\ R'_2 \oplus$, ..., $\oplus\ R'_n$, $M'_i = R'_i + (L' - L'_i)$ and $r'_i : L'_i \Rightarrow R'_i$, the symbol $\oplus$ is used to indicate combination of preconditions or postconditions (see Algorithm 7.1) of the rules.

The refinement criteria build a bi-directional mapping between an abstract rule and the sequence of concrete rules via the derived simplified concrete rule. Therefore, the required *behavioral-preservation* is ensured since the abstract behavior has a corresponding sequence of concrete behaviors. The required *behavioral-constraints* is also ensured since the sequence of concrete behaviors has a corresponding abstract behavior. Similar to the approach proposed in [145], our definition of refinement is semantic-driven since we also check whether the scenario is reachable. The main difference is that we define a fixed mapping between the abstract rule and concrete rules, which is not proposed in [145]. Besides, we also associate the completely different concrete rules with the abstract rules through constructing the overall precondition and postcondition of the rule sequence. It allows then correctness check of the completely different concrete behaviors. Our approach also allows definition of the rule sequence and thus its correctness check.

Before applying the approach, we need to construct the important L' and R' at first. In order to keep semantic consistency, we can not just put all the preconditions L'_i and postcondition R'_i of the rules together. The rule defines a certain modification of instance graphs. It happens often that some elements of the postcondition graph are created in the intermediate step, which are deleted again by another rule. Those elements should not be included in the result graph. Or some elements of the precondition are derived after applying another rule, which should not be included in the start graph neither.

We derive L' and R' through a step by step approach, which means, we combine firstly the first two rules, whose result will be used to combine the next rule, and so on. The rules will be combined according to the specified sequence. We denote the result of L' as $L' = L'_1 \oplus\ L'_2 \oplus$, ..., $\oplus\ L'_n$, and the result of R' as $R' = R'_1 \oplus\ R'_2 \oplus$, ..., $\oplus\ R'_n$. We construct an algorithm to get the result graph of L' and R'. The algorithm is based on the theory of minimality of derived

rules in single pushout graph rewriting, whose theoretical proof is given by the researchers [83, 40]. By corresponding explicit constructions they have proved that there is a sequentially composed rule for each derivation sequence.

[Algorithm 7.1]

1. Let $i = 1$.
2. Add all nodes and edges that belong to L'_i, which defines the precondition of the rule r'_i.
3. Add all nodes and edges that belong to L'_{i+1} but do not belong to L'_i. Since it can be that some elements in L'_{i+1} are already present in L'_i. We do not need to add these ones, but only those that are specific to L'_{i+1}. We denote it as $L'_{i+1} - L'_i$.
4. Inside $L'_{i+1} - L'_i$, the elements can be further divided into two sets: those that are created after applying the previous rule r'_i, and those that are not created afterwards (see Fig. 7.10). We need to delete those that are created after executing the rule, since they do not need to be included in the overall precondition. We denote the nodes and edges that are created after applying the rule r'_i as $lc_i = R'_i - L'_i$. We describe the step as: delete $(L'_{i+1} - L'_i) \bigcap lc_i$. Till now, we get the result of $L'_i \oplus L'_{i+1} = L'_i + (L'_{i+1} - L'_i) - (L'_{i+1} - L'_i) \bigcap lc_i$ (see Fig. 7.11).
5. Similarly, we can get the result of R' by: Add all nodes and edges that belong to R'_{i+1}, which defines the postcondition of the rule.
6. Add all nodes and edges that belong to R'_i but do not belong to R'_{i+1}. Since it can be that some elements in R'_{i+1} are already present in R'_i. We do not need to add these ones, but only those that are specific to R'_i. We denote it as $R'_i - R'_{i+1}$.
7. Inside $R'_i - R'_{i+1}$, the elements can be further divided into two sets: those that are deleted after applying the rule r'_{i+1}, and those that are not deleted afterwards (see Fig. 7.10). We need to delete those that are deleted after applying the rule, which are intermediate elements that should not appear in the overall postcondition. We denote the nodes and edges that are deleted after applying the rule r'_{i+1} as $rd_{i+1} = L'_{i+1} - R'_{i+1}$. We describe the step as: delete $(R'_i - R'_{i+1}) \bigcap rd_{i+1}$. Till now, we get the result of $R'_i \oplus R'_{i+1} = R'_{i+1} + (R'_i - R'_{i+1}) - (R'_i - R'_{i+1}) \bigcap rd_{i+1}$ (see Fig. 7.11).
8. Let $L'_{i+1} = L'_i \oplus L'_{i+1}$, $R'_{i+1} = R'_i \oplus R'_{i+1}$, $i = i + 1$, and go to step 2, till $i = n$. Afterwards, we get the result $L' = L'_n$, $R' = R'_n$.

Based on the derived L', the intermediate graph M'_i can be easily got by applying the rule r'_i one after another:

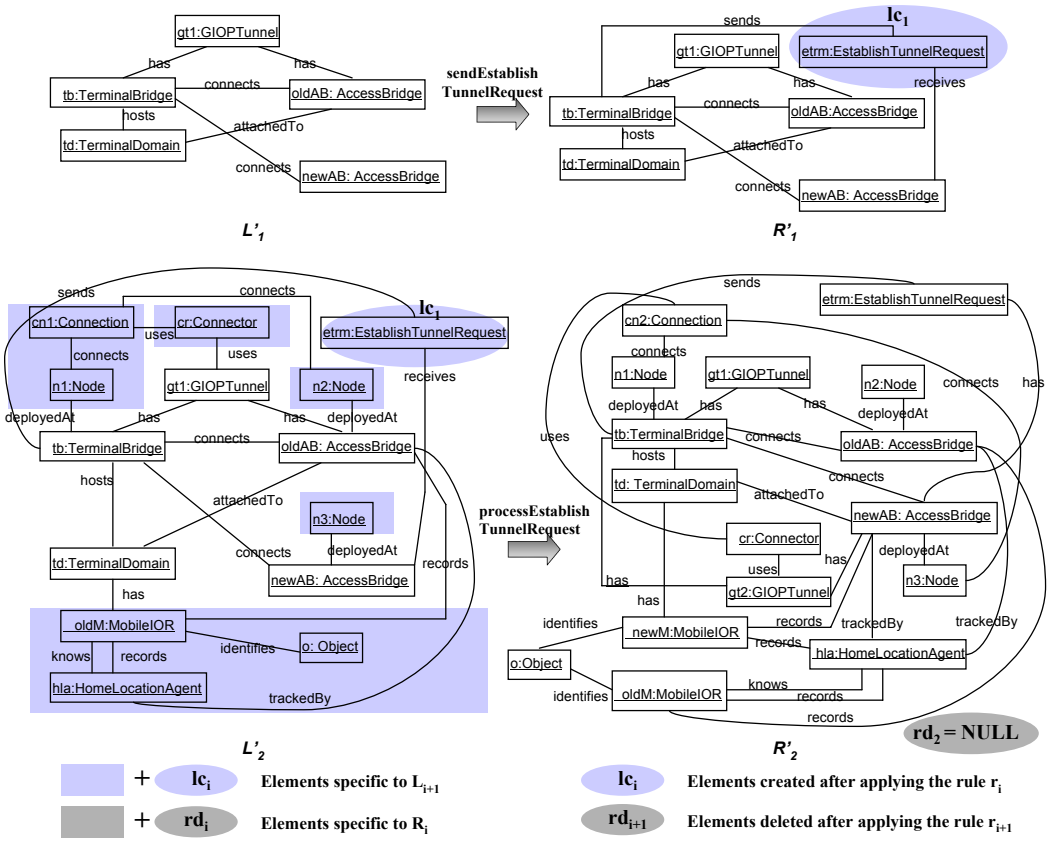

Figure 7.10: Construction of L' and R' using Algorithm 7.1 (Part 1)

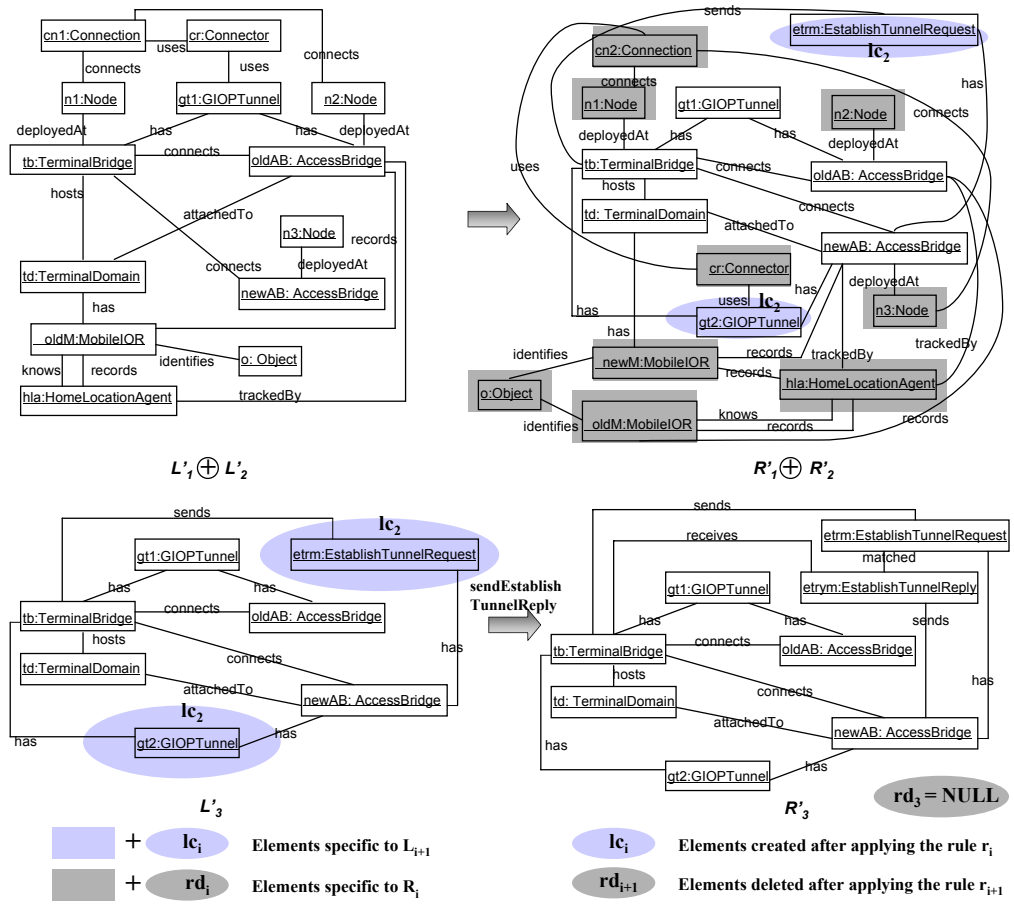

Figure 7.11: Construction of L' and R' using Algorithm 7.1 (Part 2)

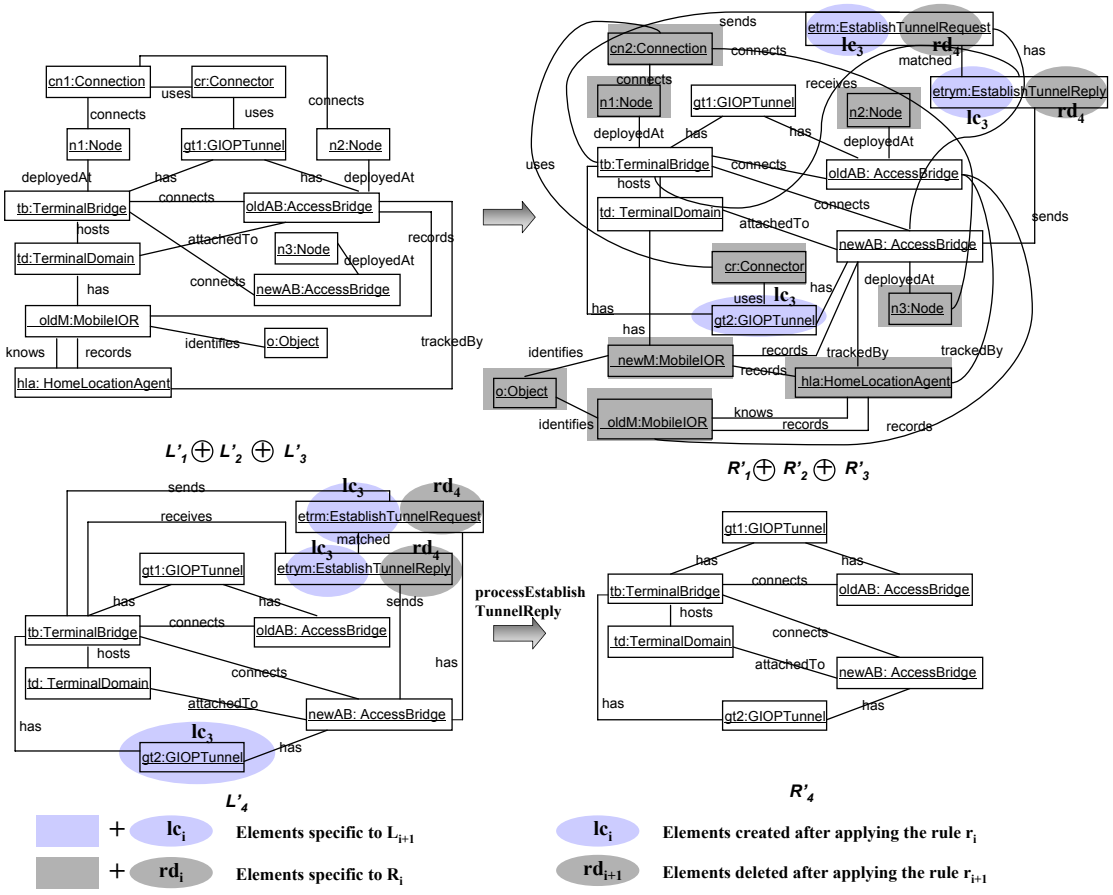

Figure 7.12: Construction of L' and R' using Algorithm 7.1 (Part 3)

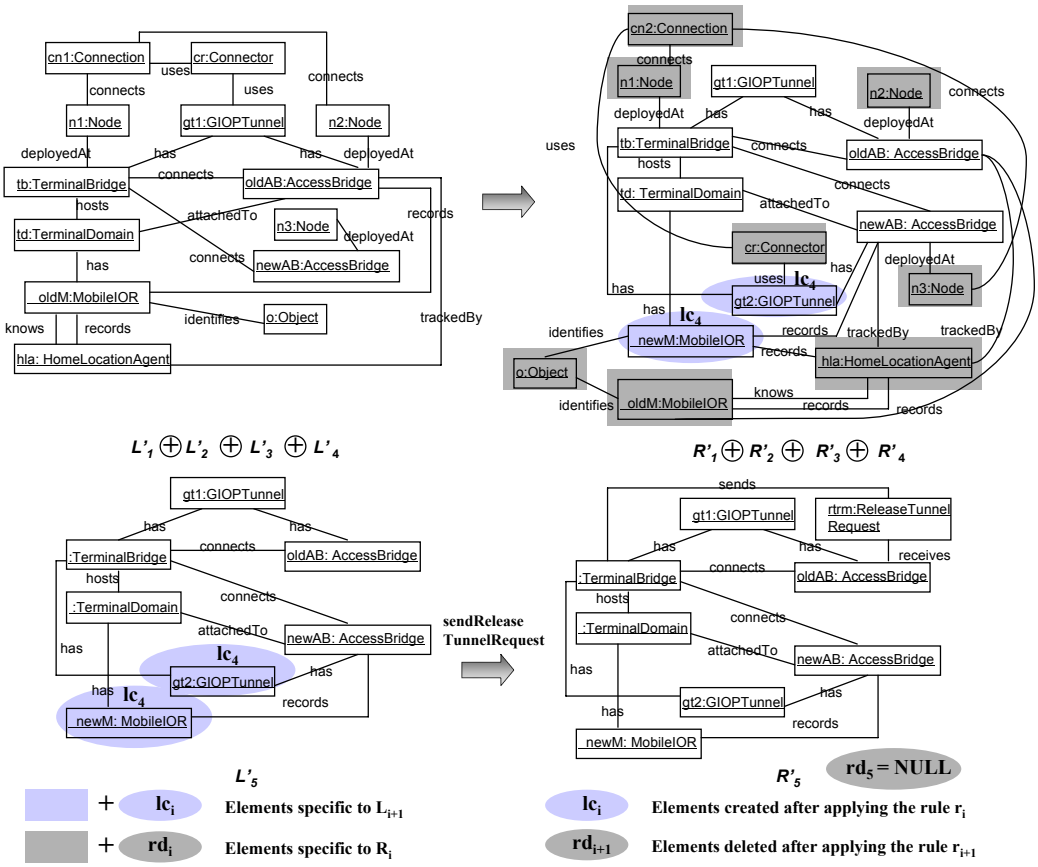

Figure 7.13: Construction of L' and R' using Algorithm 7.1 (Part 4)

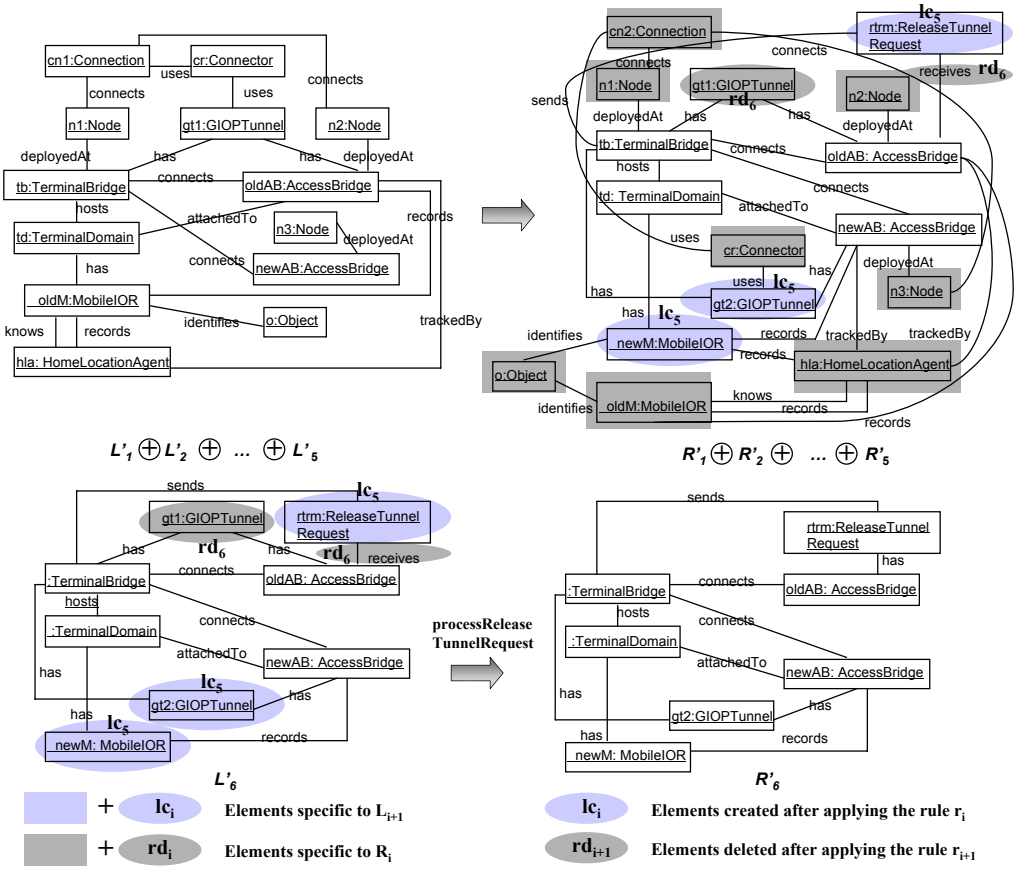

Figure 7.14: Construction of L' and R' using Algorithm 7.1 (Part 5)

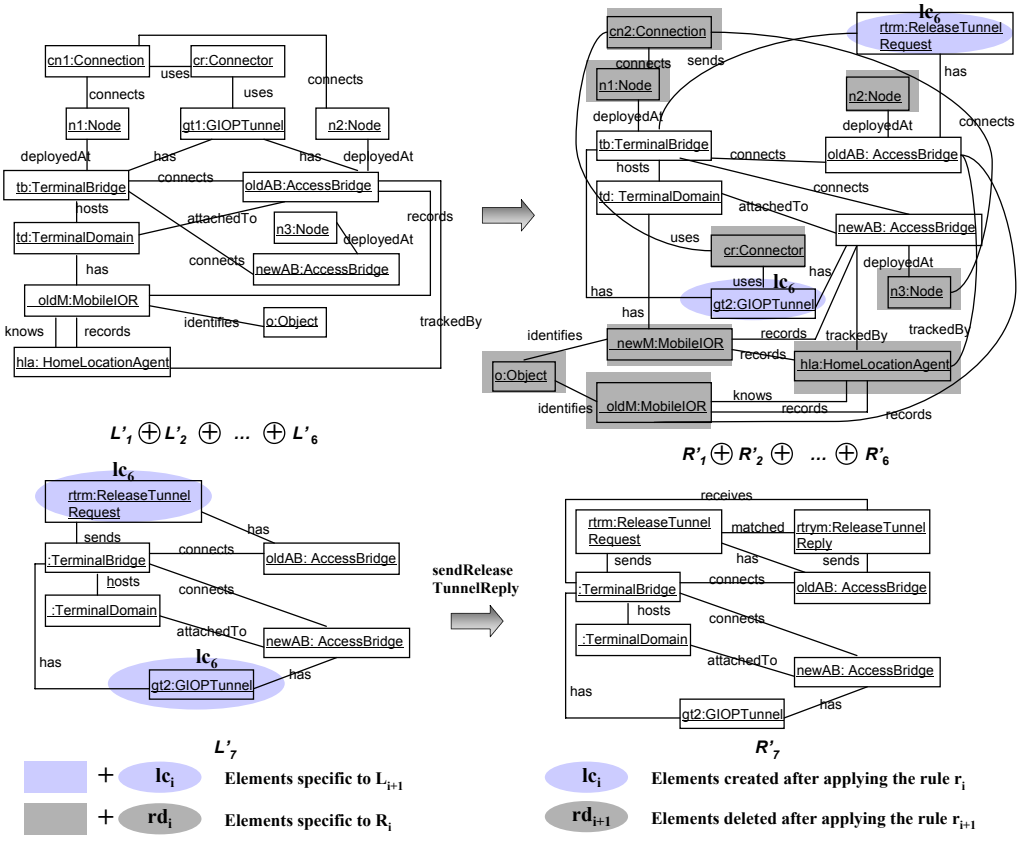

Figure 7.15: Construction of L' and R' using Algorithm 7.1 (Part 6)

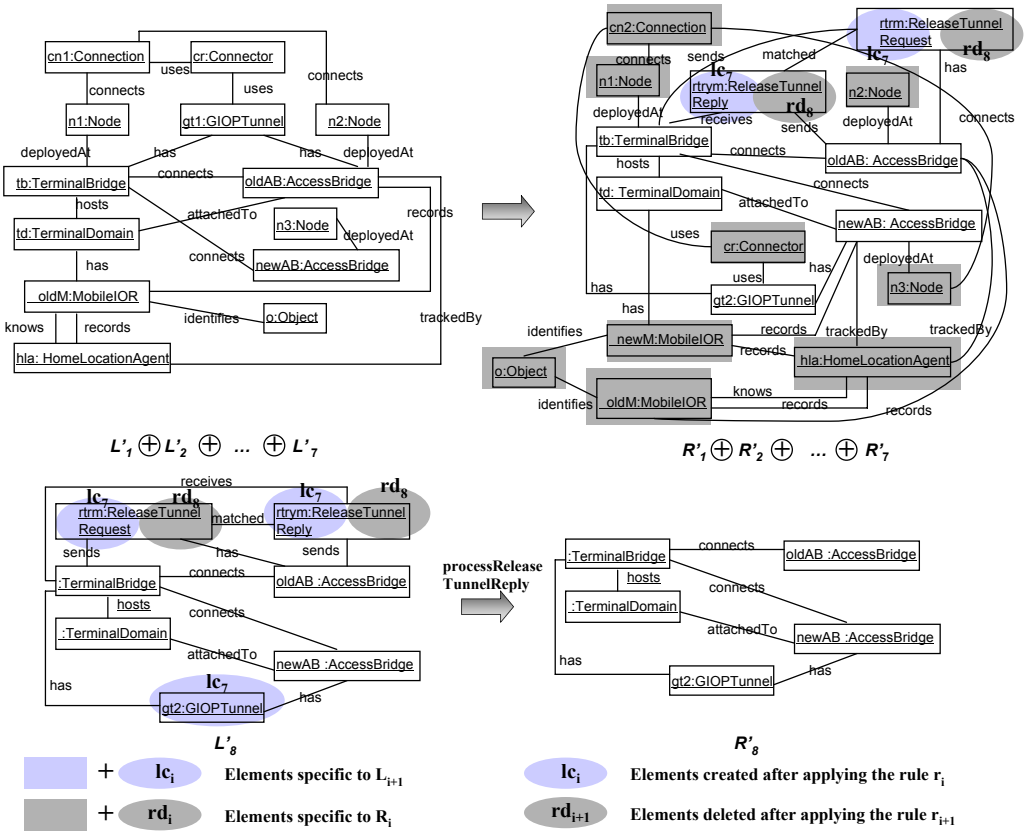

Figure 7.16: Construction of L' and R' using Algorithm 7.1 (Part 7)

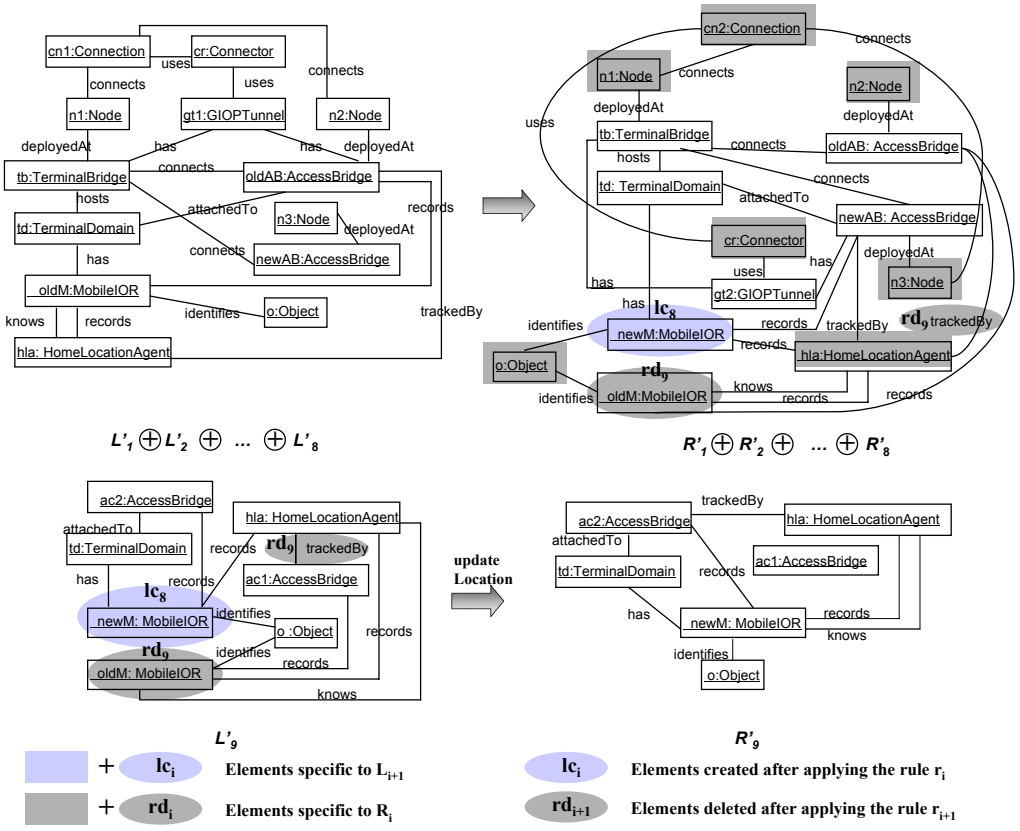

Figure 7.17: Construction of L' and R' using Algorithm 7.1 (Part 8)

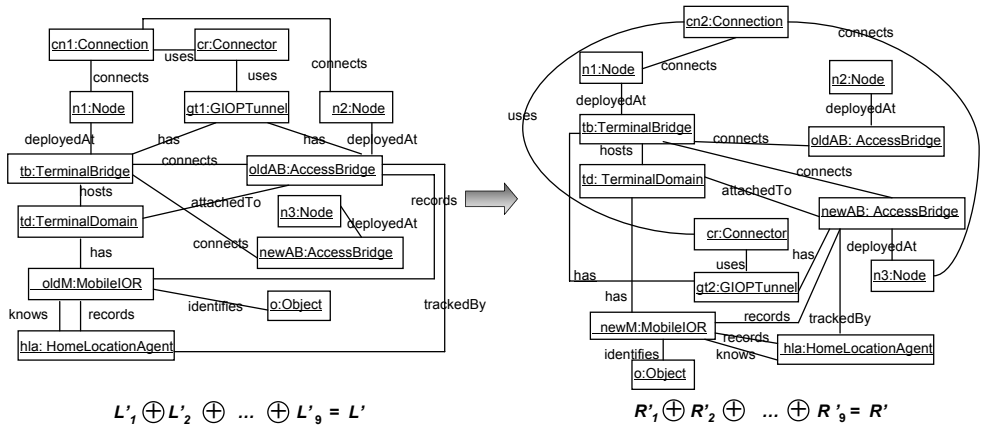

Figure 7.18: Construction of L' and R' using Algorithm 7.1 (Part 9)

[Algorithm 7.2]

1. Let $i = 1$.
2. Add all nodes and edges that belong to R'_i, which defines the postcondition of rule r'_i.
3. Add all nodes and edges that belong to L' but do not belong to L'_i, i.e., add $L' - L'_i$. It adds elements that are specific to L'. Afterwards, we get the result of $M'_i = R'_i + (L' - L'_i)$.
4. Let $L' = M'_i$, $i = i + 1$, and go to step 2, till $i = n - 1$.

An Example

As an example, consider the rule *handOverConcrete* in the platform-specific concrete style, which is refined to a sequence of nine rules in the platform-independent concrete style (see Fig. 6.26).

We derive L' and R' through a step by step approach, which means, we combine firstly the first two rules, whose result will be used to combine the next rule, and so on. The rules will be combined according to the specified sequence.

We illustrate how to use Algorithm 7.1 to construct L' and R'. We construct at first $L'_1 \oplus L'_2$ (see Fig. 7.10, Fig. 7.11). We add all nodes and edges that belong to L'_1, which defines the precondition of r'_1 (see Step 2, Algorithm 7.1). Then we add, as Step 3, all nodes and edges that belong to L'_2 but do not belong to L'_1, i.e., the elements specific to L'_2, which are *etrm:EstablishTunnelRequest*, *cr:Connector*, *oldM:MobileIOR*, *hla:HomeLocationAgent*, *o:Object*, *cn1:Connection*, *n1:Node*, *n2:Node*, *n3:Node* and the connected edges. As Step 4, the elements created after applying the previous rule r'_1 should be deleted inside L'_2, which is denoted as $lc_1 = R'_1 - L'_1$. *etrm:EstablishTunnelRequest* is the only node that is created after applying r'_1. Therefore, we should delete it and the connected edges in the result graph of $L'_1 \oplus L'_2$ (see Fig. 7.11), which includes now all the elements from L'_1, and the elements specific to L'_2, but without *etrm:EstablishTunnelRequest*.

Afterwards, we go on constructing $R'_1 \oplus R'_2$. We add all nodes and edges that belong to R'_2, which defines the postcondition of the rule r'_2 (see Step 5, Algorithm 7.1). Then we add, as Step 6, all nodes and edges that belong to R'_1 but do not belong to R'_2, i.e., the elements specific to R'_1. Since all the elements in R'_1 are already included in R'_2, we do not need to add anything then. As Step 7, the elements deleted after applying rule r'_2 should be deleted inside R'_1, which is denoted as $rd_2 = L'_2 - R'_2$. The result is *NULL*, and we do not need to delete any elements. Therefore, the result graph of $R'_1 \oplus R'_2$ includes all the elements from R'_2. The result graph of $L'_1 \oplus L'_2$ and $R'_1 \oplus R'_2$ is presented in Fig. 7.11.

Based on the result of $L'_1 \oplus L'_2$ and $R'_1 \oplus R'_2$, $L'_1 \oplus L'_2 \oplus L'_3$ and $R'_9 \oplus R'_8 \oplus R'_7$ can be further constructed. We go back to Step 2, and add all the elements that belong to $L'_1 \oplus L'_2$. As Step 3, the elements specific to

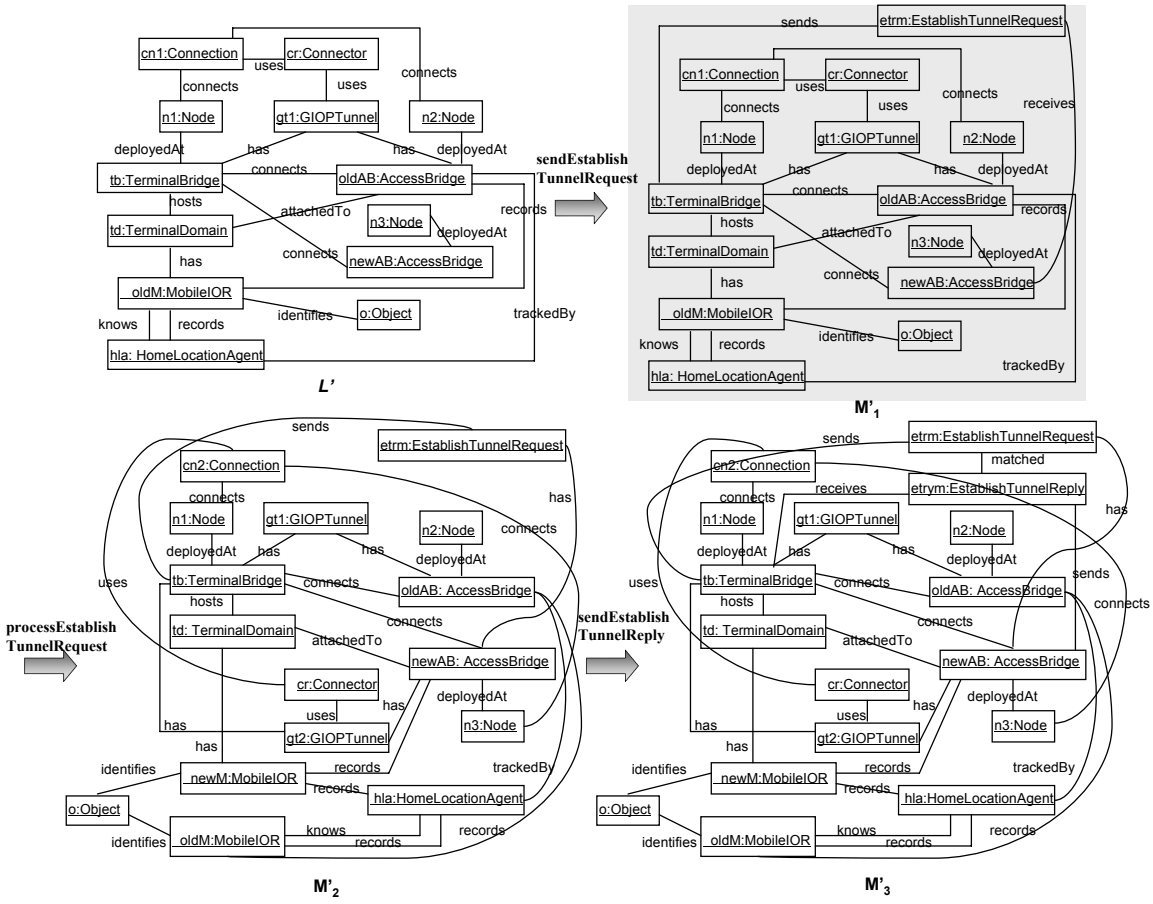

Figure 7.19: Construction of the middle graph M_i' using Algorithm 7.2 (Part 1)

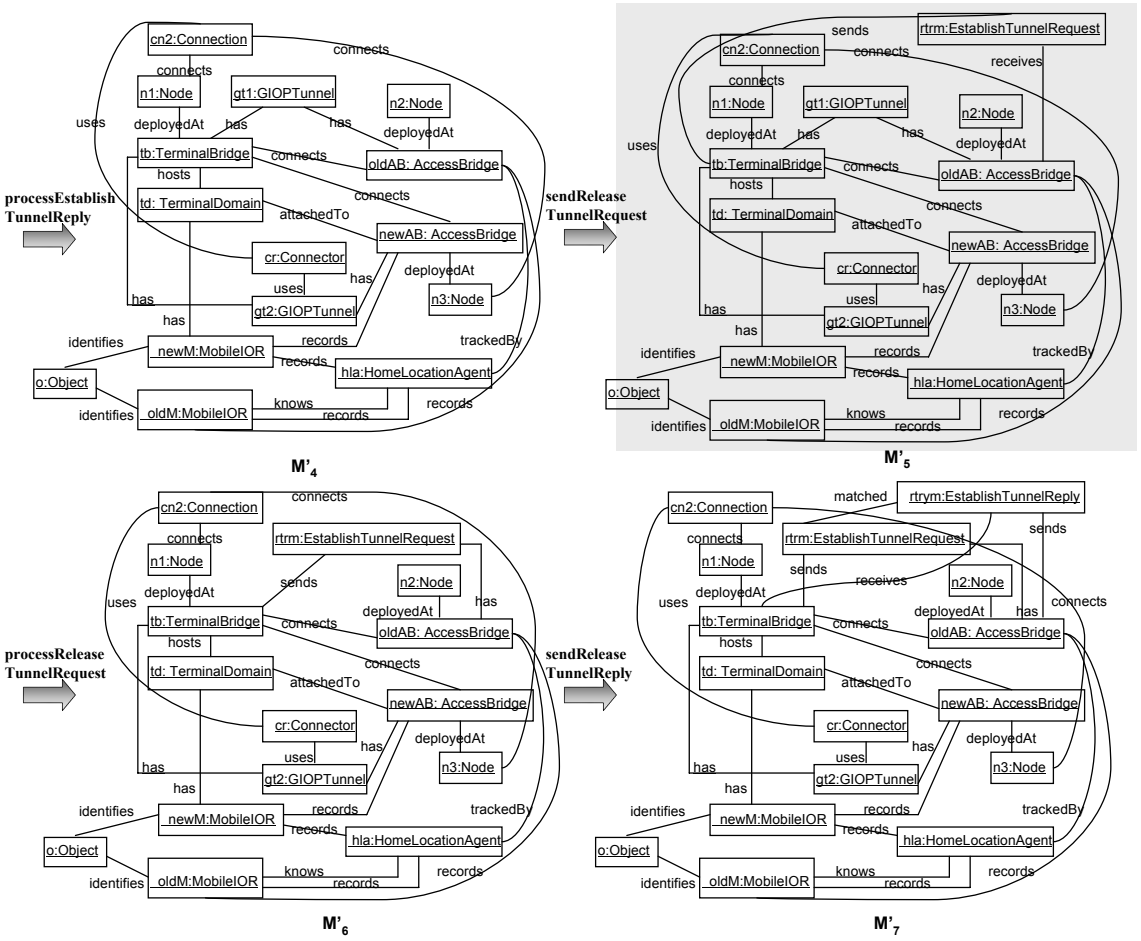

Figure 7.20: Construction of the middle graph M_i' using Algorithm 7.2 (Part 2)

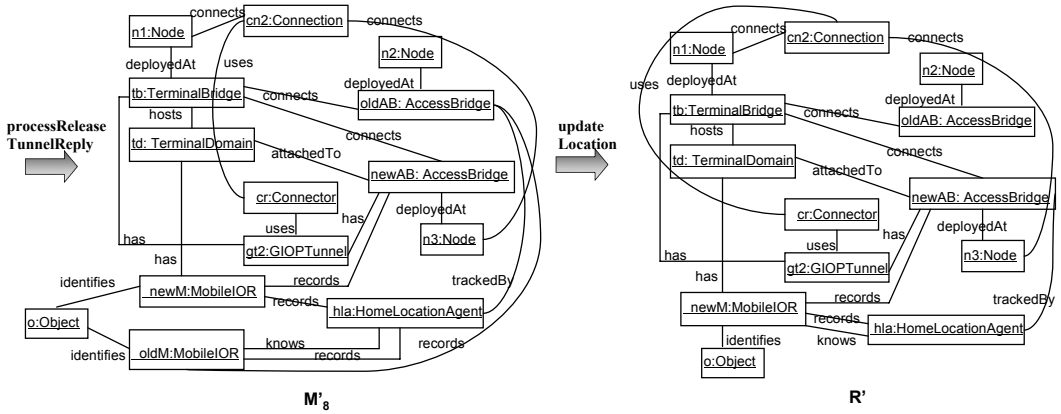

Figure 7.21: Construction of the middle graph M_i' using Algorithm 7.2 (Part 3)

L'_3 need to be added, which are *etrm:EstablishTunnelRequest, gt2:GIOPTunnel* and the connected edges (see Fig. 7.11). As Step 4, the elements that are created after applying the previous rule should be deleted inside L'_3, i.e., lc_2 needs to be deleted. The set lc_2 includes *etrm:EstablishTunnelRequest, gt2:GIOPTunnel* and the connected edges. The result graph of $L'_1 \oplus L'_2 \oplus L'_3$ is shown in Fig. 7.12, which is the same as the result graph of $L'_1 \oplus L'_2$, since the elements specific to L'_3 can be created in the previous rules r'_2 and r'_1.

We go to Step 5, and add all the elements that belong to $R'_1 \oplus R'_2$. As Step 6, the elements specific to R'_3 need to be added, which are *cn1:Connection, cr:Connector, oldM:MobileIOR, newM:MobileIOR, hla:HomeLocationAgent, o:Object, n1:Node, n2:Node, n3:Node* and the connected edges (see Fig. 7.11). As Step 7, the elements that are deleted after applying the rule r'_3 should be deleted inside R'_7, i.e., rd_3 needs to be deleted. The set rd_3 is *NULL*. The result graph of $R'_1 \oplus R'_2 \oplus R'_3$ is shown in Fig. 7.12.

Similarly, we can construct the result of $L'_1 \oplus L'_2 \ldots \oplus L'_4$ (see Fig. 7.13), which is once more the same as the one of $L'_1 \oplus L'_2 \oplus L'_3$, since the elements specific to L'_4 can be created in the previous rule (see Fig. 7.12). The result graph of $R'_1 \oplus L'_2 \ldots \oplus L'_4$ is shown in Fig. 7.13. This time, the set rd_4 includes *etrm:EstablishTunnelRequest* and *etrym:EstablishTunnelReply*, which also belong to the set lc_3. Therefore, only the elements *cn1:Connection, cr:Connector, oldM:MobileIOR, newM:MobileIOR, hla:HomeLocationAgent, o:Object, n1:Node, n2:Node, n3:Node* from $L'_1 \oplus L'_2 \oplus L'_3$ will be added to R'_4, but ont the elements belonging to rd_4.

Repeating the algorithm, we get the same result graph for $L'_1 \oplus L'_2 \ldots \oplus L'_5$, $R'_1 \oplus R'_2 \ldots \oplus R'_5$, and till $L'_1 \oplus L'_2 \ldots \oplus L'_9$, $R'_1 \oplus R'_2 \ldots \oplus R'_9$. Since the elements specific to $L'_5, L'_6, \ldots, L'_9$ can be created in the previous rule, the result of $L'_1 \oplus L'_2 \ldots \oplus L'_5$ to $L'_1 \oplus L'_2 \ldots \oplus L'_9$ remains the same as $L'_1 \oplus L'_2 \ldots \oplus L'_4$. The result graph and the graphs for calculating are presented in Fig. 7.13 to Fig. 7.18 respectively.

Based on the derived L', the intermediate graph M'_i can be got by applying the rule r'_i (see Algorithm 7.2). We show M'_i in Fig 7.8, Fig 7.9, Fig 7.24. By applying the rule one by one on L', we can check whether the scenario (given by the sequence of concrete rules) is reachable, i.e., whether M'_i and R' are reachable through applying the fixed rule sequence.

Afterwards, we can derive a new concrete rule r' *handOverWCORBA* (see Section 6.5) whose precondition is $L' = L'_1 \oplus L'_2 \ldots \oplus L'_9$ and whose postcondition is $R' = R'_1 \oplus R'_2 \ldots \oplus R'_9$, i.e., $r' : L' \Rightarrow R'$. The rule is a simplified version of the refined sequence of rule, which preserves the semantics consistency. Therefore, it can be used to substitute the complicated rule sequence. It can then simplify the rule sequence and enhance understandability of the model. We can use it to check whether the refinement is correct. We can start with L', and check whether R' is reachable by applying the rule sequence.

Correctness Check

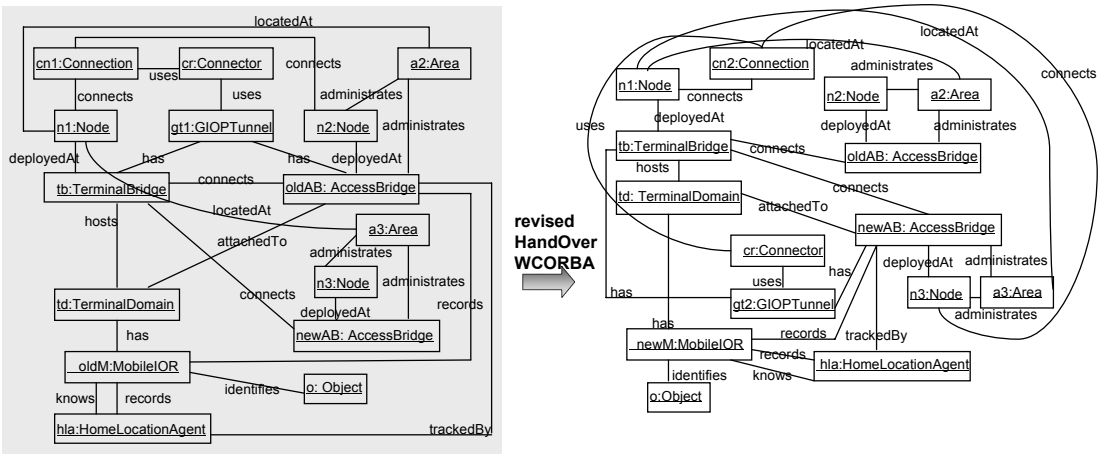

Figure 7.22: The revised rule of *HandOverWCORBA*

The defined concrete rules are semantically consistent since the scenario is realizable in the concrete rules. However, if we use the refinement correctness criteria (see Def. 7.5) to check whether the rule *handOverWCORBA* is a correct refinement of the more abstract rule *handOverConcrete* (see Fig. 7.8), we can conclude that they are not correct refinement since L' does not refine L and R' does no refine R, which means, L' does not contain equivalent elements (i.e., *a2:Area*, *a3:Area* and the connected edges) for all elements of L, and R' does not contain equivalent elements (i.e., *a2:Area*, *a3:Area* and the connected edges) for all elements of R (see Def. 7.2). The inconsistency is caused because the elements *a2:Area* and *a3:Area* are needed in the abstract rule *handOverConcrete* in order to present the physical location of the component clearly, although they are not changed after executing the rule. However, the physical location of the component is implicitly defined in the concrete rule *handOverWCORBA* through the definition of *TerminalBridge* and *AccessBridge*. Therefore, we did not include these elements in the concrete rules in order to keep simplicity.

Inconsistency Recovery

The inconsistency can be recovered easily by adding the needed elements in the rules since the elements are not changed after executing the rule. Therefore, we get a new rule *revisedHandOverWCORBA* (Fig. 7.22) after we add *a2:Area*, *a3:Area* and the connected edges, which is now consistent with the rule *handOverConcrete* in the conceptual style. Correspondingly, we can easily achieve consistency of the refined concrete rules by adding the elements to precondition and postcondition of the second rule *precessEstablishTunnelRequest*. The L' and R' of the modified rules are now correct refinement of L and R respectively.

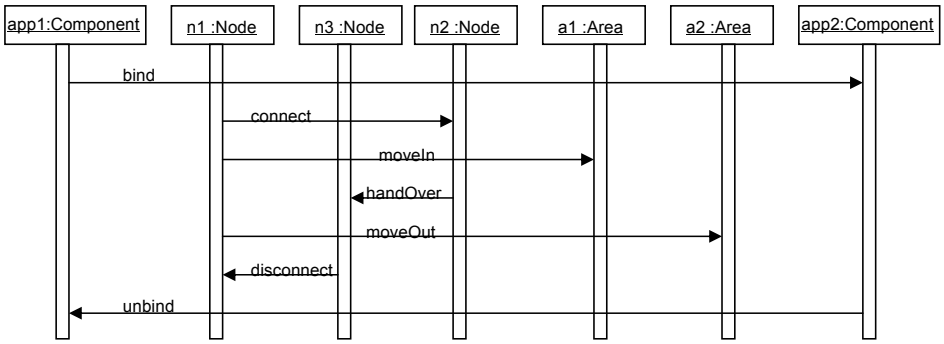

Figure 7.23: The abstract scenario in the conceptual style

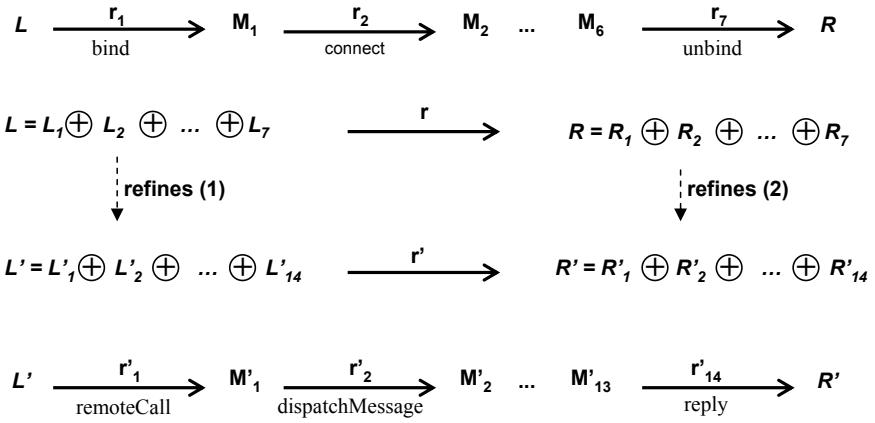

Figure 7.24: Refinement of a scenario-mapped rule

7.4.2.3 Refinement of the scenario-mapped rule

The refinement of the scenario-mapped rule is about mapping an abstract scenario to a concrete one, i.e., a fixed sequence of abstract rules is mapped to a fixed sequence of concrete rules. In such a refinement, the completely different concrete rules are associated with the abstract rules through a scenario construction, since they are added in order to realize a specific abstract scenario. In order to construct the refinement relationship of such rules, we need to construct the scenario at first, which is done by arranging the rules in the UML sequence diagram. We take the mapping of the rules *request, callReturn, reply, dispatchMessage, sendMessage* in the platform-independent concrete style as an example, which can not be mapped directly to any rules in the conceptual style. However, we can associate them with the rules in the conceptual style through the scenario given in the sequence diagram of Fig. 6.11 or of Fig. 6.12. We take the one in Fig. 6.11 as an example. Correspondingly, we need to construct an abstract scenario in the conceptual style, which is done through arranging the equivalent abstract rules using the UML sequence diagram (see Fig. 7.23).

Afterwards, we get two equivalent sequence diagrams specified in the abstract style and the concrete style respectively. Following the same idea as the refinement of the sequence-mapped rule, we construct the initial graph L and L' for the abstract scenario and concrete scenario respectively, each of which contains the needed precondition of the rules (Fig. 7.24). The rules are applied then one by one following the sequence. We denote them as $L \overset{r_1}{\Longrightarrow} M_1 \overset{r_2}{\Longrightarrow} M_2 \ldots M_{m-1} \overset{r_m}{\Longrightarrow} R$ in the abstract style, and $L' \overset{r'_1}{\Longrightarrow} M'_1 \overset{r'_2}{\Longrightarrow} M'_2 \ldots M'_{n-1} \overset{r'_n}{\Longrightarrow} R'$ in the concrete style, where M_i and M'_j are the created intermediate graph, R and R' are the last graph that contains the result (i.e., postcondition) after applying the last rule. Afterwards, we derive a new abstract rule $r : L \Rightarrow R$ and a new concrete rule $r' : L' \Rightarrow R'$, each of which is a simplified version of the refined sequence of abstract / concrete rules respectively. The rule r transforms the overall precondition L to the overall postcondition R. The rule r' transforms the overall precondition L' to the overall postcondition R'. As explained in last section, the new rules preserve semantics of the sequence of rules since they are based on a scenario construction. Therefore, they can be used to substitute the complicated rule sequence.

We use the derived rule $r : L \Rightarrow R$ and $r' : L' \Rightarrow R'$, the intermediate graph $M_1, M_2, \ldots M_{m-1}$ and $M'_1, M'_2, \ldots M'_{n-1}$ to check whether the refinement is correct. Starting with L, we check at first whether M_i, R are reachable by applying the fixed rule sequence $s = r_1, r_2, ..., r_m$. Afterwards, we start with L', and check whether M'_j, R' are reachable by applying the fixed rule sequence $s' = r'_1, r'_2, ..., r'_n$. We define the correctness criteria of the refinement as:

Def. 7.6 For a scenario-mapped rule with a sequence $s = r_1, r_2, ..., r_m$ specified in the abstract transformation system $\mathcal{G}$, the concrete scenario with a sequence $s' = r'_1, r'_2, ..., r'_n$ in the concrete transformation system $\mathcal{G}'$ is a correct refinement, if L' refines L and if R' refines R, i.e., $abs_t(L') =$

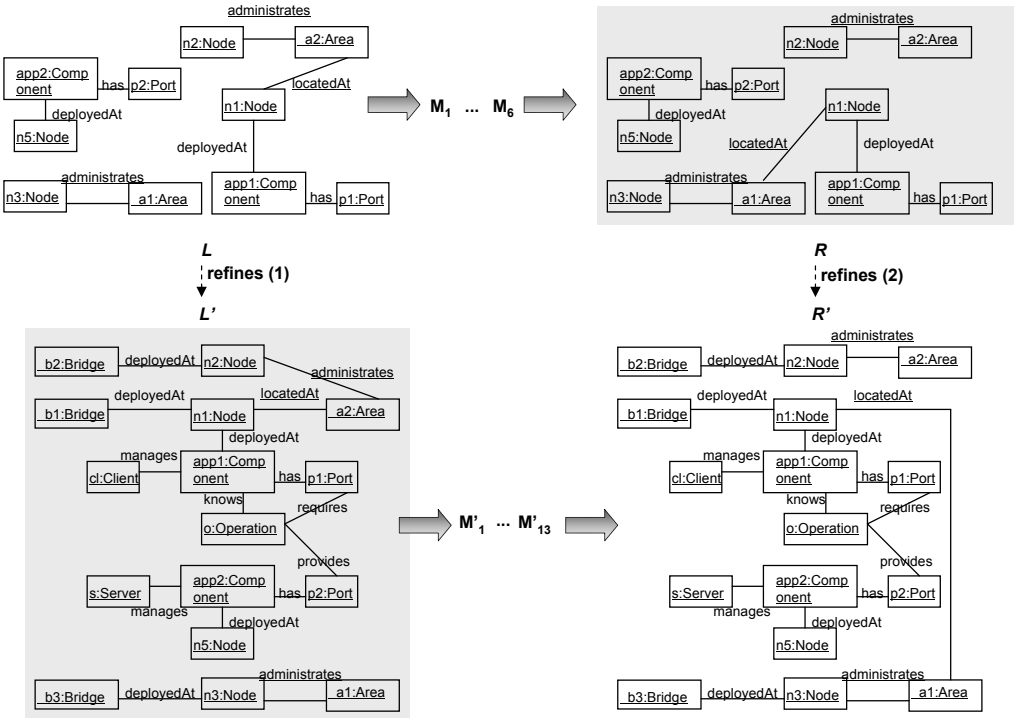

Figure 7.25: An example of the scenario-mapped rule

$L \wedge abs_t(R') = R$, and if $L \stackrel{r_1}{\Longrightarrow} M_1 \stackrel{r_2}{\Longrightarrow} M_2 \ldots M_{m-1} \stackrel{r_m}{\Longrightarrow} R$ and if $L' \stackrel{r'_1}{\Longrightarrow} M'_1 \stackrel{r'_2}{\Longrightarrow} M'_2 \ldots M'_{n-1} \stackrel{r'_n}{\Longrightarrow} R'$, where $L = L_1 \oplus L_2 \oplus, \ldots, \oplus L_m$, $R = R_1 \oplus R_2 \oplus, \ldots, \oplus R_m$, $M_i = R_i + (L - L_i)$, $r_i : L_i \Rightarrow R_i$, $L' = L'_1 \oplus L'_2 \oplus, \ldots, \oplus L'_n$, $R' = R'_1 \oplus R'_2 \oplus, \ldots, \oplus R'_n$, $M'_j = R'_j + (L' - L'_j)$, and $r'_j : L'_j \Rightarrow R'_j$, the symbol $\oplus$ is used to indicate combination of preconditions or postconditions (see Algorithm 7.1) of the rules.

The refinement criteria build a bi-directional mapping between the sequence of abstract rules and the sequence of concrete rules via the derived simplified rules. Therefore, the required *behavioral-preservation* is ensured since the sequence of abstract behaviors has a corresponding sequence of concrete behaviors. The required *behavioral-constraints* is also ensured since the sequence of concrete behaviors has a corresponding sequence of abstract behaviors. We associate the completely different concrete rules with the abstract rules through constructing the overall precondition and postcondition of the rule sequence. It allows us to ensure the correctness of the completely different concrete behaviors. Besides, we can also check the correctness of the rule sequence for both of the abstract rules and the concrete rules.

The process of creating L, R, L', R' and M_m, M'_n is the same as the one presented in the refinement of the sequence-mapped rule (see Algorithm 7.1, 7.2). We present only the L, R, L', R' in Fig. 7.25. The refined concrete rules are

correct refinement since the instance graph R' is a correct refinement of R, and L' is a correct refinement of L, and $L \stackrel{r_1}{\Longrightarrow} M_1 \stackrel{r_2}{\Longrightarrow} M_2 \ ... \ M_6 \stackrel{r_7}{\Longrightarrow} R$, and $L' \stackrel{r'_1}{\Longrightarrow} M'_1 \stackrel{r'_2}{\Longrightarrow} M'_2 \ ... \ M'_{13} \stackrel{r'_{14}}{\Longrightarrow} R'$, according to the correctness criteria defined in Def. 7.6.

7.4.2.4 Behavioral refinement

Based on the definition of the refinement for the different rules, the overall refinement criteria of the behavioral part is given as:

Def. 7.7 The set of graph transformation rules $\mathcal{R}'$ for a concrete transformation system $\mathcal{G}'$ is a refinement of a set of graph transformation rules $\mathcal{R}$ for an abstract transformation system G,

- if for all the direct-mapped and single-mapped rule $r : L \Rightarrow R$ in the $\mathcal{R}$, there exists a refined concrete rule $r' : L' \Rightarrow R'$ in the $\mathcal{R}'$;
- and if for all the sequence mapped rule $r : L \Rightarrow R$ in the $\mathcal{R}$, there exists a refined concrete rule sequence $s' = r'_1, r'_2, ..., r'_n$ in the $\mathcal{R}'$;
- and if for all the scenario-mapped rules with a sequence $s = r_1, r_2, ..., r_m$ in the $\mathcal{R}$, there exists a refined concrete scenario in the $\mathcal{R}'$ with a sequence $s' = r'_1, r'_2, ..., r'_n$;
- and if all the concrete rules in the $\mathcal{R}'$ are covered in the previous refinement relationship.

We build three kinds of refinement criteria for different rules: the single-mapped rule, the sequence mapped rule and the scenario-mapped rule. The criteria construct bi-directional mappings between the abstract rule and the corresponding concrete one(s). That is, an abstract rule is mapped to a concrete rule, an abstract rule is mapped to a fixed sequence of concrete rule, a fixed sequence of abstract rule is mapped to a fixed sequence of concrete rule. Therefore, the required three kinds of *behavioral-preservation* are ensured since the abstract behavior has a corresponding concrete behavior. The required *behavioral constraint* is also ensured since every concrete behavior has a corresponding abstract behavior.

7.4.3 Refinement of the TGTS - based style

We can now define the style refinement criteria after explaining the structural refinement and behavioral refinement:

Def. 7.8 A concrete style $\mathcal{G}' = \langle TG', C', \mathcal{R}' \rangle$ is a refinement of an abstract architectural style $\mathcal{G} = \langle TG, C, \mathcal{R} \rangle$, if the concrete type graph TG' is a *structural-preserved* refinement of the abstract graph TG and if the set of abstract graph transformation rules $\mathcal{R}'$ is a refinement of the set of concrete graph transformation rules $\mathcal{R}$ and if all elements of TG' are used in the $\mathcal{R}'$.

The structural refinement criteria satisfy the required *structural-preservation* since every abstract structural element has a corresponding concrete one through structural mappings. The *behavioral-preservation* and *behavioral constraint* are also ensured by constructing bi-directional mappings between the abstract rule and the corresponding concrete one(s). As having mentioned before, we can not map the concrete-specific structural elements to the abstract ones in the structural mappings. The required *structural-constraint* of the concrete-specific structural elements is ensured during the rule mappings since they are used in the rules in order to realize a specific functionality, however, still partially. We still need to add one criterion for this, that is, all the concrete structural elements should be used in the concrete rules. By doing so, we can ensure that all concrete structural elements are constructed in order to realize a specific abstract task or functionality. It is not allowed to have structural elements that are irrelevant to the functionality. Finally, the required *structural-constraint* is fulfilled.

7.5 Evaluation and comparison

As concluded in the previous section, our refinement approach preserves all the required properties. The refinement criteria include both structural ones and behavioral ones. Through structural mappings that can map the abstract elements to the concrete ones, the structural refinement criteria satisfy the required *structural-preservation* since every abstract structural element has a corresponding concrete one. However, the required *structural-constraint* is only partially satisfied since we can not map the newly added, concrete-specific structural elements to the abstract ones. We have further made this in the behavioral refinement. By constructing bi-directional mappings between the abstract rule and the corresponding concrete one(s), the behavioral refinement criteria satisfy the *behavioral-preservation* and *behavioral constraint*. We can partially ensure the *structural-constraint* during the rule mappings since the concrete-specific structural elements are used in the rules in order realize a specific functionality. By adding the last criterion, that is, all concrete structural elements should be used in the concrete rules, the required *structural-constraint* is finally completely fulfilled.

Since our approach is mainly based on the mapping of the rules, we call it *rule mapping-based refinement*. This differentiates our approach from that proposed in [145, 15], which also investigates refinement for the styles specified using graph transformation systems. However, they do not define fixed mappings between rules, but only between structural elements and transformation states. Our refinement criteria enable us to check the correctness of newly added elements through a scenario construction. We construct the overall precondition and postcondition of the rule sequence, and then check whether the scenario is correct through checking if the postcondition is reachable with a given precondition. It allows then correctness check of the completely difference concrete behaviors and structural elements. It also allows the correctness check of the rule sequence.

Our refinement between an abstract architectural style and a concrete style is style-based. That means, it is defined between two architectural styles at the meta-level rather than between individual architectures. This achieves required *reusability* since the relationship can be applied to any instances of the styles.

Some researchers [145, 15] construct consistency checks between two graph transformation systems using model-checking theory, which is achieved through checking whether all the states in one transition system are reachable in another transition system. Therefore, they need to explore and compare all the possible states between two graph transformation systems, which makes the approach inefficient and impractical to large systems because the state space is too large to be explored. In order to solve the state explosion problem, they also introduce test theory based consistency check. They select a limited number of traces from one transition system as test cases, and check whether they are reachable in another transition system. The difficulty of the approach is that they cannot derive exactly the needed states for test cases, although they try to use a predicate function to help to do so. Therefore, their checking is again transferred to the reachability analysis problem in the model checking theory, but with limited definition of states.

Compared to that, our approach allows us to construct the needed transformation state graphs and transformation sequence for correctness checking. Therefore, we do not need to explore and compare all the possible transition states between two graph transformation systems. This makes our approach more efficient and practical to large systems. It also enables us to use the existing graph transformation simulation tool Fujaba [1] as the basis for automation. The required *tool support* is also satisfied. We can check whether the abstract transformation sequence and the concrete transformation sequence are reachable. This is especially helpful to check the correctness of the newly added architectural elements. The detail explanation of Fujaba will be given in the next chapter.

Chapter 8

Style Simulation and Tools

8.1 Overview

Up to now, we have introduced how to specify the architectural style using Typed Graph Transformation Systems (TGTSs) and how to refine an abstract style to a concrete one. One of the key issues in model-based development, like in all other engineering areas, is to ensure that the product delivered meets its requirements. In our case, it is important to ensure that the architectures in a same hierarchy belong to the same family of middleware, i.e., they should realize and provide the same functionalities, although they belong to different abstraction layers and use different design strategies. However, this is difficult to achieve since the model is often complicatedly constructed and thus difficult to be checked, especially manually.

In order to solve the problem, we construct a framework that checks whether the specified architectures on different abstraction layers provide the required functionalities (see Fig. 8.1), with the help of a standard reference application. The application covers the requirement and it is used to validate the functionality completeness of the architectures. It invokes and uses the Application Programming Interfaces (APIs) provided by each style, namely, Concrete API and Conceptual API. From the execution result we can check and compare whether the architectures belong to the same family of middleware. Specifically, we develop the framework based on the simulation technique with tool support. Having a formal semantics, the typed graph transformation system enables us to analyze properties of the constructed model through simulation. Simulating the dynamic behavior of the system allows the designers to execute the system and to play with specific scenarios. The designers can concentrate on the key aspects of the architecture. It can also detect errors and improve the confidence of the model.

The style describes a specific kind of software architecture with interesting aspects like mobility, middleware platform, which bring new possibilities of applying simulation. Besides the mentioned behavioral consistency check, we will develop other two ways to use simulation: to validate the model efficiently, and to automate the refinement consistency check. In addition, we also focus on providing a practical and usable process and environment to help the design and development

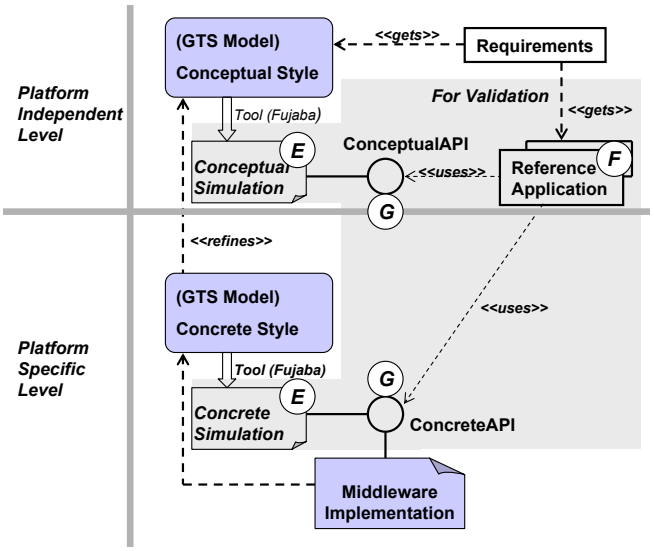

Figure 8.1: The simulation framework for behavioral consistency check

of the style. The tool support of style specification, style refinement and other related concept is very important too. After comparing the available simulation tools, we decide to choose the existing Fujaba simulation environment as our basis [65]. Fujaba is a CASE (Computer-Aided Software Engineering) tool that aims to push graph transformation systems to a broader industrial usage. Fujaba allows a seamless integration of object-oriented software development and graph transformation systems, which facilitates greatly the designers and developers who are not familiar with graph transformation systems. With its support, we develop a style - based engineering process for style development, which includes style specification, style validation, refinement consistency check, behavioral consistency check and code generation.

The chapter is organized as follows: After giving our requirements for the tool, we introduce and compare the available TGTS-based simulation tools in Section 8.2. In Section 8.3, we explain at first how to use Fujaba environment to specify and simulate the style. Afterwards, we illustrate the three ways to use Fujaba simulation for further style-based analysis and automation. We also conclude the style - based engineering process for style development.

8.2 Graph transformation simulation tools

There exist various graph transformation system based analysis methods and tools [47], which differ in various concepts, techniques and focuses. To choose which one to use depends on the requirements and purposes of the users. We list at first some requirements in order to judge their suitability for style specification, refinement, simulation and behavioral consistency check, etc. Afterwards, we will evaluate and

compare three possibly suitable simulation tools, which are Fujaba, PROGRES and AGG. Finally, we choose the Fujaba simulation environment as the basis for further style-based analysis and development.

8.2.1 Requirements for the tool

1. Style specification The tool should allow us to input and edit the full style specification, i.e., the type graph and the graph transformation rules. More specifically, we specify the type graph using UML class diagrams, and the rules at the instance level using object diagrams (see Section 5.3). Therefore, it is necessary that the tool supports the definition of related concepts like *classes*, *attributes*, *cardinalities multi inheritance*, *associations*, *aggregation*, etc. The best would be an UML class diagram editor that supports direct class diagram notations. The definition of *a single graph transformation rule* by its left- and right-hand sides should be supported. It is helpful if the tool supports complicated *programmed graph transformations*, since it allows users to specify advanced control flow with user-defined order. It provides the possibility to combine a set of rules to one complicated rule.

2. Type check / static semantics analysis When specifying and applying rules, we need to ensure that the instance graphs are correctly typed over the type graphs. It is quite error-prone when designers check the correctness of rules by hand for a large system. Many errors are caused because of incorrect typing of the object diagram to the class diagram. For instance, it happens often that *cardinalities*, *associations* and *inheritance* of instance graphs are not correctly specified according to the class diagram. Consequently, the capability of UML class diagram-based static semantics analysis (see Section 5.3) is significantly important. It decides how easily the users can develop the style.

3. Simulation mode Having a formal semantics, the typed graph transformation system enables us to analyze dynamic behavior of the constructed model through simulation. Our basic requirement for simulation is that it should support the execution of the rules with a fixed sequence, which is required for the specification of component interaction (see Section 5.3). Further, we require two different modes to execute the rule:

1. The *interactive mode* that allows users to decide certain transformation sequences by user-specific rule selections for every transformation step. This allows users to execute one rule at a time. Therefore, the rules in a fixed sequence will be executed one after another with users' interaction.
2. The *automatic mode* that applies a sequence of rules that are pre-defined by the users.

4. Refinement In order to enable automated refinement checks between two graph transformation systems as described in Chapter 7, we require a tool that should:

1. Allow us to specify an *abstraction function* between two type graphs (see Section 7.4).
2. Enable *structural refinement checks* for a given pair of instance graphs based on the abstraction function (see Section 7.4).
3. Implement the Algorithm 7.1 and 7.2 for *constructing the derived graphs* i.e., the overall precondition L', the overall postcondition R', and the intermediate graph M_i' for refinement consistency check, given a fixed sequence of rules (see Section 7.4).
4. Allow *reachability check* for the constructed L', R' and M_i', by applying the fixed rule sequence (see Section 7.4). Normally, the reachability check can be accomplished by GTS-based model checking and simulation techniques.

5. Behavioral consistency check In order to automate behavioral consistency checks between two architectures as outlined in the above overview, we require a tool that should:

1. Allow us to define the *standard application* for validation, which includes a sequence of operations.
2. Allow us to specify the *APIs* for the styles. The APIs should provide the needed operations that will be invoked and used by the standard application.
3. Enable the *invocation and execution* of the operations. One basic requirement is that it should support the execution of the rules with a fixed sequence. The best would be that it supports both interactive and automatic mode of execution.
4. Allow us to check and *compare the execution result* of two systems. A very basic requirement is that the execution result should be easily readable and understandable.

6. Style - based development We focus on providing a practical and usable process and environment to help the design and development of the style. Usability, understandability, *code generation*, *prototype generation*, *reverse engineering*, and other aspects of the tool are very important in order to attract software designers and developers to use the approach. For instance, seamless integration of graph transformation systems with common (Object Oriented) design and implementation languages like UML, C++ or Java is important. Especially, using *UML-like notations* to edit the rules can help a lot, since the graph grammar notations are too proprietary and difficult to use for users who are not familiar with them.

The above listed requirements show that we mainly focus on the practical usage of the tool to help the design and development of the style. We do not consider other more theoretical aspects of graph transformation theory, such as detailed graph data models, transformation productions, etc. Correspondingly, we will evaluate next three tools that are famous for promoting the practical usage of graph transformation systems as a design and implementation means. The evaluation only reflects their suitability for our specific requirements.

8.2.2 AGG

The AGG (Attributed Graph Grammar) system [13] aims at the specification and prototypical implementation of applications with complex graph-structured data. It's underlying graph model is the graph structure of labeled and attributed hierarchical graphs. For the user interface, typed and attributed directed graphs are chosen as graph data model. The editor of transformation rules consists of left-hand side and right-hand side. Although AGG supports the modeling of classes, attributes and cardinalities, it does not provide static semantics analysis of cardinalities, inheritance, associations, etc. Therefore, the users need to check themselves whether the rules are correctly typed over the UML class diagrams.

We can simulate a graph transformation system using AGG's analysis techniques. AGG supports two different modes to execute the transformation rule. The interactive mode allows the user to select a certain rule and to determine a certain matching of its left-hand side in the host graph. The second mode, the automatic mode, is more sophisticated. It applies not only one rule at a time but also a sequence of rules that are pre-defined by the users. Each rule is applied as often as possible, until no more match for this rule can be found. Then, the next rule is tried and applied if possible. The graph transformation stops when all rules have been applied in the given order as often as possible.

The researchers in the graph transformation area have been made effort to enhance the practical usage of graph transformation systems as a design and implementation means [22]. For this purpose, the AGG system was extended by a sophisticated graph pattern matching algorithm for the automatic execution of AGG rules. Besides, AGG combines graph transformations with the object-oriented programming language Java such that AGG may be used as a general purpose graph transformation engine in high-level Java applications.

AGG does not support refinement check between two graph transformation systems. However, reachability check can be accomplished by the simulator. The behavioral consistency check is not supported by AGG neither. Again, the invocation and execution can be performed by the simulator.

8.2.3 PROGRES

The PROgrammed Graph REplacement System PROGRES is developed to mechanize the development of integrated programming support environments by graph

transformation systems [130]. The main goal of the environment is to provide tools for various phases of a software life cycle from analysis over design to programming together with accompanying task like configuration and project management. PROGRES is based on the data model of Entity Relationship diagrams (Graph Schema). It supports the modeling of classes, attributes, cardinalities, and single inheritance. PROGRES offers the concept of PROgrammed Graph REplacement Systems for the description of complex graph transformations. It's semantics is formally defined based on the rather general formalism of programmed graph or structure replacement system. It does not only offer imperative deterministic control structures for programming complex graph transformations, but allows one to construct them using a Prolog-like depth first search and backtracking programming style.

PROGRES provides quite complete support of static semantics analysis. For example, PROGRES comprises hundreds of static semantics rules that check consistency of a specification. The inheritance rules ensure the correctness of multiple inheritance. The cardinality qualifier rules ensure that the defined graph patterns do not contain obvious violations of edge cardinality constraints, etc. Type checking rules are about the actual operation parameters are checked against their formal counterparts, assignments are checked for type equivalence, graph patters and path expressions are checked against graph schema, etc.

Besides allowing stepping through the specification, which is similar to the interactive mode in AGG, PROGRES interpreter also executes pre-defined order of rules that is defined by explicit control flow constructs. These constructs include a nondeterministic choice operator which can be used for random rule selections. The interpreter computes possible matchings and randomly applies rules to the current host graph. When the program execution runs into a dead end where none of the rules can be applied any more, the interpreter initiates a back-tracking mechanism which undoes previous rule applications until an alternative execution path has been found.

Besides C, Modula-2, tcl/ tk code generation, the PROGRES environment offers means for the rapid prototyping of applications from their graph grammar specification.

The same as AGG, PROGRES does not support the concept of refinement analysis and behavioral consistency check between two graph transformation systems.

8.2.4 Fujaba

Although the above mentioned tools AGG and PROGRES have been improved a lot in usability, understandability, etc., there were still major problems to overcome in order to attract software designers and developers to the usage of graph rewrite systems. Two of the main problems are, that (1) the graph grammar notations are too proprietary and (2) that there exists no seamless integration of graph rewrite systems with common (Object Oriented) design and implementation languages like UML and C++ or Java. In order to overcome these deficiencies, Fujaba

(From UML to Java And Back Again) environment was created. Indeed, Fujaba environment is a CASE (Computer-Aided Software Engineering) tool that aims to push graph grammars to a broader industrial usage [54]. Fujaba and the kernel Story Diagrams allow a seamless integration of object-oriented software development and graph transformation systems. Generally, Story Diagrams enhance object-oriented software development methods by appropriate means for modeling the evolution and dynamic behavior of complex object structures.

Story Diagrams adapt standard object-oriented modeling languages i.e. UML class diagrams, activity diagrams, and collaboration diagrams. The semantics of Story Diagrams is based on its predecessor PROGRES. Story diagrams adopt main features from PROGRES, e.g., directed attributed graphs, programmed graph transformations with parameterized rules. However, story diagrams enhance the data model of graphs towards the object-oriented data model, i.e., the UML class diagram. The PROGRES graph model is extended to support for ordered, sorted, and qualified associations and aggregations.

The transformation rules are executed according to the control flow the user has specified in the underlying story diagram. Since the rules are parameterized, the user can also use parameter values to determine a specific matching for the rule. The interpreter selects one of them non-deterministically if there are several possible matchings. Fujaba drops the backtracking mechanisms of PROGRES since extensive experiences have shown that it is seldom used. This enables Fujaba to translate Story Diagrams and Story Patterns into standard object-oriented Java code. Besides, the Fujaba environment provides a simulator, the dynamic object browser, which visualizes the current object graph of the designed system. It allows users to select a certain rule and to determine a certain matching of its left-hand side in the host graph, which is similar to the interactive mode of AGG.

In addition, Fujaba aims to provide round-trip engineering of the UML diagrams [54, 103]. It provides not only code generation but also the recovery of UML diagrams from Java code. One may analyze the application code, recover the corresponding UML diagrams, modify these diagrams, and generate new code into the remaining application code.

The same as AGG and PROGRES, Fujaba does not support the concept of refinement analysis and behavioral consistency check between two graph transformation systems.

8.2.5 Evaluation and comparison

We summarize the result of the introduced tools in Table 8.1. The tools have different data models. Although UML class diagram related concepts are roughly supported by each of the tools, Fujaba provides the best capability of class diagram specification with its class diagram editor. The users can specify class diagrams using class diagram notations. All the tools allow us to enter and edit graph transformation rules. AGG only supports the specification of a single graph transformation rule, while both PROGRES and Fujaba support complicated programmed graph

Required Aspects \ Tools		AGG	PROGRES	FUJABA
1. Style Specification	Type Definition	node and edge types	Entity Relationship diagrams (Graph Schema)	UML class diagrams
	Supported Data Modeling	1. classes 2. attributes 3. cardinalities	1. classes 2. attributes 3. cardinalities 4. single inheritance	1. classes 2. attributes 3. cardinalities 4. multi inheritance 5. qualified associations 6. aggregation
	Rule Definition	simple graph transformation	programmed graph transformations with control flow construction	programmed graph transformations with control flow construction (UML Activity Diagrams and Collaboration Diagrams)
2. Type Check (Static Semantics Analysis)		No	1. cardinalities 2. single inheritance	1. cardinalities 2. multi inheritance 3. associations 4. aggregations
3. Simulation Mode		1. interactive mode 2. automatic mode	1. interactive mode	1. interactive mode
4. Refinement	Abstraction Function	No	No	No
	Structural Refinement Checks	No	No	No
	Derived Graph Construction	No	No	No
	Reachability Check	Yes	Yes	Yes
5. Behavioral Consistency Check	Standard Application	No	No	No
	API Specification	No	No	No
	Invocation &Execution	Yes	Yes	Yes
	Execution Result Comparison	No	No	No
6. Style – Based Development	Code Generator	Java	C, Modula-2, tcl/ tk	Java
	Integration of UML	No	No	UML Class Diagrams, Activity Diagrams and Collaboration Diagrams
	Prototype Generator	No	Yes	Yes
	Reverse Engineering	No	No	Yes

Table 8.1: Comparison of the GTS Simulation tools

transformations with control flow construction. Inside them, Fujaba provides the best support of static semantics analysis.

Regarding simulation, only AGG supports both interactive mode and automatic mode. Fujaba and PROGRES support only interactive mode. However, their complicated programmed graph transformation concept provides an alternative in some degree, which executes pre-defined order of rules that is defined by explicit control flow constructs. The users can therefore combine a set of rules to one complicated programmed rule.

Fujaba provides the best possibility of style-based development. It supports Java code generation, prototype generation, reverse engineering, etc. Besides, it is integrated with the well-known UML Class Diagrams, Activity Diagrams and Collaboration Diagrams, which greatly enhance the understandability and usability of the tool.

Concerning refinement, none of the tools supports our refinement concepts between two graph transformation systems. We can not find the definition of abstraction functions, structural refinement checks, or derived graph construction. However, each of the tools provides reachability check with the support of simulator, which at least allows us to automate the behavioral refinement consistency check.

Similarly, none of the tools provides the behavioral consistency check between two graph transformation systems. The concepts of standard application, API specification and execution result comparison are not supported. Nevertheless, the important part, the invocation and execution, is supported by the simulator. We can develop further automatic behavioral consistency check based on the simulator.

Based on the comparison and summary, we can conclude that Fujaba is the best choice for our further style based development and automation. It enables us to develop a practical process for the style development, e.g., specification, simulation, refinement consistency check, behavioral consistency check, code generation, etc. We will introduce in the next sections how we explore further style-based analysis and automation using the Fujaba simulation environment.

8.3 Style-based simulation

We will introduce at first how to use Fujaba (Version 4.1.0) to specify and simulate the style. Afterwards, we will explain three kinds of simulation-based analysis and automation: efficiently validating the style model, style refinement consistency check and behavioral consistency check. In the last section, we will conclude the style - based engineering process.

8.3.1 Style specification and simulation

The object-oriented CASE tool Fujaba (see Fig. 8.5) supports the specification of a graph transformation system using UML class diagrams and story diagrams, a combination of graph transformation rules and activity diagrams. Story diagrams

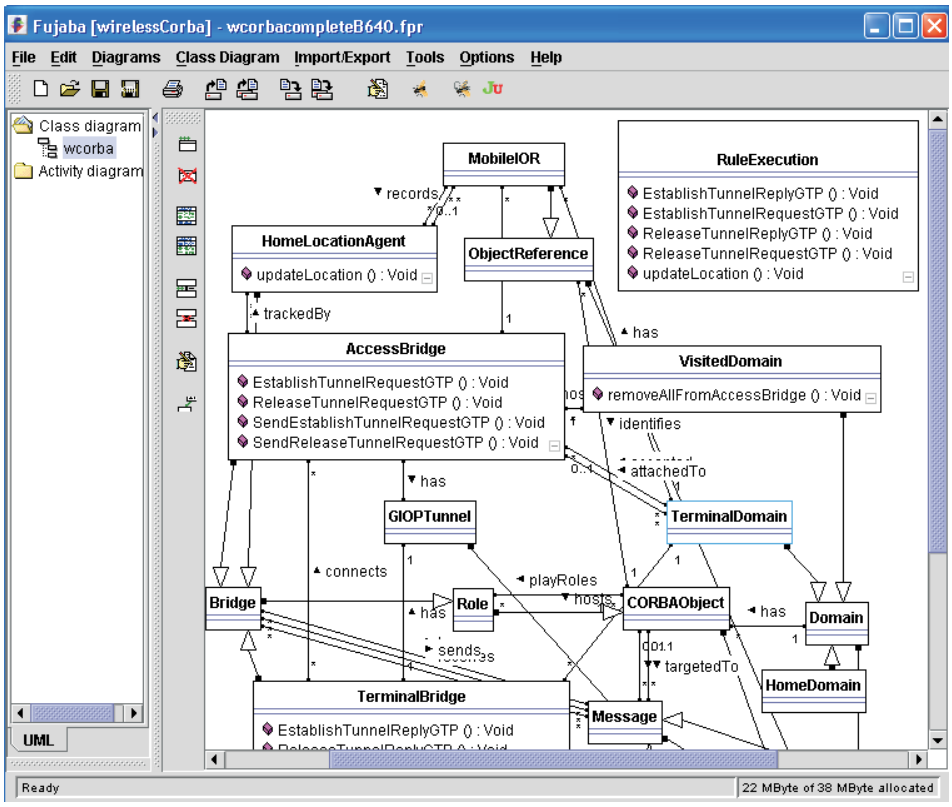

Figure 8.2: Screen shot of a type graph specified with Fujaba class diagram editor

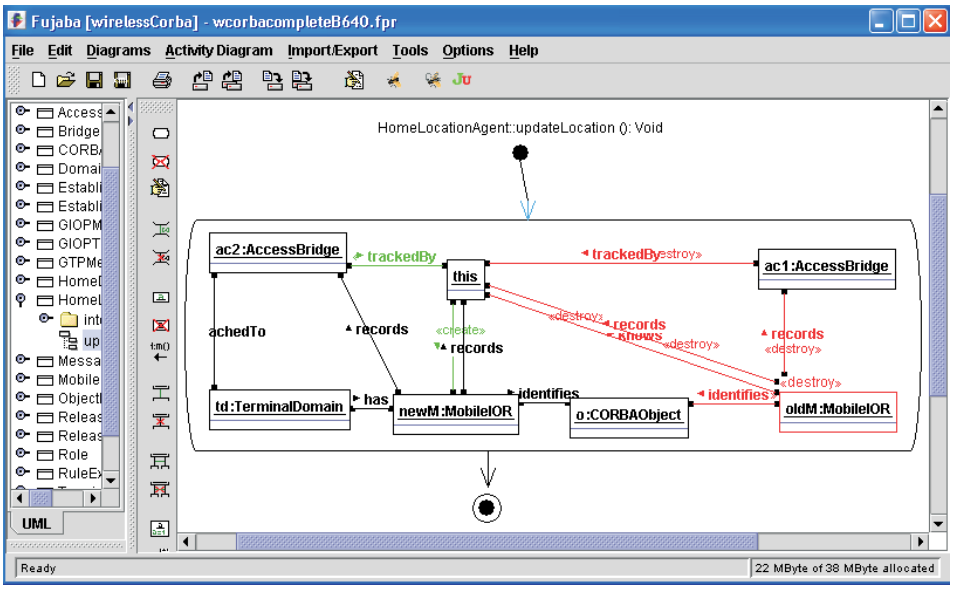

Figure 8.3: Screen shot of a transformation rule in the Fujaba Story diagram editor

adopt UML class diagrams for the specification of graph schemes, UML activity diagrams for the (graphical) representation of control structures, and UML collaboration diagrams as notation for graph transformation rules.

We use the previous specified styles (see Section 6.3, 6.4, 6.5) as examples to show how to specify the style using Fujaba. Taking as the first step, we use the Fujaba class diagram editor to specify the structural model. The editor allows us to create and modify classes, attributes, associations, cardinalities, multi-inheritance, etc. Figure 8.2 shows a snap shot of the type graph of the platform-specific concrete style (i.e., Wireless CORBA style) specified using the class diagram editor. We can assign methods to classes, which correspond to the graph transformation rules. For example, class *HomeLocatioAgent* contains one method *updateLocation*. For the non design-specific conceptual style that does not assign rules to specific classes, we can create a separate class, for instance, *RuleExecution*, whose functionality is to provide a means to activate required methods or rules.

Afterwards, we specify the graph transformation rules using the Fujaba story diagram editor, which adapts UML activity diagrams to represent explicit control flow graphically. Thus, the editor supports the definition of complicated programmed graph transformations. The basic structure of a Story Diagram consists of a number of so-called activities shown by big rectangles with rounded left- and right-hand sides. For convenient referring, the activities are numbered at their upper right corner. Activities are connected by transitions, that specify the execution sequence. Execution starts at the unique start activity represented by a filled circle. Execution proceeds following the outgoing transition(s). It allows users to specify advanced control flow with user-defined order. We can also combine a set

of rules to one complicated rule. Besides, it uses UML collaboration diagram notions to present left- and right-hand side of a rule within a single graph. We show in Fig. 8.3 a single graph transformation rule that contains one activity. As we can see, the green elements with ≪*creates*≫ represent nodes and edges to be added after executing the rule. The red elements with ≪*destroys*≫ are to be deleted. The other elements are black, which remain unchanged. During constructing the rules, the editor does already some static semantics check. It only allows us to create elements that are correctly typed over the class diagram. This helps designers very much to create a consistent and correct specification.

After finishing the style specification, we can generate automatically executable Java source code. The UML classes are translated to Java classes, while the rules are translated to Java methods [54, 103]. Besides class frames and method declarations, method bodies are also generated. We show partly the generated Java source code of rule *updateLocation* in Fig. 8.4, which is a Java method with method bodies that are corresponding to the dynamic behavior definition of the rule. The generated code does not require an extensive library and may be integrated seamlessly with other system parts and it is platform independent due to the usage of pure Java.

To observe the running system, Fujaba provides a simulator, Dynamic Object Browsing system (Dobs), which supports the stepwise execution of graph transformation rules through the Java Virtual Machine [57]. Dobs integrates a public domain Java source code debugger, the JSwat system, which allows using the dedicated functionality of the Java Debug Interface (JDI) library to analyze the object structure in the debuggee virtual machine. This enables Dobs to visualize the current object structure as the current host graph, while execution is frozen by JSwat. Dobs also allows executing a method (rule) graph at the method level, besides executing line by line at the source code level. This enables us to step through the execution of a system at the graph rewrite rule level of abstraction.

We show the process of doing simulation in Fig. 8.3. After executing the generated system as step 4 in Dobs, we need to input an initial object configuration of the system (as step 5). Afterwards, we can execute the system at the rule level and have a direct feeling how the system works. From the execution result of the rules in Dobs, we can also check whether the specified model reacts in the desired way. In case of errors, we can go back to the original graph, divide the rule into its atomic steps, and debug the models [57]. The code used for the simulation can be used inside the running system as well, since Fujaba simulates the system by generating source code out of the specification and observing the running system. Such an approach closes the gap between the simulated system and the software running on the real system [104].

Besides, Dobs also allows us to adapt the appearance of objects and to define more specific which objects should be visible and which should be hidden. For instance, we have shown in Fig. 8.6 the objects with all qualified associations, which can be hidden too to make the graph clearer, as shown in Fig. 8.7 and Fig. 8.8. We can use the standard user interface as starting point for the development of the final

```
   public void updateLocation()
{
   de.tu_bs.coobra.ObjectChangeCause cause = new
de.tu_bs.coobra.ObjectChangeStringCause("updateLocation()");
   de.tu_bs.coobra.ObjectChange.pushCause( cause );
   try

   {
      boolean fujaba__Success = false ;
      AccessBridge ac1 = null ;
      AccessBridge ac2 = null ;
      CORBAObject o = null ;
      Iterator fujaba__IterThisKnowsMIOROldM = null ;
      Iterator fujaba__IterThisRecordsMIORNewM = null ;
      MobileIOR newM = null ;
      MobileIOR oldM = null ;
      TerminalDomain td = null ;

      try
      {
         fujaba__Success = false ;

         // bind oldM: MobileIOR
         fujaba__IterThisKnowsMIOROldM = this.iteratorOfKnowsMIOR () ;
         while ( !(fujaba__Success) && fujaba__IterThisKnowsMIOROldM.hasNext ()
)
         {
            try
            {
               oldM = (MobileIOR) fujaba__IterThisKnowsMIOROldM.next () ;

               // check To-One-Link 'recordsHLA' between oldM and this
               JavaSDM.ensure ( (oldM.getRecordsHLA () != null) &&
oldM.getRecordsHLA ().equals (this) ) ;

               // bind ac1: AccessBridge
               ac1 = oldM.getRecordsAB () ;
               JavaSDM.ensure ( ac1 != null ) ;

               // check To-Many-Link 'trackedByHLA' between ac1 and this
               JavaSDM.ensure ( ac1.hasInTrackedByHLA (this) ) ;

               // bind o: CORBAObject
               o = oldM.getIdentifiesCORBAO () ;
               JavaSDM.ensure ( o != null ) ;

               // bind newM: MobileIOR
               fujaba__IterThisRecordsMIORNewM = this.iteratorOfRecordsMIOR ()
;
               while ( !(fujaba__Success) &&
fujaba__IterThisRecordsMIORNewM.hasNext () )
```

Figure 8.4: Part of the Java source code generated by Fujaba

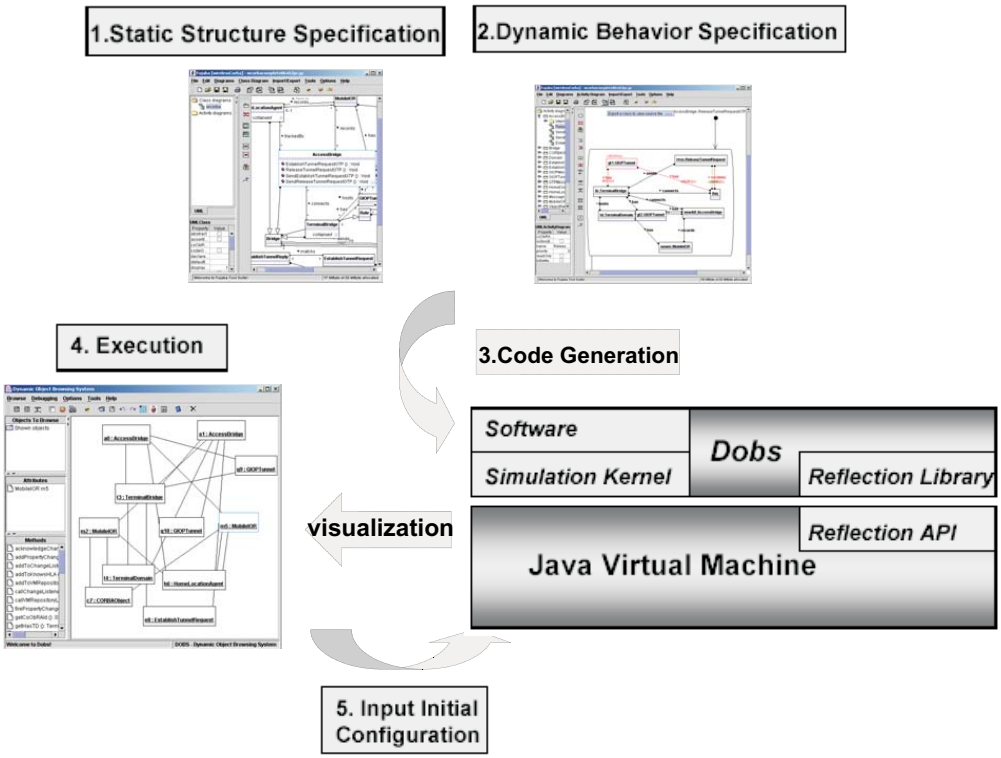

Figure 8.5: Fujaba simulation

user interface.

8.3.2 Efficient validation

Given a specified graph transformation system G = ⟨ TG, C, R ⟩, we can validate the correctness of the specification using simulation. We define a start graph S_0 to describe the initial configuration of the system, and an application scenario as a sequence of rules. The resulting trace of the sequence of rules can be validated through Fujaba Dobs simulation. In this way we can test whether the model reacts in the desired way, thus validating functional completeness of the system. We can further describe the validation process in a more formal way. If exists a transformation sequence or path p = $S_0 \xrightarrow{r_1} S_1 \xrightarrow{r_2} ... \xrightarrow{r_n} S_n$, we will say that $r_1; r_2; ...r_n$ is a *legal sequence* of the rules for the initial state S_0.

A Test Case A test case is a pair $(S_0, r_1; r_2; ...r_n)$, which consists of an initial state S_0 and a legal sequence of execution of the rules (methods) for the initial state.

A Test Result For a given test case T = (S_0, r_1; r_2; ...r_n), a test result is a sequence

$S_1; S_2; ...S_n$, such that S_i is the state of the system immediately after rule r_n has been executed.

An Expected Test Result For a given test case T = (S_0, r_1; r_2; ...r_n), an expected test result is a sequence $Er_1; Er_2; ...Er_n$, such that Er_i is the expected state of the system immediately after rule r_i has been executed.

A Correct Specification A TGTS specification is a correct one, if for all test cases, the corresponding test result is the same as the the corresponding expected test result, i.e., if $S_i = Er_i$.

This definition is based on the quite mature theory of conformance testing in the area of Labeled Transition Systems (LTS) [20, 147, 148], which checks that a specification fulfills its requirements, or that an implementation fulfills its specifications. The typed graph transformation system is a special kind of LTS, where the states of the system are object graphs, and its transitions are labeled by method expressions representing occurrences of operations.

There exist two possibilities to create test cases. We can either execute random actions, or design specific test scenarios that define the requirements for the specification. Although the former exercises a reasonable space of possible action sequences, it still can not guarantee a full coverage of the required scenarios. While by using the well chosen test scenarios we can validate exactly the required aspects of the specification through checking whether the desired scenario is realizable. During specifying the styles, we have already defined some required scenarios using UML sequence diagrams (see Section 6.5), which represent some requirements for the styles. At the same time, we can also create additional specific test scenarios that test special aspects of the specification. For example, the expected movement style of the components, the interaction process between a client and a server, etc. We will explain later in Section 8.5 how to create test scenarios through a quite complete example.

One important and difficult point with testing lies on the creation of the *initial state* S_0 of a test case, and the creation of the *expected test result* $Er_1; Er_2; \ldots Er_n$. The initial state S_0 should be reasonable enough to allow the users to pursue the following execution of the sequence of rules. That means, it should cover at lease the minimal preconditions of all the involved rules. At the same time, the expected test result should be correct itself in order to allow a correct judgement. It happens often that the test result is wrongly judged as incorrect, although the specification is proved to be correct later on. This is normally caused by inputting an incomplete initial state S_0, or the reference test result is wrong itself, both of which result in naturally wrong test result or wrong judgement. This makes the validation process itself error-prone and very inefficient.

We can solve this problem by constructing the minimal initial state S_0 that contains the needed precondition of all of the rules in the sequence, and the minimal expected test result $Er_1; Er_2; ...Er_n$. Recalling the refinement criteria of the sequence-mapped rule (see Section 7.4), we have created Algorithm 7.1 and 7.2

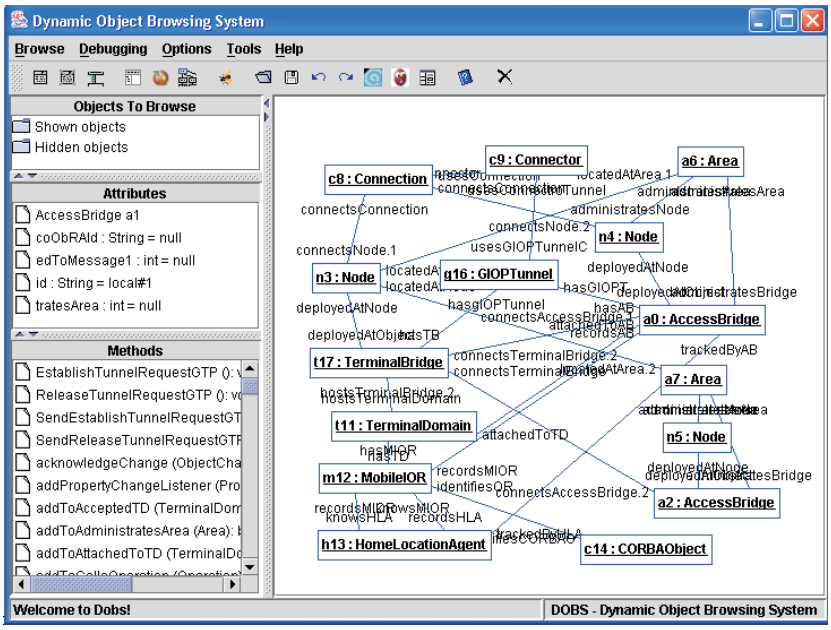

Figure 8.6: The initial graph for the scenario

that construct the overall precondition and postcondition of the rules in a fixed sequence based on a scenario construction, then it checks whether the scenario is reachable. Besides, the intermediate graph can be created too. It allows us to apply the rules one after another following the sequence. We denoted it as $L' \overset{r'_1}{\Longrightarrow} M'_1 \overset{r'_2}{\Longrightarrow} M'_2 \ ... \ M'_{n-1} \overset{r'_n}{\Longrightarrow} R'$, where M'_i is the created intermediate graph, L' is the initial graph that contains postcondition for applying the rules, R' is the last graph that contains the result (i.e., postcondition) after applying the last rule. Similarly, we can construct the minimal requirement for S_0 and the minimal expected test result $Er_1; Er_2; ...Er_n$ using Algorithm 7.1 and Algorithm 7.2. We get then $S_0 \supseteq (L' = L'_1 \oplus L'_2 \oplus, ..., \oplus L'_n)$, $Er_i \supseteq M'_i$, and $Er_n \supseteq (R' = R'_1 \oplus R'_2 \oplus, ..., \oplus R'_n)$.

Now, we will take an already specified important scenario, terminal-initiated handoff scenario of the platform-specific style (see Fig. 6.26, Section 6.5), as an example to show how to validate the scenario. The scenario was also used as an example in the refinement part to show how to correctly refine a single rule to a sequence of rules (see Section 7.4). We have presented the process of how to construct the L', M'_i, and R' from Fig. 7.10 to Fig. 7.21. Therefore, the corresponding minimal initial state S_0 is $L' = L'_1 \oplus L'_2 \oplus, ..., \oplus L'_9$ presented in Fig. 7.22, which is the revised rule after recovering the inconsistency according to the refinement correctness criteria. The minimal expected test result $Er_1; Er_2; ..., Er_8$ is the constructed intermediate graph $M_1; M_2; ...M_8$ respectively, and Er_9 is the overall postcondition R'.

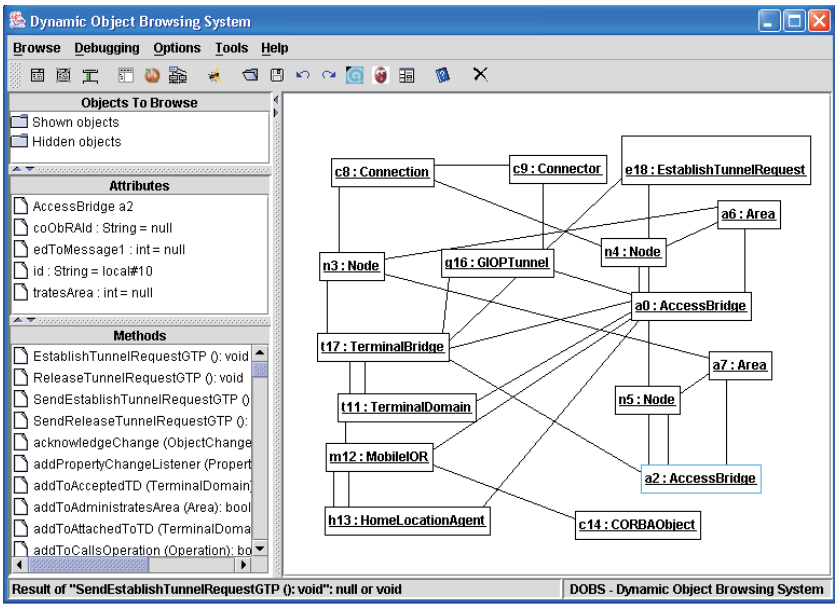

Figure 8.7: The graph after executing the first rule *SendEstablishTunnelRequest*

After starting Dobs to execute the model, we input the initial state S_0 (See Fig. 7.22). Afterwards, we execute the scenario according to the defined sequence of rules. The initial graph of the scenario in Dobs is shown in Fig. 8.6. We layout the graph in the same style as in Fig. 7.22, in order to ease comparison. The object names are given by Dobs automatically, which are named after the first letter of the class name and an unique number. Based on this graph, we execute the first rule *SendEstablishTunnelRequest* by calling the *SendEstablishTunnelRequest* method on object AccessBridge *a2*. We show the graph obtained after executing the rule in Fig. 8.7, which is the same as the calculated middle graph M_1' (see Fig. 7.19), except that we add now the needed elements in order to recover the refinement inconsistency. They are *a6:Area*, *a7:Area* and the connected edges, which keep unchanged during the execution of the whole scenario. Since the expected test result Er_1 is the same as M_1', we can conclude that the execution result $S_1 = Er_1$. We go on executing the rule sequence till the last rule *updateLocation*, whose result graph is shown in Fig. 8.8, which is the same as the revised R' (see Fig. 7.22). Again, we can conclude that the execution result $S_9 = Er_9$, since the expected test result Er_9 is the same as M_9'.

From the execution trace of the rules, we can check whether the test result is correct. In the given example, we start with the initial state $S_0 = L'$, and we get the execution result $S_1; S_2; ...S_9$, with $S_0 \xrightarrow{r_0} S_1 \xrightarrow{r_1} ... \xrightarrow{r_9} S_9$. After comparing the execution result $S_1; S_2; ...S_9$ to the minimal corresponding expected test result $Er_1 = M_1; Er_2 = M_2; ..., Er_8 = M_8; Er_9 = R'$, we can conclude that the test result is correct, for the given test case $(S_0, r_0; r_1; ...r_9)$, since $S_i = Er_i$.

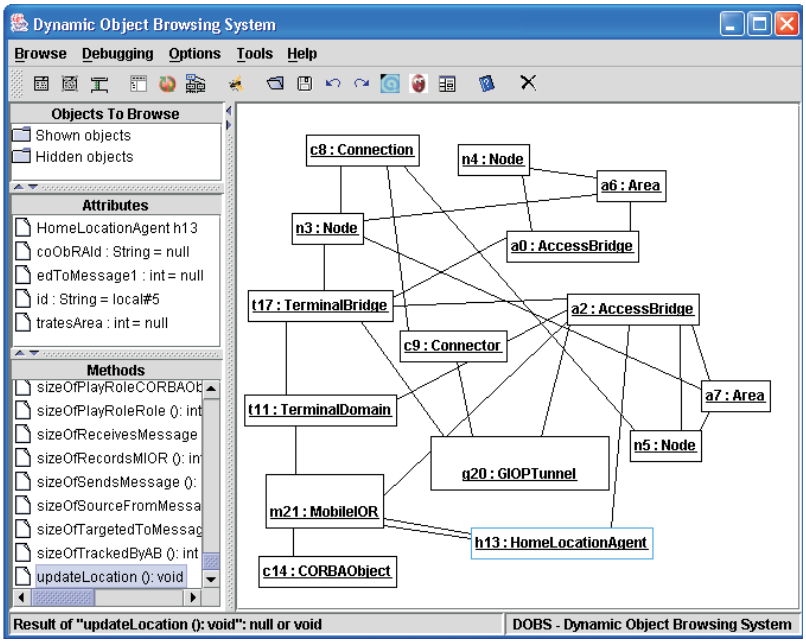

Figure 8.8: The last graph after executing the last rule *updateLocation*

Therefore, the system supports the application scenario defined by the test case. Through the construction of the minimal initial state $S_0 = L'$ and the minimal expected test result $Er_1; Er_2; ...Er_n$, we can greatly enhance the correctness and efficiency of the validation process.

8.3.3 Refinement consistency check

Recalling our definition of refinement in Chapter 7, we build the refinement relationship mainly based on the fixed mapping of rules between two graph transformation systems. Especially, we build refinement criteria for the sequence mapped rules and scenario mapped rules through a scenario-based construction, i.e., we construct the overall precondition, intermediate status, postcondition of the scenario using Algorithm 7.1 and 7.2. Afterwards, we check the correctness of the scenario through checking whether the scenario is reachable. This reachability check is rather complicated and error-prone if done manually. With the help of the Fujaba simulation tool Dobs, we can automate the refinement consistency check for behavioral refinement, although we can not do complete refinement checks.

Suppose that we have a constructed scenario for a refined rule with $L' \overset{r'_1}{\Longrightarrow} M'_1 \overset{r'_2}{\Longrightarrow} M'_2 \ldots M'_{n-1} \overset{r'_n}{\Longrightarrow} R'$, where L' is the initial graph that contains postcondition for applying the rules, M'_i is the created intermediate graph, R' is the last graph that contains the result (i.e., postcondition) after applying the last rule. We

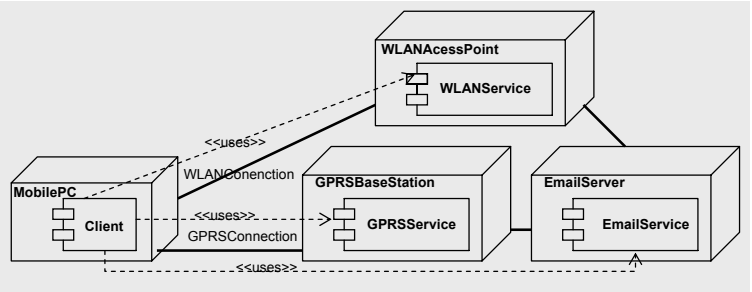

Figure 8.9: The application scenario (UML deployment diagram)

can check the correctness of the scenario by executing the rule sequence, taking L' as the input for the initial configuration. In last section, we have illustrated how to validate the scenario efficiently using the terminal-initiated handoff scenario of the platform-specific style, which is also a refined sequence mapped rule. At the same time, we have also proved that the scenario is reachable, i.e., all the constructed graphs are reachable. Therefore, we can conclude that the sequence of rules is a correct refinement.

Some researchers [145] develop the style refinement consistency check based on the model checking technique. They check whether all the states in one transformation system are reachable in another transformation system, vice versa. Hence, they require explicit generation of graph transition systems from the given GTSs, which is offered by model checkers like CheckVML and GROOVE. However, the approach is inefficient and impractical to large systems because the state space is too large to be explored. Different from [145], our style-based refinement theory does not rely on the exploration of a graph transition system. It is more important that the tool supports the execution of a fixed sequence of rules. Therefore, it enables us to use the simulator Dobs to automate the refinement consistency check, which is more suitable to large systems.

8.3.4 Behavioral consistency check

As having introduced in the overview, we develop a framework that checks whether the specified architectures on different abstraction layers provide the required functionalities (see Fig. 8.1), with the help of a standard reference application. This can check whether the architectures in a same hierarchy belong to the same family of middleware, i.e., they should realize and provide the same functionalities. We take the previous specified styles of the middleware for nomadic networks (see Section 6.3, 6.4, 6.5) as examples to show how to do the behavioral consistency check.

8.3.4.1 The reference application

The first step of the behavioral consistency check is to design a standard reference application (F), which should cover the requirements for the middleware and it is

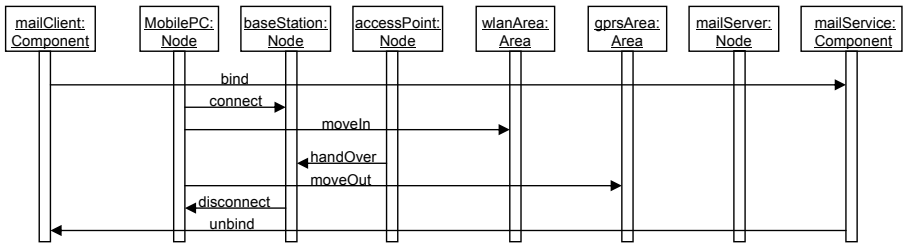

Figure 8.10: The standard reference application scenario

used to validate the functionality completeness of the architectures. In section 6.2, we have explained that the middleware for nomadic networks focuses on how to provide continuous connectivity and other services when components move across the structured spaces where handover protocols are often used. Therefore we design an application that tests these main aspects: Assume a mobile PC equipped with Wireless LAN and GPRS cards, hosting a component that needs to access an Email server on a stationary host, which is located in the fixed network. We suppose that we have two structured areas for movement: the office and outside. Wireless LAN and GPRS networks are both available in the office, while outside only GPRS is available. Because of the higher speed and cheaper price, the system should use the Wireless LAN whenever available. Therefore, when the user moves from the outside area to the office, the Email session based on GPRS should not be interrupted while the underlying connection is being changed to Wireless LAN. We show the components for the scenario in the deployment diagram of Fig. 8.9. The interaction between components is also defined in the sequence diagram (see Fig. 8.9). The interaction for the application scenario is rather simple, since the middleware below should provide the needed services, such as component invocation, component interaction, network connection, and handover, etc, which are hidden to the application.

8.3.4.2 Application Programming Interfaces (APIs)

As the second step, the Application Programming Interfaces (APIs)(G) should be provided by each of the styles, which encapsulate the needed operations that will be invoked and used by the standard application. We define several operations for the APIs, as shown in Fig. 8.11. Since the three styles are specified on different abstract layers that have different focuses, we extend the style models to provide a consistent API. In order to do so, we encapsulate the rules of the style to provide consistent services that allow the application to use.

As we can see, the offered APIs are the same as the rules specified in the conceptual style for the middleware, since the conceptual style specifies structured space of nomadic networks and the main functionality of the middleware. Therefore, we use directly the rules and it does not need to be extended in this case.

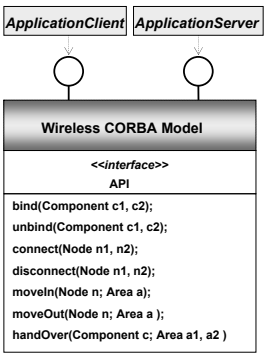

Figure 8.11: APIs for the reference application

Such a definition of APIs allows us to test the detailed aspects of the application scenario, which are very important in the validation process. It enables us to go into details and locate errors more easily. We can also define more abstract APIs that hide the implementation details of the middleware to the application.

The rules specified in other two concrete styles need to be encapsulated in order to provide the same APIs, since they contain refined rules that model more design-specific component interaction patterns. Because we define the refinement relationship mainly based on the fixed mapping of rules, we can now define the rule encapsulation based on the mappings. We show in Table 8.2 the encapsulations for the APIs. For instance, we encapsulate the two rules *remoteCall* and *dispatchMessage* of both conceptual styles for the *bind(Component c1, c2)*. The sequence of rules *connectBridge*, *sendMessage*, *dispatchMessage*, *request*,*callReturn* of the platform-independent concrete style are encapsulated for the *connect(Node n1, n2)*. We need to notice, this encapsulation is based on the scenario - mapped rules (see Table 7.4, Chapter 7), where the rules like *request*, *callReturn*, *reply*, *dispatchMessage*, *sendMessage* in the platform-independent concrete style are very different, which can not be mapped directly to any rules in the conceptual style. However, they can be associated with the abstract rules through a scenario definition, since they are added in order to realize a specific application scenario in the abstract style.

In Fujaba, we can do the encapsulation at the specification stage. We define a special class *APIs* that contains all the operations for the application. Using Fujaba story diagram editor, we can combine the sequence of rules into one complicated rule. Fujaba story diagram editor supports complicated programmed graph transformations, and it allows users to specify advanced control flow with user-defined order. For instance, we present in Fig. 8.12 a simple example of combining the two rules *remoteCall* and *dispatchMessage* of the platform-specific concrete style into one rule *bind(Component c1, c2)* for the APIs. The *remoteCall* is specified as the first activity, while *dispatchMessage* is specified as the second activity. The two activities are connected by transitions, that specify the execution sequence.

Style / APIs	Conceptual Style	Platform-independent concrete style	Platform-specific concrete style (Wireless CORBA)
bind(Component c1, c2)	bind	remoteCall	remoteCall
		dispatchMessage	dispatchMessage
connect(Node n1, n2)	connect	connectBridge	connectBridge
		sendMessage	buildTunnel sendMessageConcrete
		dispatchMessage	dispatchMessage
		request	request
		callReturn	callReturn
moveIn(Node n; Area a)	moveIn	moveIn	moveIn
handOver(Component c; Area a1, a2)	handOver	handOverConcrete	sendEstablishTunnelRequest rocessEstablishTunnelRequest send EstablishTunnelReply processEstablishTunnelReply sendReleaseTunnelRequest processReleaseTunnelRequest sendReleaseTunnelReply processReleaseTunnelReply updateLocation
moveOut(Node n; Area a)	moveOut	moveOut	moveOut
disconnect(Node n1, n2)	disconnect	dispatchMessage	dispatchMessage
		sendReplyMessage	sendReplyMessageConcrete releaseTunnel
		dispatchMessage	dispatchMessage
unbind(Component c1,c2)	unbind	dispatchMessage	dispatchMessage
		reply	reply

Table 8.2: Rule encapsulations for the APIs

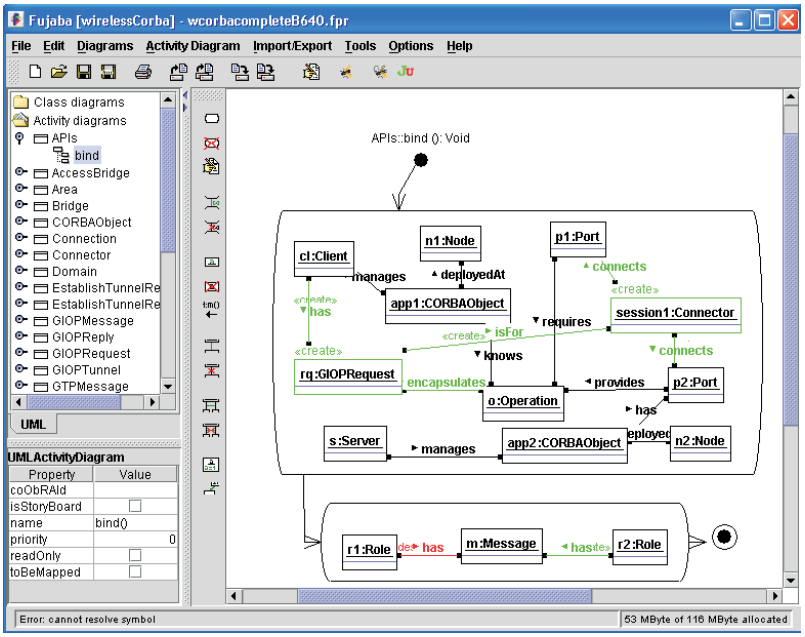

Figure 8.12: Combination of the rules in Fujaba Story Diagram

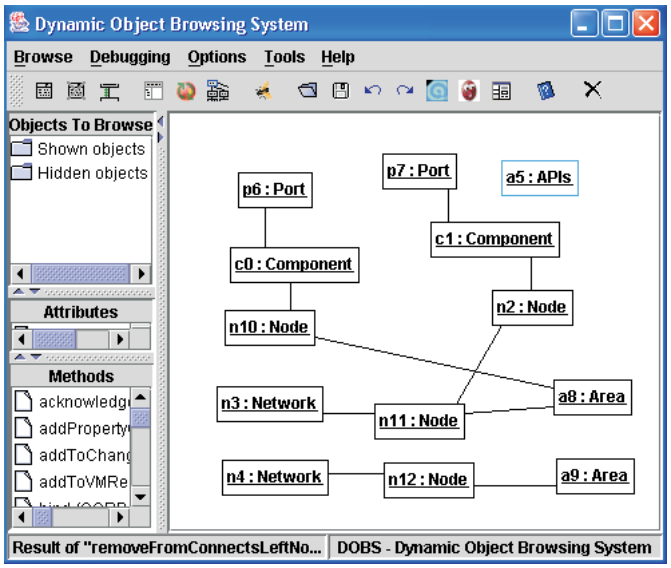

Figure 8.13: The initial graph of the application scenario for the conceptual style

8.3.4.3 Invocation and execution

Based on the APIs provided by the three styles, we can invoke and execute the operations to validate the application scenario. After running the system using Dobs (E), we need to build an initial system configuration that consists the elements to be tested. For instance, Fig. 8.13 is the initial graph of the application scenario for the conceptual style. In Fig. 8.13, *Component c0* is the application component that represents the email client, while *Component c1* represents the email server. Both of the components have a port, i.e., *Port p6* and *Port p7*, *Component c0* is mobile and it is deployed on *Node n10*. *Component c1* locates on the fixed network side. There are two areas for roaming. *Area a8* stands for the GPRS area, while *Area a9* stands for the WLAN area. The email client is located in the GPRS *Area a8*. The *Node n2* for the email server is connected to *Node n11*. We activate one after another the methods of *APIs a5*, which provides the invocation operations to the application. In the end, we get the last graph of the application scenario, which is shown in Fig. 8.14. Now, the mobile *Node n10* is located in both areas. Although the history of session activation and handover is not visible here, we can still see that the *Node n2* is now connected to *Node n12*, but not *Node n11* anymore, since *Node n10* uses now the network provided by *Node n12*. From the execution result, we can conclude that the required application is supported by the conceptual style.

Similarly, we have the initial configuration for the platform-specific style in Fig. 8.15. This graph is much more complicated as it includes design specific elements. We layout the elements is the same way as in the conceptual style, although the object id is not assigned the same anymore. The definition of *Node*, *Port*, *Area* and *Network* keeps the same. The *Component* is now changed to *CORBAObject*.

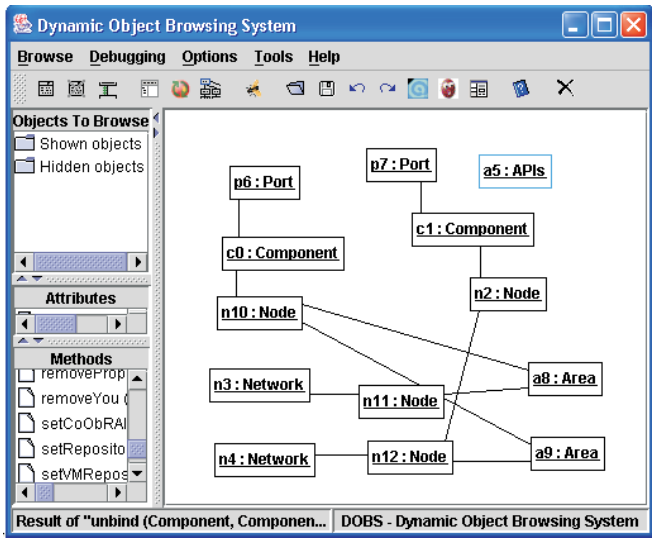

Figure 8.14: The last graph of the application scenario for the conceptual style

The *Operation o4* is introduced here, with *Port p2 requires* it and *Port p3* provides it. The session concept is refined to object invocation here. *Client c5* and *Server s6* manage the application objects, which are responsible for interactions between objects. Besides, the interaction is now handled between bridges, i.e. *TerminalBridge t13*, *AccessBridge a17* and *AccessBridge a18*. *MobileIOR m15* identifies the mobile object *CORBAObject c0* and recorders the currently attached *AccessBridge a17*. *HomeLocationAgent h16* records *MobileIOR m15* and tracks *AccessBridge a17*, where the mobile *CORBAObject c0* is located in. Again, we activate one by one the methods of *APIs a21*, which encapsulates the operations provided to the application. In the end, we get the last graph of the application scenario, which is shown in Fig. 8.16. Besides the same changes appearing in the conceptual style (see Fig. 8.14), we can observe other changes. The *TerminalDomain t14* is now attached to the new access bridge *AccessBridge a18*, but not the old one anymore. At the same time, *HomeLocationAgent h16* tracks the new access bridge, but not the old one. The old *MobileIOR m15* is deleted since it is not valid any longer. A new *MobileIOR m22* is created, which identifies the mobile object and records the new access bridge. *HomeLocationAgent h16* records also the new *MobileIOR m22*, not the old *MobileIOR m15*. In this case, the next possible invocations between the mobile client and server will be directed to the new access bridge. We can also conclude that the required application is supported by the platform-specific style, since the execution of the rule sequence is successfully pursued and the result is correct.

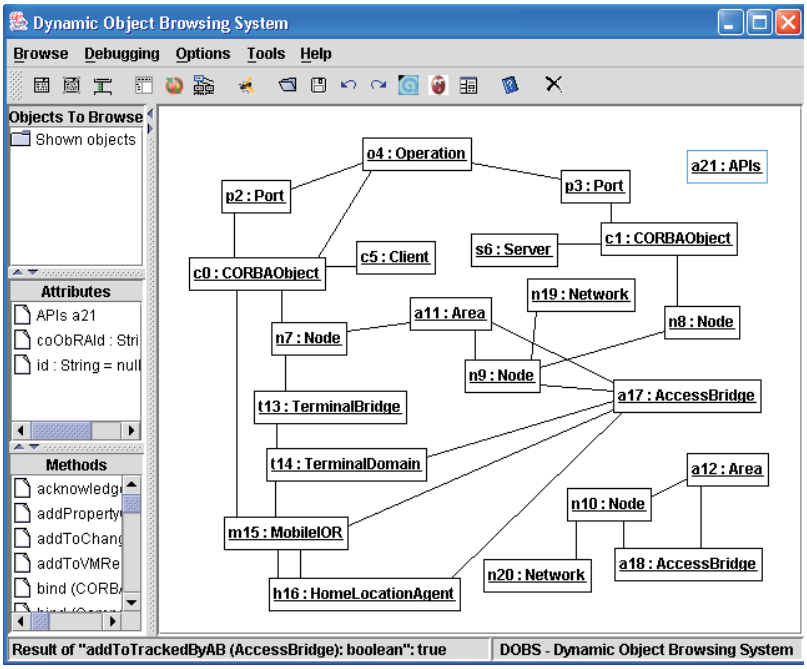

Figure 8.15: The initial graph of the application scenario for the platform-specific (Wireless CORBA) style

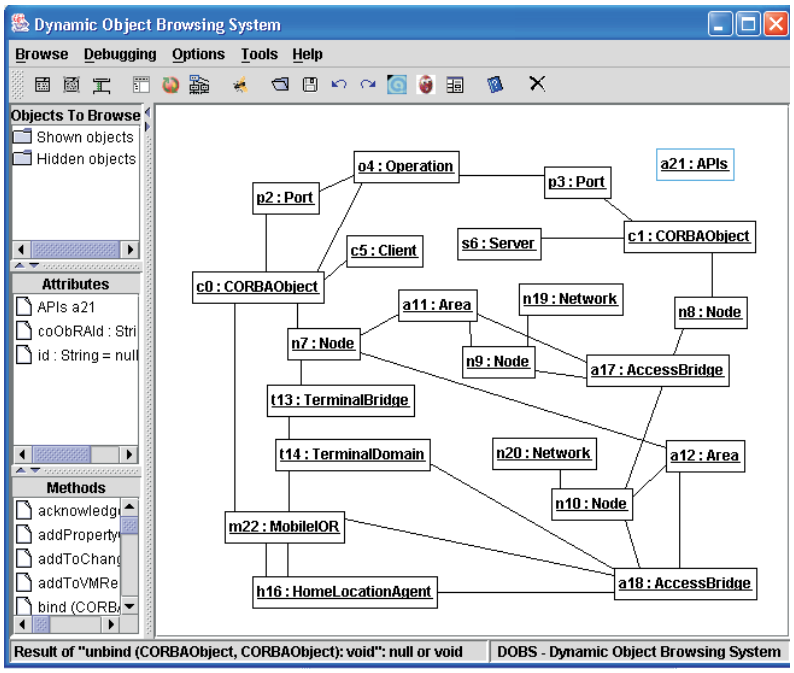

Figure 8.16: The last graph of the application scenario for the platform-specific (Wireless CORBA) style

8.3.5 Style - based engineering

Till now, we have introduced how to use Fujaba to specify and simulate the style, how to efficiently validate the style model, how to automate style refinement consistency check, and how to do behavioral consistency check. We will end this section by summarizing a process that helps the design and development of the style. Let us assume that we already have ideas what the style should include, and how to refine it to a more concrete one. We start with specification of the style:

1. Style specification Taking as the first step, we use Fujaba to edit the full style specification, i.e., the type graph (UML class diagrams) and the graph transformation rules. Fujaba provides an UML class diagram editor that supports rather complete UML class diagram notions, for instance, classes, attributes, qualified associations, cardinalities, multi-inheritance, etc. After inputting the type graph using the class diagram editor, we specify the graph transformation rules using story diagram editor. It adapts UML activity diagrams to represent explicit control flow graphically, which enables us to define complicated programmed graph transformations. Besides, it uses UML collaboration diagram notions to present left- and right-hand side of a rule within a single graph. Such an integration of well-known UML-like notations and graph transformation systems facilitates greatly the designers and developers who are not familiar with graph transformation systems.

During constructing the class diagram and rules, the editors do already some static semantics check. For instance, they check consistency of the class diagram, the users are only allowed to create instance elements that are correctly typed over the class diagram, etc. This helps the designers very much to create a consistent and correct specification. It enables the designers to focus on the content of the style, but not the distracting semantic consistency checks, which makes the style specification process efficient.

2. Style validation Having finished the specification, we validate it using the Fujaba simulator Dobs. Dobs allows executing a method (rule) graph at the method level, besides executing line by line at the source code level. This enables us to step through the execution of a system at the graph rewrite rule level of abstraction. From the execution result, we can also check whether the specified model reacts in the desired way. In case of errors, we can go back to the original graph, divide the rule into its atomic steps, and debug the models. Especially, we can define a start graph S_0 to describe the initial configuration of the system, and an application scenario as a sequence of rules. The resulting trace of the sequence of rules can be validated through Fujaba Dobs simulation. In this way we can test whether the style supports the application scenario, thus validating functional completeness of the system.

In addition, we provide an algorithm that allows us to create not only the initial state S_0 of a test case, but also the expected test result. This avoids the

wrong judgement of a test result, which is caused either because of inputting an incomplete initial state that does not ensure the validity of following rule executions, or because of a wrong reference test result. By doing so, we can greatly enhance the correctness and efficiency of the validation process.

3. **Refinement consistency check** Supposing that we get now an abstract style specification and a refined concrete one, the next step is to use Dobs to check the behavioral refinement consistency, since we build the refinement relationship mainly based on the fixed mapping of rules between two graph transformation systems. With the help of Dobs, we can check the correctness of the sequence-mapped rules and scenario-mapped rules through checking whether the refined sequence of rules is reachable. By doing so, we avoid the exploration of a complete graph transition system, which is inefficient and impractical to large systems because the state space is too large to be explored. It is more suitable to large systems.

   This helps a lot to ensure the correctness of the refined style, although we can not do complete automatic refinement checks, i.e., the required abstraction function, structural refinement checks and derived rule construction (see Section 8.2).

4. **Behavioral consistency check** Afterwards, we use the behavioral consistency check framework to check whether the architectures in a same hierarchy belong to the same family of middleware, i.e., they should realize and provide the same functionalities. At first, we need to design a standard reference application that should cover the requirements for the middleware and it is used to validate the functionality completeness of the architectures. Later on, the Application Programming Interfaces (APIs) should be provided by each of the styles, which encapsulate the needed operations that will be invoked and used by the standard application. We do the encapsulation based on the rule mappings and with the help of Fujaba story diagram editor, which allows us to combine the sequence of rules to one complicated rule. Having based on the APIs provided by the three styles, we can invoke and execute the operations in Dobs to validate the application scenario, to see, whether the style supports the same required scenario. From the execution result we can check and compare whether the architectures belong to the same family of middleware.

5. **Code generation** Finally, we get the validated styles that should be correct after passing the above mentioned different consistency check processes. Taking as the last step, we generate automatically executable Java source code. The specified UML classes are translated to Java classes, while the rules are translated to Java methods. Besides class frames and method declarations, method bodies are also generated. The generated code does not require an extensive library and may be integrated seamlessly with other system parts and it is platform independent due to the usage of pure Java.

Chapter 9

Conclusion

In the previous chapters, we have explained the whole architectural style-based approach, which includes the modeling language, style specification, style refinement, simulation, validation and behavioral consistency check. The overall objective of the thesis is to develop an architectural style-based approach to help the design and development of middleware for mobile systems. In this last chapter, we will at first review the objectives and requirements previously given for the approach, and evaluate how our approach fulfills them (in Section 9.1). Afterwards, we will explain (in Section 9.2) how to use the specified style to help the design and analysis of a new middleware at different design stages, taking the previous specified three-layered style of the middleware for nomadic networks as an example. We will also summarize the contributions (in Section 9.3), and give an outlook on future work (in Section 9.4).

9.1 Evaluation

In Section 1.3, we identified roughly four main objectives that the architecture centric approach should include, which are the architectural style, the modeling language, model-based analysis, refinement and consistency check. Later on, we gave the detailed requirements in the following chapters. In Section 3.2, we further classified the requirements into two well organized groups, which are style specification and the modeling language. We will now take this classification, separate the evaluation into two parts, and consider how our approach fulfills them.

9.1.1 Evaluating the style specification

The requirements for the style specification are satisfied by our TGTS (Typed Graph Transformation System) and meta-modeling based approach. As having explained in Chapter 5, we specify the style through defining the style definition language (i.e., metamodel, or type graph, and graph transformation rules) that is specialized for the middleware for mobile systems. More specifically, we create a specialized metamodel and a set of rules for defining the style for the middleware.

Our style specification can be separated into two parts: the structural part and the behavioral part. We specify the structural part using the type graph, which is based on the MOF (meta object facility) that provides the standard OO modeling concepts including MOF packages, classes and associations. They are visualized using UML class diagrams. The object diagrams are called instance graphs that are typed over the type graph. We use the type graph to define the vocabulary, allowed configurations, constraints, and allowed type definition of the style. The instance graphs of the type graph represent both the declarative definition of an architectural style and the style configuration. The behavioral part describes the runtime behavior of the system, which is specified using graph transformation rules at the instance level. Correspondingly, the specification covers the following required items:

Type definition We distinguish two different typing relations in our specification. One is between the style (class diagrams) and the model (object diagrams). Another one is within the models, for instance, between component types and components.

Style vocabulary Using the class diagram, we can easily specify the required vocabulary for the architectural style. The required architectural design elements such as wireless connection, space and middleware construction components are specified in a unified manner as classes in the same type graph. The other needed elements can be easily added because of the extensibility and adaptability of the modeling language, which is explained in the next section.

Configuration In the class diagram, we can specify the relationship and connection between the architectural elements using associations. The allowed configurations are then specified in the class diagram by defining a type relationship for the associated component and connector types and their instances building a certain runtime configuration.

Configuration constraints The class diagram consists of already cardinalities and other constraints. Therefore, it can easily be used to reflect the topological constraints of the architectural configurations.

Behavior Based on the graphic architectural configurations, we can specify dynamic changes and component interactions in a natural manner using graph transformation rules. We further differentiate two kinds of rules: rules for dynamic changes (with a nondeterministic sequence) and rules for component interaction (with a fixed sequence). Rules for dynamic changes specify the reconfiguration of the style in the presence of mobility, which includes for instance the movement style of the components in the specified space, the disconnection of the wireless connection, etc.

Rules for component interactions specify the component communication and collaboration, whilst the coordination and synchronization between different actions and components are specified using UML Interaction diagrams. The advantage to use rules in this case is that it allows us to specify naturally the dynamic reconfiguration among the coordinating components concerned in interactions. By doing so, we can specify the very dynamic component interaction pattern supported in the middleware for mobile systems.

Using graph transformation rules to specify both dynamic changes and component interactions enables us to integrate the two parts smoothly and directly into one specification with the same semantic background. This enhances the simplicity of the specification very much, and it eases the designers and developers to understand the model and to use the approach.

9.1.2 Evaluating the modeling language

Besides satisfying the requirements for the style specification, our TGTS and metamodeling based modeling language also fulfills the following requirements. Our approach provides a practical and usable process and environment to help the design and development of correct and consistent styles.

Understandability Visual representation and formal semantics are the main advantages of the modeling language. It has not only a graphic, easy understandable syntax, but also an operational semantics. The class diagram is very powerful to describe complex structures of a system, while the graph transformation system provides a sound way to formally describe the behavior of the system. The combination of the both can model concepts and ideas in a direct and intuitive way. At the same time the derived model is clear and without ambiguities.

Analysis Having a formal semantics, the typed graph transformation system enables us to analyze properties of the constructed models using modeling checking or simulation with various available tool supports. Especially, simulating the dynamic behavior of the system allows the designers to execute the system and to play with specific scenarios. The designers can concentrate on the key aspects of the architecture. It can also detect errors and improve the confidence of the model. We have explored the existing tool Fujaba as the basis for further style-based simulation (see Chapter 8). It allows us to validate efficiently the functionality completeness of the style. Besides behavioral analysis, it also allows us to pursue static semantic analysis, which includes for instance consistency check for the class diagram, type check for the instance graphs, etc. This helps the designers very much to create a consistent and correct style.

Due to the unified specification of different architectural elements as classes in the same type graph, the integrated specification of dynamic changes and component interactions as transformation rules, we can reason about mobility related aspects, such as location and wireless network connection, dynamic changes and

dynamic component interaction, etc., in the same manner as we reason about other components in a distributed system.

Formal semantics As already mentioned, the TGTS has a formal semantics. It enables us to formally specify the structure and behavior of a system. The derived model is clear and without ambiguities. It also supports execution, analysis, transformation, automation and toolsets.

Adaptability and extensibility As mentioned in Section 5.2, our approach is similar to the metamodeling-based language driven development approach that adapts the language (but not the model) to new application domains and new requirements. The TGTS provides a flexible way to extend or change the models. This allows adaption of a specialized model to meet new requirements of a new application. The adaption includes two parts: the type graph and rules. Class diagrams (Type Graph) provide a powerful and flexible way to model architectural elements and their relationships. New architectural elements can be easily added through adding proper associations and classes. While unnecessary elements can be deleted together with the related associations. The relevant rules built on instance graphs can be adapted easily to be typed over the adapted type graph. Rules can be added or deleted without violating the semantics of the TGTS.

Style refinement When developing the style, we use a stepwise refinement-based approach in order to decrease complexity and enhance reusability. We start from a simple abstract (i.e., conceptual) style that is refined to a more concrete style (platform-independent), which is further refined to an even more concrete style (platform-specific). In order to ensure that the refined concrete style is a correct refinement of the abstract one, we formalize the refinement relationship between two abstract layers based on the mapping of rules. We develop refinement correctness criteria that include both structural ones and behavioral ones, which allow us to check the correctness of the newly added rules and structural elements. Our formalization enables us to use the existing graph transformation simulation tool Fujaba as the basis for automating the refinement consistency check. It is more efficient and practical to large systems compared to the formalization based on model checking theory, since we do not need to explore the complete system state.

We have further identified in Section 7.2 much more detailed requirements for the refinement, such as functional preservation and functional constraints, structural preservation and functional constraints, reusability, etc. They identify the desired properties that need to be preserved during the refinement process. Later on in Section 7.4 and 7.5, we have concluded that all the required items are satisfied with our approach. We will not repeat them here anymore.

Consistency check We develop a framework that supports behavioral consistency check between models on different abstract layers. It checks whether the

specified architectures in a same hierarchy belong to the same family of middleware, i.e., they should realize and provide the same functionalities, although they belong to different abstraction layers and use different design strategies. Besides, we automate the consistency check process with the support of Fujaba.

Usability Besides supporting simulation, we also focus on providing a practical and usable process and environment to help the design and development of the style. After comparing the available simulation tools, we choose Fujaba [1] environment as our basis. Fujaba is a CASE (Computer-Aided Software Engineering) tool that aims to push graph transformation systems to a broader industrial usage. Fujaba allows a seamless integration of object-oriented software development and graph transformation systems. Its usage of well-known UML class diagrams, activity diagrams and collaboration diagrams as notations facilitates greatly the designers and developers who are not familiar with graph transformation systems. With its support, we develop a style - based engineering process for style development, which includes style specification, style validation, refinement consistency check, behavioral consistency check and code generation.

9.2 Relevance to practice

During developing the approach, we have emphasized understandability and usability of the approach, in order that the designers and developers can use it in practice to help the design and development of the middleware. The modeling language has a graphic, easy understandable syntax. It can describe complex structures and systems, and model concepts and ideas in a direct and intuitive way. The style is specified as simply as possible, for instance, we use graph transformation rules to specify both dynamic changes and component interactions in the behavioral part, for the purpose of enhancing simplicity and integrity. The stepwise refinement allows designers and developers to develop a complex, huge system step by step. It can decrease the complexity of the specified models and ease the design and development process. The automation of refinement consistency check and bahavioral consistency check lightens the check process. The style - based engineering process helps a lot to efficiently develop correct styles with tool support, which also allows a seamless integration to the well-known object oriented design. All these facilitate greatly the designers and developers to use the approach.

Now we will explain how to use the specified style to help the design and analysis of a new middleware at different design stages, taking the previous specified three-layered style of the middleware for nomadic networks (in Chapter 6) as an example.

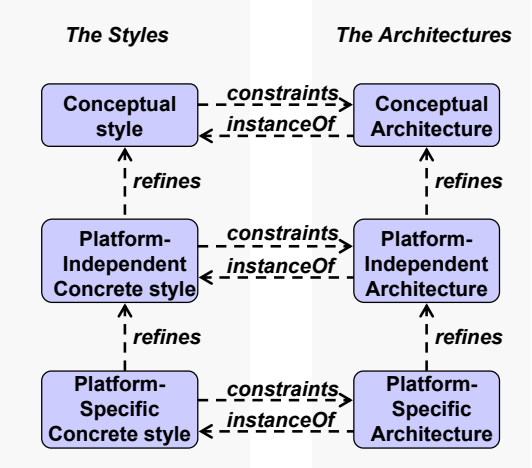

Figure 9.1: Styles and architectures

9.2.1 Style and design

We have constructed the three-layered style of the middleware for nomadic networks. Each of the style captures different architectural aspects or views that are resulted from the already established design knowledge or successful experience made by the practitioners in the middleware area. When building a new middleware for nomadic networks, the designers and developers do not need to explore all possible alternatives for its supported architecture. Instead, they can use the constructed architectural style that is effective for the middleware. They can define the design as instances of the style, or they can use the style as a reference model for further improvement and development (in Fig. 9.1). The style exposes the important issues and thus helping them to make effective choices and locate the best solution more easily. It enables them to ignore complications and alternatives that are not relevant to the middleware. By structuring the design space for a family of related middleware, the style can drastically simplify the process of building a middleware, reduce costs of implementation through reusable infrastructure, and improve system integrity through style-specific analysis and checks.

9.2.2 The conceptual style and design

When designing a new software, we generally start with a simple conceptual architecture that should cover the core functionality. The conceptual style specifies a style of mobile systems for nomadic networks, or a mobility style supported by the nomadic network, which includes the movement style of the components, and the specific service provided by this kind of mobile system. It is about the requirement analysis for the middleware and it can be used at the very early stage of designing the middleware.

For example, we define the conceptual architecture as an instance of the style,

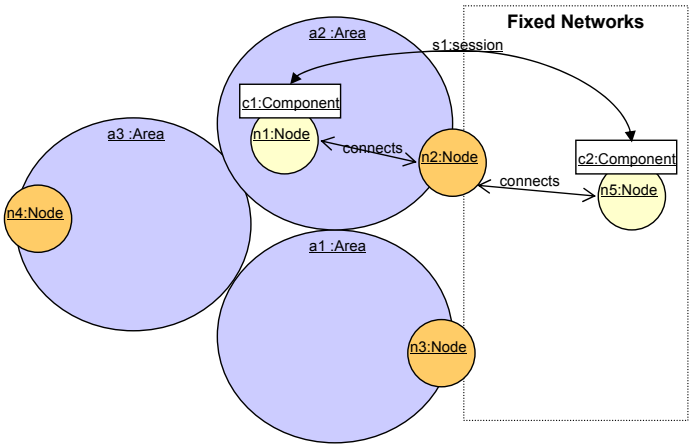

Figure 9.2: A conceptual architecture before executing the Rule *HandOver*

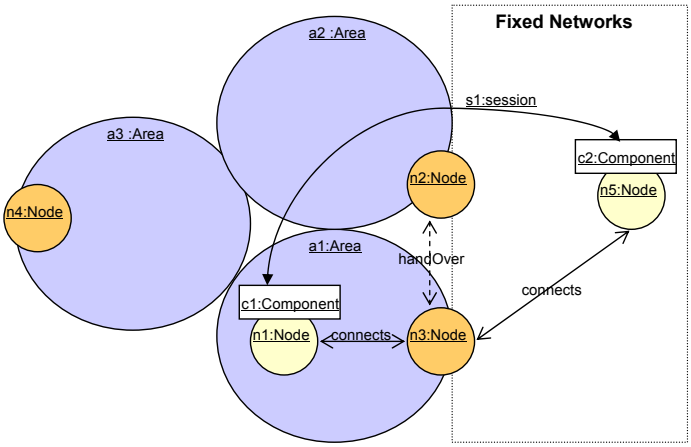

Figure 9.3: A conceptual architecture after executing the Rule *handOver*

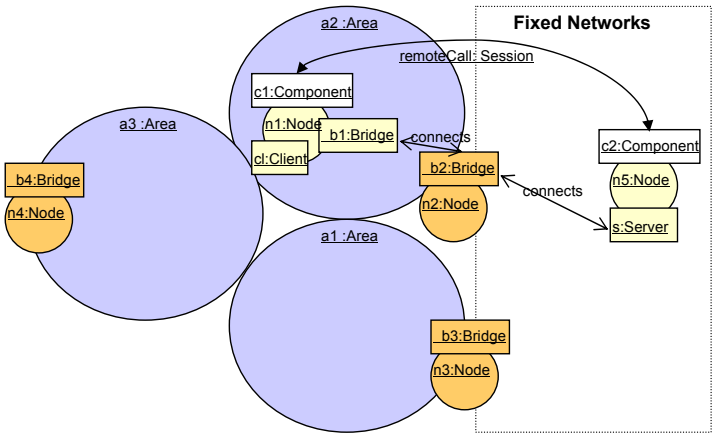

Figure 9.4: A platform-independent concrete architecture before executing the Rule *HandOver*

i.e., it is a configuration of the conceptual style in the instance graph (see Section 6.3). In the conceptual architecture, the network is divided into two main parts: the wireless network, and the fixed network. The wireless network is partitioned further into different structured areas *Area a1, a2, a3*, which are administrated by the Node *n1, n2, n3* respectively. Besides, the architecture includes a *Node n5*, which is located in the fixed network, while *Node n1* is a mobile node that can roam inside the areas. The main functionality of the middleware is to provide continuous service to the roaming components. This is modeled in the conceptual style (see Section 6.3) as *handover* rule, which keeps a session between the components.

In order to explain the model clearly, we will illustrate the most interesting rule handover through two rough pictures: Fig. 9.2 and Fig. 9.3 are about the architecture directly before and after the implementation of the rule respectively. Component *c1* locates in the Area *a2*, and has a session with Component *c2* that is on the fixed network side. The physical network connection used by the session is provided by the Node *n2*. When Component *c1* moves to Area *a1*, the old network connection will not be valid anymore, and a handOver will be processed between Node *n2* and *n3*. As a result, the session will be hold continuously, and the network connection used by the session now is provided by the Node *n3*.

9.2.3 The platform-independent concrete style and design

Based on the conceptual architecture, a more concrete architecture is created, which integrates more and more design specific aspects. We can define the concrete architecture as instances of the platform-independent concrete style, namely the platform-independent concrete architecture. The component interaction pattern is integrated now into the core functionality specified in the conceptual style. In other words, it specifies the design-specific component interaction pattern adapted

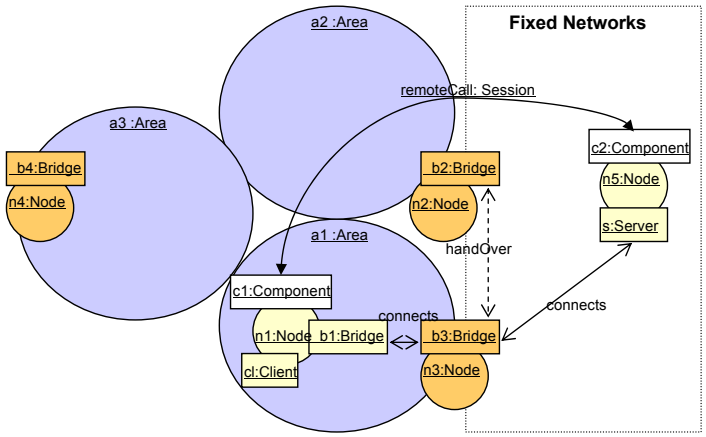

Figure 9.5: A platform-independent concrete architecture after executing the Rule *HandOver*

for the mobility style of the nomadic network. Correspondingly, the architecture is constrained by the style.

For example, the conceptual architectures in Fig. 9.2 and Fig. 9.3 are refined into Fig. 9.4 and Fig. 9.5 respectively. They add more elements and rules in order to support component interaction RPC model. The session in the conceptual style is refined into the remote procedural call, which is built between the client and server that are responsible for the interaction between application components, for example, marshalling and unmarshalling the message for a remote procedural call. *Bridge b1*, *b2*, *b3*, *b4* are added to connect the mobile component and the connectivity management nodes. The transmitting of messages for a procedure call and the hold of session connectivity will be pursued by bridges. The handover rule in the conceptual style is refined into a concrete handover rule that is now pursued between bridges, which keeps the remote procedural call continuously when the node roams.

9.2.4 The platform-specific concrete style and design

We can further refine the platform-independent concrete architecture into a platform-specific concrete architecture, which can be used when define a quite complete design of the middleware. On this layer, design-specific concepts and elements will be added according to the platform-specific concrete style specification. For example, the *Bridge* in the platform-independent concrete architecture is refined into two different bridges: each of the *Access Bridge* is located in a partitioned wireless network area, the Terminal Bridge is together with the mobile terminal. *Visited Domain*, *Terminal Domain* and *Home Domain* are introduced that represent the specific partition of the *area*. The structured areas for wireless network belong to one Visited Domain, which is hosted and managed by the access bridge. The Home Domain is used for the *Home Location Agent*. Besides, the message related part

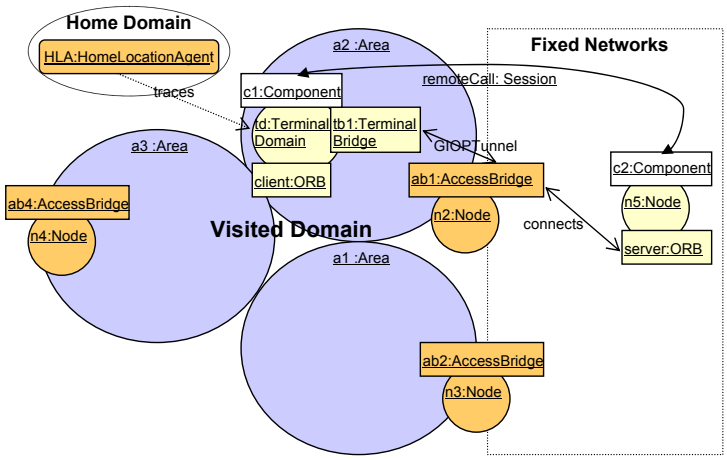

Figure 9.6: A platform-specific concrete architecture before executing the Rule *HandOver*

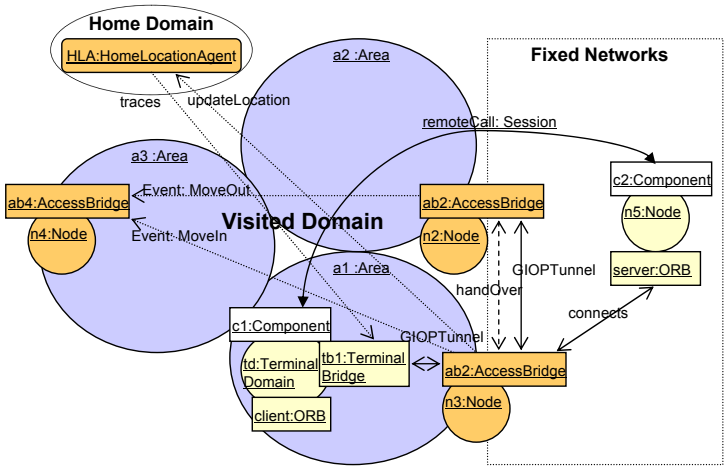

Figure 9.7: A platform-specific concrete architecture after executing the Rule *HandOver*

is refined into much more detailed version that defines how to transfer and process the messages for invocation (called GIOP message here) using specific message transmission protocols and tunnels, i.e., GTPs.

Many rules are added at this level, which are mostly used for the GTP message processing. The handover rule is refined to a sequence of rules, which are mainly about building and releasing the tunnel for transmitting the messages. The platform-independent architectures in Fig. 9.4 and Fig. 9.5 are refined into Fig. 9.6 and Fig. 9.7 respectively.

We only illustrate the main elements for the architecture in the pictures. The different elements for message processing are not given, which are quite complex. However, the pictures are already much more complicated than the one for the platform-specific concrete architecture and the conceptual architecture. It would be very difficult for the designers and developers to understand without other two more abstract architectures. Using the structured styles can greatly simplify the architectural design, and help the designers and developers fully understand how mobility influences the component interaction, and how the architecture evolves.

9.3 Contributions

The overall contribution of this thesis is an architectural style-based approach for helping the design and development of the middleware for mobile systems. By providing a concrete example of how to construct the style for a class of related middleware, and how to use the style to help the design and development of a new middleware, we have shown that the architectural style-based approach is useful and practical. In detail, we can separate mainly the contribution into the following categories.

The architectural style of the middleware for mobile systems We have developed a new architectural style for the middleware that captures the already established design knowledge or successful experience in this area. The style is middleware-induced. That means, instead of the general top-down approach adopted by the software architecture community, we take a bottom-up approach that originates from the results that practitioners have achieved in the middleware area. We think a class of related forms of the middleware induces the definition of the architectural style, with each specific middleware of the class defining a variation of that style.

In order to decrease complexity and enhance reusability, we specify the style on different abstract layers, namely, conceptual style (on more abstract layer) and concrete style (on more design-specific layer). The conceptual style specifies a style of mobile systems, or a mobility style, which includes the movement style of the components, and the specific service provided by this kind of mobility, etc. The concrete style models the more design-specific component interaction pattern adapted for the mobility style. The dynamic

change and functionalities specified in the conceptual style are refined into the concrete style.

This separation can simplify the specification of the style, and help the designers and developers fully understand how mobility influences the component interaction, and how the models evolve. This differentiates our style from the classical architectural styles that do not have such a separation. Besides, in contrast to the classical architectural style, we combine the specification of dynamic change and component interaction, since dynamic change is an important characteristic of mobile systems, and it also influences component interaction very much.

The architectural style of the middleware for nomadic networks We have explained the process of constructing the style through a concrete example: the architectural style of the middleware for nomadic networks. This type of middleware is quite mature from technique's point of view, and many existing middleware can be categorized into it. In spite of that, there is no common agreement or understanding of the middleware, not to mention available design standards that help the design and analysis of the middleware.

Our style represents a common form of design for this class of middleware, which originates from the results that practitioners have achieved in this area. We have constructed the style on three different abstract layers, i.e, the conceptual style, the platform-independent concrete style and the platform-specific concrete (Wireless CORBA) style, in order to decrease complexity and enhance reusability. Each of the styles captures different architectural aspects or views, and they can be used to help the design and analysis of the middleware at different design stages. The conceptual style specifies a style of mobile systems for nomadic networks, or a mobility style supported by the nomadic network, which includes the movement of the components, and the specific service provided by this kind of mobile system. It is about the requirement analysis for the middleware and it can be used at the very early stage of designing the middleware. The platform independent concrete style specifies the design-specific component interaction pattern adapted for the mobility style. We choose the adapted RPC as the example. The style can be used when we design a specific component interaction pattern for the middleware at a more abstract level. The platform-specific concrete style is a more refined version of the platform independent concrete style. It can be used when we define a complete design specification of the middleware, e.g., the Wireless CORBA specification in our case.

The graph transformation system based modeling To the existing contributions, we add the new role of graph transformation system as modeling mobility and the architectural style of the middleware for mobile systems. Although exploiting the same graph transformation system to specify the style, we apply it in a different way from [145], since the requirements of the style

vary. [145] focuses on formalizing the TGTS to define a general style for distributed systems, whereas we focus on modeling the specific aspects required by the style of the middleware for mobile systems, and we have provided a dedicated style definition language that meets the style requirements, i.e, we have created a specialized metamodel and a set of rules for defining the style of the middleware for mobile systems. We have specified the mobility related structural elements like space, wireless network connection, and the elements for the middleware. We have also specified dynamic changes and component interactions in the presentation of mobility, etc.

A conceptual framework for the middleware for mobile systems We have developed a conceptual framework for understanding and comparing different middleware, which is abstracted from specific implementation details and summarizes the main characteristics of the middleware. The generalization enables us to compare different approaches from a same point of view, and observe similarities and common aspects from the great diversities of the middleware. The current research in the middleware area focuses mainly on technical details and design strategies of the different middleware. What is missing is to apprehend the middleware from an abstract layer, which should abstract from particular product characteristics and provide a conceptual framework for understanding and comparing the different approaches. Our framework fills in this blank.

Modeling mobility Through our exploration of the specification of the style of middleware for mobile systems, we have contributed to an improved understanding of modeling mobility. For example, we have shown that it is very useful to specify a confined mobility area for the moving objects, since mobility is influenced and constrained by the under infrastructure. We have also shown that it is helpful to combine the specification of dynamic change and component interaction, since dynamic change is an important characteristic of mobile systems, and it also influences component interaction very much. The current approaches to model software architecture for distributed systems generally foster a separation of the two aspects since they do not relate each other.

Style - based refinement We have developed a new approach for correctly refining the architectural style specified using typed graph transformation systems. Especially, we formalize the refinement relationship between two abstract layers based on the mapping of rules. We derive a simplified rule for a sequence of rules through a scenario based construction. The derived rule preserves the semantics of the sequence of rules, thus it can be used to substitute the complicated rule sequence. By using the derived simplified rule, we build a bi-directional mapping between a sequence of rule and another rule sequence. We have also developed an algorithm to construct the derived rule.

Our approach is very different from that proposed in [145], which does not define fixed mappings between rules, but only between structural elements and transformation states. They check then whether all system states of an abstract model are also reachable at the concrete level, no matter by which order of transformation rules, vice versa. Their approach is not suitable for the systems that require correctness check of the fixed sequence of rules. In addition, it is very difficult to check the correctness of the completely different concrete behavior using their approach.

Our approach can check the correctness of the sequence of rules. It also enables us to check the correctness of newly added structural and behavioral elements through the scenario construction, which associates the completely different concrete elements with the abstract ones. Besides, our formalization can construct exactly the needed state graphs and transformation sequences for checking, which makes the approach efficient and practical to large systems. It also enables us to use the existing graph transformation simulation tool Fujaba as the basis for automating the refinement consistency check.

Style-based consistency check We have developed a framework that supports behavioral consistency check between two styles on different abstract layers. It checks whether the specified architectures in a same hierarchy belong to the same family of middleware, i.e., they should realize and provide the same functionalities, although they belong to different abstraction layers and use different design strategies. Besides, we have automated the consistency check process with the support of Fujaba.

Style-based validation We have developed an approach that can greatly enhance the correctness and efficiency of a validation process. We have constructed the minimal initial state and the expected test result with the help of an algorithm. This can solve the often happened problem caused by inputting an incomplete initial state for a test scenario, or the reference test result is wrong itself, both of which result in wrong test result or wrong judgement.

9.4 Future work

In spite of the previous summarized contributions, we can identify a number of issues for possible future research.

9.4.1 Industry project experience

We have made great effort to improve the understandability and usability of the approach, in order that the designers and developers can use it in practice to help the design and development of the middleware. We have explained how to use

the specified style to help the design and analysis of a new middleware at different design stages. We have also taken a concrete middleware Wireless CORBA as an example. However, all these parts have been carried out on the academic side. What we still miss is the real project experience in industry. Although this is not an easy task, we are still quite optimistic about this point, since real project experiences with Fujaba in other application domains have proved that the integration of typed graph transformation system and object oriented design is a powerful and practical method for software design and development [80].

9.4.2 Automation and tool support

Since the specification is often complicatedly constructed and thus difficult to be checked, automation and tool support can save considerable time and effort and improve the overall quality of the model and product. Although we have explained how to use the tool Fujaba to pursue efficient style validation, to help refinement consistency check, and to automate behavioral consistency, there still exist some open issues for future work.

Refinement consistency check For the refinement consistency check, we still need automation of the required abstraction function, structural refinement checks and derived rule construction (see Section 8.2). Especially, the derived rule construction is a very interesting point. It allows us to derive a simplified rule for a given sequence of rules. The derived rule preserves semantics of the sequence of rules since it is based on a scenario construction. Therefore, it can be used to substitute the complicated rule sequence. Besides our refinement process, such a derived rule is also very useful in other situations. For instance, we can simplify greatly the specification by combining a sequence of rule into one rule. We can derive exactly the needed initial status and expected test result for efficient validation, etc. In addition, one basic point for refinement-based development is the support of specifying and comparing two graph transformation systems in one project. However, this is not supported by the current graph transformation tools.

Behavioral consistency check For the behavioral consistency check, we can further improve the degree of automation and reusability by using a wrapper, which defines the refinement relationship between a concrete model and a conceptual model. As an adapter between Concrete API and Conceptual API, the wrapper encapsulates and maps the operations implemented in Concrete API to the operations defined in Conceptual API. Providing type transformation and semantic match, the Wrapper forwards operation calls to Conceptual API to the operation calls to Concrete API. In this way, the application can always use the more abstract interface while the concrete operations offered by the platform remain hidden. The concept of the wrapper definitely requires the support of specifying and comparing two graph trans-

formation systems in one project. As mentioned, this is not supported by the current tools.

Simulator with automatic mode There is still one important issue to mention for the Fujaba simulator Dobs. Currently, it supports only the interactive mode that allows users to decide certain transformation sequences by user-specific rule selections for every transformation step. It does not support automatic mode that applies a sequence of rules that are pre-defined by the users. However, automatic mode is very helpful in the case when the users want to check and debug the correctness of a sequence of rules. The tool AGG supports automatic mode by using a sophisticated graph pattern matching algorithm. We can refer to their solution for further development.

Finally, we have to notice that full automation is certainly unrealistic today. There is still a long way to go.

9.4.3 Development of other architectural styles

The goal of an architectural style is to capture solutions that were already in use, but not to make up solutions. There is a rule for how to build a style: before being eligible for inclusion, the style has to occur in at least two systems, or in two different organizations, and so on [36]. We have presented a new architectural style: the architectural style of the middleware for nomadic networks. It is still very useful if we can build other architectural styles for other class of related systems, since the style stands for a design standard for one area. A very possible one is the architectural style of the middleware for ad hoc networks. Recalling the conclusion and commonalities generalization made in Section 2.4, we have roughly classified the reviewed middleware into two classes: the middleware for nomadic networks, and the middleware for ad hoc networks. We have also generalized the commonalities of both middleware types regarding main functionality, component interaction and space definition. Our experience with the architectural style of the middleware for nomadic networks provides a good foundation for building the architectural style of the middleware for ad hoc networks. Although this class of middleware presents even more diversities, they can be still captured if we consider the more general aspects: space definition and main functionality.

Although our architectural style is specified to support the particular needs of the middleware for mobile systems, it clearly has potentials for supporting other types of systems and applications. This is supported by the adaptability and extensibility of the modeling language. It allows the language to easily adapt to new application domains and to evolve to meet new requirements. Besides, we can generalize the whole approach as a general refinement-based development process based on graph transformation systems, which makes the approach reusable for design and development of other complex software systems.

9.4.4 Model based testing

A crucial part of the development process is testing, which is often used in software engineering for validating properties. We can identify two possible future researches in this area.

Test oracle All software testing methods depend on the availability of an oracle [96, 118], that is, some methods for checking whether the system under test has behaved correctly on a particular execution. Executable formal specifications can be used as test oracles to produce the results expected for a test case. By comparing the result of a call to the actual implementation with the result of a call to the simulation, the test oracle can be used to check the correct execution of an operation. As a specification for a concrete middleware, the specified platform-specific concrete style can be extended to be a test oracle with the support of a simulator. Since the concrete model is platform independent concerning the independency of specific programming languages, hardware platforms and concrete implementation methods, it can be reused as a reference to test the correctness of implementations on different platforms.

The test oracle is very helpful for testing the implementation of a middleware for mobile systems. Such a testing is difficult due to the huge number of components in the system and the multitude of possible interactions between these components. The current available testing systems have problems to deal with. For example, the frame testing system described in [79] for Wireless CORBA is implementation related, i.e. the frame system can be only used for one specific implementation, and a new frame system needs to be developed for a new implementation of Wireless CORBA. Besides, it is very difficult to design test scenarios for such systems, and the expected test result can not be given. All these make the testing inefficient. We can construct a test oracle that provides the same test environment as the frame system, but our test oracle can be reused for different implementations. Besides, it is quite easy to design test scenarios with the help of the simulation, at the same time, the expected test result is also produced.

Testing mobility In reality, it is difficult and expensive to test the mobility support of a certain platform, which requires devices supporting wireless communication and specific tools to check the coordination logic of involved hardware and software components. Simulating the mobile platform can provide a simple and cheaper way to test the mobility aspects of the platform. Through this means, the context aspects of the platform like locations, network connections can be simulated directly, thus a dynamic execution environment can be provided for the context-aware applications, which is also difficult to test in reality.

Appendix A

OMG Wireless CORBA IDL

MobileTerminal.idl

```
//File: MobileTerminal.idl
#ifndef _MOBILE_TERMINAL_IDL_
#define _MOBILE_TERMINAL_IDL_
#include <orb.idl>
#include <IOP.idl>
#pragma prefix "omg.org"
module MobileTerminal {
    interface HomeLocationAgent;
    interface AccessBridge;
    typedef sequence<octet>  TerminalId;
    typedef sequence<octet>  GIOPEncapsulation;
    typedef sequence<octet>  GTPEncapsulation;
    struct Version {
        octet major;
        octet minor;
    };
    struct ProfileBody {
        Version                                 mior_version;
        octet                                   reserved;
        TerminalId                              terminal_id;
        sequence<octet>                         terminal_object_key;
        sequence<IOP::TaggedComponent>          components;
    };
    struct HomeLocationInfo {
        HomeLocationAgent  agent;
    };
    struct MobileObjectKey {
        Version             mior_version;
        octet               reserved;
        TerminalId          terminal_id;
        sequence<octet>     terminal_object_key;
    };
    enum HandoffStatus {
```

```
    HANDOFF_SUCCESS,
    HANDOFF_FAILURE,
    NO_MAKE_BEFORE_BREAK
};
const octet  TCP_TUNNELING = 0;
const octet  UDP_TUNNELING = 1;
const octet  WAP_TUNNELING = 2;
struct GTPInfo {
    Version       gtp_version;
    octet         protocol_level;
    octet         protocol_id;
                        //  values 0xE0...0xFF are reserved for internal use
};
struct AccessBridgeTransportAddress {
    GTPInfo                tunneling_protocol;
    sequence<octet>        transport_address;
};
typedef sequence<AccessBridgeTransportAddress>
    AccessBridgeTransportAddressList;
typedef string ObjectId; // same as CORBA::ORB::ObjectId
typedef sequence<ObjectId> ObjectIdList;
                // same as CORBA::ORB::ObjectIdList
exception IllegalTargetBridge {};
exception TerminalNotHere {};
exception UnknownTerminalId {};
exception UnknownTerminalLocation {};
exception InvalidName{}; // same asCORBA::ORB::InvalidNam
interface HomeLocationAgent {
    void update_location (
        in TerminalId          terminal_id,
        in AccessBridge        new_access_bridge
    ) raises (UnknownTerminalId, IllegalTargetBridge);
    boolean deregister_terminal (
        in TerminalId          terminal_id,
        in AccessBridge        old_access_bridge
    ) raises (UnknownTerminalId);
    void query_location (
        in  TerminalId              terminal_id,
        out AccessBridge            current_access_bridge
    ) raises (UnknownTerminalId, UnknownTerminalLocation);
    ObjectIdList list_initial_services ();
    Object resolve_initial_references (
        in ObjectId  identifier
    ) raises (InvalidName);
};
interface HandoffCallback {
    void report_handoff_status (
        in HandoffStatus  status
    );
};
interface AccessBridge {
    ObjectIdList list_initial_services ();
    Object resolve_initial_references (
         in ObjectId  identifier
    ) raises (InvalidName);
```

```
        boolean terminal_attached (
            in TerminalId  terminal_id
        );
        void get_address_info (
            out AccessBridgeTransportAddressList  transport_address_list
        );
        void start_handoff (
            in TerminalId           terminal_id,
            in AccessBridge         new_access_bridge,
            in HandoffCallback      handoff_callback_target
        );
        void transport_address_request (
            in  TerminalId                          terminal_id,
            out AccessBridgeTransportAddressList
new_access_bridge_addresses,
            out boolean                             terminal_accepted
        );
        void handoff_completed (
            in TerminalId       terminal_id,
            in HandoffStatus    status
        );
        void handoff_in_progress (
            in TerminalId       terminal_id,
            in AccessBridge     new_access_bridge
        );
        void recovery_request (
            in  TerminalId          terminal_id,
            in  AccessBridge        new_access_bridge,
            in  unsigned short      highest_gtp_seqno_received_at_terminal,
            out unsigned short
highest_gtp_seqno_received_at_access_bridge
        ) raises (UnknownTerminalId);
        void gtp_to_terminal (
            in TerminalId           terminal_id,
            in AccessBridge         old_access_bridge,
            in unsigned long        gtp_message_id,
            in GTPEncapsulation     gtp_message
        ) raises (TerminalNotHere);
        void gtp_from_terminal (
            in TerminalId           terminal_id,
            in unsigned long        gtp_message_id,
            in GTPEncapsulation     gtp_message
        ) raises (UnknownTerminalId);
        void gtp_acknowledge (
            in unsigned long        gtp_message_id,
            in GTP::ForwardStatus   status
        );
        void handoff_notice (
            in TerminalId       terminal_id,
            in AccessBridge     new_access_bridge
        );
        void subscribe_handoff_notice (
            in TerminalId       terminal_id,
            in AccessBridge     interested_access_bridge
        ) raises (TerminalNotHere);
```

```
    };
};
#endif
```

Module MobilityEventNotification

```
//File: MobileTerminalNotification.idl
#ifndef _MOBILE_TERMINAL_NOTIFICATION_IDL_
#define _MOBILE_TERMINAL_NOTIFICATION_IDL_
#include <orb.idl>
#include <IOP.idl>
#include "MobileTerminal.idl"
#pragma prefix "omg.org"
module MobileTerminalNotification {
    struct HandoffDepartureEvent {
        MobileTerminal::TerminalId          terminal_id;
        MobileTerminal::AccessBridge        new_access_bridge;
    };
    struct HandoffArrivalEvent {
        MobileTerminal::TerminalId          terminal_id;
        MobileTerminal::AccessBridge        old_access_bridge;
    };
    struct AccessDropoutEvent {
        MobileTerminal::TerminalId  terminal_id;
    };
    struct AccessRecoveryEvent {
        MobileTerminal::TerminalId  terminal_id;
    };
    struct TerminalHandoffEvent {
        MobileTerminal::AccessBridge  new_access_bridge;
    };
    struct TerminalDropoutEvent {
        MobileTerminal::TerminalId  terminal_id;
    };
    struct TerminalRecoveryEvent {
        MobileTerminal::TerminalId  terminal_id;
    };
};
#endif
```

Module GTP GIOP Tunneling Protocol

```
//File: GTP.idl
#ifndef _GTP_IDL_
#define _GTP_IDL_
#include "MobileTerminal.idl"
#pragma prefix "omg.org"
module GTP {
    struct GTPHeader {
        octet                gtp_msg_type;
        octet                flags;
        unsigned short       seq_no;
        unsigned short       last_seq_no_received;
```

```
        unsigned short        content_length;
    };
     typedef short  RequestType;
    const short  INITIAL_REQUEST = 0;
    const short  RECOVERY_REQUEST = 1;
    const short  NETWORK_REQUEST = 2;
    const short  TERMINAL_REQUEST = 3;
    struct InitialRequestBody {
        MobileTerminal::TerminalId              terminal_id;
        MobileTerminal::HomeLocationAgent       home_location_agent_reference;
        unsigned long                           time_to_live_request;
    };
    struct RecoveryRequestBody {
        MobileTerminal::TerminalId       terminal_id;
        MobileTerminal::HomeLocationAgent  home_location_agent_reference;
            struct LastAccessBridgeInfo {
            MobileTerminal::AccessBridge  access_bridge_reference;
            unsigned long             time_to_live_request;
            unsigned short            last_seq_no_received;
        } last_access_bridge_info;
        unsigned long  time_to_live_request;
    };
    typedef RecoveryRequestBody NetworkRequestBody;
    typedef RecoveryRequestBody TerminalRequestBody;
    union EstablishTunnelRequestBody switch (RequestType) {
        case INITIAL_REQUEST: InitialRequestBody initial_request_body;
        case RECOVERY_REQUEST: RecoveryRequestBody
recovery_request_body;
        case NETWORK_REQUEST: NetworkRequestBody network_request_body;
        case TERMINAL_REQUEST: TerminalRequestBody terminal_request_body;
    };
    typedef short  ReplyType;
    const short  INITIAL_REPLY = 0;
    const short  RECOVERY_REPLY = 1;
    const short  NETWORK_REPLY = 2;
    const short  TERMINAL_REPLY = 3;
    enum AccessStatus {
        ACCESS_ACCEPT,
        ACCESS_ACCEPT_RECOVERY,
        ACCESS_ACCEPT_HANDOFF,
        ACCESS_ACCEPT_LOCAL,
        ACCESS_REJECT_LOCATION_UPDATE_FAILURE,
        ACCESS_REJECT_ACCESS_DENIED,
        ACCESS_REJECT_RECOVERY_FAILURE
    };
    struct InitialReplyBody {
        AccessStatus              status;
        MobileTerminal::AccessBridge  access_bridge_reference;
        unsigned long             time_to_live_reply;
    };
    struct RecoveryReplyBody {
        AccessStatus              status;
        MobileTerminal::AccessBridge  access_bridge_reference;
        struct OldAccessBridgeInfo {
            unsigned long   time_to_live_reply;
```

```
                unsigned short  last_seq_no_received;
            } old_access_bridge_info;
            unsigned long  time_to_live_reply;
        };
        typedef RecoveryReplyBody NetworkReplyBody;
        typedef RecoveryReplyBody TerminalReplyBody;
        union EstablishTunnelReplyBody switch (ReplyType) {
            case INITIAL_REPLY: InitialReplyBody initial_reply_body;
            case RECOVERY_REPLY: RecoveryReplyBody recovery_reply_body;
            case NETWORK_REPLY: NetworkReplyBody network_reply_body;
            case TERMINAL_REPLY: TerminalReplyBody terminal_reply_body;
        };
        struct ReleaseTunnelRequestBody {
            unsigned long  time_to_live;
        };
        struct ReleaseTunnelReplyBody {
            unsigned long  time_to_live;
        };
        struct HandoffTunnelRequestBody {
            MobileTerminal::AccessBridgeTransportAddressList
    new_access_bridge_transport_address_list;
        };
        struct HandoffTunnelReplyBody {
            MobileTerminal::HandoffStatus  status;
        };
        struct OpenConnectionRequestBody {
            GIOP::TargetAddress target_object_reference;
            unsigned long       open_connection_request_id;
            unsigned long       timeout;
        };
        enum OpenConnectionStatus {
            OPEN_SUCCESS,
            OPEN_FAILED_UNREACHABLE_TARGET,
            OPEN_FAILED_OUT_OF_RESOURCES,
            OPEN_FAILED_TIMEOUT,
            OPEN_FAILED_UNKNOWN_REASON
        };
        struct OpenConnectionReplyBody {
            unsigned long               open_connection_request_id;
            OpenConnectionStatus        status;
            unsigned long               connection_id;
        };
        struct CloseConnectionRequestBody {
            unsigned long  connection_id;
        };
        enum CloseConnectionStatus {
            CLOSE_SUCCESS,
            CLOSE_FAILED_INVALID_CONNECTION_ID,
            CLOSE_FAILED_UNKNOWN_REASON
        };
        struct CloseConnectionReplyBody {
            unsigned long               connection_id;
            CloseConnectionStatus       status;
        };
        enum ConnectionCloseReason {
```

```
        CLOSE_REASON_REMOTE_END_CLOSE,
        CLOSE_REASON_RESOURCE_CONSTRAINT,
        CLOSE_REASON_IDLE_CLOSED,
        CLOSE_REASON_TIME_TO_LIVE_EXPIRED,
        CLOSE_REASON_UNKNOWN_REASON
    };
    struct ConnectionCloseIndicationBody {
        unsigned long           connection_id;
        ConnectionCloseReason   reason;
    };
    struct GIOPDataBody {
        unsigned long                       connection_id;
        unsigned long                       giop_message_id;
        MobileTerminal::GIOPEncapsulation   giop_message;
    };
    enum DeliveryStatus {
        DELIVERY_FAILED_INVALID_CONNECTION_ID,
        DELIVERY_FAILED_UNKNOWN_REASON
    };
    struct GIOPDataErrorBody {
        unsigned long   giop_message_id;
        DeliveryStatus  status;
    };
    struct GTPForwardBody {
        MobileTerminal::AccessBridge        access_bridge_reference;
        unsigned long                       gtp_message_id;
        MobileTerminal::GTPEncapsulation    gtp_message;
    };
    enum ForwardStatus {
        FORWARD_SUCCESS,
        FORWARD_ERROR_ACCESS_BRIDGE_UNREACHABLE,
        FORWARD_ERROR_UNKNOWN_SENDER,
        FORWARD_UNKNOWN_FORWARD_ERROR
    };
    struct GTPForwardReplyBody {
        unsigned long  gtp_message_id;
        ForwardStatus  status;
    };
    enum ErrorCode {
        ERROR_UNKNOWN_SENDER,
        ERROR_PROTOCOL_ERROR,
        ERROR_UNKNOWN_FATAL_ERROR
    };
    struct ErrorBody {
        unsigned short      gtp_seq_no;
        ErrorCode           error_code;
    };
};
#endif
```

Bibliography

[1] From UML to Java and Back Again:. `www.fujaba.de`.

[2] Metamodel website. `www.metamodel.com`.

[3] G. Abowd, R. Allen, and D. Garlan. Using style to give meaning to software architecture. In *SIGSOFT93: Foundations of Software Engineering*, pages 9–20. ACM Press, December 1993. Software Engineering Notes 118 (3).

[4] G. Abowd, R. Allen, and D. Garlan. Style-based refinement for software architectures. In *Second International Software Architecture Workshop (ISAW2)*, October 1996.

[5] G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. In *ACM Transactions on SoftWare Engineering and Methodology*, volume 4(4), pages 319–364, Oct. 1995.

[6] G. D. Abowd, A. K. Dey, R. Orr, and J. Brotherton. Context-awareness in wearable and ubiquitous computing. In *1st International Symposium on Wearable Computers*, pages 179–180, 1997.

[7] I. Akyildiz, J. Mcnair, H. Uzunalioglu, and W. Wang. Mobility management in next-generation wireless systems. *Proceedings of the IEEE*, 87:1347–1384, 1999.

[8] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[9] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *In Proceedings of Fundamental Approaches to Software Engineering (FASE'98)*, volume LNCS 1382, pages 21–37. Springer-Verlag, March 1998.

[10] L. Andrade, P. Baldan, and H. Baumeister. AGILE: Software architecture for mobility. In *Recent Trends in Algebraic Develeopment, 16th Intl. Workshop (WADT 2002)*, volume 2755 of *LNCS*, Frauenchiemsee, 2003. Springer-Verlag.

[11] T. Anne. Enterprise javabeans technology. Dec. 1998. `java.sun.com/products/ejb/white_paper.html`.

[12] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. Casl: The common algebraic specication language. In *Theoretical Computer Science*, 2003.

[13] R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of graph transformation to visual languages. In Ehrig et al. [47], pages 105–180.

[14] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and validation of service-oriented architectures: Application vs. style. In P. Inverardi and J. Paakki, editors, *Proc ESEC 2003: 9th European Software Engineering Conference*, pages 68–77, Helsinki, Finland, September 2003. ACM Press.

[15] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based refinement of dynamic software architectures. In *Proc. WICSA4 – 4^{th} Working IEEE/IFIP Conference on Software Architecture*, 2004.

[16] L. Bass and P. Clements. Software architecture in practice, second edition. Addison-Wesley, 2003.

[17] H. Baumeister, N. Koch, P. Kosiuczenko, and M. Wirsing. Extending activity diagrams to model mobile systems. In *Objects, Components, Architectures, Services, and Applications for a Networked World. International Conference NetObjectDays, NODe 2002*, volume 2591 of LNCS, pages 278–293, Erfurt, Germany, 2003. Springer.

[18] D. M. Berry. Academic legitimacy of the software engineering discipline. 1992. Software Engineering Institute, Technical Report CMU/SEI-92-TR-34.

[19] P. A. Berstein. Middleware: A model for distributed system services. In *Symposium on Principles of Distributed Computing*. Vol. 39, No. 2. Communications of the ACM, 1996.

[20] A. Bertolino, P. Inverardi, and H. Muccini. Formal methods in testing software architectures. In *Formal Methods for Software Architectures*, volume LNCS 2804. Springer, 2003.

[21] G. S. Blair, G. Coulson, and A. Andersen. The design and implementation of OpenORB, v2. In *IEEE DS Online*, 2001. Special Issue on Reflective Middleware.

[22] D. Blostein, H. Fahmy, and A. Grbavec. Issues in the practical use of graph rewriting. In *5th. Int. Workshop on Graph-Grammars and their Application to Computer Science*, volume LNCS 1073, pages 38–55. Springer, 1994.

[23] G. Booch, J. Rumbaugh, and I. Jacobson. The unified modeling language user guide. Addison Wesley, 1999.

[24] D. Box. Essential com. Addison-Wesley, 1997.

[25] J.S. Bradbury and et al. A classification of formal specifications for dynamic software architectures. In *International Workshop on Self-Managed Systems (WOSS'04),*. Newport Beach, California, USA, October/November 2004.

[26] L. Caires and L. Cardelli. A spatial logic for concurrency (part I). *Information and Computation*, 186(2):194–235, November 2003.

[27] A. Campbell, G. Coulson, and M. Kounavis. Managing complexity: Middleware explained. In *IT Professional*, pages 22–28. IEEE Computer Society, Sep./Oct. 1999.

[28] M. Caporuscio, A. Carzaniga, and A. L. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. In *Technical Report CU-CS-944-03, Department of Computer Science, University of Colorado*, January 2003.

[29] L. Capra, C. Mascolo, and W. Emmerich. Middleware for mobile computing. In Q. Mahmoud, editor, *Middleware for Communications*. John Wiley, 2002.

[30] L. Cardelli and A. Gordon. Anytime, anywhere. model logics for mobile ambients. In *27th ACM Symposium on Principles of Programming Languages*, pages 365–377. ACM, 2000.

[31] L. Cardelli and A.D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*. Springer-Verlag, Berlin Germany, 1998.

[32] J. Charles. Middleware moves to the forefront. pages 17–19. IEEE Computer, May 1999.

[33] Y. Chen, K. Schwan, and D. Zhou. Opportunistic channels: Mobility-aware event delivery. In *ACM/IFIP/USENIX International Middleware Conference 2003*, pages 182–201, 2003.

[34] S. W. Cheng and et al. Using architectural style as a basis for self-repair. In *Working IEEE/IFIP Conference on Software Architectures (WICSA 2002)*, August, 2002.

[35] T. Clark, A. Evans, P. Sammut, and J. Willans. Applied metamodelling: A foundation for language driven development. 2004. `www.xactium.com`.

[36] P. Clements, F. Bachmann, B. Len, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. Documenting software architectures: views and beyond. Addison Wesley, 2003.

[37] P. C. Clements. A survey of architecture description languages. In *Eighth International Workshop on Software Specification and Design*, March 1996.

[38] A. Corradini and R. Heckel. A compositional approach to structuring and refinement of typed graph grammars. In *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation"*, volume 2 of *Electronic Notes in TCS*, 1995.

[39] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–265, 1996.

[40] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In Rozenberg [122], pages 163–245.

[41] B. Courcelle. The monadic second-order logic of graphs i, recognizable sets of finite graphs. *Information and Computation*, 8521:12–75, 1990.

[42] S. Crawley, S. Davis, J. Indulska, S. McBride, and K. Raymond. Metameta is better-better. In *IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS97)*, September 1997.

[43] G. Cugola, E. D. Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. In *IEEE Transactions on Software Engineering (TSE), vol. 27*, pages 827–850, 2001.

[44] E. M. Dashofy, N. Medvidovic, and R. N. Taylor. Using off-the-shelf middleware to implement connectors in distributed software architectures. In *21st International Conference on Software Engineering, Los Angeles, CA, May 1999*.

[45] N. Davies, A. Friday, and A. Wade. L2imbo: A distributed systems platform for mobile computing. *ACM Mobile Networks and Applications (MONET) - Special Issue on Protocols and Software Paradigms of Mobile Networks*, pages 143–156, August 1998.

[46] A. Egyed and N. Medvidovic. Consistent architectural refinement and evolution using the unified modeling language. In *Proceedings of the Workshop on Describing Software Architecture with UML*, 2001.

[47] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations. Volume 2: Applications, Languages and Tools*, Singapore, 1999. World Scientific.

[48] H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.

[49] W. Emmerich. Engineering distributed objects. John Wiley, 2000.

[50] W. Emmerich. Software engineering and middleware: A roadmap. In *In The Future of Software Engineering - 22nd Int. Conf. on Software Engineering (ICSE2000)*, pages 117–129. ACM Press, May 2000.

[51] J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In Rozenberg [122], pages 1 – 94.

[52] G. Engels, R. Gall, M. Nagl, and W. Schäfer. Software specification using graph grammars. *Computing*, 31:317–346, 1983.

[53] L. Fiege, F. C. Gaertner, O. Kasten, and A. Zeidler. Supporting mobility in content-based publish/subscribe middleware. In *Middleware 2003*.

[54] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language and java. In *6th International Workshop on Theory and Application of Graph Transformation (TAGT)*. Springer Verlag, 1998.

[55] A. Gaddah and T. Kunz. A survey of middleware paradigms for mobile computing. July 2003. Carleton University Systems and Computing Engineering Technical Report SCE-03-16.

[56] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 175–188, 1994.

[57] L. Geiger and A. Zündorf. Graph based debugging with fujaba. In *Electronic Notes in Theoretical Computer Science 72*, volume 72 of *Electronic Notes in TCS*, 2002.

[58] D. Gelernter. Generative communication in linda. In *ACM Transactions on Programming Languages and Systems*, volume 7(1), pages 80–112, Jan. 1985.

[59] L. Gilman and R. Schreiber. In *Distributed Computing with IBM MQSeries*. Wiley, 1996.

[60] P. Grace. Middleware support for mobile computing applications. Sep. 2001. PhD First Year Report, Lancaster University.

[61] M. Große–Rhode, F. Parisi Presicce, and M. Simeoni. Spatial and temporal refinement of typed graph transformation systems. In *Proc. Mathematical*

Foundations of Computer Science (MFCS'98), LNCS 1450, pages 553–561. Springer-Verlag, 1998.

[62] M. Große-Rhode, F. Parisi Presicce, and M. Simeoni. Refinement of graph transformation systems via rule expressions. In *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn*, 1998.

[63] P. Guo, G. Engels, and R. Heckel. Architectural style - based modeling and simulation of complex software. In *12th Asia-Pacific Software Engineering Conference (APSEC05)*, pages 367–374. IEEE Computer Society, Dec. 2005.

[64] P. Guo and R. Heckel. Modeling and simulation of context-aware mobile systems. In *19th IEEE International Conference on Automated Software Engineering (ASE)*, pages 430–433. IEEE Computer Society, Sept. 2004.

[65] P. Guo and R. Heckel. Simulation and testing of mobile computing systems using fujaba. In *2004 Fujaba Days*, Sept. 2004.

[66] M. Haahr, R. Cunningham, and V. Cahill. Towards a generic architecture for mobile object-oriented applications. In *5th International Conference on Mobile Computing and Networking (MOBICOM99)*, Aug. 1999.

[67] C. L. Hall. In *Building Client/Server Applications Using TUXEDO*. Wiley, 1996.

[68] M. Hapner, R. Burridge, and R. Sharma. In *Java Message Service Specification*, Nov. 1999. Technical report, Sun Microsystems, `java.sun.com/products/jms`.

[69] M. J. Hawthorne and D. E. Perry. Architectural styles for adaptable self-healing dependable systems. In *ICSE 05*, pages 49–54. ACM, May 2005.

[70] R. Heckel and G. Engels. Graph transformation as a meta language for dynamic modeling and model evolution. In T. Mens and M. Wermelinger, editors, *Int. Special Session on Formal Foundations of Software Evolution, Lisboa, Portugal*, March 2001. Co-located with the European Conference on Software Maintainance and Reengineering (CSMR 2001).

[71] R. Heckel and G. Engels. Relating functional requirements and software architecture: Separation and consistency of concerns. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(5), 2002. Special issue on Separation of Concerns for Software Evolution, edited by Tom Mens and Michel Wermelinger.

[72] R. Heckel and P. Guo. Conceptual modeling of styles for mobile systems: A layered approach based on graph transformation. In *IFIP TC8 Working*

Conference on Mobile Information Systems(MOBIS) and IFIP International Federation for Information Processing, pages 65–78. Kluwer and Springer Verlag, Sept. 2004.

[73] M. Hennessy and J. Riely. A typed language for distributed mobile processes. In *Proc. ACM Principles of Prog. Lang.* ACM, 1998.

[74] M. Henning and S. Vinoski. Advanced corba programming with c++. Addison -Wesley, 1999.

[75] D. Hirsch. *Graph transformation models for software architecture styles*. PhD thesis, Departamento de Computación, Universidad de Buenos Aires, 2003.

[76] Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. In *Second ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDe'01)*, pages 27–34, 2001.

[77] ETSI European Telecommunications Standards Institute. Gsm 03.09 (ets300 527): digital cellular telecommunications system (phase 2); handover procedures. version 4.6.0, September 1996.

[78] D. Janssens and G. Rozenberg. On the structure of node-label controlled graph grammars. *Information Science*, 20:191–216, 1980.

[79] J. Kangasharju. Implementing the wireless corba specification, 2002. Laudatur Thesis, Univ. of Helsinki.

[80] H. J. Koehler, U. Nickel, J. Niere, and A. Zündorf. Integrating uml diagrams for production control systems. In *22nd International Conference on Software Engineering (ICSE)*, pages 241–251. ACM Press, 2000.

[81] P. Kogut and P. C. Clements. Feature analysis of architecture description languages. In *Software Technology Conference (STC95)*, April 1995.

[82] F. Kon, M. Roman, and P. Liu. Monitoring, security, and dynamic configuration with the dynamicTAO Reflective ORB. In *Proceedings of the Middleware 2000 Conference*. ACM/IFIP, 2000.

[83] M. Korff. Minimality of derived rules in single pushout graph rewriting. Technical Report 94/10, Technical University of Berlin, 1994.

[84] P. Kosiuczenko. Sequence diagrams for mobility. In *MobIMod workshop*, volume LNCS, Tampere, Finland, 2002. Springer.

[85] A. Lopes, J. Fiadeiro, and M. Wermelinger. Architectural primitives for distribution and mobility. In *Proc. 10th ACM SIGSOFT symposium on Foundations of software engineering (FSE 2002)*, pages 41–50, Charleston, South Carolina, USA, 2002. ACM.

[86] 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*, 21(4):336–355, 1995.

[87] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proc. ESEC 95 - 5^{th} European Software Engineering Conference*, volume 989 of *LNCS*, pages 137–153. Springer, 1995.

[88] L. S. Marks. Mark's standard handbook for mechanical engineers. McGraw-Hill, 1987.

[89] R. Marvie. Separation of concerns and metamodeling applied to software architecture handling. 2002. University of Lille, France, PHD Thesis.

[90] N. Medvidovic. On the role of middleware in architecture-based software development. In *SEKE, 02*. ACM, 2002.

[91] N. Medvidovic and D. S. Rosenblum. Domains of concern in software architectures and architecture description languages. In *The USENIX Conference on Domain-Specific Languages*, pages 199–212, October 1997.

[92] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Trans. Softw. Eng. 26*, pages 70–93, January 1998.

[93] N. Medvidovic and R. N. Taylor. A framework for classifying and comparing architecture description languages. In *The Sixth European Software Engineering Conference together with Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 22–25, September 1997.

[94] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *22nd International Conference on Software Engineering*, 2000.

[95] R. Meier and V. Cahill. Steam: Event-based middleware for wireless ad hoc networks. In *International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02)*, 2002.

[96] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for guis. In *the ACM SIGSOFT 8th International Symposium on the Foundations of Software Engineering (FSE-8)*, pages 30–39, Nov. 2000.

[97] S. Merz, M. Wirsing, and J. Zappe. A spatio-temporal logic for the specification and refinement of mobile systems. In Mauro Pezzè, editor, *Proc. Fundamental Approaches to Software Engineering, 6th International Conference (FASE 2003)*, volume 2621 of *LNCS*, pages 87–101. Springer-Verlag, 2003.

[98] M. Mikic-Rakic, N. Mehta, and N. Medvidovic. Architectural style requirements for self-healing systems. In *The first workshop on Self-healing systems*, pages 49–54. ACM, 2002.

[99] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[100] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural styles, design patterns, and objects. In *IEEE Software, 14(1)*, pages 43–52, Jan. 1997.

[101] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, 1995.

[102] A. L. Murphy, G. P. Picco, and G. C. Roman. Lime: A middleware for physical and logical mobility. In *21st International Conference on Distributed Computing Systems (ICDCS-21)*, May 2001.

[103] U. Nickel, J. Niere, and A. Zündorf. The fujaba environment. In *ICSE 2000, Limerick, Ireland*. ACM, 2000.

[104] U.A. Nickel and J. Niere. Modelling and simulation of a material flow system. In *Proc. of Workshop Modellierung (Mod)*, 2001. Bad Lippspringe, Germany, Gesellschaft for Informatik.

[105] R. De Nicola, G. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. In *IEEE Trans. on Software Engineering*, pages 315–330. volume 24(5), 1998.

[106] E. D. Nitto and D. Rosenblum. Exploiting adls to specify architectural styles induced by middleware infrastructures. In *Int. Conf. on Software Engineering*, pages 13–22. ACM Press, 1999.

[107] OMG. Issues for wireless corba ftf. `www.omg.org/issues/wireless-ftf.html`.

[108] OMG. Meta object facility (mof) specification version 1.3. March 2000, OMG TC Document formal/00-04-03.

[109] OMG. White paper on wireless access and terminal mobility in CORBA, November 1998.

[110] OMG. Common object request broker architecture(corba): Core specification, v3.0, 2002. `www.omg.org/docs/formal/02-11-01.pdf`.

[111] OMG. Wireless access and terminal mobility in CORBA, v1.0, 2003. `www.omg.org/cgi-bin/doc?formal/2003-03-64`.

[112] OMG. Wireless access and terminal mobility in CORBA, v1.0, 2003.

[113] P. Oreizy and et al. An architecture-based approach to self-adaptive software. In *IEEE Intelligent Systems*, volume 14, pages 54–62, May/June 1999.

[114] C. Perkins. Ad-hoc networking. In *Addison-Wesley*, Jan. 2001.

[115] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. In *ACM SIGSOFT Software Engineering Notes*, volume 17, pages 40–52. ACM, 1992.

[116] R. H. Perry and et al. Perry's chemical engineers' handbook. McGraw-Hill, 1984.

[117] F. Plasil and M. Stal. An architectural view of distributed objects and components in corba, java rmi and com/dcom. In *Software Concepts and Tools, 19(1)*, pages 14–28, 1998.

[118] D. J. Richardson, S. L. Aha, and T. O'Malley. Specifictlon-based test oracles for reactive systems. In *International Conference on Software Engineering (ICSE)*, pages 105–118, May 1992.

[119] G.-C. Roman, P. J. McCann, and J. Y. Plun. Mobile UNITY: reasoning and specification in mobile computing. In *ACM Transactions on Software Engineering and Methodology*, pages 250–282, 1997.

[120] G.-C. Roman, A. L. Murphy, and G. P. Picco. Coordination and mobility. In *Coordination of Internet agents: models, technologies, and applications*, pages 253–273. Springer-Verlag, 2001.

[121] G.-C. Roman, G. P. Picco, and A. L. Murphy. Software engineering for mobility: A roadmap. In A. Finkelstein, editor, *Proc. ICSE 2000: The Future of Software Engineering*, pages 241–258. ACM Press, 2000.

[122] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.

[123] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Symposium on Principles of Distributed Computing*, pages 1–7, 1996.

[124] R. E. Schantz and D. C. Schmidt. Research advances in middleware for distributed systems: State of the art. In *IFIP 17th World Computer Congress - TC6 Stream on Communication Systems*, pages 1–36. Kluwer, 2002.

[125] B. Schilit and N. W. Adams. Context aware computing applications. In *Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE Computer Society, December 1994.

[126] W. N. Schilit. *A System Architecture for Context-Aware Mobile Computing*. PhD thesis, Columbia University, may 1995.

[127] D. Schollmeier, I. Gruber, and M. Finkenzeller. Routing in mobile ad hoc and peer-to-peer networks: A comparison. In *Workshops on Web Engineering and Peer-to-Peer, Computing, Networking 2002*, volume LNCS 2376, pages 172–186. Springer Verlag, 2002.

[128] A. Schürr. Progress: A VHL-language based on graph grammars. In *4th Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 532*. Springer-Verlag, 1991.

[129] A. Schürr. Programmed graph replacement systems. In Rozenberg [122], pages 479 – 546.

[130] A. Schürr, A. J. Winter, and A. Zündorf. PROGRES: Language and environment. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, pages 487–550. World Scientific, Singapore, 1997.

[131] A. Schürr, A.J. Winter, and A. Zündorf. Graph grammar engineering with PROGRES. In *5th European Software Engineering Conference (ESEC'95), Sitges, LNCS 989*, pages 219–234. Springer-Verlag, 1995.

[132] J. Seitz, N. Davies, M. Ebner, and A. Friday. A corba based proxy architecture for mobile multimedia applications. In *2nd IFIP/IEEE International Conference on Management of Multimedia Networks and Services*, November 1998.

[133] R. Sessions. Com and dcom. John Wiley, 1998.

[134] M. Shaw. Prospects for an engineering discipline of software engineering. In *IEEE Software*, pages 15–24, Nov. 1990.

[135] M. Shaw. What makes good research in software engineering? In *International Journal on Software Tools for Technology Transfer*, volume 4, pages 1–7, 2002.

[136] M. Shaw, R. Deline, and et al. Formulations and formalisms in software architecture. In *In Computer Science Today: Recent Trends and Developments*, volume LNCS 1000, pages 307–323. Springer-Verlag, 1995.

[137] M. Shaw and D. Garlan. Software architecture: Perspectives on an emerging discipline. Prentice-Hall, 1996.

[138] M. Shaw and D. Garlan. Abstractions for software architecture and tools to support them. In *IEEE Transaction of Software Engineering 21*, pages 314–335, Apr. 1995.

[139] M. Shaw and D. Garlan. Characteristics of higher-level languages for software architecture. December 1994. Technical Report, CMUCS-94-210, Carnegie Mellon University.

[140] I. Sommerville. Software engineering. Addison-Wesley, 1995. 5th edition.

[141] Sun. Javaspaces. In *The JavaSpaces Specification web page*. `http://www.sun.com/jini/specs/js-spec.html`.

[142] A. S. Tanenbaum and M. van Steen. Distributed systems: Principles and paradigms. Prentice Hall, 2002.

[143] P. Tarasewich, R. C. Nickerson, and M. Warkentin. Issues in mobile e-commerce. In *Communications of the Association for Information Systems*, pages 41–64, 2002.

[144] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead Jr., J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6):390–406, June 1996.

[145] S. Th*ö*ne. Dynamic software architectures: A style-based modeling and refinement technique with graph transformations. PHD Theis, University of Paderborn, October 2005.

[146] J. M. Thomas and A. R. William. Inside corba: Distributed object standards and applications. Addison - Wesley, 1997.

[147] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. In *Software-Concepts and Tools*, volume 17(3), pages 103–120, 1996.

[148] J. Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99*, volume LNCS 1664, pages 46–65, 1999.

[149] S. Trigila, P. Reynolds, K. Raatikainen, and B. Wind. Mobility in long-term service architectures and distributed platforms. In *IEEE Personal Communications Magazine 5(4)*, pages 44–55, Aug. 1999.

[150] U. Varshney, R. Vetter, and R. Kalakota. Mobile commerce: A new frontier. In *IEEE Computer, 33(10)*, pages 32–38, 2000.

[151] S. Vestal. A cursory overview and comparison of four architecture description languages. Honeywell Technology Center, February 1993.

[152] S. Vestal, P. Manual, and V. Technical. Metah programmer's manual version 1.09. Technical report, Honeywell Technology Center, April 1996.

[153] R. Want and et al. An overview of the parctab ubiquitous computing experiment. In *IEEE Personal Communications 2 (6)*, pages 28–43, 1995.

[154] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The active badge location system. In *ACM Transactions on Information Systems 10(1)*, pages 91–102, 1992.

[155] A. L. Wolf. Succeedings of the second international software architecture workshop (isaw-2). In *ACM SIGSOFT Software Engineering Notes*, volume 22, pages 42–56, January 1997.

[156] P. Wyckoff, S. McLaughry, and T. Lehman. Tspaces. *IBM Systems Journal*, pages 454–474, 1998.

[157] A. Zeidler and L. Fiege. Mobility support with rebeca. In *23rd International Conference on Distributed Computing Systems, Workshop on Mobile Computing Middleware*, 2003.

List of Figures

List of Tables