Service Specification and Matching based on Graph Transformation

D I S S E R T A T I O N

in Computer Science

submitted to the

Faculty of Computer Science,

Electrical Engineering, and Mathematics

University of Paderborn

by Alexey Cherchago

in partial fulfillment of the requirements for the degree of

doctor rerum naturalium (Dr. rer. nat.)

Paderborn 2006

Abstract

One of the main tasks of modern application integration projects is to allow one business unit (requestor) to use services offered by another business unit (provider). When software systems of business partners are composed, an import interface of the requestor system is connected to an export interface of the provider system. Here, the import interface specification containing the requestor's requirements for a needed service has to be matched against the export interface specification describing the provided service. Usually, software engineers carry out matching of interface specifications manually; it makes the design and implementation of composite software expensive and error-prone. Therefore, the demand for instruments that would automate the matching procedure is high.

In this thesis, we develop a new technique facilitating integration of software systems. To this end, we solve a problem of how to construct visual and formal interface specifications comprising *semantic* descriptions. Our method also includes a matching procedure that checks compatibility of such specifications.

Interface specifications consist of structural and behavioral compartments. The structural compartment, given by a signature analogous to those appearing in algebraic specifications, defines operation declarations. The behavioral compartment, modeled by a conditional graph transformation system (GTS), contains operation contracts in the form of graph transformation rules. The rules of conditional GTS are equipped with *loose* semantics to describe operations in the import interface, and with *strict* semantics to describe those in the export interface. Composition of two compartments leads to an integral interface specification which is represented by the novel concept of *parameterized conditional GTS*.

We develop three kinds of compatibility relations underlying the matching procedure. The intended correspondence between declarations and contracts of the required and provided operations is reflected by *structural* and *behavioral* compatibility relations which are established over the corresponding compartments of interface specifi-

cations. These two compatibility relations are combined into an *integral* compatibility relation which links the integral specification of the import interface to the one of the export interface. Furthermore, the constructed relations are equipped with rigorously formulated semantic requirements to compatibility and are justified against them.

The introduced mathematical theory is supplemented with a *conceptual framework*. It is aimed at generating interface specifications that are suitable for automation of the matching process. The framework is based on an industry standard that outlines a uniform way of generating specifications. We use the standard issued by the Open Travel Alliance (OTA) in our example scenario where we develop and match standard-based interface specifications of Web services taken from the travelling business domain.

Compatibility of interface specifications is necessary but not sufficient for accurate interactions between systems. The integration process is based on the assumption that these systems are correct. First and foremost, this correctness means that interface specifications representing externally visible parts of systems are consistent with implementations which appear internally in the systems. To check this assumption, we propose a model describing external as well as internal parts of a system. The model, formally represented by a *graph transformation module*, defines consistency relations between external and internal specifications and allows to validate correctness of systems prior to the integration. The proposed model and the matching procedure developed in the thesis are the key elements of a technology designed to improve the application integration process, making it theoretically well-defined and practically machine-processable.

Acknowledgments

I owe a deep debt of gratitude to my doctoral advisor, Prof. Dr. Gregor Engels, for his invaluable assistance and patience to show me how to do research. I am thankful for his insights on the "big picture", for many technical details, and for very helpful comments on a draft of the thesis.

I would like to thank my co-supervisors Prof. Dr. Wilhelm Schäfer and Prof. Dr. Leena Suhl. They provided me with very useful comments on the initial ideas of the thesis which have been presented at the intermediate exam.

A number of people at the University of Paderborn helped me over the years. First of all, I owe thanks to Dr. Reiko Heckel (presently at the University of Leicester). Reiko is an endless source of information and insights on technical issues. He has a way of asking questions that open up new angels on a topic. He helped me to acquire self-confidence and add rigour to my thoughts. This thesis would have never been possible without his help and collaboration.

I thank Marc Lohmann and Sebastian Töne (presently at Trinkaus und Burkhardt) for their helpful suggestions in many interesting discussions, and also all the people in the Database and Information Systems group for their unconditional support and friendly interest. I am grateful to Vladimir Rubin who gave many valuable comments and have always encouraged me.

My work has been organizationally and financially supported by the International Graduate School "Dynamic Intelligent Systems" at the University of Paderborn. I feel very grateful to people from the staff of the graduate school, who keep things running smoothly and who cheerfully make all the administrative hassles go away.

Last but not least, I extend my heartfelt thanks to my family for support, patience, and faith in me.

Alexey Cherchago
Paderborn, February 2006

Contents

1 Introduction **1**

1.1 Integration of Software Applications 2

1.1.1 Mainframe-based and Client/server Systems 2

1.1.2 Middleware and EAI Platforms 3

1.1.3 Web Services 4

1.2 Problem Description 6

1.2.1 Semantic Markup for Web Services 7

1.2.2 Correctness of Coupled Systems 8

1.2.3 Problems Summary 10

1.3 Roadmap 10

2 An Approach to Service Specification and Matching **13**

2.1 Conceptual Framework 13

2.1.1 Intra- and Inter-company Application Integration 14

2.1.2 Dynamic Selection of Useful Services 15

2.1.3 Behavioral Specification of Service Operations 16

2.1.4 Standard-based Integration of Travelling Business Applications 19

2.1.5 Requirements 24

2.2 Service Interface Specifications 25

2.2.1 Structural Compartment of Interface Specification 26

2.2.2 Behavioral Compartment of Interface Specification 31

2.2.3 Integration of Structural and Behavioral Compartments 38

2.3 Matching of Service Interface Specifications 40

2.3.1 Structural Compatibility 41

2.3.2 Behavioral Compatibility 44

2.3.3 Integral Compatibility 47

2.4 Summary 50

3 Parameterized Conditional Graph Transformation 53
3.1 A Formal Account of Service Interface Specifications 53
3.1.1 Service Signature 54
3.1.2 Conditional Graph Transformation Systems 55
3.1.3 Parameterized Conditional Graph Transformation Systems . . 59
3.2 A Formal Account of Matching Service Interface Specifications 60
3.2.1 Signature Morphism 60
3.2.2 Substitution Morphism 61
3.2.3 Justification of Substitution Morphism 65
3.2.4 Parameterized Substitution Morphism 68
3.3 Summary 69

4 Modular Specifications of Coupled Systems 71
4.1 Software Component Models 72
4.1.1 Model-based Testing Techniques 72
4.1.2 Modeling Components by Module Specifications 73
4.1.3 Consistency of Interface Specifications 74
4.1.4 Requirements 75
4.2 GTS Morphisms, Systematically 76
4.2.1 Example Scenario 76
4.2.2 Candidates for Module Intra-connectors 78
4.2.3 Framework for GTS Morphisms 80
4.2.4 Locus of the Substitution Morphism 86
4.3 Application of GTS Morphisms 87
4.3.1 Extended Example Scenario 87
4.3.2 Intra-connectors of Modules 89
4.3.3 Inter-connectors of Modules 90
4.3.4 A Formal Account of Component Model 91
4.4 Summary 92

5 Related Work 93
5.1 Semantic-driven Specifications and Matching 93
5.1.1 Semantic Web Services 94
5.1.2 Approaches in the Travelling Business Domain 99
5.1.3 Graph Transformation for Service Specifications 100
5.1.4 Component Specification Matching for Software Reuse 101

5.2 Automation of a Matching Procedure . . . 103
5.2.1 Software Environments for Semantic Web Services . . . 103
5.2.2 Tool Support of the Matching Procedure based on Graph Transformation . . . 105
5.2.3 Implementation of Matchers for Software Reuse Approaches . . 105
5.3 Algebraic and Graph Transformation Modules . . . 107
5.3.1 Modularity Concepts in Algebraic Specification Languages . . . 107
5.3.2 Graph Transformation Modules . . . 108
5.4 Summary . . . 109

6 Conclusion and Future Work **113**
6.1 Summary and Evaluation . . . 113
6.1.1 Semantic-driven Integration of Software Systems . . . 114
6.1.2 Semantic-driven Integration vs. Stated Requirements . . . 115
6.1.3 Summary of the Proposal for Component Model . . . 116
6.1.4 Component Model vs. Stated Requirements . . . 117
6.2 Thesis Contributions . . . 118
6.3 Open Problems and Future Work . . . 119
6.4 Epilogue . . . 121

Chapter 1

Introduction

In the last decade, the Internet and Web have totally transformed the way business is conducted. New technologies eliminated communication barriers and allowed companies to exploit new business opportunities at the global electronic market place. Competitive pressures increased the need to improve operation of the enterprise software by consolidating different business units to form larger information fields. A deeper and more effective integration of software systems is currently is high demand.

Systems to be integrated are often described by means of *interfaces*. In this case, linking systems together means connecting an import interface of one system to an export interface of another system. To build up the required connection, the systems have to be proven compatible. To check compatibility, one has to match interface specifications of the systems.

The question whether application integration can be carried out successfully and effectively strongly depends on three main factors. The first factor is completeness of information published in the interfaces, i.e. syntactic and *semantic* characteristics of interacting systems. Secondly, the published information has to be machine-processable—it allows to check compatibility of systems (semi-)automatically. Finally, the interface specifications providing external system's descriptions have to be adequately related to internal system's implementations. To this end, it is necessary to construct a model which tracks consistency of the external and internal parts of a system thus ensuring correctness of coupled systems. From now on, by coupled systems we shall mean software components that require integration.

Our ultimate challenge is to establish a technique that addresses the factors mentioned above with an aim to expand and enrich the application integration process.

1.1 Integration of Software Applications

Once a company acquires a new business application, it is always desired or sometimes even required to integrate the new system into the existing IT infrastructure. There exists a wide range of integration techniques driven by different kinds of business requirements and technological innovations. Before identifying problems that accompany the application integration, we briefly review the current techniques.

1.1.1 Mainframe-based and Client/server Systems

In the early days of business computing, the most important question was how to automate a huge amount of previously manual operations such as payroll, order processing, or accounting. All these tasks were still to be managed, and the questions of interoperability or portability simply did not arise at this stage. In software systems, usually based on mainframes [122], the presentation, application logic, and resource management layers were merged into a single tier. An interaction with these *monolithic* systems took place through dumb terminals being the only entry points from the outside. In fact, the systems represented black boxes, and their integration was too expensive to develop and maintain.

The *client/server* paradigm [105, 118] gains momentum by the evolution of computation hardware and the emergence of PCs and workstations. Since it was no longer necessary to keep the presentation layer together with the resource management and application logic layers, the former was separated, and it could utilize the computational power of a PC. The result was a client/server system where the client had an ability to further process the information provided by the server (cf. Fig. 1.1 on the left).

Development of such systems forced software engineers to think in terms of published interfaces. In order to develop client applications and link them to a server, the server needed to have a known, stable interface specifying invocation requirements. Individual programs, running on a server, responsible for the implementation of application logic were called *services*.

An expansion of servers with stable interfaces posed new requirements that client/server systems could not address. While companies became more decentralized and geographically dispersed, servers created islands of information where a set of clients could communicate with only a limited number of servers. The increase in network bandwidth provided by local area networks (LANs) technically enabled

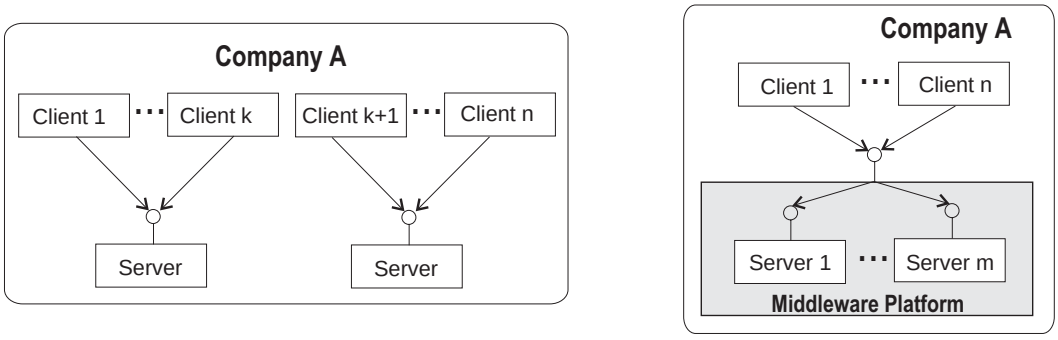


Figure 1.1: Client/server (on the left) and middleware-based (on the right) integration of applications.

combination of different servers. It allowed to expand their availability for clients, but a lack of proper infrastructure was still an obstacle.

1.1.2 Middleware and EAI Platforms

A platform for integrating a collection of servers under a common interface is known as middleware [13] (cf. Fig. 1.1 on the right). The use of middleware led to further expansion of services provided by servers. In fact, the functionality resulting from the middleware-based integration can be regarded as yet another service. Not only integration of servers took place but also integration of services implementing quite complicated business logic. In contrast to servers, almost no significant effort has been made to standardize interfaces of services. Integration of services, especially those provided by different middleware platforms, remained a challenging task. At the same time, as the number of LANs started to grow, and different branches within a company implemented their own middleware-based systems, the need for communication between different middleware platforms has become apparent. Enterprise application integration (EAI) has emerged in response to this need (cf. Fig. 1.2).

EAI [128] can be seen as a step forward in the evolution of middleware, extending its capabilities to cope with the integration of services provided by heterogeneous, coarse-grained applications possibly resided at different middleware platforms. Nowadays, EAI is based on two types of platforms—message brokers [9, 90] and workflow management systems [10, 89].

While the type of integration supported by EAI platforms has been implicitly limited to LANs, emergence of Web technologies enables information exchange on a large scale—in the Internet. A strong temptation to implement Internet-wide collaboration

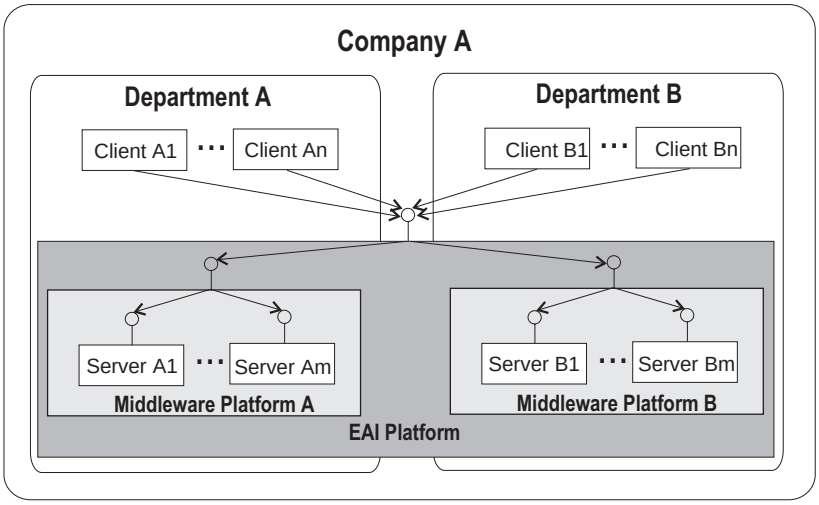


Figure 1.2: Application integration over EAI platform.

initiates efforts to integrate applications across the Internet. The EAI technologies were originally developed for intra-enterprise application integration, but the need to interact was not limited to the systems within a single company. Advantages that can be derived from automating company's business processes can also be obtained from automating business processes encompassing several organizations. It all lead to the current efforts around Web services—application development has shifted towards a service-oriented paradigm.

1.1.3 Web Services

Instead of dealing with the complexity of incompatible applications within EAI projects, a Web services platform [66] introduces a number of conventions the coupled systems must fulfill. Integration projects focus on composing externally uniform Web-based applications. Applications may be held by the same department or enterprise as well as by different companies—this approach supports both intra- and inter-enterprise application integration (cf. Fig. 1.3).

The Web service technologies represent state-of-the-art implementation of a service-oriented architecture (SOA) [112] being a style of software systems' design where services available in a network such as the Web play a role of the key organizational units. From a technical perspective, services are software modules or components with well-defined interfaces that clearly separate externally accessible interfaces from their internal implementations.

Three basic roles are usually distinguished in SOA-based interactions: service

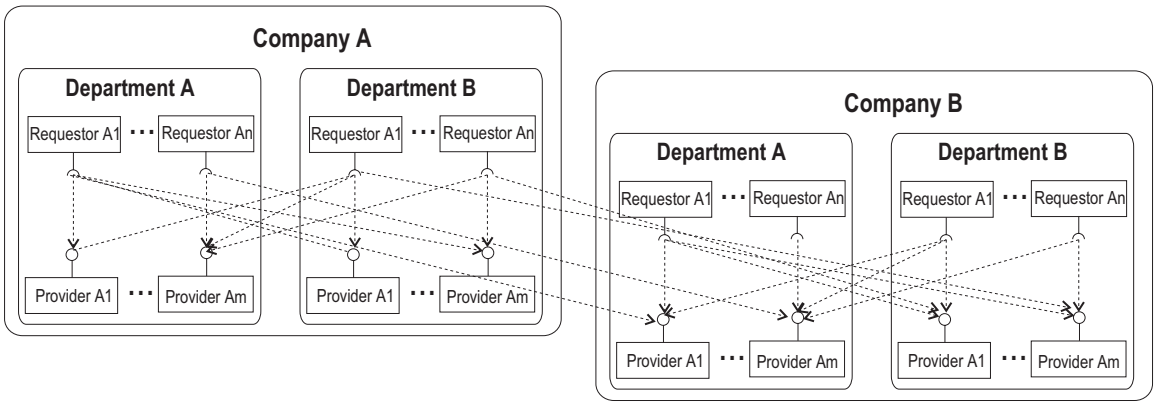


Figure 1.3: Application integration over Web services platform.

provider, service requestor, and service registry. A *service provider* is a software component that implements a service described in the provided (or export) interface specification. A *service requestor* is a component that intends to import some functionality satisfying the required (or import) interface specification. In order to obtain this functionality, the requestor invokes the provider's component through its interface, i.e. the two components are integrated.

A *service registry* defines a way to publish and lookup information about services. A provider disposes its service description in the registry which is queried by the requestors looking for services. The service registry returns to the requestor a list of service descriptions satisfying the submitted query, and the requestor reveals which of the obtained candidates is the most appropriate one. Registry and its contents can be organized in such a way that requestors are able to locate services not only at design-time but also at run-time that provides a sound background for the *dynamic* integration of applications.

According to the definition proposed by the UDDI Consortium in [141], Web services are *self-contained, modular business applications that have open, Internet-oriented, standard-based interfaces*. Such applications are designed to support interoperable machine-to-machine interaction over a network. This interoperability is gained through a set of XML-based open standards for communication, interface description, and discovery of services.

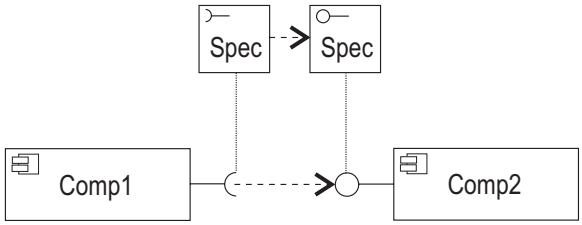


Figure 1.4: Matching of interface specifications.

1.2 Problem Description

The common objective of all application integration technologies is to supply the client or requestor with services implemented by individual servers or by a collection of servers aggregated under a common interface. Matching between the import/required interface (available only internally or specified externally) and the export/provided interface (typically specified externally) of coupled systems appears at the heart of all integration approaches (cf. Fig. 1.4).

Typically, interface specifications contain syntactic information on operations. Semantics of operations is described by means of plain textual annotations, therefore the interface specification matching can be carried out only by a software engineer. Thus, vagueness of semantic descriptions leads to the lack of tool support, and this makes the design and implementation of integrated software expensive and error-prone.

In order to improve the integration process, interface specifications have to be enriched with machine-processable semantic markups. This would enable automatization of matching procedure and leave out human intervention. In an attempt to develop an appropriate technique, one has to tackle two main issues: how to define semantic descriptions and how to match the interface specifications containing such descriptions. We consider these issues in the context of the Web services platform. This is due to the dynamic discovery of services (see Subsection 2.1.2 for a general discussion)—a technique that we believe can now be realized by the mechanism we propose. However, the obtained results can be used for application integration over other platforms as well.

The integration process is based on the implicit assumption that published interfaces are consistent with their internal implementations. But the mentioned assumption needs to be somehow justified. Here one should develop a model which integrally specifies coupled systems and, in particular, determines consistency rela-

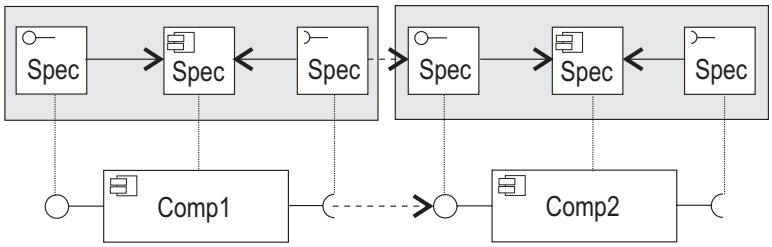


Figure 1.5: Integral specifications of coupled systems.

tions between internal and external specifications (cf. Fig. 1.5). Furthermore, this model lets us check correctness of systems participating in the integration.

The problems indicated above will be closely examined in the next two subsections.

1.2.1 Semantic Markup for Web Services

The World Wide Web Consortium (W3C) in [152] defines two aspects of the complete specification of a Web service: syntactic or *structural* description and service *semantics*. A number of ongoing projects in industry and in academic community are aiming to find out an appropriate form of semantics representation. At this point, however, technologies manipulate mainly structural characteristics.

Available Technologies

The Web Service Description Language (WSDL) [28] is proposed by W3C to specify services and their interfaces. An interface description includes a collection of service operations together with their input and output parameters, and technical characteristics of the service. This kind of information, referred to in [12, 152] as the "documentation of mechanics of the message exchanged", specifies the format of data transmitted between requestor and provider. However, by examining the WSDL description, we cannot unambiguously determine what the service does. We can see syntax of its inputs and outputs, but we do not know what these mean or what changes to the environment the service makes. Thus, semantics of a service is not covered by the WSDL standard.

The Universal Description, Discovery and Integration (UDDI) [141] specification defines a way to publish and locate information about Web services and extends the WSDL specification with semantic information—textual annotations and categorization data in a form of various keywords. While plain textual information is

not machine-interpretable, the keyword search assumes the categorization data or taxonomies to be fixed and universally recognized by integrating partners. However, this can only be the case if required and provided systems are developed in close coordination that is uncommon in the cross-organizational interactions.

The Semantic Web

The Semantic Web technology [151] is a promising solution of the semantics problem. It proposes to standardize the representation of semantic data on Web services via employing ontologies. An ontology is a formal definition of a common set of terms used to portray and represent a domain of knowledge [40]. Work on Semantic Web tends to revolve around industry standards enabling the ontology creation (e.g., Web Ontology Language (OWL) [39] or WSMO [57]). There exist techniques that use ontologies to semantically specify service capabilities (see, e.g., [107, 41, 43]).

In the above approaches, service semantics, in particular, semantics of service operations and their parameters, is determined either via referencing single elements of the corresponding ontologies or by means of logic expressions. In the former case, static constituents of the ontology can hardly reflect the dynamic nature of services—their behavior generates modifications in the internal environments of the integrating partners. In the latter case, logic expressions provide, in fact, structural restrictions for values of service characteristics rather then specify semantic annotations. This means they reside at the same level as WSDL specifications.

Thus, a technique extending structural specification of Web services with truly behavioral machine-processable semantic markup is still an open issue.

1.2.2 Correctness of Coupled Systems

Interface specifications reflect quite intricate structure comprising internal IT systems of companies. External specifications of coupled systems or components must be consistent with internal implementations, otherwise it does not make much sense to compare interface specifications.

Consistency of Interface Specifications

To achieve correspondence between internal and external parts, the creation of interface specifications should be adequately blended with the techniques employed for the development of company's applications. One should be able to derive interface

specifications from internal models or to carry out a consistency check between internal and external specifications. This consistency check is performed analogously to the comparison of interface specifications of coupled components. In either case, an interface description language has to conform with the one employed internally.

Model-based software development approaches, e.g., the Model Driven Architecture (MDA) [114] and a diagrammatic notation of the Unified Modeling Language (UML) [116], may serve as a sound base for description of internal specifications. Such approaches allow to construct platform independent models. It makes internal specifications comparable with external specifications that are free from the service's technical implementation. However, the model-based visual notations, such as the UML, have a common drawback—the lack of precise semantics. It significantly complicates automatic matching of such descriptions.

The use of models to align interface description with internal artefacts is advocated in [75]. While this approach discusses how to derive interface descriptions from internal models, it does not introduce relations ensuring consistency between internal and external descriptions. This can be done if the ideas of [75] are placed into a formal setting which clearly specifies the sought-for relations and their computing.

Models of Coupled Systems

It is obvious that a system resulting from the integration process would show an expected behavior only if its constituents function in the proper way. Therefore, integration always goes along with the need to analyze coupled systems or components and their specifications, both external and internal. In order to perform this analysis, one should have a model that describes a given component. To build up such a model, a structuring unit is required. We use a notion of a *module* [50] for this purpose.

In general, modules in specification or programming languages consist of *basic specifications*, like a component's body, export and import interfaces, and *intra-connectors* representing dependencies between these specifications. Following [82], in order to create a module description, one must find an appropriate language employed for the basic specifications and define relations being used for the intra-connectors. Once the module description of a component is given, it can be employed as an input for model-based testing techniques [21] to check correctness of the component expected to be a part of the compound system.

1.2.3 Problems Summary

Below, we concisely formulate the major concerns we intend to address in our presentation.

- How to enrich interface specifications of coupled systems with *semantic markups*.
- How to *match* interface specifications containing semantic markups.
- How to check *consistency* of interface specifications with the internal part of a system participating in the integration.
- How to construct a *model* which aggregates in its structure internal and external specifications, and tracks their consistency.

1.3 Roadmap

The presented work is structured as follows.

In Chapter 2, we introduce an approach to development and matching of interface specifications using an example scenario from the Web services domain. The example scenario contains interface specifications of the requestor and provider components for Web services that book flight tickets. The specifications are derived from a travel industry standard issued by the OpenTravel Alliance (OTA) [2]. We introduce an integral notion of structural and behavioral compatibility between the import interface of the requestor's component and the export interface of the provider's component. While structural information is represented by operation declarations, behavioral descriptions are given by contracts expressed as conditional graph transformation rules. The integration of structural and behavioral descriptions is facilitated by *typed and parametrized* graph transformation systems augmenting the rule-based descriptions of behavior by a type graph and operation declarations. The matching relation taking into account this combination is called an *integral compatibility relation*.

The construction of integral compatibility relation is based merely on the intuition reported by the example scenario. Chapter 3 formalizes the concepts from Chapter 2, allowing to determine semantic requirements to compatibility of systems interfaces (cf. Fig. 1.6). After reviewing some basic notions of the graph transformation theory, *parameterized substitution morphisms* are introduced as a formal counterpart of

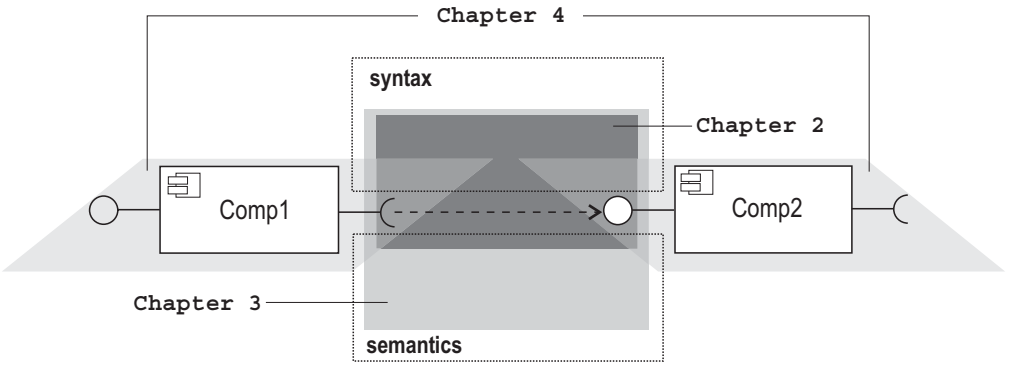


Figure 1.6: Overview of the thesis structure.

the compatibility relation. We demonstrate that substitution morphisms satisfy the rigorously formulated semantic requirements to compatibility.

Chapter 4 is concerned with constructing a model which fully specifies the systems to be connected (cf. Fig. 1.6). Here the behavior-driven descriptions employed so far for the import and export interfaces are reused to portray the interface implementations appearing in the components' bodies. First of all, we establish a framework that determines relations connecting external and internal component specifications. These relations guarantee consistency between specifications that underlies the correct interaction between the parties. Then, the import, export and body specifications as well as their connecting relations are aggregated in a model portraying entire integrating component. The model is given in the form of a graph transformation *module*.

To justify originality of the approach introduced in the thesis and identify its possible extensions, Chapter 5 provides an overview of related work and compares the concepts developed in this work with the existing proposals. In particular, we consider several approaches presenting semantic markups for interface specifications which originate in the Semantic Web, graph transformation, and Component-Based Software Engineering domains, along with software environments used to match these specifications. We also discuss component models that are formally portrayed by graph transformation and algebraic specification modules.

The concluding Chapter 6 summarizes and evaluates the main results of our work, concisely formulates practical and theoretical contributions, and indicates some open problems and directions for future research.

Bibliographical note: Preliminary results of this work have been published in [27, 53, 77, 78, 79].

Chapter 2

An Approach to Service Specification and Matching

This chapter examines the problem of dynamic integration of service-oriented applications deployed on the Web services platform. In particular, we focus on a dynamic service selection which amounts to matching the interface specifications of the required and provided services. Parameterized conditional graph transformation systems are introduced to support the automatic matching of interface specifications comprising structural as well as behavioral characteristics of services.

A standard-driven conceptual framework enabling service specification and matching is discussed in the next section along with an industry standard underlying this framework. This standard guides interactions among partners in the travelling business domain. It is used in Section 2.2 to construct an example scenario with the required and provided interface specifications of a Web service for booking flight tickets. Section 2.3 describes a matching procedure and illustrates it by means of specifications developed in the previous section. The final section of this chapter contains a summary of the presented ideas.

2.1 Conceptual Framework

Due to the unbounded diversity between the requestor and provider systems their interface specifications can not always be reconciled without manual assistance. Therefore, our presentation starts with a conceptual framework comprising a number of constraints which enable automatization. Foremost, these constraints can be derived from the intra- and inter-company application integration scenarios discussed below.

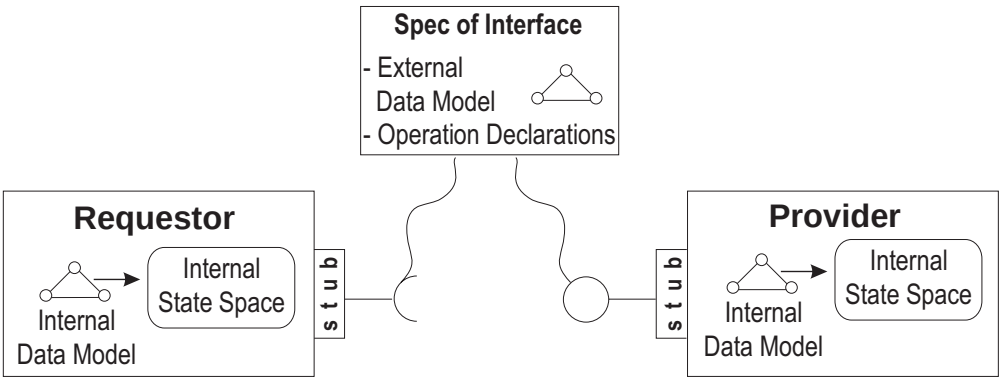


Figure 2.1: Integration of intra-company applications.

2.1.1 Intra- and Inter-company Application Integration

According to the definition given in Subsection 1.1.3, a Web service is a modular application, i.e. it consists of three basic parts: an import interface, an export interface, and a body. Body implements service(s) offered at the export interface, eventually using the features required at the import interface. An *interface description language* (alternatively interface definition language) (IDL) [5], is a common means to specify interfaces of modular systems. Different kinds of IDLs are usually considered in the context of application integration based on middleware and EAI platforms.

As already mentioned, a typical scenario supported by these platforms assumes the intra-company integration, where the required and provided systems are developed by the same or neighbouring teams. The following steps are distinguished in the integration process (cf. Fig. 2.1). Firstly, the service provider specifies the exported functionality in the IDL. It merely describes *structural* information, such as data types or a *data model* together with *declarations of service operations* in terms of their inputs and outputs. The second step is to compile the IDL description in order to construct stubs facilitating the interaction between the requestor and provider systems. Finally, the requestor may statically or dynamically connect to the provider application.

In the described scenario, the requestor does not explicitly specify its requirements to the desired service. The parties share the same interface specification (produced by the provider) that can be unambiguously interpreted by the requestor. Semantics of different operations, the order in which they should be invoked, and other possibly non-functional properties of services are assumed to be known in advance by the developer of the requestor system (or a person responsible for the system integration).

Moreover, an integration platform defines and constrains many aspects of the service description and interaction process. These aspects not specified as the part of service description become implicit. Thus, an IDL specification of the provided service containing only structural information is considered sufficient for integration purposes in intra-company setting.

The ultimate goal of Web services is to enable dynamic interaction in a completely open community of businesses, i.e. inter-company application integration, where the requestor application can automatically find adequate services and service providers, discover how to interact with the service, and finally invoke the service, all automatically, without manual intervention.

The first problem brought up by the dynamic interaction is a dynamic selection of a service by the requestor system.

2.1.2 Dynamic Selection of Useful Services

A dynamic service selection in cross-organizational interactions is thickened by the lack of implicit settings existing in the intra-company application integration scenario. So, the service interface descriptions have to cover aspects beyond the structural specification. However, the IDL for Web services, i.e. the Web Service Description Language (WSDL) [28], contains more or less the same amount of information as IDLs of the middleware or EAI platforms. Hereinafter, we will see that the structural service description is necessary but not sufficient to determine an integrability of the loosely-coupled systems.

The only information externally available on the systems intended to be integrated is their interface specifications. In order to ensure *compatibility* of requestor requirements with an offered service, the required and provided interface specifications have to be matched with each other.

The first aspect that must be tackled by the matching procedure is to reconcile data models issued in the interface specifications. Such data models, called *external*, abstractly portray the internal system data that is hidden in the opaque bodies of the requestor and provider components (cf. Fig. 2.2).

Due to the high heterogeneity of systems, their interface specifications and, in particular, external data models may arbitrarily diverge. Development of a procedure that would adequately relate any given data models is quite complicated problem—it can hardly be solved in general. Nevertheless, let us suppose for the moment that this problem is settled.

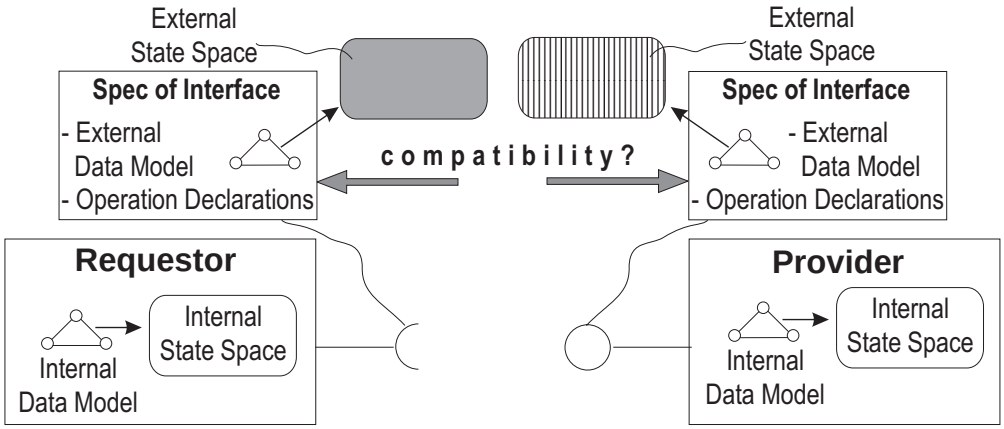


Figure 2.2: Integration of inter-company applications.

The next step deals with comparison of required and provided operations declared in interface specifications. For each required operation it is necessary to find a corresponding provided operation satisfying the requestor requirements.

On one hand, even operations with similar structure, i.e. identical input and output parameters, may be developed for completely different purposes. On the other hand, the WSDL specification of a service lacks for semantics of declared operations. Thus, in order to guarantee that a provided operation actually carries out what is expected by the requestor, structural descriptions of operations have to be accompanied with *behavioral* markups indicating missions of service operations.

2.1.3 Behavioral Specification of Service Operations

A *state space* abstraction is one of possible settings for behavioral specification of operations exposed by a software system. A data model of a system plays a role of a descriptor or generator for the state space, where different legal instances of the data model, i.e. specific system states, represent elements of the state space.

For example, an internal data model gives rise to an *internal state space* (cf. Fig. 2.2). An invocation and execution of an operation in some of the internal system states leads to a transition of the system into the next state. In this connection, the source and target of the transition are called *pre-state* and *post-state*, accordingly.

The external data model, in turn, generates an *external state space* which constituents are also related by the transitions. The transitions in the external state space are specialized by the corresponding internal transitions via extension of the former with implementation-related issues and concerns. A characterization of the

transition process can be employed for a technique portraying behavior of system operations.

Design by Contract Technique

In our approach behavior of operations is specified by *contracts* constituting a *behavioral compartment* of the service interface specification. Originally, the concept of a contract was introduced by Bertrand Meyer in an object oriented design technique called Design by Contract [108].

An operation contract consisting of *pre- and post-conditions* is used to ensure correctness of interaction between a supplier of operation and clients calling the operation. The pre-condition describes the conditions that must be fulfilled prior to operation invocation. The post-condition portrays the effect of the operation, i.e. state changes that occur when the operation completes successfully. Typically, pre- and post-conditions are assertions, e.g., Boolean expressions, stating some properties of the entities in the system states.

Web Services Contracts

The contracts introduced in the Design by Contract technique, however, diverge from the ones intended to be constructed for the service operations. Firstly, in the original approach only the supplier operation is equipped with the contract. It is evaluated at runtime by means of the input information accompanying the client call and outputs yielded by the invoked operation. Here a precondition violation is interpreted as a bug in the client, and a postcondition violation indicates a bug in the supplier.

In the Web services setting, both interacting parties have operation contracts, and their compatibility is not fixed beforehand. In the Design by Contract, on the contrary, the client development is guided by the supplier specification. Moreover, compatibility between required and provided systems is revealed at (dynamic) design time ante the actual interaction is launched.

Secondly, the contracts in [108] are assumed to be established over elements of the internal state space that are common for the supplier of the operation and for its callers. Due to modularity of requestor and provider systems, assertions defined over the internal state spaces must not be publicly displayed, since they contain private information. One can try to establish the contracts over the external state spaces. But that, however, makes another problem.

Sovereignty of integrated systems implies that their external state spaces and

contracts are also sovereign. It is meaningless to match such contracts, because they characterize behavior of operations in terms of distinct state spaces. Moreover, while the provider has exhaustive information on the context of transformations imposed by its operations, the requestor lacks for this information which is necessary for the contract construction. Since implementations of operations are beyond the scope of the requestor system, its internal and external data models may simply not contain the types constituting the context.

Thereby, to specify the contracts and to make them comparable with the offered service descriptions, the requestor needs some prior information on the external state space of the provider system. This information, however, can be obtained only if the parties know each other before the triggering of the service discovery process and that, in turn, contradicts the truly dynamic nature of integration of loosely-coupled systems.

An alternative way is to oblige the requestor and provider to determine their interface specifications in terms of some common framework. The role of such framework can be played by an industry standard prescribing formats of data recommended for the interactions in a specific business domain.

Standard-based Interface Specifications

The central position in such industry standard is occupied by a data model, called *standard data model*, facilitating a uniform way of data exchange between trading partners (cf. Fig. 2.3). A construction of a *standard-based service interface specification* starts with an abstract description of the internally used data via constituents of the standard data model.

While all elements of the standard-based data model of the provider descend from its internal data model, the standard-based data model of the requestor may extend its internal data model with the context elements representing the requestor assumptions on the state space of the provider. These elements come from the standard data model and are employed just to develop contracts of required operations.

The industry standard may also prescribe a number of *business activities*[1] that are typical for the domain. Each activity is accomplished with a textual explanation of its mission and a declaration of an operation automating this activity or a declaration of messages transmitted between interacting partners at the activity execution. The activity-related information is employed by the parties to establish *standardized*

[1] A definition of the term "business activity" can be found in [148].

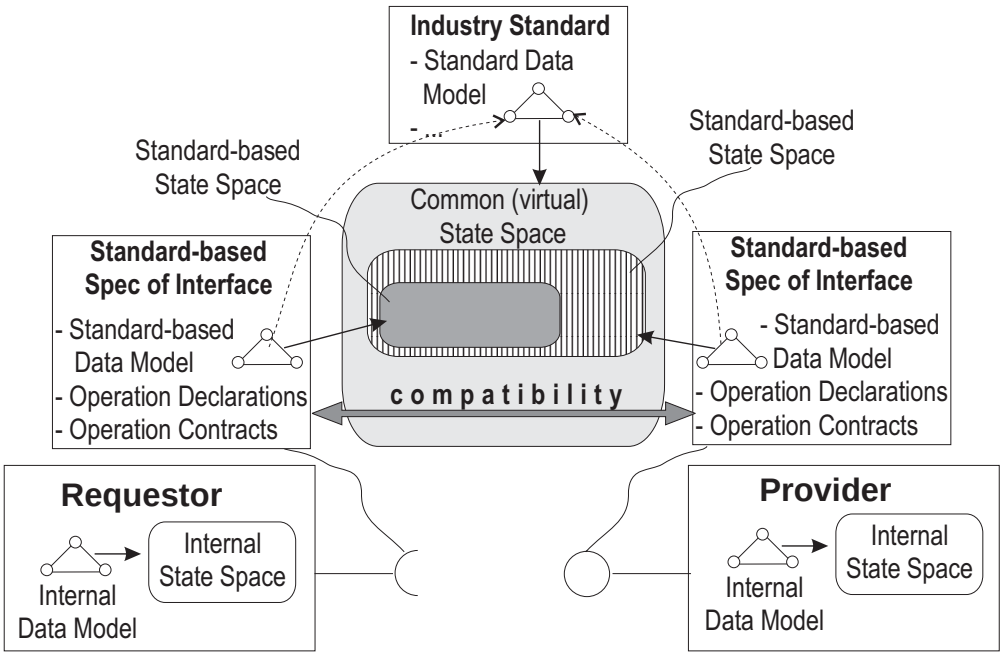


Figure 2.3: Standard-based integration of inter-company applications.

declarations of their own operations.

Then the declared operations are equipped with the contracts clearly determining their missions. The contracts are specified over state spaces generated by the standard-based data models. Due to the conformance of these data models to the standard one, the standard-based state spaces can be embedded into a *virtual* state space produced by the standard model (cf. Fig. 2.3). Now, the contracts of the required and provided operations express comparable constraints which can be matched in the scope of virtual state space playing the same role as the internal state space in the Design by Contract technique.

To give a more concrete example, an industry standard for the travelling business domain and its application for the development of standard-based service interface specifications are discussed in the next subsection.

2.1.4 Standard-based Integration of Travelling Business Applications

Nowadays, travel information services are offered mainly by Global Distribution Systems (GDS), such as Sabre [129], Galileo [63], Amadeus [6], or Worldspan [153]. As legacy systems, GDSs rely on their own private formats of data representation and

swapping which are significantly distinct in different GDSs. Heterogeneity of existing systems makes it difficult to carry out even static integration between partners.

Standardization efforts in travel industry are realized under the aegis of a consortium called the OpenTravel Alliance (OTA) [2]. The key players in the industry including airline, hotel, car rental, rail, and tour companies are involved in the OTA activity. OTA develops standards providing an agreed to format for exchanging data between and among travelers and travel-related businesses which, in turn, actively use the OTA specifications in practice. To date, there are at least three OTA compliant Web services being launched by Sabre [134], Datalex [120], and Galileo [133].

The standard data model and message declarations in the form of XML schemas are the main constituents of the OTA document. The messages portray availability checking, booking, renting, reservation, reservation canceling and modifying for the hotel, airline, and vehicle sectors. A part of the OTA standard addressing the airline flight related information will be used throughout this chapter to illustrate our approach.

In the rest of this subsection we demonstrate possible discrepancies between standard-based specifications of required and provided services that have to be reconciled by the matching procedure.

Compatibility of Service Data Models

To allow service requestors and service providers to interact in a consistent manner, first of all, it is necessary to relate their data models. If standard-based data models of the parties comply with the same version of the data model prescribed by the OTA specification, they actually represent different fragments of it and can easily be related. Unfortunately, it is not always the case.

Since 1998, when OTA was formed, it has issued ten versions of the standard, and it was continually refined and improved. Consequently, the standard data model is also evolving from version to version. That means even if requestor and provider specifications are OTA compliant, it does not guarantee that their standard-based data models are fully agreed.

It is evident that a common origin of data models intended to be related simplifies the matching procedure. However, the matching complexity can be additionally reduced, if vendors of the industry standards either provide a backward compatibility or establish a number of transformations between data models appearing in the different releases of the standard.

Compatibility of Service Operations

In addition to the standard data model, the OTA specification prescribes a number of business activities which typically appear at interactions among different travel services and their clients. The formats of request-response messages transmitted by the systems at executions of the activities are described by pairs of XML-schemas conforming with the standard data model.

Such a *message-oriented* specification of the interactions is usually applied to the systems adhering to a document-style of interaction, where data in exchanged documents is predefined by messages. An alternative way, that is common for a RPC-style of interaction, is to describe business activities via *operation declarations* [5]. In contrast to the message-oriented notation, where each unit of communication is represented by a single message, an operation declaration can be considered as a pair of request-response message schemas, such that the types of input and output parameters in the declaration correspond to the types of the elements constituting the request and response message schemas, accordingly.

While both notations are interchangeable, in our presentation we stick to the second one due to the following reasons. Firstly, the operation-oriented notation is quite popular in the conventional software modeling techniques, like the UML [116]. Secondly, it is supported by a wide range of formal methods and tools that can be reused in our work. Thirdly, the proposed approach to the behavioral specification of the service constituents is tailored to the operation-oriented specification of business activities and their after-effects.

Next, we consider two OTA-based examples to manifest the necessity of behavioral markups in service interface specifications.

Example 2.1.1. The XML schemas OTA_AirBookRQ and OTA_AirBookRS, fragments of which are visually represented in Fig. 2.4, specify the format of request and response messages for the OTA business activity AirBook allowing to book a flight. Here the items framed by solid lines stand for the types of compulsory message elements, by dashed lines for the optional ones.

Templates defined by the XML schemas can be used by developers of interface specifications to establish standardized declarations of service operations. By the standardized operation declaration we mean that a pair of request-response message schemas from the OTA specification constrains the types which may be used in the declaration. In particular, elements in the request message schema predefine the

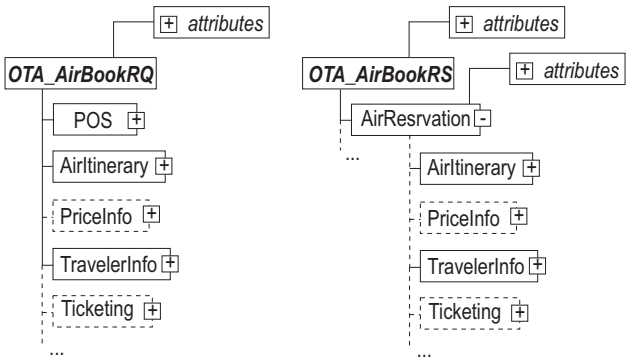


Figure 2.4: Request and response message schemas for the OTA business activity AirBook.

types of input parameters, and elements in the response message schema—the types of output ones.

Since a message schema is composed by compulsory as well as optional elements, we assume that a legitimate operation declaration must contain at least parameter types corresponding to compulsory elements of message schemas, and all other parameter types (if any) must refer to optional elements of message schemas. This means that each pair of request-response message schemas gives rise to a number of legitimate operation declarations that are different from each other.

In general, interface specifications comprise the standardized declarations as well as *user-defined* ignoring the standard message schemas. Moreover, even business activities precisely portrayed in the industry standard may be specified by user-defined declarations due to the disparity of conventional templates with requirements of a service developer.

Sample declarations over the message schemas OTA_AirBookRQ/RS shown below specify the required operation airReserv and the provided operation airBook destined for a flight booking:

airReserv:POS,AirItinerary,TravelerInfo,Ticketing → AirResrvation /*requestor
airBook:POS,AirItinerary,TravelerInfo → AirResrvation /*provider

Both declarations are legitimate, because they contain the input types POS, AirItinerary, TravelerInfo and the output type AirResrvation that are compulsory elements of the message schemas in Fig. 2.4. In addition, the input of the required operation is extended by the parameter type Ticketing appearing as the optional element in the message schema OTA_AirBookRQ.

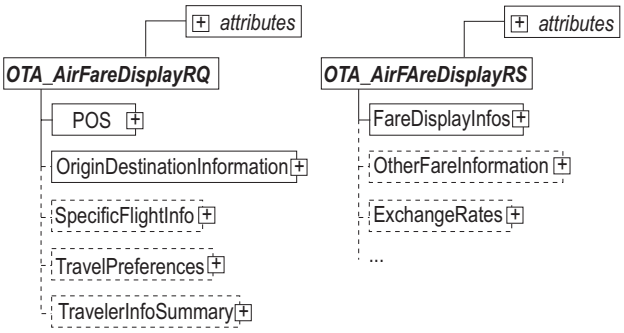


Figure 2.5: Request and response message schemas for the OTA business activity AirFareDisplay.

In spite of structural differences, such as extra input in the required operation and different operation names, the aim of both operations is analogous to the one stated in the OTA standard. If structural differences do not entail any problems for interactions between the parties, it is reasonable to expect that a matching procedure includes the provided operation in the list of candidates that may be successfully used by the service requestor. However, the fact that operations carry out the same functions can not be automatically detected without behavioral markups attached to the operations. △

The presented example demonstrates that compatibility between operations of required and provided services does not always require structural identity of the compared operations. The next example shows that even the operations with coincident declarations may expose different behaviors causing their incompatibility.

Example 2.1.2. Two declarations placed below specify the required and provided operations for a business activity meant to display fares between a given city pair:

airFare:POS,OriginDestinationInformation → OriginDestinationOptions /*requestor
airFare:POS,OriginDestinationInformation → OriginDestinationOptions /*provider

The declaration of the provided operation is aligned with the current version of the OTA standard [4] containing the message schemas OTA_AirFareDisplayRQ and OTA_AirFareDisplayRS for this business activity (cf. Fig. 2.5). The requestor, in turn, employs one of the earlier versions of the standard (e.g., [3]), where this business activity is absent.

Due to the OTA documentation, an execution of the business activity is not accompanied with an inventory check for available seats on flights fares of which

are displayed. While the provided operation has been implemented according to the industry standard, the requestor, for example, may expect that the inventory check is carried out.

It is worth to mention that this muddle may cause serious problems when the parties start to interact. To avoid such problems, one should compare the contracts of the operations even if the expected behavior could seem to be identical. △

To summarize, a standard-based service interface specification consists of a structural compartment with (standard-based) data model and operation declarations, and a behavioral compartment containing operation contracts (cf. Fig. 2.6). Compatibility of required and provided specifications assumes a check of structural and behavioral resemblance of the corresponding specifications' compartments. Thereafter, a pair of service interface specifications with structural (behavioral) descriptions that are apt to each other is called *structurally (behaviorally) compatible*. Service interface specifications are called *integrally compatible* if they are structurally and behaviorally compatible.

2.1.5 Requirements

Before we turn to the detailed discussion of our approach, we collect a number of requirements guiding our work.

1. *Formal visual notation(s) for service interface specifications.* Visual diagrammatic notations play an important role in the design and understanding of complex software systems. Structured Analysis [38], UML [116], SDL [56], IEC Function Block Diagram [88] are prominent examples of such visual modeling techniques which are daily exploited in industry.

   While the graphical syntax of such notations makes them suitable for human comprehension, their semantics is not clearly (formally) stated. This impedes automatic reasoning over properties of graphical specifications. Our purpose is to find a technique which absorbs intelligibility of graphical notations together with the precision of formal methods enabling automation.

2. *Compliance with standard model-driven techniques of software development.* Model-driven software development (MDSD) [114] becomes prevalent in software engineering, where one of the key places is occupied by graphical languages, such as the UML. The prosperity of newly introduced techniques de-

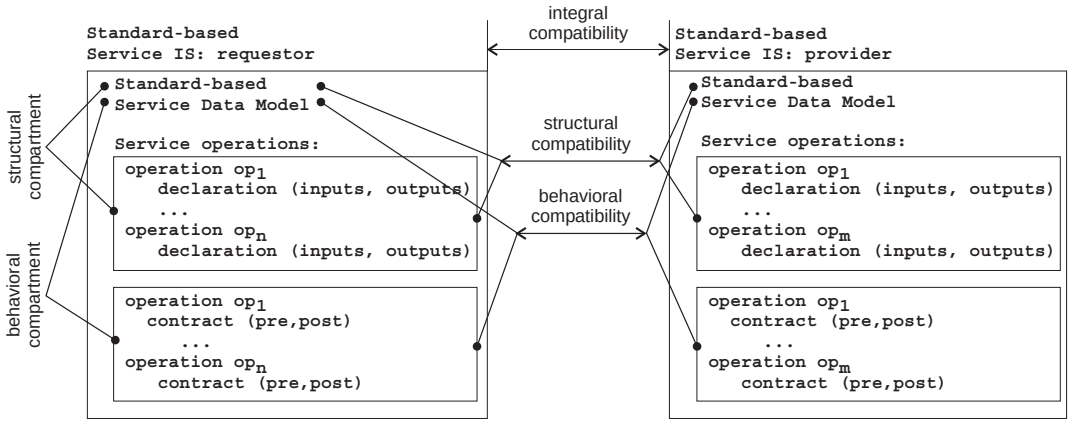


Figure 2.6: Compatibility of service interface specifications.

pends on their integrability and interoperability with existing methods used by software companies at the conceptual as well as the notation layers.

3. *Compliance with the technological stack of the Web services platform.* Since we are developing our approach in the platform-independent manner (at the level of models), its concepts have to be translated into the technological stack of the Web services platform consisting of XML-based standards, such as WSDL. While this problem is beyond the scope of the presented work, the possibility of such translation has to be taken into account.

4. *Modularity and extensibility of service interface specifications.* In our work we concentrate on the two aspects of services (structural and behavioral) that are placed in the corresponding compartments of interface specifications. This structure of the specification allows to clearly distinguish between different perspectives of service specification and extend it with additional compartments containing information, e.g., about required or provided business processes, exception conditions and error handling information, security profile, transactional profile and recovery semantics, service-level management agreement, etc.

We shall keep in mind the above ideas while we scrutinize service interface specifications and their constituents.

2.2 Service Interface Specifications

In this section, we present a lightweight formalization of interface specifications and establish an example scenario comprising the required and provided interface spec-

ifications of a Web service for booking flight tickets. The sample specifications are developed according to the OTA standard that was discussed in the previous section.

A structural compartment of the interface specification is shown in the next subsection. Subsection 2.2.2 demonstrates a behavioral compartment containing service operation contracts in the form of conditional graph transformation rules. Integration of structural and behavioral compartments is introduced in Subsection 2.2.3, where the service interface specification is modeled by a parameterized conditional graph transformation system.

2.2.1 Structural Compartment of Interface Specification

As already mentioned, the first constituent of the structural compartment in required and provided interface specifications is the standard-based data model (cf. Fig. 2.3), also called ontology in the context of semantic Web [59].

Data Model

In general, the data model describes data types and their relationships shared and reused across a detached application, a system of applications or an entire business domain.

Example 2.2.1. A structural compartment of the provided interface specification for the flight reservation service is shown in Fig. 2.7. The interface specification is aligned with the OTA standard version 2005A [4] describing the business domain data in the form of XML schemas. To meet Requirement 1 of Subsection 2.1.5, the XML schemas are translated into a visual notation resembling the simplest form of UML class diagrams. The translation is carried out with the help of the Eclipse plug-in *hyperModel* [23] enabling bi-directional transformations between XML schemas and graphical UML models.

The upper part of Fig. 2.7 represents a fragment of the data model obtained as the output of the tool *hyperModel*. While the type names in the data model originate in the OTA standard, the automatically assigned names of relationships between the types have been changed to more indicative ones to increase readability of the model.

The constituents of the data model are interpreted as follows. A point of sale (type POS) or travel agency is authorized by a passenger (type TravelerInfo) to book a ticket for an air itinerary (type AirItinerary) between a pair of locations (type OriginDestinationInformation). Each itinerary consists of a number of segments (type Flight-

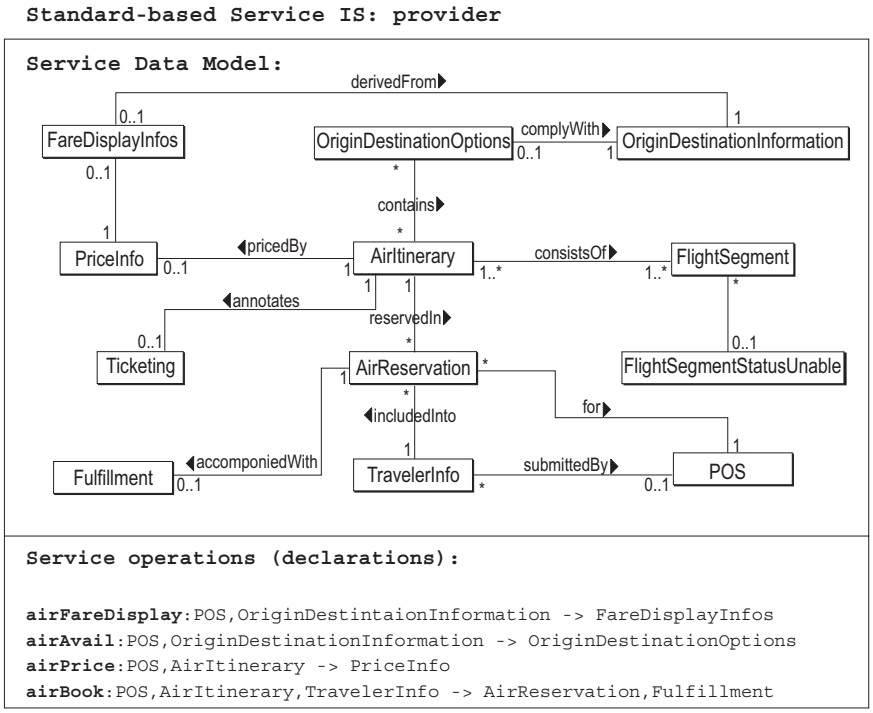


Figure 2.7: Structural compartment of the provided interface specification: data model (top) and operation declarations (bottom).

Segment) which are annotated with a status showing availability of vacant places (type FlightSegmentStatusUnable).

In the OTA standard, the flight segment status is defined as the FlightSegment attribute which ranges over the values such as available or OK (number of vacant places is equal to or exceeds some predefined constant), wait_list_open (number of vacant places is less then some predefined constant), unable (no free places), etc. Due to the fact that our approach does not explicitly support the attributes, the status information in our data model is reflected by the flag FlightSegmentStatusUnable indicating that the flight segment is unable for booking.

An air itinerary may be equipped with a ticketing information (type Ticketing), such as flight segment or passenger reference numbers, used to carry out the ticket arrangement. A successful completion of booking is indicated by a reservation tag (type AirReservation) and payment information being relevant to the booking (type Fulfillment). △

Our main intention is to construct a procedure which allows to automatically match service interface specifications. It is assumed that required and provided interface specifications are compared without any assistance of software engineers. For this, we need to find an appropriate formalism for each constituent of the interface specification.

Due to the notation which is commonly used for specifying data models, it is reasonable to employ the notion of a *graph* for the formal representation of such models. A graph G is usually described by a set of vertices G_V and a set of edges G_E. These sets are constructed in such a way that each edge e in G_E has a source vertex $src(e)$ and a target vertex $tar(e)$ in G_V. We say that a graph G is a *subgraph* of a graph H, denoted by $G \subseteq H$, if $G_V \subseteq H_V$, $G_E \subseteq H_E$, $src_G(e) = src_H(e)$, $tar_G(e) = tar_H(e)$, for all $e \in G_E$.

Vertices and edges of the data model graph, called a *type graph*, contain type declarations and relationships between these types. A type graph TG serves as a schema which generates a number of *instance graphs* connected with TG via a structure preserving mapping. This mapping ensures compliance of instances with the structural properties encoded in TG and associates with each vertex and edge x of the instance graph G its type t from TG. We write $x : t$ to denote that the element x of the graph G has the type t.

Example 2.2.2. A type graph TG representing a fragment of the provider's data model and an instance graph G are shown in Fig. 2.8. A structure preserving mapping

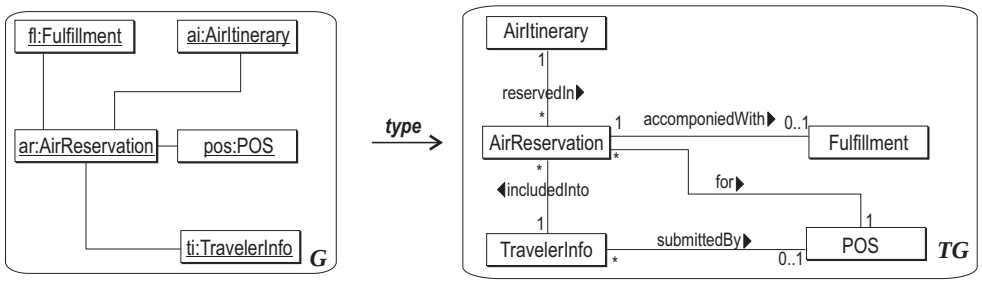


Figure 2.8: Instance graph (left) typed over type graph (right).

between G and TG can be specified by defining $type(v) = t$ for each vertex $v : t$ of G. Extending this to the edges of G, preservation of structure means that, for example, an edge between vertices v_1 and v_2 must be mapped into an edge in the type graph TG between $type(v_1)$ and $type(v_2)$. Usually, the types of edges in the instance graph are not explicitly indicated, since they can be easily derived from the types of vertices.

Providing analogy with object-oriented modeling, the type graph can be viewed as a class diagram, and the instance graph as an object diagram. △

Operation Declarations

In addition to the data model, the structural compartment of interface specification contains declarations of operations composing a service interface.

Example 2.2.3. Declarations of provided operations for the flight reservation service are shown in the lower part of Fig. 2.7. The first operation airFareDisplay obtains an identifier of the travel agency (POS) together with codes of departure and arrival airports (OriginDestintaionInformation) as input, and returns fares of the flights serving the chosen route (FareDisplayInfos). Having the same input, the second operation airAvail displays all available flights (OriginDestinationOptions) between given locations. Both operations are defined by the standardized declarations over the message schemas OTA_AirFareDisplayRQ/RS and OTA_AirAvailRQ/RS, accordingly.

The operation airBook requires a caller to submit its identifier (POS), a passenger information (TravelerInfo) and a code of the itinerary intended to be booked (AirItinerary). It yields an acknowledgment on the reservation (AirReservation) together with payment details (Fulfillment). The declaration of this operation is user-defined, because its output contains a parameter with the type Fulfillment, which does not appear as an element in the message schema OTA_AirBookRS depicted in Fig. 2.4. △

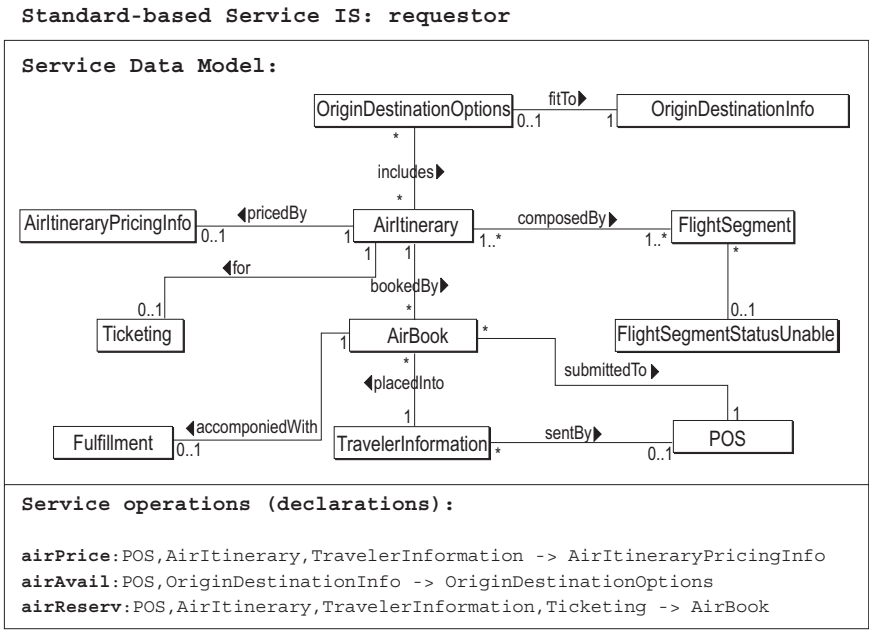


Figure 2.9: Structural compartment of the required interface specification: data model (top) and operation declarations (bottom).

Analogously to the data model, the second constituent of the structural compartment has to be equipped with a formal counterpart enabling automatic matching. Having this objective, an operation declaration is modeled by the expression $p : v \to w$ which consists of an operation name p and sequences of input $v = v_1, \ldots, v_n$ and output $w = w_1, \ldots, w_m$ parameter types from the type graph TG. A pair $\mathcal{S} = \langle TG, P \rangle$, where TG is a type graph and $P = (P_{v,w})$ is a family of sets with operation declarations, is called a *service signature*.

Example 2.2.4. The OTA standard version 2001C [3] guides the construction of the required interface specification depicted in Fig. 2.9.

A fragment of the data model yielded by the tool *hyperModel* processing the XML data schemas of this standard are given in the upper part of Fig. 2.9. In general, the required data model is quite similar to the provided one. However, due to the lack of compatibility between the issues of the OTA standard, the semantically identical types in the models have different identifiers. For example, the types AirBook, AirItineraryPricingInfo, and TravelerInformation from the OTA standard version 2001C appear in the version 2005A under the names AirReservation, PriceInfo, and TravelerInfo, accordingly.

The required operation declarations are depicted in the lower part of Fig. 2.9. The

first operation airPrice calculates a total price (AirItineraryPricingInfo) of an itinerary (AirItinerary) booked for the specific number of passengers (TravelerInfo). The mission of this operation is different from the one of the provided operation airFareDisplay which simply returns flight fares. The next two operations airAvail and airReserv serve for the same purposes as the operations airAvail and airBook in Fig. 2.7, though their declarations are slightly different.

The standardized declaration based on the message schemas OTA_AirAvailRQ/RS describes the required operation airAvail. The declarations of the two remaining operations from the required interface are user-defined. △

Note that in contrast to the object-oriented technology, where each operation is defined in the context of a specific class implementing this operation, the declared service operations are not equipped with their implementers. This kind of information refers to the internal characteristics of a system and therefore should not appear in the interface specification.

While the service signature specifies the structural aspect of the required or provided functionality, the behavior of a service shall be described by contracts established for the service operations.

2.2.2 Behavioral Compartment of Interface Specification

There are different approaches to contract specification employing logic-based descriptions [96], algebraic specification languages [26, 156], etc. (see Chapter 5 for a general discussion). The problem is that these formalisms are rarely applied by software engineers due to the lack of skills in formal methods.

All the existing techniques do not meet our requirements (cf. Subsection 2.1.5 requirements 1 and 2). We aim at a notation that is close to standard software modeling languages and at the same time has a formal semantics. This visual formal notation for contracts is provided by *typed graph transformation* [35, 36]. In the following we extensively use graph transformation theory and introduce a number of new concepts needed for our approach.

Typed Graph Transformation

The idea of typed graph transformation is to see run-time states as directed graphs, typed over a type graph TG representing the data model. State changing operations are described by *graph transformation rules* (or productions) $p : s$ consisting of a rule

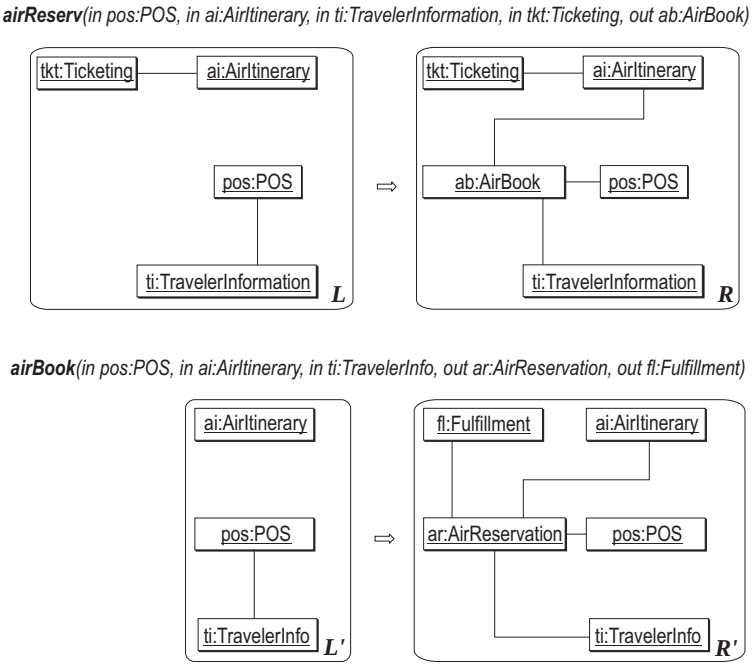


Figure 2.10: Graph transformation rules airReserv (top) and airBook (bottom).

name p, and a span $s = (L \supseteq K \subseteq R)$, where L, K and R are TG-typed instance graphs. The left-hand side L and the right-hand side R describe a part of the system state before and after execution of the operation, that is, the pre- and postconditions, and the context graph K contains those elements that are read but not deleted by the operation. For a rule $p : s$ we usually assume that the graph K is the intersection $K = L \cap R$. In this case, we denote the rule by $p : L \to R$.

Example 2.2.5. Graph transformation rules specifying the required operation airReserv and the provided operation airBook are shown in Fig. 2.10. They represent the first attempt to describe behavior of the corresponding operations and will be refined later. The left-hand sides of the rules contain elements standing for the input parameters of the operations and their relationships. For example, the edges between the objects ti:TravelerInformation and pos:POS in the requestor rule, and the objects ti:TravelerInfo and pos:POS in the provider rule denote the fact that the passenger profile is submitted by a specific travel agency identified by the point of sale code. The output parameters of the operations appear in the right-hand sides of the rules.

Note that the rule names are followed by sequences of elements defining the parameters of the specified operations. These sequences actually represent parameter expressions which will be discussed in the next subsection. △

A system migration from the state G to the state H under the execution of an operation specified by the rule p is modeled by a graph transformation step. A *transformation step* $G \stackrel{p}{\Longrightarrow} H$ from G to H using a rule $p : L \rightarrow R$ requires that a renaming of L occurs as a subgraph in G. Then, $L \setminus R$ (which consists of all nodes and edges of L not belonging to R) is removed from G, and $R \setminus L$ is added to the result. This leads to the derived graph H which contains a renaming of R as a subgraph. The rule application is only permitted if after the deletion step, the resulting structure is a graph again.

The effect encoded in the rule is defined by the elements which have to be deleted (exist only in L), created (exist only in R), and preserved (exist in K) under the rule application. The deleted and created elements are denoted by $del(p)$ and $add(p)$, accordingly.

The application deletes and creates exactly what is specified by the rule. There exists an implicit *frame condition* stating that everything that is not rewritten explicitly by the rule is left unchanged. Due to this fact, the rule semantics described above is called *strict*.

Example 2.2.6. Fig. 2.11 demonstrates a transformation step via the rule airBook. First of all, we look for an occurrence of the left-hand side L' of the rule in the typed graph representing (a fragment of) the system state. This occurrence is marked by the dashed rectangles in the source graph G. Then the elements matching $del(airBook)$, i.e. the edge between the objects 07G4325:POS and id7H:TravelerInfo, are deleted, and the elements corresponding to $add(airBook)$ marked by the dashed rectangle in the target graph H are added to the result. The newly created objects 7H39B4:AirReservation and 7H39FL:Fulfillment are obtained as the output of the provided operation specified by the rule airBook. The elements of the graphs G and H which match the elements of the graphs L' and R' corresponding to the parameters of the operation airBook follow the name of the rule in the figure. △

Loose Semantics of Graph Transformation Rules

The strict rule semantics is pertinent for the contracts of provider, who obviously has complete information on supplied functionality. The required contracts are incomplete specifications of service behavior, because a developer of the required system has only a loose idea of provided services. In particular, context elements which form the requestor assumptions on the provided behavior may be underspecified. Loose or

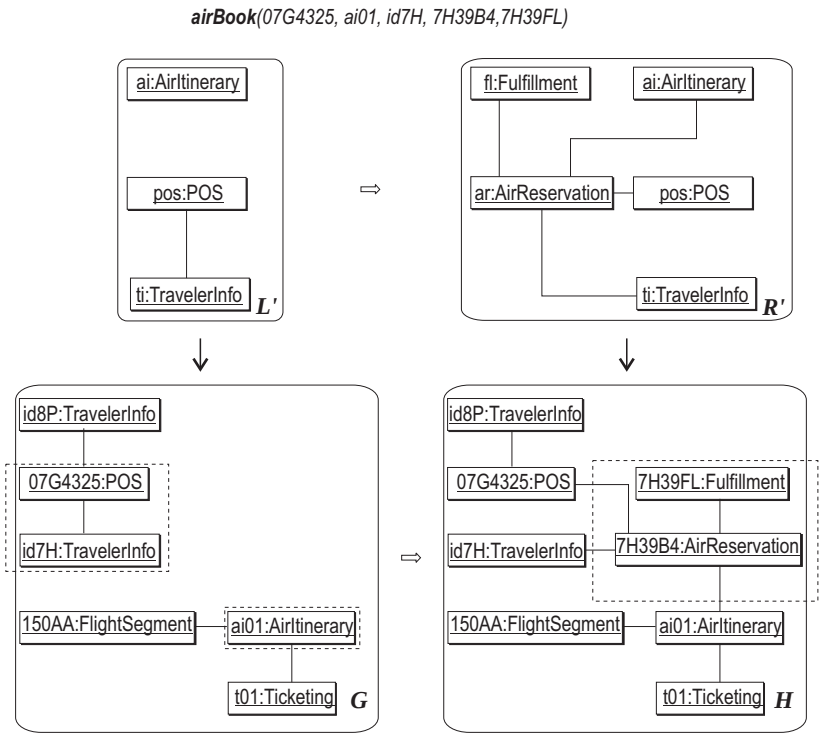


Figure 2.11: A sample transformation step via the rule airBook.

underspecified contracts describe minimally admissible effects that can be exceed in more powerful provided operations.

Therefore, contract rules of required operations have to be interpreted in a more liberal way: *at least* the elements of the graph G matched by $del(p)$ are removed, and *at least* the elements matched by $add(p)$ are added. This rule semantics, called *loose*, introduces the notion of a *graph transition* which leaves out the frame condition.

Like a transformation step, a graph transition $G \overset{p}{\rightsquigarrow} H$ from G to H via p requires that L occurs in G and carries out the modifications of G being explicitly encoded in the rule, but there may be unspecified deletions and additions as well.

Example 2.2.7. A sample graph transition is shown in Fig. 2.12. It applies the rule airReserv while in parallel the edge between the objects t07:Ticketing and ai07:AirItinerary is deleted, and a new edge between the objects t07:Ticketing and 8P39B4:AirBook is created. The "spontaneous" deletion and creation illustrate an effect which is unspecified by the rule airReserv. Analogously to the previous example, the name of this rule is followed by the elements of the graphs G and H corresponding to the elements representing the parameters of the operation airReserv. △

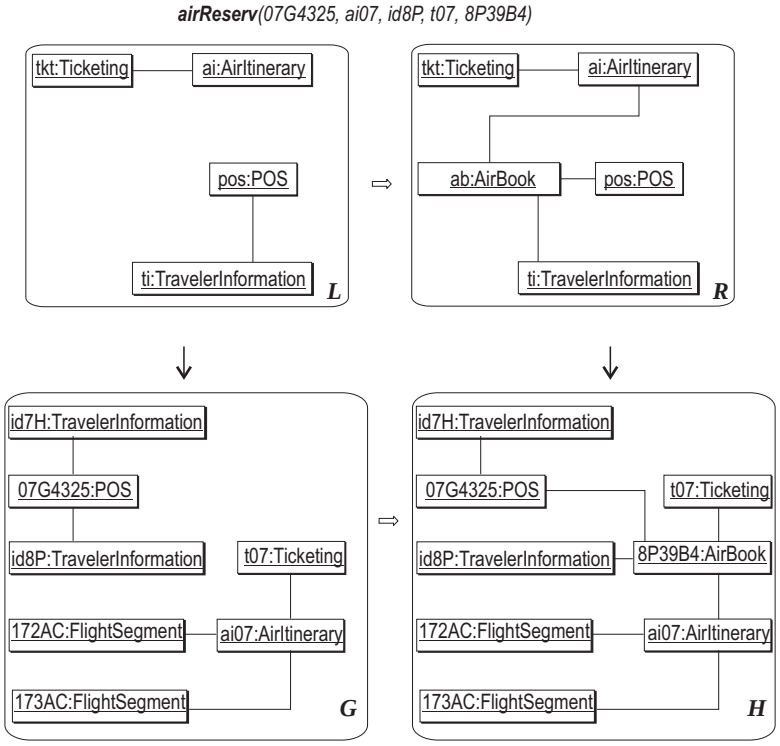


Figure 2.12: A sample graph transition via the rule airReserv.

Graph Transformation System

While the behavior of a single operation is described by a graph transformation rule, several behavioral descriptions can be aggregated in a (typed) graph transformation system (GTS). A *graph transformation system* $\mathcal{G} = \langle TG, P, \pi \rangle$ augments TG-typed spans $s = L \to R$ with a type graph TG and unique identifiers p from a set of rule names P, where correspondence between the spans and rule names is defined by a mapping π.

Remark 2.2.8. Our notions of graph, transformation rule, transformation step, etc. are standard in the double-pushout (DPO) approach to graph transformation [36] (see also Subsection 3.1.2) which provides strict rule semantics.

Graph transitions are introduced in [81] (see also Subsection 3.1.2). Technically speaking, they are based on a double-pullback (DPB) construction, unlike graph transformations that are classically defined using a double-pushout construction. The DPB approach introduces a more general loose rule semantics which allows unspecified changes under application of rules. △

Conditional Productions

As we will see in the next section, behavioral compatibility of required and provided operations is ensured via matching of precondition and effect parts of contract rules. Thus, preconditions and effects of rules have to be clearly distinguished. However, the specification of the precondition is mixed up with the part of effect (deleted items) in the left-hand side of the rule.

A refinement of the rule's structure avoiding this problem is obtained by using *positive application constraints* in the form $L \subseteq \hat{L}_P$, where L is the left-hand side of a rule span and $\hat{L}_P$ is a *positive precondition pattern*. The elements constituting $\hat{L}_P$ compose a context required for the rule application.

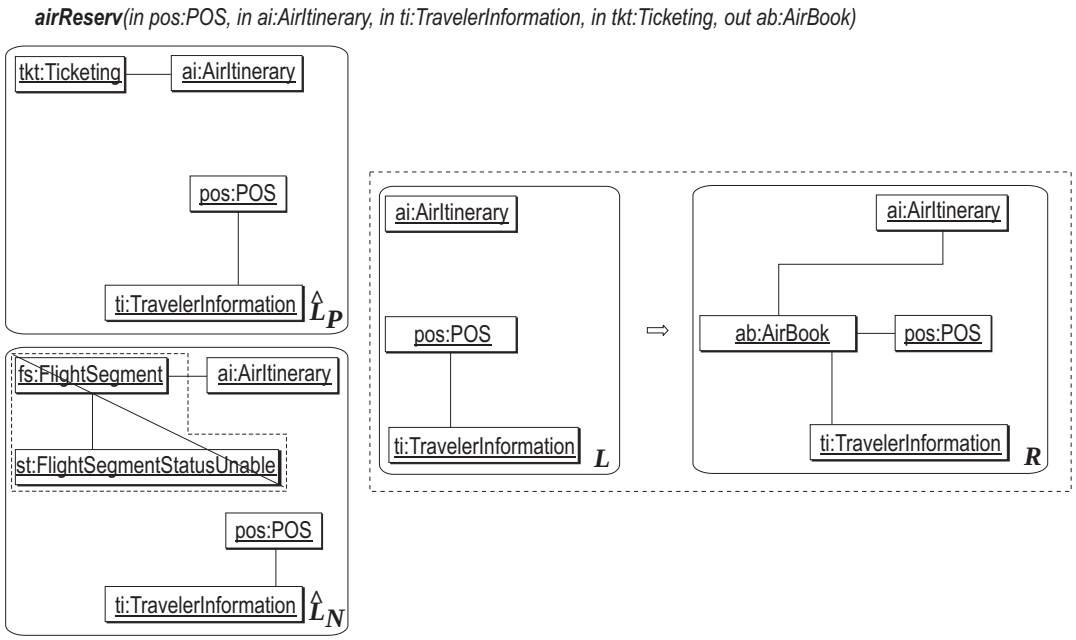
Note that in case of strict semantics, use of positive constraints does not increase expressiveness of rules. Positive constraints can be integrated by extending both the left- and the right-hand side with the elements required, but not deleted by the rule. This is correct because we are sure that everything that is not explicitly deleted is not deleted at all.

This is no longer true under loose semantics, where positive precondition may extend the left-hand side of the rule with elements that are required to be there, but may be deleted spontaneously, without explicit specification.

Positive constraints allow us to assert an *existence* of patterns in graphs. Quite frequently though, it is necessary to express that something must not be the case for a rule to be applied. However, the language introduced so far does not allow to specify *non-existence* of patterns. To fill this gap, the expressive power of the contract language is increased by *negative application constraints* $L \subseteq \hat{L}_N$, where $\hat{L}_N$, called a *negative precondition pattern*, extends L with the elements that must not be present in a graph when the rule is applied.

An *application condition* over the rule span $s = L \rightarrow R$ is given by a set $A(s) = \{\hat{L}_P, \hat{L}_N | L \subseteq \hat{L}_{P/N}\}$ being a part of conditional graph transformation rule. A *conditional graph transformation rule* is an expression of the form $p : s \text{ if } A(s)$, where p is a rule name, s is a rule span, and $A(s)$ is an application condition over this span.

A transformation step or transition via a conditional rule also needs to find a match of L in G. Then it is necessary to check whether occurrences of positive (negative) precondition patterns corresponding to the chosen match are contained (are not contained) in the source graph G. If it is the case, we say that the match of L in G satisfies the application condition and proceed analogously to the unconditional case. Otherwise application of the rule at the chosen match is forbidden.



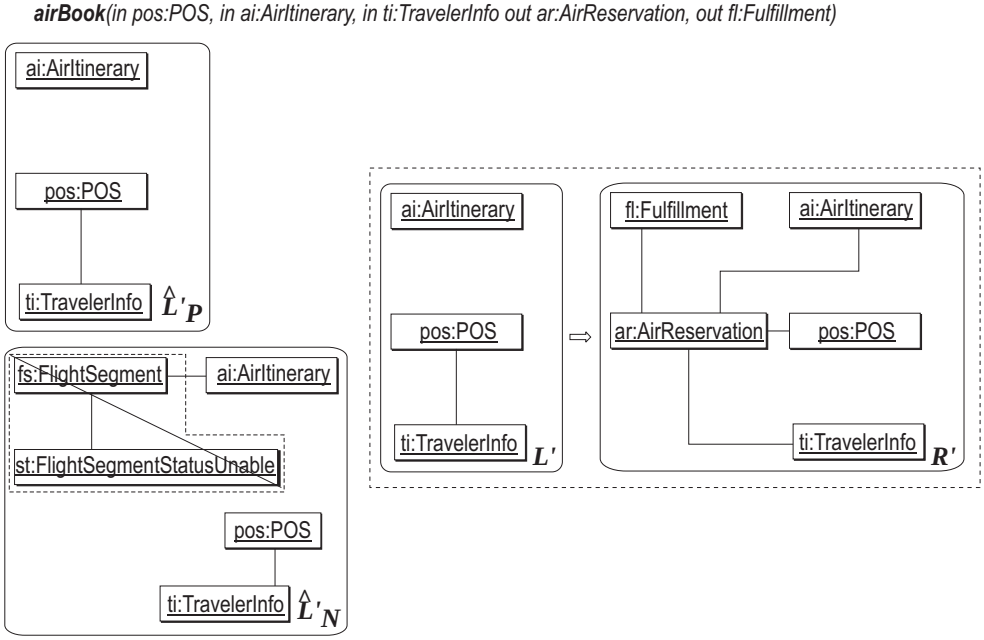


Figure 2.13: Contract rules for the operations airReserv (top) and airBook (bottom).

A *conditional graph transformation system* $\mathcal{C} = \langle TG, P, \pi \rangle$ is similar to the unconditional one, but the mapping π assigns identifiers to conditional TG-typed rule spans s if $A(s)$.

Example 2.2.9. Fig. 2.13 shows conditional rules defining contracts of the required operation airReserv and the provided operation airBook. The parts of Fig. 2.13 marked by the dashed rectangles portray the effects expected by the requestor and guaranteed by the provider, accordingly. Output parameters of the operations appear in the right-hand sides of the rules. $\hat{L}_P$ and $\hat{L}'_P$ specify positive precondition patterns containing the input parameters of the operations.

Apart from travel agency, air itinerary and passenger data that is present in the positive precondition patterns of both rules, $\hat{L}_P$ additionally contains a ticketing information. The requestor considers the parameter tkt:Ticketing as the context element which may be expected by the service provider as the input data. At the same time, this parameter is not needed for the following computations in the requestor system, therefore it appears only in the positive precondition pattern and its further behavior is left unspecified. It can be unrestrictedly manipulated by the provider. The negative precondition patterns $\hat{L}_N$ and $\hat{L}'_N$ of the rules prevent us from booking a flight which has no free places.

Fig. 2.14 shows a transition via the conditional rule airReserv, where the point of sale 07G4325 requests to book the air itinerary ai02 for the passenger(s) id8P. It is not difficult to see that the chosen match of L in the graph G satisfies the positive and negative application constraints. The graph G also contains the alternative itinerary ai01 booking of which is not allowed by the negative application condition, because all places in the flight segment 185AD of this itinerary have been already taken.

Notice that in contrast to the transition in Fig. 2.12, the unspecified effect of the considered transition includes deletion of the vertex t02:Ticketing and the edge connecting it with the vertex ai02:AirItinerary. This is possible, because the deleted elements of the rule in Fig. 2.14 are underspecified. △

2.2.3 Integration of Structural and Behavioral Compartments

Structural and behavioral compartments of a service interface specification, i.e. service signature $\mathcal{S} = \langle TG, P \rangle$ and conditional graph transformation system $\mathcal{C} = \langle TG, P, \pi \rangle$, are developed independently from each other so far. In the following, we relate them in order to get an integral description of a service interface.

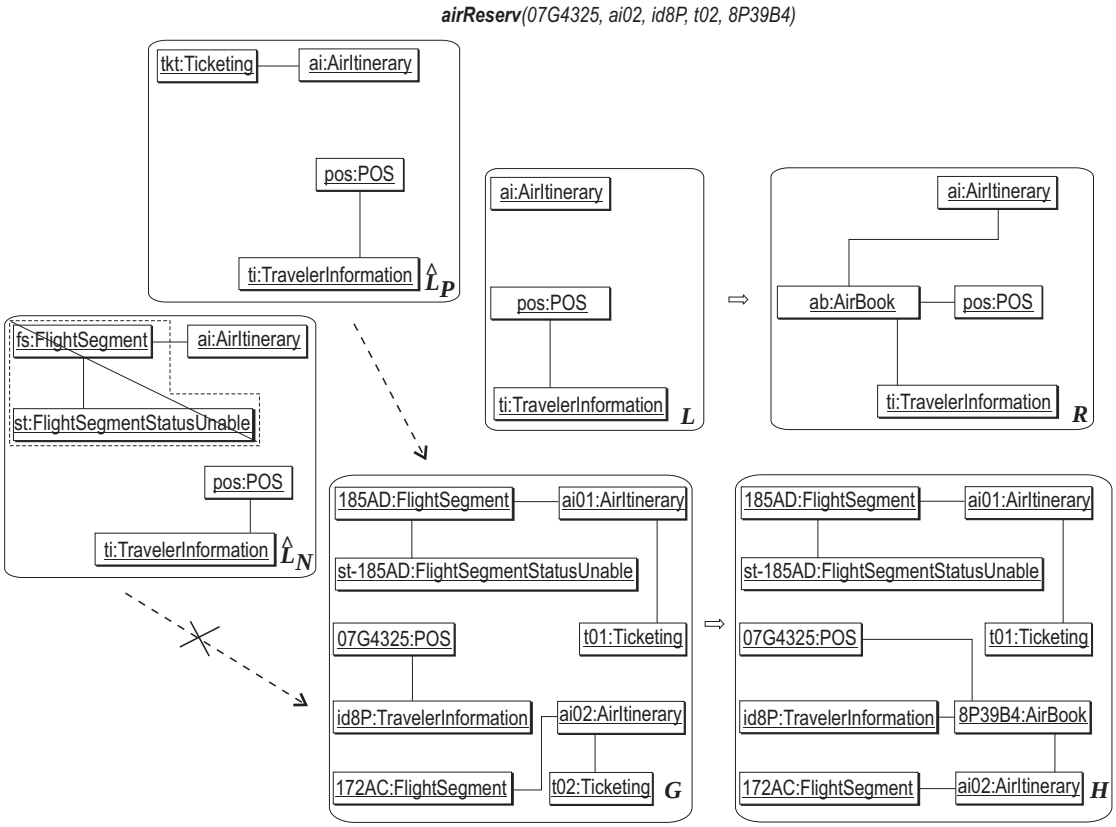


Figure 2.14: A sample graph transition via the conditional rule airReserv.

First of all, we discuss how to combine operation declarations and conditional graph transformation rules describing behavior of these operations. An operation declaration specifies types of input and output parameters appearing as elements in the corresponding contract rule. The positive precondition pattern $\hat{L}_P$ and the right-hand side R of the rule span contain these elements. The desired integration is captured by the notion of *parameterized conditional rules* names of which are given by *parameter expressions* of the form $p(x, y)$, where $p \in P_{v,w}$ is an operation declaration, and $x = x_1, \ldots, x_n$ and $y = y_1, \ldots, y_m$ are sequences of input and output parameters conforming to the declared types.

A *parameterized conditional rule* over a signature $\langle TG, P \rangle$ is a construction $p(x, y) : s \text{ if } A(s)$ composed by a parameter expression $p(x, y)$ and a conditional TG-typed rule span $s \text{ if } A(s)$, such that for $p \in P_{v,w}$, $s = L \to R$, and $A(s) = \{\hat{L}_P, \hat{L}_N\}$ the parameter sequences x and y contains the elements appearing in $\hat{L}_P$ and R. In general, a transformation step or transition via a parameterized conditional rule is analogous to the one via the ordinary rule, however it can be additionally annotated with the elements from the source and target graphs matching the declared parameters (cf., e.g., Fig. 2.11).

A *parameterized conditional graph transformation system* $\mathcal{CP} = \langle TG, P, CP \rangle$ is composed of a signature formed by TG and P, and a set CP which contains parameterized conditional rules established over this signature. Analogously to the unconditional and conditional graph transformation systems, we assume that each rule in CP has a unique name p.

To summarize, a service interface specification is modeled by a parameterized conditional graph transformation system $\mathcal{CP} = \langle TG, P, CP \rangle$, where a data model, operation declarations and contracts are given by a type graph TG, a set P, and a set CP with contract rules, respectively.

2.3 Matching of Service Interface Specifications

Next, we discuss compatibility of required and provided interface specifications containing client's requirements for a useful service and service descriptions. The notion of compatibility is motivated by a *substitution principle*: replacement of abstract operation descriptions in the required system by concrete operations implemented in the provided system should guarantee that the behavior of the compound system is acceptable for the parties.

The structural and behavioral aspects of compatibility are considered in Subsections 2.3.1 and 2.3.2. They are aggregated in Subsection 2.3.3 that describes a matching procedure between required and provided interface specifications. Here the requisite conformity between the parties is determined by an integral compatibility relation over parameterized conditional graph transformation systems portraying the services.

2.3.1 Structural Compatibility

Structural compatibility of required and provided interface specifications is revealed via matching of service signatures. Correspondence of service signatures is modeled by a *structural compatibility relation* consisting of a renaming and an extension.

In general, signatures of required and provided services can be developed in different name spaces. The parties may use different types and operation names for the same concepts. Thus, one of the signatures, e.g., the required one, has to be renamed in order to be comparable with another one. For this purpose we introduce a renaming relation $\mathcal{S} \overset{ren}{\longleftrightarrow} \mathcal{S}'$, which determines a one-to-one correspondence[2] between types and operation names of the original and renamed signatures.

Renaming of types

To find out an appropriate renaming for the types in the required signature, it is necessary to match its type graph with the one of the provided signature so that the associated types correspond semantically to each other. As already mentioned in Subsection 2.1.4, industry standards, such as the OTA specification, facilitate the (automatic) development of the desired renaming.

A fragment of the renaming relation for the example scenario is presented in Table 2.1.

Renaming of operation names

Operation names in the required signature have to be renamed in such a way that the corresponding operation names in the provided signature should have compatible declaration parts. Structural compatibility between two operation declarations developed in the common type context is motivated by the following semantic requirements. On one hand, the provider needs all declared inputs in order to execute

[2]See Remark 2.3.4 at the end of Subsection 2.3.3.

Original types	*Renamed types*
AirBook	AirReservation
AirItineraryPricingInfo	PriceInfo
OriginDestintaionInfo	OriginDestinationInformation
TravelerInformation	TravelerInfo
...	...

Table 2.1: Renaming of the required types.

its operation. On the other hand, the requestor is interested in the provided operation only if it returns all expected outputs.

Syntactic criteria ensuring these semantic requirements are based on matching of the input and output type sequences composing operation declarations. We say that a sequence $\alpha = \alpha_1, \ldots, \alpha_n$ is a *subsequence* of a sequence $\beta = \beta_1, \ldots, \beta_m$, denoted as $\alpha \prec \beta$, if there exist integers $i_1 < i_2 < \ldots < i_n$ such that $\alpha_j = \beta_{i_j}$ for all α_j. Squences are *equal*, denoted as $\alpha = \beta$, if they contain the same elements at the same positions.

Different notions of structural compatibility of the required $p : v \to w$ and provided $p' : v' \to w'$ operation declarations are demonstrated in the table below.

Compatibility	*Inputs*	*Outputs*
Exact	$v = v'$	$w = w'$
Input-preserving	$v = v'$	$w \prec w'$
Output-preserving	$v \succ v'$	$w = w'$
Generalized	$v \succ v'$	$w \prec w'$

Table 2.2: Structural compatibility of operation declarations.

Exact compatibility requires that operations have the same inputs and outputs. Next two variants of compatibility relax dependencies between sequences of input and output parameter types, respectively. In the input-preserving case, a pair of operation declarations is required to have identical inputs, while the provided operation is allowed to return outputs beyond those declared for the required operation. The dual requirements underly the output-preserving compatibility. The generalized compatibility combining relaxations appearing in the input- and output-preserving variants provides the most general requirements. The required operation declaration has to

contain at least the input parameters indicated in the provided declaration, and the provided operation has to return at least the output parameters appearing in the required operation declaration. An appropriate notion of structural compatibility has to be chosen based on demands of the business domain and the platform on which requestor and provider systems are implemented.

Example 2.3.1. Let us check structural compatibility of the operations specified in Fig. 2.7 and Fig. 2.9. The renaming of the type OriginDestintaionInfo in the required operation airAvail to the provided type OriginDestintaionInformation makes the inputs and outputs of this operation identical to those of the provided operation airAvail. This implies exact compatibility of the operation declarations. Since the name of the required operation coincides with the provided one, it should not be changed in the renamed signature.

The declarations of the operations airReserv and airBook differ in several ways. While the former operation has the extra input Ticketing, the latter operation contains the extra output Fulfillment. However, the inputs offered by the requestor are sufficient for the provider, and the outputs proposed by the provider satisfy the requestor's expectations. The described deviations between the declarations are permitted under the generalized notion of structural compatibility. If this kind of compatibility is admissible for the parties, then the required operation is renamed to airBook.

An example of structural incompatibility is given by the declarations of the operations airFareDisplay and airPrice. The differences between appropriately renamed inputs and outputs of the operations violate the syntactic criteria shown in Table 2.2. The provided operation airFareDisplay assumes to obtain OriginDestinationInformation as input, but this parameter type does not appear in the required operation declaration. Moreover, the only output of the required operation airPrice is AirItineraryPricingInfo. However, this output does not belong to the outputs of the provided operation airFareDisplay. △

Extension of service signatures

A provided service usually has more capabilities than a requestor is asking for. Therefore, a provided signature may contain more types and operations than a required one. This fact is captured by the notion of a *signature extension*.

A signature $\mathcal{S}_2 = \langle TG_2, P_2 \rangle$ extends a signature $\mathcal{S}_1 = \langle TG_1, P_1 \rangle$, written $\mathcal{S}_1 \subseteq \mathcal{S}_2$, if the type graph and the set of operation declarations of $\mathcal{S}_1$ are extended in $\mathcal{S}_2$, i.e.

$TG_1 \subseteq TG_2$ and $P_1 \subseteq P_2$. If the chosen notion of structural compatibility varies from the exact one, then the identical operation names in P_1 and P_2 may have different declaration parts. Therefore, the relation between P_1 and P_2 associates, in fact, operation names rather then entire declarations.

Having defined the renaming and the extension, we are able to formulate a relation between the required $\mathcal{S}_1$ and provided $\mathcal{S}_2$ signatures. This relation models their structural compatibility. A *structural compatibility relation* $\mu = (\mathcal{S}_1 \overset{ren}{\longleftrightarrow} \mathcal{S}_1^2 \subseteq \mathcal{S}_2)$ from $\mathcal{S}_1$ to $\mathcal{S}_2$ is a renaming of $\mathcal{S}_1$ such that $\mathcal{S}_2$ represents an extension of the renamed signature $\mathcal{S}_1^2$.

2.3.2 Behavioral Compatibility

Behavioral compatibility of interface specifications is examined via matching of conditional graph transformation systems that describe behavior of required and provided services. A *behavioral compatibility relation* consisting again of a renaming and an extension models the requisite conformity between the GTSs. A construction of this relation will be discussed in this subsection.

To compare behavioral specifications designed in different name spaces, the required GTS is translated into the name space of the provider using a renaming relation. A renaming relation $\mathcal{C} \overset{ren}{\longleftrightarrow} \mathcal{C}'$ defines a one-to-one correspondence between the types, the rule names, and the (vertices and edges of the) rules of the GTSs $\mathcal{C}$ and $\mathcal{C}'$.

Type renaming is based on mapping between the required and provided type graphs. This mapping induces a retyping procedure which allows to compare the rules sharing the same type context. We continue our discussion under the assumption that the systems use the same types consistently.

Renaming of conditional (contract) rules

Matching of the required and provided contract rules underlies an appropriate renaming in the requestor GTS. To find out syntactic criteria for the pair of rules guaranteeing behavioral compatibility of the contracted operations, we have to determine semantic requirements for compatibility between required and provided contracts in general. These semantic requirements depend on how the parties interact with each other.

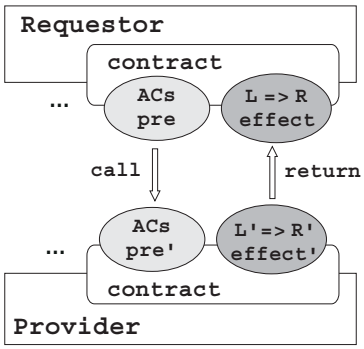


Figure 2.15: Compatibility between required and provided operation contracts.

Fig. 2.15 shows an invocation of the provided operation by the requestor and provider's reply, together with the relevant preconditions and effects constituting the contracts. The interaction consists of the following steps:

1. requestor is willing to submit input data specified in pre to issue a call for the provided operation;

2. provider assumes that the submitted input satisfies the requirements on the invocation described in pre' and calls its operation;

3. provider executes its operation and guarantees the effect described in $effect'$;

4. requestor assumes that the provided effect fulfills assumptions specified in $effect$ and obtains the result of the operation call.

To make sure that the service implemented by the requestor system with the help of the provider works as expected, we have to verify that the assumptions given in 2 and 4 indeed hold. This would be the case if (1) pre implies pre', and (2) $effect'$ implies $effect$. Intuitively, the two semantic requirements can be translated in terms of graph transformation rules as follows.

The precondition part of a rule restricts its applicability. So, the first implication is guaranteed if the applicability of the required rule is preserved or *extended* in the provided rule. The effect part of the rule describes system state modifications induced by a transformation step or transition via this rule. Then the second implication is ensured if the effect of the required rule is preserved or *extended* in the provided rule. Thus, the required contract rule is behaviorally compatible to the provided contract rule, if the latter appropriately extends the former.

Extension of conditional (contract) rules

The extension of a rule p by another rule p' is modeled by a *subrule relation*. A rule $p : s \text{ if } A(s)$ with $s = L \to R$ and $A(s) = \{\hat{L}_P, \hat{L}_N\}$ is a *subrule* of $p' : s' \text{ if } A(s')$ with $s' = L' \to R'$ and $A(s') = \{\hat{L}'_P, \hat{L}'_N\}$, written $p \subseteq p'$, if $\hat{L}'_P \subseteq \hat{L}_P$ and $\hat{L}_N \subseteq \hat{L}'_N$ (applicability is extended), and $L \subseteq L', R \subseteq R'$ (left- and right-hand sides are extended), $del(p) \subseteq del(p')$ (more is deleted by p'), and $add(p) \subseteq add(p')$ (more is added by p').

The subrule relation assumes that the associated rules employ the same names for the shared elements in the spans and application conditions, but it is not the case with the contract rules produced in different name spaces. That means, if we are able to find a renaming of the elements appearing in the required rule p which turns p into the subrule of the provided rule p', then the renamed version of the required rule will be annotated with the name of the matched provided rule.

Example 2.3.2. Let us check behavioral compatibility of the operations airReserv and airBook which are specified by the rules depicted in Fig. 2.13. Here we assume that the objects ti:TravelerInformation and ab:AirBook in the rule airReserv are renamed to the objects ti:TravelerInfo and ar:AirReservation, respectively.

In order to check whether the required operation meets invocation requirements of the provider, we compare the precondition parts of the rules. The invocation requirements to the callers of the provided operation are stated in the positive and negative precondition patterns $\hat{L}'_P$ and $\hat{L}'_N$. While the former is enriched in the requestor rule by the object tkt:Ticketing and the edge between this object and the object ai:AirItinerary, the latter is identical (modulo retyping) to the negative precondition pattern $\hat{L}_N$. This means applicability of the required rule is extended in the provided rule.

Then, we match the effect parts of the rules, because the benefit obtained by applying the provided operation should satisfy the expectations of the client. The object fl:Fulfillment and the edge from this object to the object ar:AirReservation are created in the system state after the execution of the operation airBook. These elements are not present in the right-hand side of the requestor rule airReserv, because it is sufficient to obtain only a confirmation in the form of a reservation tag. Nevertheless, the provided effect extending the required one fits the client requirements.

Since the renamed version of the required rule turns to be a subrule of the provider rule, its name has to be changed to airBook. Furthermore, the subrule relation between

the contract rules guarantees behavioral compatibility of the contracted operations. △

Extension of conditional GTSs

It remains to determine an extension for a pair of behavioral specifications developed in a common name space. A conditional graph transformation system $\mathcal{C}'$ *extends* another one $\mathcal{C}$, written $\mathcal{C} \subseteq \mathcal{C}'$, if the type graph and rule names of $\mathcal{C}$ are extended, i.e. $TG \subseteq TG'$ and $P \subseteq P'$, and for each rule name $p \in P$ the associated rule in $\mathcal{C}$ is a subrule of the corresponding rule in $\mathcal{C}'$.

Now, we aggregate the developed concepts in a relation, which ensures behavioral compatibility of conditional graph transformation systems specifying required and provided services. A *behavioral compatibility relation* $\nu = (\mathcal{C}_1 \overset{ren}{\longleftrightarrow} \mathcal{C}_1^2 \subseteq \mathcal{C}_2)$ from $\mathcal{C}_1$ to $\mathcal{C}_2$ is a renaming of $\mathcal{C}_1$ such that $\mathcal{C}_2$ represents an extension of the renamed system $\mathcal{C}_1^2$.

2.3.3 Integral Compatibility

In the previous subsections, we have established the relations which model structural and behavioral kinds of compatibility for interface specifications of required and provided services. Structural compatibility guarantees the appropriate relationship between inputs and outputs of the associated operations, which may have, however, completely different missions. Behavioral compatibility ensures semantic resemblance of the operations, which may be structurally inconsistent.

At this point, we discuss integral compatibility combining structural as well as behavioral aspects. It is modeled by an *integral compatibility relation*. This relation consisting of a renaming and an extension associates required and provided service interfaces specifications in the form of parameterized conditional graph transformation systems.

A renaming relation $\mathcal{CP} \overset{ren}{\longleftrightarrow} \mathcal{CP}'$ between two parameterized conditional GTSs is defined analogously to the one in behavioral compatibility relation. It links the types, the rule names, and the elements of the system rules.

Since a type renaming is still the same as in the previous subsection, for the rest of the presentation we assume that the systems are defined in the same type context.

Renaming of rule (operation) names

Rule names in parameterized conditional GTSs are equipped with parameter type declarations. A rule name in the provided system becomes a renaming candidate for the required rule name, if the parameter type declarations of the two are structurally compatible. Then, we proceed with the behavioral check, where the contract rules associated with the rule names containing structurally compatible declarations are examined for behavioral compatibility.

Behavioral compatibility is captured by the notion of a rule extension that is already discussed in the previous subsection and modeled via a subrule relation. Now the subrule relation has to be redefined for conditional rules with parameters.

Extension of parameterized conditional (contract) rules

A parameterized conditional rule $p(x, y) : s \text{ if } A(s)$ with $p : v \to w$ is a *subrule* of $p'(x', w') : s' \text{ if } A(s')$ with $p' : v' \to w'$, written $p(x, y) \subseteq p'(x', y')$, if a conditional rule $p : s \text{ if } A(s)$ is a subrule of $p' : s' \text{ if } A(s')$, and $x = x'$ and $y = y'$.

The formulated notion of a subrule relation assumes the exact version of structural compatibility. If, for example, the generalized version of structural compatibility is chosen, then the equality between the parameter sequences has to be replaced for the subsequence relation $x \succ x'$ and $y \prec y'$.

Thus, the replacement of the required rule name for the provided one is allowed to be performed, if the parameter type declarations of the two are structurally compatible, and there exists renaming for the rule associated with the required name which transforms this rule into the subrule of the one associated with the provided name.

Example 2.3.3. Let us examine integral compatibility of the operations airReserv and airBook which are specified by parameterized conditional rules in Fig. 2.13. First of all, one should check structural compatibility between the parameter type declarations augmenting the rule names. We have shown in Example 2.3.1 that the declaration of the operation airReserv renamed according to Table 2.2 and the declaration of the operation airBook satisfy the requirements imposed by the generalized notion of structural compatibility.

Then, we check behavioral compatibility which amounts to matching of the conditional rules associated with the rule names airReserv and airBook. According to Example 2.3.2, renaming of the objects ti:TravelerInformation and ab:AirBook in the

rule airReserv into the objects ti:TravelerInfo and ar:AirReservation makes this rule a subrule of the rule airBook. This guarantees behavioral compatibility between the considered operations.

Combining the outputs of structural and behavioral compatibility tests, we can conclude that the provided rule indeed extends the required rule. Therefore, the renamed version of the latter has to be equipped with the identifier airBook. Moreover, the established subrule relation guarantees integral compatibility between the operations airReserv and airBook. △

Extension of parameterized conditional GTSs

Before we turn to definition of an integral compatibility relation, we have to do one more thing—to define the notion of an extension for parameterized conditional GTSs. A parameterized conditional GTS $\mathcal{CP}'$ *extends* another one $\mathcal{CP}$, written $\mathcal{CP} \subseteq \mathcal{CP}'$, if a signature $\mathcal{S}' = \langle TG', P' \rangle$ of $\mathcal{CP}'$ extends a signature $\mathcal{S} = \langle TG, P \rangle$ of $\mathcal{CP}$, and for each rule name $p \in P$ the associated rule in $\mathcal{CP}$ is a subrule of the corresponding rule in $\mathcal{CP}'$.

Finally, we define the sought-for integral compatibility relation, which ensures structural and behavioral compatibility between parameterized conditional GTSs that entirely describe required and provided services. An *integral compatibility relation* $\varsigma = (\mathcal{CP}_1 \overset{ren}{\longleftrightarrow} \mathcal{CP}_1^2 \subseteq \mathcal{CP}_2)$ from $\mathcal{CP}_1$ to $\mathcal{CP}_2$ is a renaming of $\mathcal{CP}_1$ such that $\mathcal{CP}_2$ represents an extension of the renamed system $\mathcal{CP}_1^2$.

Note that the integral compatibility relation is constructed on top of informally given semantic requirements underlying compatibility of interface specifications of coupled systems. In particular, the behavioral compatibility relation is based merely on intuition obtained from the example scenario. This does not allow to check whether the introduced syntactic procedure adequately reflects the semantic requirements. Therefore, in the next chapter, we provide a formalization of the introduced concepts based on the existing theory of the algebraic approach to graph transformation [51, 45, 36]. This allows to rigorously formulate semantic requirements and to carry out the required justifications.

Remark 2.3.4. In the next chapter, integral compatibility relations are formally described as morphisms between parameterized conditional graph transformation systems. Employing the style of presentation proposed in [76], we presented such a

morphism as decomposed into an isomorphism and an inclusion (called a renaming and an extension, respectively). All morphisms expressible in this way are injective. Based on the definition of morphisms of parameterized conditional GTSs given in Subsection 3.2.4, however, we could also represent compatibility relations where different elements in the required interface specification are identified in the provided one. △

2.4 Summary

In this chapter, we have introduced the notion of service interface specification embracing structural and behavioral compartments. While the structural compartment is described by the service signature, the conditional GTS is employed for the definition of the behavioral compartment containing operation contracts. The graph transformation rules contracting service operations are equipped with loose semantics in the required interface specification and with strict semantics in the provided interface specification. An aggregation of structural and behavioral compartments leads to an integral service specification in the form of parametrized conditional GTS.

We have also explored compatibility of required and provided interface specifications. The structural and behavioral aspects of compatibility are checked by matching of service signatures and conditional GTSs, accordingly. The intended conformity between the corresponding compartments is specified by the structural and behavioral compatibility relations. These aspect-specific relations are absorbed in the integral compatibility relation which models the required correspondence between integral service interface specifications.

In our approach, matching of service specifications is assisted by an industry standard such as the one issued by the OTA consortium for the travelling business domain. We illustrated the development of standard-based service interface specifications and their comparison by an example that shows the OTA-compliant specifications of the required and provided Web services for booking flight tickets.

It is necessary to mention that structural and behavioral characteristics of services are not the only features able to facilitate the discovery process. The interface specification can be extended, e.g., with description of a required or provided business process [148].

Generally speaking, a business process specifies the execution order of operations that lead to a specific business outcome. A business process in the requestor specifica-

tion represents a fragment of the global business process underlying the functioning of the requestor system. The operations sought by requestor have to be mapped into this fragment and adequately blended with the existing context. A business process in the provider specification defines how to use the service and prevents sequences of operation calls leading to the improper interaction between the parties. Augmentation of a service description with business process data (along with other business-specific, industry-specific, or organization-specific requirements) yet remains an open issue. A detailed discussion of primary and auxiliary characteristics used to specify services can be found in [112].

Chapter 3

Parameterized Conditional Graph Transformation

In this chapter we reuse and extend the existing results of the algebraic double-pushout (DPO) and double-pullback (DPB) approaches to graph transformation to formalize concepts and ideas presented in Chapter 2. Section 3.1 is devoted to the rigorous description of service interface specifications by means of parameterized conditional graph transformation systems (GTSs). Section 3.2 formalizes the structural and behavioral compatibility relations and verifies correctness of the latter against formally given semantic requirements. An aggregation of the two relations allows to construct a parameterized substitution morphism that determines the intended correspondence between parameterized conditional GTSs. Finally, Section 3.3 contains a summary of formal stratum for our approach.

3.1 A Formal Account of Service Interface Specifications

As already mentioned, automation of matching procedure requires a formal counterpart for each constituent of a service interface specification. The following section addresses this problem. A formalization of the structural compartment via service signature is considered in the next subsection. Subsection 3.1.2 starts with an overview of basic notions of the DPO and DPB approaches to graph transformation. It also demonstrates conditional GTSs that describe services behaviorally. In Subsection 3.1.3, service signatures and conditional GTSs are combined in order to obtain an integral service specification given by parameterized conditional GTSs.

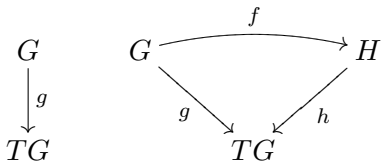


Figure 3.1: Typed graph and graph morphism.

3.1.1 Service Signature

To develop structural and behavioral parts of a service interface specification, we start from a data model. As proposed in Subsection 2.2.1, this data model can be represented by a type graph. First of all, let us introduce a class of graphs that we consider in further discussion.

Definition 3.1.1 (graphs, graph morphisms). A *directed unlabeled graph* is a tuple $G = \langle G_V, G_E, src^G, tar^G \rangle$, where G_V is a set of vertices (or nodes), G_E is a set of edges (or arcs), $src^G, tar^G : G_E \to G_V$ are two functions associating with each edge its source and target vertex.

A *graph morphism* $f : G \to H$ is a pair of functions $\langle f_V : G_V \to H_V, f_E : G_E \to H_E \rangle$ preserving source and target, that is, $src^H \circ f_E = f_V \circ src^G$ and $tar^H \circ f_E = f_V \circ tar^G$. With componentwise identities and composition this defines the category **Graph**. △

In Chapter 2, *typed graphs* have been used to conceptualize the relation between a schema and its instances. Formally, given a graph $TG \in |\mathbf{Graph}|$, the category $\mathbf{Graph}_{TG}$ of *TG-typed graphs* and *TG-typed graph morphisms* [35] is the comma category $(\mathbf{Graph} \downarrow TG)$. That is, its objects graph morphisms $g : G \to TG$ into TG and arrows from $g : G \to TG$ to $h : H \to TG$ graph are morphisms $f : G \to H$ such that $h \circ f = g$ (cf. Fig. 3.1). In Chapter 2, TG is called a *type graph*, and the graphs G and H are called *instance graphs*.

A type graph together with operation declarations represent a structural description of a service defined below.

Definition 3.1.2 (service signature). A *service signature* $\mathcal{S}$ is a pair $\langle TG, P \rangle$ consisting of a type graph TG and a family of sets $P = (P_{v,w})_{v,w \in |TG|^*}$ with operation declarations of the form $p : v \to w$ for $p \in P_{v,w}$. △

We proceed with a formal counterpart of the behavioral compartment of a service interface specification.

Figure 3.2: Typed span morphism (left) and double-pushout (or -pullback) diagram (right).

3.1.2 Conditional Graph Transformation Systems

Hereinafter, we recall main concepts of the *double-pushout* (DPO) [36] and *double-pullback* (DPB) [81] approaches to graph transformation providing a formal setting for specification of the behavioral compartment. Both approaches are presented using typed graphs [35].

Graph productions (or rules) are specified by spans of injective graph morphisms $L \xleftarrow{l} K \xrightarrow{r} R$. The *left-hand side* L contains the items that must be present before an application of production, the *right-hand side* R those that are present afterwards, and the *context graph* K specifies the "gluing items", i.e. the elements which are read during the application, but are not consumed.

Definition 3.1.3 (typed span, graph transformation system). A *TG-typed span* s is an expression of the form $(L \xleftarrow{l} K \xrightarrow{r} R)$, where l and r are injective TG-typed graph morphisms.

Given TG-typed spans $s = (L \xleftarrow{l} K \xrightarrow{r} R)$ and $s' = (L' \xleftarrow{l'} K' \xrightarrow{r'} R')$, a *$TG$-typed span morphism* $e : s \to s'$ is a tuple $\langle e_L, e_K, e_R \rangle$ of TG-typed graph morphisms commuting with l, l', r and r' (cf. Fig. 3.2 on the left). With componentwise identities and composition this defines the category $\mathbf{Sp}_{TG}$. If the two commutative squares are pushouts or pullbacks, the corresponding categories are denoted by $\mathbf{Sp}_{TG}^{DPO}$ and $\mathbf{Sp}_{TG}^{DPB}$, respectively.

A (typed) *graph transformation system* $\mathcal{G} = \langle TG, P, \pi \rangle$ consists of a *type graph* TG, a set of production names P, and a mapping π associating with each production name p a TG-typed span $\pi(p)$. If $p \in P$ is a production name and $\pi(p) = s$, we say that $p : s$ is a production of $\mathcal{G}$. △

DPO Graph Transformations

In the DPO approach, transformation of graphs is defined by a pair of pushout diagrams, a so-called double-pushout construction. A *double-pushout (DPO) diagram d*

is a diagram as in Fig. 3.2 on the right, where (1) and (2) are pushouts. Operationally speaking the application of the production to the graph G consists of two steps: the elements of G matched by $L \setminus l(K)$ are removed, and a copy of $R \setminus r(K)$ is added to D.

Gluing the graphs L and D over their common part K yields again the graph G, i.e. the left-hand square (1) forms a *pushout complement*. Only in this case the application is permitted. Similarly, the derived graph H is the gluing of D and R over K, which creates the right-hand side pushout square (2). The resulting *double-pushout (DPO) diagram* represents the transformation of G into H.

Definition 3.1.4 (DPO graph transformation). Given a graph transformation system $\mathcal{G} = \langle TG, P, \pi \rangle$ and a production $p : (L \xleftarrow{l} K \xrightarrow{r} R)$, a *(DPO) transformation step* in $\mathcal{G}$ from G to H via p, denoted by $G \overset{p/d}{\Longrightarrow} H$, is a diagram like in the right of Fig. 3.2, where both (1) and (2) are pushout squares. We also write p/d if G and H are understood, and denote by $top(d)$ and $bot(d)$ the top and bottom span of d.

A *transformation sequence* $\rho = \rho_1 \ldots \rho_n : G \Rightarrow^* H$ in $\mathcal{G}$ via $p_1, \ldots, p_n$ is a sequence of transformation steps $\rho_i = (G_i \overset{p_i/d_i}{\Longrightarrow} H_i)$ such that $G_1 = G, H_n = H$ and consecutive steps are composable, that is $G_{i+1} = H_i$ for all $1 \leq i < n$.

The *category of transformation sequences over* $\mathcal{G}$ denoted by $\mathbf{Trf}(\mathcal{G})$ has all graphs $G \in \mathbf{Graph}_{TG}$ as objects and all transformation sequences in $\mathcal{G}$ as arrows. △

The existence of the pushout complement (1), and hence of a direct derivation[1] $G \overset{p/d}{\Longrightarrow} H$, is characterized by the *gluing conditions* [45]. The *dangling condition* ensures that the structure D obtained by removing from G all objects to be deleted is indeed a graph, that is, no edges are left "dangling" without source or target node. The *identification condition* states that objects from the left-hand side may only be identified by the match if they also belong to the context graph K (and are thus preserved).

As already mentioned in the previous chapter, the DPO approach to graph transformation obeys the implicit frame condition ensuring that the changes to the given graph G are exactly those specified by the production. However operation contracts may represent incomplete specifications of operations, as it happens, e.g., with contracts of required operations.

[1] The pushout (2) always exists since the category $\mathbf{Graph}_{TG}$ is cocomplete.

DPB Graph Transitions

A more liberal notion of production application is provided by the double-pullback (DPB) approach to graph transformation [81]. The DPB approach introduces *graph transitions* and generalizes DPO by allowing additional, unspecified changes. Formally, graph transitions are defined by replacing the double-pushout diagram of a transformation step with a double-*pullback*.

Definition 3.1.5 (DPB graph transitions). Given a graph transformation system $\mathcal{G} = \langle TG, P, \pi \rangle$ and a production $p : (L \xleftarrow{l} K \xrightarrow{r} R)$, a *transition* in $\mathcal{G}$ from G to H via p, denoted by $G \overset{p/d}{\rightsquigarrow} H$, is a diagram like in the right of Fig. 3.2, where both (1) and (2) are pullback squares.

A transition is called *injective* if both g and h are injective graph morphisms. It is called *faithful* if it is injective, and the morphisms d_L and d_R satisfy the following condition: for all $x, y \in L$, $y \notin l(K)$ implies $d_L(x) \neq d_L(y)$, and analogously for d_R.

A *transition sequence* $\rho = \rho_1 \dots \rho_n : G \rightsquigarrow^* H$ in $\mathcal{G}$ via $p_1, \dots, p_n$ is a sequence of faithful transitions $\rho_i = G_i \overset{p_i/d_i}{\rightsquigarrow} H_i$ such that $G_1 = G, H_n = H$ and consecutive steps are composable, that is, $G_{i+1} = H_i$ for all $1 \leq i < n$.

The *category of transitions over* $\mathcal{G}$, denoted by $\mathbf{Trs}(\mathcal{G})$, has all graphs $G \in \mathbf{Graph}_{TG}$ as objects and all transition sequences in $\mathcal{G}$ as arrows. △

The condition ensuring the faithfulness of transitions means that d_L and d_R satisfy the identification condition of the DPO approach with respect to l and r. Notice that any pushout square of two given morphisms such that one of them is injective is also a pullback square. Thus, every DPO transformation is also a DPB transition. Each faithful transition, in turn, can be regarded as a transformation step plus a change-of-context [81]. This is modeled by additional deletion and creation of elements before and after the actual step. In the following we stick in our presentation to the faithful transitions.

Having defined the appropriate semantic interpretations of the productions we proceed with the structural refinement allowing clear separation of their precondition and effect parts.

Conditional Graph Transformation

As already discussed, the desired refinement is achieved by extending productions with positive and negative application conditions [72] shown in the following definition.

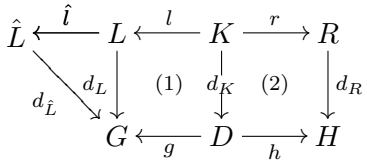


Figure 3.3: Conditional span.

Definition 3.1.6 (conditional span). An *application condition* $A(s) = \langle AP(s), AN(s)\rangle$ over a TG-typed span $s = (L \xleftarrow{l} K \xrightarrow{r} R)$ consists of two sets of typed graph morphisms $AP(s)$ and $AN(s)$ outgoing from L which contain positive and negative constraints, respectively. $A(s)$ is called positive (negative) if $AN(s)$ ($AP(s)$) is empty.

Let $L \xrightarrow{\hat{l}} \hat{L}$ be a positive or negative constraint and $L \xrightarrow{d_L} G$ be a typed graph morphism (cf. Fig. 3.3). Then d_L *P-satisfies* $\hat{l}$, if there exists a typed graph morphism $\hat{L} \xrightarrow{d_{\hat{L}}} G$ such that $d_{\hat{L}} \circ \hat{l} = d_L$. d_L *N-satisfies* $\hat{l}$, if it does not *P-satisfy* $\hat{l}$.

Let $A(s) = \langle AP(s), AN(s)\rangle$ be an application condition and $L \xrightarrow{d_L} G$ be a typed graph morphism. Then d_L *satisfies* $A(s)$, if it *P-satisfies* at least one positive constraint and *N-satisfies* all negative constraints from $A(s)$.

A *conditional span* is an expression of the form s if $A(s)$, where $s = (L \xleftarrow{l} K \xrightarrow{r} R)$ is a TG-typed span, and $A(s)$ is an application condition over s. It is *applicable* to a graph G via $L \xrightarrow{d_L} G$ if d_L satisfies $A(s)$. △

Notice that positive application conditions consist of a disjunction of positive constraints, in contrast with the conjunction in [72]. That means positive and negative conditions are, in fact, dual to each other.

A formal specification of the behavioral compartment in the service interface specification is provided by conditional graph transformation systems.

Definition 3.1.7 (conditional GTS). A *conditional graph transformation system* $\mathcal{C} = \langle TG, P, \pi\rangle$ consists of a type graph TG, a set of rule names P, and a mapping π providing for each rule name p a TG-typed conditional span $\pi(p)$. If $p \in P$ is a production name and $\pi(p) = s$ if $A(s)$, we say that $p : s$ if $A(s)$ is a conditional production of $\mathcal{C}$.

Given a conditional production $p : s$ if $A(s)$ of $\mathcal{C}$, then a *transformation step* (*transition*) in $\mathcal{C}$ from G to H via p is a transformation step (transition) via an unconditional production $p : s$ such that $d_L \in d$ satisfies $A(s)$ (cf. Fig. 3.3) △

For a conditional production $p : s$ if $A(s)$ with $A(s) = \langle AP(s), AN(s)\rangle$ we assume that the injective morphisms $L \xrightarrow{\hat{l}} \hat{L}$ representing positive or negative constraints are indeed inclusions, and that the pairwise intersection of all the $\hat{L}$ is well-defined and equal to L. That means, the left-hand side of a rule and all its constraints are defined over a common name space, a fact that will be relevant when introducing parameterized productions.

We proceed with an integral specification of service interface.

3.1.3 Parameterized Conditional Graph Transformation Systems

The notion of parameterized production plays a central role in the integration of the two specification compartments. First of all we introduce a parametrization of unconditional productions. Here the input and output parameters, which types are determined by the operation declarations, represent elements of the left- and right-hand sides of the production span, respectively.

We denote by $v = v_1 \dots v_n$ sequences of types $v_i \in |TG|$. Corresponding sequences of elements of a TG-typed graph X are written as $x \in X_v$ which is short-hand for $x_1 : v_1 \dots x_n : v_n$ with $x_i : v_i \in X$.

Definition 3.1.8 (parameterized production and GTS). Given a signature $\mathcal{S} = \langle TG, P\rangle$, and a pair of TG-typed graphs $\langle X, Y\rangle$, a set of *parameter expressions over* $\mathcal{S}$ *and* $\langle X, Y\rangle$ is defined by $\mathcal{E}_P(X, Y) = \{p(x, y) | p \in P_{v,w}, x \in X_v, y \in Y_w\}$.

A *parameterized production* pp over $\mathcal{S} = \langle TG, P\rangle$ is an expression of the form $p(x, y) : s$, where $s = (L \xleftarrow{l} K \xrightarrow{r} R)$ is a TG-typed span, and $p(x, y) \in \mathcal{E}_P(L, R)$ is a parameter expression over $\langle L, R\rangle$, i.e. $x \in L_v$ and $y \in R_w$ for $p \in P_{v,w}$.

A *parameterized graph transformation system* $\mathcal{GP} = \langle TG, P, PP\rangle$ is a triple, where TG and P compose a signature, and PP is a set of parameterized productions over this signature, such that $p(x, y) : s_i$ for $i = 1, 2$ implies $s_1 = s_2$. △

The definition states that each production in $\mathcal{GP}$ has unique name, so the parameterized GTS can be represented by the GTS $\mathcal{G} = \langle TG, P, \pi\rangle$, where P contains parameter expressions for each production from PP, and π associates $p(x, y)$ with s if and only if the corresponding production appears in PP. Hence transformations or transitions in $\mathcal{GP}$ are similar to those in the graph transformation system $\mathcal{G}$.

Next, we turn to parametrization of conditional productions. In contrast with the unconditional case, input parameters in conditional productions are represented by the elements of positive precondition patterns $\hat{L}^i_P$.

Definition 3.1.9 (parameterized conditional production and GTS). A *parameterized conditional production cp* over $\mathcal{S} = \langle TG, P \rangle$ is an expression of the form $p(x, y) : s \text{ if } A(s)$, where $s \text{ if } A(s)$ is a TG-typed conditional span, and $p(x, y) \in \mathcal{E}_P(\mathcal{L}, R)$ is a parameter expression over $\langle \mathcal{L}, R \rangle$, i.e. $x \in \mathcal{L}_v$ and $y \in R_w$ for $p \in P_{v,w}$ and $\mathcal{L} = \bigcup \hat{L}^i_P$, assuming that the union $\mathcal{L}$ of all positive precondition patterns is well-defined.

A *parameterized conditional graph transformation system* $\mathcal{CP} = \langle TG, P, CP \rangle$ is a triple, where TG and P compose a signature, and CP is a set of parameterized conditional productions over this signature, such that $p(x, y) : s_i \text{ if } A(s_i)$ for $i = 1, 2$ implies $s_1 \text{ if } A(s_1) = s_2 \text{ if } A(s_2)$. △

The parameterized conditional GTS combines in its structure the formalisms established in the two previous subsections, i.e. service signature $\mathcal{S} = \langle TG, P \rangle$ and conditional GTS $\mathcal{C} = \langle TG, P, \pi \rangle$. It is not difficult to see, that transformations or transitions in $\mathcal{CP}$ are also similar to those in the graph transformation system $\mathcal{C}$.

3.2 A Formal Account of Matching Service Interface Specifications

In this section, we introduce a novel concept of parameterized substitution morphism providing a formal representation of the matching procedure between the required and provided service interface specifications. To do this, one should formalize the structural and behavioral compatibility relations discussed in Subsections 2.3.1 and 2.3.2 and adequately aggregate them.

3.2.1 Signature Morphism

The structural compatibility relation is specified by a signature morphism which directly reflects the semantic requirements on the correspondence between operation declarations (cf. Subsection 2.3.1).

Definition 3.2.1 (signature morphism). A *signature morphism* $f = \langle f_{TG}, f_P \rangle : \langle TG, P \rangle \to \langle TG', P' \rangle$ consists of a type graph morphism $f_{TG} : TG \to TG'$ and a mapping of production names $f_P : P \to P'$ such that for each operation declaration $p : v \to w \in P$ there exists an operation declaration $f_P(p) : v' \to w' \in P'$ with $f^*_{TG}(v) = v'$ and $f^*_{TG}(w) = w'$. △

The definition captures exact compatibility of operation declarations in the signatures. We therefor speak of *exact signature morphisms*. The three remaining variants of structural compatibility can be obtained if the equality between the sequences of input and output parameter types in Def. 3.2.1 are replaced with the subsequent relations according to Table 2.2, leading to the notions of *input-preserving*, *output-preserving*, and *generalized signature morphisms*.

3.2.2 Substitution Morphism

In order to establish a relation ensuring the desired correspondence between behavioral specifications of services, i.e. conditional GTSs, foremost, one should formulate semantic requirements underlying such relation.

Due to the substitution principle declared in the beginning of Section 2.3, abstract productions of the requestor system $\mathcal{C}$ are expected to be replaced for the concrete productions of the provider system $\mathcal{C}'$. First of all, such replacement has to preserve applicability of all substituted productions. Otherwise, the requestor may strike on unexpected limitations at the invocation of provided operations.

Secondly, the behavior portrayed by the provider system should not be weaker then the one expected by the requestor. Each GTS describes a behavior in terms of transformation or transition sequences obtained via application of its productions. So, the second requirement is fulfilled if each transformation step in $\mathcal{C}'$ implies a transition in $\mathcal{C}$. These two requirements are captured by the following definition.

Definition 3.2.2 (substitutability). Given conditional graph transformation systems $\mathcal{C} = \langle TG, P, \pi \rangle$ and $\mathcal{C}' = \langle TG', P', \pi' \rangle$, we say that $\mathcal{C}'$ is substitutable for $\mathcal{C}$ if there exists a functor $F : \mathbf{Trf}(\mathcal{C}') \to \mathbf{Trs}(C)$ such that for all graphs $G' \in |\mathbf{Trf}(\mathcal{C}')|$ and for all transition sequences $\rho : F(G') \to \_ \in \mathbf{Trs}(\mathcal{C})$ there exists a transformation sequence $\rho' : G' \to \_ \in \mathbf{Trf}(\mathcal{C}')$ with $F(\rho') = \rho$. △

Functor F translating states of $\mathcal{C}'$ into states of $\mathcal{C}$ can be realized by the retyping induced by a morphism between two type graphs of the systems [68].

Definition 3.2.3 (retyping). A graph morphism $f_{TG} : TG \to TG'$ induces a *forward retyping functor* $f^{>}_{TG} : \mathbf{Graph}_{TG} \to \mathbf{Graph}_{TG'}$, $f^{>}(g) = f \circ g$ and $f^{>}(k : g \to h) = k$ by composition as shown in Fig. 3.4 on the left, as well as a *backward retyping functor* $f^{<}_{TG} : \mathbf{Graph}_{TG'} \to \mathbf{Graph}_{TG}$, $f^{<}(g') = g^*$ and $f^{<}(k' : g' \to h') = k^* : g^* \to h^*$ by pullbacks and mediating morphisms as shown in Fig. 3.4 on the right. △

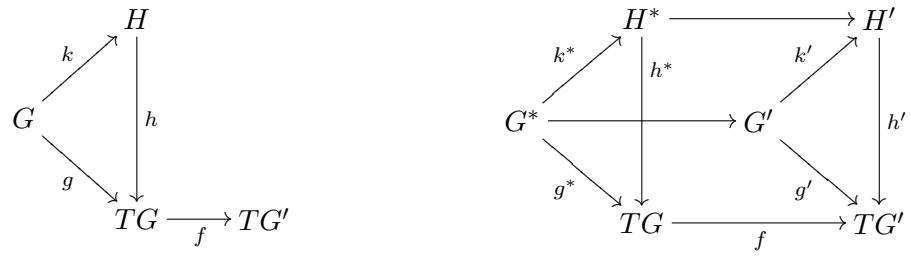


Figure 3.4: Forward (left) and backward (right) retyping functors.

$$\begin{array}{ccccccc} G_0 & \overset{p_1/d_1}{\rightsquigarrow} & H_0 = G_1 & \overset{p_2/d_2}{\rightsquigarrow} & H_1 = G_2 & \cdots \\ \uparrow{\scriptstyle f^{<}_{TG}} & & \uparrow{\scriptstyle f^{<}_{TG}} & & \uparrow{\scriptstyle f^{<}_{TG}} & \\ G'_0 & \overset{p'_1/d'_1}{\Longrightarrow} & H'_0 = G'_1 & \overset{p'_2/d'_2}{\Longrightarrow} & H'_1 = G'_2 & \cdots \end{array}$$

Figure 3.5: Substitution in detail.

Let us check in detail what actually happens when the abstract productions p_i of the requestor system $\mathcal{C}$ are substituted for the concrete productions p'_j of the provider system $\mathcal{C}'$. This assumes that requestor and provider are actual components which communicate at runtime.

The starting point is a graph $G'_0 \in \mathbf{Graph_{TG'}}$, representing the state of the provider component (cf. Fig. 3.5).

The substitution consists of the following steps:

- G'_0 is projected to $G_0 \in \mathbf{Graph_{TG}}$.
- If a production p_1 is applicable to G_0 on the requestor side, the same should hold for the corresponding provider production p'_1.
- A transformation step $G' \overset{p'_1/d'_1}{\Longrightarrow} H'$ is performed by the provider which projects to a transition $G_0 \overset{p_1/d_1}{\rightsquigarrow} H$ via the corresponding production in the requestor view.

Thus, the requestor receives an update to its local view from the state of the provider, and the cycle can start anew.

Example 3.2.4. We illustrate the update process by means of the conditional productions airReserv and airBook introduced in Example 2.2.9. The graph G_0 and G'_0 describing initial states of the provider and requestor components are depicted in Fig 3.6.

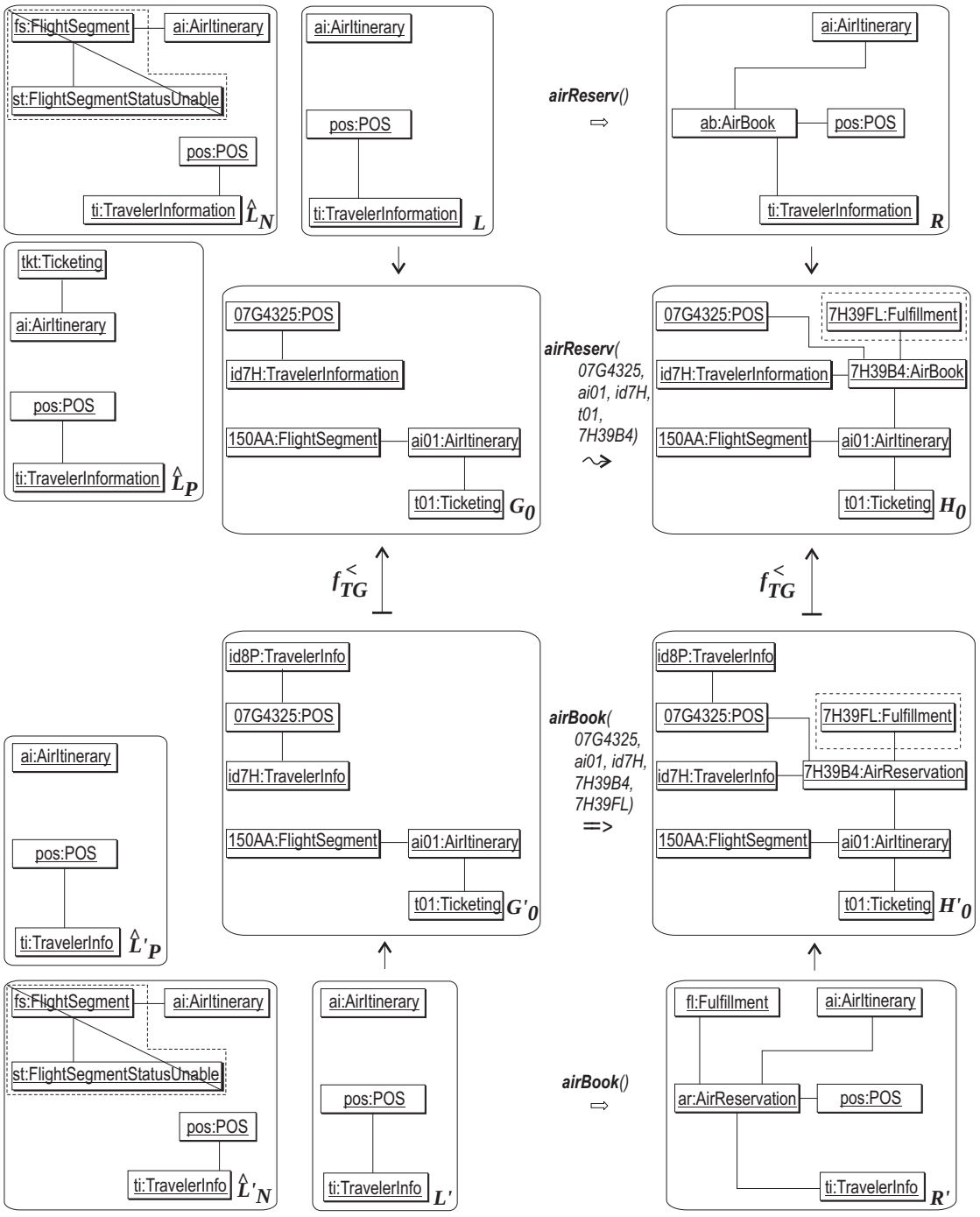


Figure 3.6: Updating the requestor system state.

Projecting the graph G'_0 to the graph G_0, the backward retyping functor $f^{<}_{TG}$ renames the type of the object id7H in G'_0 to the type TravelerInformation in G_0. It is not difficult to see that the production airReserv is applicable to the graph G_0. Since the same holds for the production airBook and the graph G'_0, a transformation step is constructed that leads to the graph H'_0. Then this transformation is projected to the transition via the production airReserv. Here the backward retyping functor $f^{<}_{TG}$ renames the type of the object 7H39B4 in H'_0 to the type AirBook in the graph H_0.

The target graph H_0 of the transition contains the object 7H39FL:Fulfillment and the edge between this object and the object 7H39FL:Fulfillment being an effect which is not encoded in the production airReserv. The creation of the mentioned object and the edge in the updated state H_0 of the requestor component is modeled by the unspecified effect of the transition. △

Now we are ready to construct a formal counterpart of the behavioral compatibility relation.

Definition 3.2.5 (substitution morphism). Given conditional graph transformation systems $\mathcal{C} = \langle TG, P, \pi \rangle$ and $C' = \langle TG', P', \pi' \rangle$, a *substitution morphism* $f^{sub} = \langle f_{TG}, f_P \rangle$ is given by a type graph morphism $f_{TG} : TG \to TG'$ and a mapping of production names $f_P : P \to P'$ such that for all $p \in P$ and $p' = f_P(p) \in P'$ where $\pi(p) = s$ if $A(s)$ and $\pi'(p') = s'$ if $A(s')$

1. there exists a TG-typed span morphism $e : s \to f^{<}_{TG}(s') \in \mathbf{Sp}^{DPB}_{TG}$ forming a faithful transition (cf. Fig. 3.7 on the right), and

2. applicability of p implies that of p' in $\mathbf{Graph}_{TG'}$, i.e.

   (a) for each $f^{>}_{TG}(L \xrightarrow{\hat{l}} \hat{L}) \in f^{>}_{TG}(AP(s))$ there exist $L' \xrightarrow{\hat{l}'} \hat{L}' \in AP(s')$ and a graph homomorphism $\hat{h}_P : \hat{L}' \to f^{>}_{TG}(\hat{L})$ such that the corresponding square in Fig. 3.7 on the left commutes;

   (b) for each $L' \xrightarrow{\hat{k}'} \hat{L}' \in AN(s')$ there exist $f^{>}_{TG}(L \xrightarrow{\hat{k}} \hat{L}) \in f^{>}_{TG}(AN(s))$ and a graph homomorphism $\hat{h}_N : f^{>}_{TG}(\hat{L}) \to \hat{L}'$ such that the corresponding square in Fig. 3.7 on the left commutes.

△

Notice that the effect parts of the conditional productions are matched in the type context of $\mathcal{C}$ in contrast to the precondition parts compared in the type context

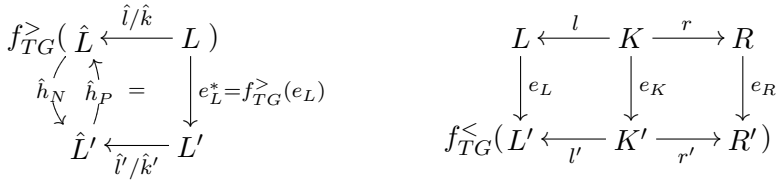


Figure 3.7: Substitution morphism and conditional productions (the functors $f^{>}_{TG}$ and $f^{<}_{TG}$ are applied to the entire application constraint of p in the left part of the figure and to the entire bottom span in the right part of the figure, respectively).

of $\mathcal{C}'$. The precondition part of the requestor production is forwardly retyped for the matching, because the input data specified by this precondition will be interpreted in the type system of the provider executing the operation.

Analogously, the effect part of the provider production is backwardly retyped, since the result of the operation will be interpreted and used in the requestor type system. Moreover, the typed span morphism $f^{>}_{TG}(s) \to s' \in \mathbf{Sp}^{DPB}_{TG'}$ does not always imply the existence of the typed span morphism $s \to f^{<}_{TG}(s') \in \mathbf{Sp}^{DPB}_{TG}$ which has to be actually ensured. For example, if the type graph morphism f_{TG} is not injective then the implication does not hold (see Section 4.2 for a general discussion).

3.2.3 Justification of Substitution Morphism

While the semantic requirements on structural compatibility are directly reflected by the signature morphism, a correspondence between the semantic requirements on behavioral compatibility and substitution morphism is not so obvious. The justifications for the definition of the substitution morphism are presented in the following theorem.

Theorem 3.2.6. *The semantic requirements of Def. 3.2.2 hold if and only if the syntactic requirements of Def. 3.2.5 hold.*

Proof. *"If"*: We have to show that Def. 3.2.5 implies Def. 3.2.2. The existence of a functor between two categories of sequences requires that each individual step in $\mathcal{C}'$ is mapped to a sequence in $\mathcal{C}$. By induction, this mapping is extended to sequences in $\mathcal{C}$. However, we will deal with the simpler case where a transformation step in $\mathcal{C}'$ is actually mapped to a single transition in $\mathcal{C}$.

One should demonstrate, first of all, that transformation steps via $\mathcal{C}'$ production can be considered as transitions via the corresponding $\mathcal{C}$ production. Secondly, ap-

plicability of this $\mathcal{C}$ production has to imply applicability of the $\mathcal{C}'$ production under the construction of the transformations and transitions associated by the functor F.

Assume two conditional productions $p : s \text{ if } A(s)$ and $p' : s' \text{ if } A(s')$ where $p \in P$ and $p' = f_P(p) \in P'$.

1. Let us apply backwardly retyped p' to the graph $f^{<}_{TG}(G)$ at $f^{<}_{TG}(d_{L'})$ and construct a transformation step with the underlying span $f^{<}_{TG}(G \xleftarrow{g} D \xrightarrow{h} H)$ as depicted in Fig. 3.8 on the right. Since l' and r' are injective, this transformation step is also a (faithful) transition. By assumption, for each pair of productions $p : s \text{ if } A(s)$ and $f_P(p) : s' \text{ if } A(s')$ there exists a TG-typed span morphism $e : s \to f^{<}_{TG}(s') \in \mathbf{Sp}^{DPB}_{TG}$ forming a faithful transition. Now, both transitions can be vertically composed using the composition of the underlying pullback squares. The faithfulness of the composed transition via the production of $\mathcal{C}$ follows from the preservation of the identification condition under the composition of pullback squares.

2. It is left to show that if $f^{>}_{TG}(d_L)$ satisfies the application condition of forwardly retyped p, then $d_{L'}$ satisfies the application condition of p'. This induces two problems:

   (a) $d_{L'}$ (cf. Fig. 3.8 on the left) must *P-satisfy* at least one positive constraint of p'. Since $\hat{h}_P$ exists by assumption (Def. 3.2.5.(2a)), $d_{\hat{L}'}$ can be constructed by $f^{>}_{TG}(d_{\hat{L}}) \circ \hat{h}_P$. It is not difficult to see that $d_{\hat{L}'} \circ \hat{l}' = d_{L'}$.

   (b) $d_{L'}$ (cf. Fig. 3.8 on the left) must *N-satisfy* all negative constraints of p', i.e. there does not exist $d_{\hat{L}'} : \hat{L}' \to G$ such that $d_{\hat{L}'} \circ \hat{k}' = d_{L'}$. Assume to the contrary the existence of $d_{\hat{L}'}$. Since $\hat{h}_N$ exists by assumption (Def. 3.2.5.(2b)), we can construct $f^{>}_{TG}(d_{\hat{L}}) = d_{\hat{L}'} \circ \hat{h}_N$ with $f^{>}_{TG}(d_{\hat{L}}) \circ \hat{k} = f^{>}_{TG}(d_L)$ which is a contradiction.

   Thus, $d_{L'}$ satisfies the application condition of p'.

Combining the two parts of the proof, we obtain that the functor specified in Def. 3.2.2 can indeed be constructed.

"Only if": Assume two conditional productions $p : s \text{ if } A(s)$ with $s = (L \xleftarrow{l} K \xrightarrow{r} R)$ and $p' : s' \text{ if } A(s')$ with $s' = (L' \xleftarrow{l'} K' \xrightarrow{r'} R')$ such that $p' = f_P(p)$. To prove that Def. 3.2.2 implies Def. 3.2.5.1/2, respectively.

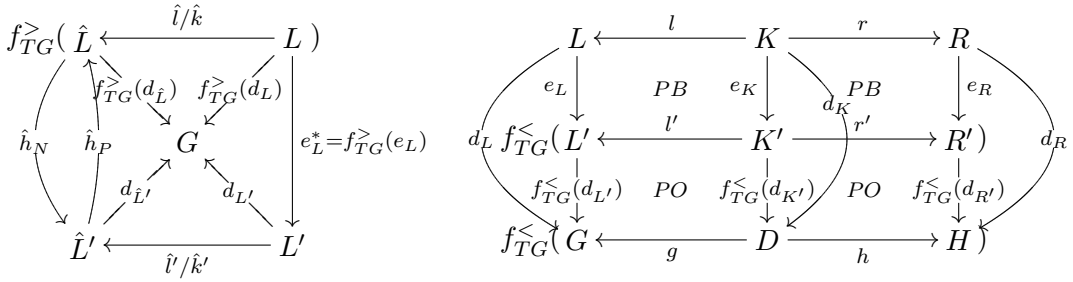


Figure 3.8: Substitution morphism implies semantic requirements (the functors $f^{>}_{TG}$ and $f^{<}_{TG}$ are applied to the entire application constraint of p in the left part of the figure and to the entire bottom span in the right part of the figure, respectively).

1. To show that there exists $e : s \to f^{<}_{TG}(s') \in \mathbf{Sp}^{DPB}_{TG}$, we can apply backwardly retyped p' to the graph $f^{<}_{TG}(L')$ at the identity mapping. If p' is applicable, we can create a transformation step with the bottom span $f^{<}_{TG}(s') = f^{<}_{TG}(L' \xleftarrow{l'} K' \xrightarrow{r'} R')$. Consequently (see Def. 3.2.2), there exists a faithful transition $f^{<}_{TG}(L') \overset{p/e}{\rightsquigarrow} f^{<}_{TG}(R')$ via the production p using the same bottom span $f^{<}_{TG}(s')$. If we can not apply backwardly retyped p' to the graph $f^{<}_{TG}(L')$, then the premise is false, and the conclusion is trivially true.

2. Two questions have to be considered for the second part of Def. 3.2.5:

   (a) The existence of a graph homomorphism $\hat{h}_P : \hat{L}' \to f^{>}_{TG}(\hat{L})$ between the positive precondition patterns of the productions. We can apply forwardly retyped p to the graph $f^{>}_{TG}(\hat{L})$ at $m := f^{>}_{TG}(\hat{l})$ where $\hat{l} \in AP(s)$ as depicted in Fig. 3.9 on the left. Since m satisfies $f^{>}_{TG}(\hat{l})$, and Def. 3.2.2 entails preservation of applicability from $\mathcal{C}$ to $\mathcal{C}'$, there exists m' satisfying the constraint $\hat{l}' \in AP(s')$ of p'. This implies the existence of a graph homomorphism $\hat{h}_P : \hat{L}' \to f^{>}_{TG}(\hat{L})$. The commutativity of the corresponding square in Fig. 3.9 on the left follows from the commutativity of the diagrams (1),(2),(3), and (4). The commutativity of the diagrams (1),(2), and (3) can easily be shown. To prove that the diagram (4) commutes, one has to assume the existence of a typed graph morphism $f^{>}_{TG}(L) \to f^{>}_{TG}(\hat{L}) := m' \circ e^*_L \neq m$ and obtain a contradiction.

   (b) The existence of a graph homomorphism $\hat{h}_N : f^{>}_{TG}(\hat{L}) \to \hat{L}'$ between the negative precondition patterns of the productions. We can apply forwardly retyped p to the graph $\hat{L}'$ at some n as depicted in Fig. 3.9 on the right. Def. 3.2.2 implies that if n *satisfies* $f^{>}_{TG}(\hat{k})$ for $\hat{k} \in AN(s)$, then n' *satisfies*

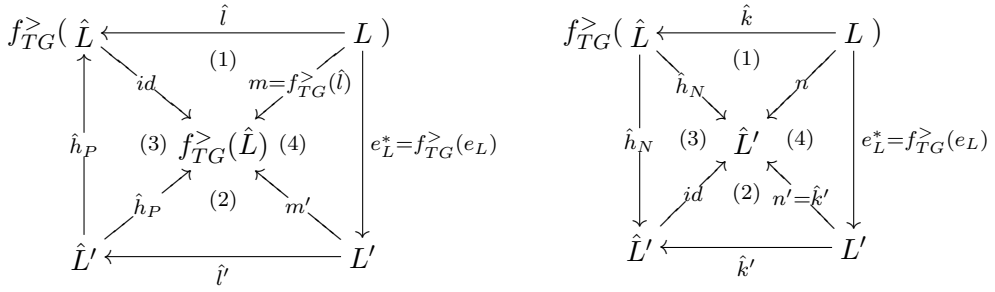


Figure 3.9: Semantic requirements implies substitution morphism (the functor $f^{>}_{TG}$ is applied to the entire application constraint of p).

$\hat{k}' \in AN(s')$. We can reformulate this as: if n' *does not satisfy* $\hat{k}'$, then n *does not satisfy* $f^{>}_{TG}(\hat{k})$.

Now we try to apply p' to the graph $\hat{L}'$ at $n' := \hat{k}'$. It is possible to see that the premise of the statement above is true (n' *does not satisfy* $\hat{k}'$), so is the conclusion, i.e. n *does not satisfy* $f^{>}_{TG}(\hat{k})$. This may happen only if there exists a graph homomorphism $\hat{h}_N : f^{>}_{TG}(\hat{L}) \to \hat{L}'$ that was required to be proved. The commutativity of the corresponding square in Fig. 3.9 on the right follows from the commutativity of the diagrams (1),(2),(3), and (4). The only problem here is the commutativity of the diagram (4) which can be solved analogously to (a).

□

Once we have justified the established substitution morphism, we turn to integral compatibility between service interface specifications.

3.2.4 Parameterized Substitution Morphism

The integral compatibility relation combining the features of the structural and behavioral ones is formalized by a *parameterized substitution morphism*. This morphism associates the required an provided service interfaces specifications in the form of parameterized conditional graph transformation systems.

Definition 3.2.7 (parameterized substitution morphism). Given parameterized conditional graph transformation systems $\mathcal{CP} = \langle TG, P, CP \rangle$ and $\mathcal{CP}' = \langle TG', P', CP' \rangle$, a *parameterized substitution morphism* $f^{sub+} = \langle f_{TG}, f_P \rangle$ is a signature morphism given by a type graph morphism $f_{TG} : TG \to TG'$ and a mapping of production names $f_P : P \to P'$, where for all $p \in P$ and $p' = f_P(p) \in P'$ the

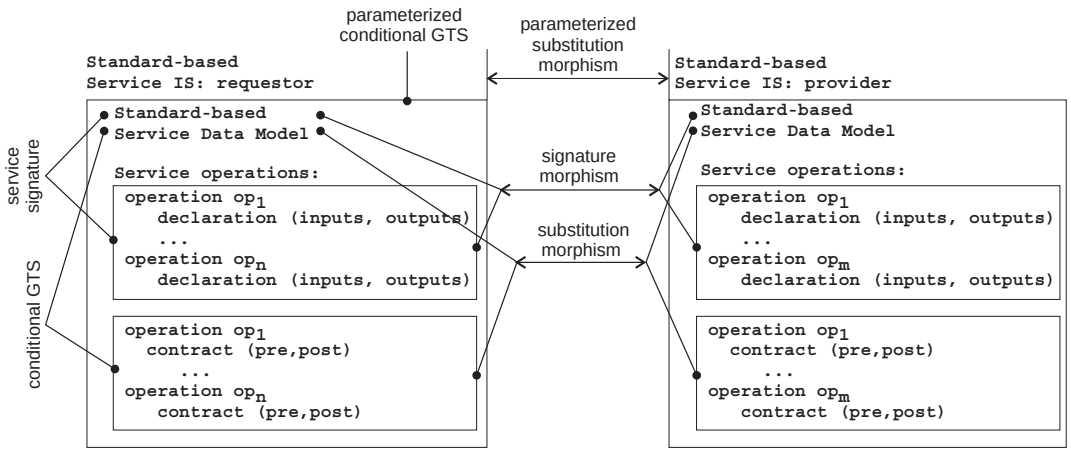


Figure 3.10: Formalization of service interface specifications and their matching.

conditional spans of productions $cp = p(x, y) : s$ if $A(s) \in CP$ and $cp' = p'(x', y') : s'$ if $A(s') \in CP'$ satisfy Requirements 1 and 2 of Def. 3.2.5. $\triangle$

Depending on the kind of signature morphism employed in the definition (see Table 2.2), we distinguish between the *exact*, *input-preserving*, *output-preserving*, and *generalized parameterized substitution morphisms*. In the rest of the thesis we stick to the exact version of the morphism.

A number of examples illustrating an application of parameterized substitution morphism can be found in Subsections 4.3.2 and 4.3.3 of the next chapter.

The final section summarizes the main results of this chapter and relates them to the concepts developed in the previous chapter.

3.3 Summary

In this chapter, we have introduced a formal background for the approach to service specification and matching discussed in Chapter 2. An overview of our formalization is presented in Fig. 3.10.

The structural description of a service is given by a signature similar to those known from algebraic specifications [50]. A correspondence between such signatures, referred to in Subsection 2.3.1 as the structural compatibility relation, is defined by the signature morphism.

The conditional graph transformation systems proposed in Subsection 2.2.2 to specify service behavior are considered in categorical setting of the DPO and DPB approaches to graph transformation. This style of presentation allows to describe

formally the semantic requirements underlying our approach and to represent the behavioral compatibility relation constructed in Subsection 2.3.2 by means of a substitution morphism over conditional GTSs. We have proved the equivalence between the syntactic requirements imposed by the substitution morphism and the established semantic requirements for behavioral compatibility of service interface specifications.

Developing a formal counterpart of the integral service description and the integral compatibility relation presented in Subsections 2.2.3 and 2.3.3 respectively, we have augmented the theory of graph transformation with the notions of parameterized conditional GTSs and parameterized substitution morphisms relating these GTSs. The latter has been obtained by extending the substitution morphism with a structural check implemented by the signature morphism.

Chapter 4

Modular Specifications of Coupled Systems

In Chapters 2 and 3, we have discussed interface specifications augmented with behavioral information and their inter-connecting in the context of cross-organizational interactions. Now the focus of our presentation is extended beyond external specifications. We construct a model aggregating external and internal specifications of coupled systems or components. Their correct functioning predetermines proper work of the compound system—our model allows to validate required properties of coupled components.

This chapter is organized as follows. In the next section we position our proposal with respect to the existing component models, discuss the use of model-based testing in the application integration, and recall the basic concepts of graph transformation modules used for construction of the component model. In Section 4.2, we introduce a framework for classifying and systematically defining morphisms of graph transformation systems, we also construct two sample morphisms in the context of this framework. The introduced morphisms as well as a parameterized substitution morphism appearing in Subsection 3.2.2 are employed in Section 4.3 for intra- and inter-connectors of modules. Finally, this section introduces a formal definition of the component model specified by graph transformation modules. Section 4.4 concludes the chapter with a summary of the obtained results.

The concepts and ideas of this chapter are partially based on the joint work with G. Engels and R. Heckel (see [53]).

4.1 Software Component Models

A survey of the literature reveals a wide range of fundamentally diverse proposals for component models which can be divided into three main groups as proposed in [103]. The first group consists of models in which components are defined by object-oriented programming languages. *JavaBeans* [109] and *Enterprise Java Beans* (EJB) [110], where components are obviously implemented in Java, represent examples belonging to this group. The second group is characterized by the employment of an IDL (interface definition language) together with a mapping from the IDL to programming languages for component implementations. Component models of this group are, e.g., COM using the Microsoft IDL [18], CORBA using the OMG IDL [115], or Fractal [22] which can employ any IDL.

Those models in which components are defined by means of architecture description or (formal) specification languages make up the third group. For example, UML [116], KobrA [8], Koala [142], SOFA [121], PECOS [113] and Pin [92] illustrate approaches based on architecture description languages. An overview of these approaches is presented in [30].

The importance of formal specification languages for component models has been well-recognized in software engineering [58, 65]. A component model constructed in this chapter is based on proposals which originate in the algebraic specification and graph transformation domains, where different kinds of built-in *modularity* concepts are extensively employed to adequately reflect the structural and behavioral features of software components. A detailed discussion on these proposals and their comparison with our model appear in Section 5.3.

It is worth noting that any model should be equipped with an appropriate mechanism for checking its correctness. Therefore, in the next subsection, we overview techniques designed for analysis of models to be constructed.

4.1.1 Model-based Testing Techniques

Analytical means to allow the reasoning about properties of software systems are usually divided into verification and validation techniques. The first group of techniques, such as model checking or theorem proving, have a long tradition in formal specification languages such as algebraic specifications [49], Z [136], CSP [86], or Petri nets [125]. Despite of the relative maturity of formal verification within academic community, its practical use in software engineering is complicated by the following

reasons. Firstly, it is the complexity of formal specification techniques and the lack of training of software engineers in applying them. Secondly, there are also well-known limitations of formal verification such as the state-explosion problem within model checking. Common to all verification techniques is that they based on a formal semantics of applied specification or programming languages, where an examined property may be assured with mathematical rigor.

While validation techniques may detect errors in the system being investigated, they cannot prove any property in a definite way as it happens in formal verification. One of the classic techniques widely used for software system validation is *testing*. Testing is based on the construction of test strategies for a property including subsequent execution of parts or all of the system according to these strategies [14]. Since testing takes place at a lower level of abstraction, the range of properties that can be validated is much greater than using formal verification [54].

With the advance of model-based software development approaches many attempts have been made to develop testing methods rely on software system models [21]. The major assumption underlying the model-based tasting is the existence of a system model which is used for studying the system. In particular, the system model can be employed to generate complete test suites to show conformance of the model and the actual implementation, or just to derive "interesting" test cases to examine specific system's characteristics.

As already mentioned, the successful use of application integration techniques is stipulated by the possibility to validate specifications of the coupled components. On one hand, the existing model-based tasting approaches may be extensively employed for this task. On the other hand, each validation approach needs an input data which in our case is provided by the component model. It is our objective to establish a procedure for the construction of such model.

4.1.2 Modeling Components by Module Specifications

The major problem has to be considered in the scope of the model construction is how to integrally portray a component, i.e. its internal and external parts along with relationships between them. In our approach the notion of *module* facilitates this process.

Modularization is a well-known concept to structure software systems and their specifications. Modules have been initially introduced in the programming language frameworks, e.g., Ada packages [97] or Modula-2 modules [98]. The first steps in the

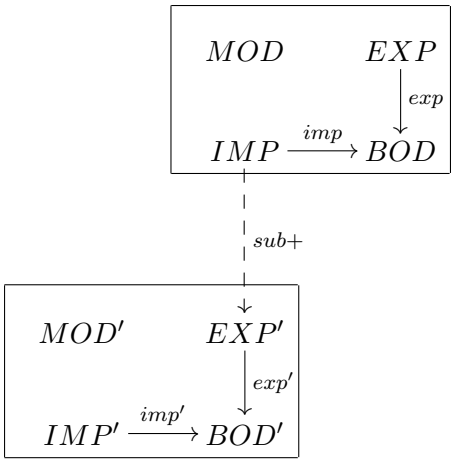


Figure 4.1: Module specifications and their inter-connecting.

application of modules in modeling of software systems have been made in the context of algebraic and logic specifications.

For example, an algebraic specification module MOD [50] consists of a body BOD providing the implementation and of interfaces IMP for import and EXP for export describing, respectively, required and provided functionality. (In addition, a parameter PAR is provided to allow for generic modules, but this feature will not be relevant for our purposes.) All specifications are connected through algebraic specification morphisms (cf. Fig 4.1).

Since the mid nineties [47], there is an increasing interest in the transfer of modularity concepts from algebraic specifications and programming languages to graph transformation systems [101, 131, 69] (see also the survey in [82]). For instance, modules of typed graph transformation systems (TGTS modules) in [69] follow the structure of algebraic specification modules, replacing the specifications BOD, IMP, and EXP by graph transformation systems intra-connected by different kinds of morphisms.

4.1.3 Consistency of Interface Specifications

At least five fundamentally different notions for morphisms of graph transformation systems [34, 126, 68, 80, 83] can be found in the literature. They reflect a wide range of objectives, such as inclusions, refinements, or views and enjoy diverse semantic properties.

So far there has been no general approach for comparing and relating the GTS morphisms. Hence, to choose appropriate candidates for the module intra-connectors

one has to establish a framework to systematically handle the existing proposals, and if necessary to construct new ones. We introduce such framework that makes it possible to review the existing definitions and to provide a recipe for deriving the appropriate definitions from given semantic requirements.

Moreover, the intra-connectors constructed in the framework rigorously determine the consistency relations between the interface specifications and component bodies. While the *export-body* intra-connector guarantees that the service offered at the export interface is indeed implemented by the component, the *import-body* one ensures that the component can benefit from the functionality asked for at the import interface.

Having defined consistent specifications of the source and target components by the modules MOD and MOD', the integration of the parties now means to relate the import (required) interface IMP of MOD with the export (provided) interface EXP' of MOD' by the inter-connector, such as parameterized substitution morphism introduced in the previous chapter (cf. Fig. 4.1).

4.1.4 Requirements

The following list of requirements for a technique which allows to obtain a component model conducts our presentation.

1. *Rigorous formulation of the consistency relations between interface specifications and internal part of an integrating component.* The complexity of the coupled components makes an analysis of their internal and external specifications quite complicated task. If such analysis is carried out by software engineers manually, then it becomes inefficient. Therefore, one has to provide an adequate tool support that in turn requires the consistency relations to be rigorously formulated.

2. *Formal visual notation of component model and its compliance with standard model-driven techniques of software development.* This requirement is imposed by the same motivations accompanying Requirements 1 and 2 stated in Subsection 2.1.5 for the approach to service specification and matching.

3. *Aptitude of the component model for model-based testing.* Since our major goal is to enable the model-based testing of the coupled components, their integral specifications have to be compatible with the verification techniques intended to be applied.

Now, let us turn to the detailed presentation of our approach.

4.2 GTS Morphisms, Systematically

In this section, we establish a framework for classifying and systematically defining GTS morphisms based on a number of standard "ingredients", such as homomorphisms between type graphs and mappings between sets of productions. After introducing in the next subsection an example scenario containing a specification of mutual exclusion algorithm, in Subsection 4.2.2 two sample morphisms will be discussed informally in the context of the introduced example. Then, in Subsection 4.2.3, the constituents of the framework are presented and combined, yielding definitions of the sample morphisms. In Subsection 4.2.4, we discuss a locus of the parameterized substitution morphism in the framework and its relation with the existing proposals.

4.2.1 Example Scenario

As running example, a specification of a mutual exclusion algorithm with deadlock detection is developed throughout this chapter. The body of the module $MUTEX$ which describes a component implementing a part of the algorithm, is shown in the example below. The import and export interfaces of $MUTEX$ will appear in the next subsection.

Example 4.2.1 (MUTEX). The graph transformation system in Fig. 4.2 models a distributed algorithm for mutual exclusion (MUTEX). This example is derived from a small case study in [76] and tailored to our presentation.

Two basic types, processes P (drawn as black nodes) and resources R (drawn as light boxes), constitute the type graph presented in the upper-left corner of the figure. While a resource *request* is modeled by an edge going from a process to a resource, the fact that the resource is currently *held_by* the process is shown by an edge in the opposite direction.

The mutual exclusion is implemented by a token ring algorithm. The processes in the token ring are arranged in a cycle. Two neighbor processes are connected by an edge running from the current to the *next* process. This edge is given by a loop in the type graph. A default position for introducing new processes and resources is marked by the head pointer h.

An edge with a white flag denotes a token which is passed from process to process along the ring. In order to get an access to a resource, a process waits for the cor-

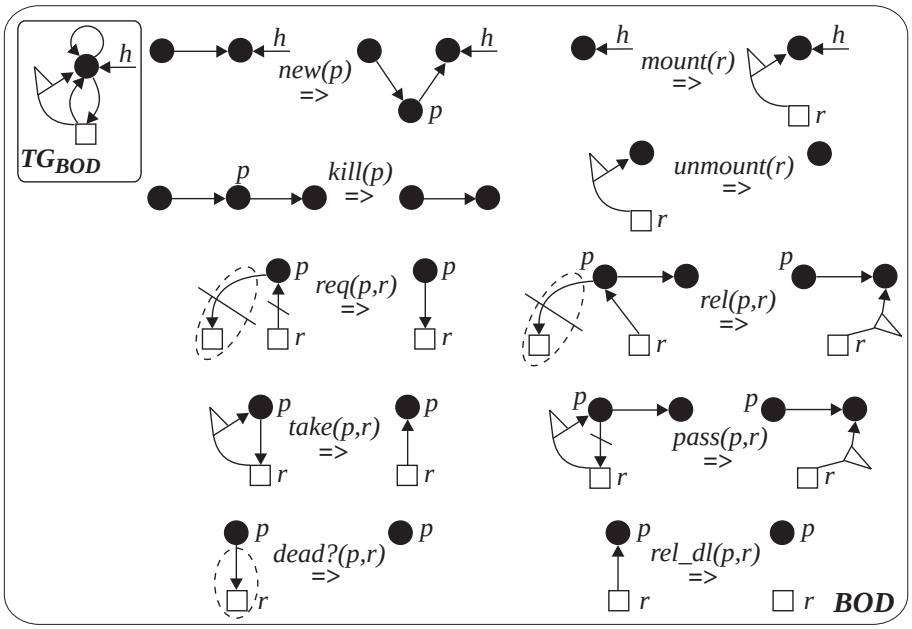


Figure 4.2: Graph transformation system modeling body of the module MUTEX.

responding token. Mutual exclusion is achieved by uniqueness of the token for each resource in the system.

Productions of the graph transformation system are interpreted as follows. The first four productions are used for creating and killing processes (*new* and *kill*), and for mounting and unmounting resources (*mount* and *unmount*). The productions *req*, *take*, and *rel* allow processes to issue requests, take resources, and release them upon *regular* completion of their task, respectively. The negative application conditions of the production *req* ensure that a process cannot issue more then one request at a time. The negative application condition of the production *rel* prevents the release of a resource r while the process requests another resource, since r may still be required to complete the given task.

The last two productions *dead*? and *rel_dl* are intended to be applied in deadlock situations which may result from competition of processes for non-sharable resources. Since the MUTEX algorithm does not have the capability to detect deadlocks, the production *dead*? assumed to be obtained as the import from another module. The dotted part of this production represents a positive application condition. This condition restricts applicability of the production to situations where the process has a pending request for a resource.

The production *rel_dl* implements the resolution of detected deadlocks by forcing

Figure 4.3: GTSs modeling import IMP (left) and export EXP (right) interfaces of the module MUTEX.

the release of the resource held by the involved process. △

4.2.2 Candidates for Module Intra-connectors

In general, a GTS morphism $f : \mathcal{G} \to \mathcal{G}'$ defines a relation between the behaviors issued by the systems $\mathcal{G}$ and $\mathcal{G}'$. So, a systematic approach to the analysis of such relations should always start by identifying the kind of semantic properties assumed to be ensured.

Preservation of Behavior

We start with an example of *behavior-preserving* morphisms providing a first attempt at describing the relation between the export interface EXP of a module with its body BOD (cf. Fig 4.1). Since the export interface specifies the features offered for import by other modules, the specification of these features in EXP has to be consistent with their implementation in the body. That means, the behavior induced by EXP should be preserved by BOD. In particular, it is necessary to guarantee that applicability of EXP productions implies applicability of the corresponding BOD productions. The mission of behavior-preserving morphism is to ensure this property.

Example 4.2.2 (behavior-preserving morphism). A service provided by the module MUTEX is deadlock resolution specified by the production *rel_dl* in the export interface EXP (cf. Fig. 4.3 on the right). It shall be imported by an external deadlock detection module to break up detected deadlocks. In order to guarantee that BOD preserves the behavior of EXP, each transformation sequence in EXP should imply a corresponding sequence in BOD.

Comparing the GTS EXP and BOD, first of all we examine a relation between their type graphs. In the given example, the type graph TG_{EXP} containing all the types relevant for deadlock resolution is a subgraph of TG_{BOD}. More generally, a morphism between type graphs ensures that all types of the source system (in our

case TG_{EXP}) have a correspondence in the target (TG_{BOD}). If the morphism is not an inclusion, a type in the target may have a different name than its source, or two different types in the source may be mapped to the same target type.

As discussed in Section 2.3 (see Def. 3.2.3), the type graph morphism enables to convert graphs, productions, and also transformations typed over the source into such typed over the target by a renaming of their types. This gives us the opportunity to compare two systems by translating the productions of the source system into ones typed over the target.

Due to the subgraph relation between TG_{EXP} and TG_{BOD} the translation of the EXP production to the type system of BOD does not change anything in this production. The comparison reveals that the production identical to *rel_dl* is already present in TG_{BOD}, even with the same name. Generally, we might consider a mapping of production names if different names have been used for corresponding productions in the two systems. So, it seems to be the case that the required implication between the transformation sequences of EXP and BOD holds. △

The behavior-preserving morphisms as discussed above are originally introduced in [67, 68]. While in our example the export interface EXP is just a subsystem of BOD, in [68] the relation between export interface and body may be represented by *spatial* or *temporal* refinements.

Reflection of Behavior

We proceed with an example of *behavior-reflecting* morphisms determining a relation between the import interface IMP and the body BOD of a module (cf. Fig 4.1). The idea is that the productions required at IMP have at least the effect of the productions specified at BOD. Otherwise, the body could not use the imported productions in the internal implementations. This can be expressed as a reflection of the BOD transformations by IMP transitions.

Example 4.2.3 (behavior-reflecting morphism). As mentioned already, deadlock detection represents an external feature abstractly described by the production *dead*? in the import interface IMP (cf. Fig. 4.3 on the left).

Reflection of BOD behavior by IMP means that for each transformation in BOD we require a corresponding transition in IMP. As with behavior-preserving morphisms, we have to specify relations between type graphs and productions of the two systems.

For type graphs, a morphism from TG_{IMP} to TG_{BOD} ensures that BOD has at least the same types as IMP. In order to check that transformations in BOD are reflected by transitions in IMP, we have to compare productions of the two systems. Since we are interested in reflection rather than preservation of steps, we translate the productions of BOD to IMP *against* the direction of the type graph morphism. That means, besides the renaming of types, elements of the productions are removed if their type in BOD does not have a pre-image in IMP under the type graph morphism.

Then, the BOD behavior is reflected by IMP if each IMP production can be embedded into the translated version of the corresponding BOD production. In our case, the production *dead*? of BOD coincides with the one in IMP after the translation. △

Morphisms which reflect transformations in the target system by transitions in the source one have been introduced in [76] to specify the relation between different views of a system model.

4.2.3 Framework for GTS Morphisms

So far we have informally discussed how semantic requirements determine the definitions of GTS morphisms. Now, we are going to make this explicit in terms of a four-step recipe. First, however, we introduce a notation for specification of production span relations (cf. Def. 3.1.3). This notation will be used in the construction of recipes for GTS morphisms.

Production Span Relations

A list of possible relations between production spans typed over the same type graph is presented in the following definition. For the moment we ignore the parametrization and application conditions.

Definition 4.2.4 (production span relations). Given productions $p : s$ and $p' : s'$ with $s, s' \in \mathbf{Sp}_{TG}$, and a typed span morphism $e : s \to s'$, we say that

- the span s is *identical* to the span s', written $s \xrightarrow{id} s'$, if the typed span morphism e is the identity in $\mathbf{Sp}_{TG}$,
- the span s is a *DPO-subspan* of the span s', written $s \xrightarrow{\rightarrow} s'$, if the typed span morphism $e \in \mathbf{Sp}_{TG}^{DPO}$,

- the span s is a *DPB-subspan* of the span s', written $s \overset{\rightsquigarrow}{\longrightarrow} s'$, if the typed span morphism $e \in \mathbf{Sp}_{TG}^{DPB}$ and forms a faithful transition.

△

This list could be further extended by relations between a single production and a collection of productions such as the *spatial* and *temporal* refinements, but this is beyond the scope of our presentation

Recipe for Construction of GTS Morphisms

Now we are in the position to construct a four-step recipe for GTS morphisms. In each step we introduce a number of options and motivate possible choices for the recipe ingredients based on the semantic requirements. First of all, we formulate an initial assumption underlying the construction.

Assumption: Without loss of generality we assume that the GTS morphism $f : \mathcal{G} \to \mathcal{G}'$ for the graph transformation systems $\mathcal{G} = \langle TG, P, \pi \rangle$ and $\mathcal{G}' = \langle TG', P', \pi' \rangle$, and the type graph morphism $f_{TG} : TG \to TG'$ have the same direction. That means, the target system has at least the types as the ones of the source system, but possibly more.

Step 1. The first variation point is the *relation between the sets of production names* of $\mathcal{G}$ and $\mathcal{G}'$. Here it is most convenient to use total functions, rather than general relations. For example, a mapping from P to P' designates for each $p \in P$ one corresponding $p' \in P'$—the relation is *left total* and *right unique*. This option should be used for behavior-preserving morphisms, where each transformation of the source system has to be associated with a transformation of the target system. Dually, a mapping in the opposite direction provides for each $p' \in P'$ one $p \in P$—*left unique* and *right total* relation which is suitable for the behavior-reflecting morphisms.

Step 2. The next alternative is introduced by the *context of comparison*, i.e. where the corresponding productions of the two systems are compared. This can be done either in the context of $\mathcal{G}'$ using the forward retyping $f_{TG}^{>} : \mathbf{Graph_{TG}} \to \mathbf{Graph_{TG'}}$ of $\mathcal{G}$ productions, or in the context of $\mathcal{G}$ using the backward retyping $f_{TG}^{<} : \mathbf{Graph_{TG'}} \to \mathbf{Graph_{TG}}$ of $\mathcal{G}'$ productions (cf. Def. 3.2.3). The forward retyping is appropriate for behavior-preserving morphism, since the objective in this case is

the construction of transformations in the target from the ones existing in the source system. By analogy, the backward retyping is used for behavior-reflecting morphisms.

We continue with the specification of production span relations between productions of the two systems. For pairs of corresponding productions as defined in *Step 1* and modulo the retyping functor selected in *Step 2* this means to decide for the direction of the relation in *Step 3* and its kind in *Step 4*.

Step 3. The *direction of the production span relation*, i.e. if the span of the production p is required to be a subspan of p', or vice versa, depends on the desired relation between the sets of transformations or transitions of the two systems. If the span of p is a subspan of p' then each transformation step via p' implies a transformation step or transition via p. Thus, behavior-preserving morphisms generally require that $\mathcal{G}'$ production spans turn to be subspans of $\mathcal{G}$ productions, while behavior-reflecting morphisms specify the dual requirement.

Remark 4.2.5. Note that it may be the case that a production span relation between $\pi(p)$ and $\pi'(p')$ holds when considered over the larger type graph of $\mathcal{G}'$ using forward retyping, but not if compared via backward retyping (projection) over the smaller type graph of $\mathcal{G}$. The converse is also true, i.e. a production span relation may hold over $\mathcal{G}$, but not over $\mathcal{G}'$.

This motivates why the comparison of productions is always done in a type context of the system where the existence of transformations or transitions should be ensured, i.e. the target system if behavior shall be preserved, and the source system if behavior shall be reflected.

Step 4. Finally, we have to select the *kind of production span relation* that the comparison shall be based upon. The identity between the spans of p and p' ensures that all transformations via p are also transformations via p'. If the span of p is a DPO- or DPB-subspan of p', respectively, then each transformation step via p' implies a transformation ($\xrightarrow{}$) or transition ($\overset{\sim}{\longrightarrow}$) via p. The dual holds if we replace p and p'.

Relations between different choices are summarized in Table 4.1. Combinatorially, we obtain eight different notions. Numbers 4 and 5 represent, respectively, behavior-reflecting and behavior-preserving morphisms discussed above.

Next we formally introduce the semantic requirements for behavior-preservation and behavior-reflection and definitions of the morphisms.

	forward retyping $f^{>}_{TG}$		backward retyping $f^{<}_{TG}$	
	left-total right-unique relation	left-unique right-total relation	left-total right-unique relation	left-unique right-total relation
$\xrightarrow{*}$	– 1	– 2	DPO/DPB_{ϵ} 3	DPO,DPB,[=] 4
$\xleftarrow{*}$	=,[DPO/DPB] 5	– 6	– 7	– 8

Table 4.1: Ingredients of GTS morphism recipe.

Behavior-preserving Morphism

The semantic requirements for preservation of behavior between the graph transformation systems $\mathcal{G}$ and $\mathcal{G}'$ are presented in the following definition.

Definition 4.2.6 (preservation of behavior). Given graph transformation systems $\mathcal{G} = \langle TG, P, \pi \rangle$ called the source system and $\mathcal{G}' = \langle TG', P', \pi' \rangle$ called the target system, we say that the target system preserves the behavior of the source system if there exists a functor $F : \mathbf{Trf}(\mathcal{G}) \to \mathbf{Trf}(\mathcal{G}')$. △

As discussed above, this requires that each production $p \in P$ has a corresponding production $p' \in P'$. Hence, a mapping $f_P : P \to P'$ is chosen in *Step 1*. To ensure the preservation of sequences in $\mathcal{G}'$, the comparison of production spans is done in the context of $\mathcal{G}'$ and, therefore, forward retyping is applied in *Step 2*.

The mapping in *Step 1* must guarantee the desired relation between the transformations in the two systems. This is achieved if in *Step 3* the production spans in $\mathcal{G}$ are not extended by those in $\mathcal{G}'$. The choices in *Step 4* ensuring behavior preservation range from identity to DPB-subspan relations. The identity is the most common one, because it results in the embedding of $\mathcal{G}$ into $\mathcal{G}'$, while the other two variants would mean that the productions of $\mathcal{G}$ are reduced in $\mathcal{G}'$.

The behavior-preserving morphism specified in cell 5 of Table 4.1 is formally defined below.

Definition 4.2.7 (behavior-preserving morphism). Given graph transformation systems $\mathcal{G} = \langle TG, P, \pi \rangle$ and $\mathcal{G}' = \langle TG', P', \pi' \rangle$, a *behavior-preserving morphism* $f^{pres} = \langle f_{TG}, f_P \rangle$ is given by a type graph morphism $f_{TG} : TG \to TG'$ and a

mapping of production names $f_P : P \to P'$ such that for each $p \in P$ there exists $e : \pi'(f_P(p)) \to f^{>}_{TG}(\pi(p))$ being the identity in $\mathbf{Sp}_{TG'}$. △

The justifications for the following claim can be found in [67, 68].

Fact 4.2.8. Behavior-preserving morphisms $f^{pres} : \mathcal{G} \to \mathcal{G}'$ satisfy the requirements of Def. 4.2.6. □

Just to consider another example, the candidate in cell 6 differs from the one above in the direction of the mapping between production names. That means, to each $p' \in P'$ a $p \in P$ is associated. If we require the existence of the production span relation for all pairs of productions thus associated, this guarantees a partial preservation of behavior only, i.e. for those transformations in $\mathcal{G}$ via productions with corresponding productions in $\mathcal{G}'$.

Behavior-reflecting Morphism

To continue on the right-hand side of the table, the semantic requirements for behavior-reflecting morphisms are specified below.

Definition 4.2.9 (reflection of behavior). Given graph transformation systems $\mathcal{G} = \langle TG, P, \pi \rangle$ called the source system and $\mathcal{G}' = \langle TG', P', \pi' \rangle$ called the target system, we say that the source system reflects the behavior of the target system if there exists a functor $F : \mathbf{Trf}(\mathcal{G}') \to \mathbf{Trs}(\mathcal{G})$. △

That means, each transformation step in $\mathcal{G}'$ implies a *transition* in $\mathcal{G}$, a liberal requirement compared to reflecting transformations in *transformations*.

By the same arguments as above, in *Step 1* we assume a mapping of production names from P' to P. The context of comparison is the source system that leads to the use of backward retyping in *Step 2*. To fulfill the semantic requirements, productions in P are required to be extended by the corresponding P' productions in *Step 3*. Both DPO- and DPB-subspan relations are reasonable in *Step 4*. The first would, in fact, guarantee the stronger reflection property based on transformations only.

The behavior-reflecting morphism appearing in cell 4 of Table 4.1 is introduced in the following definition.

Definition 4.2.10 (behavior-reflecting morphism). Given graph transformation systems $\mathcal{G} = \langle TG, P, \pi \rangle$ and $\mathcal{G}' = \langle TG', P', \pi' \rangle$, a *behavior-reflecting morphism*

$$\begin{array}{ccc} f^{>}_{TG}(s) \to s' & \begin{array}{c}\not\Rightarrow\\ \not\Leftarrow\end{array} & s \to f^{<}_{TG}(s') \\ \Uparrow\not\| \| \not\Downarrow & & \Uparrow\not\| \| \not\Downarrow \\ f^{>}_{TG}(s) \leftarrow s' & \begin{array}{c}\Rightarrow\\ \not\Leftarrow\end{array} & s \leftarrow f^{<}_{TG}(s') \end{array}$$

Figure 4.4: Preservation of production span relations.

$f^{refl} = \langle f_{TG}, f_P \rangle$ is given by a type graph morphism $f_{TG} : TG \to TG'$ and a mapping of production names $f_P : P' \to P$ such that for each $p' \in P'$ there exists $e : \pi(f_P(p')) \to f^{<}_{TG}(\pi'(p')) \in \mathbf{Sp}^{DPB}_{TG}$ forming a faithful transition. △

The proof of the following is obvious.

Fact 4.2.11. Behavior-reflecting morphisms $f^{refl} : \mathcal{G} \to \mathcal{G}'$ satisfy the requirements of Def. 4.2.9. □

A variant of the behavior-reflecting morphism specified by cell 3 has been used in [76]. The difference from the one of cell 4 is the mapping of production names which goes in the same direction as the mapping of types, i.e. from P to P'. Using DPB-subspan relation and assuming in each GTS an empty ϵ-production, each step in $\mathcal{G}'$ using a production without a corresponding production in $\mathcal{G}$ is associated with an ϵ-transition. In this way, the behavior is indeed reflected by $\mathcal{G}$. If we consider, instead, DPO-subspan relation, we obtain a partial reflection of the target transformations by the source ones.

The GTS morphism specified in cell 3 will be formally defined in the next subsection, where we examine a relation of the substitution morphism to the proposals considered above.

It turns that none of the other alternatives in Table 4.1 preserves or reflects behavior. Variants 1 and 2 are not behavior-preserving, because the production span relation allows production spans in the target system to be larger than in the source. Hence, additional preconditions may be introduced that makes productions in $\mathcal{G}'$ applicable in less situations.

Similarly, variants 7 and 8 are inadequate for the behavior reflection, since production span relations are not in general preserved by the retyping (see Remark 4.2.5).

The preservation properties for the production span relations are summarized in Fig. 4.4.

4.2.4 Locus of the Substitution Morphism

The definition of the substitution morphism (cf. Def. 3.2.5) consists of two parts. The first part ensures reflection of behavior enacted by the target system in the source one. Therefore, behavior-reflecting morphisms are appropriate here, but only for those productions of the target system which are associated to productions of the source (cf. cell 3 of Table 4.1). The second part guarantees preservation of applicability for the productions of the source system in the target. This is similar to behavior-preserving morphisms (cf. cell 5 in Table 4.1) except that the application condition is considered instead of actual productions.

At this point, it is necessary to stress a very important role of application conditions providing a refinement of the production structure. The combination of requirements on behavior-reflection and behavior-preservation in the substitution morphism makes unconditional productions of the related systems essentially identical. This happens, because behavior-reflection is ensured by the production span relation being, in fact, inverse the one in behavior-preservation. Thus, the application conditions allows to establish a more flexible relation between the independently developed specifications.

In order to demonstrate that the substitution morphism represents indeed an extended variant of the behavior-reflecting morphism, the latter specified by cell 3 of Table 4.1 is formally presented in the definition below. Since our general intention is to manipulate with GTS morphisms that check not only behavioral but also structural compatibility of compared systems, the definition contains a parameterized version of this morphism.

Definition 4.2.12 (parameterized behavior-reflecting morphism).
Given parameterized graph transformation systems $\mathcal{GP} = \langle TG, P, PP \rangle$ and $\mathcal{GP}' = \langle TG', P', PP' \rangle$, a *parameterized behavior-reflecting morphism* $f^{refl+} = \langle f_{TG}, f_P \rangle$ is a signature morphism given by a type graph morphism $f_{TG} : TG \to TG'$ and a mapping of production names $f_P : P \to P'$, where for all $p \in P$ and $p' = f_P(p) \in P'$ the spans of productions $pp = p(x, y) : s \in PP$ and $pp' = p'(x', y') : s' \in PP'$ satisfy Requirement 1 of Def. 3.2.5, i.e. for the spans s and s' there exists a TG-typed span morphism $e : s \to f^{<}_{TG}(s') \in \mathbf{Sp}^{DPB}_{TG}$ forming a faithful transition. △

Comparing this definition with Def. 3.2.7, it is not difficult to see that the (parameterized) substitution morphism is a kind of (parameterized) behavior-reflecting

morphism accompanied with the requirements on preservation of applicability.

Further we discuss the application of GTS morphisms as intra- and inter-connectors of modules.

4.3 Application of GTS Morphisms

In this section we revise the initial proposals for module *intra-connectors* appearing in Subsection 4.2.2 and demonstrate application of the discussed morphisms for intra- and inter-connectors of modular specifications. To illustrate the external use of parameterized substitution morphisms, the example scenario is extended with a module implementing the algorithm for distributed deadlock detection (DDD). The DDD module is considered in the following subsection.

4.3.1 Extended Example Scenario

The algorithm for distributed deadlock detection specified by the module MOD' is depicted in the lower part of Fig. 4.5. The upper part of this figure shows the module MOD modeling the algorithm for mutual exclusion discussed in Example 4.2.1.

A deadlock detection service offered by MOD' at the export interface EXP' is asked for by the module MOD at the import interface IMP (cf. IMP and EXP' in Fig. 4.5). At the same time, MOD' lacks for deadlock resolution capabilities which are provided, in turn, by the module MOD at the export interface EXP (cf. IMP' and EXP in Fig. 4.5). While such a relation between module interfaces, called *cyclic import*, might be problematic for practical realization, it allows to properly illustrate different kinds of module connectors.

Example 4.3.1 (DDD). The main purpose of MOD' is to observe processes and resources and detect a deadlock if asked to do so. In a graph representing a system state, a deadlock appears as a cycle of *request* and *held_by* edges, where one process requests a resource held by another process and simultaneously holds a resource requested by it. The distributed deadlock detection uses *blocked* messages, represented by edges with a black flag, in order to detect such cyclic dependencies.

A deadlock detection is initiated by a process p waiting for a resource r. The process uses the production *dead*? to send a *blocked*-message to r. This feature is offered by MOD' at EXP' for external use, e.g., by MOD. If the resource is held by another process which itself is waiting for a resource, the message is passed on using *waiting*. If this is not the case, which is checked by a negative application condition,

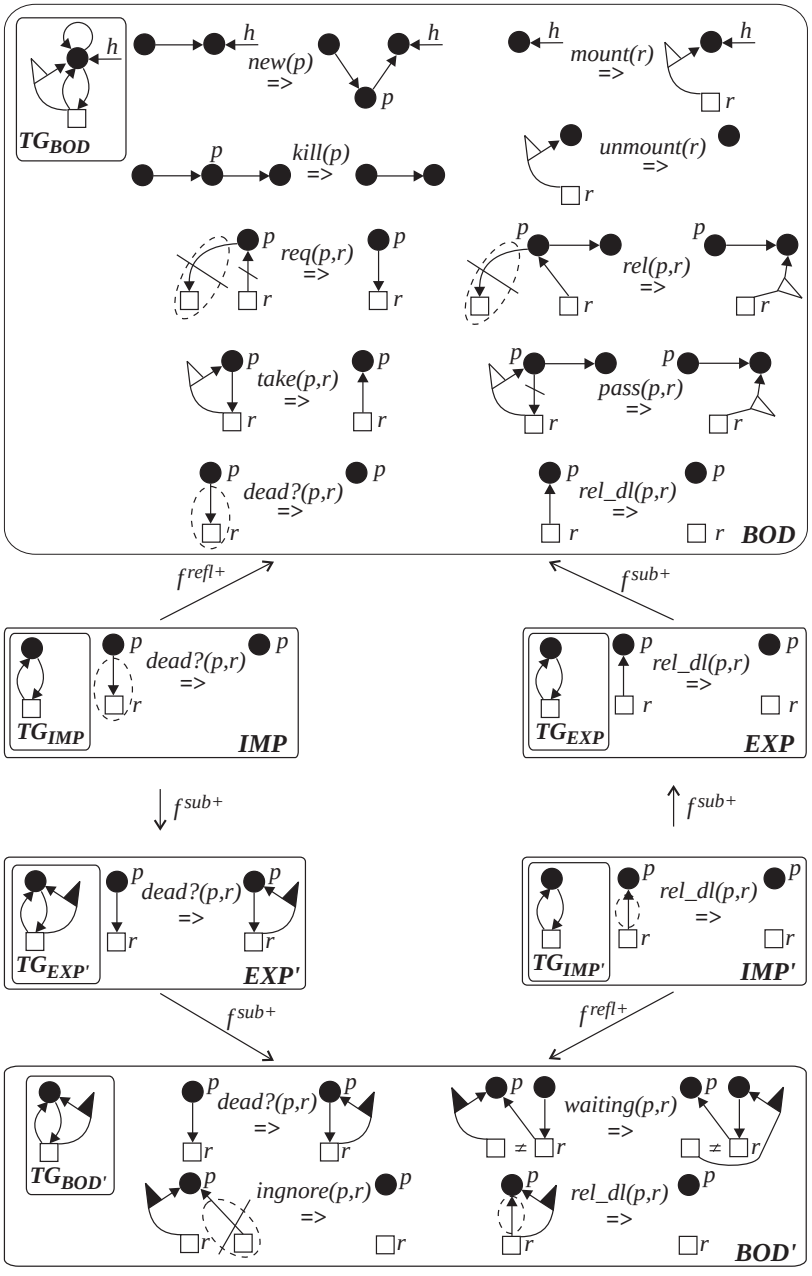


Figure 4.5: Modules specifying algorithms for mutual exclusion (upper) and distributed deadlock detection (lower).

the message is deleted by the production *ignore*. Due to the mutual exclusion, each resource is held by only one process. Hence, if the message arrives at a resource which is held by the original sender, a cycle has been detected.

Since MOD' is only destined for deadlock detection, deadlock resolution is described only abstractly by the production *rel_dl*, which deletes the *blocked*-message, but does not decide how the deadlock is actually resolved. This production in the import interface IMP' needs to be replaced by the production of MOD with the same name. The positive application condition of *rel_dl* restricts applicability of the production to the system states where a resource is held by the process, i.e. to the situations being meaningful for the deadlock resolution. △

4.3.2 Intra-connectors of Modules

We proceed with the discussion on intra-connectors relating the import interface and body of a module. In Subsection 4.2.2 (parameterized) behavior-reflecting morphisms were proposed for this purpose. In order to ensure a *consistency* between the import interfaces and the internal implementations specified in the bodies of the modules in Fig. 4.5, the constituents of MOD and MOD' shell be verified against the requirements of Def. 4.2.12.

First of all, we establish a signature morphism consisting of a type graph morphism f_{TG} and a mapping f_P which relates production names enriched with parameter type declarations. In the module MOD and MOD' the type graphs of the import interfaces are subgraphs of the ones containing in the bodies. So, the type graph morphisms in both cases are given by inclusions. The productions in the source and target systems are identified by their names, i.e. *dead*? for IMP and BOD, and *rel_dl* for IMP' and BOD'. Since the compared production names have the same parameter types, the systems IMP and BOD, and IMP' and BOD' are structurally compatible.

The required relation (cf. Def. 4.2.12) between the spans of the productions *dead*? in IMP and BOD follows from the identity of the compared spans. The relation between the spans of the productions *rel_dl* in IMP' and BOD' holds, because the span of the BOD' production becomes identical to the one of the IMP' production after the backward retyping. Hence, the specifications at IMP and IMP' are integrally compatible with BOD and BOD', respectively, that guarantees consistency of the internal and external descriptions.

In contrast with the import-body connector, the requirements towards the export-

body connector shall be strengthened. Behavior-preservation guarantees that applicability of productions in the export interface implies applicability of the body productions. However, this property would be satisfied even for empty body productions. In fact, we also require that effect encoded in the body productions is at least the one promised by the productions in the export interface. Hence, we "upgrade" the proposed candidate to the export-body connector and define it by the parameterized substitution morphism. Next we shall demonstrate that the relations between exports and bodies of the modules in Fig 4.5 are indeed parameterized substitution morphism.

Structural compatibility of the specifications IMP and BOD is guaranteed by a signature morphism composed of the type graph morphisms in the form of inclusion and exact compatibility of parameter type declarations of the productions *rel_dl*. The same kind of signature morphism underlies structural compatibility of the IMP' and BOD'. Then, following Def 3.2.7, one should check preservation of applicability from the export interface to the body and reflection of effects of the body productions by the ones of the export interface. Due to the fact that the productions *rel_dl* of EXP and *dead*? of EXP' are identical to the body productions with the same names in the modules MOD and MOD', the required properties obviously hold.

4.3.3 Inter-connectors of Modules

In Chapter 3 we have introduced the parameterized substitution morphism and justified its use for inter-connecting of software components. In this subsection we illustrate application of this morphism in the context of example scenario.

Let us first discuss the relation between the interfaces IMP and EXP' shown in Fig 4.5. The signature morphism between the parties is given by the type graph morphism f_{TG} from TG_{IMP} to $TG_{EXP'}$, which is actually an inclusion, and by the mapping f_P between the sets of production names. This mapping is unique, because both interfaces contain only one production *dead*? which name is equipped with the parameter type declarations being the same for IMP and EXP'.

After ensuring structural compatibility, we proceed with behavioral compatibility. First of all, one should check the preservation of applicability from IMP to EXP' (cf. Def 3.2.7). Each of the productions has one positive application constraint being the union of the left-hand side and the dotted part in the IMP production, and coinciding with the left-hand side in the EXP' production. The application conditions in the two productions are the same, because the forward retyping of the EXP' production

does not introduce any changes. Thus, applicability is preserved.

The second step is reflection of effects. While the backward retyping of the EXP' production gets rid of the *blocked*-message, it is still bigger in context and effect than the IMP production. This is allowed by the DPB-subspan relation which can be established between the spans of the productions.

Combining the two results, we can conclude that the import interface IMP is associated with the export interface EXP' by the parameterized substitution morphism. This, in turn, implies integral compatibility of the considered specifications and guarantees that the service provided by the module MOD' satisfies the requirements of the module MOD.

Now we discuss a relation between the interfaces IMP' and EXP. It is not difficult to see that these specifications are structurally compatible.

Since the type graphs $TG_{IMP'}$ and TG_{EXP} of the two systems are the same, the retyping does not change the productions. The positive application constraints of the productions *rel_dl* coincide that means preservation of applicability from IMP' to EXP. Reflection of effects is ensured by the DPO-subspan relation (being also a DPB-subspan relation) between the two productions in spite of the bigger context of the EXP' production which additionally contains the *held_by* edge.

Hence, the import interface IMP' and the export interface EXP are also connected by the parameterized substitution morphism ensuring integral compatibility of the parties.

4.3.4 A Formal Account of Component Model

Having considered a number of examples, we are ready to define formally a graph transformation module specifying the component model.

Definition 4.3.2 (graph transformation module). Given parameterized conditional graph transformation systems IMP, BOD, EXP, a parameterized behavior-reflecting morphism $imp : IMP \to BOD$, and a parameterized substitution morphism $exp : EXP \to BOD$, a *graph transformation module* is a system $MOD = (IMP \xrightarrow{imp} BOD \xleftarrow{exp} EXP)$, where imp and exp are called intra-connectors.

Given graph transformation modules $MOD = (IMP \xrightarrow{imp} BOD \xleftarrow{exp} EXP)$ and $MOD' = (IMP' \xrightarrow{imp'} BOD' \xleftarrow{exp'} EXP')$, if there exists a parameterized substitution morphism $sub+ : IMP \to EXP'$ then we say that MOD is *inter-connected* to MOD' by $sub+$ (cf. Fig. 4.1). △

While the definition introduced above is similar to the one presented in [69], there are two significant differences. First of all, the specified modules define not only behavioral but also structural characteristics of the portrayed components, i.e. their operation declarations. Secondly, behavioral annotations of operations may be equipped with strict (DPO) as well with loose (DPB) semantics—this is important for operations in the import interface.

4.4 Summary

This chapter provides two main contributions: a systematic presentation of morphisms of graph transformation systems along with a recipe of how to define new variants, if needed, in a generic framework; and a model specifying coupled components in the form of graph transformation modules.

On one hand, the first result provides a solution for the consistency problem, where relations between import and export interfaces, and internal specifications of systems are precisely defined by the parameterized behavior-reflecting and substitution morphisms, respectively. On the other hand, the constructed framework represents a reaction to the multitude of proposals and variants for GTS morphisms that exist in the literature.

A model has been motivated by the need to check correctness of integrating components. Graph transformation modules employed for the model consist of tree parameterized conditional GTS specifying the import, export and body along with intra-connector morphisms introduced in the context of the generic framework. Integration is modeled by the inter-connector that relates the import (required) interface and the export (provided) interface of modules.

Chapter 5

Related Work

A revision of related work given in this chapter serves two purposes. First of all, it allows to justify the approach established in the thesis and demonstrate its originality with respect to current proposals in the academic community and in industry. Secondly, it helps to reveal possible extensions and improvements of our work which are collected in Section 6.3 of the next chapter.

Available techniques for constructing interface specifications and software environments for matching these specifications are considered in Sections 5.1 and 5.2, respectively. Section 5.3 reviews component models in the form of modular specifications that have been proposed in the algebraic specification and graph transformation domains. In Section 5.4, we conclude with a summary highlighting uniqueness of ideas developed in the thesis.

5.1 Semantic-driven Specifications and Matching

In this section, we start with works originating in the Semantic Web domain, where we cover general approaches (Subsection 5.1.1) and the ones focusing on travel business services (Subsection 5.1.2). Then we proceed with an approach modeling semantics of Web services by means of graph transformation rules (Subsection 5.1.3). Finally, we overview specification techniques developed in the area of Component-Based Software Engineering (CBSE) allowing to retrieve reusable software components (Subsection 5.1.4).

5.1.1 Semantic Web Services

In general, specification of properties and capabilities of Web services amounts to definition of basic concepts of a service application domain and their relationship. The starting point here would be to provide a markup language (or knowledge representation language) for this purpose. Over the last several years, a number of Semantic Web markup languages have been proposed. These include the Resource Description Framework (RDF) [99], RDF Schema [20], DAML+OIL [33], and, most recently, OWL [39] and WSMO [150].

OWL-based Web Service Ontology

The Web Ontology Language OWL designed by the W3C Web Ontology Working Group is a semantic markup language for publishing and sharing ontologies on the World Wide Web. Derived from DAML+OIL, OWL is a description logic-based language for taxonomic information. It is built on top of XML and RDF(S) and characterized by a well-defined semantics and a wide range of constructs, including *classes*, *subclasses*, and *properties* with *domains* and *ranges*, for describing Web entities. Also, further restrictions on membership in classes as well as restrictions on domains and ranges, such as cardinality restrictions, can be expressed in the language.

OWL-S [31] (originally called DAML-S) is an OWL ontology for Web services developed by a coalition of researchers and industry partners, such as BBN Technologies, Carnegie-Mellon University, De Montfort University, Nokia, Stanford University, SRI International, etc. The OWL-S ontology determines a set of Web service specific primitives.

The upper ontology of OWL-S consists of a *ServiceProfile* for describing service advertisements, a *ServiceModel* for describing the actual program that realizes the service, and *ServiceGrounding* for describing the transport-level messaging information associated with execution of the program. *ServiceGrounding* is quite similar to the Web Service Description Language (WSDL) [28].

We concentrate on the OWL-S profile which defines a service as a function of three basic types of information: on organization that offers the service, on the function the service computes, and on a host of features that specify characteristics of the service. An essential component of the profile is the functional description representing two aspects of the service functionality. The first aspect is an information transformation in the form of input and output parameters of service operations. The second one is given by state changes produced by the execution of the service in the form of

precondition and effect constraints.

The current version of OWL-S specification does not mandate any language for expressing constraints, leaving the choice of the language to a modeler. However, the OWL-S specification refers to two main candidates for this role. They are the Semantic Web Rules Language (SWRL) [87], under development at W3C, and DRS described in [106]. Both languages specify constraints by logic formulas that bound input and output parameters of service operations, i.e. they incorporate purely syntactic restrictions on parameters. Thus, the problem of semantic annotaions remains open in OWL-S as well as earlier in WSDL.

Approaches Related to DAML-S/OWL-S

In the following, we examine a number of approaches that underly and extend DAML-S/OWL-S.

McIlraith et al. in [107] propose a markup of Web services in the DAML family of Semantic Web languages and employ agent technologies for automated Web service discovery, execution, composition, and interoperation. The Web service markup, representing, in fact, a core set of DAML-S, consists of two basic kinds of ontologies: domain-independent and domain specific. A domain-independent ontology defines the general class *Service* having two subclasses. The first subclass, called *PrimitiveService*, represents stand-alone Web-executable computer programs that do not assume ongoing interactions between a user and a service. The second subclass, called *ComplexService*, contains complex services consisting of multiple services and supports aggregate interaction scenarios.

The domain-independent Web service ontologies are augmented by domain-specific ontologies which extend them with concepts that are specific for individual services. Service semantics is defined via an association with each service a set of *Parameters* reflected in the domain-specific ontology. For example, one may define a travel business ontology containing the class BookTicket with the subclasses BookTrainTicket and BookAirlineTicket. The latter may be equipped with the subclasses BookUALTicket and BookLufthansaTicket which provide semantic annotations for two booking services.

The authors of [107] show how the domain-specific ontologies facilitate the discovery process which is carried out by automatic agents established in the programming language (Con)Golog. However, service description via a single ontological element ignores operational nature of Web services. Operation of a Web service imposes cer-

tain alterations in the real world, such as transitions in state spaces of requestor and provider. A service description should reflect these transitions in semantic annotations.

Furthermore, ontologies of the kind described above usually contain very specific, concrete product information, such as BookUALTicket or BookLufthansaTicket. Extending generic ontologies with elements characterizing specific service instances gives rise to another question—how to construct and maintain such ontologies. With the emergence of a new service on the market (or modification of an old one), the ontology has to be augmented by elements that are relevant to the newcomer. At the same time, the construction of ontology is a quite complicated and long-continuing process. It requires an across-the-board agreement between many companies which may use by now previously standardized ontologies. Rule-based specifications, on the contrary, represent structures built over the terms of ontology (or data model), and that makes such specifications much more flexible.

A framework for Web service semantics is constructed in the work of Dogac et al. in [41]. They extend the DAML-S upper ontology to describe services with complementary functionality, discover them according to the properties of products, and relate service descriptions to electronic catalogs (e.g., the Common Business Library (CBL) catalog definition or RosettaNet Technical Dictionary).

The proposed extension represents an ontology consisting of services and products. The top level element in the ontology is the class *Product* that has *PhysicalProduct* and *VirtualProduct* as subclasses. The class *Service* being a subclass of *VirtualProduct* defines a generic service type. Names of different generic services, such as Car_Rental_Service or Travel_Reservation_Service, are used to locate specific services representing particular implementations of the generic ones. The discussed approach allows to integrate the introduced extension with UDDI registries—it is very important for practical application. However, the service semantics in [41] is defined analogously to [107] and this causes the same problems.

Paolucci et al. in [119] design a matching algorithm between service advertisements and service requests, where DAML-S is adopted for service descriptions. The algorithm supports multiple degrees of similarities between specifications. It compares inputs and outputs of required and provided services and performs inferences on the subsumption hierarchy established in DAML-S. The degree of match, e.g., exact, plug-in, or subsumes, is determined by the distance between concepts in the

taxonomy tree.

On one hand, this work extends keyword-based searching capabilities of the UDDI registry. On the other hand, service specifications are compared merely by operation signatures. So, the authors establish, in fact, an advanced structural compatibility test. It is advanced in the sense that it allows to relate types that are at different levels in the type hierarchy (instead of lexical identity between types).

METEOR-S Approach

The work discussed below represents a part of the METEOR-S project [117] of the LSDIS laboratory at the University of Georgia. It addresses the entire life cycle of Semantic Web process, involving semantic annotations, discovery, composition and orchestration of Web services. The initial version of the approach presented in [135] has been currently extended and revised in the Web Service Semantics technical notes on the language WSDL-S. The technical notes are published by the LSDIS laboratory and IBM in [1].

The main idea of the approach is to map declarations of WSDL documents, such as operations and their parameters, to elements of appropriate ontologies. This allows users to search for operations based on ontological concepts with well-defined meaning. For example, the operations buyTicket and cancelTicket of a service in the travelling business domain are related to the ontological concepts TicketBooking and TicketCancellation, and the input TravelDetails and the output Confirmation of a WSDL specification are mapped to the ontological concepts TicketInformation and ConfirmationMessage.

Each operation may have a number of preconditions and effects. As usual, preconditions must be true prior to the operation execution, while the effects determine changes in the world after the operation execution. The operation buyTicket, for example, may have the precondition and effect represented by the ontological concepts ValidCreditCard and CardCharged-TicketBooked-ReadyForPickUp. Preconditions and effects can be added as children of the WSDL element operation.

The authors also show how to embed the semantic information into UDDI registry. In the process of Web service publication, different kinds of ontologies are stored using the UDDI structure *tModels* (and *CategoryBags*). *tModels* are metadata constructs which are used to characterize and categorize businesses and their services. The registry is equipped with four kinds of *tModels*. The first *tModel* represents an ontology determining semantic annotations for general functionality of operations.

Ontologies defining semantics of inputs and outputs are contained in the second and third *tModels*, respectively. The general semantic annotations and those for inputs and outputs of operations are combined in the context of the fourth *tModel*.

The first advantage of the proposed technique is the possibility to choose a notation for semantic annotations, such as UML or OWL(-S). This is not the case in DAML-S or OWL-S as they are targeted at the corresponding ontology languages. The second advantage is in the idea to express operation semantics by means of preconditions and effects. However, static ontological concepts employed for annotations of operation signatures are not flexible enough to be reused for such behavioral constraints as preconditions and effects.

Web Services Modeling Ontology (WSMO)

The Web Services Modeling Ontology (WSMO) [57, 150] is a Semantic Web services initiative led by the Digital Enterprise Research Institute (DERI) at the University of Innsbruck and by several EU projects. Its aim is to provide a *conceptual model* and a *formal semantic markup language* describing all relevant aspects of Web services.

The conceptual model of WSMO is based on the Web Service Modeling Framework (WSMF) presented by Fensel and Bussler in [60]. Four main elements that WSMO inherits from WSMF are: ontologies (domain specific terminologies for describing other elements), Web services (specifications of provided functionality), goals (specifications of functionality that a Web service should provide from the user perspective), and mediators (mechanisms to link possibly heterogeneous components built on WSMF/WSMO descriptions).

The semantic markup is given by the Web Service Modeling Language (WSML) [37, 149] that allows to write semantic annotations of Web services according to the conceptual model. Provided and required services are described in WSML by means of service capabilities. The capabilities are defined by non-functional properties, imported ontologies, preconditions, postconditions, assumptions, and effects. A *postcondition* specifies what a Web service offers to a client, when the *precondition* is met in the information space. An *effect* describes how the execution of the Web service changes the world, given that the *assumption* over the world state is met before execution. Preconditions, assumptions, postconditions and effects are expressed through a set of axioms in F-logic [104].

Applying logic to express behavioral specifications has two negative consequences. First of all, logic-based constraints bounding operation parameters do not explain

what the operation actually does, i.e. they represent syntactic annotations instead of semantic ones. Secondly, logic-based approaches have long proved to be hardly usable in real software development process.

5.1.2 Approaches in the Travelling Business Domain

We have illustrated our ideas by a Web service scenario for booking flight tickets. Therefore, we discuss a couple of projects concentrating on travel information services.

HARMONISE Project

The mission of the Harmonise project [61, 91] is to solve the interoperability problem of data formats used by different players in the travel and tourism industry. The authors propose an ontology-mediated process based on a framework for data integration called *Harmonise Platform* (HP).

HP consists of three basic elements. An *Interoperability Minimum Harmonisation Ontology* (IMHO) is a tourism ontology which is used to model and store the basic concepts representing the content of information exchanges in tourism transactions. A *Harmonise Interchange Representation* (HIR) is an interchange format to specify the instance data applied for interoperable tourism transactions. A set of *mapping rules* is aimed at the translation of the transaction data from internal or proprietary formats to HIR, and vice versa.

Different tourism companies employ HP to keep their own data format. They exchange information based on IMHO through *Harmonisation Gateways* that play a role of mediators. Each company willing to join the marketplace supported by HP has acquire such gateways.

A realization of gateway-based interactions consists of customization and cooperation phases. In the former phase, a user constructs a set of mappings between its internal data and the HIR format. It then allows to interact with other users in the communication phase. Currently, the authors of the project are working on IMHO and on formal specification of mapping rules.

The Harmonise project manifests that a common external data format is an ultimate necessity for the dynamic integration between heterogeneous travel services. This claim goes along our guidelines as well as the results of Harmonise—they can be used to determine a relation between external and standard-based service interface specifications, and to facilitate automatic construction of these specifications.

SATINE Project

The SATINE project [123, 42] realizes a semantic-based interoperability framework for the tourism industry. The framework provides mechanisms for publishing, discovering, composing and invoking Web services through their semantics in peer-to-peer networks.

The basic idea behind SATINE is to enable service providers to wrap their existing applications as Web services, annotate them semantically, and advertise these services in SATINE network. SATINE network relies on a peer-to-peer infrastructure and deploys semantic querying and routing mechanisms to discover appropriate services. Service descriptions are based on OWL-S and may be stored either directly in peer nodes or in semantically enriched UDDI and ebXML [44] service registries attached to peers.

In order to create appropriate semantic annotations, the authors exploit domain knowledge exposed by the Open Travel Alliance [2]. The semantic annotations are given by *Service Functionality* and *Service Message* ontologies that are constructed from the OTA request/response message schemas arranged into a class hierarchy. The *Service Functionality* ontology describes the general meaning of a Web service employing a set of messages semantics of which is specified in the *Service Message* ontology. For example, a Web service instance THY_Ucak_Rezervasyonu can be classified with the AirBookingService node of the *Service Functionality* ontology to indicate that it is an air reservation service. A fragment of the *Service Message* ontology for the message AirBookRQ looks like the right-hand segment of Fig. 2.4.

We believe the SATINE project implements a number of interesting ideas, such as the *Service Functionality* ontology which can be used to define general characteristics of travel services. However, a limited number of message declarations in the OTA standard causes serious problems for *Service Message* ontologies application. It is not clear how to construct these ontologies for user-defined messages that are obviously out of the standard. We avoid this problem—our approach allows to describe operations beyond those predefined in the OTA standard.

5.1.3 Graph Transformation for Service Specifications

The ideas introduced by Hausmann et al. in [75] follow mainly the guidelines quite similar to those of our approach. Graph transformation rules in [75] are defined over a domain ontology and are used to represent contract-based behavioral descriptions of service operations. However, there is a number of important technical differences

that distinguish it from our work.

The strengths of [75] lie in a methodology that develops operation contracts in the context of a standard model-based development process. We would also like to note that a matching procedure in [75] is implemented in a prototypical tool chain (see Subsection 5.2.2). This implementation represents a very effective and useful instrument. In the following, we highlight the aspects that distinguish the approach of Hausmann et al. from ours.

Firstly, the introduced in [75] matching procedure, defined in a set-theoretic style, does not enjoy a formal operational semantics. There are no precise semantic requirements on compatibility of service interface specifications, and as a consequence, correctness of the proposed technique has been justified only by examples. Moreover, the lack of application conditions limits expressiveness of the contract language in [75].

Secondly, the approach of Hausmann et al. does not take structural information on service operations into consideration. Behavioral compatibility provides only necessary but not sufficient conditions for successful interaction of required and provided systems.

Thirdly, the discussed approach does not support any retyping procedure. It happens due to the assumption that parties establish service specifications over the same ontology. However, as illustrated by the sample interface specifications in Fig. 2.7 and Fig. 2.9, even compliance with the same industry standard does not guarantee the identity between ontologies (domain data models) employed by parties.

5.1.4 Component Specification Matching for Software Reuse

A problem of discovering a component or a service satisfying specific requirements is not new. A huge amount of work has been done in the area of Component-Based Software Engineering (CBSE). Its aim is to increase reliability and maintainability of software through reuse. Central role here plays development of techniques for creating component descriptions and their matching. These techniques differ in constituents involved in a matching procedure (structural specifications or signatures, and behavioral specifications), the way these constituents are specified (logic formulas or algebraic specification languages), and granularity of compared components (single operations or functions, and collection of operations or modules).

Below we treat only a few works that somehow relate to the ideas presented in the thesis. A detailed analysis of the whole spectrum of approaches in the CBSE domain can be found in [154].

One of the most elaborate approaches is created by Zaremski and Wing in [155] and [156]. They consider two kinds of descriptions, signatures and behavioral specifications, for two kinds of components, functions and modules. Signature specification and matching is presented in [155]. A signature of a function is simply its type, and a signature of a module is a multi-set of user-defined types and a multi-set of function signatures. A wide range of matchings, such as exact, generalized, or specialized ones, are introduced for functions and modules. The structural compatibility relation constructed in our work was inspired by [155].

Behavioral specification matching is discussed in [156], where a function behavior is defined by pre- and post-conditions written as predicates in first-order logic. Behavioral compatibility of two functions is modeled by equivalence or implication between the pre- and post-conditions depending on the chosen kind of match. The developed matching procedure is extended to collections of functions in order to check behavioral compatibility between modules. The approach uses the programming language ML, and Larch/ML (a Larch interface language for ML) is employed to specify ML functions and modules.

Jeng and Cheng in [96] define two different matches for component descriptions in order-sorted predicate logic (OSPL). Components are modules that consist of inherit clauses and a set of function specifications. A function specification is defined by a pre-/post-condition pair terms of which are expressed in OSPL. While the first kind of match, called relaxed exact match, is primarily syntactic, the second match, called logical, is based on the subsumption relation between clauses.

Chen et al. in [26] introduce a framework for both signature and behavioral specification matching, where components are specified in the algebraic specification language ASL. Components are portrayed by modules consisting of a set of sorts, a set of operations on the sorts, and a set of axioms for the operations. Their compatibility is specified as an *implements* relation: a source component is an implementation of the target one if the signatures match, and a class of models in which the axioms of the source are satisfied is a subclass of the one for the target.

There are two main differences that distinguish our work from those mentioned above. The first difference lies in the operational interpretation of graph transformation rules. It allows to reflect behavioral aspect of service operations adequately. Second, we propose a visual model-based approach—it can be easily integrated into standard model-driven techniques of software development in contrast to logic-based or algebraic specification approaches.

5.2 Automation of a Matching Procedure

In the following, we discuss implementations of the approaches reviewed in the previous section. We start with software systems providing tool support for the Semantic Web techniques, then consider a prototypical tool chain that checks behavioral compatibility of service specifications based on graph transformation, and, finally, analyze systems developed in the context of software reuse approaches.

5.2.1 Software Environments for Semantic Web Services

Tool Support for DAML-S/OWL-S

There is a multitude of tools based on DAML-S/OWL-S concepts. All of them can be divided into two main groups. The first group consists of environments providing capabilities for creating and maintaining OWL-S service descriptions (e.g., Protégé-OWL Ontology Editor [100], OWL-S Editor [132], or ASSAM Web Service Annotator [85]). The second group of tools aims at matching service specifications representing these ontological descriptions.

Let us take a look at a typical tool from the second group. OWL-S Matcher (OWLSM) [93] implements an algorithm enabling different degrees of matching, e.g., *subsumes* or *exact*, between inputs and outputs of services which are annotated by elements of OWL-S ontologies. Similarly to our approach, the matching procedure is based on the contravariant relation between input and output types of the required and provided services. In addition, the reasoning process covers elements of a general service description, namely, a service category. The algorithm consists of two phases. First of all, it separately tests compatibility between inputs, outputs, and categories of compared services. Then, the results obtained in the first phase are put together in the second phase.

Contrary to mechanisms that return only success or fail, outputs of OWLSM are ranked according to the matching degrees. It makes a selection of a service more flexible. However, the proposed procedure operates only structural descriptions and ignores behavioral characteristics of services. OWLSM could possible be a good candidate for a structural compatibility checker in our approach. For this purpose, it should only be augmented by a module that performs behavioral compatibility test.

Web Service Discovery Infrastructure (MWSDI) for METEOR-S

The Web Service Discovery Infrastructure (MWSDI) [144] (see also [62]) was developed as a part of the METEOR-S project. It represents a scalable infrastructure for semantic publication and discovery of Web services. One of the central components in MWSDI is a discovery engine based on a three-step algorithm.

In the first step, the algorithm matches ontological concepts that describe entire operations appearing in WSDL specifications of required and provided services. In the second step, the resulting set from the first phase is ranked on the basis of similarity between input and output concepts of compared operations. This step is analogous to the structural compatibility check from our approach. The optional third phase, quite similar to our behavioral compatibility check, involves ranking based on similarity between precondition and effect concepts of required and provided operations.

Until now, only the first and the second steps of the algorithm have been implemented in MWSDI. This tool can be noted for the automatic generation of Web service semantic annotations, but these annotations are limited to elements of ontologies specifying inputs and outputs of compared services. Therefore, the "resolution" of the discovery process in MWSDI does not exceeds the one of OWLSM.

Web Services Execution Environment (WSMX) for WSMO

The Web Services Execution Environment (WSMX) [55, 73] is a reference implementation of WSMO and its underlying formal language WSML. It relies on a set of loosely-coupled components for discovery, selection, and invocation of Web services.

An interaction of a user with the environment starts with a translation of concrete goal (required) specification. It can be expressed in natural language or in some specific formalism and has to be translated into the internal format of WSMX. Then, the translated goal is matched against formal descriptions of registered Web services. In case of success, it returns one or more service descriptions. The most appropriate service selected by the user is invoked further, and the invocation result is returned to the user.

The ultimate vision of WSMX developers is to automate *keyword-based* discovery, *lightweight semantic* discovery, and *heavyweight semantic* discovery. In the first case, keywords from a goal are matched against keywords from services descriptions. In the second case, controlled vocabularies with universally recognized semantics are used in matching process. The heavyweight semantic discovery introduces a relation between inputs and outputs of required and provided service operations that are equipped

with axiomatic annotations in F-Logic.

According to [73], the current implementation of WSMX offers only the keyword-based discovery that is purely syntactic procedure. While the match can be done in different dimensions, e.g., based on non-functional property values of goal and services descriptions, it does not take into account neither structural no behavioral service characteristics. This means discovery processes in OWLSM and MWSDI are more powerful than the one offered by WSMX.

5.2.2 Tool Support of the Matching Procedure based on Graph Transformation

In the following, we take a look at a prototypical tool chain introduced by Hausmann et al. in [74]. As we have already mentioned, it appears to be an effective and useful instrument. Augmented by a structural compatibility checker, e.g., OWLSM or MWSDI, it could be successfully applied to automate our approach as well.

The Attributed Graph Grammar System (AGG) [15] is used in the tool chain as a visual editor. It allows to generate service descriptions and requests in the form of contract rules typed over a type graph. Here the type graph serves as an abstract representation of an ontology. In order to employ DAML+OIL as RDF-file format to specify an ontology of a specific domain and to use this ontology in AGG, the Java-based tool *Daml2Agg* is implemented. This tool translates a DAML+OIL ontology into a type graph in the file format of AGG. A translation of contract rules created in AGG back to DAML+OIL is realized by the second tool *Agg2Daml*.

Ontologies and contract rules in DAML+OIL are given in a form of RDF graphs. Therefore, a matching algorithm operates RDF graphs as well. It is based on the open source semantic Web toolkit Jena [24], more precisely, the RDQL (a query language for RDF) implementation of Jena. The toolkit allows to specify a graph pattern that is located in a graph to yield a set of matches. The algorithm constructs RDQL queries to check subgraph relations between corresponding parts (pre-/post-conditions and effects) of contract rules.

5.2.3 Implementation of Matchers for Software Reuse Approaches

Finally, we consider tools that implement approaches shown in Subsection 5.1.4. A general mission of such tools is to detect required components in software libraries by structural and behavioral compatibility tests.

Zaremski and Wing implement a matcher [155] that checks structural compatibility between functions of Standard ML (SML) via comparison of their signatures. The matcher is incorporated into the signature-based retrieval tool *Beagle*. Given a query and a set of relaxations, Beagle uses an appropriate match to compare each function in a library with a query and returns a set of functions structurally compatible to the query. Beagle's user interface is quite simple, it is just gnu-emacs and a mouse. A user defines a query and selects the desired relaxations before performing a search. An output is a list of functions types of which match the query along with a pathname to a file that contains the function.

The authors also provide a semi-automatic approach to checking behavioral compatibility. They use the Larch Prover (LP) [64] to prove a match of two behavioral descriptions in the form of Larch/ML specifications. LP is a theorem prover for a subset of multisorted first-order logic. Since LP is designed as a proof assistant, rather than an automatic theorem prover, some of the sample proofs in [154] require user assistance.

In the approach of Chen et al. component descriptions in ASL are translated into specifications written in the knowledge representation language *Telos* [111] for storage and other manipulations. Retrieval of software components is based on structural matching between signatures of goal (required) specifications and those of reusable (provided) components. A retrieval mechanism implemented in this approach is supported by the Database Management System *ConceptBase* [95] that is an object-oriented database environment. Abstract formal specifications of components are mapped into database objects, and the goal specifications—into a database query that can be processed by *ConceptBase*.

A prototype system for applying behavioral compatibility test in [96] is implemented by Jeng and Cheng in the Quintus ProWindows language[1], a dialect of Prolog that supports object-oriented organization of graphical elements. The system allows to construct a hierarchical library and retrieve reusable components from this library. Components are classified to form a two-tiered hierarchy. The lower-level hierarchy generated by a subsumption test algorithm represents generality relationships among components, where the parent component is more general then the child one. The higher-level hierarchy is generated by a clustering algorithm that is applied to the most general components of the lower-level hierarchy. The retrieval process is based on the subsumption test algorithm proposed in [25], traditional resolution meth-

[1] A product of Quintus Computer Systems, Inc.

ods invented by Robinson [127], and several-order-sorted unification methods, such as [11, 145].

Structural compatibility matchers, such as the one constructed by Zaremski and Wing, may provide a substantial assistance to automate our matching procedure along with the Semantic Web proposals. Behavioral compatibility matchers, instead, can not be directly used because of conceptual distance between graph transformation and formalisms underlying these matchers.

5.3 Algebraic and Graph Transformation Modules

The component model proposed in the previous chapter is inspired by modularization approaches developed in the algebraic specification and graph transformation domains. These approaches are discussed in the next two subsections and compared to modules introduced in our work.

5.3.1 Modularity Concepts in Algebraic Specification Languages

A common feature of modern algebraic specification languages (see, e.g., [7, 49, 70]) is *specification-building operations* for constructing large specifications in a structured fashion from smaller and simpler ones. Less usual in specification languages are features for describing modular structure of software components, where external descriptions, such as export and import interfaces, have to be clearly separated from specifications of internal implementations.

Some languages, including Common Algebraic Specification Language (CASL) [32], Common Object-oriented Language for Design (COLD) [58], *Extended ML* (EML) [130], Spectral [102], ACT TWO [50], provide the required mechanisms. A special kind of specifications, called *architectural specifications* [16], are introduced in CASL. An architectural specification consists of a list of *unit declarations*, indicating the required component modules with specifications for each of them, together with a *unit term* that describes the way in which these modules are to be combined. Such architectural specifications can be considered as a kind of import interface that extends standard CASL specifications.

COLD, a wide-spectrum language in the tradition of VDM and Z, introduces the notion of a *design component* that relates internal and external descriptions of a software system. In COLD, one writes $\mathsf{COMP}\ x : K := L$ to describe a component with the name x, interface specification K, and specification L acting as implementation

of K. Here L is considered as a refinement of K. This is similar to EML and Spectral, except that parameterized components are supported, and L is only required to refine K up to "behavioral equivalence".

Let us provide slightly more extended analysis of a *module* concept in the language ACT TWO. This proposal strongly influences on modularization approaches in graph transformation domain. ACT [29] is an approach to formal software development that includes a language ACT ONE [49] for writing parameterized specifications, referred as *types*, with conditional equational axioms, and an extension called ACT TWO [50]. ACT ONE provides only simple specification-building operators, like union, renaming, and instantiation, but no means for separation of external and internal descriptions. ACT TWO rectifies this disadvantage by means of modules that consist of four specifications (cf. Subsection 4.1.2):

- *Import interface*: This describes sorts and operations that the module requires to be supplied by other modules.
- *Export interface*: This describes sorts and operations that the module supplies for use by other modules.
- *Body*: This defines a construction of exported functionality in terms of the imported features. This construction may involve auxiliary operations that are not exported.
- *Parameter*: This describes parameters that are common to the entire module or modular system in which the module appears, e.g., the underlying character set.

These four specifications are written in ACT ONE extended by first-order axioms. In addition, there exist module-building operations (composition, union, instantiation and renaming) that are module-level analogues to specification-building operations.

One can note that the modules of ACT TWO have a structure (except for a parameter part) similar to the one of our module specifications. However, we can not perform more detailed analysis here. The employed formalisms, i.e. graph transformation and algebraic specifications, lie conceptually far from each other.

5.3.2 Graph Transformation Modules

Several modularization approaches, such as GSSPEC [46, 47], GRACE [84], DIEGO [139], PROGRES [147], TGTS [69], have been established in the graph trans-

formation community. Classification and comparison of modularity concepts of these approaches can be found in [82]. The differences of our modules from the existing proposals are summarized in the next section. Below we take detailed look at the TGTS proposal as we consider it quite closely related to our work.

Modules in TGTS employ an architecture similar to the one of algebraic specification modules from the language ACT TWO. The only difference is that import, export, and body of a module are represented by typed graph transformation systems. While the import-body intra-connector is defined by an inclusion morphism, the export-body intra-connector is specified by a *refinement relation*.

In general, a refinement maps an elementary operation of the more abstract specification (export interface) to a composite operation with the same effect in the more concrete specification (body). *Temporal* and *spatial* refinement relations for typed graph transformation systems have been defined in [67, 68]. In a spatial refinement, each rule is refined by an amalgamation (i.e., a parallel composition with sharing) of rules, while in a temporal refinement it is refined by a sequential composition. This approach introduces operations of module composition and union. This allows to built complex module systems keeping intra-connector relationships hidden. Refinement relations along with the operations of composition and union are highly desirable for our approach as well.

However, there are two main advantages of our modules over the TGTS ones. Firstly, structural information in the module specifications does not appear in TGTS modules. Its importance cannot be overestimated, while it provides a basis for a component model. Secondly, in contrast to the TGTS modules, the rules in our modules are equipped with application conditions. On one hand, application conditions significantly increase expressiveness of the contract language. On the other hand, conditional rules with clearly separated precondition and effect parts have a structure suitable for a contravariant matching procedure. This procedure, in turn, underlies the flexibility of the inter-connector defined by inclusion in the TGTS approach.

5.4 Summary

To sum up the discussion given above, we highlight the points that make our results on interface specification and matching unique. First and foremost, the proposed formalism for semantic annotations of service operations incorporates two main points that are crucial for specification of distributed applications [76]: an explicit descrip-

tion of system states and a formal support of state transformations that may occur in the system. In contrast to logic-based techniques (see, e.g., [37, 96, 156]) having a weak support of both aspects, and algebraic specification (AS) approaches (see, e.g., [26]) mainly supporting only the first aspect, the semantic annotations in the form of graph transformation rules provide truly behavioral (operational) specifications (cf. Table 5.1).

Our revision of Semantic Web proposals shows that a number of approaches, such as [107, 41, 117], simply ignore the operational character of services. Semantics of a service is described by references to solitary elements of ontologies. It is doubtful that one static element of an ontology can provide unambiguous and precise description of such a dynamic entity as a service. Moreover, ontological descriptions that contain characteristics of specific service instances become difficult to construct and maintain.

	Logic-based approaches	AS-based approaches	Semantic Web approaches	Our approach
Syntactic matching	x	x	x	x
Semantic matching	x	x	x	x
Operational character of behavioral (semantic) specs				x
Flexibility with respect to data model (ontology)	x	x		x
Integral matching	x	x	x	x
Model-driven specs				x

Table 5.1: Comparison of techniques for interface specification and matching.

Finally, most existing specification techniques employ formalisms that lie far away from conventional software engineering methods. In our case, the domain data model and graph transformation rules are based on notations being familiar to software engineers. In particular, there exist various approaches (see Chapters 2 and 3 of [48]) that apply graph transformation in the context of model-driven software development.

In the following, we concisely formulate the main advantages of our work over the conventional approaches for specification of software components. Table 5.2 con-

solidates the discussion. While we concentrate on the models developed in the graph transformation community, the table is extended with a column for algebraic specification (AS) modules. The contents of this column is based on Subsection 5.3.1.

First of all, our method seems to be the only one that provides a systematic approach to construction of intra- and inter-connectors of modules (cf. Table 5.2). Our framework relates heterogeneous concepts appearing in the literature and allows to derive new connectors based on semantic requirements imposed by different application scenarios.

	AS modules	GraTra modules	Our approach
Separation of external and internal specs	x	x	x
Consistency between external and internal specs (intra-connectors)	x	x	x
Framework for systematic design of intra-connectors			x
Loosely defined semantic specs of operations	n/a	(x)	x
Flexibility of inter-connector (contravariance)			x
Aptitude for model-based testing		x	x

Table 5.2: Comparison of component models represented by modular specifications.

Secondly, all available module concepts come with strict semantics for rules specifying import and export interfaces, and body[2]. While strict semantics is natural for rules in the export interface specification, the rules in the import interface specification represent, in fact, incomplete or underspecified descriptions (cf. Subsection 2.2.2). We treat this incompleteness adequately by loose semantics used for these rules.

Thirdly, it is a flexibility of the proposed inter-connector between the import

[2]In general, loose semantics could be assumed in GCSPEC and GRACE approaches. However, the module concepts appearing in [47, 84] come with a classical strict semantics.

interface of the requestor and the export interface of the provider module that distinguishes our approach from the existing ones. The proposed inter-connector generalizes usually employed inclusion relation (ensuring that the associated rules of the systems are isomorphic to each other). We allow the entailment of applicability from import to export as well as the entailment of effects in the opposite direction. This substantially extends the class of compatible modules beyond the ones containing only isomorphic rules in the corresponding interfaces.

Chapter 6

Conclusion and Future Work

In this chapter, we provide a summary of our solutions to the problems outlined in Introduction. We also evaluate these solutions with respect to the requirements stated in Subsections 2.1.5 and 4.1.4. To conclude with, we mention several open issues that were left out of the scope of this work. In regards to this, we propose possible directions for future research.

6.1 Summary and Evaluation

The thesis consists of two tightly interlaced parts. External specifications of software systems, i.e. their *interfaces*, are discussed in the first part. We demonstrate how to construct interface specifications, extend them with semantic markup, and check compatibility of semantic-based interface specifications. In this part of the thesis, we focus on integration scenarios in the Web services domain, in particular, on aspects related to the automatic service selection. But we also claim that approach introduced here can be employed in a more general setting for intra- and inter-enterprise integration of software applications.

In the second part, the discussion goes beyond the systems' interfaces, and the method used for external specifications is reused internally to model the entire systems involved in integration. Here external and internal specifications are assembled using the notion of a *module*. The proposed model servers as a staring point in validation of coupled systems—their correctness is crucial for the accurate functioning of the entire integrated system. In particular, it allows to check consistency of interface specifications with (specifications of) their implementations.

6.1.1 Semantic-driven Integration of Software Systems

In an interface specification, a description of the required or provided service is arranged in structural and behavioral compartments. The structural compartment, formally given by a signature analogous to those appearing in algebraic specifications, introduces service operation declarations. Semantics of these operations is defined in the behavioral compartment. It contains contract-based annotations of operations in the form of conditional graph transformation rules. The behavioral compartment is modeled by conditional graph transformation systems. Rules in these systems are equipped with loose (DPB) semantics to contract operations in the import interface and with strict (DPO) semantics to contract those in the export interface. An aggregation of structural and behavioral compartments allows to introduce an integral service specification. It is represented formally by parameterized conditional graph transformation systems.

We construct three kinds of compatibility relations, all of them might be used in the service selection process. The intended correspondence between declarations and contracts of service operations is reflected by structural and behavioral compatibility relations which are established over the corresponding compartments of required and provided interface specifications. Integral compatibility combines structural and behavioral compatibility relations of both systems participating in the integration. In other words, it allows to check overall compatibility of coupled systems.

Since the integral compatibility relation is defined on top of the informally given semantic requirements on behavioral compatibility, the syntactic constraints imposed by the relation lack for justifications. To solve this problem, the developed concepts are placed into the categorical setting, where we rigorously portray the semantic requirements and redefine the compatibility relations by means of morphisms between formal objects modeling service characteristics. Here the structural, behavioral and integral compatibility relations are specified by signature, substitution, and parameterized substitution morphisms, respectively. This allows to carry out the required justifications, in particular, to show the equivalence between the syntactic requirements provided by the substitution morphism and the semantic requirements of behavioral compatibility.

To ease application of our results in practice, we introduce a conceptual framework. The central place in the framework is occupied by an industry standard providing a uniform way of constructing interface specifications. The aim of this construction is to produce exactly those interface specifications that would serve as suitable

inputs for the automatic selection process. As an example scenario, we use the standard issued by the Open Travel Alliance (OTA) to develop and match standard-based interface specifications of Web services in the travelling business domain.

6.1.2 Semantic-driven Integration vs. Stated Requirements

Following the summary of the first part of the thesis, we check whether the obtained results meet the requirements collected in Subsection 2.1.5.

1. *Formal visual notation(s) for service interface specifications.* Our approach employs parameterized conditional graph transformation systems to specify service interfaces. This graph-based notation has a well-established semantics. It is defined, e.g., in the algebraic approaches to graph transformation [36]. Thus, the requirement is obviously fulfilled.

2. *Compliance with standard model-driven techniques of software development.* The interface specifications based on graph transformation are highly compatible with the UML being the de facto standard language for model-driven software development (MDSD). While a type graph (or ontology) is portrayed in the form analogous to UML class diagrams, graph transformation rules can be represented, e.g., as UML collaboration diagrams [124].

   The task of embedding contract-based annotations defined by graph transformation rules into a software development process can be fulfilled using methodology proposed in [75]. This methodology allows a software engineer to systematically create operation contracts in the development process which is guided by the model-driven and design-by-contract principles. In addition, application of graph transformation for specification and (prototypical) implementation of software systems is supported by a number of tools, such as AGG [15], PROGRES [147], or Fujaba [94, 140].

3. *Compliance with the technological stack of the Web services platform.* Since the technological stack of the Web services platform consists of XML-based standards, the use of proposed service interface specifications in the discovery process requires, first of all, their representation in the from of XML descriptions. For this purpose, one can employ the Graph eXchange Language (GXL) [146] and the Graph Transformation Exchange Language (GTXL) [137] that were recently introduced in the graph transformation community.

GXL is designed as a general graph exchange format, while GTXL allows to portray data contained in graph transformation systems. GTXL uses GXL to describe the graph part of a graph transformation system and additionally extends it with specifications of (conditional) graph transformation rules. XML is chosen as an underlying technology in both languages which are supported by several tools. For example, AGG was proposed in [74] as a visual editor for constructing service descriptions. It allows to translate visual graph-based specifications into GXL and GTXL.

Having presented semantic annotations of service operations in an appropriate format, the next question is how to embed these annotations into the existing Web services standards, such as WSDL. One of possible ways to do this is to extend the operation construct in WSDL as demonstrated in [1].

4. *Modularity and extensibility of service interface specifications.* Interface specifications in our approach are composed of compartments characterizing different aspects of services. This "architectural" solution answers the modularity requirement. To illustrate the extensibility of our proposal, we sketch out how to augment interface specifications with control structures reflecting interactive behavior of a service.

   Transactions of the PROGRES approach or transformation units available in GRACE [84] are the possible candidates for this task. Alternatively, control structures may be expressed in process algebra. For example, in the work of Bracciali et al. in [19] interface specifications contain interaction patterns defined in a subset of π-calculus. In either case, such specifications of interactive service behavior can be attached to the existing structure in a form of an additional compartment.

6.1.3 Summary of the Proposal for Component Model

To build up the component model defined in the second part of this work, one needs to fulfill the following steps. First, to select an appropriate notation for internal specifications defining a component's body. Second, to determine relations between external and internal specifications. And finally, to aggregate the two kinds of specifications and their intra-connecting relations.

The component body is specified by parameterized conditional graph transformation systems. Translation of such model-based descriptions into programming lan-

guages used at the implementation level is supported by a number of approaches proposed in the graph transformation community (see, e.g., [15, 147, 140]).

Having uniformly portrayed external and internal specifications, we introduce a generic framework for a systematic presentation and construction of morphisms of graph transformation systems. The morphisms describing the intra-connectors are derived from formally given semantic requirements on the intended correspondence between external and internal specifications. In particular, the framework facilitates building parameterized behavior-reflecting and substitution morphisms which stand for the import-body and export-body intra-connectors, respectively. Furthermore, one can use the framework to classify and compare graph transformation morphisms existing in the literature.

In order to aggregate specifications and intra-connectors, we need a structuring unit. It is provided by a graph transformation (GTS) module. GTS modules consist of three graph transformation systems (import, body, export) and the intra-connectors defined in the generic framework. The proposed component model tracks consistency between the external and internal specifications. It is also equipped with the inter-connector designed in the first part of the thesis, where the import (required) interface and the export (provided) interface of the modules are related by the parameterized substitution morphism modeling this inter-connector.

6.1.4 Component Model vs. Stated Requirements

We are ready to show that the results summarized above meet the requirements underlying this part of our work. The requirements were first stated in Subsection 4.1.4.

1. *Rigorous formulation of the consistency relations between interface specifications and internal part of an integrating component.* The import-body and export body intra-connectors are used to model the intended correspondence of the interface specifications with the specification of component's body. Since the intra-connectors are defined by parameterized behavior-reflecting and substitution morphisms, the requirement is fulfilled.

2. *Formal visual notation of component model and its compliance with standard model-driven techniques of software development.* In our approach, the notations for description of external and internal specifications of components coincide. The component specifications are uniformly portrayed by parameterized conditional graph transformation systems. Thus, we meet the requirement under

the same justifications as in Subsection 6.1.2 (cf. items 1 and 2).

3. *Aptitude of the component model for model-based testing.* First of all, we would like to mention interactive simulation techniques as a means to verify correctness of component models represented by GTS modules. Many existing graph transformation tools, such as Fujaba [94, 140] or PROGRES [147], offer an interactive visual environment for simulating rule-based models in order to analyze behavior of an application in various situations. Simulation allows designers to play with "what if" scenarios to detect defects in the constructed models.

   In addition to automated reasoning by simulation, the theory of graph transformation provides the basis for static analysis. For instance, critical pair analysis [17] is a powerful technique to statically detect potentially conflicting rule pairs by automatic generation of sample models for which the application of the two rules would be in conflict.

   Available analytical instruments are not limited to approaches based on validation. Reachability analysis by graph parsing [71] or model checking graph transformation systems [143] are the examples of verification techniques.

6.2 Thesis Contributions

Summing up the above discussion, the main practical contributions of this thesis are:

- The technique for generation of *formal and visual interface specifications* containing structural and behavioral (semantic) characteristics of declared operations.

- The *matching procedure* allowing to reveal structural and/or behavioral compatibility between interface specifications of the coupled systems. Application of the matching procedure is illustrated by an example taken from the Web services domain.

- The *model* in the form of a graph transformation module which describes interface specifications and their implementations appearing internally in coupled systems. It also defines consistency relations between interface specifications and internal implementations, and allows to validate correctness of coupled systems prior to the integration.

To formalize the mentioned concepts, we extensively use results obtained in the graph transformation community. A number of new concepts is introduced as well. This lets us claim the thesis possesses the following novelties on the formal account:

- The notion of a *parameterized conditional graph transformation system* as a formal counterpart of service interface specification, where the system's rules may be equipped with strict (DPO) and loose (DPB) semantics.
- The *parametrized substitution morphism* representing the compatibility relation between service interface specifications in the form of parameterized conditional graph transformation systems, and justification of the established morphism against rigorously formulated semantic requirements for compatibility.
- The framework for classifying and systematically defining morphisms of graph transformation systems.
- The notion of a *graph transformation module* consisting of parameterized conditional graph transformation systems which solves the problem of underspecification by means of loose (DPB) semantics in the import interface.

6.3 Open Problems and Future Work

Even though our work addresses various aspects that we consider crucial for the problem we intended to tackle, a few questions were left out of its scope. Let us discuss some of them in this section.

First, the presentation needs to be extended to typed graphs with attributes [52] and sub-typing [138]. The demand for such extension is demonstrated by the example scenario appeared in Chapter 2, where the lack of attribution did not allow to specify the flight segment status as it appears in the OTA standard, i.e. as the attribute of the type FlightSegment.

One of the most important features needed for our work is tool support. In the previous chapter, we have shown a few isolated approaches to automation of structural and behavioral compatibility tests in the literature. The question of integration of these isolated approaches along with their adjustment to the formal setting proposed in this thesis still remains open.

One of the implicit assumptions made in our work is that the required and provided interfaces have the same granularity, i.e. the required and provided operations introduce comparable modifications of system states. In a more general case, the

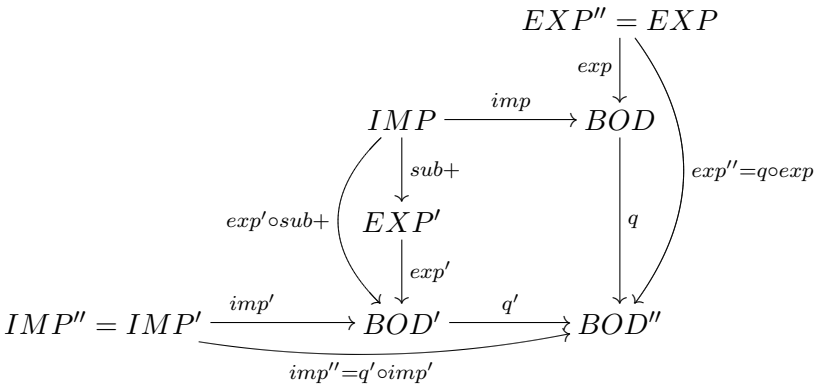


Figure 6.1: Composite of modular specifications.

required functionality can be refined in the provided interface, where one required operation is implemented by several provided operations. For instance, a booking process defined in the required interface by the solitary operation airReserv may be represented in the provided interface by a set of operations which amalgamated effect is compatible to the one of airReserv. An adaptation of the matching procedure to this situation can be done with the help of results obtained in [67, 68]. Spatial and temporal refinements of typed graph transformation systems established in these works have to be built into the compatibility relations proposed in the thesis.

One of the possible tasks in the context of modular specifications is a generalization of module intra-connectors by refinements. Refinements may be used to specify implementation relations, like the one between export and body, in particular, if the exported operations are implemented in the body via a set of elementary operations.

The next possible task is to develop a procedure that would specify the compound system by composition of modules specifying the coupled systems. Then, a compound system specification can be checked by model-based testing technuqes analogously to the specifications of the coupled systems.

The main idea of a composition procedure for the modules MOD and MOD' connected by the parameterized substitution morphism $sub+$ is to create a new module MOD'' having the import interface of MOD', the export interface of MOD and a body implementing the features of both MOD and MOD' (cf. Fig. 6.1).

The fact that import-export and export-body connectors are both described by parameterized substitution morphisms allows us to relate import interface IMP of MOD with body BOD' of MOD', and to construct the body of module MOD'' via the composition of the two graph transformation systems describing BOD and BOD' over the third one IMP specifying the import interface of MOD. Development of

such a composition procedure can use instruments introduced in [68, 76].

6.4 Epilogue

We envision a world where information technologies provide a full spectrum of technological and methodological means for rapid application integration. This would improve corporate agility, speed up time-to-market for new products and services, reduce IT costs, and increase operational efficiency of business units. We believe our work represents a small step towards a new technology that will significantly facilitate integration of software systems.

Bibliography

[1] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M. Schmidt, A. Sheth, and K. Verma. Web Service Semantics—WSDL-S. A joint UGA-IBM Technical Note, version 1.0, April 2005. `http://lsdis.cs.uga.edu/projects/meteor-s/wsdl-s/`.

[2] Open Travel Alliance. `http://www.opentravel.org/`.

[3] Open Travel Alliance. OTA Specification version 2001C, February 2002. `http://www.opentravel.org/2001c.cfm`.

[4] Open Travel Alliance. OTA Specification version 2005A, June 2005. `http://www.opentravel.org/2005a.cfm`.

[5] G. Alonso, F. Casati, H. Kuno, and V Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, 2004.

[6] Amadeus. `http://www.amadeus.com/index.jsp`.

[7] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner. *Algebraic Foundations of Systems Specification*. Springer-Verlag, 2001.

[8] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wust, and J. Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2001.

[9] BEA Systems, Inc. BEA WebLogic Integration: Rapid business integration, 2005. `http://www.bea.com/content/products/weblogic/integrate/`.

[10] BEA Systems, Inc. BEA WebLogic Server 9.0 Overvew, 2005. `http://www.bea.com/content/products/weblogic/server/`.

[11] C. Beierle, U. Hedtstück, U. Pletat, Peter H. Schmitt, and Jörg H. Siekmann. An Order-Sorted Logic for Knowledge Representation Systems. *Artificial Intelligence*, 55(2):149–191, 1992.

[12] B. Benatallah, M.-S. Hacid, and C. Rey. Semantic Reasoning for Web Services Discovery. In *Proc. of the WWW 2003 Workshop on E-Services and the Semantic Web (ESSW'03)*, 2003.

[13] P.A. Bernstein. Middleware: A Model of Distributed System Services. *Communications of ACM*, 39(2):86–98, 1996.

[14] Alfs T. Berztiss. Verification and validation. In S. K. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, volume 2, pages 367–388. World Scientific, 2002.

[15] M. Beyer. *AGG An Algebraic Graph System, User Manual*. Technical University of Berlin, Department of Computer Science, 1993.

[16] M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in CASL. In *Proc. 7th International Conference on Algebraic Methodology and Software Technology (AMAST'98), Manaus, Brasil*, volume 1548 of *LNCS*, pages 341–357. Springer-Verlag, 1999.

[17] P. Bottoni, G. Taentzer, and A. Schürr. Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In *Proc. VL'2000: IEEE International Symposium on Visual Languages*, pages 59–60, Washington, DC, USA, 2000. IEEE Computer Society. Long version available as technical report SI-2000-06, University of Rom.

[18] D. Box. *Essential COM*. Addison-Wesley, 1998.

[19] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.

[20] D. Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recomendation February 10, 2004. `http://www.w3.org/TR/rdf-schema/`.

[21] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *LNCS*. Springer-Verlag, 2005.

[22] E. Bruneton, T. Coupaye, and J. Stefani. The Fractal component model. Technical Report Specification V2, The ObjectWeb Consortium, 2003.

[23] D. Carlson. *Modeling XML Applications with UML: Practical e-Business Applications*. Addison-Wesley, 2001.

[24] Jeremy J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: Implementing the semantic web recommendations. Technical Report HPL-2003-146, Hewlett Packard Laboratories, 2003.

[25] Chin-Liang Chang and Richard C. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., Orlando, FL, USA, 1997.

[26] P. S. Chen, R. Hennicker, and M. Jarke. On the retrieval of reusable software components. In *Proc. of the 2nd International Workshop on Software Reusability*, pages 99–108. IEEE Computer Society Press, March 1993.

[27] A. Cherchago and R. Heckel. Specification matching of web services using conditional graph transformation rules. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), Rome, Italy*, volume 3256 of *LNCS*, pages 304–318. Springer-Verlag, 2004.

[28] M. Chinnici, R. Gudgin, J.-J. Moreau, J. Schlimmer, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0. Part 1: Core Language, W3C Working Draft, August 2005. `http://www.w3.org/TR/wsdl20/`.

[29] I. Claßen, H. Ehrig, and D. Wolz. *Algebraic Specification Techniques and Tools for Software Development - The Act Approach*. AMAST Series in Computing Vol. 1. World Scientific Publishing Co., 1993.

[30] P. Clements. A survey of architecture description languages. In *Proc. 8th International Workshop on Software Specification and Design*, pages 16–25. ACM Press, 1996.

[31] The OWL Services Coalition. OWL-S specification version 1.0, November 2003. `http://www.daml.org/`.

[32] CoFI Task Group on Language Design. CASL – The CoFI algebraic specification language – Summary (version 1.0), 1998. `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.

[33] D. Connolly, F. van Harmelen, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A Stein. DAML+OIL (March 2001) Reference Description, W3C Note, March 2001. `http://www.w3.org/TR/daml+oil-reference`.

[34] A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and J. Padberg. The category of typed graph grammars and their adjunction with categories of derivations. In *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94*, pages 56–74. Springer-Verlag, 1996.

[35] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.

[36] 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 G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163–245. World Scientific, 1997.

[37] J. de Bruijn, H. Lausen, R. Krummenacher, A. Polleres, L. Predoiu, M. Kifer, and D. Fensel. The Web Service Modeling Language WSML, WSML Final Draft, October 2005. `http://www.wsmo.org/TR/d16/d16.1/v0.21/`.

[38] T. De Marco. *Structured Analysis and System Specification*. Prentice-Hall, 1978.

[39] M. Dean, G. Schreiber, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language Reference, W3C Working Draft, March 2003. `http://www.w3.org/TR/2003/WD-owl-ref-20030331/`.

[40] M. Denny. Ontology building: A survey of editing tools, November 2002. `http://www.xml.com/pub/a/2002/11/06/ontologies.html`.

[41] A. Dogac, I. Cingil, G. Laleci, and Y. Kabak. Improving the Functionality of UDDI Registries through Web Service Semantics. In *Proc. of the Third International Workshop on Technologies for E-Services*, volume 2444 of *LNCS*, pages 9–18. Springer-Verlag, 2002.

[42] A. Dogac, Y. Kabak, G. Laleci, S. Sinir, A. Yildiz, S. Kirbas, and Y. Gurcan. Semantically Enriched Web Services for the Travel Industry. *ACM Sigmod Record*, 33(3), September 2004.

[43] M. Dumas, J. O'Sullivan, M. Heravizadeh, D. Edmond, and A. Hofstede. Towards a semantic framework for service description. In *Proc. of the IFIP Conference on Database Semantics.* Kluwer Academic Publishers, April 2001.

[44] ebXML Technical Architecture Specification version 1.04. `http://www.ebxml.org/specs/index.htm#reference_materials`.

[45] H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Proc. 1st Graph Grammar Workshop*, volume 73 of *LNCS*, pages 1–69. Springer-Verlag, 1979.

[46] H. Ehrig and G. Engels. Towards a module concept for graph transformation systems. Technical Report 93-34, Leiden University (The Netherlands), 1993.

[47] H. Ehrig and G. Engels. Pragmatic and semantic aspects of a module concept for graph transformation systems. In *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94*, volume 1073 of *LNCS*, pages 137–154. Springer-Verlag, 1996.

[48] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools.* World Scientific, Singapore, 1999.

[49] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EACTS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin, 1985.

[50] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Berlin, 1990.

[51] H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE Press, 1973.

[52] H. Ehrig, U. Prange, and G. Taentzer. Fundamental theory for typed attributed graph transformation. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), Rome, Italy*, volume 3256 of *LNCS*, pages 161–177. Springer-Verlag, 2004.

[53] G. Engels, R. Heckel, and A. Cherchago. Flexible interconnection of graph transformation modules: A systematic approach. In H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, and G. Taentzer, editors, *Formal Methods in Software and System Modeling*, volume 3393 of *LNCS*, pages 38–63. Springer-Verlag, 2005.

[54] G. Engels, J. M. Küster, R. Heckel, and M. Lohmann. Model based verification and validation of properties. In *Proc. of UNIGRA'03 - Uniform Approaches to Graphical Process Specification Techniques (@ ETAPS 2003), Warsaw, Poland, April 5-6, 2003*, volume 82 of *ENTCS*, 2003.

[55] Web Service Modelling eXecution environment (WSMX) working group. `http://www.wsmx.org/`.

[56] O. Fargemand and A. Olsen. Introduction to SDL-92. *Computer Networks and ISDN Systems*, 26:1143–1167, 1994.

[57] C. Feier, D. Roman, A. Polleres, J. Domingue, M. Stollberg, and D. Fensel. Towards Intelligent Web Services: Web Service Modeling Ontology (WSMO). In *Proc. International Conference on Intelligent Computing (ICIC) 2005, Hefei, China*, 2005.

[58] L. M. G. Feijs. *Formal Specification and Design.* Cambridge University Press, 2005.

[59] D. Fensel. *Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce.* Springer-Verlag, 2001.

[60] D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, 2002.

[61] O. Fodor, M. Dell'Erba, F. Ricci, and H. Werthner. Harmonise: a solution for data interoperability. In *Proc. 2nd IFIP Conference on E-Commerce, E-Business and E-Government, Lisbon, Portugal*, 2002.

[62] Semantic Tools for Web Services. `http://www.alphaworks.ibm.com/tech/wssem/`.

[63] Galileo. `http://www.galileo.com`.

[64] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch prover. Technical Report SRC-RR-82, Hewlett-Packard Company, December 1991.

[65] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering, 2nd Edition*. Prentice Hall, 2002.

[66] K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web Services architecture. *IBM Systems Journal*, 41(2):170–177, 2002.

[67] M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Concrete spatial refinement construction for graph transformation systems. Technical Report SI 97/10, Università di Roma La Sapienza, Dip. Scienze dell'Informazione, 1997.

[68] M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Spatial and temporal refinement of graph transformation systems. In *Proc. of Mathematical Foundations of Computer Science 1998*, volume 1450 of *LNCS*, pages 553–561. Springer-Verlag, 1998.

[69] M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Refinements and modules for typed graph transformation systems. In J.L. Fiadeiro, editor, *Proc. Workshop on Algebraic Development Techniques (WADT'98), at ETAPS'98, Lisbon, April 1998*, volume 1589 of *LNCS*, pages 138–151. Springer-Verlag, 1999.

[70] John V. Guttag, James J. Horning, S. J. Garland, and K. D. Jones. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[71] S. Gyapay, Á. Schmidt, and D. Varró. Joint optimization and reachability analysis in graph transformation systems with time. In *Proc. GT-VMT 2004, International Workshop on Graph Transformation and Visual Modelling Techniques*, volume 109 of *ENTCS*, pages 137–147. Elsevier, 2004.

[72] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4):287–313, 1996.

[73] A. Haller, E. Cimpian, A. Mocan, E. Oren, and C. Bussler. WSMX—a semantic service-oriented architecture. In *Proc. International Conference on Web Services (ICWS 2005), Orlando, Florida (USA)*, July 2005.

[74] J.H. Hausmann, R. Heckel, and M. Lohmann. Model-based discovery of web services. In *Proc. 2004 IEEE International Conference on Web Services (ICWS 2004) July 6-9, 2004, San Diego, California, USA*, 2004.

[75] J.H. Hausmann, R. Heckel, and M. Lohmann. Model-based development of Web service descriptions enabling a precise matching concept. *International Journal of Web Services Research*, 2(2):67–85, 2005.

[76] R. Heckel. *Open Graph Transformation Systems: A New Approach to the Compositional Modelling of Concurrent and Reactive Systems.* PhD thesis, TU Berlin, 1998.

[77] R. Heckel and A. Cherchago. Application of Graph Transformation for Automating Web Service Discovery. In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. `http://drops.dagstuhl.de/portals/5/`.

[78] R. Heckel and A. Cherchago. Structural and behavioural compatibility of graphical service specifications. *Journal of Logic and Algebraic Programming*, 2006. To appear.

[79] R. Heckel, A. Cherchago, and M. Lohmann. A formal approach to service specification and matching based on graph transformation. *ENTCS*, 105:37–49, 2004.

[80] R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Mathematical Structures in Computer Science*, 6(6):613–648, 1996.

[81] R. Heckel, H. Ehrig, U. Wolter, and A. Corradini. Double-pullback transitions and coalgebraic loose semantics for graph transformation systems. *Applied Categorical Structures*, 9(1), January 2001. See also TR 97-07 at `http://www.cs.tu-berlin.de/cs/ifb/TechnBerichteListe.html`.

[82] R. Heckel, G. Engels, H. Ehrig, and G. Taentzer. Classification and comparison of modularity concepts for graph transformation systems. In Ehrig et al. [48], pages 669–690.

[83] R. Heckel, G. Engels, H. Ehrig, and G. Taentzer. A view-based approach to system modelling based on open graph transformation systems. In Ehrig et al. [48], pages 639–667.

[84] R. Heckel, B. Hoffmann, P. Knirsch, and S. Kuske. Simple modules for GRACE. In *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn*, 1998.

[85] A. Heß, E. Johnston, and N. Kushmerick. ASSAM: A Tool for Semi-Automatically Annotating Semantic Web Services. In *Proc. 3rd International Semantic Web Conference (ISWC 2004), Hiroshima, Japan*, volume 3298 of *LNCS*, pages 320–334. Springer-Verlag, 2004.

[86] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[87] I. Horrocks, Peter F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language, Combining OWL and RuleML, W3C Member Submission, May 2004. `http://www.w3.org/Submission/SWRL/`.

[88] International Electrotechnical Commission (IEC). Part 3: Programming Languages. Technical Report IEC 1131-3, IEC Geneva, 1993.

[89] International Business Machines Corporation (IBM). MQ Series Workflow for Business Integration, 1999. `http://www-306.ibm.com/software/integration/mqfamily/library/`.

[90] International Business Machines Corporation (IBM). WebSphere MQ Integrator Broker: Introduction and Planning, 2002. `http://www-306.ibm.com/software/integration/mqfamily/library/`.

[91] Tourism Harmonisation Network: Harmonise Project IST-2000-29329. `http://www.harmonise.org/`.

[92] J. Ivers, N. Sinha, and K. Wallnau. A Basis for Composition Language CL. Technical Report CMU/SEI-2002-TN-026, CMU SEI, 2002.

[93] Michael C. Jaeger, G. Rojec-Goldmann, C. Liebetruth, G. Mühl, and K. Geihs. Ranked Matching for Service Descriptions using OWL-S. In *Proc. Kummunikation in Verteilten Systemen (KiVS) February 2005, Kaiserslautern, Germany*, 2005.

[94] Jens H. Jahnke, Wilhelm Schäfer, and Albert Zündorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In *Proc.*

of European Software Engineering Conference (ESEC/FSE), volume 1302 of *LNCS*. Springer-Verlag, September 1997.

[95] M. Jarke, R. Gallersdörfer, M.A. Jeusfeld, M. Staudt, and S. Eherer. ConceptBase—a deductive object base for meta data management. *Journal of Intelligent Information Systems, Special Issue on Advances in Deductive Object-Oriented Databases*, 4(2):167–192, 1995.

[96] J.-J. Jeng and B. H. C. Cheng. Specification matching for software reuse: A foundation. In *Proc. of the ACM SIGSOFT Symposium on Software Reusability (SSR'95)*, pages 97–105, April 1995.

[97] Warren H. Jessop. Ada packages and distributed systems. *ACM SIGPLAN Notices*, 17(2):28–36, 1982.

[98] K.N. King. *Modula-2: A Complete Guide*. Houghton Mifflin College Div, 1988.

[99] G. Klyne and J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Recommendation February 10, 2004. `http://www.w3.org/TR/rdf-concepts/`.

[100] H. Knublauch, Ray W. Fergerson, Natalya F. Noy, and Mark A. Musen. The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. In *Proc. 3rd International Semantic Web Conference (ISWC 2004), Hiroshima, Japan*, volume 3298 of *LNCS*, pages 229–243. Springer-Verlag, 2004.

[101] H.-J. Kreowski and S. Kuske. On the interleaving semantics of transformation units—a step into GRACE. In *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94*, pages 89–106. Springer-Verlag, 1996.

[102] B. Krieg-Brückner and D. Sannella. Structuring specifications in-the-large and in-the-small: higher-order functions, dependent types and inheritance in Spectral. In *Proc. colloquium on Combining Paradigms for Software Development, Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 494 of *LNCS*, pages 313–336. Springer-Verlag, 1991.

[103] K.-K. Lau and Z. Wang. A Taxonomy of Software Component Models. In *Proc. 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 88–95. IEEE Computer Society Press, 2005.

[104] H. Lausen and D. Roman. WSMO F-Logic Syntax, WSMO Working Draft, March 2004. `http://www.wsmo.org/2004/d16/d16.4/v01/`.

[105] D. Lowe. *Client/Server Computing for Dummies, 2nd Edition*. Hungry Minds, Inc., 1997.

[106] D. McDermott. DRS: A Set of Conventions for Representing Logical Languages in RDF, January 2004. `http://www.cs.yale.edu/homes/dvm/daml/DRSguide.pdf`.

[107] S. McIlraith, T. C. Son, and H. Zeng. Semantic Web services. *IEEE Intelligent Systems. Special Issue on the Semantic Web*, 16(2):46–53, March/April 2001.

[108] B. Meyer. *Object-oriented Software Construction, 2nd Edition*. Prentice Hall, 1997.

[109] Sun Microsystems. JavaBeans Specification, Version 1.01, 1997. `http://java.sun.com/products/javabeans/docs/spec.html`.

[110] Sun Microsystems. Enterprise JavaBeans Specification, Version 2.0, 2001. `http://java.sun.com/products/ejb/docs.html`.

[111] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos—a language for representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, 1990.

[112] E. Newcomer and G. Lomow. *Understanding SOA with Web Services*. Addison Wesley Professional, 2004.

[113] O. Nierstrasz, G. Arevalo, S. Ducasse, R. Wuyts, Andrew P. Black, Peter O. Müller, C. Zeidler, T. Genssler, and R. van den Born. A component model for field devices. In *Proc. 1st International IFIP/ACMWorking Conference on Component Deployment*, volume 2370 of *LNCS*, pages 200–209. Springer-Verlag, 2002.

[114] Object Management Group (OMG). Model Driven Architecture (MDA), 2001. `http://www.omg.org/mda`.

[115] Object Management Group (OMG). CORBA Component Model, V3.0, 2002. `http://www.omg.org/technology/documents/formal/components.htm`.

[116] Object Management Group (OMG). UML 2.0 Superstructure Specification, 2003. http://www.uml.org/.

[117] Large Scale Distributed Information Systems (University of Georgia). METEOR-S: Semantic Web Services and Processes. http://lsdis.cs.uga.edu/projects/meteor-s/.

[118] E. Orfali, D. Harkey, and J. Edwards. *Client/Server Survival Guide, 3rd Edition*. John Wiley and Sons, Inc., 1999.

[119] M. Paolucci, T. Kawmura, T. Payne, and K. Sycara. Semantic matching of Web services capabilities. In *Proc. First International Semantic Web Conference*, 2002.

[120] J. Peters. Datalex: Web services in the Travel Industry, 2002. http://www.datalex.com/pdfs/Web_services.pdf.

[121] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proc. 4th International Conference on Configurable Distributed Systems (ICCDS'98), Annapolis, Maryland, USA*, pages 43–52. IEEE Press, May 1998.

[122] N. Prasad. *IBM Mainframes: Architecture and Design, 2nd Edition*. McGraw-Hill Osborne Media, 1994.

[123] SATINE Project. http://www.srdc.metu.edu.tr/webpage/projects/satine.

[124] Heckel. R. and St. Sauer. Strengthening UML collaboration diagrams by state transformations. In H. Hußmann, editor, *Proc. Fundamental Approaches to Software Engineering (FASE'2001), Genova, Italy*, volume 2185 of *LNCS*, pages 109–123. Springer-Verlag, April 2001.

[125] W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.

[126] L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. PhD thesis, TU Berlin, 1996.

[127] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[128] W.A. Ruh, F.X. Maginnis, and W.J. Brown. *Enterprise Application Integration: A Wiley Tech Brief*. John Wiley and Sons, Inc., 2000.

[129] Sabre. `http://www.sabre.com`.

[130] D. Sannella and A. Tarlecki. Extended ML: past, present and future. In *Proc. 7th Workshop on Specification of Abstract Data Types*, volume 534 of *LNCS*, pages 297–322. Springer-Verlag, 1991.

[131] A. Schürr and A. Winter. UML packages for PROgrammed Graph REwrite Systems. In *Selected Papers of 6th International Workshop on Theory and Application of Graph Transformations (TAGT'98), Paderborn, Germany*, volume 1764 of *LNCS*, pages 396–409. Springer-Verlag, 1999.

[132] J. Scicluna, C. Abela, and M. Montebello. Visual Modelling of OWL-S Services. In *Proc. IADIS International Conference WWW/Internet, Madrid, Spain*, October 2004.

[133] Galileo Web Services. `http://xml.coverpages.org/GalileoGlobalWS.html`.

[134] Sabre Web Services. `http://www.sabretravelnetwork.com`.

[135] K. Sivashanmugam, K. Verma, A. Sheth, and J. Miller. Adding semantics to web services standards. In *Proc. First International Conference on Web Services (ICWS03), Las Vegas, Nevada*, pages 395–401, 2003.

[136] J. M. Spivey. *Introducing Z: a Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[137] G. Taentzer. Towards Common Exchange Formats for Graphs and Graph Transformation Systems. In *Proc. Uniform Approaches to Graphical Process Specification Techniques (UNIGRA01)*, 2001.

[138] G. Taentzer and A. Rensink. Ensuring structural constraints in graph-based models with type inheritance. In M. Cerioli, editor, *Proc. Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005*, volume 3442 of *LNCS*, pages 64–79. Springer-Verlag, 2005.

[139] G. Taentzer and A. Schürr. DIEGO, another step towards a module concept for graph transformation systems. In *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation"*, volume 2 of *ENTCS*, 1995.

[140] From UML to Java and Back Again: The Fujaba homepage. `http://www.fujaba.de`.

[141] UDDI Consortium. UDDI Executive White Paper, October 2004. `http://uddi.org/pubs/uddi-exec-wp.pdf`.

[142] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, March 2000.

[143] D. Varró. Towards symbolic analysis of visual modelling languages. In P. Bottoni and M. Minas, editors, *Proc. GT-VMT 2002: International Workshop on Graph Transformation and Visual Modelling Techniques*, volume 72 of *ENTCS*, pages 57–70, Barcelona, Spain, October 2002. Elsevier.

[144] K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar, and J. Miller. METEOR-S WSDI: A Scalable Infrastructure of Registries for Semantic Publication and Discovery of Web Services. *Journal of Information Technology and Management, Special Issue on Universal Global Integration*, 6(1):17–39, 2005.

[145] C. Walther. Many-sorted unification. *Journal of the ACM*, 35(1):1–17, January 1988.

[146] A. Winter, B. Kullbach, and V. Riediger. An Overview of the GXL Graph Exchange Language. In *Software Visualization: International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001. Revised Papers*, volume 2269 of *LNCS*, pages 324–336. Springer-Verlag, 2002.

[147] A. Winter and A. Schürr. Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems. Technical Report 97-3, RWTH Aachen, FG Informatik, October 1997.

[148] Workflow Management Coalition. Terminology and Glossary, Februrary 1999. `http://www.wfmc.org`.

[149] Web Service Modeling Language (WSML) working group. `http://www.wsmx.org/`.

[150] Web Service Modeling Ontology (WSMO) working group. `http://www.wsmo.org/`.

[151] World Wide Web Consortium (W3C). Semantic Web, 2001. `http://www.w3.org/2001/sw/`.

[152] World Wide Web Consortium (W3C). Web Services Architecture, W3C Working Group Note 11, February 2004. `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/`.

[153] Worldspan. `http://www.worldspan.com/`.

[154] A.M. Zaremski. *Signature and specification matching.* PhD thesis, Carnegie Mellon University, Pittsburg, Pa., January 1996.

[155] A.M. Zaremski and J.M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(2):146–170, April 1995.

[156] A.M. Zaremski and J.M. Wing. Specification matching of software components. In *Proc. 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT95)*, volume 20(4) of *ACM SIGSOFT Software Engineering Notes*, pages 6–17, October 1995.