

# Dynamic Software Architectures

A Style-Based Modeling and Refinement Technique  
with Graph Transformations

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 Sebastian Thöne

in partial fulfillment of the requirements for the degree of  
doctor rerum naturalium  
(Dr. rer. nat.)

Paderborn, October 2005



# Abstract

A good architectural design allows to capture the overall complexity of large, distributed systems at a higher level of abstraction. This is especially important for reconfigurable systems where the architectural configuration is subject to (constant) changes at runtime. When designing such a dynamic architecture, the software architect has to bring the functional business requirements and the available communication and reconfiguration mechanisms of the intended target platform in line.

As it is a complex task to incorporate these often diverging requirements into the architectural model, we propose a stepwise approach similar to the MDA initiative. We start with a platform-independent model capturing the business requirements and add platform-specific details in a later step. For each level of platform abstraction and associated platform, we define an architectural style which describes the characteristics of the platform. This way, conformance to the architectural style entails consistency between model and the underlying platform.

Besides run-time configurations of components and connections, architectural models also comprise the description of processes that control the communication and reconfiguration behavior. To provide operational semantics, we formalize architectural models as graphs and architectural styles as graph transformation systems. UML is added as high-level modeling language on top, and profiles are used to adapt UML to certain architectural styles.

Due to the stepwise procedure, we also have to ensure the mutual consistency between models at different levels of abstraction. For this purpose, we define formal refinement criteria which require that both structural and behavioral properties are preserved at the lower level of abstraction. Based on refinement relationships between abstract and platform-specific architectural styles, an algorithm allows to verify that all abstract, business-level behavior can also be realized in the platform-specific architecture and that no new behavior is added. These innovative refinement techniques for graph transformation models facilitate a stepwise, platform-consistent development of dynamic software architectures.



*To Andrea and our son Dominik.*



# Acknowledgment

As this work would not have been possible without the support I received from many different people, let me take this chance to express my gratitude to them.

First of all, I would like to mention my doctoral advisor Prof. Dr. Gregor Engels. Being a member of his research group “Database and Information Systems”, I could always seek his advice and benefit from his comprehensive knowledge of the research area. Regularly pushing for further improvements of my drafts, he substantially contributed to the quality of this thesis.

Thanks also go to Prof. Dr. Wilhelm Schäfer and Prof. Dr. Leena Suhl for their readiness to become my co-supervisors. In this context, let me especially thank Gregor Engels and Wilhelm Schäfer for writing the reviews of this thesis.

During my research, I had the lucky chance to collaborate with Luciano Baresi (presently at the Politecnico di Milano), Dániel Varró (presently at the Budapest University of Technology and Economics), and especially with Reiko Heckel (presently at the University of Leicester). I benefited very much from their excellent expertise, our productive discussions, and our cooperation on several joint publications. Similarly, I would like to thank all members of the Database and Information Systems group in Paderborn for the friendly and convenient working atmosphere I could experience there.

My work has been accompanied and financially supported by the *International Graduate School Dynamic Intelligent Systems* at the University of Paderborn. I feel very grateful for this support and wish the graduate school a successful continuation of their program for the benefit of many future Ph.D. students.

Last but foremost, I would like to thank my wife Andrea for never giving up her patience and warm-hearted encouragement.

Thank you all very much!

Sebastian Thöne  
Paderborn, October 2005





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Architecting large, distributed software systems . . . . .	1
1.2	Dynamic software architectures . . . . .	7
1.3	Platform-consistent development . . . . .	10
1.3.1	Platform consistency . . . . .	10
1.3.2	Different levels of platform abstraction . . . . .	13
1.3.3	Architecture refinement . . . . .	16
1.4	Structure of the thesis . . . . .	19
<b>2</b>	<b>Survey of related work</b>	<b>21</b>
2.1	Some historical notes . . . . .	21
2.2	Requirements . . . . .	22
2.2.1	Requirements for architecture descriptions . . . . .	23
2.2.2	Requirements for platform descriptions . . . . .	25
2.2.3	Requirements for architecture refinements . . . . .	26
2.3	Existing approaches and open problems . . . . .	28
2.3.1	Formal methods for dynamic architectures . . . . .	28
2.3.2	Platform awareness in architecture descriptions . . . . .	37
2.3.3	Architecture refinement techniques . . . . .	41
2.4	Summary . . . . .	46
<b>3</b>	<b>The style-based approach – an overview</b>	<b>47</b>
3.1	Style-based modeling . . . . .	48
3.1.1	Formal, graph-based descriptions . . . . .	48
3.1.2	UML as concrete syntax . . . . .	54
3.2	Style-based refinement . . . . .	58
<b>4</b>	<b>Graph transformation theory – the formal background</b>	<b>63</b>
4.1	Graphs and graph schemas . . . . .	64
4.1.1	Type graphs and typing morphisms . . . . .	65
4.1.2	Attributes and typed attributed graphs . . . . .	69

4.1.3	Cardinalities and other constraints . . . . .	72
4.1.4	Graph schemas . . . . .	74
4.2	Graph transformations . . . . .	74
4.2.1	Informal introduction and example . . . . .	75
4.2.2	Operational semantics . . . . .	78
4.2.3	Graph transformation systems . . . . .	85
4.3	Transformation-based system models . . . . .	85
4.3.1	Reachability properties and parsing problems . . . . .	86
4.3.2	Graph transition systems . . . . .	87
<b>5</b>	<b>Architectural styles as graph transformation systems</b>	<b>93</b>
5.1	Structural parts and their instantiation . . . . .	94
5.2	Behavioral parts and their instantiation . . . . .	97
5.2.1	Reconfiguration mechanisms . . . . .	97
5.2.2	Communication mechanisms . . . . .	100
5.2.3	Instantiating platform mechanisms in architectural behavior . . . . .	102
5.3	A style for service-oriented architectures . . . . .	112
5.4	Summary . . . . .	118
<b>6</b>	<b>Style-based modeling of dynamic software architectures</b>	<b>119</b>
6.1	Profile for service-oriented architectures . . . . .	120
6.1.1	Stereotypes concerning structure . . . . .	120
6.1.2	Stereotypes concerning behavior . . . . .	122
6.2	Translation into the semantic domain . . . . .	129
6.3	Summary . . . . .	135
<b>7</b>	<b>Style-based refinement of dynamic software architectures</b>	<b>137</b>
7.1	Syntactical versus semantical approaches . . . . .	138
7.2	Structural refinement . . . . .	141
7.2.1	Abstraction relationship between styles . . . . .	142
7.2.2	Structural refinement criterion . . . . .	145
7.3	Behavioral refinement . . . . .	148
7.3.1	Behavior preservation . . . . .	148
7.3.2	Observational substitutability . . . . .	156
7.4	Testing behavioral refinement . . . . .	162
7.4.1	Testing behavior preservation . . . . .	162
7.4.2	Testing observational substitutability . . . . .	166
7.5	Summary . . . . .	167

<b>8</b>	<b>Existing tool support for modeling and refinement</b>	<b>171</b>
8.1	Graph transformation tools . . . . .	173
8.1.1	Required features . . . . .	173
8.1.2	PROGRES . . . . .	175
8.1.3	AGG . . . . .	177
8.1.4	FUJABA . . . . .	178
8.1.5	CheckVML . . . . .	179
8.1.6	GROOVE . . . . .	181
8.1.7	Summary . . . . .	182
8.2	UML tools and tool integration . . . . .	184
8.3	Practical example with GROOVE . . . . .	187
8.4	Summary . . . . .	193
<b>9</b>	<b>Conclusion</b>	<b>195</b>
9.1	Evaluation against the requirements . . . . .	196
9.2	Conclusions and future work . . . . .	198
<b>A</b>	<b>Architectural style for component-based architectures</b>	<b>217</b>
A.1	Graph schema . . . . .	217
A.2	Graph transformation rules . . . . .	219
<b>B</b>	<b>Architectural style for service-oriented architectures</b>	<b>229</b>
B.1	Graph schema . . . . .	229
B.2	Graph transformation rules . . . . .	233
<b>C</b>	<b>Abstract model of the travel agency architecture</b>	<b>243</b>
<b>D</b>	<b>Concrete model of the travel agency architecture</b>	<b>247</b>



# Chapter 1

## Introduction

The thesis at hand shall contribute to ongoing research on model-driven development techniques for large and possibly distributed software systems. To overcome the complexity of such systems, they have to be decomposed into self-contained components which can be developed independently of each other. The integration of these components happens at the level of *software architecture*, which represents an abstract view on the system focusing on the configuration of and communication between components while neglecting their internal structure and behavior.

Our work deals with the model-driven development of an advanced kind of software architecture, namely *dynamic* architectures which can be reconfigured at runtime. In the following sections, we provide an introduction to the current challenges in this context and the problems we want to address by this thesis. At the end of the chapter, we provide an overview of the rest of this thesis.

### 1.1 Architecting large, distributed software systems

There is a growing need for large, but distributed and loosely coupled software systems. Some of the key factors behind their popularization are increased system availability through better fault tolerance, parallel execution, as well as improved scalability and flexibility [80].

As a typical example, which we use throughout the thesis, consider the development of a modern virtual enterprise, namely an *electronic travel agency*. The agency should allow customers to individually arrange and book journeys including flight, hotel accommodation, car rental and so on. Customers do not need to solve all their constraints themselves any more, but employ a

more or less intelligent software agent which autonomously negotiates with the travel agency system.

Figure 1.1 sketches the planned distributed system with the travel agency system, the client's personal agent, and other parties involved, i. e., the airline providing the flight, the hotel providing the accommodation and the client's home bank handling the payment for the journey.

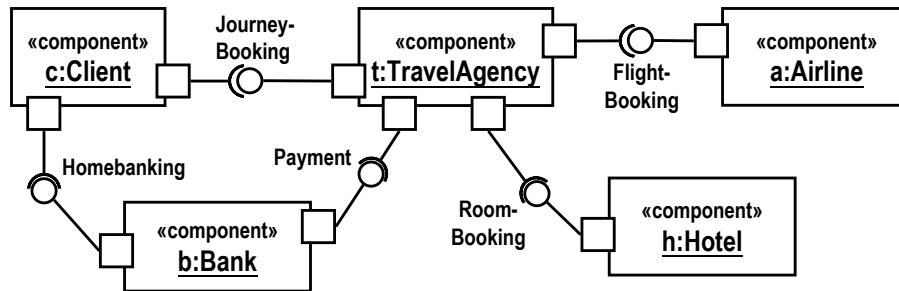


Figure 1.1: Components involved in the electronic travel agency

When developing such software systems, we are facing the following difficulties and requirements:

- Due to its size and complexity, we have to modularize and decompose the system into smaller units. For the sake of reusability, the current trend towards component-based software suggests to divide a system into self-contained functional components [148]. Some of these components can, e. g., be reused from previous projects or purchased as commercial off-the-shelf components.
- We have to cope with existing systems of other business partners which are not under own control. When developing the electronic travel agency, for instance, we have to integrate the systems of partner airlines, hotel groups, banks, and so forth.

Moreover, as business partners usually protect and hide the internal structure of their software systems, the integration can only be based on interface descriptions.

- Some parts of the overall system might even be unknown at design time. For instance, we do not know in advance which proprietary or vendor-specific software agents the clients will use to access the travel agency. However, in order to maximize the market share, the travel agency should at best provide access to all of them.

We can address these difficulties of complexity, component integration, and lacking knowledge only by means of abstraction, omitting unnecessary or not available details. This is where *software architecture* as an abstract view of the system’s runtime structure comes into play. Hiding some of the details through encapsulation helps to better identify and sustain the properties of the system [142].

As one cannot succinctly describe which details can be abstracted away over all domains and system sizes, there is much controversy over a general definition of software architecture [73]. We adopt as our working definition a slightly edited version of that provided by Shaw and Garlan [144]:

“Software architecture [is an abstract level of design that] involves the description of [runtime] elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.”

In this definition, we emphasize the runtime aspect because some authors use the term software architecture for the code structure of a system rather than its runtime structure [45]. However, the task of structuring source code into modules is better known as “programming in the large” [129]. Although it might be advantageous if the modular structure of the source code matches the decomposition structure of a running system, the individual software components are usually implemented sharing parts of the code (e.g., libraries). Therefore, architectural design and programming in the large, though closely related, are separate design activities.

During the last decade, a number of dedicated *Architecture Description Languages* (ADL) have been proposed to capture software architectures. Their essential constructs to describe the runtime structure of a system are *components*, *interfaces*, and *connectors* which are assembled to *architectural configurations* [107].

**Components and interfaces.** In general, components are abstractions for heterogeneous pieces of software written in different programming languages and of varying granularity, from primitive functions to complex applications. For a component to be “composable” with other components, it needs to be sufficiently self-contained. Also, it needs to come with a clear specification of what it requires and provides [148].

Given this specification in form of *provided* and *required interfaces*, we can abstract from internal details and encapsulate a component’s implementation by explicit interaction points to its environment, also called *ports*. An interface description reveals, e.g., under which name a functionality can be accessed and its input/output behavior in terms of data units and formats.

The concept of a “black box” component is useful for abstracting from implementation details, e. g., those of the travel agency system in our example, and to reason about system parts whose details are not known, e. g., those run by external business partners. Despite the lacking details, we can investigate the mutual compatibility of the components based on their interface descriptions. For example, we can describe that our travel agency component can accept journey requests from any arbitrary client component as long as the client adheres to the `JourneyBooking` interface (cf. Fig. 1.1).

Related components like, e. g., our travel agency and its clients or business partners, communicate with each other by exchanging messages, signals or data units. A correct communication depends on the mutual compatibility of their *communication behavior*. Thus, we have to describe a component’s behavior at the architectural level, too. However, in order to retain the encapsulation principle and not to uncover any internal, implementation-specific computation behavior, the behavior description should only reveal the component’s externally observable behavior.

In favor of the reusability intended by component-based software, most ADLs allow to describe specific component *types* which can be instantiated multiple times in a single architectural configuration with all instances sharing the properties defined for the type. This way, types become abstractions that encapsulate functionality, interface definitions, and communication behavior into reusable building blocks [107, 145].

**Connectors.** In an architectural configuration, compatible interfaces or ports of different components are connected so that these components can communicate with each other. Since such connections have to bridge not only physical distances but also logical obstacles like diverging name spaces, data formats, or communication protocols, they should be made explicit as architectural elements of their own, called *connectors* [51].

Sometimes, connectors function as adapters bridging even different component interfaces. Their implementation typically involves some computation such as format conversion, buffering, event dispatching, synchronization over shared variables, client-server protocols and so forth [145]. Following the main research stream, we distinguish connectors from components because they do not provide additional functionality to the current application; on the contrary, some other approaches treat connectors as special components [97].

Analogously to components, ADLs commonly allow to define reusable connector *types* which can be instantiated several times. The type definitions include interface-like descriptions of a connector’s end points revealing which component interfaces can be glued by the connector. In order to enable type



checking and to prove that two interfaces can be glued by a certain connector type, a formal interface representation of both component and connector types has been proposed, e. g., by Allen and Garlan [6].

The terminology of components and connectors, nowadays widely adopted by the software architecture community, is in the tradition of Garlan and Shaw [51, 144]. In the beginning, slightly different terms have been used in parallel. For example, Perry and Wolf distinguish *processing elements* which perform transformations on data, *data elements* which contain the information that is used and transformed, and *connecting elements* which glue the different pieces of the architecture together [125].

**Architectural configurations.** Needless to say, an architecture model should not only define the various component and connector types but also how the system’s runtime configuration is built out of multiple instances thereof. For this reason, an ADL allows to construct component and connector instances as well as bindings between their interfaces.

The resulting configuration determines the paths along which component instances can communicate with each other. This way, the configuration structure strongly influences the global behavior of the system. A configuration description may also cover additional resources relevant for the communication behavior, e. g., data units, documents, and so on.

In order to reason about the global behavior and to decide if, e. g., the business processes of our travel example can be realized as required, an ADL should provide a suitable notion of *behavior composition* which allows to derive the overall behavior from a parallel composition of the concurrently running components. Moreover, the ADL should be equipped with *operational semantics* allowing to execute and simulate the composed behavior. The advantages of formal semantics are further discussed in the following paragraphs.

**Formal methods.** The description of a software architecture helps to document and understand the high-level relationships among involved subsystems. This allows engineers to make principled choices between different design alternatives and to build new systems as variations of old ones. Also, such descriptions enable them to communicate system designs to other engineers and stakeholders. Eventually, they facilitate software maintenance because documentations of the system’s structure and properties reduce the time spent on understanding the code [144].

Those documentation purposes can already be fulfilled by informal “lines and boxes” models, which we encounter in most early practices of software

architecture descriptions [146]. However, the real challenge in software architecture research is not to use architecture descriptions as documentations but to exploit them in all phases of a software development process – from requirements analysis up to testing.

In this context, the main deficiency of “lines and boxes” modeling languages is their missing *formal semantics*. By formal semantics we mean a precise calculus or symbolic logic which allows a computer to represent the meaning of the various modeling elements and to perform computations and manipulations on these elements. In particular, the semantics of behavior descriptions should be operational, i. e., the specified behavior should be executable by appropriate interpreters.

A recent representative for modeling languages without formal semantics is the Unified Modeling Language (UML)<sup>1</sup>. As a general purpose modeling language, it can also be used for architecture descriptions [106]. For instance, Fig. 1.1 describes the architectural configuration of our distributed travel system as a UML *component diagram*. However, without formal semantics the diagram is not more than a modern variant of a “lines and boxes” model.

Unlike UML and similar languages, the research area of *formal methods* is especially concerned with the development of specification techniques that include formal semantics and can therefore be analyzed and simulated by appropriate algorithms and tools. An increasing number of proposals impressively proves the benefits of using formal methods in conjunction with architecture descriptions for driving system construction from requirements to implementation.

For instance, the work by van Lamsweerde [153] shows how software architectures can systematically be derived from system requirements, and Heckel and Engels [59] show how to ensure the consistency between functional requirements and software architecture. Formal descriptions of component types and interfaces enable type checking and help to decide whether components with complementary functionality will be able to interact properly [6, 75]. Besides, one can detect faults and inconsistencies like deadlocks already at an early stage of the development process [88]. This also holds for non-functional properties like performance or dependability which heavily depend on the software architecture [7, 76]. Eventually, formal architecture models can also support later development phases like testing [17].

The weakness of many formal methods is their decreasing efficiency when it comes to the analysis of real-size models. A prominent example is the *state explosion problem* [26] in the area of model checking. It describes the fact that the size of the state space representing the behavior of a system often

---

<sup>1</sup>[www.uml.org](http://www.uml.org)

grows exponentially in the number of processes and variables involved.

Fortunately, the high level of abstraction we are facing in software architectures reduces the impact of this problem: As we neglect the components' internal computations, the complexity of a model is decreased while preserving the subset of properties we are interested in. Following the “divide-and-conquer” paradigm, one can analyze internal properties of individual components separately, and, with the help of compositional analysis strategies, derive properties of the composite configuration from the analysis of its individual constituents. A recent application of this strategy for model checking is presented in [52].

Another often-mentioned weakness of formal methods is their limited usability for practitioners, which is closely linked to complex mathematical notations. Thus, a promising research direction currently observable in the area of model-driven development is the combination of visual modeling languages as a user-friendly notation with formal methods as semantic domain for computer-based analysis (for instance, see [89]).

In this thesis, we will propose a similar strategy for modeling an advanced kind of software architectures, namely *dynamic* architectures. However, facing the many existent ADLs, we are not primarily interested in “yet another” notation or ADL. Instead, we concern ourselves with a well-defined, uniform, and flexible formal method for architecture descriptions which could in principle be combined with different concrete notations.

## 1.2 Dynamic software architectures

In recent times, we are facing new system types which cannot be captured by static architectures with a fixed configuration any more. Instead, the architecture of such systems is subject to constant changes triggered by certain states, events or user requests. Examples include

- **“24/7” systems** that have to run 24 hours a day and, thus, require online updates, i. e., without shutting down the entire system,
- **self-healing systems** [50] that own a more or less intelligent mechanism for autonomous reactions to upcoming inconsistencies,
- **context-sensitive systems** that adapt themselves to changing conditions in their environment like, e. g., different levels of user experience,
- **mobile systems** [8] that reconfigure their architecture in the case of changing locations in order to provide location-specific services to their users,

- **loosely coupled enterprise systems** that allow business partners to flexibly connect to other enterprises in a supply chain, to form virtual enterprises, and to adapt to changing market conditions in order to maximize the current profit.

Our electronic travel agency example falls into the last category as it has to deal with several different airlines, hotel companies, and banks. Depending on an incoming client request, the travel agency should select and connect to airlines serving the client's destination, to hotel companies offering accommodation at that place, and to the bank managing the client's personal bank account. This situation is sketched in Fig. 1.2.

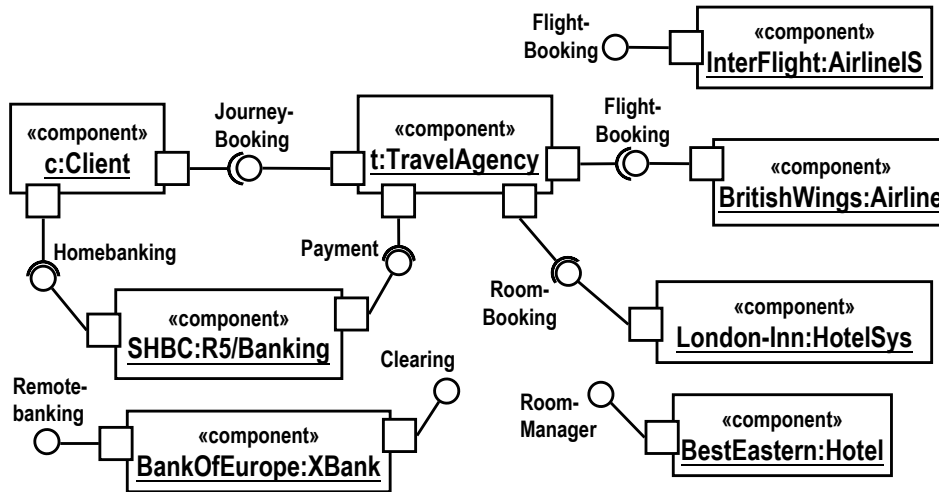


Figure 1.2: The electronic travel agency as dynamic architecture

As shown in this architecture model, different business partners usually run different system types. For example, while the fictive bank SHBC operates an R5/Banking system, its competitor BankOfEurope runs the XBank system; the same holds for the hotel and airline components. The travel agency does not only need to cope with these different component types but also with the different interfaces they adhere to, e. g., **Payment** vs. **Clearing**.

Even more challenging, the travel agency should be able to discover and communicate with new systems which are not known in advance, e. g., airlines or hotel companies which enter the market stage providing new services or attractive start-up offers. To satisfy the above demands, the global configuration of the travel application is subject to frequent runtime changes.

When designing an architecture for such a flexible system, we already have to take possible runtime changes of its architectural configuration into

account. These changes are called *dynamic reconfigurations*, and architectures capable of such changes are called *dynamic architectures*. While, in the beginning, software architecture research was mainly restricted to static architectures, the need for dynamic architectures caused growing attention to this topic in recent years.

Dynamic architectures raise several issues to be considered in architecture descriptions [123], among which are

- **Reconfiguration operations:** The four fundamental reconfiguration operations are the addition and removal of components and connectors, respectively. However, we also want to support more complex reconfiguration operations composed out of these four basic ones.
- **Reconfiguration triggers:** Runtime reconfigurations may be triggered by the current state of the system or a component, called *programmed reconfiguration* [40], or may be requested unexpectedly by the user or some unanticipated event, called *ad-hoc reconfiguration* [40].

In the case of programmed reconfigurations triggered by components themselves, the reconfiguration behavior has to be incorporated into the component’s overall behavior specification.

- **Reconciliation with component behavior:** We cannot assume that the concurrent execution of the architecture’s components may be suspended during reconfiguration. Thus, an important issue is how communication and reconfiguration behavior play together. For instance, we do not allow to remove a connector from a configuration while the connected components are still communicating with each other.
- **Synchronization:** Complex reconfiguration operations might affect several elements of the current configuration. As the default behavior in distributed systems is asynchronous, there is a need for *synchronization points* which allow all components and connectors involved in a reconfiguration to “agree” or “disagree”.

In the case of dynamic architectures, an architectural configuration captures a system state at a certain point in time only. In order to trace and simulate the evolution of the system from a given initial configuration, we require that an ADL does not only provide operational semantics for communication behavior but also for possible reconfiguration operations.

For a *complete* specification of dynamic architectures, one has to integrate the specification of architectural structure, component behavior, and architectural reconfiguration as shown in Fig. 1.3 [20]. In this work, we intend to

provide a description technique for dynamic architectures which combines all three dimensions at the semantic level using uniform operational semantics for both kinds of architectural behavior. We believe this to be advantageous because we can avoid additional constructs for composing separate formal methods for communication and reconfiguration. Moreover, a uniform formalism makes models easier to understand and manage, and we can better employ existing tool support and standard analysis techniques.

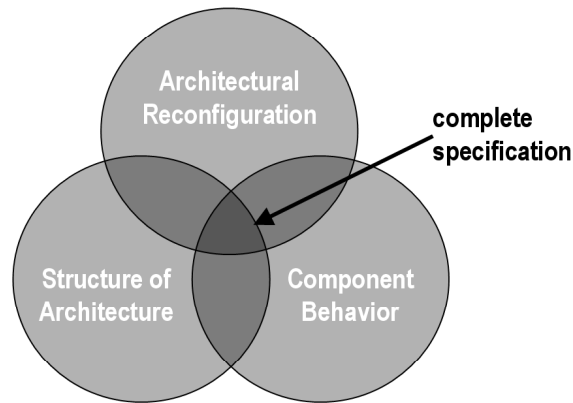


Figure 1.3: Complete specification of dynamic software architectures [20]

## 1.3 Platform-consistent development

When developing large software systems and reasoning about their runtime behavior, the architect has to take the desired target platform into account. In this section, we will motivate the notion of *platform consistency* and propose to develop platform-consistent architectures using explicit platform models in conjunction with a stepwise refinement approach.

### 1.3.1 Platform consistency

Distributed systems like the electronic travel agency are increasingly built on distributed object or component middleware platforms. The direct use of networking primitives or proprietary technologies is no longer a viable option because such approaches affect the maintainability and interoperability with other applications [80]. Instead, standardized middleware technologies

such as *CORBA*<sup>2</sup> and *Enterprise Java Beans*<sup>3</sup> are usually preferred for the development of such systems.

Thus, a crucial problem of the development of distributed systems is the selection of a suitable middleware platform. The right choice is driven by different factors among which are the realizability of both functional and non-functional requirements, the interoperability with platforms used by business partners, and the current business policy of the company.

The evaluation of these factors heavily depends on the capabilities and restrictions of the candidate platforms. In the context of dynamic software architectures, we are especially interested in the mechanisms they provide to enable component communication and dynamic reconfigurations:

- **Communication mechanisms** prescribe what kind of communication between components is supported by a platform. Examples include direct message exchange, remote procedure call, publish-subscribe policies [161], event broadcasting, blackboard communication [117], and so on. Each of these mechanisms has specific features. For our travel system, for instance, we require a flexible platform which allows the travel agency to communicate with (ideally) arbitrary business partner applications.
- **Reconfiguration mechanisms** prescribe which reconfiguration operations are supported by a platform. For our travel system, for instance, we require a platform which allows the travel agency to flexibly connect to third-party components according to incoming user requests and to automatically detect new market participants.

The travel agency's requirements on communication and reconfiguration mechanisms could be satisfied, e.g., by a platform that supports *service-oriented computing* [124]: In a service-oriented architecture (SOA), service providers make component interfaces public and allow business partners to access them as *services*. For this purpose, they publish service descriptions in standardized formats via third-party *discovery services*. With the help of specialized look-up functions, service requesters can implement automatic service discovery facilities which are able to find suitable services at runtime. Thus, we assume that our fictive travel agency project shall be realized on an SOA-enabled platform which supports loose component coupling and dynamic component discovery.

Anyway, whatever middleware platform is chosen, the software architect has to make allowance for its capabilities and restrictions when designing an

---

<sup>2</sup>[www.corba.org](http://www.corba.org)

<sup>3</sup>[java.sun.com/products/ejb/](http://java.sun.com/products/ejb/)



architecture which shall be realized on top of it. We call this requirement for architectural models *platform consistency*. In particular, the behavior specifications of an architectural model must not conflict with the mechanisms of the chosen platform. This means that only those communication and reconfiguration operations are allowed which can be implemented based on the underlying platform mechanisms. Otherwise, the desired behavior would not be realizable on that platform. In a SOA, for instance, one cannot connect to a new service unless the corresponding service description is available.

Moreover, the mechanisms a platform provides are accompanied by certain *topological constraints*. For instance, blackboard communication always requires a central blackboard component while direct message exchange can do without any central communication server. Or, service-oriented systems require explicit discovery services to publish and detect new components. To be platform-consistent, architectural configurations have to satisfy those platform-specific topological constraints, too.

**Platform models.** In order to formalize the notion of platform consistency, we require a *platform model* that precisely describes the specific characteristics of a certain platform type. It should at least comprise

1. a platform-specific vocabulary,
2. a set of topological constraints,
3. available communication mechanisms, and
4. available reconfiguration mechanisms.

The platform-specific vocabulary defines classifiers and roles known at the platform level like, e. g., blackboard, client, server, event dispatcher, discovery service, and so on. It should also define relationships between these classifiers, e. g., that a server can be *registered* at a discovery service. The remaining parts of the platform model, i. e., constraints and mechanisms, can then be phrased in terms of the platform-specific vocabulary.

A platform model should be reusable for different architectures. Hence, we have to distinguish the vocabulary of a platform model from the component and connector types of individual architectures. While the former define *application-independent* concepts like, e. g., **Client**, **Server**, etc., the second should be used to define *application-specific* component and connector types, e. g., **R5/Banking** or **XBank** in our example (Fig. 1.2).

Having defined a platform model for our target platform, we require a formal relationship between architecture and platform model which shows how



the architecture instantiates the platform-specific concepts and mechanisms. As depicted in Fig. 1.4, we distinguish between structural and behavioral instantiation:

- **Structural instantiation** reveals how the platform-specific vocabulary is used to classify the structural elements, i. e., the type definitions and architectural configurations. Based on such a classification, one can then check the satisfaction of topological constraints.
- **Behavioral instantiation** links architectural behavior to the corresponding platform mechanisms defined in the platform model. The execution semantics of the ADL has to be flexible enough to interpret the behavior definitions according to these mechanisms.

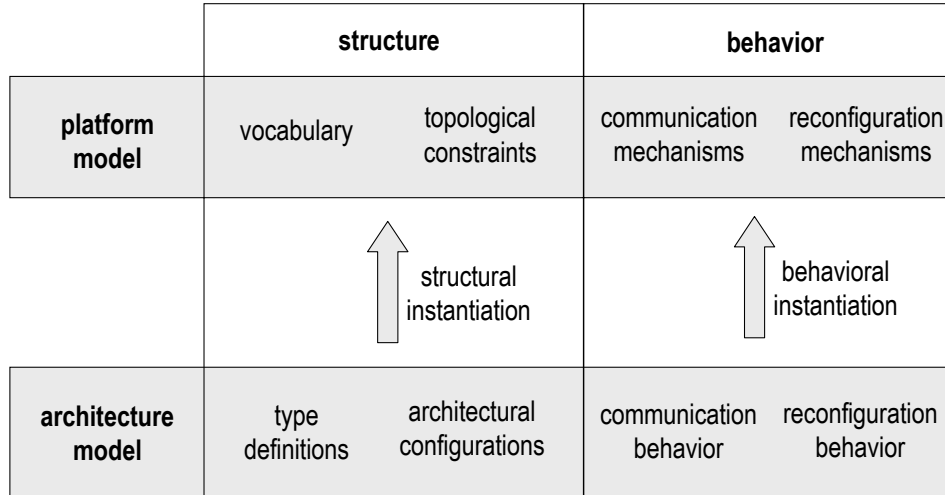


Figure 1.4: A platform model and its instantiation

### 1.3.2 Different levels of platform abstraction

The choice of a suitable middleware platform and its incorporation into a software architecture model is a difficult task because we have to come up with a model that equally conforms to system requirements and platform capabilities. In order to achieve such a platform-consistent architecture, we advocate a stepwise modeling approach. Following the “separation of concerns” paradigm, we consider it advantageous to deal with system requirements at a business-oriented level and to add details of the chosen target platform in a later stage of the development process.

The resulting business-oriented architecture can be used to communicate with customers and business experts while the platform-specific architecture can be discussed with IT experts and programmers. Moreover, the business architecture can remain unchanged while deriving several platform-specific variants for different platform candidates. This way, we can strengthen the reusability and portability of the business-oriented model.

Such a stepwise modeling approach is also put forward by the *Model-Driven Architecture* (MDA)<sup>4</sup> initiative of the *Object Management Group* (OMG). According to MDA, software development should start with a *platform-independent model* (PIM) which is driven by business processes rather than platform-related aspects.

Later, we add the missing details required to map to a certain target platform and convert the PIM into a *platform-specific model* (PSM). Eventually, the PSM shall be transformed into a working implementation on the selected middleware platform. The (future) vision of the MDA initiative is to automate the PIM-to-PSM and PSM-to-code conversions by software tools [81]. Provided such automated transformations, the approach promises a substantial productivity gain in software development [81].

In the MDA terminology, “a PIM exhibits a *certain degree* of platform independence so as to be suitable for use with a number of different platforms of similar type” [108]. This formulation reveals that platform independence is not meant to be absolute but always related to the assumptions we make about the computational infrastructure. In this sense, a platform-independent model is platform-specific with respect to an *abstract platform*, i. e., an abstract, technology-neutral virtual machine.

Generalizing this observation, we can conclude that every model of a stepwise approach like MDA belongs to a certain level of *platform abstraction*. At each level, one imposes more concrete assumptions about the underlying platform, formally defined by suitable platform models as introduced in the previous section. In the MDA terminology, a “platform model provides a set of technical concepts, representing the different parts that make up a platform and the services provided by that platform” [108]. Platform models can also be used to guide the necessary MDA model transformations.

Returning to our travel example, we require at least two different platform models: an abstract one for generic component platforms and a more concrete one for service-oriented platforms. The abstract platform model can be used as foundation of the business-oriented variant of our architecture concentrating on business-relevant, functional components as known from Fig. 1.1. For this purpose, it should define a general notion of component

---

<sup>4</sup>[www.omg.org/mda](http://www.omg.org/mda)

and a reconfiguration mechanism that allows components to “somehow” find and use each other.

The concrete platform model can be used as foundation of a more platform-specific architecture, now also taking into account middleware-related questions like *how* the components can find each other. As we decided to employ a service-oriented middleware like, e.g., Web Services, the platform model should define concepts like *service*, which is a special component exposing its functionality over a network to service requesters, and *discovery service*, which is a third-party component mediating between service requesters and providers. Further on, it should define the corresponding reconfiguration mechanisms for an automated discovery of new services.

Without question, there could also be alternative platform models representing other types of middleware platforms the travel system could be deployed on. Also, the service-oriented platform model could be further specialized into descriptions of vendor-specific middleware products. This way, we achieve a *hierarchy* of different platform models which comprises various paths for stepwise modeling (cf. Fig. 1.5).

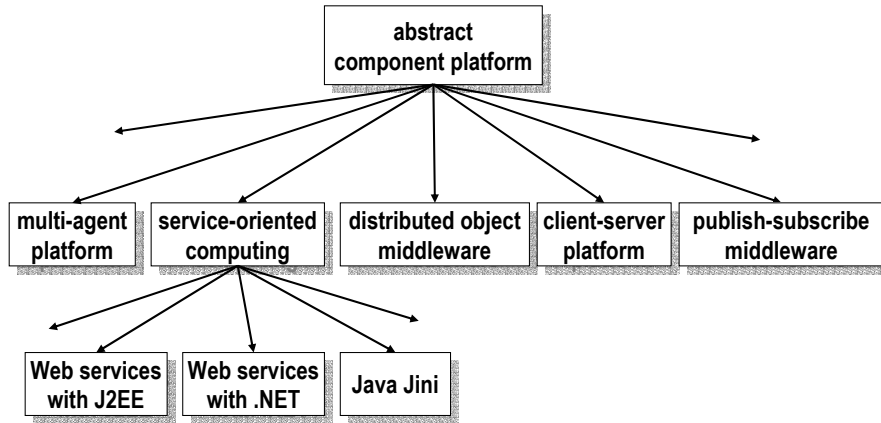


Figure 1.5: Hierarchy of platform models

When defining such a hierarchy of platform models in a top-down order, it is advantageous if previous platform models can easily be extended by more detailed vocabulary, constraints, and mechanisms for the next platform hierarchy level. For this reason, we aim at a specification technique that ensures a good *extensibility* of platform models.

However, given two consecutive platform models of the hierarchy, the question remains (1) how to derive an architecture for the concrete platform model from a given architecture for the abstract platform, and (2) how to

check if two given architecture variants, one for the abstract platform and one for the more concrete platform, are consistent to each other. We will elaborate on these problems in the next section.

### 1.3.3 Architecture refinement

The conversion of architectures from a high level of abstraction to a lower level is also called *architecture refinement*. Its two major challenges are (1) the automated derivation of concrete architectures from abstract ones (*refinement construction*) and (2) a formal consistency check for the two variants (*refinement check*).

**Refinement construction.** Refinement construction is similar to the problem of generating code from software models, which is still open research. Although there is a number of CASE tools with code generation capabilities, they are either restricted to certain application domains or limited to structural aspects generating code skeletons only. Only very few of them (e. g., FUJABA<sup>5</sup>) are able to produce entire applications including operation bodies. However, in order to produce meaningful code, these code generators require very precise input models which are already close to the code level.

Consequently, we cannot expect to generate implementation code from an abstract, business-oriented architecture model. Nevertheless, the challenge remains to derive at least a platform-specific model which reveals how business-relevant components can be deployed on the desired target platform.

In this thesis, we do not try to solve the refinement construction problem by providing a functional refinement operator that enables arbitrary automatic model transformations from a high level of platform abstraction to a lower level. Since abstract models are lacking information about the details that have to be inserted, we rather believe that this problem can only be solved by a semi-automated technique still involving user interactions. Even if we assume that a platform model provides all required platform-related knowledge and guides the transformation, the refinement process will still involve several degrees of freedom requiring user-specific design decisions.

For these reasons, we will not elaborate on automated refinement constructions but confine ourselves to some brief remarks on how the underlying model transformations could in general be specified. Instead, our main objective concerning architecture refinement are formal refinement checks which allow to compare a given pair of concrete and abstract architecture models.

---

<sup>5</sup>[www.fujaba.de](http://www.fujaba.de)

Nevertheless, refinement checks can to a certain extent also support refinement constructions. In an iterative refinement process, for instance, an engineer improves initially unsatisfactory refinements step by step. In every iteration, the engineer can apply a refinement check to detect remaining inconsistencies.

**Refinement check.** After completion of a (manual) refinement construction, we have to check if the resulting concrete architecture is indeed a refinement of the abstract architecture. This is true if the platform-specific architecture satisfies the same requirements as the business-oriented architecture. For this check, we require a formal criterion which allows us to compare both structure and behavior of the two architectures. Hence, we subdivide the problem into structural and behavioral refinement.

*Structural refinement* includes the preservation of all functional, business-relevant components at the platform-specific level. However, it is not sufficient to check the existence of corresponding components, but we also have to look for corresponding connections in the platform-specific topology which enable the components to carry out all business-relevant interactions.

*Behavioral refinement* includes the following two requirements:

1. **Behavior preservation:** A refinement has to preserve the complete business-relevant behavior at the platform-specific level. This means that all business-level scenarios of communication and reconfiguration operations can also be realized in the concrete architecture in terms of platform-specific communication and reconfiguration mechanisms.

Behavior-preserving refinement is important because it is the key factor for realizing the application's business processes on the desired target platform. For instance, if our travel agency can discover hotels offering accommodation in a certain place and book rooms there, then this business process should also be realizable in the platform-specific, service-oriented architecture. The same holds for all other processes and scenarios of the travel system.

2. **Observational substitutability:** Dually to behavior preservation, observational substitutability (sometimes also called *behavior reflection*) means that every behavior possible in the concrete architecture corresponds to a behavior in the abstract architecture. This way, the refined behavior can be used as a specialization, or substitution, of the original, abstract behavior.

The substitutability restriction guarantees that all safety constraints satisfied by the business-level architecture are also satisfied by the

platform-specific variant. For instance, if a client cannot book a journey without paying in the business architecture, then this scenario is also forbidden in the platform-specific architecture. In summary, “the concrete representation should not produce any externally-observable behavior that the abstract representation could not have produced” [48].

What makes behavioral refinement checks rather difficult is the fact that platform-specific mechanisms for communication and reconfiguration operations might be completely different from the abstract mechanisms assumed for the business-level architecture. Due to this divergence, it might be impossible to syntactically match abstract and concrete behavior definitions, but one has to decide behavior preservation and observational substitutability at a *semantic* level considering the actual effects of the invoked operations.

Another difficulty is the fact that sometimes different occurrences of the same abstract operation have to be refined differently, depending on the current configuration or the history of past operations. Consequently, a behavioral refinement check should allow for different, context-dependent refinements.

For instance, the creation of a connector between our travel agency component and an airline’s booking system is at the refined, service-oriented level only possible after the travel agency has retrieved a description of the flight booking service. However, in situations where the travel agency knows the description already, e. g., due to previous look-up actions, the required retrieval operation can be omitted.

For the sake of *reusability*, we are looking for a refinement criterion which is based on a relationship between the underlying abstract and concrete platform models rather than the specific architectures. A refinement relationship between platform models is advantageous because it can be reused to check the refinement for any pair of architectures conforming to these two platform models (see Fig. 1.6).

Eventually, we are aiming at a formalization of the refinement criterion and a corresponding checking algorithm. The intention behind is to employ software tools which can automatically decide whether a platform-specific architecture is a valid structural and behavioral refinement of the abstract architecture or not. Such a tool together with a powerful formalism for describing dynamic architectures and underlying platform models would certainly aid the stepwise, platform-consistent development of dynamic software architectures.

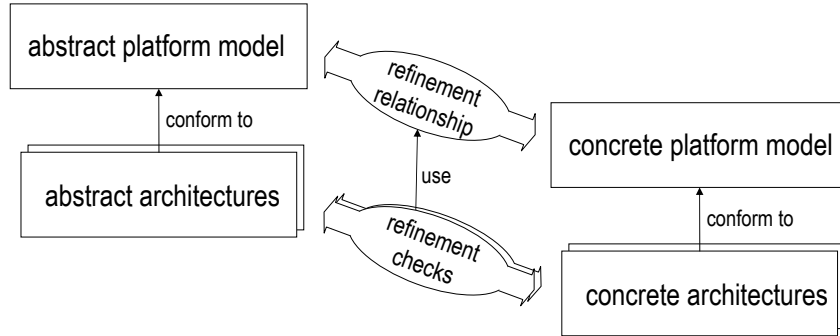


Figure 1.6: Reusable refinement relationship between platform models

## 1.4 Structure of the thesis

After Chapter 1 has provided first insights into the field of dynamic software architectures, motivated the need for suitable architecture description languages, and outlined the challenges of platform-consistent architecture development in a model-driven methodology like MDA, the rest of the thesis is structured as follows.

Chapter 2 at first summarizes the requirements for a platform-consistent architecture development we want to address in this thesis and then reviews related work against these requirements revealing existent achievements and shortcomings.

Chapter 3 provides an informal overview of the style-based approach we propose for platform-consistent architecture development, before the subsequent chapters describe the technique in more detail and with formal underpinning.

Chapter 4 contains a general introduction to the theory of graphs, graph transformation models, and graph transition systems as the formal method we use to underpin our approach.

Chapter 5 explains how to formally define architectural styles as graph transformation systems. In our approach, architectural styles are used as platform models describing the reconfiguration and communication capabilities of a certain (middleware) platform. The idea is illustrated along two examples: a platform-independent, component-based style and a platform-specific, service-oriented style.

Chapter 6 supplements the formal representation of architectural models as instance graphs of a certain architectural style by a more user-friendly modeling layer based on UML profiles.

Chapter 7 extends the style-based modeling approach by formal criteria

for checking whether a platform-specific architecture is a valid refinement of an abstract, business-level architecture. We provide an algorithm for automated refinement checks in terms of both behavior preservation and observational substitutability. This enables a stepwise and consistent development of dynamic software architectures from business requirements to platform-specific models.

Chapter 8 discusses tool support for our modeling and refinement approach and evaluates the suitability of existing graph transformation tools. Eventually, Chapter 9 concludes the thesis with a evaluation of the results against the requirements we started with and an outlook on future work.

**Bibliographical note.** Preliminary results of this work have already been published in [11, 12, 13, 14, 61, 63, 64].



# Chapter 2

## Survey of related work

After some brief notes about the history of software architecture research in general, we will now survey existing proposals from the software architecture community which also address (some of) the problems stated in the previous chapter. For this purpose, we will at first recollect the requirements from Chapter 1 and then evaluate to what extent the current state of the art satisfies these requirements and what deficiencies demand further research.

### 2.1 Some historical notes

In the late 1960s, software developers started to realize the increasing need for an engineering basis of their discipline. The term “*software engineering*” was first introduced at the famous Garmisch NATO conference in 1968 [116]. From then on, the term has been achieving popularity for describing processes, design activities, and tooling for software development.

During the 1970s and 1980s, software engineers mainly concentrated on the code level and structured programming. Their main design issues were the development and choice of suitable algorithms and data structures. At the same time, first languages and methodologies for data and software modeling were proposed, e. g., entity-relationship diagrams [24], structured analysis [102], state-charts [57], etc. However, at least in the beginning, this pioneer work was too separated from the programming world, and, besides, the two fields of database engineering and application programming did not find together.

Over the intervening years, the increasing complexity of software systems pushed the evolution of object-oriented modeling and programming languages which support better modularization and reuse of program code. Also, software engineers learned that an integrated view helps to reconcile

conflicting requirements. This led to a growing interest in integrated, multi-purpose modeling languages in the 1990s, with the UML being the most prominent example. In the second half of the last decade, model-driven development initiatives like MDA together with code generation techniques gained growing attention for bridging the gap between models at high levels of abstraction and those at a lower level down to program code.

During this time, *software architecture* as a new discipline “has emerged as a natural evolution of design abstractions, as engineers have searched for better ways to understand their software and new ways to build larger, more complex software systems” [144].

In the second half of the 1990s, the field of software architecture research matured as a software engineering discipline of its own [143], indicated by a significant number of publications and books, starting with the one by Shaw and Garlan [144], as well as by dedicated software architecture tracks at all important software engineering conferences and a special series of “Working IEEE/IFIP Conferences on Software Architecture (WICSA)” [101].

While the area of static architecture descriptions was relatively well researched at that time, the need for flexible and adaptive systems drew more and more attention to the specification of *dynamic architectures*.

This new research direction was heavily influenced by earlier work about *configuration programming* [84] for distributed systems. In configuration programming, formal configuration languages like, e. g., *Gerel* [40], are developed to specify the configuration and reconfiguration of a distributed system. Such configuration languages are then used in conjunction with ordinary programming languages which define the component behavior. In fact, some of the approaches for configuration programming have later evolved to be used for the specification of dynamic software architectures [20].

## 2.2 Requirements

In the following paragraphs we will recollect the requirements motivated in Chapter 1 (note the corresponding page references). These requirements have to be satisfied in order to provide a formal but user-friendly technique for a model-driven, platform-consistent development of dynamic software architectures. In subsequent sections, the resulting requirements list will serve us as an evaluation framework for existing related work and, later on, also for our own proposal.

Although there are, without doubt, many interesting and challenging problems that go beyond our objectives, we deliberately decided to limit the scope of this thesis to the requirements listed below. For instance, we

do not investigate the modeling of *transactional behavior*, which is required in order to group operations into atomic sets [123], and the definition of *exceptional behavior*, which is required to handle exceptions that occur during communication or reconfiguration operations.

### 2.2.1 Requirements for architecture descriptions

At first, we recollect features an ADL should provide for the design of dynamic software architectures. According to the objective of the thesis, we put a special focus on the description of dynamic reconfigurations. A broader classification framework and survey of ADLs can be found, e. g., in [107].

(1) **Type definitions:**

The ADL should allow to define types of architectural elements like components and connectors in order to encapsulate functionality, interface information, and behavior definitions into reusable building blocks which can be instantiated multiple times. This includes the definition of provided and required interfaces or ports which are assigned to the individual component and connector types. (p. 4)

(2) **Communication behavior:**

In the frame of the type definitions, the ADL should allow to describe how instances of these types communicate with each other. Such communication behavior should either be directly associated to component and connector types or indirectly using interfaces or ports with associated behavior definitions. The behavior descriptions have to define which communication is taking place between which elements and at which time. As a basic requirement, the description language has to provide appropriate constructs to define the order of actions such as structured programming constructs (sequence, branching, loop, etc.) or transition rules. (p. 4)

(3) **Runtime configurations:**

As we are interested in the runtime architecture of a system, the ADL should allow to describe architectural configurations consisting of various component and connector instances and additional resources like data elements. As we consider components and connectors changing their state due to communication behavior and configurations being altered due to reconfiguration operations, a configuration represents the system state at a particular point in time only. (p. 5)

**(4) Reconfiguration operations:**

For the description of dynamic architectures, the ADL should allow to specify reconfiguration operations of various degrees of complexity from simple component/connector creation/removals up to more complex reconfigurations. (p. 9)

**(5) Programmed reconfigurations:**

Besides the definition of ad-hoc reconfigurations, the ADL should also support programmed reconfigurations. For this purpose, it should allow to define the invocation of reconfiguration operations by components and connectors and to integrate such invocations into their behavior definitions. (p. 9)

**(6) Reconciliation of communication and reconfiguration:**

In order to avoid inconsistent behavior definitions, the ADL should support the reconciliation of communication and reconfiguration behavior. For this purpose, appropriate language constructs should prevent errors like, e. g., the removal of a connector which is still used for an ongoing communication. (p. 9)

**(7) Asynchronous behavior with synchronization points:**

Besides a concurrent, asynchronous component behavior within a distributed architectural configuration, the ADL should allow to define synchronization points which are required for communication or reconfiguration steps involving several participants at once. (p. 9)

**(8) Operational semantics:**

The ADL should be equipped with operational semantics to allow the execution and simulation of the concurrent behavior of all component instances in a certain configuration. As we want to deal with dynamic architectures, such execution semantics should not be limited to the communication happening in a fixed configuration, but should also simulate changing configurations according to reconfiguration operations. (p. 6)

**(9) Analysis capabilities:**

In order to analyze a dynamic architecture, the formalism should be open to sophisticated analysis based on its operational semantics. For instance, we want to decide if a state with certain desired or undesired properties is reachable during system execution or not. (p. 9)

**(10) Uniform formalism:**

In order to apply sophisticated, technically mature analysis tools, it is

advantageous to base the ADL and its semantics on a uniform formal method which covers both communication and reconfiguration behavior. (p. 9)

(11) **Usability:**

Since formal methods are often difficult to use by practitioners, the ADL should be equipped with a user-friendly and well-known notation like, e.g., UML. This notation should serve as a front-end hiding the underlying formalism from the user. (p. 7)

## 2.2.2 Requirements for platform descriptions

In order to create platform-consistent architecture models, we require a platform description language which allows to describe platform types by corresponding platform models. Moreover, such platform models have to be respected by architecture descriptions. This leads to additional requirements for the ADL, too (cf. Section 1.3.1).

(12) **Vocabulary:**

The platform description language should allow to define a platform-specific vocabulary with classifiers like entity types and relationships between them. The elements of the vocabulary should be distinguished from the component and connector types of architecture descriptions and rather comprise application-independent platform constructs. (p. 12)

(13) **Topological constraints:**

The platform description language should also allow to add topological configuration constraints to a platform model. These constraints restrict the way elements classified by the vocabulary can be assembled to architectural configurations. (p. 12)

(14) **Communication mechanisms:**

We require a formalism to define communication mechanisms as generic operations over the platform vocabulary. The mechanisms should describe how components can communicate on the respective platform, e.g., by direct messages or through a special mediator. The specification should also include preconditions, e.g., in order to ensure that involved entities are ready to perform the respective operation. As with the other parts of a platform model, the mechanisms should be kept application-independent and reusable. (p. 11)

**(15) Reconfiguration mechanisms:**

The platform description language should enable to specify the reconfiguration operations a platform supports. In order to allow for many different platform types with diverse reconfiguration policies, its expressiveness should cover *arbitrary* reconfigurations. Similar to the communication mechanisms, the reconfiguration mechanisms should be defined as generic operations over the platform vocabulary and kept independent of a concrete application. (p. 11)

**(16) Extensibility:**

All parts of a platform model should be easily extensible in case that new constructs, constraints or mechanisms have to be integrated. This feature is especially important for augmenting a platform model with further details in order to derive a more specific model at a lower platform hierarchy level. (p. 15)

**(17) Structural instantiation:**

The ADL should provide a possibility to instantiate the structural constructs of a platform model. For this purpose, the elements of an architecture description should be mapped to the platform-specific vocabulary. Then, one can, e.g., check if the architecture satisfies the topological constraints of the platform model. (p. 13)

**(18) Behavioral instantiation:**

The ADL should also instantiate and respect the behavioral constructs of a platform model. In order to express architectural behavior in terms of platform services, the individual communication and reconfiguration actions should refer to corresponding mechanisms defined in the platform model. Moreover, the execution semantics of the ADL should be flexible enough to interpret the behavior definitions according to the specified platform mechanisms. (p. 13)

### 2.2.3 Requirements for architecture refinements

In order to realize a stepwise refinement approach for dynamic architectures across different levels of platform abstraction, the formal framework has to be extended by suitable notions of architectural refinement (cf. Section 1.3.3). For this purpose, we want to address the following requirements related to refinement checks.

**(19) Structural refinement:**

We require a structural refinement criterion in order to check if all

business-relevant and functional entities of an abstract configuration are preserved at the concrete level. (p. 17)

(20) **Behavior preservation:**

We require a behavioral refinement criterion which guarantees that the abstract behavior is preserved at the more platform-specific level and that all abstract communication and reconfiguration scenarios can also be realized in the refined architecture. (p. 17)

(21) **Observational substitutability:**

Moreover, the behavioral refinement criterion should guarantee observational substitutability, which means that every behavior possible in the concrete architecture belongs to a correspondent behavior in the abstract architecture. (p. 17)

(22) **Semantic refinement:**

The refinement criterion should also be checkable if the communication and reconfiguration mechanisms applied at the two levels of abstraction differ to a very large extent. For this reason, the behavior definitions of the two architectures should be compared at a *semantic* level considering the actual effects of the invoked operations rather than based on a *syntactic* matching of abstract and platform-specific operations. (p. 18)

(23) **Context dependency:**

If a certain communication or reconfiguration operation occurs several times in the abstract architecture's behavior definitions, the behavioral refinement should allow for different, context-dependent refinements of these occurrences, because the right refinement can depend on both the current configuration and the history of past actions. (p. 18)

(24) **Reusability:**

A refinement relationship should be defined in terms of the two involved platform models rather than in terms of individual architectures so that it can be applied to any two architectures conforming with these platform models, respectively. (p. 18)

(25) **Checking algorithm and tool support:**

In order to *automatically* decide whether a platform-specific architecture is a valid refinement of an abstract architecture or not, we seek for an algorithm and tool support which implements the formal refinement criteria. (p. 18)

## 2.3 Existing approaches and open problems

From the research history sketched at the beginning of this chapter, a number of description techniques, languages, and formal methods has emerged which are relevant for our work. In this section, we will evaluate related approaches against our requirements<sup>1</sup> in order to provide an overview of the current 'state of the art' and to uncover remaining challenges. Analog to the requirements list from Section 2.2, the survey is subdivided into three parts, namely

1. the modeling of dynamic architectures with formal methods,
2. the consideration of platform-specific aspects in architecture descriptions, and
3. architecture refinement techniques which could be employed for step-wise development across multiple levels of platform abstraction.

All of the three parts end with a table summarizing the evaluation results.

### 2.3.1 Formal methods for dynamic architectures

Comparing the three fields we want to survey, the largest amount of contributions has already been produced for modeling dynamic architectures. Consequently, a number of various description languages has been proposed, together with a bunch of formalisms for their operational semantics.

The following summary of formalisms for dynamic architectures, grouped by the underlying formal method, is partially based on a more extensive survey which has recently been published by Bradbury et al. [20, 21]. However, at this point, we are especially concerned with the requirements listed in Section 2.2.1.

**Partially ordered event sets.** RAPIDE, one of the first ADLs, has been developed by Luckham et al. [94, 95] in the first half of the 1990s. RAPIDE considers component types as reusable entities, but not connectors (1). Instead, a configuration (3) links component instances by *in-line* connections.

RAPIDE is an event-based language, i. e., the execution semantics of a described system is expressed by a history of events. For this purpose, every component in a RAPIDE architecture is equipped with a rule-based behavior specification defining which events a component generates in reaction to certain incoming events (2).

---

<sup>1</sup>The respective cross-references printed in italics refer to the enumeration in Section 2.2.



Due to its operational semantics, a RAPIDE model can be simulated by a *poset browser* yielding a partially ordered set (poset) of events generated and received by the modeled components. The partial ordering reveals causal dependencies among these events. Luckham et al. argue that these partial orders are better suited to represent concurrent behavior than event traces (8)(9).

In contrast to most other ADLs in the early 1990s, RAPIDE already shows first moves towards dynamic architectures. It does not allow arbitrary reconfigurations of the topology but, at least, accounts for runtime creation of new components and for dynamic connections which can flexibly handle changing numbers of participants (4).

Only in a later extension [155], the explicit creation and removal of both components and connectors is considered under the notion of *execution architecture*. The authors introduce special purpose events for each of the basic reconfiguration operations. However, we miss an explanation of how these execution architectures are combined with their previous event-based approach for communication behavior (5)(6).

**Process algebras.** Starting in the late 1970s, process algebras have been developed as a formal description technique for complex computer systems with communicating, concurrently executing components [16]. Their main advantages are twofold: First, they enable compositional modeling which allows to compose large models out of multiple concurrent processes including necessary synchronizations (7). Second, they are equipped with formal, denotational semantics (8) translating process models into labeled transition systems that are open to further analysis (9) based on *model checking* [25, 27].

Due to these charming features, a number of well-known ADLs apply certain variants of process algebras as formal method to specify the behavior of architectural components. For instance, Allen and Garlan propose their ADL *Wright* [3, 5, 6] which is based on Hoare's process algebra CSP (Communicating Sequential Processes [72]). Wright uses CSP processes in order to formalize communication behavior (2) which is associated to the interfaces of component and connector types (1). Based on model checking with the FDR toolset<sup>2</sup>, one can then prove that all connected interfaces of a certain configuration (3) are compatible and, e. g., do not deadlock.

Later, Allen and Garlan extend Wright by constructs for dynamic reconfiguration, called *Dynamic Wright* [4]. For this purpose, they introduce operators for the creation and removal of components and connectors which can be composed into more complex reconfiguration operations. Since the semantics of the reconfiguration operators are expressed by translations into

---

<sup>2</sup>[www.fsel.com](http://www.fsel.com)

plain CSP, Dynamic Wright is a uniform approach (10).

Despite the reconfiguration operators, the required support for arbitrary reconfigurations (4) is only partially solved. The problem is that CSP is inherently limited to a static configuration of processes. As a workaround, the authors had to restrict the architectural dynamism to a *finite set* of configurations which have all to be known in advance. Thus, the actual effect of a reconfiguration action is to select the proper part out of the predefined CSP expressions so that the interactions conform to the new configuration [4].

The reconfiguration operators are not directly inserted into the components' CSP processes but are executed by a special "configurator" component. By communicating with the configurator, the other components can trigger desired reconfigurations (5).

Dynamic Wright does not provide any feature for the reconciliation with component behavior; the architect has to ensure manually that no inconsistent reconfiguration behavior is defined (6).

The restriction of CSP to static configurations does not hold for another process algebra, namely Milner's  $\pi$ -calculus [111]. The  $\pi$ -calculus is a *mobile* calculus specifically developed for communicating systems with changing structure. It describes a system as a set of independent processes which communicate via named channels. Processes are not directly named but can be accessed by channel names. Processes can be replicated, and channel names can be transmitted to other processes representing the creation of a new link.

A well-known ADL using the  $\pi$ -calculus as semantics is *Darwin* developed by Kramer, Magee, et al. [97, 99]. Since Darwin was originally designed as a configuration language for a distributed programming environment called *Regis* [98], it focuses on the correct configuration of component instances and the bindings between provided and required services. As with other configuration languages, the authors deliberately excluded the incorporation of component behavior specifications, although expressible with the  $\pi$ -calculus.

Darwin's denotational semantics into the  $\pi$ -calculus represents components as processes and connections as channel names. As a consequence, connectors are not treated as first-class entities (1). Due to the expressiveness of the underlying  $\pi$ -calculus (process replication and channel name transfers), the language supports the replication of component instances and changes of the current bindings. However, it is still not possible to express arbitrary reconfiguration operations like, e. g., component removal (4) [21].

Only later, Kramer and Magee extended their approach by behavior specifications for the individual components of an architecture [86, 87]. For this purpose, they specify components and their behavior by finite state processes (FSP [100]), a simple compositional process algebra whose semantics

is expressed in terms of labeled transition systems. While, in contrast to the  $\pi$ -calculus, FSP is much easier to understand, the combination of two different process algebras affects the uniformness of the approach (10).

Unfortunately, the authors do not elaborate how the two approaches are integrated concerning dynamic reconfigurations: Darwin's reconfiguration operators cannot be used in FSP. Instead, FSP can only simulate reconfiguration operations. For instance, component removal is simulated by switching its behavior to an idle state and component creation by reactivating such deactivated processes. As a consequence, the number of possible configurations is a priori bounded, similar to Wright. However, the direct insertion of reconfiguration operations into FSP processes facilitates programmed reconfigurations, at least for the restricted number of reconfiguration operations (5).

For reconciling the FSP reconfiguration operations with the general component behavior (6), Kramer and Magee follow an earlier paradigm introduced in [85] which states that a component has to be in a *quiescent* state before it can participate in a reconfiguration. Thus, the behavior specifications of the individual component types have to distinguish between active and quiescent states of the component's computation. Only under this restriction, communication and reconfiguration behavior can be combined, and the resulting overall behavior can be animated and analyzed by their *Labelled Transition System Analyzer* (LTSA)<sup>3</sup>.

In a related approach, Canal et al. propose *LEDA* [22], an ADL which makes use of the  $\pi$ -calculus, too. In this case, also behavior specifications of components are expressed by  $\pi$ -calculus processes, which makes the approach more uniform (10). On the other hand, since the bare  $\pi$ -calculus is not only used as underlying semantics but even for the concrete notation, the usability of the language is very limited (11).

LEDA defines reusable types for components which can be instantiated and composed to form architectural configurations (3). However, connections are not typed as connectors but defined within configurations, similar to Rapide and Darwin (1). With the help of a  $\pi$ -calculus interpreter, a LEDA model can be simulated and checked for local behavior compatibility of connected components (9).

LEDA's support for reconfiguration (4) is bounded by the same limitations mentioned above for Darwin. The direct insertion of reconfiguration operations into the  $\pi$ -calculus processes makes programmed reconfigurations possible within the given bounds (5). However, there is no support for a reconciliation of reconfigurations with the remaining behavior (6).

---

<sup>3</sup>[www-dse.doc.ic.ac.uk/concurrency/](http://www-dse.doc.ic.ac.uk/concurrency/)

From the above observations, we can conclude that process algebras are a proper formal method for the definition of component behavior in terms of communicating processes. On the other hand, they show clear limitations when it comes to the specification of structure and structural change as it is required for dynamic architectures. Besides, their syntax is very low-level and comparable to a programming language. For these reasons, another formal method has gained more and more attention, namely graphs and graph transformations.

**Graph transformations.** A quite natural way to specify the structure of a software architecture is to use *graphs*. Changes to the graph can be described by *graph transformation rules*. Each rule consists of a left-hand side and a right-hand side graph. The application of a rule changes a graph by replacing an occurrence of the rule's left-hand side by a copy of the rule's right-hand side (see also Chapter 4).

Le Métayer [91] describes architectural configurations by graphs where nodes represent computational entities (components) and edges represent communication links (connectors). Node and edge labels, formally defined as unary and binary relations, are used to assign component and connector types (1). The set of allowed graphs is constrained by a *context-free* graph grammar. The problem with context-free graph grammars is that they cannot express all possible topologies (3). For instance, they exclude cyclic structures like rings of arbitrary size.

The approach considers only components, but no connectors (1), as active elements with a description of their behavior using a CSP-like notation. As an extension to CSP, the behavior descriptions contain generic communication commands which inherently respect the topology of the current configuration (2). Le Métayer [91] provides a denotational semantics for this language in terms of labeled transition systems.

Reconfiguration operations are defined by conditional graph transformation rules. By applying such a transformation rule, the graph representing the current configuration is partially rewritten. Unlike the graph grammar, the transformation rules are not context-free and, thus, allow the description of arbitrary reconfiguration operations (4).

The rules are applied non-deterministically simulating ad-hoc reconfigurations. Additional preconditions concerning the current topology and component states can be used to restrict reconfigurations to certain phases of the component behavior only (6). However, the explicit invocation of reconfiguration operations by components as required for programmed reconfiguration is not possible (5).

Due to the combination of two different formal methods, namely process algebra for communication behavior and graph transformation for reconfiguration behavior, the approach is not uniform (10), its semantics is complicated by additional constructs to integrate the two worlds (8), and the inability to apply standard analysis tools limits its analysis capabilities (9).

A different kind of graphs, namely *hypergraphs*, are used by Hirsch et al. [68, 69]. In hypergraphs, edges represent components while nodes represent connectors. The reason is that hyperedges can connect more than two nodes. Similar to the Le Métayer approach, labels are used to represent component and connector types (1), but also the different states of a component at runtime. Again, conditional graph transformation rules are used to describe dynamic changes of the current configuration.

Different to the Le Métayer approach, Hirsch et al. do not use any process algebra to specify the component behavior but, instead, define additional graph transformation rules for modeling the communication patterns of the architecture. For this purpose, they insert additional information about incoming or outgoing signals or messages into the graph which can be modified by communication rules (2).

We consider this uniform approach to both communication and reconfiguration a promising direction because the semantics of the model can completely be defined by graph transformation theory (8). This way, “by analyzing the derivation tree it is possible to have all the computations of the system allowing the verification of properties of the architecture, like for example, deadlock” [69] (9).

The graph transformation rules for reconfiguration operations can be constrained over the current state of involved hyperedges as given by their current label. The labels of hyperedges are in turn changed by transformation rules for communication operations. This way, the application of communication and reconfiguration operations can be coordinated through this state information (6).

However, the satisfaction of a rule’s preconditions does only *enable* but not *enforce* its application. The actual rule application happens non-deterministically. As a consequence, we cannot model programmed reconfigurations with components invoking reconfiguration operations themselves (5).

An earlier weakness of Hirsch et al.’s approach was its restriction to *context-free* transformation rules for both communication and reconfiguration rules. “With this type of rules, two separate edges cannot be bound later, so for example, an architecture instance that has a pipeline style cannot be converted, after its creation, into a ring” [69]. In the meantime, however, they invented concepts for synchronized rule applications [70, 67]. This

way, they are now able to compose arbitrarily complex configurations and reconfigurations out of several decentralized transformations (3)(4)(7). On the other hand, the extension by so-called synchronization algebras affects the uniformness and the usability of the formal method (10).

Another approach to model both communication and reconfiguration behavior using graph transformations are *distributed graphs* and *distributed graph transformations* proposed by Taentzer et al. [150]. Distributed graphs consist of two types of graphs, namely *network graphs* and *local graphs*.

A network graph represents the architecture of the system with nodes for components and edges for connections (3). Component can be typed using edge labels, but connectors are not modeled as first-class elements (1).

Every node in the network graph has a corresponding local graph representing the current state of the component. The nodes of the local graph typically represent data and interface elements such as objects in an object-oriented system. However, we fear that the usage of local graphs describing the internal structure of a component could break the encapsulation required for the architectural level of abstraction.

Distributed graph transformation rules span both the network and the local level: For every network node in a distributed rule, the rule contains its local graph, too. This way, distributed transformation rules can be used to model internal computations, communication between components (2), and changes at the network level (4).

The inclusion of local graphs allows to constraint the application of network level reconfigurations by conditions on the current local component state. For instance, one can ensure that components to be disconnected or removed are in a quiescent state (6). However, similar to the Hirsch approach, the explicit invocation of reconfiguration operations, programmed into the component behavior, is not possible (5).

In contrast to the Hirsch et al. approach, Taentzer’s distributed graph transformations are not restricted to context-free transformations. Thus, a rule can simultaneously affect several nodes of the network graph, which also allows to realize synchronized behavior (7).

In [149], Taentzer provides operational semantics for distributed graph transformation rules based on a well-established algebraic approach in Category theory (8). Graph transformation rules with this semantics can be simulated and analyzed using the AGG tool<sup>4</sup>. However, AGG does only support “flat” transformation rules; the support for distributed transformation rules is under development (9).

---

<sup>4</sup>[tfs.cs.tu-berlin.de/agg/](https://tfs.cs.tu-berlin.de/agg/)

Another graph transformation-based approach, formalized as an algebraic framework in Category theory, is proposed by Wermelinger and Fidadeiro [156, 157]. They represent architectures as graphs and components as nodes, too (3). However, different to Taentzer et al., they also model connectors as nodes of the graph, and they define the behavior of components and connectors by abstract programs in the algebraic program design language COMMUNITY [44].

A COMMUNITY program defines input and output variables as well as private and shared actions for interactions with other components (2). The concept of actions being shared with other components allows for synchronizations of different components (7). The behavior definitions are associated to reusable type definitions for both components and connectors (1).

Connectors are special programs that glue various *roles*. Each role is defined by its own COMMUNITY program. Components must *refine* one of the roles to be accepted for that connector. Such a refinement relationship between COMMUNITY programs is formally given by a so-called *superposition morphism*, an algebraic relationship which defines a mapping between the variables and shared actions of two programs. Consequently, the edges of a configuration graph correspond to such superposition morphisms linking components and connectors.

The main advantage of the algebraic representation of a configuration is the possibility to automatically compute a composite program, in Category theory terminology called *colimit*, which represents the global behavior of a given configuration and allows its simulation [157]. A suitable analysis tool called COMMUNITY Workbench is currently under development [122] (9).

Dynamic reconfiguration operations are specified as graph transformation rules (4). Similar to Le Métayer’s approach, the reconfiguration rules are constrained over the current state of the component instances as indicated by their variable values. This way, it can be ensured that components are in a consistent state when being involved in a reconfiguration (6).

Since an architectural configuration is given as a graph of algebraic programs and morphisms between them, an algebraic graph transformation approach – similar to the one used by Taentzer et al. – can be applied to define operational semantics for reconfiguration operations [156]. However, a joint operational semantics for communication and reconfiguration is missing (8).

In [158], the approach is extended by a script language which expresses composite reconfiguration operations using high-level constructs like sequencing, choice, and iteration (11). Ad-hoc reconfigurations correspond to user invocations of such scripts, whereas programmed reconfigurations are automatically invoked whenever certain preconditions are satisfied. But, components can still not control the programmed reconfigurations themselves (5).



Table 2.1: Comparison of approaches to dynamic architecture description

Requirement	type def.	communication beh.	configurations	reconfiguration op.	programmed reconf.	reconciliation	synchronization	op. semantics	analysis	uniformness	usability
Approach	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
Rapide	o	+	+	o	?	?	?	+	+	+	-
Dyn. Wright	+	+	+	o	+	-	+	+	+	+	-
Darwin	o	+	+	o	o	+	+	+	+	-	o
LEDA	o	+	+	o	+	-	+	+	+	+	-
Le Métayer	o	+	o	+	-	+	+	+	o	-	o
Hirsch et al.	+	+	+	+	-	+	+	+	+	-	-
Taentzer et al.	o	+	+	+	-	+	+	+	o	+	o
Wermelinger et al.	+	+	+	+	o	+	+	o	+	o	o

- = not satisfied; o = partially satisfied; + = completely satisfied

**Summary.** The results of our evaluation are summarized in Table 2.1. Obviously, process algebras and graph transformation are the two dominating formal methods for the specification of dynamic architectures.

The strength of process algebras in composing concurrent behavior of several architectural components and connectors is traded off against their weakness when it comes to the specification of arbitrary reconfiguration operations (4) and the reconciliation of these operations with the general behavior (6). Even the  $\pi$ -calculus provides only limited support for such reconfigurations.

On the contrary, graph transformations have proved their strength for the specification of structural aspects, i. e., architectural configurations (3) and arbitrary reconfigurations (4). Moreover, with the help of additional application conditions, it is relatively easy to reconcile the graph transformation rules for reconfigurations with the remaining communication behavior (6).

Le Métayer and Fiadeiro et al. combine graph transformations with process algebra-related languages for specifying the component behavior, what affects the uniformness of their approaches (10). On the contrary, Hirsch and Taentzer show how graph transformation rules can be used to also express



communication behavior. Their assumption that the expressiveness of graph transformations is sufficient for a complete specification of dynamic architectures is reinforced by existing work using graph transformation rules to define the semantics of process algebras like the  $\pi$ -calculus [83].

Another shortcoming observed for all considered graph transformation-based approaches, however, is the fact that all enabled rules are applied non-deterministically without guarantee for their actual application. This contradicts the intention of programmed reconfigurations where components should be able to explicitly invoke and control reconfiguration operations themselves (5). Thus, more research effort has to be put into this issue in order to increase the suitability of graph transformation theory for the specification of dynamic architectures.

A remaining weakness of the existing proposals is their limited usability (11). Due to their inherent visualization, the graph-based description languages are better to read and understand than bare process algebras. However, so far, none of them provides high-level front-end notations like, e.g., UML which could further raise their acceptance by practitioners.

### 2.3.2 Platform awareness in architecture descriptions

As outlined in Section 1.3.1, architecture descriptions have to conform to the capabilities and restrictions of the target middleware platform. This insight has only been sparsely recognized within the software architecture community so far. Consequently, there are only very few proposals for explicit platform description languages.

Even in the context of the MDA initiative, the formulation of appropriate platform models guiding transformations from PIM to PSM is still an open question. Nevertheless, we will now outline the existing ideas and evaluate them against our requirements from Section 2.2.2.

**Platform-specific ADLs.** As a trivial solution, special-purpose ADLs have been proposed to describe architectures for specific target platforms. For instance, the languages C2SADEL [105] and the above mentioned ADL *Darwin* [97] fall into this category because they have been specifically developed for the middlewares C2 [151] and Regis [98], respectively.

However, platform-specific ADLs can only be used if the platform they have been designed for is chosen as target platform. Therefore, the architect would have to translate his model into a completely different ADL whenever he wants to evaluate the suitability of a different platform or whenever he wants to port the model to another level of platform abstraction.

**Platform-specific connector implementations.** A more general strategy to incorporate middleware-related aspects into architectural design is pursued by Medvidovic et al. [31, 103, 104]. They focus on connectors that encapsulate the communication between components. As communication is usually based on the underlying middleware platform, connectors are considered as abstractions for the platform-specific communication mechanisms (14).

Medvidovic et al. propose to design an application’s architecture at first in an abstract way. After all components and their topology are known, one or more middleware platforms are chosen to implement the connectors. Previously developed middleware-based connectors can be reused if they own the desired properties. This way, at least concerning connector types, a kind of platform-specific vocabulary is created (12). And, extensibility is given because new connector types can simply be added to the collection (16).

However, Medvidovic et al. provide a partial solution to platform consistency only because they restrict the role of middleware to the implementation of connectors; they neither consider any platform-induced classifiers for components and other elements (12), nor topological constraints (13), nor available reconfiguration mechanisms (15).

**Platform-specific architectural styles.** Another approach which leads to the possibility to use a *single* ADL for architectures on various platform types or at different levels of platform abstraction has been proposed by Di Nitto and Rosenblum [118]. They recommend to capture the assumptions and constraints imposed by a middleware platform as a *middleware-induced architectural style*. Similar to our objective, they intend to “guide designers in the definition of an architecture compliant with a pre-selected middleware infrastructure” or “support designers in the identification of the most suitable middleware infrastructure for a specific architecture” [118].

The notion of *architectural style* is a general software architecture concept which is used to define families of related architectures and their common properties [2, 49]. Prominent examples of architectural styles are, among others, layered style, pipe-filer style, client-server style, blackboard style, etc. [144].

A style is related to the concept of *patterns* which is employed in software engineering whenever reusable solutions can be provided for recurring problems. For instance, in object-oriented modeling and programming, so called *design patterns* [47] are used to capture approved design experience for recurring problems and to help engineers solving similar cases. The two terms can be distinguished as follows: “Styles generally provide guidance and analysis for building a broad class of architectures in a specific domain

whereas patterns focus on solving smaller, more specific problems within a given style” [112].

In the classical understanding, an architectural style prescribes a fixed *vocabulary* of architectural elements and *configuration constraints* formulated over this vocabulary [49]. The vocabulary defines logical concepts like, e.g., layers in layered architectures, as well as categories or roles for architectural elements like, e.g., client and server roles for client-server architectures. The vocabulary also defines relationships among its elements, possible property assertions, and some semantic interpretation. The configuration constraints restrict the topology of possible architectural configurations. An architecture description which adheres to a style by using its vocabulary and satisfying its configuration constraints is also called *member* or *instance* of the style.

Although this classical notion of architectural style does not take any patterns of communication or reconfiguration behavior into account, we can observe parallels between an architectural style capturing platform-specific assumptions and what we require as a platform model (see Section 2.2.2). Thus, when claiming that an architecture which shall be implemented on a certain middleware platform should adhere to the style induced by that platform, Di Nitto and Rosenblum anticipate our notion of platform consistency.

The usage of architectural styles as platform models does not contradict their original purpose to define families of related architectures, because establishing platform consistency for different applications deployed on the same platform type requires very similar tasks: the same vocabulary is used, the same topological constraints have to be satisfied, and the same platform mechanisms can be used only. Therefore, concerning platform-specific aspects, the resulting architectures are very similar as well, what suggests to subsume them under a single architectural style.

In their work, Di Nitto and Rosenblum do not propose a new language for defining middleware-induced architectural styles, but they survey the capabilities of some existing ADLs to allow the definition of such styles. From their case study they derive a collection of additional requirements for ADLs including the possibility to define

- architectural styles with component and connector types and a mechanism for instantiating a style in the definition of an architecture,
- topological constraints that must be respected by any instantiation of the style,
- connectors, and
- the behavior of components and connectors.

From our point of view, the shortcoming of this requirements list is the confusion of architecture level and style level concepts. For instance, Di Nitto and Rosenblum do not require that a style comprises *application-independent* classifiers like, e.g., **Client** and **Server** which are used at the architecture level to categorize *application-specific* component and connector types like, e.g., R5/Banking or XBank (12)(17).

Analogously, the behavioral aspects of style and architecture are not separated, either: At the style level, we expect the definition of general communication *mechanisms* (14), while the description of the concrete communication behavior instantiating these mechanisms should be reserved for the architecture model (as in Fig. 1.4) (18).

Neither do Di Nitto and Rosenblum consider dynamic architectures nor does the classical notion of architectural style comprise reconfiguration mechanisms (15). Thus, their proposal can up to now only be applied to static architectures, and further research is required in order to capture also the reconfiguration mechanisms provided by a platform.

**Summary.** As shown in Table 2.2, existing work on platform-consistent architecture descriptions is still rare and incomplete. Platform-specific ADLs cannot cope with different platform types, and the work by Medvidovic et al. does not lead to an independent platform description language which can be combined with ADLs.

Table 2.2: Comparison of approaches to platform-aware description

Approach	Requirement						
	vocabulary (12)	constraints (13)	communication mech. (14)	reconfiguration mech. (15)	extensibility (16)	structural instantiation (17)	behavioral instantiation (18)
Medvidovic	○	–	+	–	+	○	○
Di Nitto/Rosenblum	○	+	○	–	–	○	–

– = not satisfied; ○ = partially satisfied; + = completely satisfied

Only the strategy of Di Nitto and Rosenblum using architectural styles as platform models is promising, but it has to be extended to better integrate platform-specific communication and reconfiguration mechanism, in particular for dynamic architectures. Moreover, their strategy has so far not been realized with concrete platform and architecture description languages.

### 2.3.3 Architecture refinement techniques

As expounded in Section 1.3.2, it is not enough to describe platform-consistent architectures, e.g., in a middleware-induced architectural style, but we also require a refinement technique which allows us to check if a platform-specific architecture still conforms to its underlying business-oriented architecture.

Although many different ADLs have been developed to specify software architectures, only very few of them support architecture refinement [107]. In particular, there is – to our knowledge – no approach to refining architectures across different levels of platform abstraction (according to the requirements in Section 2.2.3). Nevertheless, we can learn from existing work about architecture refinement as follows.

**Diverse purposes of refinement.** Refinement is a well-known design principle in software engineering. First ideas in the context of program development go back to Dijkstra [33] and Wirth [160]. In the sense of a systematic top-down methodology, Wirth, for instance, argued for the expansion of high-level program instructions to lower-level macros and procedures.

In the meantime, the notion of refinement has been employed for various purposes. Consequently, when surveying modern approaches to refinement for software architecture development, we encounter several proposals which do not suit our purpose of bridging different levels of platform abstraction. In the following, we briefly summarize three examples in order to distinguish them from the more relevant approaches which will be discussed afterwards.

As a first example, remember the LEDA approach [22] mentioned in Section 2.3.1. Its authors also investigate architectures at different levels of abstractions and check for valid refinement relationships. However, in doing so, they follow a very restrictive understanding of refinement dealing only with the specialization and decomposition of individual component types. They check if the communication behavior of a refined component is still compatible to its parent component type, so that the specialized component can substitute the original one. Thus, in a LEDA refinement, the overall arrangement of component instances is not changed as it might be required in

our understanding of refinement with architecture models being ported to a different level of platform abstraction.

Another, completely different purpose of refinement has been pursued by Batory et al. [15]. They consider *feature refinement*, which means extending models, code, and other artifacts in order to integrate additional features within every refinement step.

The third example of refinement aims at the facilitation of code generation. Bolusset and Oquendo [18] propose to use rewriting logic in order to *translate* architecture models from an abstract ADL to an implementation language which can be used to generate code. However, the authors only sketch their ideas and do not present any example for their rewriting rules. Moreover, the conversion between different languages is not the same as bridging different levels of platform abstraction. In particular, the authors do not discuss how, e.g., architectural behavior can in general be preserved at the lower level.

We do not further discuss the above approaches because we neither want to look into the inside of components nor add any extra-functionality to an architecture nor generate code, but we rather want to port a business-level architecture to a more platform-specific level considering all the restrictions and mechanisms of the chosen target platform.

**Refinement of single architectures.** A first contribution to refinement in our sense has been provided in the frame of the RAPIDE project [94], already mentioned in Section 2.3.1. Remember that system executions are represented as partially ordered event sets in RAPIDE (*posets*). In order to compare architectures at different levels of abstraction and to enable corresponding refinement checks, RAPIDE provides so-called *event pattern mappings* which can be used for translating posets of a concrete architecture execution to the abstract level. Then, one can check if the abstracted events of the concrete architecture also satisfy the constraints specified for the abstract architecture.

Although these pattern mappings are a powerful and flexible concept for validating executions of the concrete architecture against abstract constraints, they do not ensure that *every* concrete execution has an abstract equivalent, as required for complete observational substitutability (21). “This kind of architectural refinement rather looks like a conformance test” [18].

Moreover, the validation of the concrete behavior against abstract constraints does not provide concrete equivalents for every abstract execution as required for behavior preservation (20).

Moreover, the proposed notion of refinement completely neglects struc-

tural correspondences between the two architectures to be compared. Although this allows for more flexibility, we believe that at least some structural relationships between abstract and concrete architectures are required in order to preserve the structural integrity of the business level model (19).

**Style-based approaches.** In the previous section, we have already elaborated on Di Nitto’s and Rosenblum’s idea to use architectural styles as platform models and to describe platform-consistent architectures as instances of platform-specific styles. Consequently, when pursuing this strategy, architectural refinement means the transformation of an architecture in an abstract style into an instance of a more concrete style.

Di Nitto and Rosenblum already propose to define a generic style for a family of related middleware platforms, e. g., event-based systems, and a variation of that style for specific members of that family [118]. However, they do not further discuss or formalize any refinement relationship between such a generic style and its specializations.

Refinement of software architectures based on styles has first been introduced by Moriconi et al. in 1994/95 [113, 114], but not using styles as platform models. Building on a formalization in first-order logic, they propose a rule-based approach of refinement replacing a structural pattern in an abstract style by its realization in the concrete style. The resulting translation may involve changing the representation of components, interfaces, and connectors as well as aggregating, decomposing, or eliminating these abstract objects in the concrete architecture (19).

A refinement rule consists of a pair of architecture *schemas*, one in the abstract and one in the concrete style. A schema is a partial architecture description with variables instead of architectural objects. Whenever the abstract schema can be instantiated by substituting its variables by objects of a given architecture, then the concrete schema is used to generate the corresponding part of the refined architecture.

The local refinements are combined to form the composite concrete architecture. In order to ensure the correctness of the resulting concrete architecture by construction, the individual refinement rules have to be proved correct and compositional first. The correctness proof is based on a mapping between the concepts of the two styles involved. Given such a *style mapping*, one has to show that every refinement rule conforms to this mapping. The proof technique applied by Moriconi et al. is based on Hoare’s approach [71] to reasoning about the correctness of implementations.

In order to formally reason about architectural styles, architectures, and refinement patterns, Moriconi and his coauthors represent them as theories in



first-order logic and claim that, by translation from a given textual language, their approach becomes independent of the concrete choice of language.

Following the classical understanding of architectural styles, they only include structural aspects in their style definitions, i. e., classifiers, relationships, and configuration constraints. As a consequence, their approach is restricted to static configurations and does not consider any communication or reconfiguration behavior. Obviously, this makes the checking of both behavior preservation (20) and observational substitutability (21) useless.

Even though it is restricted to structural refinement, the work of Moriconi et al. is a first, valuable contribution to architectural refinement because the rule-based solution allows for an automated construction and validation of the refined architecture (25). However, the announced software tool has – to our knowledge – not been realized so far. Besides, the approach does not yet consider user interactions usually required for complex refinement constructions (see p. 7 for user interaction for construction).

Another advantage of their strategy to use architectural styles as source and target domains of refinement rules is the high reusability, as the rules can be applied to any two instances of the underlying styles (24). In this context, their work makes clear that we have to understand the semantic correlations between the involved architectural styles in order to check the correctness of such refinement rules.

In [48], Garlan generalizes the idea of style-based refinement and stresses the fact that, due to the reusability (24), it is more powerful to have refinement rules defined at the style level rather than for individual style instances. “For example, we might determine that any event-based system can be implemented as an object-oriented system if we transform each component so that it calls an event dispatcher, and we transform the event connectors to a dispatcher-mediated, broadcast connector” [48] (see Fig. 2.1).

While style-based refinement is more powerful and generally useful, it sometimes cannot take advantage of special cases and contexts of use and can thus be applied less effectively. In order to allow refinement rules which are only applicable to some but not all instances of a certain style, Garlan proposes to define a *substyle* which restricts the set of architectures to those amenable to the special refinement rules.

Furthermore, Garlan suggests to characterize refinement relationships by an *abstraction function* which maps every instance of the concrete style to an instance of the abstract style. In contrast to the approach of Moriconi et al., the use of an abstraction function allows for several concrete architectures all refining the same abstract architecture (23).

The work by Garlan can be considered as a generic study of the problem.



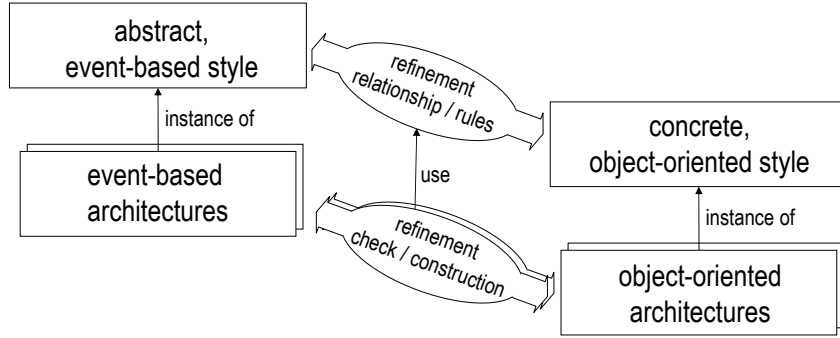


Figure 2.1: Style-based refinement relationships

His methodological suggestions remain abstract and leave room to different implementation techniques and formal methods.

**Summary.** As we can learn from the above observations, summarized in Table 2.3, a meaningful refinement of architectural models cannot go without some semantic refinement relationship between the concepts at the abstract and the concrete level. By lifting this relationship to the underlying architectural styles, the style-based approaches of Moriconi et al. and Garlan achieve the required reusability and, thereby, reduce the effort required for defining these relationships.

Moreover, the style-based approaches attract our sympathy because they smoothly fit to the previously elaborated idea to use architectural styles as platform models. Under these circumstances, style-based architecture refinement could be employed for bridging different levels of platform abstraction.

However, the shortcomings of the existing architectural refinement approaches are obvious: They are all lacking sufficient support for the refinement of architectural behavior in terms of behavior preservation (20) and observational substitutability (21). Although a number of refinement theories has been proposed for different formal methods like process algebras [53] or graph transformations [55], none of them has been seriously applied to the domain of software architecture so far. Hence, further research is required to enhance the idea of style-based refinement and to incorporate appropriate refinement concepts for both communication and reconfiguration behavior.

Table 2.3: Comparison of approaches to architecture refinement

	Requirement						
Approach	structural ref. (19)	beh. preservation (20)	substitutability (21)	semantic ref. (22)	context dependency (23)	reusability (24)	algorithm and tools (25)
RAPIDE	–	–	o	+	–	–	+
Moriconi et al.	+	–	–	–	–	+	o
Garlan	+	–	–	–	o	+	–

– = not satisfied; o = partially satisfied; + = completely satisfied

## 2.4 Summary

From the preceding survey of related work, we can conclude that there is no complete approach to a platform-consistent development of dynamic software architectures. Especially for the two subproblems of combining architecture descriptions with platform models and of refining architectures across different levels of platform abstraction, we are still missing convincing solutions that go beyond abstract strategies.

Moreover, if we found solutions to the individual subproblems, the challenge would remain to integrate them into a homogeneous, uniform approach for the platform-consistent development of dynamic software architectures. This is difficult because the required formal method has to encompass both structure and behavior in three different contexts, namely architecture models, platform models, and architectural refinement.

## Chapter 3

# The style-based approach – an overview

After the evaluation of existing proposals, we will try to close the apparent gaps by a new approach to platform-consistent architecture development. This chapter provides a first informal introduction into the approach and shall help to understand the interrelations between subsequent chapters.

Since architectural styles have turned out to be useful both as platform models (see Section 2.3.2) and as source/target domains for architectural refinement (see Section 2.3.3), we propose a *style-based* approach and, similar to Di Nitto and Rosenblum, employ middleware-induced architectural styles as platform models. However, in contrast to the classical understanding of architectural styles they have (see p. 39), we claim that a style should not only comprise structural patterns with vocabulary and configuration constraints but also behavioral aspects which can be used to represent communication and reconfiguration mechanisms of a certain middleware platform.

We will show that *graph transformation theory* is an appropriate formal method to capture such an extended notion of architectural style. Thus, our work continues the line of software architecture research using graph transformation systems reviewed in Section 2.3.1. To the existing contributions, we will add the new role of graph transformation systems as platform-specific architectural styles. Based on such styles, we will then investigate appropriate refinement conditions for a stepwise, platform-consistent architecture development technique.

The first section of this chapter covers the style-based modeling of platform-consistent, dynamic architectures; the second section then turns to style-based architectural refinement.

## 3.1 Style-based modeling

A style-based modeling process involves two different tasks and corresponding expert roles: While a *style architect* has to provide architectural styles capturing characteristics and mechanisms of available target platforms, the *application architect* has to provide platform-consistent architecture descriptions conforming to one of these styles.

Concerning the representations these experts shall use for architectural styles and architecture models, we are facing a natural trade-off between formality and usability: While formal representations often benefit from precise, mathematical semantics supporting model simulation and analysis, they are usually more complex and difficult to handle for end users. Due to this trade-off, purely formal representations would probably decrease the acceptance rate of the approach by practitioners.

For this reason, we propose a two-layered, hybrid approach combining graph transformation theory with a well-known, user-friendly modeling language like UML. As depicted in Fig. 3.1, UML serves as a “front-end” architecture description language whose operational semantics is formally defined by graph transformation systems. To be more precise, UML provides the basis for various style-specific ADL *variants*, and each of these variants is semantically backed up by its own graph transformation system.

For defining architectural styles, style architects need expertise in both areas: they assemble syntactical style constructs in a *UML profile*, setting up a new UML variant, and define a *graph transformation system* as the underlying operational semantics. By a formal *relation*, they fix the correspondences between style syntax and semantics.

Thanks to this syntax-semantics relation, application architects can confine themselves to the UML layer only: After they have modeled an architecture as instance of the style with UML, they can use CASE tools to *translate* these models into the corresponding graph-based representation. Beyond this translation, CASE tools can also exploit the style semantics in order to compute a *graph transition system* which is open to further analysis, simulation, and – as we will see later – behavioral refinement checks.

In the following subsections, we will explain the two (vertical) layers shown in Fig. 3.1 in more detail.

### 3.1.1 Formal, graph-based descriptions

The formal description of dynamic architectures is sketched in the right part of Fig. 3.1. We advocate graphs to represent architecture models and graph transformation systems, consisting of a *graph schema* and a set of *graph*

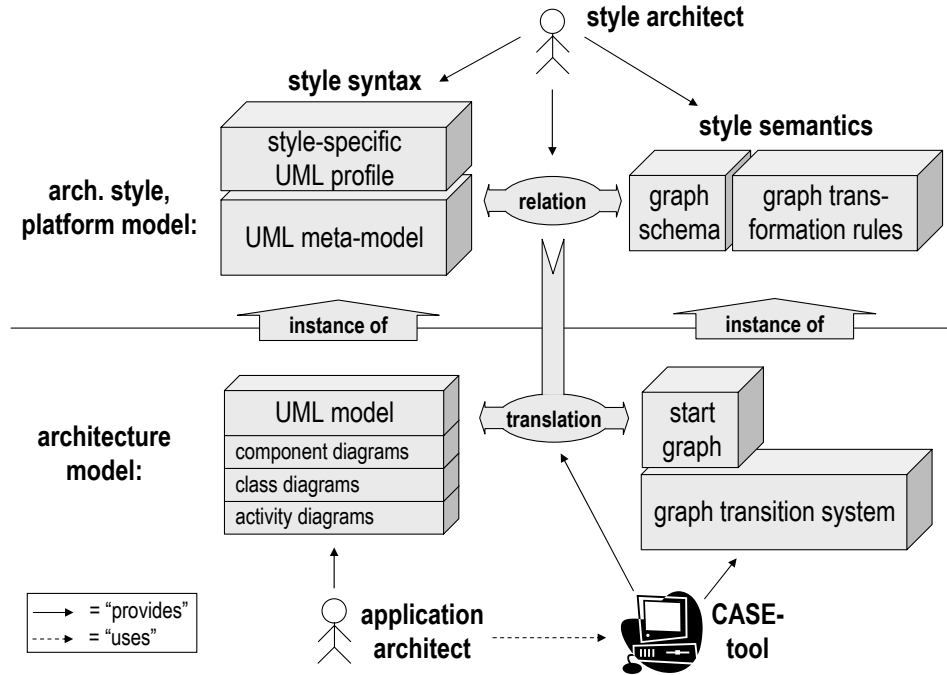


Figure 3.1: Style-based modeling – an overview

*transformation rules*, to represent architectural styles. Based on the operational semantics of graph transformation rules, we will be able to express an architecture’s behavior in form of a graph transition system, which is derived from a dedicated *start graph* representing the initial configuration.

Our decision for graph transformation theory is in the tradition of existing graph transformation-based ADLs (as reviewed in Section 2.3.1), which have already documented the expressiveness and suitability of graph transformation for formal descriptions of dynamic architectures. But, while they use graph transformation rules to describe the behavior of a particular application, we conceptually separate styles from individual architectures by lifting the transformation rules to the application-independent style level.

We will show that graph transformation systems smoothly fit to represent architectural styles: the graph schema defines the style vocabulary (12) plus topological constraints (13), and the *graph transformation rules* describe available communication (14) and reconfiguration mechanisms (15). This way, we provide better style definition capabilities than the existing ADLs surveyed by Di Nitto and Rosenblum [118] which reflect the structural part of architectural styles only, without communication or reconfiguration mechanisms.

**Graph schema and instance graphs.** As the name suggests, a graph schema characterizes a certain class of graphs. For this purpose, it defines a *type graph* with node and edge types and how instances thereof can be composed to legal graphs. Additional *constraints* can be used to further restrict the class of legal graphs. Members of the schema, which are typed over the type graph and satisfy the relevant constraints, are called *instance graphs*. Using a familiar notation, we can think of a graph schema as UML class diagram and of instance graphs as UML object diagrams [121].

Since we use a graph schema as part of a platform-specific architectural style, its type graph establishes a platform-specific vocabulary and the additional constraints reflect platform-specific, topological constraints on instance graphs. Examples of type graphs and constraints can be found for an abstract component-based style and a service-oriented style in Chapter 5.

Valid instance graphs of such a graph schema represent architectures conforming to that style. The correctness of an instance graph with respect to the type graph can easily be checked by the existence of a graph homomorphism as explained in Section 4.1. This way, we can realize the required structural instantiation of platform models by architectures through a straight-forward type checking algorithm (17).

Graphs are an abstract and generic means of representation which is frequently used for both structural and behavioral modeling in software engineering. Thus, provided that the graph schema defines a comprehensive vocabulary, the corresponding instance graphs can be used to capture all aspects of architecture descriptions including component and connector types (1), their communication (2) and reconfiguration behavior (5), as well as individual runtime configurations with the current state of interaction (3).

**Graph transformation rules.** The dynamic part of a architectural style, defining the available communication and reconfiguration mechanisms of a platform, is captured by the graph transformation rules. Each rule defines a certain modification of instance graphs. As our instance graphs represent architectural configurations and current interaction states, such transformations smoothly fit to represent reconfigurations or communication steps.

Note that, in contrast to the approaches of Le Métayer [91] and Hirsch et al. [68, 69], we do not use graph transformation rules as production rules of a graph grammar restricting the set of legal graphs for architecture descriptions; this issue is already handled by the declarative graph schema. Similar to Taentzer’s approach [150], the transformation rules rather specify what actually *happens* when communication or reconfiguration operations are invoked.

A graph transformation rule defines the preconditions and effects of a single operation in terms of two graph patterns: The *left-hand side* pattern is required to occur in an instance graph before the rule can be applied, and the *right-hand side* pattern specifies the appearance of this occurrence after a rule application. Thus, the differences between left- and right-hand side determine the actual changes of the identified occurrence. As both graph patterns are typed over the style's type graph, the rules conform to the platform-specific vocabulary.

As a first example, consider the two symmetric rules for component connection and disconnection in Fig. 3.2.<sup>1</sup> From left to right, the rule **connect** requires that there are two components, one requiring an interface which is provided by the other. By applying the rule, the two components connect their ports. The **disconnect** rule works the other way round. More complex examples can be found in Chapter 5.

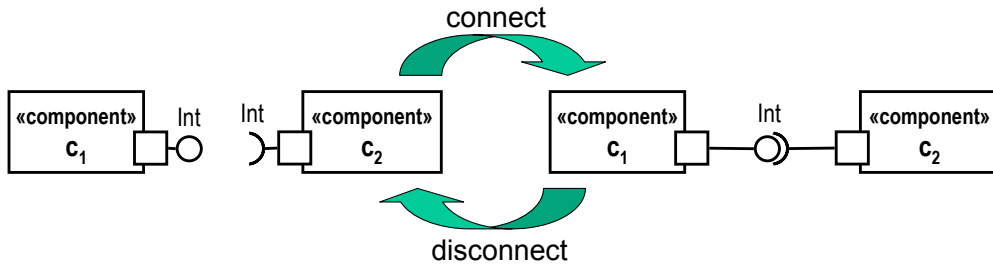


Figure 3.2: Transformation rule examples for component (dis-)connection

Besides transformation rules expressing reconfiguration operations like **connect** and **disconnect**, there are also transformation rules expressing communication operations. This is possible because we will also equip the instance graphs with nodes and edges for the current communication status, e. g., for the signals and messages which are currently on the way. Any progress of ongoing communication can then be captured by suitable graph transformations, too.

This way, we can describe both communication and reconfiguration mechanisms with graph transformation rules. In contrast to most existing graph transformation-based ADLs (cf. Table 2.1), we achieve a uniform formal underpinning for both kinds of architectural behavior, as desired in Req. (10). For brevity, we also call communication-related transformation rules *communication rules*, and reconfiguration-related ones *reconfiguration rules*.

<sup>1</sup>As the formal rule notation follows in Chapter 4 only, we represent the two rule sides using an intuitive UML syntax.

**Simulation of architectural behavior.** Due to the operational semantics of graph transformation rules (formally defined in Section 4.2), the communication and reconfiguration operations can be *simulated* by applying the transformation rules to instance graphs. As our instance graphs represent architectural runtime states, any application of a graph transformation rule yields a new runtime state simulating the effect of the corresponding operation. From the new state, another rule application leads to another state, and so on.

When simulating an architecture model in this way, we have to ensure that the communication and reconfiguration rules are only applied according to the respective component behavior specifications. In Chapter 5, we will show how to couple the rule applications with the behavior specifications of the current architecture model so that a simulation exactly reflects how components and connectors invoke the available platform services.

Since a distributed architecture contains many components running in parallel, usually several components want to invoke a communication or reconfiguration operation simultaneously. However, since simultaneous rule applications easily lead to conflicts if the individual applications are not independent from each other, we will approximate the required parallelism by a *concurrent* execution semantics.

For this purpose, we will exploit another important property of graph transformation systems, namely its inherent *non-determinism*: In the absence of additional control constructs and other means like priorities, an interpreter of the graph transformation system can arbitrarily choose one of the currently applicable rules and one of the valid left-hand side occurrences. Thus, in each step, the progressing component is selected non-deterministically. This way, we achieve an interleaving semantics which approximates parallelism by concurrent behavior.

**Graph transition systems and model analysis.** Based on the simulation of single communication and reconfiguration steps, we will derive a *graph transition system* as a complete representation of all possible runtime evolutions of an architecture. A graph transition system has instance graphs as states and rule applications as state transitions. From a given start graph, it is constructed by recursively applying all enables rules to previously derived instance graphs. This way, the transition system reflects all possible execution traces of the concurrent architectural behavior.

A simplified<sup>2</sup> example of a graph transition system is shown in Fig. 3.3. Its

---

<sup>2</sup>As instance graphs are formally introduced in Chapter 4 only, the runtime states are sketched using UML component diagrams.



start graph, depicted at the left, represents a possible initial configuration of the travel system with all participating components still being unconnected. The other states and their incoming and outgoing transitions represent all possible applications of the two reconfiguration rules **connect** and **disconnect** known from Fig. 3.2.

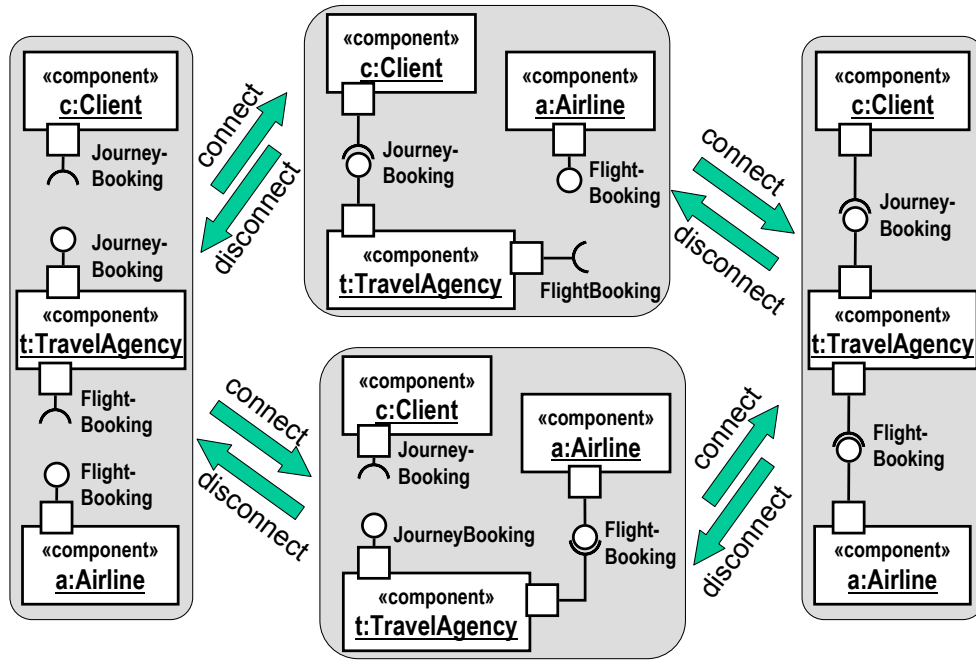


Figure 3.3: Transition system as operational architecture model

Such a graph transition system representing the complete architectural behavior is an important prerequisite for various analysis and model checking tools which can verify a wide bunch of properties. For instance, we will be able to check if a certain desired or undesired state or sequence of states can occur during system execution (9). Model checking based on graph transformation systems is a young field of ongoing research. In Chapter 8, we will present two promising tools and discuss their capabilities as well as limitations.

Beyond analysis in general, graph transition systems also play an important role in our approach for checking behavioral refinement at a semantic level, as explained in Section 3.2.

### 3.1.2 UML as concrete syntax

The flexibility of graphs as generic means for architecture descriptions is traded off against a decreasing usability when it comes to models of larger size. Especially when including various aspects like type and behavior definitions as well as runtime configurations in a single instance graph, these graphs will quickly become intricate and contradict our usability requirement (11).

For this reason, we add a UML layer on top of the graph-based representation, as depicted in the left column of Fig. 3.1. Instead of using intricate instance graphs, application architects can then design software architectures – modularized into different views – using various UML diagram types. Internally, CASE tools will translate these models into the corresponding formal graph representation so that the above mentioned graph transition system can still be derived and used for further analysis and refinement checks.

**UML diagrams.** UML’s visual representations borrow from various well-known notations for both software structure and behavior, e. g., use cases [78], class diagrams [135], state charts [57], and message-sequence charts [77]. Meanwhile, UML has become a de-facto standard in practical software modeling. And, a reasonable subset of UML is suited to represent different views on software architectures [106].

For example, *UML component diagrams* represent the structure of an architecture in terms of components, their interfaces, and their connections. Component diagrams can be used at two levels: at the type level to define component types and at the instance level to define runtime configurations of component instances. Revisiting our travel application, the component diagram in Fig. 3.4 shows some of the component types, associated ports, and provided (circle) or required (half-circle) interfaces. The corresponding instance-level diagram depicting a possible runtime configuration has already been shown in Fig. 1.1.

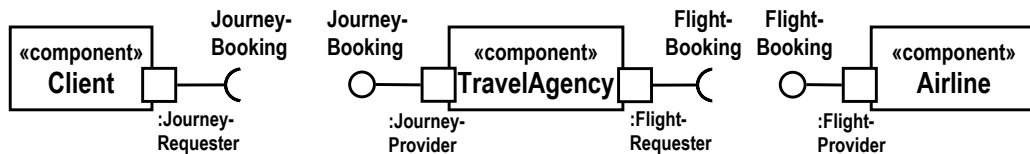


Figure 3.4: UML component diagram

Other diagram types supporting architecture modeling are, among others, *UML class diagrams* for interface definitions and *state charts* or *activity diagrams* for behavior of components and connectors.

**UML profiles.** Unfortunately, UML does not provide any first-class construct for defining and incorporating architectural styles. However, as architectural styles restrict the set of legal architecture models concerning vocabulary, constraints, and allowed operations, they can be considered part of the language definition rather than part of the model. Accordingly, we propose to define architectural styles at the UML meta-level, namely by *UML profiles*.

In order to understand UML profiles, one has to know that UML's *abstract* syntax is fixed by a *MOF meta-model* [121]. A profile extends this meta-model by *stereotypes* and additional *constraints*. Stereotypes specialize existing meta-model elements and can be used by a style architect to adapt UML to the style vocabulary. Constraints restrict the rules for composing modeling constructs and can be used by a style architect to impose topological constraints of the style.

This way, a style architect derives a style-specific UML variant. Application architects can then use the new, style-specific UML variant for their architecture models. According to the UML principles, these models are instances of the extended meta-model and have to adhere to the profile definitions and restrictions. Consequently, they automatically comply with the underlying style and platform model, respectively.

For example, consider a style architect defining a UML profile for service-oriented architectures. In SOA, software components providing their functionality to other components are called *services*. In order to distinguish them from ordinary components, the style architect could introduce a new stereotype `«service»`. An application architect who wants to describe a SOA variant of our travel application could then assign this stereotype, e.g., to the service components `TravelAgency` and `Airline`, as shown in Fig. 3.5.

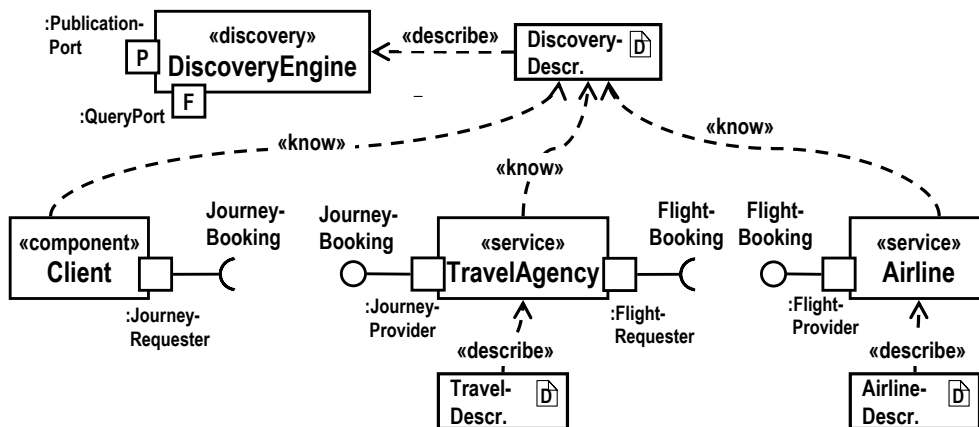


Figure 3.5: UML component diagram in a service-oriented style

The functionality and interfaces of a service are published using suitable description documents. As these descriptions play a central role for dynamic service discovery and binding by service requesters, they are included as stereotyped artifact symbols in the architecture model, too. Moreover, the special dependency stereotypes `«describe»` and `«know»` reveal which service a description refers to and who else knows this description, respectively.

A service-oriented architecture also contains a *discovery service* mediating between service providers and requesters. It is modeled like a service but with the special stereotype `«discovery»`. If the other components `«know»` its description, they can initiate service publication and discovery activities.

**Modeling platform-specific behavior.** As mentioned at the beginning of this chapter, architectural styles also comprise available communication and reconfiguration mechanisms for the behavioral part of architecture models. With the UML profiling mechanism, such platform mechanisms can be expressed by stereotypes specializing generic UML action types to be used for behavior descriptions, e. g., in activity diagrams.

This way, the stereotyped actions of an activity diagram represent components or connectors invoking style-specific communication and reconfiguration operations. In this work, we prefer UML activity diagrams to state charts because their action-centered notation better allows to insert the stereotypes for style-specific operations.

Figure 3.6 shows two activity diagram examples, one for the **TravelAgency** and one for the **Client** component type. Both diagrams are given in an business-oriented style which provides mechanisms for connecting and disconnecting components, for sending and receiving remote interface calls, and for sending and receiving response messages. The stereotypes like `«connect»` or `«disconnect»` indicate which platform mechanism an action refers to, and the lower labels indicate which element of the architecture the action refers to.

In the example, the **TravelAgency** component awaits connections to its **JourneyProvider** port and incoming calls to a provided **bookJourney** operation. Assuming that booking a journey includes booking a suitable flight, the next action connects the **FlightRequester** port to an airline component where the **bookFlight** operation can be called. After possible further actions, which we omit in the example, the **TravelAgency** sends a response message to the client, closes the opened connections again, and waits for new requests from other clients. The behavior of the **Client** component is complementary to that of the **TravelAgency**: The client sends the call and receives the response of the travel agency.

The service-oriented style includes additional mechanisms, e. g., for ser-

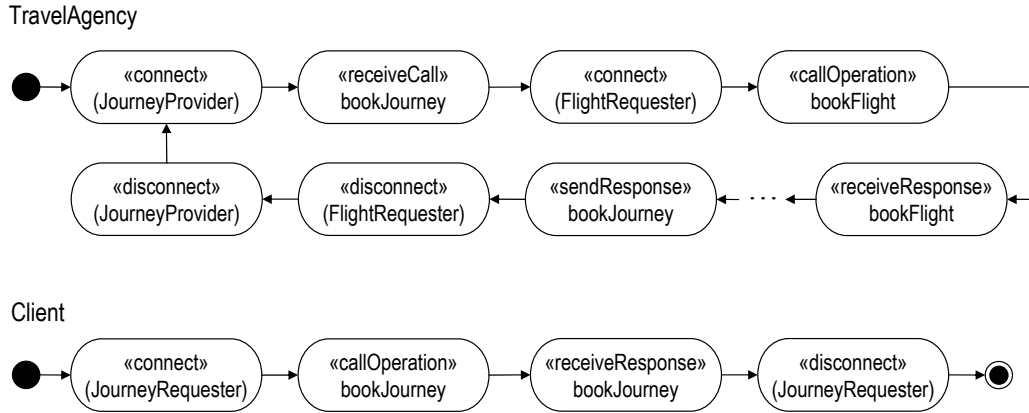


Figure 3.6: UML activity diagrams

vice publication and discovery. The complete definition of the corresponding UML profile can be found in Chapter 6.

**Semantic domain for UML models.** Although the new action (stereo-) types refer to available platform operations, they do not at all explain what happens to a configuration when such an operation is executed. This is due to the fact that stereotypes are syntactic constructs only, and there is no means in the profiling mechanism to formally define their execution semantics. However such semantics is required for computations like simulation or behavioral refinement checks (8).

The solution to this problem lies in a relation between the UML layer and the graph transformation layer: If style architects relate the syntactical constructs of the UML profile with the transformation rules they have defined in the graph transformation layer, then these rules provide the required semantics definitions. For instance, the transformation rules of Fig. 3.2 can reveal the effects of the «connect» and «disconnect» actions used in Fig. 3.6. Now it becomes obvious why we have called the UML profile in Fig. 3.1 *style syntax*, whereas the graph transformation system is referred to as the *style semantics*.

If simulation or analysis tools want to execute a UML architecture model according to the graph transformation-based semantics, they have to *translate* the model into an equivalent instance graph which they can apply the graph transformation rules to. In order to facilitate such translations between the UML- and the corresponding graph-based representation of an architecture, the style architect has to formally define the syntax-semantics *relation* between the extended UML meta-model and the graph schema of the graph

transformation system (cf. Fig. 3.1).

The idea to use graph transformation systems as semantic domain for UML models is inspired by the approach of Engels et al. [42], called *dynamic meta-modeling*. They apply graph transformations to specify operational semantics of modeling languages like UML, too.

Besides the necessity to define the meaning of new, style-specific language extensions, there are two more reasons for a separate semantic domain: First, the standard UML semantics definition is an *informal*, general-purpose description only which does not allow any serious analysis or simulation, either. Second, we consider the choice of UML as one possible option for the syntax of architecture models only; with the separate semantic domain, our approach can also be combined with different notations if desired.

We agree that the usage of style-specific UML profiles and graph schemas, respectively, can be considered as defining platform-specific ADLs. However, different to the platform-specific ADLs criticized in Section 2.3.2, our ADL variants are defined in a systematic and uniform way, share a large part of the syntax (UML), and have the same semantic underpinning (graph transformations). Moreover, the approach can be applied to arbitrary platform types.

## 3.2 Style-based refinement

For a stepwise platform-consistent development as motivated in Chapter 1, the modeling of architectural styles and dynamic architectures is not enough. On top, we will introduce a refinement approach which allows to confirm whether a concrete architecture (given, e. g., in a platform-specific style) complies with an abstract one (given, e. g., in a business-oriented style).

The survey in Section 2.3.3 has exhibited that most existing solutions for architectural refinement concentrate on refinement of structure but do not sufficiently support refinement of behavior. In particular, the requirements behavior preservation (20), observational substitutability (21), semantic refinement (22), and context dependency (23) are not satisfied. Our intention is to complement the existing work by especially focusing on these aspects of behavioral refinements.

Of course, we also aim at a smooth integration with the style-based modeling approach sketched in the previous subsection. In this context, it is important to note that we can use architectural styles to describe platform models at various levels of abstraction (cf. Fig. 1.5). This way, styles represent the source and target domain of a refinement step, respectively. Actual

refinement checks for a given pair of abstract and concrete architectures will be based on a predefined relationship between the involved architectural styles. This principle of *style-based* refinement has already been advocated by Moriconi et al. [114] and Garlan [48] (see Section 2.3.3).

The relationship between abstract and concrete styles shall help to bring instances of these styles to the same level of abstraction. This is a prerequisite for *comparing* the structural and behavioral parts of two models which undergo a refinement check. Basically, there are two options to overcome the gap between the two levels of abstraction: One could either refine the abstract architecture by adding all necessary details about the underlying middleware and how to use its capabilities, or project the concrete architecture to the abstract level by removing all these details. Both alternatives, refinement and abstraction, are sketched in Fig. 3.7.

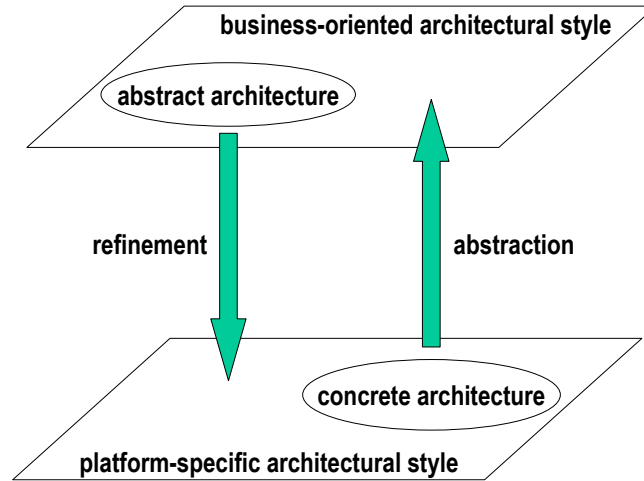


Figure 3.7: Refinement vs. abstraction

For a given pair of abstract and concrete architectural styles, it is easier to define an abstraction function removing platform-specific details rather than a refinement function. This is because refinement functions adding such details usually depend on a number of application-specific design decisions and, thus, cannot hold for all instances of the two styles. On a service-oriented middleware, for example, it depends on the individual application which business component is refined into a service and which not.

Consequently, we will base our refinement checks on abstraction functions for every pair of related abstract and concrete architectural styles. We define the abstraction functions at the style level so that they apply to any instance of the concrete style (24).

Figure 3.8 illustrates the effect of an abstraction function for service-oriented architectures, applied to the SOA-specific configuration of our travel application (Fig. 3.5). The abstraction function removes platform-specific elements (e.g., descriptions and discovery services) and translates remaining elements into the abstract vocabulary (e.g., «service» into «component»).

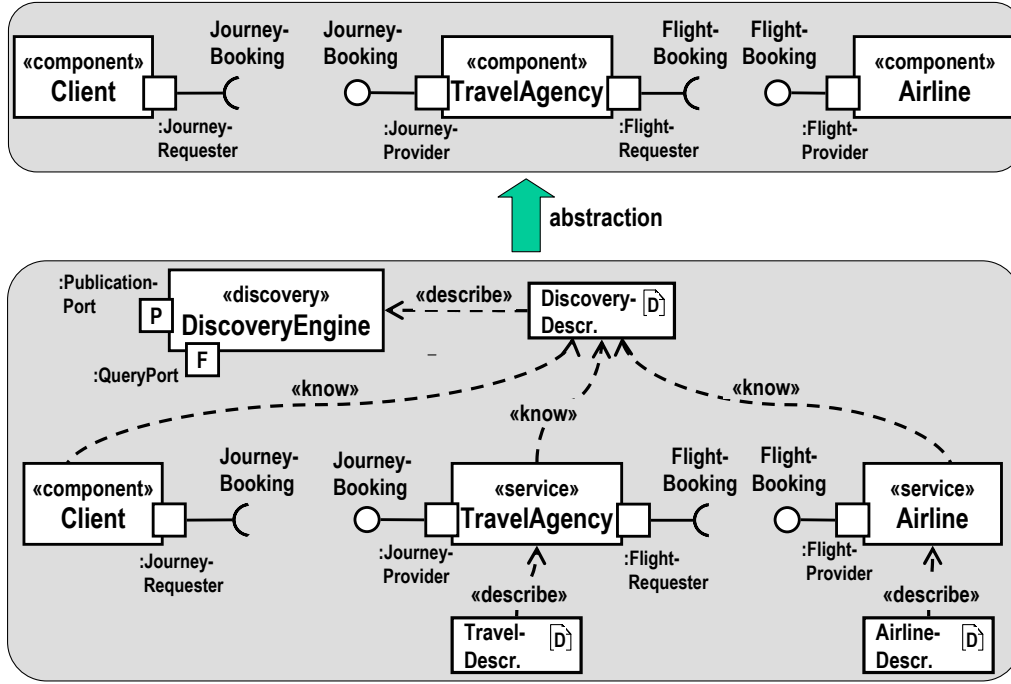


Figure 3.8: Effect of an abstraction function

Though Fig. 3.8 shows the effect of an abstraction function in the UML syntax, we formally define these functions in our semantic domain, i.e., for graphs being instances of the corresponding graph schemas. In doing so, we benefit from the clarity of the semantic domain, do not have to cope with syntax-specific constructs, and retain the independence of a specific notation.

Hence, an abstraction function translates instances of the concrete graph schema into instances of the abstract graph schema. There is a range of possibilities to express this relationship between the two graph schemas, from simple mappings between node and edge types up to more complex ones defined by special graph transformation rules (see Section 7.2.1).

After a concrete architecture has been projected to the abstract level using an abstraction function, one can easily compare it with the abstract model and check if both contain the same business-relevant components and functional entities.



So far, we have outlined a rough strategy for structural refinement checks only; more details will be given in Section 7.2. An analogous method for behavioral refinement checks would require a fixed relationship between the communication and reconfiguration operations of the platform-specific and the platform-independent architectural style. Using graph transformation systems as semantic domain, this relationship would amount to a mapping between the corresponding graph transformation rules.

However, a solution based on such a fixed mapping would not satisfy our requirements of semantic interpretation (22) and context dependency (23): Semantic interpretation demands a solution which is capable to deal with divergent platform mechanisms which cannot simply be matched because of, e. g., different side effects. And, context dependency prohibits fixed mappings because it requires alternative refinements in different contexts.

For these reasons, we propose another approach which allows to decide behavior preservation and observational substitutability without a fixed mapping between abstract and concrete operations. Instead, our solution is based on a comparison of the two *graph transition systems* representing the complete behavior of the concrete and the abstract architecture (see Section 7.3).

Each path in one of the two transition systems stands for a possible scenario of communication and reconfiguration steps caused by the architecture's constituents. Roughly speaking, we can decide behavior preservation (20) by checking if every such path in the abstract transition system has a correspondent path in the concrete transition system, and observational substitutability (21) is decided the other way round.

As we do not assume a fixed relationship between abstract and concrete operations, the comparison of paths cannot simply be based on a comparison of individual transitions, which represent rule applications. Instead, we will present a refinement relation which compares the *states* reached on those paths. In order to identify compatible states, we can reuse the above sketched strategy for structural refinement with abstraction functions.

Figure 3.9 illustrates a state-based refinement check of two transition system paths. Given an abstraction function mapping concrete states  $s^c$  to abstract states  $s^a$ , we will check whether every concrete state has an abstract counterpart and vice versa. Moreover, the order of states must be preserved.

The example reveals that *several* consecutive states of the concrete path may be mapped to the same abstract state (but not vice versa). The rationale behind is that platform-specific behavior is more fine-grained and usually includes middleware-related operations which do not alter the business-relevant parts reflected in the abstract runtime state. Consequently, as the refinement of a transition is determined by the refinement of its source and target states, a single abstract transition may be refined into multiple transitions at the

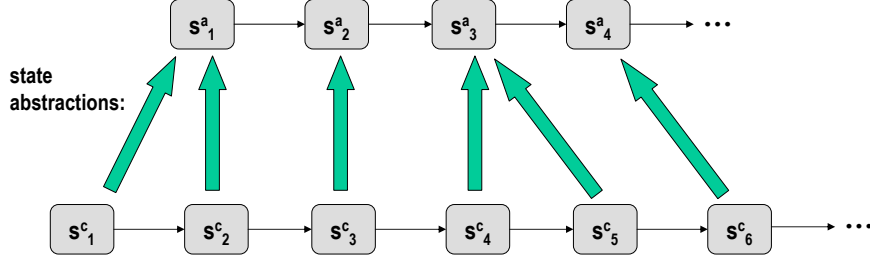


Figure 3.9: State-based refinement check of transition system paths

platform-specific level.

Remember that every transition in a graph transition system represents the application of a certain graph transformation rule. Since the refinement of a transition is not determined by a predefined mapping between these transformation rules but by a mapping between actual runtime states which subsume the effects of previous operations, different occurrences of a certain abstract operation can be refined differently. This way, we will be able to satisfy the context dependency requirement (23).

The relationship between the states of two transition systems will be formalized by suitable *simulation relations*. Their existence entails the desired behavioral refinement properties. In Section 7.3, we will present efficient algorithms which can compute whether such simulation relations exist or not.

However, we have to admit that both behavior preservation and substitutability are only decidable if the number of states in the two transition systems at hand is *finite*. In the unrealistic case that an architecture allows to create an unbounded number of new component instances at runtime, the number of reachable states would be infinite, too, and our refinement checking algorithms would never terminate. The only thing we can do in such a case is a refinement *test*, taking a finite subset of finite paths as *test cases* (see Section 7.4). Testing is also a good compromise if the size of the architectural models grows too large for complete refinement checks, e. g., caused by the state explosion problem.

After this informal overview of our style-based modeling and refinement approach for dynamic architectures, the next chapter provides the graph transformation theory we will employ as formal foundation. Readers already familiar with graph transformation theory can safely skim through Chapter 4 and confine themselves to those aspects which are new to them. In the subsequent chapters, we will then apply the theory in order to realize the above introduced approach.

## Chapter 4

# Graph transformation theory – the formal background

This chapter provides an introduction to the theory of graph transformation systems used as formal method for modeling architectural styles and software architectures in subsequent chapters. Note that we cannot cover all aspects of graph transformation theory here, but we rather select those aspects relevant for the remainder of the thesis.

Graph transformation systems in general provide an intuitive description for the manipulation of graphs and graph-based structures as they occur in representations of programming language semantics, data bases, object-oriented systems, and various kinds of software and distributed systems. Due to their formal, operational semantics, these descriptions can be analyzed and executed using suitable graph transformation tools.

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

Alternatively, formal graph-based languages can also be defined in a declarative way using a so-called *graph schema* that consists of a *type graph* and further *constraints*. By definition, all graphs that conform to the graph schema belong to the corresponding language. Since the graph schema is used at the meta level for the definition of a language, it is also referred to as the *meta-model* of the language.

Since the declarative meta-modeling approach becomes more and more popular, especially in the area of modeling languages with UML being the best-known example, we apply the meta-modeling approach in this thesis, too. Hence, for defining the set of graphs representing valid architectures of

a certain architectural style, we do not provide graph production rules but a declarative graph schema. We rather use *graph transformation rules*, which can describe any local transformations of instance graphs of the schema, in order to specify possible changes of system states, i. e., the effects of architectural operations. Together, the graph schema and the transformation rules constitute a *graph transformation system*.

In Section 4.1, we will formally define graphs and graph schemas. In Section 4.2, we define graph transformation rules and their operational semantics with the help of algebraic constructions. Then, in Section 4.3, we explain how graph transformation systems can in general be applied to model and analyze both structure and behavior of dynamic systems.

## 4.1 Graphs and graph schemas

When formally defining graphs and graph schemas, we already have to consider the kind of graph transformation systems we want to use them for. The reason is that the literature on how to define graph transformations and their semantics is split into two different mathematical techniques: the *set theoretic* approach and the *algebraic* approach. They basically differ in the way how the local effect of a graph transformation rule is embedded into the original host graph.

For this work, we deliberately decided to apply the algebraic approach invented by Ehrig, Pfender, and Schneider in the early seventies [39]. The approach builds on *Category Theory*, an abstract theory in mathematics, and defines the gluing of graphs as a categorical construction. In Category Theory, a *category* contains objects and arrows (called *morphisms*) that represent relations between objects. Categories are abstract in the sense that they represent anything with complex structure or even no structure at all. For more information on Category Theory, the interested reader is referred to [90].

A basic example of a category is **Set**, whose objects are sets and whose morphisms are total functions between sets. Similarly, we define the category **Graph** with directed, unlabeled graphs as objects and total graph morphisms as arrows:

**Definition 4.1 (Graph and Graph Morphism).** *A directed, unlabeled graph is a tuple  $G = (G_N, G_E, \text{src}_G, \text{tar}_G)$  with disjoint sets  $G_N$  of nodes and  $G_E$  of edges, and two functions  $\text{src}_G, \text{tar}_G : G_E \rightarrow G_N$  assigning source and target nodes to each edge.*

*A graph morphism  $f : G \rightarrow H$  of two graphs  $G$  and  $H$  is a pair of total functions  $f = (f_N : G_N \rightarrow H_N, f_E : G_E \rightarrow H_E)$  preserving source and target, i. e., satisfying  $\text{src}_H \circ f_E = f_N \circ \text{src}_G$  and  $\text{tar}_H \circ f_E = f_N \circ \text{tar}_G$ .*

A graph morphism is injective or surjective if these properties apply to both functions respectively. If  $f_N, f_E$  are bijective, the graph morphism is also called isomorphism. If there is an isomorphism  $i : G \rightarrow G'$ , then  $G$  and  $G'$  are called to be isomorphic ( $G \cong G'$ ).

In statements about both nodes and edges, we simply write  $x \in G$  (instead of  $x \in G_N \cup G_E$ ) and subsume the two morphism functions as  $f(x)$ .

#### 4.1.1 Type graphs and typing morphisms

Similar to class diagrams in object-oriented models or entity relationship diagrams in data models, we want to use schemas to restrict the set of allowed graphs in a declarative way. One common means to restrict the shape of an object is to prescribe a certain *type* for the object. Thus, in the following paragraphs, we will introduce the notion of *typed graphs*.

A first step towards typed graphs is to introduce node edge *labels*, resulting in *labeled graphs*. But, one major disadvantage of labels is that edge labels do not prescribe the types of source and target nodes. This additional information can only be given by a *type graph* as introduced in [29].

A type graph  $TG$  is a graph whose nodes represent node types and whose edges represent edge types. A graph that is typed over a type graph  $TG$ , also called *instance graph* over  $TG$ , is a graph  $G$  equipped with a graph morphism  $type_G : G \rightarrow TG$  that assigns a type to every node and edge in  $G$ .

An edge type in  $TG$  represents a structural relationship among nodes of  $TG$ -typed graphs. This is because, due to the typing morphism, edges of an edge type may only connect nodes of the node types that are incident to the edge type in  $TG$ . A node type can be compared to a class and an edge type can be compared to an association in a *UML class diagram* (except that we always use directed edges). Hence, we can depict type graphs by “directed” UML class diagrams, as shown in Fig. 4.1.

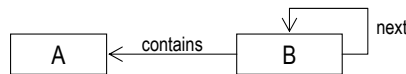


Figure 4.1: Exemplary type graph as UML class diagram

As type graphs can be represented by UML class diagrams, instance graphs can be represented by corresponding “directed” *UML object diagrams* as shown in Fig. 4.2. According to the UML syntax, each node is labeled by an identifier (optional) followed by a reference to its type. Both identifier and type reference are underlined. Edge labels do not contain an identifier but refer to the edge type only.

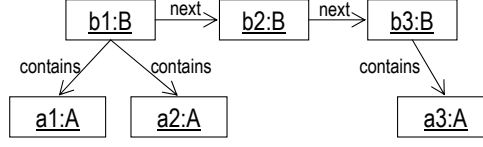


Figure 4.2: Instance graph as UML object diagram

**Node type inheritance.** A useful typing concept is the derivation of new, specialized *subtypes* from existing types. For example, one can specialize classes in class diagrams or entities in entity relationship diagrams. Such subtyping allows to reuse the definition of the existing type because a subtype *inherits* all the type restrictions of its supertype. Due to this inheritance, an instance of the subtype can be used whenever an instance of the supertype is expected.

Moreover, subtyping can be used as a means of abstraction: Whenever there is a number of similar types that have to be treated similarly in a certain situation, we can generalize the different cases by introducing a common supertype and treating this supertype only once.

For these reasons, we want to extend the flat type graphs introduced above by a *node type hierarchy* reflecting the inheritance relationships between individual node types. The following definitions of type graphs with node type inheritance are based on recent work by Bardohl et al. [9, 10]. They insert a special kind of directed edges, called *hierarchy edges*, into type graphs. The source node of a hierarchy edge represents a subtype of the target node. In order to avoid cyclic inheritance relationships, the subgraph spanned by the hierarchy edges has to be acyclic.

As a second extension to standard type graphs, they incorporate the notion of *abstract node types*, also known as *abstract classes* in object-oriented systems. An abstract node type is used as a generalization for a set of similar node types which become subtypes of the abstract type. However, only instances of the subtypes can represent real logical or physical objects while instances of the abstract supertype are merely used as *placeholders* for instances of its subtypes.

The below given definition of type graphs with inheritance edges and abstract nodes differs from that in [9, 10] because we prefer defining inheritance edges as a subset of all edges to introducing a separate inheritance graph. Afterward, we define special typing morphisms in order to type instance graphs over type graphs with inheritance. As a special kind of instance graphs, we distinguish *concrete* instance graphs which must not contain instances of abstract node types and are thus suitable to represent real system states.

**Definition 4.2 (Type Graph with Inheritance).** A type graph with inheritance is a triple  $\widehat{TG} = (TG, I, A)$  consisting of a type graph  $TG = (TG_N, TG_E, src, tar)$ , a set  $I \subseteq TG_E$  of inheritance edges, and a set  $A \subseteq TG_N$  of abstract nodes.

The subgraph  $TG|_I = (TG_N, I, src|_I, tar|_I)$  is called inheritance graph.  $TG|_I$  has to be acyclic.

For each node type  $n \in TG_N$ , the set of its subtypes, also called inheritance clan, is defined by  $clan(n) = \{n' \in TG_N \mid \exists \text{ path } n' \xrightarrow{*} n \text{ in } TG|_I\}$  where paths of length 0 are included, i. e.,  $n \in clan(n)$ .

Note that this definition also allows *multiple* inheritance, i. e., node types having more than one supertype!

Figure 4.3 shows an example of a type graph with inheritance, again depicted as UML class diagram. Abstract node types are printed in italics, and hierarchy edges are represented as a line with an hollow triangle as arrowhead. The arrowhead always points to the node representing the supertype. In the example, B1 and B2 are subtypes of *abstractB*.

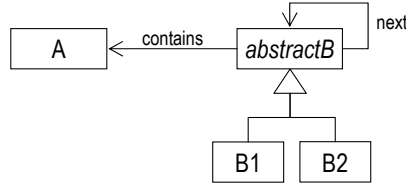


Figure 4.3: Type graph with inheritance as UML class diagram

When defining instance graphs for type graphs with inheritance, we have to allow instances of a subtype to have incoming or outgoing edges that are specified for one of its supertypes. Since such an inheritance-enabled typing cannot be expressed by graph morphisms as done for flat type graphs, the authors of [9, 10] introduce a new kind of mapping, called *clan morphism*:

**Definition 4.3 (Clan Morphism).** Given a type graph with inheritance  $\widehat{TG} = (TG, I, A)$  with  $TG = (TG_N, TG_E, src_{TG}, tar_{TG})$  and a graph  $G = (G_N, G_E, src_G, tar_G)$ ; a clan morphism  $type_G : G \rightarrow TG$  is a pair of functions  $type_G = (type_N : G_N \rightarrow TG_N, type_E : G_E \rightarrow TG_E \setminus I)$  with:

- $\forall e \in G_E : type_N \circ src_G(e) \in clan(src_{TG} \circ type_E(e))$  and
- $\forall e \in G_E : type_N \circ tar_G(e) \in clan(tar_{TG} \circ type_E(e))$ .

$type_G$  is called concrete, if  $\forall n \in G_N : type_N(n) \notin A$ .

Note that a clan morphism maps edges in  $G_E$  only to non-inheritance edges ( $TG_E \setminus I$ ). Inheritance edges are responsible for the clan constructions allowing to connect edges also to subtype instances. For a clan morphism to be called concrete, it must not map any node to an abstract node type.

If  $f : H \rightarrow G$  is a graph morphism, then  $\text{type}_G \circ f : H \rightarrow TG$  is a clan morphism, too [9]. With the help of clan morphisms, we can now provide a more general definition of instance graphs also respecting inheritance:

**Definition 4.4 (Instance Graph).** *Given a type graph with inheritance  $TG$ , a  $TG$ -typed instance graph  $\langle G, \text{type}_G \rangle$  is a graph  $G$  equipped with a clan morphism  $\text{type}_G : G \rightarrow TG$ . It is called a concrete instance graph, if  $\text{type}_G$  is a concrete clan morphism.*

*For the sake of better readability, we will write  $G$  instead of  $\langle G, \text{type}_G \rangle$  if the context reveals that we are considering instance graphs.*

In analogy to clan morphisms, we call instance graphs without abstract nodes *concrete*. While all instance graphs, also those containing abstract nodes, can be used like patterns subsuming certain conditions on system states, e. g., preconditions of a graph transformation rule (see Section 4.2), only concrete instance graphs can be used to represent concrete system states.

The two object diagrams in Fig. 4.4 provide examples of instance graphs over the type graph of Fig. 4.3. The left one (a) is not concrete as some nodes have the abstract type **abstractB**. Usually, such instance graphs are used as graph patterns which is indicated by not underlining the node identifiers in the object diagram. On the contrary, the instance graph at the right (b) is a concrete one. Due to the clan morphism construction, its nodes of type **B1** and **B2** may have edges of type **next** and **contains** as they are inherited from their common supertype **abstractB**.

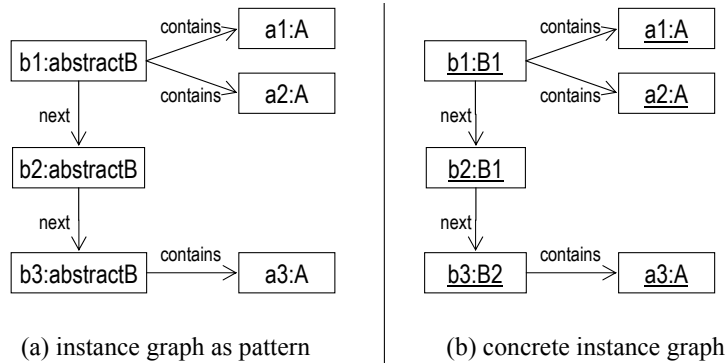


Figure 4.4: Instance graphs typed by clan morphisms



The different typings of the two instance graphs in Fig. 4.4 are related because every node of the right graph (b) has a type which is equal or a subtype of the corresponding type used in the left graph (a). As a consequence, the system states covered by (b) are a subset of the states covered by (a). This relation is formally defined in [10] as a half order relation over clan morphisms: A clan morphism is said to be a *type refinement* of another one, if it assigns more concrete node types to the same instance graph.

**Definition 4.5 (Type Refinement).** *For a given graph  $G$  and type graph  $TG$ , a clan morphism  $type'_G : G \rightarrow TG$  is a type refinement of another clan morphism  $type_G : G \rightarrow TG$ , denoted by  $type'_G \leq type_G$ , if*

- $\forall n \in G_N : type'_N(n) \in \text{clan}(type_N(n))$  and
- $type'_E = type_E$ .

Under this type refinement relation, a concrete instance graph  $\langle G, type_G^{con} \rangle$  conforms to a given pattern graph  $\langle G, type_G \rangle$ , if  $type_G^{con} \leq type_G$ .

### 4.1.2 Attributes and typed attributed graphs

Node attributes can be used to store additional information in a node. As known from object-oriented languages, an attribute consists of a name and a data value. In the context of typed graphs, we have to *declare* the attributes belonging to a certain node type by their name and data type in the type graph. In a corresponding instance graph, every instance of the node type can carry its own values for these attributes.

In connection with graph transformations, attributes can be used, e. g., as special labels that identify certain nodes along a sequence of transformation steps. They are also useful to restrict the applicability of transformation rules to situations in which involved nodes have specific attribute values. For these reasons, we will allow graph transformation rules to query current values of attributes and to assign new values during a transformation step (cf. Section 4.2).

Typed, attributed graphs and attributed graph transformations have been formally defined by Heckel et al. in [60]. They use abstract data types defined as algebras which also comprise operations performing computations on attribute values. Since we do not consider such computations on attributes, we simplify their approach concerning this aspect but, at the same time, extend it towards type graphs with node type inheritance. Node type inheritance implies that a descendant node type inherits all the attributes of its super-types. Besides, it should be possible to supplement the inherited attributes with new ones.

In contrast to earlier approaches like [93], Heckel et al. [60] regard attributed graphs as a special case of ordinary graphs. They introduce special *data nodes* that represent attribute values. Additional edges leading from ordinary nodes, now called *object nodes*, to data nodes represent the attributes.

**Definition 4.6 (Attributed Graph).** A graph  $G = (G_N, G_E, src_G, tar_G)$  is attributed over a data node set  $A$ , if  $A \subseteq G_N$  and  $\forall e \in G_E : src_G(e) \notin A$ .

Attributed graphs can occur at both levels, as type graphs and as instance graphs. In type graphs, the attributes are in fact attribute declarations, and their values are in fact sort names that determine possible values to be assigned to the attributes in an instance graph.

Similarly to the terminology for database systems, we call the set of all possible values that can be assigned to attributes a *domain*. The values belonging to the domain are partitioned into disjoint *sorts* representing different basic data types. A subset of the value domain consists of *variable names* which can be assigned to attributes instead of concrete values.

**Definition 4.7 (Domain).** A domain  $Dom = (SN, V, X, sort)$  is a tuple with a set  $SN$  of sort names, a set  $V$  of attribute values including a subset  $X \subseteq V$  of variable names, and a function  $sort : V \rightarrow SN$  associating every value and variable with a sort.

Since we want to declare node attributes in a type graph by using sort names as attribute values, type graphs are attributed over the set of sort names  $SN$ . This leads to the following extension of type graphs:

**Definition 4.8 (Attributed Type Graph).** For a given domain  $Dom = (SN, V, X, sort)$ , a  $Dom$ -attributed type graph is a type graph with node inheritance  $\widehat{TG} = (TG, I, A)$  where  $TG$  is attributed over  $SN$ .

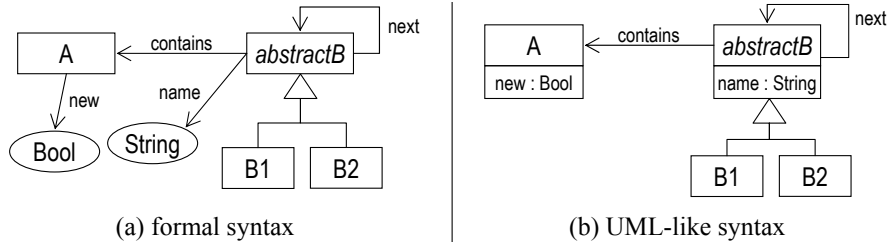


Figure 4.5: Attributed type graph

As an example, we supplement the known type graph with two attribute declarations over a domain with **String** and **Bool** as sorts. Figure 4.5(a) illustrates the formal understanding of attributes as edges leading to special data

nodes, whereas Fig. 4.5(b) uses the UML syntax printing attributes within an extra compartment of the owning class symbol. For the sake of clarity and conformance with UML, we will stick to this short-hand notation for attributes and attribute declarations in the rest of the thesis.

Instance graphs of attributed type graphs are attributed over concrete values and variables. The attribute declarations of the type graph including inherited attributes are implicitly respected by the clan morphism construction. We only have to ensure that the clan morphism maps data nodes to the right sort nodes of the type graph:

**Definition 4.9 (Attributed Instance Graph).** *Given a domain  $Dom = (SN, V, X, sort)$  and an  $Dom$ -attributed type graph  $\widehat{TG}$ , an attributed instance graph is a  $\widehat{TG}$ -typed instance graph  $\langle G, type_G \rangle$  with  $G$  attributed over  $V$  and  $type_N(v) = sort(v)$  for all  $v \in V$ .*

*$\langle G, type_G \rangle$  is a concrete attributed instance graph, if it is a concrete  $\widehat{TG}$ -typed instance graph according to Def. 4.4 and  $tar_G(e) \notin X$  for all  $e \in G_E$ .*

Note that this definition excludes variables  $x \in X$  as attribute values in concrete instance graphs. This exclusion is necessary because concrete instance graphs shall describe concrete system states, whereas variables are used as placeholders for arbitrary attribute values in graph patterns, e. g., in graph transformation rules (see Section 4.2).

Attributed instance graphs are usually infinite; e. g., for the sort  $\mathbb{N}$  of natural numbers there is a separate data node for every  $n \in \mathbb{N}$ . However, since data nodes are always kept unchanged, there is no need to explicitly represent this infinite set of data nodes. We rather represent only those data nodes that are linked to at least one object node by an attribute edge. Figure 4.6 illustrates this for an attributed version of the already known instance graph, again using the formal representation (a) and the equivalent UML syntax (b). UML object diagrams do not show the types of attributes, but they can easily be derived from the underlying class diagram.

Eventually, we extend the notion of graph morphism to attributed instance graphs as follows:

**Definition 4.10 (Morphism of Attributed Instance Graphs).** *A morphism  $f : \langle G, type_G \rangle \rightarrow \langle H, type_H \rangle$  of two instance graphs  $\langle G, type_G \rangle$  and  $\langle H, type_H \rangle$ , both attributed over  $V$ , is a graph morphism of  $G$  and  $H$  which*

- *is the identity on  $V$ ,*
- *does not map object nodes to data nodes, i. e.,  $\forall n \in G_N \setminus V : f_N(n) \notin V$ ,*
- *preserves the typing, i. e.,  $type_H \circ f = type_G$ .*

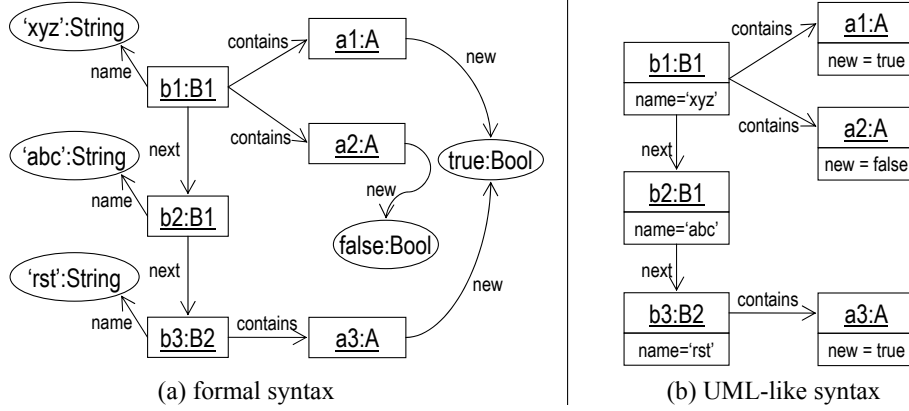


Figure 4.6: Attributed instance graph in formal and UML-like syntax

### 4.1.3 Cardinalities and other constraints

For a graph schema, a type graph is complemented with a set  $C$  of constraints which further restrict the set of valid instance graphs. In order to decide whether a given graph conforms to a schema or not, each kind of constraint has to be accompanied by a suitable notion of constraint *satisfaction*. Provided this notion, one can in principle use any constraint language or formalism; in our work, we use *cardinalities* and *OCL expressions*.

**Cardinalities.** In many cases, it is important to restrict the number of nodes which may be connected through an instance of a certain edge type. In UML class diagrams, this restriction, specified by a range of values, is called the *cardinality* of an association end. A cardinality at the target end of an edge type specifies lower and upper bounds for the number of nodes which may be connected to one source node across edges of the given edge type. A cardinality at the source end of an edge type is interpreted analogously.

Figure 4.7 revisits our type graph example and adds cardinalities to all edge types. In the UML syntax, the symbol “\*” means “unbounded”.<sup>1</sup> Thus, any instance of *abstractB* or its subtypes may **contain** an unbounded number of *A*-nodes. However, each *A*-node may only be linked to at most one *abstractB*-, *B1*-, or *B2*-node. The *next*-edges can connect nodes of the *abstractB* clan to linear lists only. For better readability, we will omit unrestricted cardinalities  $[0..*]$  in future class diagrams.

Note that cardinalities can also be used to define the multiplicities of attributes, because attributes are formally defined as special edges. In the

<sup>1</sup>formally, we define  $\forall n \in \mathbb{N} : n < *$ .

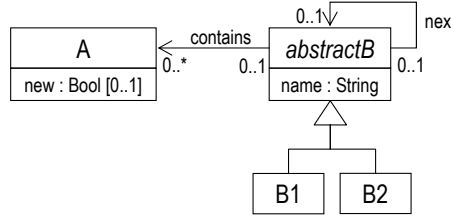


Figure 4.7: Type graph with cardinalities

example, the cardinality  $[0..1]$  for the attribute `new` in node type `A` states that it is an optional attribute. The default cardinalities for attributes are  $[0..*]$  at the source end and  $[0..1]$  at the target end of an attribute edge. This means that all instances of the object node, regardless of their number, may have at most one value for that attribute.

Given a type graph  $\widehat{TG} = (TG, I, A)$ , we formally define cardinality constraints for all non-inheritance edges  $TG_E \setminus I$  by two pairs of functions  $minSrc, minTar : TG_E \setminus I \rightarrow \mathbb{N}_0$  and  $maxSrc, maxTar : TG_E \setminus I \rightarrow \mathbb{N} \cup \{*\}$  with  $\forall e \in TG_E \setminus I : minSrc(e) \leq maxSrc(e)$  and  $minTar(e) \leq maxTar(e)$ .

In order to satisfy such cardinality constraints, we have to ensure for every node in an instance graph that the number of incoming or outgoing edges of a certain edge type remains within the given bounds:

**Definition 4.11 (Satisfaction of Cardinality Constraints).** *Given a type graph  $\widehat{TG} = (TG, I, A)$  with cardinalities  $minSrc, minTar : TG_E \setminus I \rightarrow \mathbb{N}_0$  and  $maxSrc, maxTar : TG_E \setminus I \rightarrow \mathbb{N} \cup \{*\}$ , an instance graph  $\langle G, type_G \rangle$  with  $G = (G_N, G_E, src_G, tar_G)$  and  $type_G = (type_N, type_E)$  satisfies these cardinalities, if*

- $\forall n \in G_N \forall e \in TG_E \setminus I : minSrc(e) \leq \|incoming(n, e)\| \leq maxSrc(e)$   
with  $incoming(n, e) = \{\bar{e} \in G_E \mid type_E(\bar{e}) = e \wedge tar_G(\bar{e}) = n\}$  and
- $\forall n \in G_N \forall e \in TG_E \setminus I : minTar(e) \leq \|outgoing(n, e)\| \leq maxTar(e)$   
with  $outgoing(n, e) = \{\bar{e} \in G_E \mid type_E(\bar{e}) = e \wedge src_G(\bar{e}) = n\}$ .

**OCL constraints.** More complex restrictions can be defined using the *Object Constraint Language (OCL [120])*, which is part of the UML. OCL is a formal language for expressions and constraints, e. g., invariant conditions, on UML models. While the expressiveness of OCL also allows to describe operations that, when executed, alter the system state, we apply only that part of the language which is used to add details to UML class diagrams or, in our terminology, type graphs.

For example, the following OCL expression for the type graph in Fig. 4.7 adds the restriction that an instance of node type A must always be linked to an instance of `abstractB` if the value of `new` is `false`.

```
context A inv:
  if self.new = false
  then self.abstractB->notEmpty()
```

Concerning the satisfaction of OCL constraints by an instance graph, we refer to the semantics part of the OCL language specification [120]. There it is explained how an OCL expression is evaluated in the context of a given model or, in our case, instance graph.

#### 4.1.4 Graph schemas

The combination of a type graph and constraints is called *graph schema*. A graph schema is comparable to a class diagram or ER-diagram with constraints expressing, e. g., cardinalities.

**Definition 4.12 (Graph Schema).** A graph schema  $GS = \langle Dom, \widehat{TG}, C \rangle$  consists of a domain  $Dom$ , a  $Dom$ -attributed type graph  $\widehat{TG} = (TG, I, A)$  and a set  $C$  of constraints over  $\widehat{TG}$ . The set of instance graphs over  $GS$ , denoted by  $Inst(GS)$ , is the set of all attributed instance graphs over  $\widehat{TG}$  which satisfy the constraints  $C$ .

As mentioned at the beginning of this chapter, graph schemas can be used as an alternative to graph grammars for defining formal, graph-based languages. The set of allowed graphs of a language is not generated by production rules but determined by the set of valid instance graphs. This approach has recently become popular in the context of meta-modeling; for example, the UML meta-model [121] can be regarded as a graph schema that defines the set of all valid UML models, and every UML model can formally be expressed as an instance of the UML meta-model.

## 4.2 Graph transformations

While a graph schema determines the set of valid instance graphs, graph transformation rules are used to define local transformations of graphs. In this thesis, we follow the algebraic *double pushout approach* (DPO) to graph transformation as first introduced by Ehrig et al. for untyped graphs in [39]. In [30] from the *Handbook of Graph Grammars* [134], the interested reader can find further insights into the basic mechanisms of the DPO approach.

The DPO version for *typed* graphs was first introduced by Corradini et al. in [29]. In this section, we propose another DPO variant for typed, attributed graphs with inheritance. It is based on two separate extensions of Corradini's typed DPO approach: At first, Heckel et al. extended it to attributed graphs but without node type inheritance [60], and, recently, Bardohl et al. proposed an extension for node type inheritance but without attributes [9, 10]. By combining these two extensions, we achieve graph transformation systems which smoothly fit to the notion of graph schema defined in the previous section.

### 4.2.1 Informal introduction and example

Graph transformation systems are defined by a set of rules which consist of a left-hand side and a right-hand side each. Like other rule-based specification languages, they adhere to the following *recognize-select-execute* paradigm:

1. *Recognize*: Determine the set of currently enabled rules by computing all or some occurrences of their left-hand sides in the current state.
2. *Select*: If *recognize* returns more than one applicable rule or more than one occurrence, select one rule and one occurrence of this rule for execution (following a predefined strategy).
3. *Execute*: Replace the selected occurrence of the left-hand side of the selected rule by a copy of its right-hand side.

In the case of graph transformation rules, the left-hand side and the right-hand side are instance graphs  $L$  and  $R$ . The left-hand side represents the pre-conditions of the rule while the right-hand side describes its effects and post-conditions. Both graphs do not have to be concrete, i.e., their nodes may be typed by abstract types and their attributes may have variables as values.

According to the aforementioned paradigm, such a rule can be applied to a concrete instance graph  $G$ , also called *host graph*, whenever there is an *occurrence* of the left-hand side  $L$  in  $G$ . In this context, occurrence means a subgraph of  $G$  which has the same structure as  $L$  and whose elements conform to the typing and attribute values of  $L$ . Sometimes, such an occurrence is also called a *match* of  $L$  in  $G$ .

If an occurrence of  $L$  has been found in  $G$ , the rule can be applied as follows: At first, we remove those elements from the occurrence that do not appear in the right-hand side  $R$  of the rule. Then, we use a certain *embedding* mechanism to merge the remaining graph  $D$  with an isomorphic copy of  $R$ .

Note that – in the presence of node type inheritance – we have to allow nodes in the occurrence of  $L$  to be typed by subtypes of their preimage types in  $L$ . In particular, if  $L$  contains nodes with abstract types while the host graph  $G$  is concrete, then these abstract nodes can only be matched with nodes of concrete subtypes in  $G$ .

Furthermore, in the presence of attributed graphs we require that if attributes with concrete values are specified in  $L$ , the corresponding attributes in the occurrence of  $L$  must have the same values. If variables are used as attribute values in  $L$ , they can be matched to arbitrary values of the same sort the variable belongs to. However, if a variable is used several times in  $L$  or  $R$ , we require that all its matchings in  $G$  have the same value.

**Example.** At the top, Fig. 4.8 depicts an example of a graph transformation rule. According to its left-hand side  $L$ , it requires as pre-condition the existence of a node of type **abstractB** and two nodes of type **A**. One of them has to be **contained** in the **abstractB**-node, and its attribute **new** has to be set to **false**, whereas the attribute value of the second **A**-node has to be **true**. According to the right-hand side  $R$ , applying this rule deletes the first **A**-node and connects the second one to the **abstractB**-node instead.

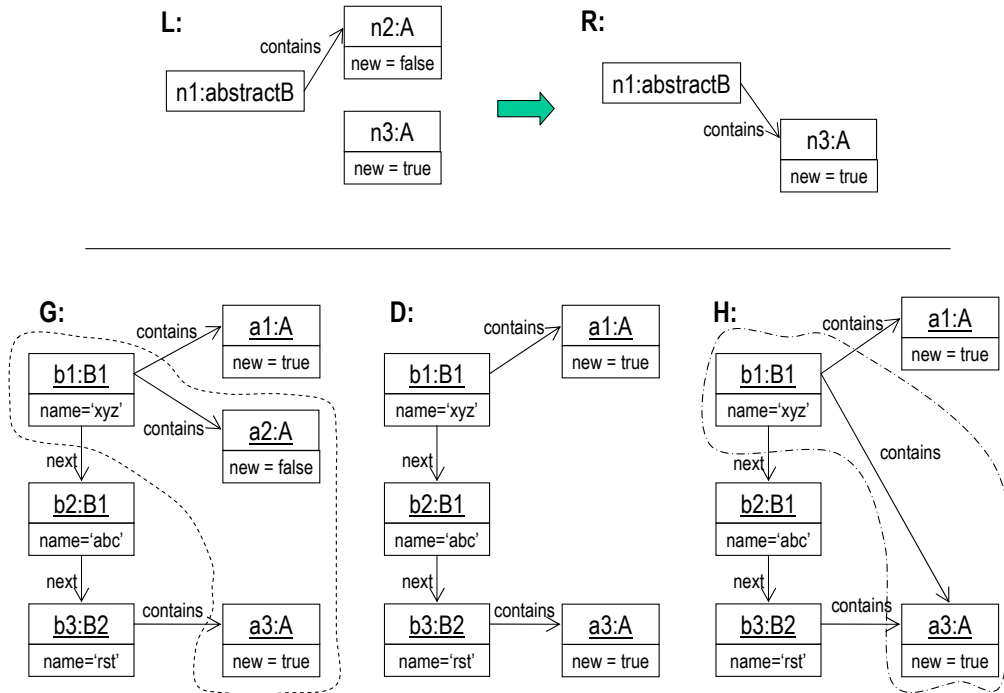


Figure 4.8: Graph transformation rule and its application



As depicted at the bottom of Fig. 4.8, the exemplary transformation rule can be applied to the instance graph of Fig. 4.6, here called  $G$ . One of the possible occurrences of the left-hand side  $L$  in  $G$  is marked by the dashed line. The occurrence has the same structure as  $L$  and, due to inheritance (cf. type graph in Fig. 4.5), compatible types and attribute values.

Note that the node identifiers of  $L$  and its occurrence in  $G$  do not have to be equal, because the rule represents a reusable pattern which is applicable to any proper match, regardless of node identifiers. Instead, node identifiers play an important role for identifying corresponding nodes of the left- and right-hand side of a rule.

The graph  $D$  at the bottom of Fig. 4.8 shows the intermediate stage of the transformation where node **a2**, the match of node **n2** in  $G$ , has already been removed. Eventually, graph  $H$  shows the final result of the transformation with node **a3**, the match of node **n3**, being connected to node **b1**.

**Negative application conditions.** So far, the left-hand side  $L$  of a transformation rule specifies which pattern has to be present in a host graph before the rule can be applied. On the contrary, we sometimes require that certain elements are *not* present when a rule is applied. In such a case, we add *negative application conditions* (NAC) to a transformation rule [56].

For example, remember the above illustrated transformation rule and its application to graph  $G$ . The rule creates a new **contains**-edge to node **n3**. Since **n3** is matched to node **a3** which already has an incoming **contains**-edge in  $G$ , **a3** eventually owns two incoming **contains**-edges in the result graph  $H$ . Obviously, this outcome is not desired because, as shown in the graph schema of Fig. 4.7, we want A-nodes to be contained in at most one B-node only.

In order to avoid such undesired matchings, we specify negative application conditions by so-called *forbidden graphs*. A transformation rule can only be applied to an occurrence of its left-hand side, if the occurrence cannot be extended to the forbidden graph.

In our example, we have to prevent a matching of node **n3** to an A-node that is already linked with a B-node. The corresponding forbidden graph is shown in Fig. 4.9(a). An alternative representation directly embedding the NAC into the left-hand side of the transformation rule is shown in Fig. 4.9(b). Under this NAC, the rule matching in Fig. 4.8 would not be valid any more, since the forbidden node **n2** could be matched with **b3**.

Sometimes, one also encounters explicitly given *positive* application conditions [56]. However, as one can simply add elements whose existence is required before a rule can be applied to the left- and right-hand sides of the rule, such positive application conditions do not need any special treatment.

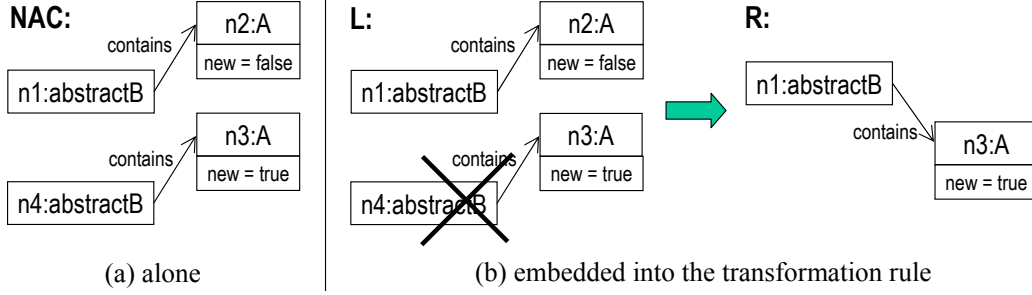


Figure 4.9: Negative application condition

### 4.2.2 Operational semantics

In order to formally reason about graph transformations and to make the transformation rules executable, we have to provide formal, operational semantics. In particular, the embedding mechanism by which the right-hand side  $R$  of a rule is incorporated into the intermediate graph  $D$  has to be precisely defined.

The algebraic approach we apply in this thesis realizes the embedding by identifying, or *gluing*, corresponding parts of  $R$  and  $D$ . The glued elements in  $D$  are exactly those from the occurrence of  $L$  which have not been deleted. They can be determined by another gluing of corresponding parts of  $L$  and  $D$ . The approach is called *Double Pushout Approach (DPO)* [39], because each of the two gluings is realized by an algebraic construction called *pushout*.

Formally, a DPO graph transformation rule consists of three instance graphs  $\langle L, type_L \rangle$ ,  $\langle K, type_K \rangle$ , and  $\langle R, type_R \rangle$ , all typed over a fixed attributed type graph  $\widehat{TG}$ . The *gluing graph*  $K$  specifies the gluing elements which are read during a rule application but not modified. The items of  $L$  which are not in  $K$  have to be deleted, and the items in  $R$  which are not in  $K$  have to be created. The graphs are connected by a pair of injective graph morphisms  $(L \xleftarrow{l} K \xrightarrow{r} R)$ , called *rule span*.

**Definition 4.13 (Graph Transformation Rule).** *For a fixed domain  $Dom = (SN, V, X, sort)$ , a graph transformation rule over a  $Dom$ -attributed type graph  $\widehat{TG}$  is given by  $p = (L \xleftarrow{l} K \xrightarrow{r} R, type, NAC)$ , where*

- $L \xleftarrow{l} K \xrightarrow{r} R$  is a rule span with injective graph morphisms  $l, r$  and graphs  $L, K, R$  attributed over  $V$ ,
- $type = (type_L : L \rightarrow TG, type_K : K \rightarrow TG, type_R : R \rightarrow TG)$  is a triple of clan morphisms, and

- $NAC$  is a set of triples  $nac = (N, n, type_N)$  with  $N$  being a graph,  $n : L \rightarrow N$  a graph morphism, and  $type_N : N \rightarrow TG$  a clan morphism,

such that the following conditions hold:

1.  $type_L \circ l = type_K = type_R \circ r$
2.  $type_{R,N}(R'_N) \cap A = \emptyset$ , where  $R'_N := R_N \setminus r_N(K_N)$
3.  $type_N \circ n \leq type_L$  for all  $(N, n, type_N) \in NAC$
4.  $l, r$ , and all  $n$  are identities on the data nodes  $V$ .

The diagram in Fig. 4.10 visualizes the definition with arrows depicting graph morphisms and dashed arrows depicting clan morphisms.  $N$  is the forbidden graph of a negative application condition where  $N \setminus L$  represents the forbidden structure.

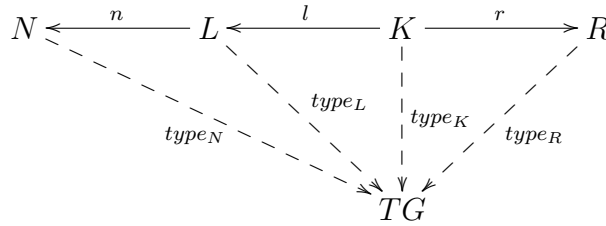


Figure 4.10: Algebraic constituents of a graph transformation rule

The first condition in Definition 4.13 states that those elements in  $L$  and  $R$  that have a preimage in  $K$  under  $l, r$  have to have the same type as their preimage in  $K$ . The second condition states that new nodes in the right-hand side  $R$  which are to be created by a rule application must not have abstract types. This is due to the fact that rules must not create instances of abstract types since they operate on concrete graphs. The third condition requires that the typing of a forbidden graph has to be compatible with the typing of the left-hand side (cf. Definition 4.5). Eventually, the fourth condition requires the rule span to keep data nodes unchanged.

If the intersection  $L \cap R$  of left- and right-hand sides is well-defined, meaning that edges which appear on both sides are connected to the same nodes and that nodes with the same identifier on both sides have the same type, etc., then  $K$  is implicitly given by  $L \cap R$ , and we can visualize a rule without  $K$  as shown in Fig. 4.9.

A graph transformation rule can be applied to a concrete instance graph  $G$  if there is an occurrence of the left-hand side as follows:

**Definition 4.14 (Occurrence of a Rule's Left-Hand Side).** Let  $p = (L \xleftarrow{l} K \xrightarrow{r} R, \text{type}, \text{NAC})$  be a graph transformation rule typed over  $\widehat{TG} = (TG, I, A)$  and attributed over  $\text{Dom} = (SN, V, X, \text{sort})$ . Let  $\langle G, \text{type}_G \rangle$  be a concrete, attributed instance graph over  $\widehat{TG}$ . A graph morphism  $o_L : L \rightarrow G$  is an occurrence of  $L$  with respect to  $p$  and  $\langle G, \text{type}_G \rangle$ , if

1. for all  $x_1, x_2 \in L$  with  $x_1 \neq x_2$  and  $o_L(x_1) = o_L(x_2)$  there are  $y_1, y_2 \in K$  such that  $l(y_1) = x_1 \wedge l(y_2) = x_2$  (**identification condition**),
2.  $\forall e \in G_E : (\text{src}_G(e) \in \text{Del}_N \vee \text{tar}_G(e) \in \text{Del}_N) \implies e \in \text{Del}_E$   
with  $\text{Del}_N := o_L(L_N \setminus l_N(K_N))$  and  $\text{Del}_E := o_L(L_E \setminus l_E(K_E))$   
(**dangling condition**),
3.  $\text{type}_G \circ o_L \leq \text{type}_L$ ,
4.  $o_L$  is the identity on  $V \setminus X$ , and
5.  $o_L$  satisfies NAC, i. e., for each  $\text{nac} = (N, n, \text{type}_N) \in \text{NAC}$  there is no graph morphism  $f : N \rightarrow G$  such that  $f \circ n = o_L$  and  $\text{type}_G \circ f \leq \text{type}_N$ .

The first two conditions of this definition form the so-called *gluing condition* of the DPO approach. They ensure the existence of the double pushout construction which is required for computing a transformation step according to Definition 4.15. The *identification condition* states that any two objects from the left-hand side  $L$  may only be identified with each other in the occurrence  $o_L(L)$ , if they also belong to the gluing graph  $K$  (i. e., if they are preserved). The *dangling condition* requires that the intermediate graph obtained by removing all elements from  $G$  which have to be deleted is indeed a graph, i. e., no edges are left “dangling” without source or target node.

The third condition states that the typing of the occurrence of  $L$  has to be a refinement of the typing of  $L$  (cf. Definition 4.5). The fourth condition states that all data nodes with concrete values remain unchanged. Altogether, we achieve equality of attribute values in the occurrence to those specified in the rule. Besides, variables will be mapped to concrete values, since variables do not occur in the concrete host graph.

The last condition ensures that a rule is not applied to an occurrence which violates a negative application condition. It requires to check if a forbidden graph can be mapped to the host graph in such a way that the mapping also includes the occurrence.

For valid occurrences of the left-hand side, the application of a rule is defined by two steps: At first, we compute a categorical double pushout diagram [30] in the category **Graph** of plain graphs and graph morphisms

as defined in Def. 4.1. Since attributes are represented by special nodes and edges, attributed graphs are implicitly covered [60].

In a second step, we construct the typing morphisms for the resulting instance graphs [9]. This way, we can save the introduction of a separate category for typed graphs as done in [29] for typed graphs without inheritance.

**Definition 4.15 (DPO Graph Transformation).** *Let  $p$  be a graph transformation rule  $p = (L \xleftarrow{l} K \xrightarrow{r} R, \text{type}, \text{NAC})$  typed over  $\widehat{TG} = (TG, I, A)$  and attributed over  $\text{Dom} = (SN, V, X, \text{sort})$ , and let  $\langle G, \text{type}_G \rangle$  be a concrete, attributed instance graph of  $\widehat{TG}$ .*

*Given an occurrence  $o_L : L \rightarrow G$ , the rule  $p$  can be applied, yielding a direct (DPO) transformation step  $\langle G, \text{type}_G \rangle \xrightarrow{p(o)} \langle H, \text{type}_H \rangle$  to a concrete instance graph  $\langle H, \text{type}_H \rangle$ , as follows:*

1. *Construct the double pushout as given by the categorical diagram  $o$  in Fig. 4.11, where top and bottom are rule spans and (1), (2) are pushouts in the category **Graph** of graphs and graph morphisms. Similarly to  $l, r$ , we require that  $g, h$  are identities on all data nodes  $v \in V$ .*
2. *Construct concrete clan morphisms  $\text{type}_D$  and  $\text{type}_H$  as follows*

- $\text{type}_D = \text{type}_G \circ g$
- $\text{type}_H(x) = \begin{cases} \text{type}_D(x') & \text{if } \exists x' \in D : x = h(x') \\ \text{type}_R(x'') & \text{else, with } x'' \in R \wedge x = o_R(x'') \end{cases}$   
for all  $x \in H_N \cup H_E$ .

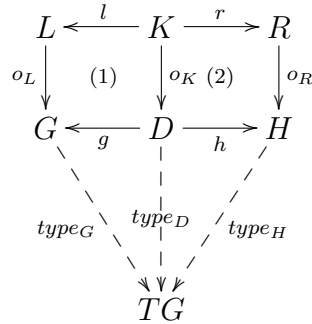


Figure 4.11: Double Pushout Diagram  $o$

Note that  $\text{type}_H$  is a well-defined clan morphism with  $\text{type}_H \circ h = \text{type}_D$  and  $\text{type}_H \circ o_R \leq \text{type}_R$  [9].

The DPO diagram  $o$  is a categorical way of representing the occurrence of a rule in a bigger context. Operationally, it formalizes the replacement of

a subgraph in a graph by two pushouts. The left-hand side pushout (1) is responsible for removing the occurrence of  $L \setminus l(K)$  in  $G$ , resulting in graph  $D$ . The right-hand side pushout (2) adds a copy of  $R \setminus r(K)$  to  $D$  leading to the derived graph  $H$ . If the occurrence  $o_L$  satisfies the gluing conditions stated in Def. 4.14, then the two pushouts exist and can be uniquely computed up to isomorphism [30].

**Pushout construction in the category **Graph**.** In order to complete the operational semantics of transformation rules, we will now briefly explain how the double pushout construction presented above can in fact be computed. Based on the general definition of a pushout in Category Theory, a pushout for the category **Graph** of graphs and graph morphisms can be defined as follows (cf. [30, 90]):

**Definition 4.16 (Pushout in the Category **Graph**).** *Given graphs  $A, B, C$  and graph morphisms  $b : A \rightarrow B$  and  $c : A \rightarrow C$ , a pushout of  $\langle b, c \rangle$  is a triple  $(D, g : B \rightarrow D, f : C \rightarrow D)$  of pushout graph  $D$  and graph morphisms  $f, g$  as in diagram 4.12, with*

1.  $g \circ b = f \circ c$  (**commutativity**),
2. for all graphs  $D'$  and graph morphisms  $g' : B \rightarrow D'$  and  $f' : C \rightarrow D'$  with  $g' \circ b = f' \circ c$  there exists a unique graph morphism  $h : D \rightarrow D'$  such that  $h \circ g = g'$  and  $h \circ f = f'$  (**universal property**).

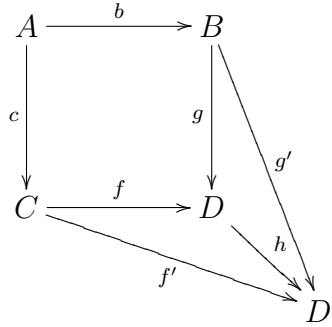


Figure 4.12: Pushout in the category **Graph**

Intuitively, a pushout is a generalized union that specifies how to merge two graphs  $B$  and  $C$  along a common subgraph  $A$ . In a first step, we compute the disjoint union of  $B$  and  $C$  and then we start to glue those nodes and edges that have common preimages under  $b$  and  $c$ . For this purpose, we compute the least equivalence relation  $\approx$  such that for all  $a \in A : b(a) \approx c(a)$ . Based

on this relation, we merge those items in the union graph that belong to the same equivalence class respectively.

The construction of a pushout is illustrated in Fig. 4.13. The graph morphisms  $b, c$  are partially indicated by dotted arrows for their node mappings. The equivalence classes of  $\approx$  can be computed by iterating over all nodes of the gluing graph  $A$ . We use the equivalence classes as node identifiers in the pushout graph  $D$  in order to highlight which nodes have been merged. The edges are merged similarly. Eventually, the graph morphisms  $f, g$  map each node and edge to its equivalence class.

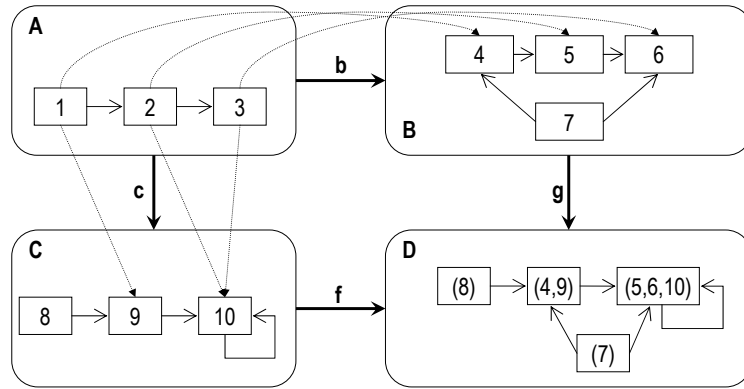


Figure 4.13: Construction of a pushout

This way, the pushout can uniquely be computed up to isomorphism for any two graph morphisms with common gluing graph (a proof is beyond the scope of this thesis). In particular, we can apply this technique to compute pushout (2) of the DPO diagram in Fig. 4.11.

However, the situation is different for pushout (1) of that diagram because its pushout graph  $G$  is already given; instead, the missing graph  $D$  is called *pushout complement*.

**Definition 4.17 (Pushout Complement in the Category Graph).** Given graphs  $A, B, D$  and graph morphisms  $b : A \rightarrow B$  and  $g : B \rightarrow D$ , a pushout complement of  $\langle b, g \rangle$  is a triple  $(C, c : A \rightarrow C, f : C \rightarrow D)$  with pushout complement graph  $C$  and graph morphisms  $c, f$  such that  $(D, g : B \rightarrow D, f : C \rightarrow D)$  is a pushout as in the diagram of Fig. 4.12.

The pushout complement graph is constructed by copying all nodes and edges from  $D$  except for those that are in  $B$  but not in  $A$  under the respective graph morphism. Thus, we receive

- $C_N = D_N \setminus g_N (B_N \setminus b_N (A_N))$

- $C_E = D_E \setminus g_E(B_E \setminus b_E(A_E))$
- $src_C = src_D|_{C_E}$
- $tar_C = tar_D|_{C_E}$

The graph morphism  $f$  has to map every node and edge in  $C$  to its origin in  $D$ , and the graph morphism  $c$  maps every node and edge in  $A$  to its image under  $f^{-1} \circ g \circ b$  (cf. pushout diagram in Fig. 4.12).

According to [30], the existence of a pushout complement is not always guaranteed. However, the gluing condition stated in Definition 4.14 characterizes a sufficient condition for its existence. Since an occurrence of a rule has to satisfy this condition, we can always compute the desired pushout complement for pushout (1) in Definition 4.15 in the above way. Moreover, since the graph morphism  $l$  shown in the same diagram is injective, the resulting pushout complement graph  $D$  of  $l$  and  $o_L$  is unique [30].

In summary, the application of a transformation rule  $p : L \rightsquigarrow R$  is performed in three steps:

1. Find an occurrence of the left-hand side  $L$  in the current graph  $G$ , formally a graph morphism  $o_L : L \rightarrow G$ , respecting the conditions stated in Definition 4.14.
2. Construct the intermediate graph  $D$  as pushout complement by removing all nodes and edges from  $G$  which are matched by  $L \setminus R$ . Since the occurrence chosen in the first step has to satisfy the dangling condition [39], we can be sure that the remaining structure  $D := G \setminus o_L(L \setminus R)$  is still a legal graph, i. e., that no edges are left dangling because of the deletion of their source or target nodes. All elements in  $D$  preserve their types from  $G$ .
3. Construct the result graph  $H$  by gluing  $D$  with a copy of  $R \setminus L$ . Nodes and edges which are already existent in  $D$  preserve their types in  $H$ . Newly created elements receive their types according to  $R$ . Note that these types are not abstract due to Definition 4.13. Thus, the resulting graph  $H$  is in fact a concrete instance graph. Moreover, we assume that all newly created nodes and edges get fresh identities, so that  $G \cap H$  is well-defined and equal to the intermediate graph  $D$ .



### 4.2.3 Graph transformation systems

Combining a graph schema and a set of graph transformation rules that operate on instance graphs of the schema results in a graph transformation system as follows:

**Definition 4.18 (Graph Transformation System).** *A graph transformation system  $\mathcal{G} = (GS, P)$  consists of a graph schema  $GS = (Dom, \widehat{TG}, C)$  and a set  $P$  of  $\widehat{TG}$ -typed, attributed graph transformation rules.*

*$\mathcal{G}$  is called consistent, if for all possible transformation steps  $G \xrightarrow{p(o)} H$  with  $p \in P$  and  $G \in Inst(GS)$  it is also  $H \in Inst(GS)$ .*

*A transformation sequence  $s = (G_0 \xrightarrow{p_1(o_1)} \dots \xrightarrow{p_n(o_n)} G_n)$  in  $\mathcal{G}$ , also denoted by  $G_0 \xrightarrow{*}_{\mathcal{G}} G_n$ , is a sequence of consecutive transformation steps such that  $\forall i = 0 \dots n : G_i \in Inst(GS)$  and  $\forall i = 1 \dots n : p_i \in P$ .*

If we assume that fresh identifiers are given to newly created elements, i. e., ones that have not been used before in a transformation sequence, then for any  $i < j \leq n$  the intersection  $G_i \cap G_j$  is well-defined and represents that part of the structure which has been preserved in the transformation from  $G_i$  to  $G_j$ . Formally, if  $x \in G_i \cap G_j$  for  $0 \leq i < j \leq n$  then  $x \in G_k$  for all  $i \leq k \leq j$ .

Note that the graphs a transformation rule consists of have to be typed over  $\widehat{TG}$  but do not necessarily have to be an instance of the graph schema  $GS$ . In other words, the transformation rules are not required to satisfy the constraints  $C$ !

Nevertheless, the transformation system can still be *consistent* in the sense that valid schema instances are always transformed into valid schema instances. However, a formal proof if a given transformation system is consistent or not is beyond the scope of this thesis. Again, the difficulty is not caused by the typing – all transformation results are valid instance graphs over  $\widehat{TG}$  by definition – but by the additional constraints, which can be given in any suitable formalism. For the rest of this thesis, we assume that the designer of a graph transformation system takes care of its consistency.

## 4.3 Transformation-based system models

After this introduction of DPO graph transformations for typed, attributed graphs with inheritance, we will now look at the possibility to use such transformation systems for modeling and analyzing dynamic software systems.

In software engineering, graph transformation systems can be used for various specification and modeling purposes. Applications range from speci-

fying visual languages to abstract data types, object-oriented programs, and software architectures (for instance, see [37]). In our case, we will use them to describe models of architectural styles, as detailed in Chapter 5.

Many of these applications have in common that graphs represent individual system states, and a transformation step represents the evolution from one state to a new state. In order to model the behavior of a software system in this way, we have to provide a dedicated start graph as follows:

**Definition 4.19 (Graph Transformation Model).** *A graph transformation model  $M = (\mathcal{G}, G_0)$  consists of a graph transformation system  $\mathcal{G} = (GS, P)$  and a fixed start graph  $G_0 \in \text{Inst}(GS)$ . The operational semantics of a model  $M$  is given by the set  $\text{Op}(M)$  of all transformation sequences  $G_0 \xRightarrow{*}_{\mathcal{G}} G_n$  in  $\mathcal{G}$  that start with  $G_0$ .*

This definition looks similar to that of a *graph grammar*. However, we do not distinguish between terminal and non-terminal elements here. Moreover, we do not define the semantics only in terms of a *language* consisting of all derivable terminal graphs, but we are also interested in the derivation paths of all reachable graphs, i. e., the transformation sequences.

### 4.3.1 Reachability properties and parsing problems

System models in general and graph transformation models in particular shall allow the engineer to reason about properties of the system at an early stage of the development. Since graph transformation models specify an initial system state and rules according to which the system may evolve, an obvious question to be analyzed is: Can the system evolve into a certain predefined target state? In the context of a graph transformation model  $M = (\mathcal{G}, G_0)$ , the same question would read: Is a certain target state  $G_T$  *reachable* from the start graph  $G_0$  by a transformation sequence in  $\mathcal{G}$ ? In the following, we call this kind of property *reachability property*.

In the case of graph grammars, the analogous problem is called the *membership problem*: Given a graph grammar and a graph  $G$ , the question is whether  $G$  is a member of the language generated by the grammar or not. In many cases, we also wish to *parse* the graph, i. e., to find a derivation tree with the start graph as its root.

If we had an algorithm solving the membership problem, it could certainly be adopted to analyze reachability properties of transformation models, too. However, according to [136], the membership problem is already undecidable for unrestricted forms of string grammars (so-called *type-0 grammars*). As a consequence, it is not decidable for unrestricted graph grammars, either,

because otherwise we could express strings as simple graphs and solve the membership problem for type-0 languages this way.

As the membership problem is only decidable for restricted forms of graph grammars, many existing approaches aim at efficient parsing algorithms for grammars that are as unrestricted as possible. One promising example is the proposal of *layered graph grammars* by Rekers and Schürr [128]. They apply a certain layering function to associate a certain numbering to all nodes and edges, and they formulate certain restrictions over graph production rules with respect to this numbering. Despite these restrictions, the expressiveness of layered graph grammars goes beyond context-free graph grammars and later extensions also allow complex constructs like negative application conditions [19].

However, the imposed layering restrictions still require a production-like rule character which fits more to graph grammars generating a graph-based language than to graph transformation models specifying system behavior. For instance, it is required that every rule creates at least one new element. Therefore, it becomes hard to model operations which just delete a certain link or object from the current system state. Consequently, layered graph grammars are inappropriate for modeling, e.g., object-oriented systems or dynamic software architectures.

A parsing algorithm for layered graph grammars would invert the production rules of the grammar and try to reduce a given graph to the predefined start graph by applying the inverted rules. Whenever there is no rule applicable any more but the start graph is not yet reached, back-tracking is used to find an alternative reduction. The rationale behind the layering is to achieve a certain order in which a parser applies the inverted rules. This order prevents cyclic transformations and guarantees termination.

In the unrestricted case without constraints such as the layering, a parser could run into an infinite loop, e.g., when two converse rules alternately create and delete the same element. This problem could be avoided if the parser stored information about all already generated intermediate graphs. Whenever it reaches an already known graph, it can trigger a back-tracking step and try another derivation. This way, the membership problem becomes decidable for the unrestricted case, but only if the graph language is finite! The information about already visited graphs can be stored in a *graph transition system* as discussed in the next subsection.

### 4.3.2 Graph transition systems

Transition systems are frequently used to represent the behavior semantics of software systems. They divide the runtime evolution of a system into

discrete *states* and use a binary *transition relation* to define possible state changes. In the case of *graph transition systems* [131], one considers graphs as representations of system states. If used as operational model of a graph transformation model, its state space contains all reachable graphs of the transformation model. Such a transition system can be generated by recursively applying all enabled graph transformation rules to the start graph. If the resulting state space of the graph transition system is finite, we can easily solve the aforementioned reachability properties, even for unrestricted forms of graph transformation systems, by searching the state space.

When deriving a graph transition system from a graph transformation model, we have to be aware that the algebraic formalization of graph transformations by the double pushout construction characterizes the derived graph of a transformation step only up to isomorphism (see [30]). Indeed, for a given rule and occurrence there may exist an infinite number of results, all isomorphic copies of each other. This is, of course, a disaster for state space analysis.

Moreover, most analysis techniques based on state transition systems suffer from computational complexity and the state explosion problem. Therefore, it is desired to keep the state space as small as possible. A general approach is to look for suitable abstraction techniques that allow to reduce the state space while preserving the same behavior with respect to certain properties. In the context of graph transformations, the behavior is determined by the applicability of graph transformation rules in a certain state, and the set of applicable rules remains the same for isomorphic graphs. Thus, we can exploit the symmetry of isomorphic graphs to reduce the state space.

For these reasons, we build the state space of a graph transition system not from concrete graphs but from *isomorphism classes* of graphs:

**Definition 4.20 (Isomorphism Class of Typed, Attributed Graphs).**

Given a graph schema  $GS$ , two instance graphs  $\langle G, type_G \rangle, \langle H, type_H \rangle$ , both attributed over data nodes  $V$ , are isomorphic,  $\langle G, type_G \rangle \cong \langle H, type_H \rangle$ , if there is an isomorphism  $i : \langle G, type_G \rangle \rightarrow \langle H, type_H \rangle$  according to Definitions 4.1 and 4.10.

An isomorphism class  $[\langle G, type_G \rangle]$  is the set of all isomorphic instance graphs:  $[\langle G, type_G \rangle] = \{ \langle H, type_H \rangle \mid \langle G, type_G \rangle \cong \langle H, type_H \rangle \}$

For the sake of better readability, we use the abbreviation  $[G]$  instead of  $[\langle G, type_G \rangle]$  if the typing morphism is not relevant.

Using isomorphism classes as states and transformation steps as transitions, a graph transformation model induces the following graph transition system:

**Definition 4.21 (Graph Transition System).** For a given graph transformation model  $M = (\mathcal{G}, G_0)$  with  $\mathcal{G} = (GS, P)$ , a graph transition system  $GTS(M) = (L, S, \Rightarrow, s_0)$  is a labeled transition system with

- $L = P$  the set of labels (with the rules in  $P$  represented by their names),
- $S = \{[G] \mid G_0 \xRightarrow{*}_{\mathcal{G}} G\}$  the set of states, each representing the isomorphism class of a graph  $G$  which is reachable in  $\mathcal{G}$  from  $G_0$ ,
- $\Rightarrow \subseteq S \times L \times S$  the transition relation, which is defined by lifting the transformation relation on graphs to isomorphism classes:  $[G] \xRightarrow{p} [H]$  iff  $G \xRightarrow{p(o)} H$  is a transformation step in  $\mathcal{G}$ ,
- $s_0 = [G_0] \in S$  the initial state.

The graph transition system  $GTS(M)$  can be generated from a model  $M = (\mathcal{G}, G_0)$  by recursively applying all enabled graph transformation rules of  $\mathcal{G}$  at each state and by matching the resulting graphs with already generated isomorphic graphs.  $GTS(M)$  can be considered as an operational model of  $M$ , because the paths in  $GTS(M)$  exactly correspond to the transformation sequences in  $Op(M)$  up to isomorphism.

**Example.** Consider a graph transformation system consisting of the type graph of Fig. 4.7 and the two transformation rules depicted in Fig. 4.14. The first rule  $p_1$ , already known from Fig. 4.9, removes an old A-node (**new** = **false**) which is attached to a B-node and connects another A-node to the B-node instead. The second rule  $p_2$  sets the **new** attribute of a contained A-node to **false** and creates a new A-node (**new** = **true**).

Based on this transformation system, consider a model with an initial system state  $G_0$  having just two B-nodes and one A node as shown at the left of Fig. 4.15. In this simple example, the only possibility to transform the initial graph results in an alternating sequence of  $p_2$  and  $p_1$  applications as indicated by Fig. 4.15. Rule  $p_2$  creates a new A-node ( $G_1$ ), and rule  $p_1$  replaces the old A-node by the new one ( $G_2$ ). Then, rule  $p_2$  could be applied again, and so forth.

However, if we want to generate the graph transition system induced by this model, we have to note that  $G_0$  and  $G_2$  are isomorphic graphs and, thus, belong to the same isomorphism class. Analogously, a potential successor of  $G_2$  in the sequence would fall into the same isomorphism class as  $G_1$ . Hence, we can conclude that any transformation sequence of the model consists of alternating switches between the two isomorphism classes  $[G_0]$  and  $[G_1]$ . As

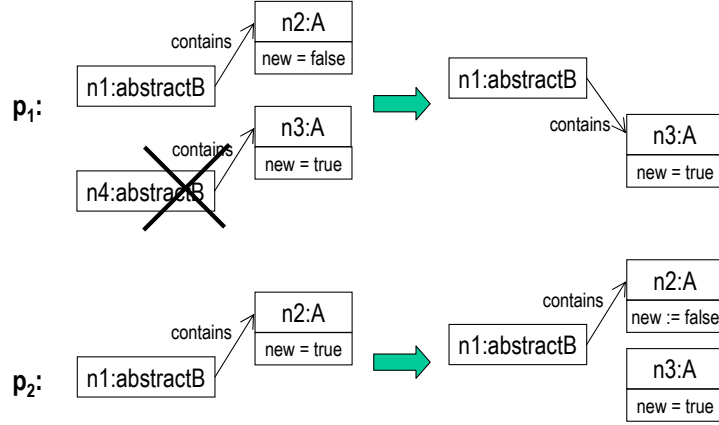


Figure 4.14: Two exemplary graph transformation rules

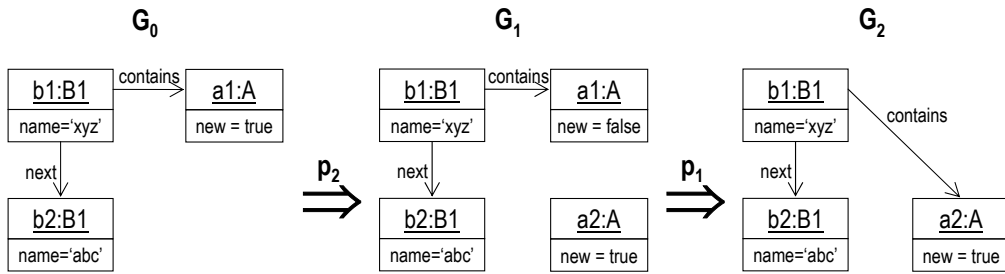


Figure 4.15: Transformation sequence

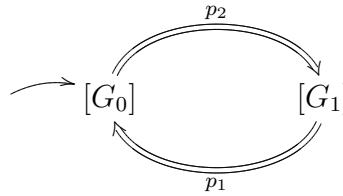


Figure 4.16: Induced graph transition system

we use isomorphism classes as states of graph transition system, the transition system induced by this model looks like Fig. 4.16.

As in the above example, we can derive graph transition systems from any graph transformation model. Besides deciding simple reachability questions, the transition systems are also open to further analysis as done, e. g., in the work by A. Rensink [131]. He intends to apply *model checking* techniques for the verification of certain system properties. In contrast to current model checking approaches which are based on propositional logic with a fixed number of propositions, he proposes an extended temporal logic which also covers allocation and deallocation (with Distefano and Katoen [34]). This is required to handle, e. g., models of object-oriented software systems with unbounded dynamic object creation and garbage collection.

In [131], Rensink proposes a further extension of the logic studied in [34] that also includes navigation expressions over graphs. This reflects the fact that, in contrast to labeled transition systems with simple states, the states of a graph transition system are complex structures which require additional means for expressing desired properties.

In this thesis, we specify dynamic software systems by graph transformation models and reason about them based on graph transition systems, too. But, instead of analyzing a single transition system, we mainly focus on relationships between different transition systems representing the same behavior but at different levels of abstraction. The intention is to allow for computer-aided checks whether one system refines the behavior semantics of the other one in terms of behavior preservation and observational substitutability.





## Chapter 5

# Architectural styles as graph transformation systems

After having summarized the relevant graph transformation theory, we now want to move to its practical application, i. e., to platform-specific architectural styles defined as graph transformation systems [11, 12].

Sections 5.1 and 5.2 reveal how a graph transformation system should be defined in order to capture structural and behavioral aspects of an architectural style, respectively. They also demonstrate how graph-based architecture descriptions can instantiate these style definitions and their operational semantics.

The concepts in the first two sections are illustrated using a simple architectural style. It represents a platform model for loosely coupled, component-based architectures. Loose coupling means that components can easily connect with and disconnect from each other. The instantiation of the style in architecture descriptions is demonstrated for architecture models of our online travel agency.

Section 5.3 completes the chapter with the introduction of a more complex style example, namely one for *service-oriented architectures (SOA)*. While the first style enables component coupling regardless of how involved components know each other, this assumption is abandoned for the SOA style: it allows to connect to a component only after a description of its services has been published by the provider and discovered by the requester.

Later on in Chapter 7, the two styles will serve to demonstrate our step-wise development approach for dynamic architectures. The first style will function as an abstract platform model which allows to focus on business-relevant aspects, while the SOA-specific style will function as a concrete platform model also incorporating platform-specific aspects.

## 5.1 Structural parts and their instantiation

In this section, we demonstrate how to define the vocabulary and topological constraints, i. e., the structural parts of an architectural style, using the graph schema of a graph transformation system.

The style we want to define shall capture the following platform characteristics: The platform supports distributed architectural configurations consisting of *components* and *connectors*. The internal structure of a component is hidden except for the *ports* they own as interaction points with their environment. A connector connects exactly two ports enabling bilateral communication between the corresponding components.

Besides architectural runtime configurations, the style should also provide concepts for describing the type level of an architecture. In particular, we assume that every port has a certain *port type* which determines the set of *interfaces* a component provides or requires at a port of that type. An interface, which represents a collection of *operations*, can also be assigned to several port types. The set of port types which are supported by a component is determined by an explicit *component type*. Similarly, a *connector type* determines which port types are allowed as end points of a connector. All of these type definitions can be reused by multiple instances in an architectural runtime configuration.

These are the structural assumptions about the (abstract) platform model our architectural style should reflect. The terms printed in italics already suggest potential candidates for the style vocabulary. The relationships between these elements make up the topological constraints.

Figure 5.1 shows how we formally represent the structural part of the style as a graph schema. The left part contains the elements for type specifications, whereas the right part contains the elements for runtime configurations.

While node types capture the vocabulary elements, edge types together with cardinalities capture the topological constraints and determine how node instances can be composed in an architecture description [11, 12]. For instance, a **Connector** has to connect exactly two **Ports**.

More complex constraints can also be fixed by OCL expressions; for instance, the constraint that a **Connector** may only connect **Ports** whose **Port-Types** are allowed by the corresponding **ConnectorType** reads as follows:

```
context Connector inv:
self.port.portType->forAll(pt |
    self.connectorType.portType->includes(pt))
```

An architecture instantiating the style constructs is formally represented as an instance graph over the graph schema. Figure 5.2 illustrates this for

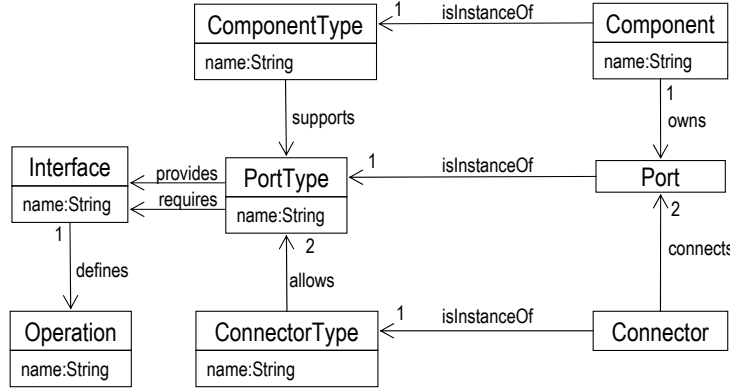


Figure 5.1: Graph schema of the component-based style

the travel agency example. The component types are taken from Fig. 3.4; we only added operations to the interfaces and connector types for potential connections between the three components.

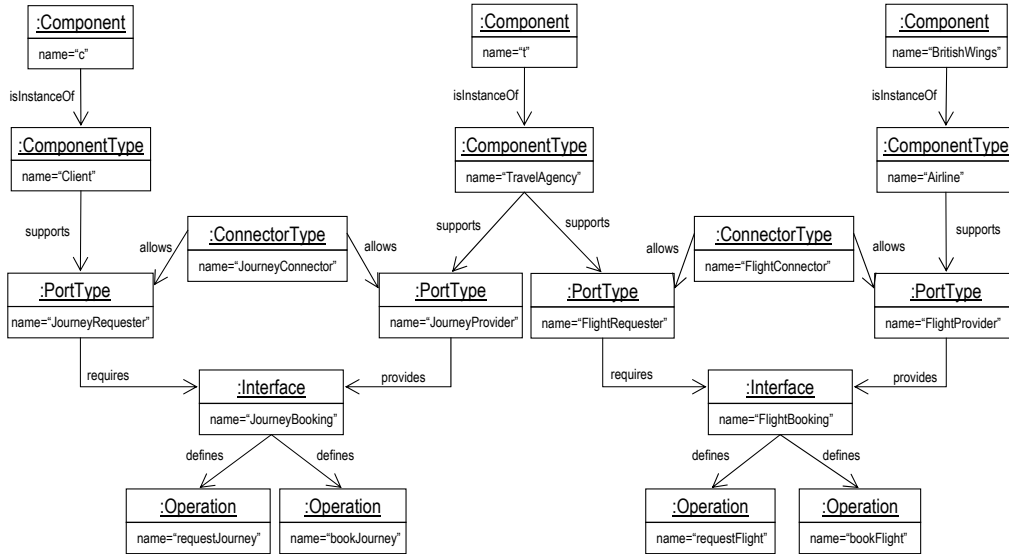


Figure 5.2: Instance graph for the travel system architecture

The example reveals how the graph-based approach allows to capture two orthogonal instantiation relationships: The structural instantiation (17) of a style by an architecture is captured by the typing morphism between graph schema and instance graph. The relationship between architectural types such as component types and their instances within an architectural

configuration (1) can be directly encoded into the instance graph provided appropriate node and edge types in the graph schema. Experts in meta-modeling will recognize the parallel to the UML meta-model [121].

The attentive reader will certainly notice that we use special **name** attributes in order to name the individual node instances. This is necessary for tracking these nodes during a runtime simulation. Remember that the instance graph represents a single runtime state only, and we later want to consider its evolution caused by communication and reconfiguration operations. However, future system states will only be explored up to isomorphism classes of their graph representations (see Definition 4.21)!

If we did not use the **name** attributes, we could not distinguish symmetric parts of the graph any more. For instance, imagine the left and right parts of Fig. 5.2 were completely symmetric except for names – as it is almost the case! Then, it would not make any difference if a connector were placed between components **c** and **t** or between **c** and **BritishWings**: Due to the symmetry, both graphs would fall into the same isomorphism class. For the analysis of the architecture, however, it does matter which components are connected and communicate with each other and which not.

For this reason, we have to provide an attribute like **name** for node types whose instances must not be matched with other instances by isomorphism. Assigning unique values to these attributes will prevent isomorphic matchings because attributes are formally defined as edges leading to special data nodes (see Definition 4.6). Note that we cannot achieve the same effect by just using distinct node identifiers, because node identifiers do not play any role in detecting graph isomorphisms (see Definition 4.1).

There are also node types which do not require such identifying attributes, especially those for transient runtime instances like **Connector** and **Port** (see Fig. 5.1). For example, connectors can be created and deleted according to the planned reconfiguration behavior (as will be explained in the following section). However, two successive occurrences of a connector between the same components do not make any functional difference and can safely be considered as isomorphic runtime states.

So far, the graph schema subsumes structural aspects of an architectural style only, i. e., the style vocabulary and topological constraints. As we will see in the next section, the consideration of behavioral aspects requires further extensions of the graph schema.

## 5.2 Behavioral parts and their instantiation

The behavioral part of an architectural style should capture platform-specific communication and reconfiguration mechanisms. In the following two Subsections 5.2.1 and 5.2.2, we show how both kinds of behavior can conceptually be treated in a uniform way using graph transformation rules.

The third Subsection 5.2.3 is important because it illustrates the instantiation of these mechanisms. We will extend the graph schema, its instance graphs, and also the graph transformation rules in order to describe how components take advantage of the platform mechanisms and how the interpretation of this component behavior can be covered by the operational graph transformation semantics.

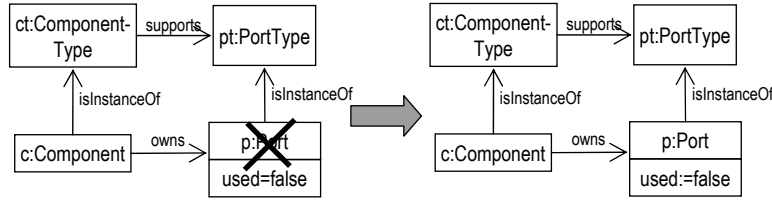
### 5.2.1 Reconfiguration mechanisms

As already pointed out in Chapter 1, an essential characteristic of a dynamic architecture is the ability to reconfigure itself. Since we represent architectural configurations as instances of a graph schema, it is obvious to express reconfigurations as transformations of these instance graphs. Due to our assumption that the underlying middleware platform determines the possible reconfiguration operations, we want to incorporate a set of predefined operations, formally defined as graph transformation rules, into the corresponding platform model.

For the platform model described by our component-based style example, we assume the following basic reconfiguration capabilities and restrictions:

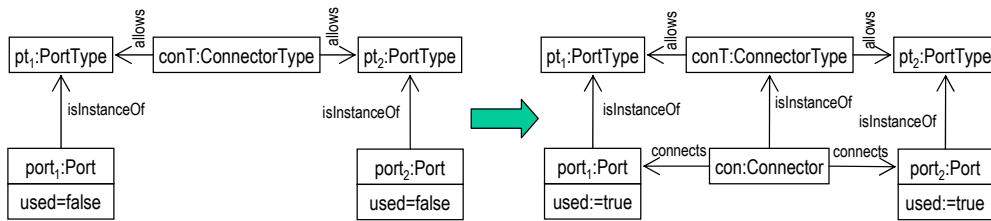
1. Components can be created as instances of a certain component type and removed again at runtime.
2. A component can open a new port, if the component type supports the port type and the component does not own any other free instance of that port type ( i. e., one that is not attached to a connector).
3. A new connector can be created and placed between two free ports, if there is a connector type which allows for the corresponding port types and if the two ports have not been used in a previous connection yet.
4. A connector can be removed again, if the connected components have finished their communication.
5. A port can be closed again, if it is not longer in use by a connector.

Each of the above reconfiguration capabilities is captured by a graph transformation rule. The rules are typed over the graph schema in Fig. 5.1. For example, consider the rule **openPort** shown in Fig. 5.3. At the left-hand side, a negative application condition ensures that component *c* has no other free port of the supported port type *pt* when the rule is applied. According to the rule's right-hand side, its application will result in the creation of such a **Port** node.

Figure 5.3: Reconfiguration rule **openPort**

The **used** attribute in nodes of type **Port** requires a minor extension of the graph schema. The attribute is necessary in order to distinguish ports which have already been used for a connection from newly created ones. Accordingly, the attribute is initially set to **false** in the **openPort** rule.

Later, when two ports are connected by a connector, their **used** attributes are changed to **true**. This is carried out by the reconfiguration rule **connect** shown in Fig. 5.4. It creates a new connector, provided that there are two components owning free ports (**used=false**) which are allowed by the connector's type.

Figure 5.4: Reconfiguration rule **connect**

As a direct consequence of Definition 4.14, transformation rules are also applicable to nodes which have more attributes than those mentioned in the rule definitions. For this reason, we can simplify the rule definitions and safely omit all attributes which are not relevant for the considered transformation; e. g., the many **name** attributes.

The remaining reconfiguration rules – **disconnect** for removing a connector, **closePort** for closing a port, **createComponent** for creating a new component instance, and **removeComponent** for deleting such instance again – have partially been published in [14] and can also be found in Appendix A. Although we have deliberately chosen simple and self-explanatory examples, the expressiveness of graph transformation rules allows us to express all kinds of reconfiguration operations.

By applying the reconfiguration rules to a given instance graph, we can simulate an architecture’s runtime evolution. Figure 5.5, for example, shows how the above introduced reconfiguration mechanisms could be applied to the configuration of Fig. 5.2 in order to place a connection between the travel agency and the client component.

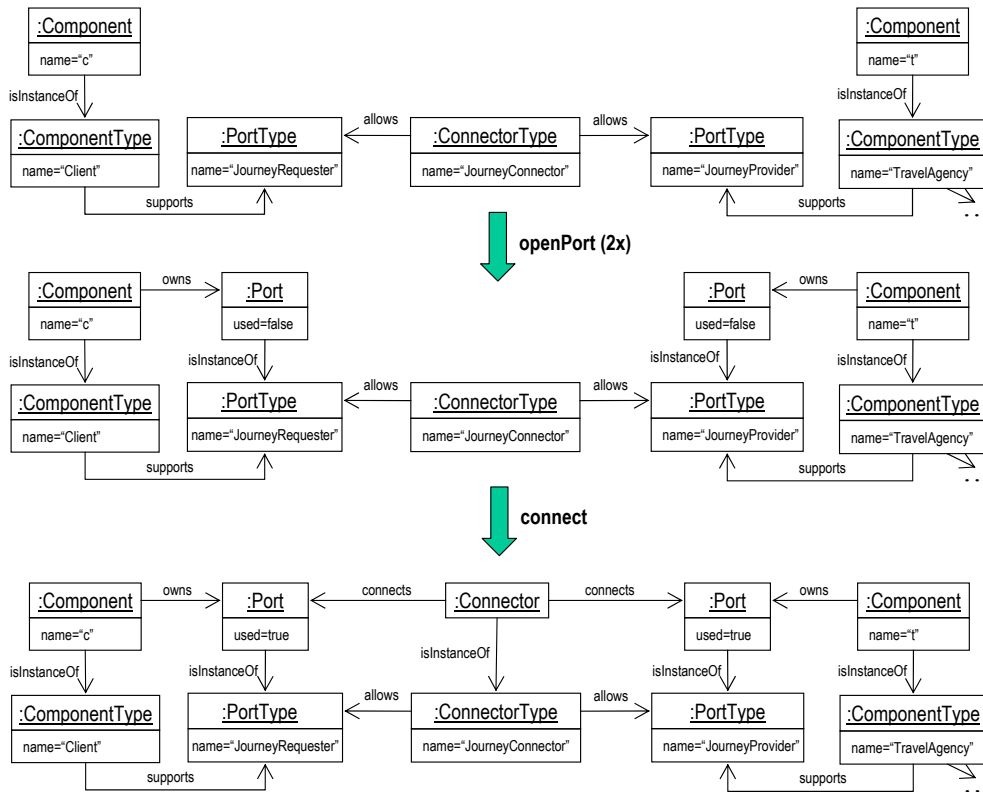


Figure 5.5: Reconfiguration scenario as transformation sequence

### 5.2.2 Communication mechanisms

Equally important as dynamic reconfigurations is the ability of components to communicate and interact with each other in an architectural configuration. Again, we believe that the chosen target platform determines the available communication mechanisms. For our component platform example, the platform model should capture the following assumptions:

1. Only bilateral communication is supported between two components using a connector.
2. A component can send a request message to a connected component in order to call an operation provided by the other component.
3. A component can receive requests for operations it provides.
4. Afterwards, the receiver can send an appropriate response message back to the sender.

Aiming at a uniform formalism, we would like to describe such platform-specific communication mechanisms by graph transformation rules, too. However, in order to do so, we somehow have to encode the state of a communication into our instance graphs. The solution we propose is to insert additional nodes and edges which represent communication-related concepts like messages, information about sender and receiver, and so forth.

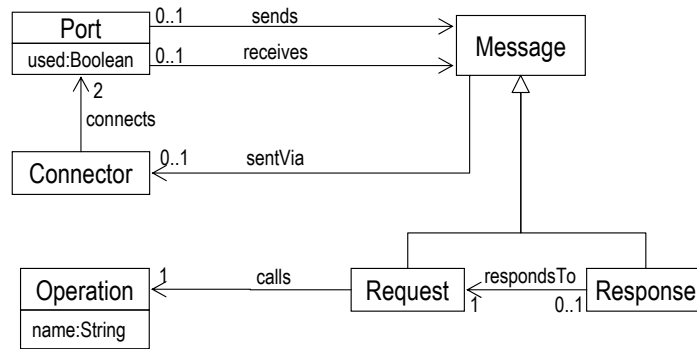


Figure 5.6: Graph schema extensions for component communication

The necessary extensions of the graph schema of the component style are shown in Fig. 5.6. The left side repeats the relevant elements of the existing graph schema (Fig. 5.1), whereas the right side introduces new node types. **Request** and **Response** nodes represent the two different kinds of messages.



A **Request** node may have an edge referring to the called operation, and a **Response** node is linked to the preceding **Request**. The common properties of the two message types are generalized into the abstract supertype **Message**: A **Message** is sent and received by a port and transmitted via a connector.

With the help of the new node and edge types, we can include information about the current communication state in instance graphs. Analogously, we can formulate graph transformation rules over these elements in order to model platform-specific communication mechanisms.

For example, the transformation rule **callOperation** in Fig. 5.7 models how a **Request** message is sent from a sender port (**from**) to a receiver port (**to**). Note that the rule can be defined without mentioning the components owning these ports; the underlying graph schema guarantees that such components will always exist in a valid instance graph. However, it is necessary to prescribe that the two ports are connected by a connector and the operation to be called belongs to an interface provided by the target port.

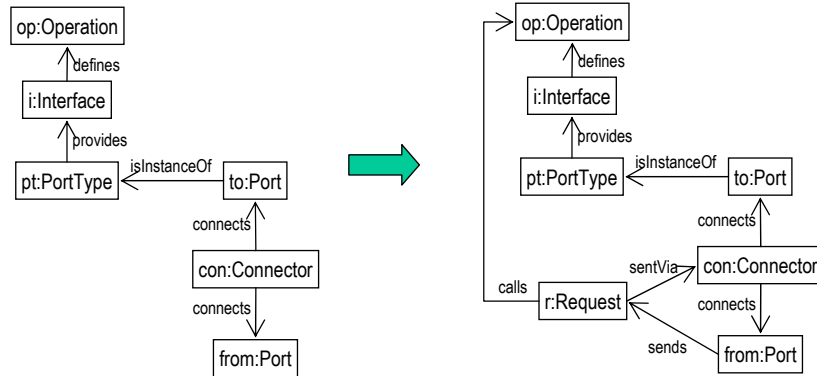
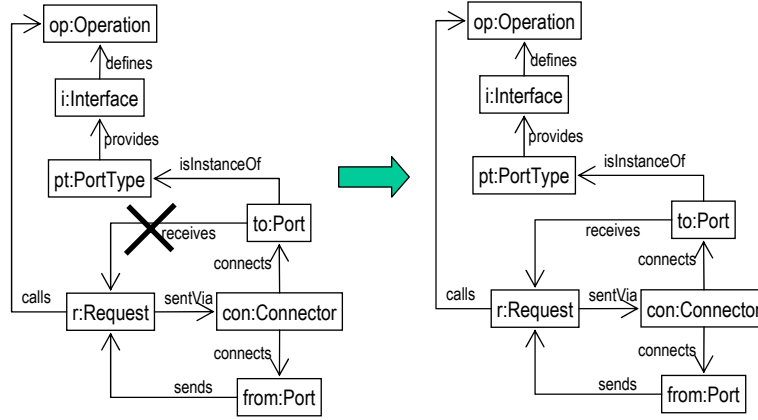


Figure 5.7: Communication rule **callOperation**

Figure 5.8 shows another example of a communication rule. It can be applied after sending a **Request** and represents the receipt of the message by inserting a **receives** edge. The negative application condition on the left-hand side ensures that the rule is not applied when such an edge already exists.

There are similar communication rules for sending and receiving **Response** messages [14]. Together with the reconfiguration rules from Section 5.2.1, the communication rules can be applied to instances of the architectural style. The resulting transformation sequences represent the possible runtime behavior of the architectural configuration.

Figure 5.8: Communication rule `receiveCall`

### 5.2.3 Instantiating platform mechanisms in architectural behavior

After having modeled the reconfiguration and communication mechanisms of a platform as graph transformation rules, the question remains how to describe architectural behavior that takes advantage of these mechanisms (18).

So far, the transformation rules for communication and reconfiguration mechanism can be applied whenever possible. For every transformation step, there are basically two degrees of freedom: one can at first choose from the applicable rules and then from the valid occurrences of the rule's left-hand side in the graph. The result of a transformation step depends on these choices which are completely non-deterministic so far.

This kind of rule application is only suitable for representing *ad-hoc* reconfigurations which are not anticipated but triggered in certain situations or by the occurrence of certain events. For this purpose, we only need to encode the events as additional preconditions into the rules' left-hand sides.

However, we also require a possibility to model *programmed* reconfigurations (5) and to prescribe a certain order of component communication (2). For example, we want to prescribe in which moment a rule such as `callOperation` shall actually be applied and which operation it shall actually call. For modeling this kind of behavior, we need to restrict the non-determinism in graph transformation systems and to control the way our rules are applied.

This can be achieved in different ways. Using priorities is not appropriate because they determine preferences between conflicting rules but cannot capture component-specific orders in which rules have to be applied. Imperative control structures like sequential composition, iteration, conditional branch-

ing, and so forth are better means. This leads to the definition of *programmed graph replacement systems* [139] two important implementations of which are PAGG (**P**rogrammed **A**ttributed **G**raph **G**rammars [54]) and PROGRES (**P**ROgrammed **G**raph **R**Eplacement **S**ystems [141], see Section 8.1.2).

However, introducing control structures on top of transformation rules requires additional means to define the operational semantics, and it also destroys the locality of actions that arises from the rule-based character of graph transformation systems. If we leave the standard graph transformation theory, we lose the uniformness of the employed formal method (10) and encounter new difficulties in deriving graph transition systems as runtime models (see Section 4.3.2).

For this reason, we propose an alternative way to restrict the non-determinism, namely by inserting additional application conditions into the transformation rules. In the following paragraphs, we show how to use positive and negative application conditions together with additional state information stored in the graphs in order to mimic explicit control structures as defined in programmed graph replacement systems. This way, we obtain meaningful models within the standard transformation semantics and without losing the locality property of transformation rules.

Based on a previous paper [63], we explain how to formally describe a component's communication and reconfiguration behavior in terms of the available platform mechanisms. *Control flows* prescribe the order in which platform mechanisms are invoked, including concurrent and branching behavior, and *action parameters* define their application context. We incorporate these aspects into the instance graphs and augment the transformation rules so that their execution semantics respects the behavior descriptions. Eventually, we show how a *synchronization* of different control flows can be realized.

**Control flow.** To insert control flow definitions into our architecture models, we equip the instance graphs with *processes* consisting of a linked chain of action nodes. Each action corresponds to the invocation of a certain platform mechanism. Processes are assigned to component types, and every component instance runs its own process instance, called *thread*. Every such thread maintains a pointer to the previously performed action and increments this *action counter* after completion of a subsequent action (see Fig. 5.9).

Obviously, we have to extend the graph schema of the architectural style in order to include such process definitions in our instance graphs. For this purpose, we supplement the underlying type graph  $TG$  by new node types  $TG_N^{beh}$  and new edge types  $TG_E^{beh}$  whose instances can be used to describe

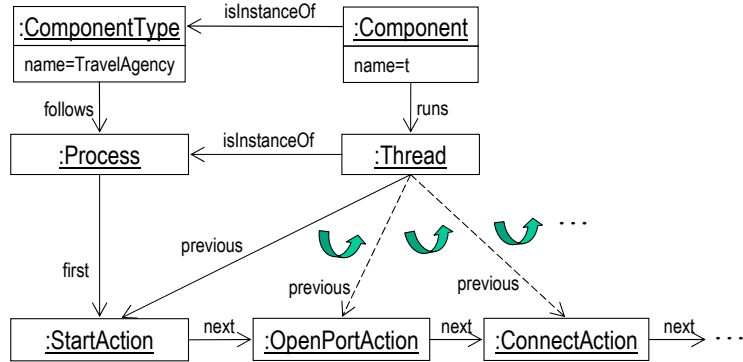


Figure 5.9: Control flow realized by processes and action counter

component behavior.

Figure 5.10 shows the necessary extensions for the graph schema of the component-based style. The central member of  $TG_N^{beh}$  is the new node type **Action**. Its instances can be linked with each other by **next** edges in order to form a control flow. If we attach more than one outgoing **next** edge to an action node, then the control flow splits after this action into two or more *alternative branches*. As there are no branching conditions, it is decided non-deterministically which branch a thread follows.

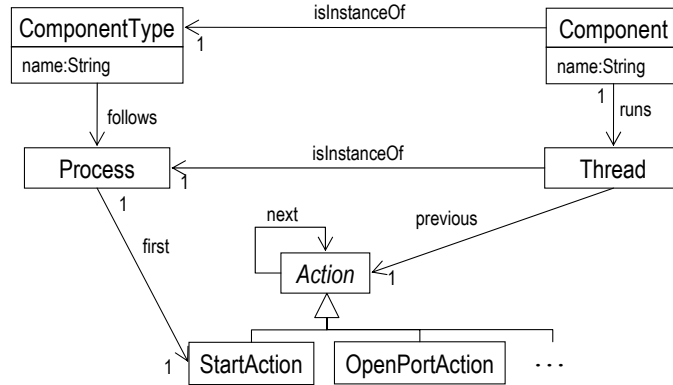


Figure 5.10: Graph schema extensions concerning control flow

**Action** is an abstract node type whose subtypes represent the available platform mechanisms. Only the special subtype **StartAction** does not refer to a particular operation but indicates the entry point of the **Process**.

The action counter is realized by a **previous** edge outgoing from every

**Thread** node. As the edge type leads to the abstract node type *Action*, its instances can in fact iterate over all action nodes of a process.

As the cardinalities in Fig. 5.10 reveal, component types may follow several processes, and every component instance may run several threads thereof. This way, we achieve a multitude of possibilities for *concurrent* behavior as it is required for distributed architectures.

But how can we ensure that our communication and reconfiguration rules are actually applied according to the specified control flow? To solve this problem, the graph transformation rules have to be adapted, too. On their left-hand sides, we insert additional application conditions over instances of the new types  $TG_N^{beh}$  and  $TG_E^{beh}$ . They have to ensure that an active element is running a thread whose next action requires the application of this rule. On the right-hand side, we extend the original rule effect by an incrementation of the action counter.

Figure 5.11 illustrates these modifications for the known reconfiguration rule **openPort**. In contrast to the previous version in Fig. 5.3, its left-hand side requires that the **Component** runs a **Thread** whose previously performed action is succeeded by an **OpenPortAction**. In other words, the component has to be in a state in which it wants – or at least agrees – to invoke the **openPort** reconfiguration mechanism. The right-hand side shows how such a rule – in addition to its original effect – increments the action counter.

In order to improve the readability of such control flow-enabled rules, a gray background delimits their original parts from the control flow-related extensions over  $TG_N^{beh}$  and  $TG_E^{beh}$ .

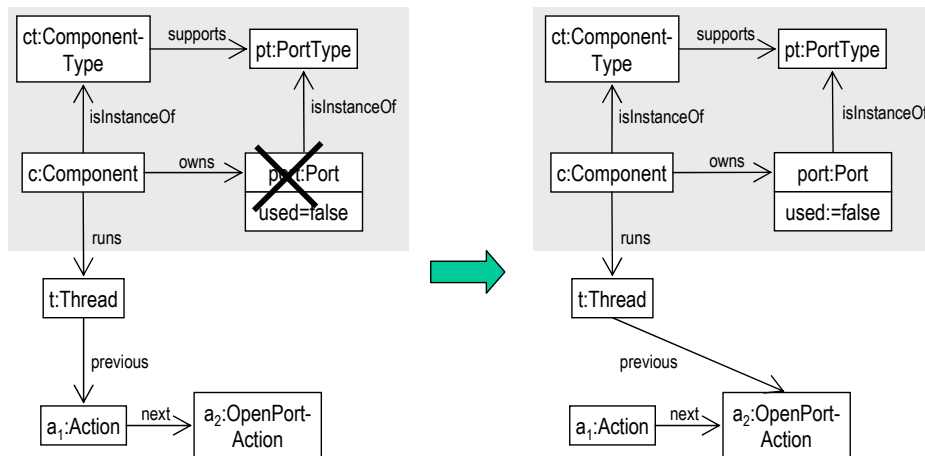


Figure 5.11: Control flow-enabled reconfiguration rule **openPort**

If all other rules of the architectural style are adapted accordingly, we do not need any further construct for interpreting the specified control flows; the execution semantics of graph transformation rules takes care of the right rule selection. And, if there are several concurrent threads in an architecture, the non-deterministic choice from the set of applicable rules ensures interleaving progress of the individual threads.

By incorporating both communication and reconfiguration actions into the same control flow, we satisfy the two requirements *programmed reconfiguration* (5) and *reconciliation of communication and reconfiguration behavior* (6): The engineer can program in which state a certain reconfiguration shall be performed, and he can decide how the reconfiguration operations are safely combined with the remaining communication behavior. As reviewed in Table 2.1, other graph transformation-based ADLs do not support programmed reconfigurations so far.

**Parametrization of actions.** Besides their order, we also need to specify the individual *application context* of actions. This includes, for instance, the operation which shall be addressed by a **CallOperationAction** or the type of port which shall be created by an **OpenPortAction**. Such additional information is usually assigned to an action using *parameters*. We distinguish two different kinds of parameters:

1. **Constant parameters:** The value of a constant parameter is shared by all instances of a process and not changed at runtime.
2. **Variable parameters:** Variable parameters refer to variables which are declared once for a process but have different values per thread. Moreover, their value can be changed at runtime.

We express constant parameters as special edges in an instance graph pointing from the concerned action nodes to those nodes which function as parameter values. Variable parameters are more complex because their value might be different for each thread instantiating the process. Hence, a variable parameter edge cannot directly point to a fixed value node any more. Instead, we introduce the new node type **Variable** and express variable parameters by an edge pointing from the concerned action node to a **Variable** node. Moreover, in order to represent the ternary relationship between variable, its value, and the owning thread, we also have to introduce so-called **Reference** nodes which own edges to the involved **Variable**, **Thread**, and value nodes.

The new node and edge types are added to the behavior-related type graph parts  $TG_N^{beh}$  and  $TG_E^{beh}$ , respectively. The resulting extension of the

style's graph schema is summarized in Fig. 5.12. As the types of the value nodes vary from variable to variable, we introduce the abstract type **RefElement** as target node for the **refersTo** edge type. It generalizes all node types whose instances might be referenced as variable values.

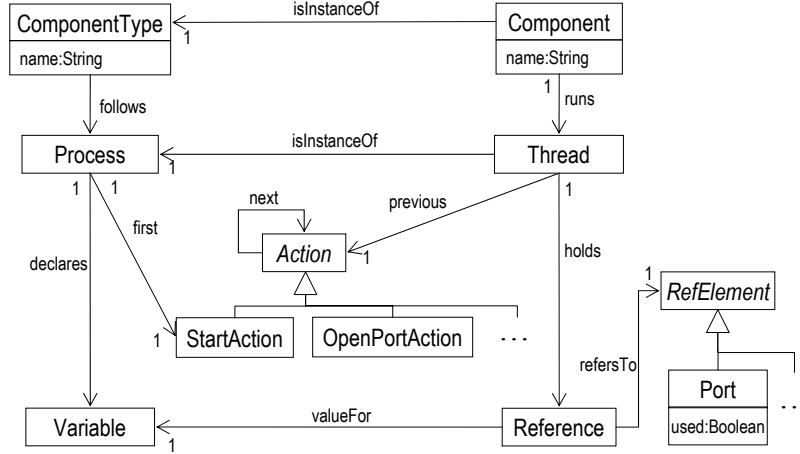
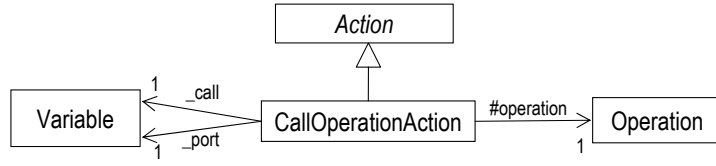


Figure 5.12: Graph schema extensions concerning variable action parameters

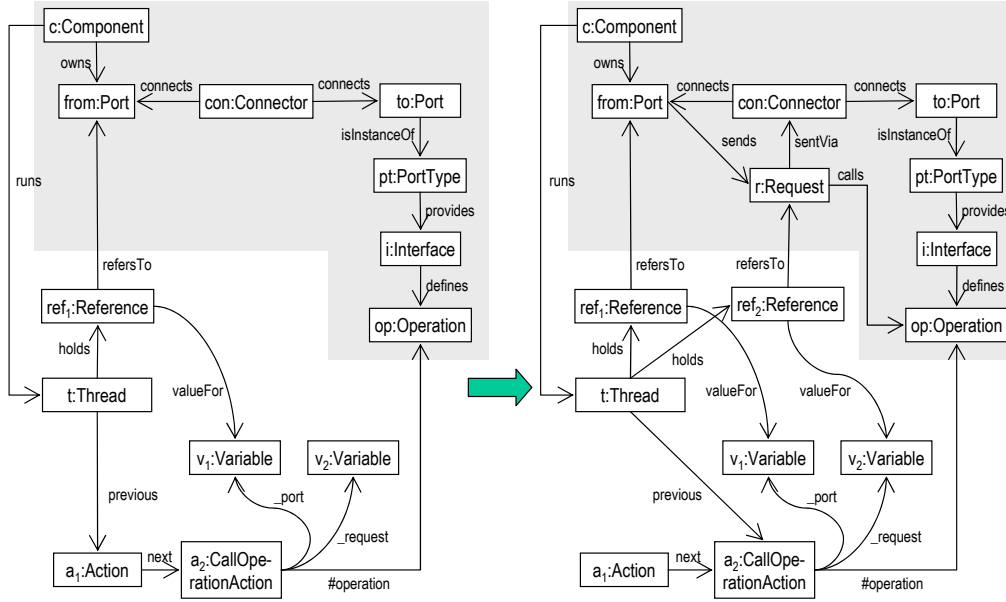
Of course, the parameter edges outgoing from action nodes have to be defined in the graph schema, too. In the case of constant parameters, the target node of the edge type determines the type of nodes which can be used as values of the parameter in an instance graph. By convention, we label such constant parameter edges with a leading hash (#).

In contrast to constant parameters, variable parameter edges are labeled with a leading underscore. As their value can be changed, they fulfill two different purposes: On the one hand, they can be used as *output* parameters if the underlying action creates a new reference as value for the variable. On the other hand, they can be used as *input* parameters if the underlying action reads the previously assigned value from the variable.

Taking the **CallOperationAction** example, Fig. 5.13 illustrates how to define the parameter edges for a certain action node type. The constant parameter **#operation** determines which operation shall be called. The variable parameter **\_port** determines the port which shall issue the call. It is a variable parameter because each thread is run by a separate component and, consequently, applies this action to a different **Port** node. The second variable parameter **\_call** is an output parameter. It can be used to fill another variable with the **Request** object created during the **CallOperationAction**. From then on, parameters of subsequent actions can access the newly created **Request** object if they refer to the same variable.

Figure 5.13: Parameter definitions for the `CallOperationAction`

Needless to say, the incorporation of action parameters requires adaptations of the corresponding transformation rules, too. Continuing the above example, Fig. 5.14 shows how the three parameters affect the transformation rule `callOperation` (cf. Fig. 5.7).

Figure 5.14: Parameter-enabled transformation rule `callOperation`

The constant parameter `#operation` and the input parameter `_port` represent additional application conditions in the rule’s left-hand side: The rule can now only be applied to that `Operation` and `Port` node, respectively, which the current parameter values refer to.

The output parameter `_call` is reflected in an additional effect of the rule’s right-hand side: The rule does not only create a new `Request` node but also a new `Reference` node. The thread holds this `Reference` in order to “remember” the assignment of the just created `Request` node to the second variable.



**Synchronization.** Due to the locality of rule applications, the default behavior modeled by graph transformations is asynchronous. However, according to Req. (7), we also want to allow concurrent components to synchronize their threads in order to participate in a common communication or reconfiguration operation. For instance, we assume that the creation or removal of a connector between two components requires “agreement” from both parties. This agreement can be expressed by a synchronized invocation of the corresponding **connect** or **disconnect** operations.

We realize such synchronizations by further extensions of the relevant transformation rules: The control flow-relevant part of a rule is duplicated for each component participating in the synchronization. Figure 5.15 illustrates this for reconfiguration rule **connect** from Fig. 5.4.

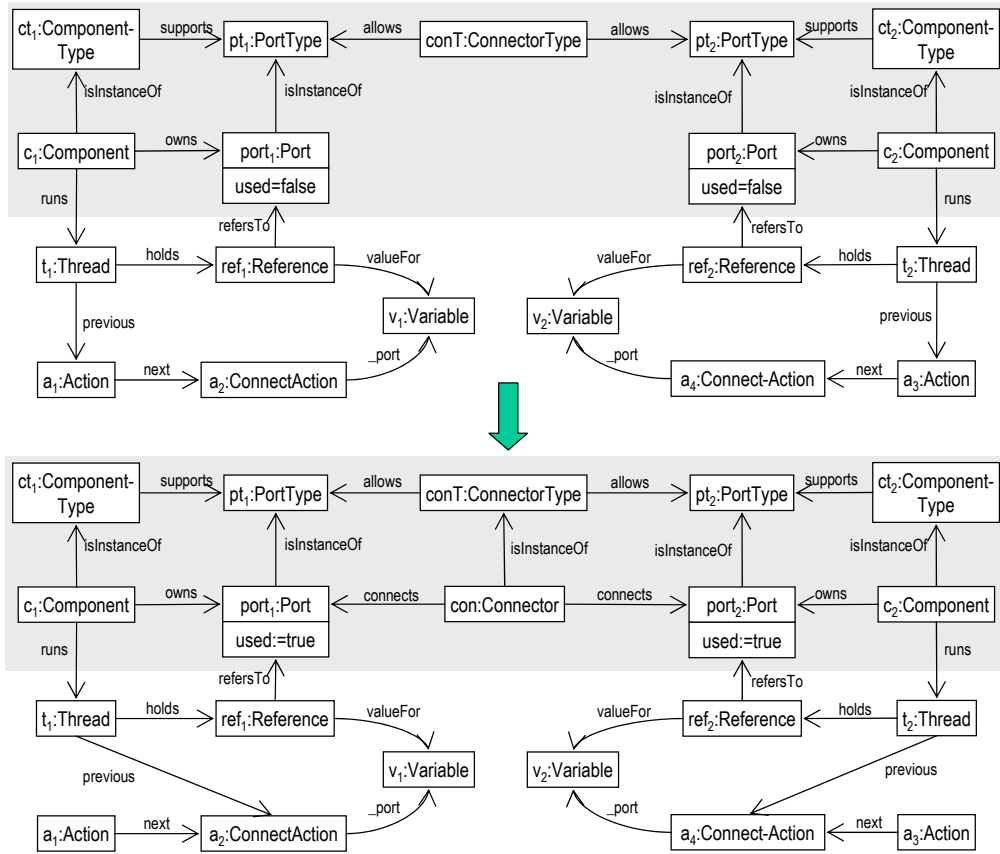


Figure 5.15: Rule **connect** enforcing thread synchronization

The rule’s left-hand side (at the top) explicitly demands two threads which both invoke the **connect** operation for a certain **Port**. Hence, if we want a

component to connect to another component, we must insert a **ConnectAction** into the process definitions of both component types. At execution time, the rule application is deferred until both corresponding threads reach the **ConnectAction**. In this sense, the action functions as a *synchronization point*.

The rule's right-hand side (at the bottom) shows that the action counters of both threads are incremented after the operation. This way, the operation leads to a synchronized progress of both threads.

**Thread management.** The introduction of processes and threads being instances thereof requires the introduction of dedicated operations for thread management. By thread management we mean the initiation of new threads and the garbage collection of finished threads.

A thread can trigger the initiation of a new thread if it contains a special action of type **NewThreadAction**. This action has a single parameter **#process**. Its value determines the process to be started: the new thread can either become an instance of the current process or of any other process defined for the component type. Figure 5.16 shows the corresponding transformation rule. The action counter of the new Thread is initially set to the **StartAction** node of the selected process.

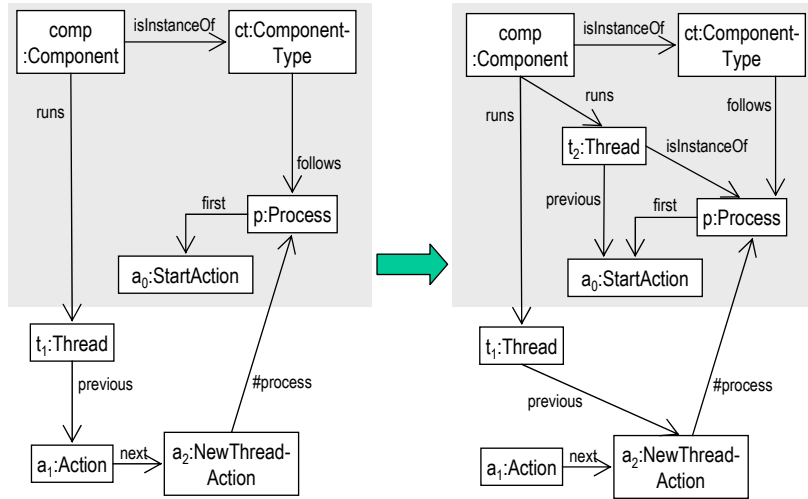


Figure 5.16: Thread management rule **newThread**

The garbage collection of finished threads is done by two other thread management rules. In contrast to the **newThread** rule, they do not require an invocation by a special action node but can be applied in the background without any progress of a thread. The only precondition is that there is a

**Thread** whose previous action has no more successor. Then, the thread is finished, and the corresponding nodes are due to clearance.

Given this precondition, rule **clearReference** (Fig. 5.17) is applied – possibly several times – in order to remove the **Reference** nodes of the finished thread. Afterwards, rule **clearThread** (Fig. 5.18) can be applied in order to remove the **Thread** node itself. Due to the dangling condition of Definition 4.14, the rule is not applied unless all **Reference** nodes have been cleared before.

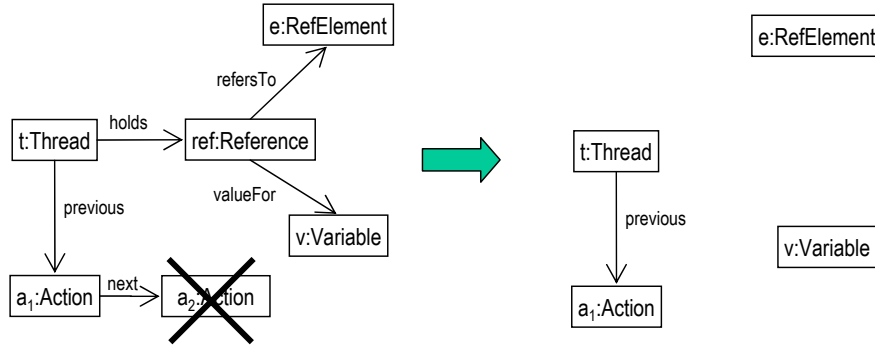


Figure 5.17: Thread management rule **clearReference**

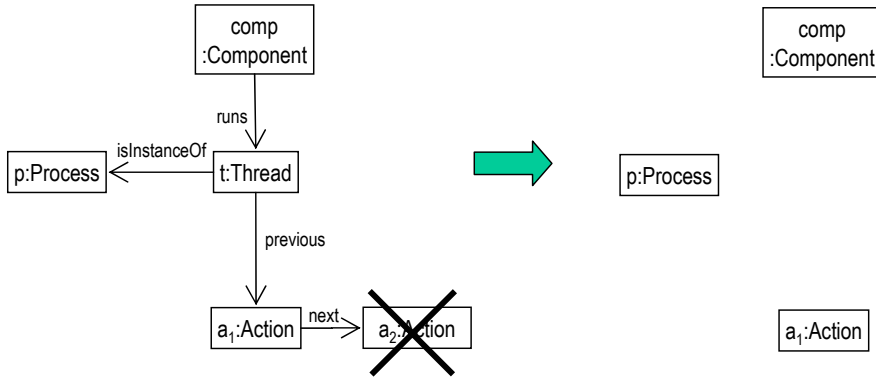


Figure 5.18: Thread management rule **clearThread**

**Structure projection.** Sometimes, it is desirable to consider only those elements of an instance graph which describe the *structure* of an architecture. For this purpose, we define the projection function  $struct_{TG}$  which removes all behavior-related elements typed over  $TG_N^{beh}$  or  $TG_E^{beh}$  [63].

**Definition 5.1 (Structure Projection).** Let  $\langle G, type_G \rangle$  be an instance graph over the type graph of an architectural style with behavior-related elements  $TG_N^{beh}$  and  $TG_E^{beh}$ . Then, the structure projection  $struct_{TG}(\langle G, type_G \rangle)$  is a subgraph  $\langle \tilde{G}, type_{\tilde{G}} \rangle$  with

- nodes  $\tilde{G}_N = \{n \in G_N \mid type_G(n) \notin TG_N^{beh}\}$
- edges  $\tilde{G}_E = \{e \in G_E \mid type_G(e) \notin TG_E^{beh}\}$ , and
- typing  $type_{\tilde{G}} = type_G|_{\tilde{G}}$

As isomorphic graphs have equal typing,  $struct_G$  is closed under isomorphism.

For the component-based style, the structure projection removes all elements whose types have been introduced in this section (collected into the packages **Processes** and **Actions** in Appendix A). In Section 7.2, we will apply the structure projection when defining a criterion for structural refinement between two architecture models.

### 5.3 A style for service-oriented architectures

After the concepts for graph transformation-based architectural styles have been illustrated using a simple component-based style, this section presents a more complex style example, namely one for *service-oriented architectures* (SOA). Previous versions have been published in [11, 12, 14, 63].

In a service-oriented architecture, business components expose their functionality as *services* over a network to other components. A service is equipped with a description of the provided functionality including information about the interface and how to access it. Recently, the service-oriented paradigm has become very popular under the label of *Web Services* [23, 124].

As shown in Fig. 5.19, SOA involves three different roles: *service providers*, *service requesters* and *discovery agencies*. The service provider runs the service and *publishes* the service description in order to enable dynamic service discovery and to allow requesters to access the service.

Since providers and requesters usually do not know each other in advance, the service descriptions are published via third-party discovery agencies. They categorize the descriptions and deliver them in response to *queries* issued by service requesters. As soon as the service requester retrieves a service description that meets its requirements, it can use it to *interact* with the service [126].

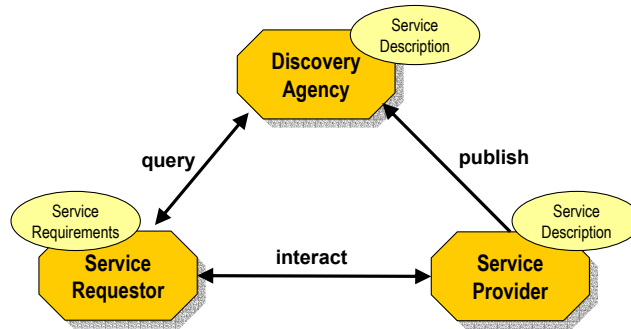


Figure 5.19: Roles in a service-oriented architecture, cf. [23]

Service-oriented architectures are very dynamic and flexible: Components and services are loosely coupled and use standardized communication protocols. As the service descriptions are exchanged at runtime, service requesters can dynamically switch from unsatisfactory services to those providing better functionality or quality. Dynamic service discovery might also be required in self-healing systems, e. g., when a service is not reachable any longer due to network problems. Similarly, our online travel agency could take advantage of this SOA feature when looking for business partners that meet the client's traveling requirements.

To consider SOA-specific features like service discovery in our architecture models, we are going to define an architectural style which formally describes the concepts of service-oriented computing. Following the idea of a platform hierarchy (cf. Section 1.3.2), the SOA-specific style extends the more generic, component-based style. This means that service-oriented architectures contain components and connectors, too, and they support the same message-based communication mechanisms.

Only the reconfiguration mechanisms are partially different: While we have the same loose coupling with ports being opened, closed, connected, and disconnected, the retrieval of a service description now becomes mandatory before connecting to a service. For this purpose, the architectural style has to incorporate the following additional platform mechanisms:

1. A service provider can publish service descriptions to third-party discovery agencies.
2. A discovery agency receives publication requests and stores the attached service descriptions.
3. A component requiring a certain service can send a service query to

the discovery service.

4. The discovery agency can receive such queries, search for an appropriate service description, and send the query result back to the requester.
5. The service requester receives and saves the description before it connects to the service.

Before we start to describe these mechanisms as graph transformation rules, we have to adapt the underlying graph schema. Services and discovery agencies can be modeled as special components using appropriate subtypes of **Component** and **ComponentType**. To distinguish the ports involved in service publication and discovery, we introduce new subtypes of **PortType**: instances of **ProviderPT** and **RequesterPT** belong to components providing and requesting a service, respectively, and instances of **PublishPT** and **FindPT** belong to discovery services. The extended graph schema is shown in Fig. 5.20.

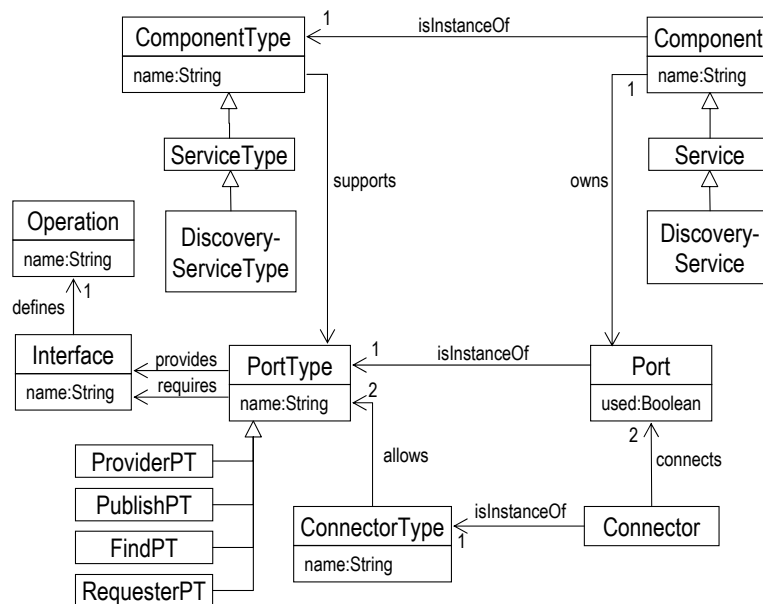


Figure 5.20: Graph schema of the service-oriented style

Another extension of the graph schema, shown in Fig. 5.21, concerns service descriptions and various new message types. Besides the already known **Request** and **Response** messages, there are three special SOA message types for interactions with discovery services, namely **ServicePublication**, **ServiceQuery**, and **QueryResult**. The first one submits a service description to a discovery service for publication, the second one refers to a port type which the service

requester requires a suitable service for, and the third one is returned by the discovery service containing a description that satisfies the query.

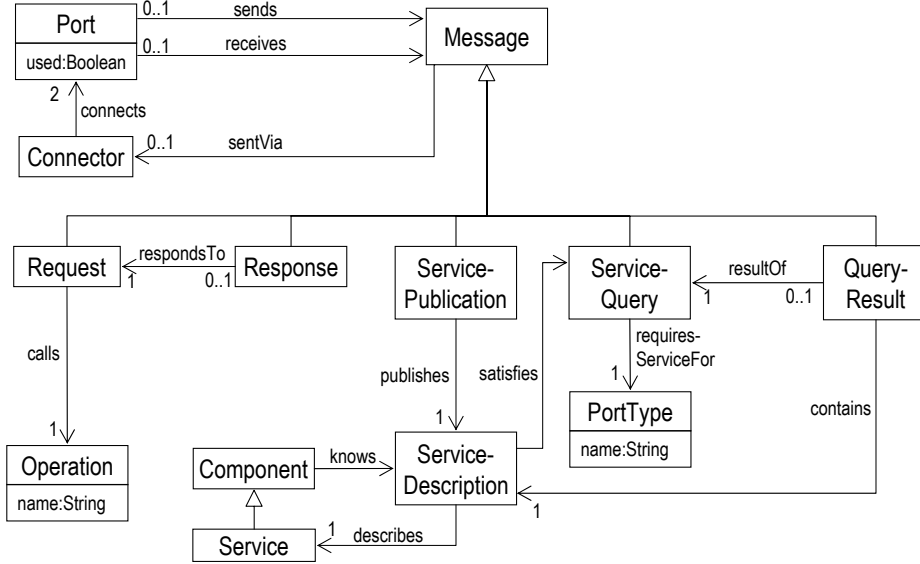
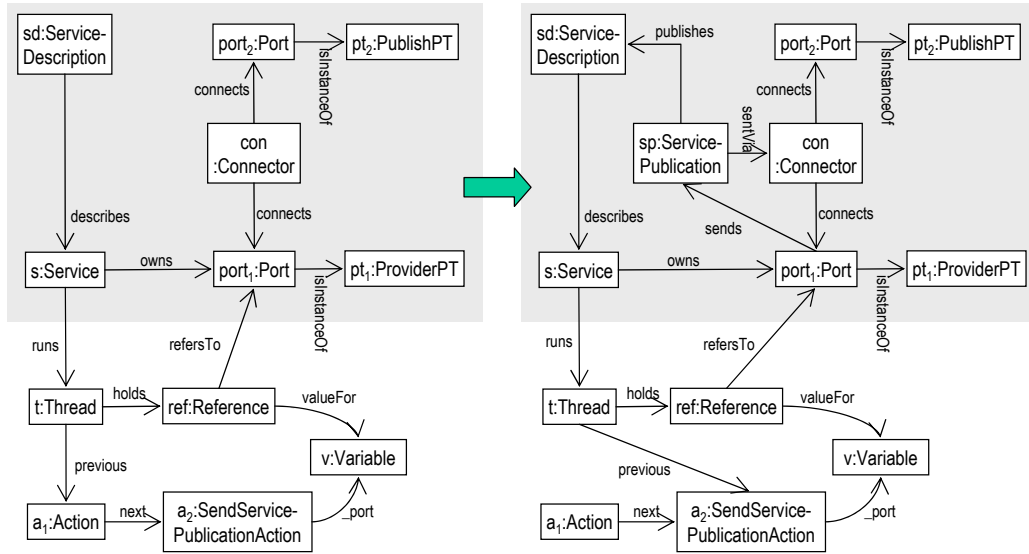


Figure 5.21: Graph schema part for communication in SOA

A **ServiceDescription** describes a specific **Service** because, in SOA, descriptions refer to deployed, addressable services rather than to service types. The **knows** relationship indicates which components have access to a description. The existence of such a **knows** relationship becomes a new precondition for connecting to a service. The accordingly extended reconfiguration rule **connect** can be found in Appendix B.

Except for the **connect** rule, the service-oriented style inherits all reconfiguration and communication rules from the component-based style. Only the SOA-specific reconfiguration mechanisms have to be modeled by new transformation rules. The complete rule specifications can be found in Appendix B. At this point, we confine ourselves to some examples.

For instance, the rule **sendServicePublication** in Fig. 5.22 shows how a service description is made public to a discovery agency. For this purpose, the **Service** requires a port of kind **ProviderPT**, which is determined by the input parameter `_port`. Through this port, it sends a **ServicePublication** message which refers to the **ServiceDescription** node. Although the rule does not prescribe a **DiscoveryService** node as addressee of the message, this is indirectly ensured by requiring a **PublishPT** port as opposite endpoint of the connector.

Figure 5.22: SOA-specific reconfiguration rule `sendServicePublication`

The receipt of such a `ServicePublication` message is modeled by rule `receiveServicePublication`. Rule `publishServiceDescription` models the storing of the description by a new `knows` link from the `DiscoveryService` (see Appendix B).

The second half of SOA-specific rules deals with the retrieval of a service description from a discovery service. The initiation and receipt of a query is handled by rule `sendServiceQuery` and `receiveServiceQuery`, respectively.

In order to answer the query, the discovery service has to select an *appropriate* service description which satisfies the requester's requirements. As the underlying algorithms are not relevant for the architectural view, we model this selection by a simple check whether the service supports a port type which can be connected to the port type the service requester requires a service for. The corresponding transformation rule is shown in Fig. 5.23.

After a satisfactory service description has been selected, the response from the discovery service is handled by rule `sendQueryResult` and `receiveQueryResult`, respectively. Eventually, rule `saveQueryResult` finishes the query communication and saves the result by creating a new `knows` edge as shown in Fig. 5.24.





## 5.4 Summary

In this chapter, we have seen how platform-specific reconfiguration as well as communication mechanisms can be described using graph transformation rules. Together with the structural parts, the resulting graph transformation systems provide platform-specific architectural styles for component-based and service-oriented architectures, respectively.

Moreover, we have shown how to extend the graph transformation systems in order to achieve meaningful behavior models including control flow definitions and action parametrization. The complete specifications of all rules and their underlying graph schemas can be found in Appendix A and B.

Since we cover everything with standard graph transformation theory and do not require additional control semantics as in *programmed* graph transformation systems, we are able to retain the uniformness of the employed formal method (10).

In the next chapter, we will show how to apply the formal architectural styles – in conjunction with appropriate UML profiles for the concrete syntax – in order to create platform-consistent architecture models.

## Chapter 6

# Style-based modeling of dynamic software architectures

According to the previous chapters, we model dynamic architectures using platform-specific architectural styles. These models are formally expressed by instance graphs over the corresponding graph transformation system.

Graph transformations are a powerful formalism. However, as already discussed in Chapter 3, instance graphs can easily grow very large and become difficult to overlook. Besides, they neither provide any visual differentiation between node types nor any modularization concepts. Hence, in continuation of previous work [61, 14], we propose to describe architecture models using UML as front-end modeling language.

Unfortunately, UML does not provide any first-class construct for defining and incorporating architectural styles. However, as architectural styles restrict the set of legal architecture models concerning vocabulary, constraints, and allowed operations, they can be considered part of the language definition rather than part of the model. Accordingly, we propose to define architectural styles at the UML meta-level, namely by *UML profiles*.

Section 6.1 illustrates how style architects can define UML profiles for platform-specific architectural styles. As the design of UML profiles is a topic of its own and beyond the scope of this thesis, we do not provide a general methodology here but discuss the main ideas along an exemplary UML profile for the service-oriented architectural style. The resulting profile is applied to some diagrams modeling a SOA-specific variant of the travel system architecture. The complete UML model of the architecture can be found in Appendix D.

However, the use of UML does not mean that we throw the graph-based representation overboard. In Section 6.2, we will rather define a relation between the extended UML meta-model and the graph schema which enables

automated translations of UML models into their graph-based representations and back again. This way, the architect is still able to apply analysis and refinement techniques that take advantage of the formal semantics of the graph transformation-based style definitions.

## 6.1 Profile for service-oriented architectures

The principal constituents of a UML profile are *stereotypes*. In Chapter 3, we have already used a number of style-specific stereotypes when modeling the online travel agency with UML. The intention behind is to transfer our style-based approach to the UML world by exploiting UML's built-in extension mechanisms and defining a *UML profile* for each architectural style. In this section, we will illustrate the definition of such a profile for the SOA-specific style introduced in the previous chapter.

According to the UML specification [121], a stereotype extends and adapts classes of the UML meta-model by refining its semantics, adding new attributes, constraints, and, optionally, a new distinguished notation. When defining a profile for a platform-specific style, the style architect has to look through the style vocabulary and check which elements are not covered by the standard UML. For every such element, he chooses a semantically close UML meta-class and derives a new stereotype in order to include the style-specific element in future UML architecture models [133, 106].

### 6.1.1 Stereotypes concerning structure

When carrying out this procedure for our service-oriented style, we start with the structure-related parts of the graph schema in Section 5.3. Those elements inherited from the generic, component-based style can mostly be modeled with ordinary UML constructs. For example, UML provides two meta-classes **Component** and **InstanceSpecification** which can be used to represent component types and components, respectively.

Nevertheless, we have to define special stereotypes for SOA-specific style elements. Figure 6.1 shows how they are formally derived from existing UML meta-classes, which are depicted in the upper part of the diagram. The extension relationships are denoted as arrows with small, filled arrowheads. Some stereotypes are specializations of others, indicated as subtypes.

We introduce the stereotype **PortType** as extension of the UML meta-class **Class** because port types can be modeled with class diagrams as shown in Fig. 6.2. Its sub-stereotypes are used in connection with ports for discovery services – analogously to the graph schema in Fig. 5.20. Associations

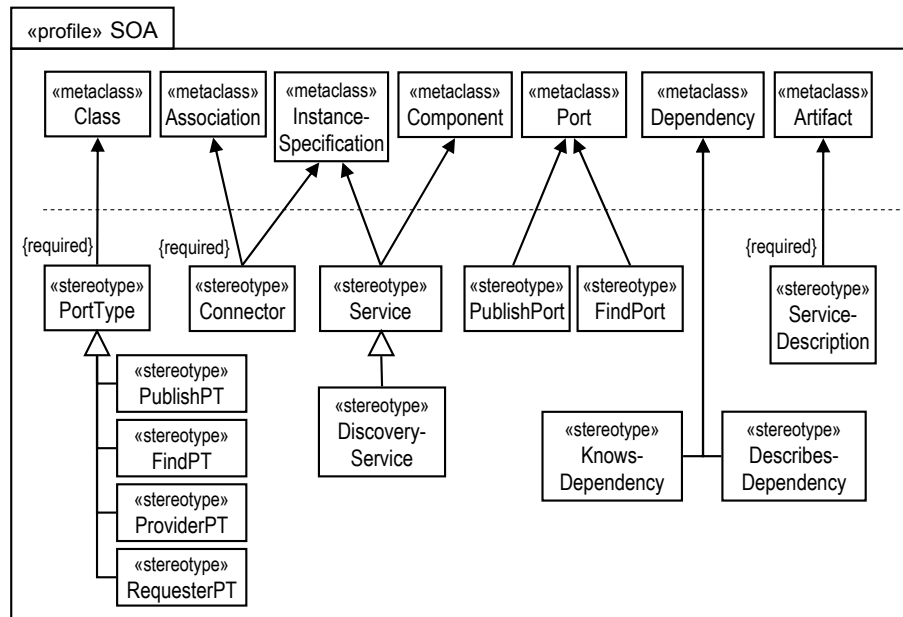


Figure 6.1: Stereotype definitions for the SOA-specific UML profile

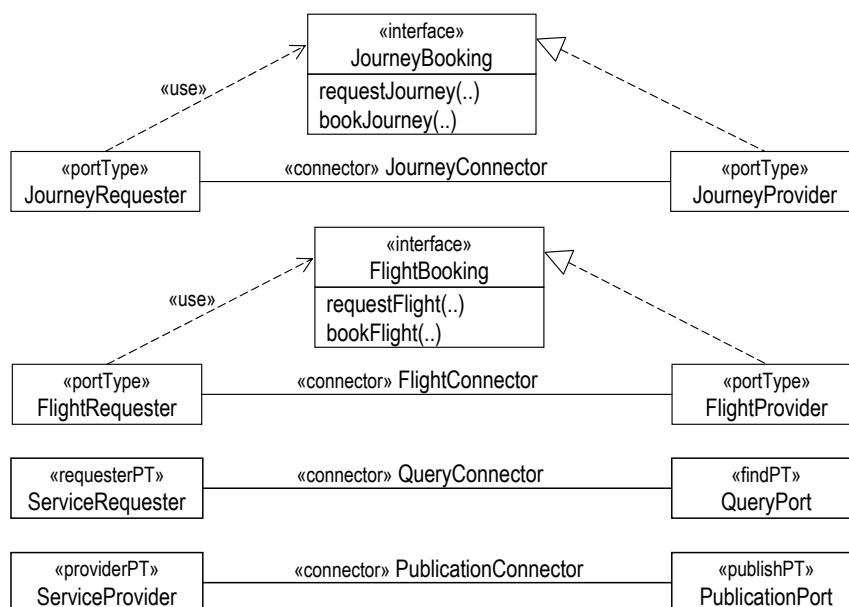


Figure 6.2: Class diagram in the SOA-specific profile

between **«portType»** classes represent connector types; they are extended by the stereotype **Connector** and labeled with **«connector»**. In the same class diagram, we can also specify the interfaces and operations a port type provides (dashed arrow with triangular arrowhead) or requires (**«use»** arrow).

As already shown in Section 3.1.2, component types are defined in UML component diagrams. In the UML abstract syntax, a component type is represented as an instance of the meta-class **Component**. For the sake of service-oriented architectures, we add the two stereotypes **Service** and **DiscoveryService**. If component diagrams describe architectural configurations at the instance level, they apply the meta-class **InstanceSpecification**. Hence, the service stereotypes have to extend this meta-class, too.

As UML 2.0 knows the notion of **Port** as interaction point of a component instance, we do not need a separate stereotype for ports in general. But, in order to highlight those ports that are used in connection with service publication and query, we introduce the stereotypes **PublishPort** and **FindPort**.

The stereotype for service descriptions extends the meta-class **Artifact**. Relationships to service descriptions are modeled by the stereotypes **KnowsDependency** and **DescribesDependency**, which extend UML's meta-class **Dependency**.




Note that Figure 6.1 defines the *abstract* syntax of the UML profile only. Although we have already applied the stereotypes in some UML diagrams, we actually had to define their *concrete* notation first. The default convention is to attach the stereotype name, enclosed between a pair of guillemets (**«»**), to the predefined symbol of the meta-class. Nevertheless, we can also define more customized notations. The concrete notation we propose for the structural SOA stereotypes is summarized in Table 6.1.

### 6.1.2 Stereotypes concerning behavior

In Section 5.2.3, we have learned how to include behavior descriptions into the graph-based architectural models. Needless to say, we prefer to carry out such behavior descriptions with UML, too. As already sketched in Section 3.1.2, we choose UML activity diagrams for this purpose because they can easily be adapted to platform mechanisms using stereotyped actions. Readers who are unfamiliar with UML activity diagrams are referred to [41] for some background information.

Before we start to define the necessary stereotypes, we have to point out that, different to our graph-based semantic domain, UML supports design-time behavior specifications only and provides no means to represent runtime information like different process instances (threads), action counters, or variable values. Nevertheless, this is no real deficiency because such information is

Table 6.1: Notation guide for SOA-specific stereotypes

Stereotypes of the SOA profile:	Notation:
PortType	«portType»
PublishPT	«publishPT»
FindPT	«findPT»
ProviderPT	«providerPT»
RequesterPT	«requesterPT»
Connector	«connector» Name
Service	«service» ServiceName
DiscoveryService	«discovery» Name
ServiceDescription	Name 
PublishPort	
FindPort	
KnowsDependency	«know» ----->
DescribesDependency	«describe» ----->

only needed for model execution in the semantic domain and can be omitted when specifying the behavior. Thus, we can safely skip the runtime-related elements of the formal style definition when defining stereotypes for behavior diagrams.

**Actions.** A UML activity diagram consists of a set of actions that are connected by control and data flow edges. Unfortunately, its flow semantics is only informally defined as a token flow [41], and the available action types are lacking formal semantics at all. We deal with this problem by allowing platform-specific actions only, whose semantics has already been well defined by our graph transformation rules. The control flow parts of our graph transformation rules even conform to the UML token flow semantics (always a single control token per activity diagram execution).

To assign the UML actions to the operations of the architectural style,

we supplement the UML profile with an abstract stereotype **PlatformAction** which is marked as **required** for every action used in an activity diagram. From this abstract stereotype, we derive a concrete stereotypes for every single platform mechanisms defined in the SOA style (see Fig. 6.3). The related action node types (and corresponding transformation rule) of the original style definition can be found in package **Actions** in Appendix B.

**Parameters.** In graph-based behavior descriptions as per Section 5.2.3, actions are parameterized by constant and variable parameters. We incorporate such parameters into our UML models in the following two ways:

1. For constant parameters, we propose to add attributes to the stereotype definitions. Whenever the attributed stereotype is applied to model an action, the value assigned to the attribute represents the value of the constant parameter. For example, the **newThread** action stereotype in Fig. 6.3 gets the attribute **process**, which determines one of the available activity diagrams to be instantiated when the action is performed.
2. Variable parameters need to be treated differently because their values are shared by several actions. For this reason, we propose to model variables as *UML data store nodes* extended by the special stereotype **Variable**. A data store node in a UML activity diagram represents a container in which actions can place an output value and from which several subsequent actions can consume this value as input [41]. The stereotype **Variable** is required to allow a distinguished notation (**«var»** instead of **«datastore»**). Output parameters can now be modeled by a data flow edge outgoing from an action node to a variable node, and input variables by an incoming data flow edge.

**Diagram examples.** Figure 6.4 presents an example of a SOA-specific activity diagram with the above introduced stereotypes. All actions carry the corresponding stereotype label. If required, attribute values are printed in parentheses below the stereotype label.

The activity diagram describes the behavior of **Client** components which want to book a journey for their owner. Right after its instantiation, a new thread starts with opening a port of type **JourneyRequester**. A reference to this port is assigned to a variable which is read by five subsequent actions. For better readability, UML allows to interrupt the corresponding data flow edges and to relate the two ends by labeled circles [41].

After the new **JourneyRequester** port has been opened, the **Client** component wants to connect this port to an appropriate journey booking ser-



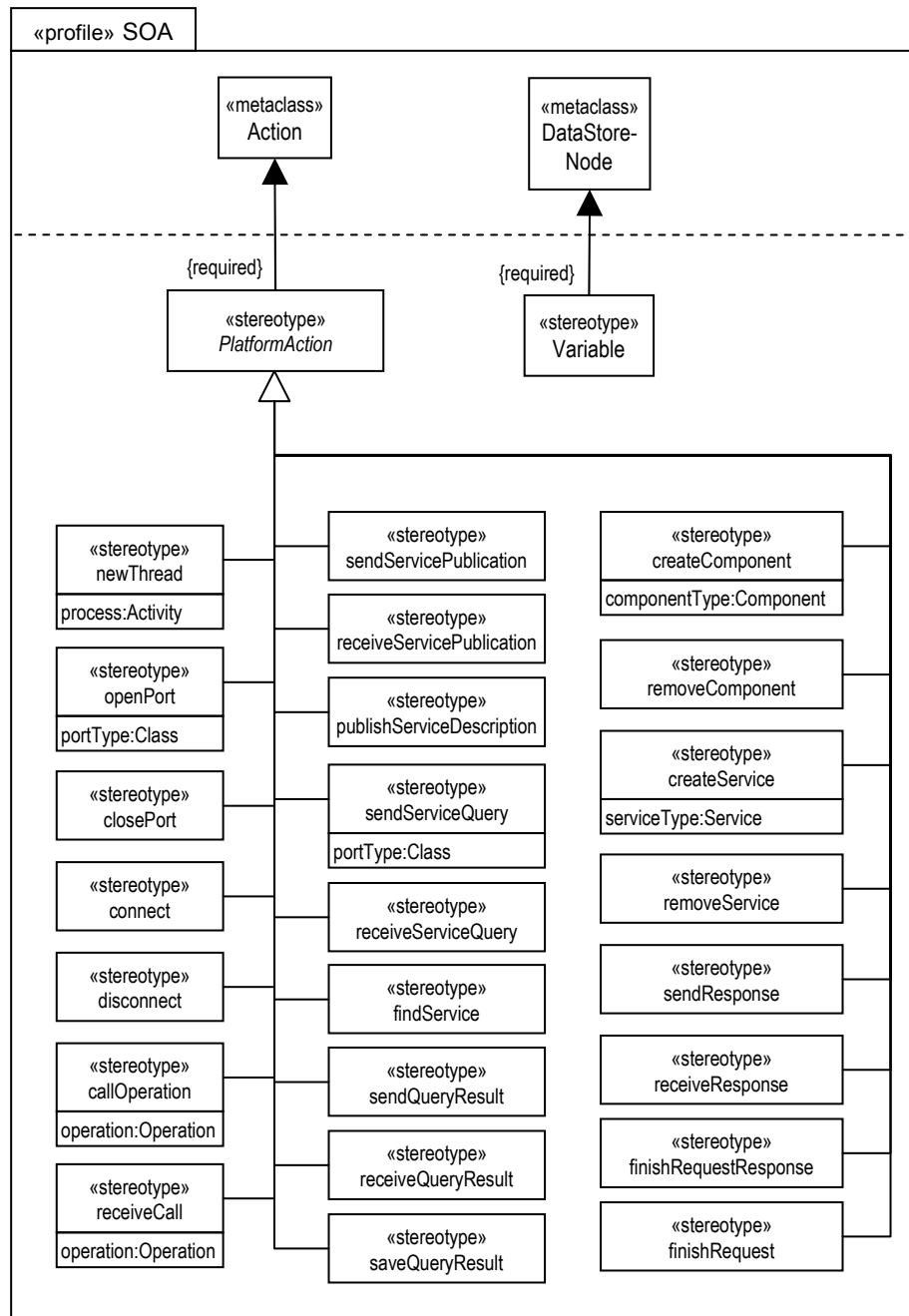


Figure 6.3: Behavior-related stereotype definitions

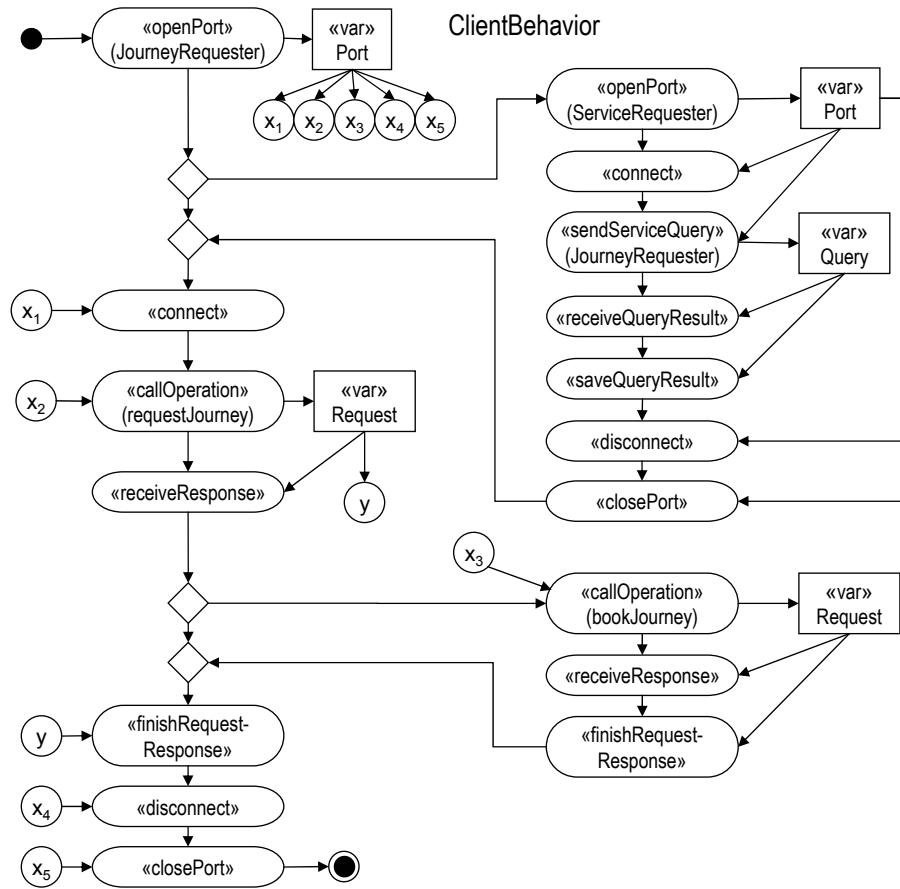


Figure 6.4: Activity diagram for Client in the SOA-specific style

vice. However, as the service-oriented style enables connections only if the requester knows the description of the required service, the control flow provides an additional branch for retrieving this service description first. The branch is optional because the client might already know suitable descriptions from previous lookups.

The service discovery branch starts with opening another port, this time of type **ServiceRequester**. According to the class diagram in Fig. 6.2, this port type can only be connected to a `«findPT»` port of a discovery agency. If the **Client** component knows the description of such a discovery service (as it is the case in Fig. 3.5), the subsequent actions will succeed in connecting to the discovery service, sending a service query for the **JourneyRequester** port, and receiving a suitable service description.

As soon as the requested service accepts a new connection (synchronized action!), the following `«connect»` action is enabled and creates a connector between service requester and requested service. Afterwards, the client calls the service's **requestJourney** operation and waits for a response with suitable journey offers. It can then either book the journey by sending another request or finish the communication without booking the offered journey.

The complementary behavior of the **TravelAgency** service is shown in Fig. 6.5. The process starts with opening a new **JourneyProvider** port at which the service accepts incoming journey requests.

According to the semantics of the SOA style, the following `«connect»` action is a synchronization point with a client who want to address the new port (see transformation rule **connect** in Appendix B). Hence, the two `«connect»` actions in **ClientBehavior** and **TravelAgencyBehavior** will be performed simultaneously.

By completing the `«connect»` action, the **JourneyProvider** port is occupied by the new connector. The next action, `«receiveCall»`, can only be performed after the service requester has called the **requestJourney** operation at the **JourneyProvider** port. It stores the incoming request in a variable and triggers the organization of a journey according to the customers demands. For this purpose, the travel agency opens a **FlightRequester** port and connects to an appropriate flight booking service. However, if it does not know the corresponding service description yet, it has to perform the optional service query branch first.

For simplicity reasons, we restrict the example to searching a suitable flight and leave out the hotel booking. After the travel agency has received the flight information, it can prepare and send a response message back to the client. The client evaluates this response and decides afterwards, whether to book the offered journey or not (cf. Fig. 6.4). If the decision is positive, the

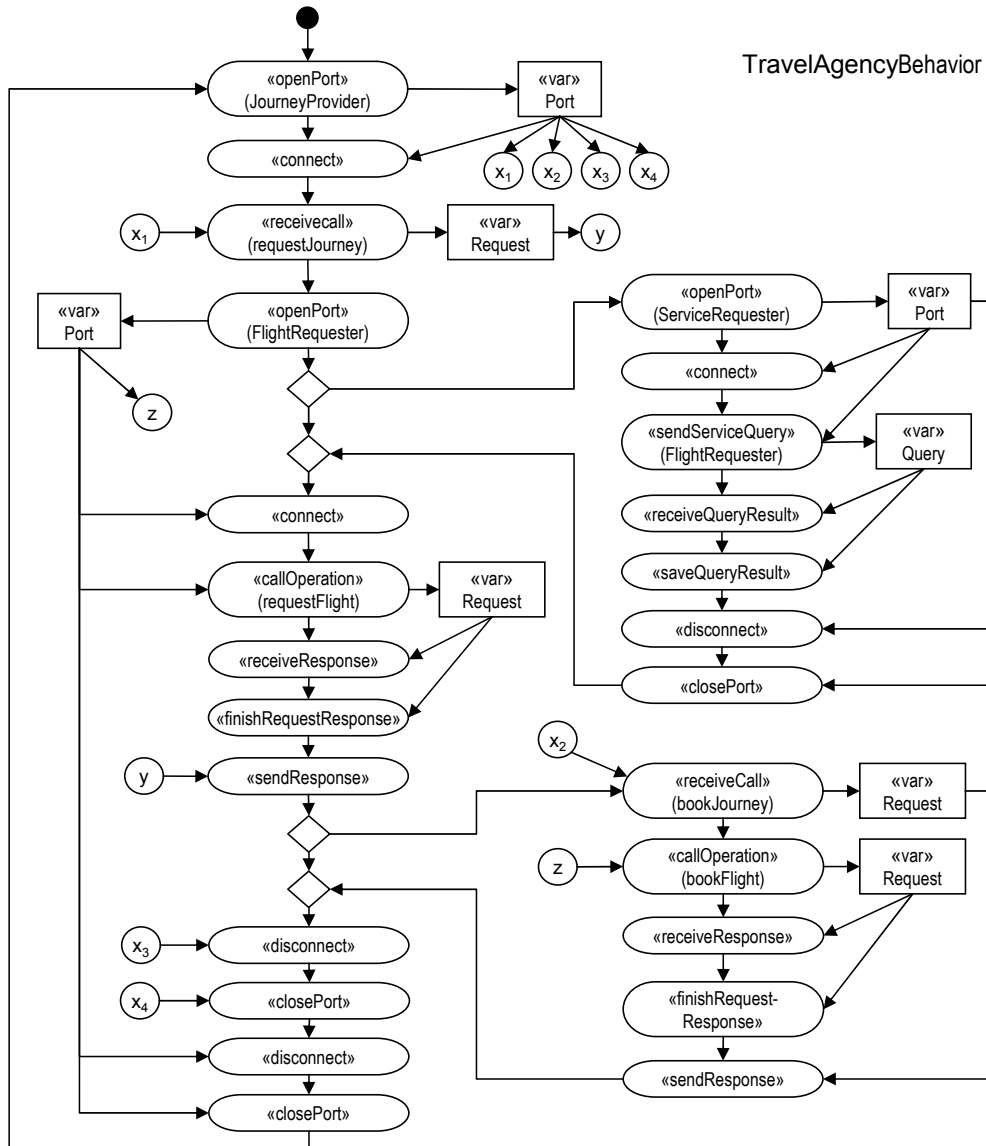


Figure 6.5: Activity diagram for TravelAgency in the SOA-specific style

client will not agree in a «disconnect» action but send another request. This triggers the lower right branch of the **TravelAgencyBehavior** which actually books the proposed flight at the airline service.

After a confirmation has been sent back to the client, the travel agency behavior returns to its main path where now both parties agree in the «disconnect» action. Finally, the connection to the flight booking service and the two ports are closed again. Then, the request is completely processed.

The loop back to the start action indicates that the travel agency can serve multiple incoming requests in sequential order. If we wanted to model a travel agency which is able to serve multiple requests at the same time, we could have inserted a «newThread» action after the first «connect» action which recursively creates a new, parallel instance of the **TravelAgencyBehavior** process as soon as the **JourneyProvider** port has been occupied. The new thread would open another **JourneyProvider** port and could then establish a connection to another client.

As we want to use the graph transformation system of the architectural style as semantic domain, another purpose of the profile is to restrict the UML activity diagrams to those constructs only which have been selected as counterparts for the graph-based behavior descriptions. The examples in Figure 6.4 and 6.5 already reveal which constructs the SOA profile allows; others are excluded. For example, the activity diagrams must not contain guarded edges because decisions are always made non-deterministically, and they must not contain join and fork nodes because concurrency is expressed by starting a new thread rather than by splitting one (see Chapter 5).

A translation between UML models and their corresponding graph-based representation in the semantic domain is discussed in the following section.

## 6.2 Translation into the semantic domain

Figure 6.6 provides an overview of a translation chain from UML diagrams to runtime models in the semantic domain and back again. The translations at the left-hand side consider the conversion of UML models from concrete to abstract syntax and vice versa. They are not part of further consideration because they can be performed by suitable UML tools: A UML editor stores a given set of diagrams internally as instance of the (profile-extended) UML meta-model. For the opposite direction, the editor provides some layout algorithm which renders a given instance of the meta-model as UML diagrams in the concrete syntax.

The translations at the right-hand side consider the step from instance

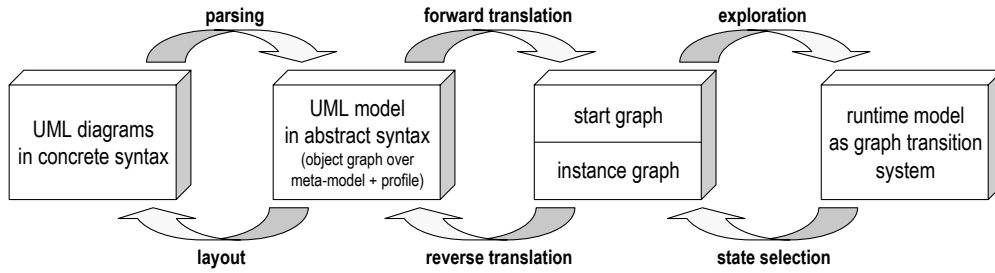


Figure 6.6: Translation chain from UML diagrams to graph transition systems

graphs to graph transition systems, i. e., from single runtime states to complete runtime models. The exploration of a graph transition system from a given start graph has already been dealt with in Section 4.3.2. The reverse “translation” is trivial because we simply select single runtime states of the transition system, which shall be translated into the UML representation.

What remains are the translations of the UML abstract syntax graph into the semantic domain, i. e., into an instance graph of the architectural style, and back again. However, before we can consider translations between these two representations, we have to recall the above mentioned difference to the semantic domain concerning runtime information: UML models show architectural behavior from a design time point of view and do not contain runtime information required during model interpretation. On the contrary, the instance graphs are also intended to capture individual runtime states including, e. g., current variable values, action counters pointing to the previous or next action, and the current “location” of signals, messages, and so forth. This information makes the instance graphs directly interpretable using graph transformation rules.

Although UML models do not include such runtime information, they can at least be used to derive a graph-based representation of the *initial* runtime state. This *start graph* is constructed out of the UML type and process definitions in conjunction with an initial configuration in which all active architecture elements are set to their initial state by default. As a direct consequence, the forward translation in Fig. 6.6 is not surjective with respect to the set of all possible instance graphs.

In the opposite direction, the translation of instance graphs back to UML models can only be partial, leaving out the runtime-related information. Hence, this reverse translation is not injective, because instance graphs differing in runtime aspects only will be translated to the same UML model.

As claimed in Fig. 3.1 and the related discussion in Section 3.1, the trans-

lations between syntax and semantic domain shall be defined once at the style level and reused for every given instance of the architectural style. For this purpose, we have to relate the relevant elements of the UML meta-model (including the style-specific UML profile) with the elements of the style's graph schema. Table 6.2 summarizes the resulting mapping for the SOA style.

For an automated translation, however, the mapping shown in Table 6.2 is not sufficient, because a translator cannot simply replace nodes and edges one by one in a complex graph structure. As both source and target of the translation process are represented as graphs, we rather have to derive a suitable *graph translator* from the given correspondence relation. One existing approach for specifying such graph translators is based on *triple graph grammars* (TGG).

Triple graph grammars are an extension of pair grammars [127]. They specify the simultaneous construction of two graphs. In addition, they build up a third graph which shows the correspondences of elements of the first graph to elements of the second one. Besides the synchronous construction of the three graphs, a triple graph grammar can also be used to translate graphs in either direction, or to check the consistency between two given graphs according to the correspondence graph [138].

This way, triple graph grammars are ideally suited to specify bidirectional translations between instances of two different graph schemas – in our case, the UML meta-model and the graph schema of the architectural style. In the past, they have also been used to build tightly-integrated software environments (IPSEN, [92]) or to specify the migration of relational to object-oriented database schemas [79].

While pair grammars are restricted to context-free productions and one-to-one correspondences between graph elements only, triple graph grammars allow for context-sensitive productions with rather complex left- and right-hand sides and m-to-n correspondence relationships.

For example, the triple graph production in Fig. 6.7 inserts a new instance of a given service type and an accompanying service description into a SOA model. The left production extends the UML abstract syntax graph by stereotyped **InstanceSpecification**, **Artifact**, and **Dependency** nodes, while the right production extends the semantic instance graph by new **Service** and **ServiceDescription** nodes. The production in the middle extends the correspondence graph.

The correspondence graph fixes the relationships between both sides by two graph morphisms, denoted as dashed arrows. In triple graph productions, the correspondence morphisms enable useful application conditions which ensure that the production is applied to already related elements only.

Table 6.2: Mapping between SOA style and UML meta-model

SOA type graph elements	UML meta-model and profile elements (package name::class name)
ComponentType	BasicComponents::Component
Component	Kernel::InstanceSpecification
ServiceType	BasicComponents::Component extended by SOA::Service
Service	Kernel::InstanceSpecification extended by SOA::Service
DiscoveryServiceType	BasicComponents::Component extended by SOA::DiscoveryService
DiscoveryService	Kernel::InstanceSpecification extended by SOA::DiscoveryService
PortType	Kernel::Class extended by SOA::PortType
ProviderPT	Kernel::Class extended by SOA::ProviderPT
PublishPT	Kernel::Class extended by SOA::PublishPT
FindPT	Kernel::Class extended by SOA::FindPT
RequesterPT	Kernel::Class extended by SOA::RequesterPT
Port	[If self.portType.oclsTypeOf(PortType)] Ports::Port
	[If self.portType.oclsTypeOf(PublishPT) or self.portType.oclsTypeOf(ProviderPT)] Ports::Port extended by SOA::PublishPT
	[If self.portType.oclsTypeOf(FindPT) or self.portType.oclsTypeOf(RequesterPT)] Ports::Port extended by SOA::FindPT
ConnectorType	Kernel::Association extended by SOA::Connector
Connector	Kernel::InstanceSpecification
Interface	Interfaces::Interface
Operation	Kernel::Operation
ServiceDescription	Artifacts::Artifact extended by SOA::ServiceDescription
knows	Dependencies::Dependency extended by SOA::KnowsDependency
describes	Dependencies::Dependency extended by SOA::DescribesDependency
Process	FundamentalActivities::Activity
Variable	CompleteActivities::DataStoreNode extended by SOA::Variable
next	BasicActivities::ControlFlow
OpenPortAction	FundamentalActivities::Action extended by SOA::openPort
ClosePortAction	FundamentalActivities::Action extended by SOA::closePort
ConnectAction	FundamentalActivities::Action extended by SOA::connect
DisconnectAction	FundamentalActivities::Action extended by SOA::disconnect
CallOperationAction	FundamentalActivities::Action extended by SOA::callOperation
...	...



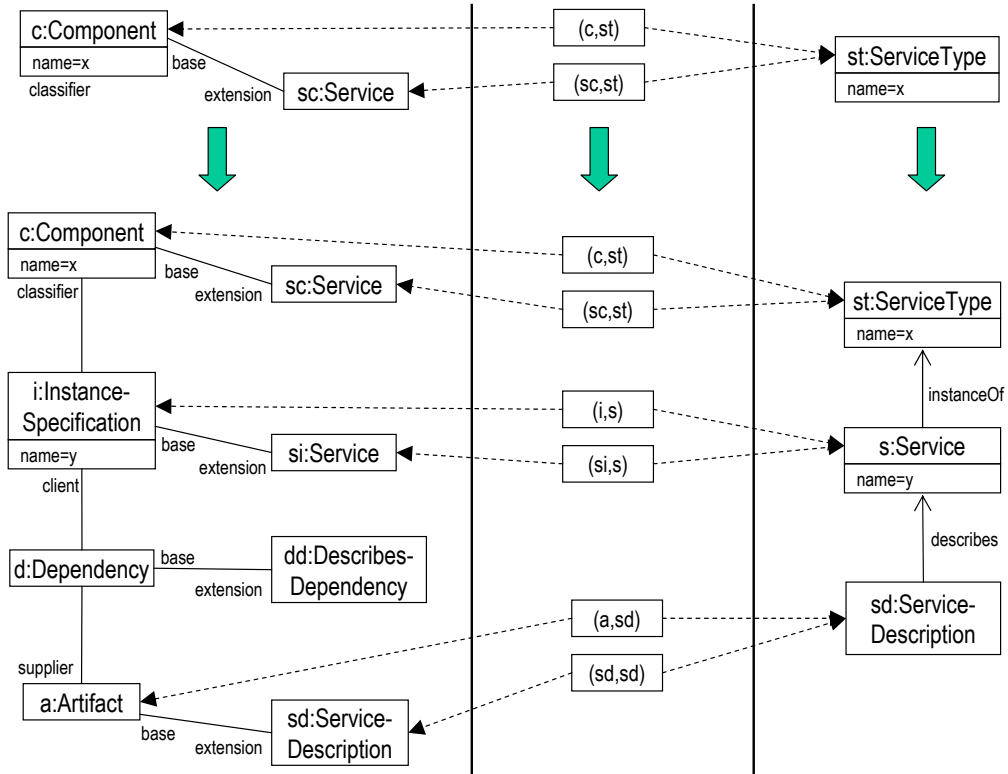


Figure 6.7: Triple graph production

Using graph morphisms entails that one can relate nodes to nodes and edges to edges, but not nodes to edges (cf. Definition 4.1). In Fig. 6.7 for example, the natural correspondence between the **DescribesDependency** node in the left production and the **describes** edge in the right production cannot be expressed. However, in most of such cases it is sufficient to relate the source and target nodes of the particular edge. If this does not suffice, a possible workaround is a more abstract representation of edges as edge-node-edge combinations.

Having defined triple graph productions which generate the complete graph languages – UML abstract syntax graphs on the one hand and instance graphs of the architectural style on the other hand, the triple graph grammar can be used to construct two graph translators. We confine the following explanations to left-right translations (LR) – in our case from UML into the semantic domain. The opposite direction works analogously.

In order to obtain an LR-translator, every triple production is split into two graph productions, a *left-to-right translating production* and a *left-local*

*production*. The left-to-right translating production is the original triple graph production, but the left-hand side of its left production being replaced by a copy of the right-hand side. This way, the left-to-right translating productions do not change the already given left graph any more but construct the right graph only. The original left production, however, is copied into the left-local production.

Given a left graph  $G_L$  to be translated into a right graph  $G_R$ , at first a *graph parser* has to determine a sequence of left-local productions which construct exactly  $G_L$ . Then, applying the corresponding sequence of left-to-right productions to  $G_L$  will produce the desired translation  $G_R$  [138].

To enable efficient parsing of the source graph, triple graph productions are required to be monotonic, i. e., they must not delete any node or edge. Moreover, to ensure termination of the parsing process, the source graph has to be finite and the left productions (for right-left translations: the right productions) have to be *strictly* monotonic, i. e., they have to create at least one new node or edge.

Due to the divergent expressiveness of our two graph languages, the last restriction requires further consideration: As the semantic instance graph contains runtime information not included in the UML syntax graph, the corresponding triple graph productions can create these elements on the right side but no equivalent elements on the left side. An example is shown in Fig. 6.8. It produces a new **Thread** node in the right graph but nothing equivalent in the UML abstract syntax graph on the left.

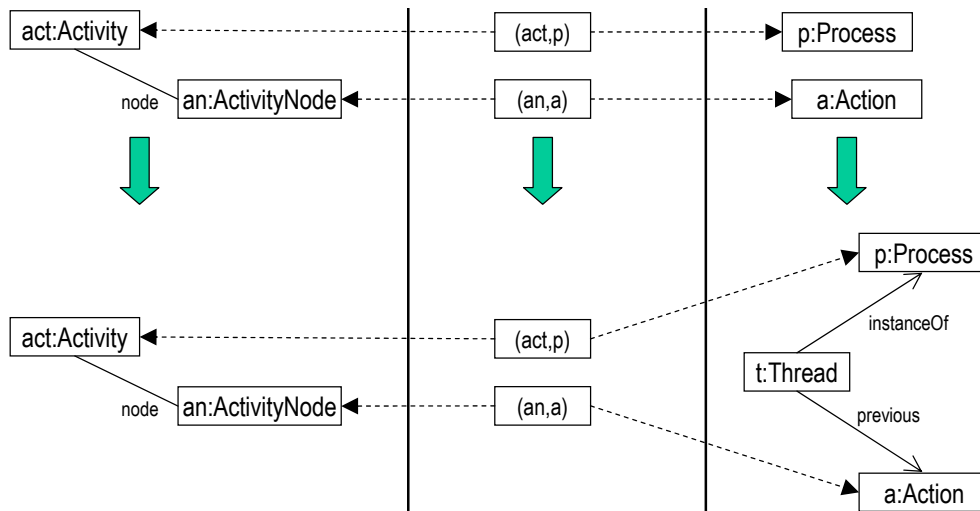


Figure 6.8: Unidirectional triple graph production

As the left production in Fig. 6.8 breaks the strict monotonicity condition, it is forbidden for LR-translations. However, the triple graph production is still required for a complete parsing of the source graph in RL-translations. In other words, we handle the divergent expressiveness by “unidirectional” triple graph productions which are enable for translations in one direction only.

The above discussion and examples reveal the suitability of triple graph grammars for both forward and reverse translations as required in the translation chain of Fig. 6.6. However, a complete specification of a triple graph grammar for our service-oriented style and the relevant parts of the UML meta-model is a major task which goes beyond the scope of this thesis.

## 6.3 Summary

This chapter completes the presented approach for *modeling* platform-consistent, dynamic software architectures. We have shown how to incorporate style-specific constructs into the UML in order to allow for more user-friendly notations of platform-consistent architecture models. The approach has been illustrated by a UML profile for our service-oriented architectural style and its application to the online travel agency system.

In the second section, we have proposed triple graph grammars as an appropriate technique for specifying machine-executable translations between the extended UML models and their equivalent representation in our semantic domain. This way, application architects can design their architectures using a style-specific UML variant, while CASE tools exploit the operational semantics of graph transformation systems in the background (cf. Fig. 3.1).

However, the problem of describing dynamic software architectures in a platform-consistent way is not just a matter of language with syntax and semantics, but also a matter of engineering in the sense of bringing functional requirements together with platform restrictions and capabilities. For this reason, the next chapter goes another step further and details our stepwise development approach, which is based on systematically refining abstract models to more platform-specific ones.



# Chapter 7

## Style-based refinement of dynamic software architectures

In the proposed MDA-like development approach, application architects model system architectures at different levels of platform abstraction – from abstract, business-oriented architectures to concrete, platform-specific ones. However, as already stated at the beginning of the thesis, they have to ensure the mutual consistency of these models in terms of structural and behavioral refinement.

In this chapter, we will discuss suitable refinement criteria for this purpose. According to the requirements outlined in Section 2.2.3, such criteria should enable the application architect to check whether a concrete architecture contains equivalent functional elements as the abstract architecture (*structural refinement* (19)), whether it preserves the abstract behavior (*behavior preservation* (20)), and whether it can always be observed as a substitution of the abstract architecture (*observational substitutability* (21)).

The discussion in Section 2.3.3 has shown that existing work about architectural refinement is rare, and the few available proposals mainly focus on structural refinement only. Hence, we will especially concentrate on criteria for behavioral refinement.

Continuing our style-based strategy, we propose to define suitable relationships between the involved architectural styles and their vocabularies. Then, we can parameterize our refinement criteria by such a style relationship so that they become applicable to any two instances of the architectural styles [13, 48].

The chapter is organized as follows. In Section 7.1, we consider related work and compare two fundamental strategies for model refinement, namely syntactical versus semantical approaches. Favoring the latter ones, we will continue in Section 7.2 with representing correspondences between an ab-

abstract and a concrete architectural style using *abstraction relationships*. These abstractions allow to map concrete states to the abstract level, which will become the basis of *structural refinement* as defined in Section 7.2.2. The structural refinement relationship can then be used to compare individual runtime states of the system.

In Section 7.3, we show how behavioral refinement can be reduced to structural refinement by comparing the reachable states of the induced graph transition systems. We will define formal criteria for both behavior preservation and observational substitutability and provide efficiently computable algorithms which check these properties for a given pair of abstract and concrete architectures. Section 7.4 contains some notes on how to *test* the two properties in cases where the size of our models exceed the practical limit for complete refinement analysis. Eventually, Section 7.5 summarizes the results.

As Chapter 6 has already shown how architectural UML models can be converted into their corresponding graph-based representation, it is sufficient to confine the following discussion about refinement to the graph-based semantic domain only.

## 7.1 Syntactical versus semantical approaches

As we formally define architectures as graph transformation models, related work on refining graph transformation models becomes of special interest at this point. Two important proposals on this topic can be found in Heckel et al. [58] and Große-Rhode et al. [55]. Both approaches base the refinement of a graph transformation system on a mapping between the involved type graphs and graph transformation rules. As one can check the existence of such mappings by looking at the two specifications only, we call their approaches *syntactical*.

Große-Rhode et al. [55], for instance, propose a refinement relationship between abstract and concrete rules that can be checked syntactically. One of the conditions requires that, e.g., the abstract rule and its refinement must have the same pre- and post-conditions except for retyping. Based on this very restrictive definition they can prove that the application of the concrete rule expression yields the same behavior as the corresponding abstract rule. The drawback of this approach is that it cannot handle those cases where the refining rule expression should have additional effects on platform-specific elements that do not occur in the abstract rule. And, due to the fixed rule mapping, the approach does not allow different, context-dependent refinements of the same abstract rule (23).

Similarly, the work by Heckel et al. [58] is based on a syntactical rela-

tionship between two graph transformation systems. This approach is less restrictive and allows additional (platform-specific) elements at the concrete level. However, it still requires some intuitive correspondences between abstract and concrete rules in order to establish a meaningful relationship. This makes its application difficult when the communication and reconfiguration rules defined at the two levels of abstraction differ to a very large extent (22).

Heckel et al. [58] prove that their refinement relationship guarantees *consistency* and *completeness*, which are similar properties as observational substitutability and behavior preservation, respectively.

Consistency means that, whenever the abstract system respects a certain consistency restriction in the sense that no invalid state is reachable, so does the concrete system. However, this property is not as powerful as observational substitutability because the consistency restrictions refer to individual states only and do not allow statements about execution traces. For instance, it is not possible to express a temporal safety property such as “every state with user access to a file requires a preceding log in state”. For observational substitutability, however, we also require the preservation of all such safety properties which hold for the abstract system (21).

Completeness means that all transformation sequences at the abstract level have a corresponding transformation sequence at the concrete level. At first glance, this property sounds similar to behavior preservation. However, we have to point at another drawback of the two syntactical approaches [55] and [58], namely that they compare graph transformation *systems* without start graph only but not graph transformation *models* according to Definition 4.19. Neglecting the start graph reduces the completeness property to a general statement about the existence of a corresponding concrete transformation sequence; one cannot conclude that this transformation sequence is still possible in a specific model, whose set of possible sequences is crucially restricted by the given start graph [58].

This observation reveals that we have to take both the transformation rules and the start graph of our graph transformation models into account when checking behavioral refinement. The combination of both aspects leads us to a comparison of the possible transformation sequences of the abstract and the concrete model. As these sequences can only be computed by executing the two models according to their operational semantics, we call such an approach *semantical*.

A suitable semantic domain for model executions are *graph transition systems* as introduced in Section 4.3.2. For this reason, we now turn to related work on the refinement of transition systems which have been well researched in the area of *concurrent system verification*. In this field, systems

and programs are usually modeled by some kind of state transition system (sometimes also called *automaton*). And, a number of refinement and simulation relations such as the well-known *bisimulation* have been defined in order to compare the runtime behavior of two models [1, 35, 152, 66, 96, 109].

Typical purposes are proving the correctness of an implementation with respect to its specification [1] and reducing the size of a model while preserving its behavioral properties. Besides, similar concepts have successfully been applied to other domains such as object-oriented modeling, e.g., in order to compare the behavior of an abstract superclass with that of its subclasses [36].

The general strategy is to define a simulation relation or homomorphism between the individual states and transitions of two systems, and to prove that the existence of such a relation entails a certain refinement property over their execution traces (e.g., trace inclusion). The existing proposals can be subdivided into relations building on a fixed mapping between state labels [1, 35, 66] and those between transition types [96]. As a transition usually represents some action of the system, the latter is also known under terms like *action refinement* [53].

In the case of graph transition systems, where states represent instance graphs and transitions represent possible transformation rule applications, it is not meaningful to build a refinement relation on a mapping between transition types: A fixed mapping between the atomic names of abstract and concrete rules could not capture information about the part of the graph to which a rule has been applied. If a rule became applicable to several subgraphs of a certain state, we would also get several outgoing transitions. Then, a simple matching of rule names could not reveal any more which of the transitions actually corresponds to the current counterpart in the other system.

Moreover, a fixed mapping between abstract and concrete graph transformation rules would prevent the necessary variability we require for context-dependent refinements (23). Our intention rather is to allow different refinements of the same abstract operation depending on the current state and previous operations. Another disadvantage of fixed rule mappings is the difficulty to establish them in cases where the transformation rules of the involved architectural styles completely diverge.

For these reasons, we will have to compare the graph transition systems on the basis of their states, taking the operational semantics and the current application context of the transformation rules into account. The challenge here is that the states are not just characterized by a finite set of atomic labels but consist of complete configuration in terms of instance graphs. Therefore, we will at first introduce a strategy to compare instance graphs of different architectural styles, called structural refinement. Then, we will continue



with proposing suitable simulation relations that make use of the structural refinement criterion in order to match corresponding abstract and concrete states.

As the existence of an appropriate simulation relation shall be used as sufficient condition for behavioral refinement properties, our intention is to tie this condition as lax as possible. For this reason, we do not want to require one-to-one or one-to-many mappings (also expressed by *functions* or *homomorphisms*) between the transition systems as proposed in [1, 36], but try to allow many-to-many *relations* between states which still entail the intended refinement properties. This way, we reduce the number of cases which fail in satisfying the sufficient condition although being a valid refinement.

When checking behavioral refinement semantically, we have to analyze the entire state space. Unfortunately, this will reduce the applicability of our approach to systems with finite state space only. However, since graph transition systems consider isomorphic states, an infinite state space can only be caused by an unbounded growth in the number of architectural elements. While this appears to be out of practical relevance, we are confronted with another problem: the *state explosion* caused by combinatorial combinations of concurrent actions [26]. This problem will affect the performance of any refinement checking algorithm, even if it is efficient in the size of the transition systems.

An alternative approach proposed by Ebert and Engels [36] circumvents the state explosion problem. They do not compute their refinement homomorphism but give guidelines for constructing the concrete transition system from the abstract one. And, they show that following these guidelines entails the existence of the desired homomorphism. Unfortunately, their approach does not fit to our application domain because it is made for object-oriented behavior specifications with the transition system describing an object's life cycle. As our transition systems are not manually constructed but automatically derived as semantic representation of concurrent architectural behavior, there is no chance of following any guidelines for manual construction. Thus, it appears that we have to live with the state explosion problem at the moment and leave this point open for future work building on innovative approaches such as compositional analysis [52] (see Section 9.2).

## 7.2 Structural refinement

The objective of structural refinement (19) is to compare the structure-related parts of two architectures and to check if all business-relevant and functional entities of the abstract configuration are preserved at the con-

crete level. Our solution to this problem is based on a reusable abstraction relationship between the involved architectural styles which translates the structure-related part of concrete style instances to the abstract level. The resulting structural refinement criterion can also be used to compare individual runtime states of a graph transition system as required for semantical behavioral refinement checks.

### 7.2.1 Abstraction relationship between styles

In order to compare a given pair of abstract and concrete architecture models, we have to relate the modeling elements of the underlying architectural styles. Such a relationship has to overcome the different vocabularies and the different levels of detail.

In Section 3.2, we have already discussed two possible directions of the relationship and emphasized the advantages of abstractions over refinements (cf. Fig. 3.7). Essentially, removing platform-specific details is easier than adding these details. Moreover, adding details amounts to refinement construction, and this implies many degrees of freedom leading to non-determinism or requiring user interactions.

For this reason, we now define an *abstraction function* which translates instances of the concrete architectural style<sup>1</sup> into instances of the abstract architectural style [13, 63, 64].

**Definition 7.1 (Abstraction Function).** *Given an abstract architectural style  $\mathcal{G}$  with graph schema  $GS$  and a concrete architectural style  $\mathcal{G}'$  with graph schema  $GS'$ , an abstraction function  $abs : Inst(GS') \rightarrow Inst(GS)$  translates instance graphs of the concrete style into instance graphs of the abstract style.  $abs$  has to be closed under isomorphism, i. e.,*

$$\forall G', H' \in Inst(GS') : G' \cong H' \implies abs(G') \cong abs(H')$$

The effect of an abstraction function is twofold: Firstly, it has to remove all platform-specific elements which are not considered at the abstract level. Secondly, it has to map the remaining elements of the concrete vocabulary to elements of the abstract vocabulary, i. e., change the typing with respect to the underlying type graphs.

Note that it is sufficient to restrict abstraction functions to *structure-related* model parts only. This is because our strategy for *behavioral* refinement checks is not based on comparing the behavior-related model parts syntactically but on a runtime simulation (see Section 7.3).

---

<sup>1</sup>As a general rule, we denote variables referring to the concrete level by primed letters.

The realization of an abstraction function can range from simple type mappings to complex transformations, depending on the characteristics of the respective architectural styles. While the following paragraphs sketch two examples, the subsequent definitions of refinement criteria in Sections 7.2.2 and 7.3 are intentionally kept independent of the way the abstraction function is defined.

*Example 7.2 (Abstraction function based on type graph mappings).* If the abstraction of a concrete instance graph simply consists of omitting platform-specific elements and adapting the types of business-relevant elements, then the abstraction function  $abs$  can be deduced from a type graph mapping between the abstract type graph  $\widehat{TG} = (TG, I, A)$  and the concrete type graph  $\widehat{TG}' = (TG', I', A')$ . Formally, such a type graph mapping is a partial graph morphism  $t : TG' \rightarrow TG$  which maps elements of  $TG'$  to elements of  $TG$  [13, 14, 58, 63].

Figure 7.1 illustrates some part of a type graph mapping between the concrete type graph  $TG'$  taken from the service-oriented style (Fig. 5.20) and the abstract type graph  $TG$  taken from the component-based style (Fig. 5.1). The complete mapping for all nodes of  $TG'$  is given in Table 7.1.

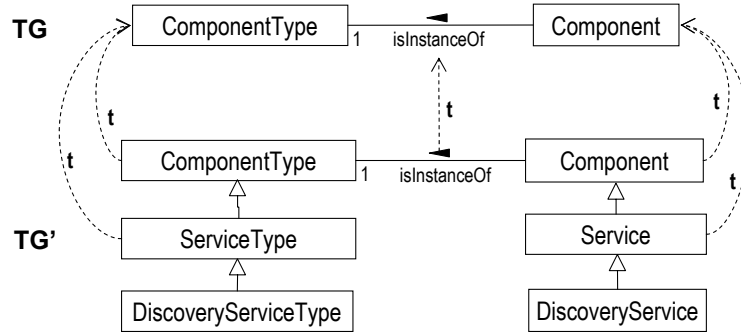


Figure 7.1: Some part of a type graph mapping [14]

The actual definition of  $t$  is driven by semantic correspondences between the elements of the two type graphs. For instance, both **Component** and **Service** nodes in a service-oriented architecture represent what we call a **Component** in the component-based style. Since there is no distinction between private and published components at the abstract level,  $t$  maps both types to **Component**.

Those elements that represent purely platform-specific concepts not occurring at the abstract level like, e.g., **DiscoveryService**, **ServiceDescription**, or the SOA-specific port and message types are not mapped to the abstract type

Table 7.1: Type graph mapping for all nodes of  $TG'$ 

ComponentType	$\mapsto$	ComponentType
Component	$\mapsto$	Component
ServiceType	$\mapsto$	ComponentType
Service	$\mapsto$	Component
DiscoveryServiceType	$\mapsto$	—
DiscoveryService	$\mapsto$	—
ConnectorType	$\mapsto$	ConnectorType
Connector	$\mapsto$	Connector
PortType	$\mapsto$	PortType
FindPT	$\mapsto$	—
PublishPT	$\mapsto$	—
RequesterPT	$\mapsto$	—
ProviderPT	$\mapsto$	—
Port	$\mapsto$	Port
Interface	$\mapsto$	Interface
Operation	$\mapsto$	Operation
Message	$\mapsto$	Message
Request	$\mapsto$	Request
Response	$\mapsto$	Response
ServiceQuery	$\mapsto$	—
QueryResult	$\mapsto$	—
ServicePublication	$\mapsto$	—
ServiceDescription	$\mapsto$	—

graph. For behavior-related elements like, e.g., **Process** and **Action** nodes, the type graph mapping is undefined, too, because the abstraction function only needs to lift structural elements to the abstract level.

The type graph mapping  $t$  induces the desired abstraction function  $abs_t$  which abstracts instance graphs typed over  $TG'$  to those typed over  $TG$ . This abstraction informally consists of (1) renaming the types of all elements whose type has an image in  $TG$  according to the definition of  $t$ , (2) deleting all nodes and edges which, due to the partiality of  $t$ , have a type in  $TG'$  but not in  $TG$ , and (3) deleting all dangling edges and those adjacent nodes whose number of connected neighbor nodes falls below the lower bound of the relevant cardinality constraint. Formally,  $abs_t$  can be defined as a functor of the graph morphism  $t$  [58].

Figure 7.2 illustrates the effect of the abstraction function  $abs_t$  for an instance graph fragment defining the **Airline** service in the SOA style. First, we apply the type mapping  $t$  and rename the types of the **ServiceType** and **Service** nodes into **ComponentType** and **Component**, respectively (1). Then, we delete the **ProviderPT** and **ServiceDescription** nodes and the **describes** edge because

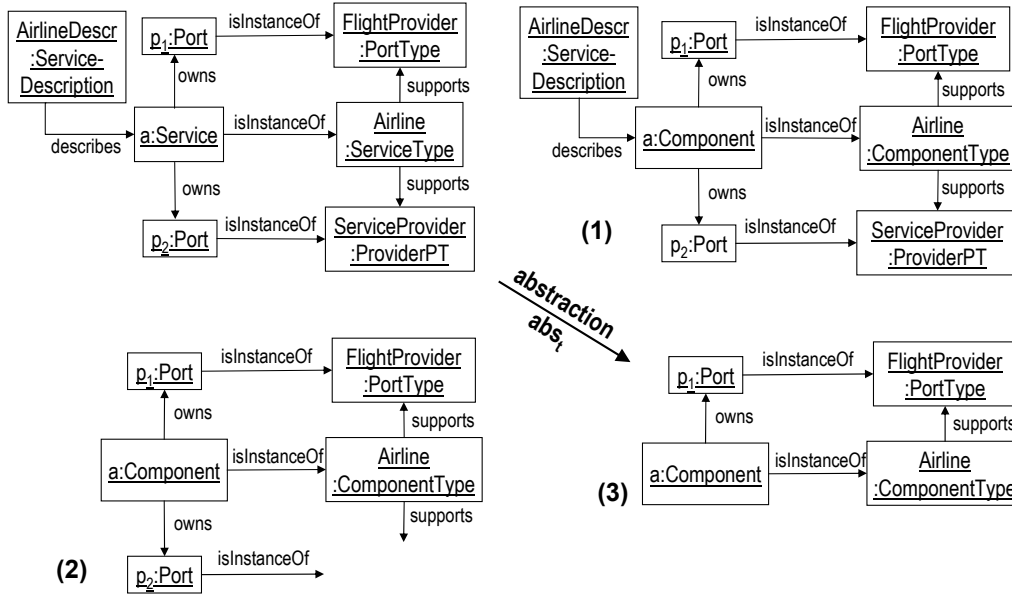


Figure 7.2: Abstraction of an instance graph

they have no mapping to  $TG$  under  $t$  (2). The deletion of the `ProviderPT` node leads to the deletion of the adjacent `Port` node in the third step, because otherwise the cardinality constraint would be violated which says that every `Port` requires a `PortType`. Eventually, all dangling edges are removed (3).

*Example 7.3 (Abstraction function based on triple graph transformations).* Other architectural styles might require more complex transformations which map entire patterns of concrete elements to abstract elements. For instance, an abstract bidirectional channel could be realized by a combination of two unidirectional channels at the platform-specific level. Such complex mappings have to be defined by more sophisticated methods.

One option is to use the *triple graph grammars* already introduced in Section 6.2. Figure 7.3 demonstrates how such a triple transformation rule could look like for the channel example. However, as all subsequent refinement concepts are independent of the actual realization of the abstraction function, we do not want to continue this example in more detail.

### 7.2.2 Structural refinement criterion

When speaking about structural refinement, we are concerned with the comparison of two architectural configurations, formally represented by two instance graphs. Obviously, the main ingredient for such a comparison is an

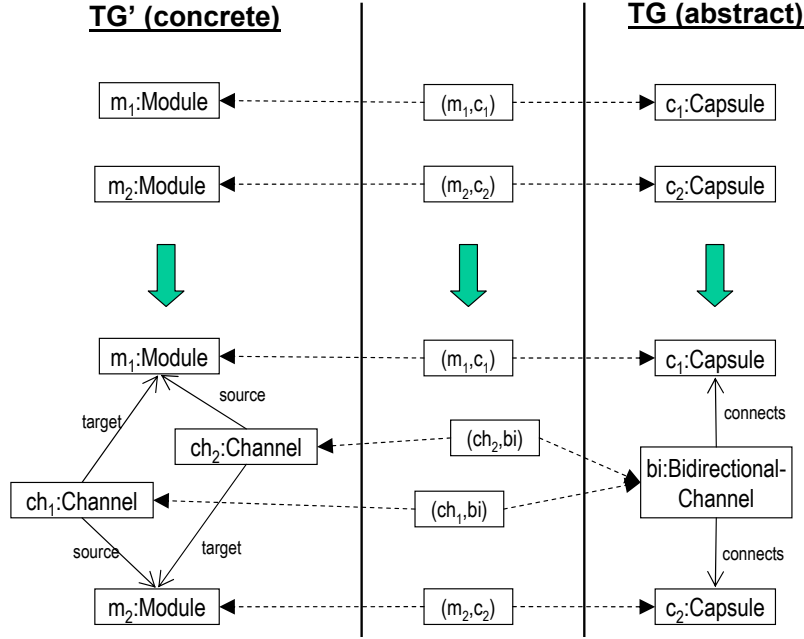


Figure 7.3: Triple graph production as part of an abstraction function

abstraction function in the sense of the previous section. However, as the behavior-related model parts do not matter for structural refinement, we first have to remove all elements which are irrelevant in terms of structure. This has formally been defined as a projection function in Definition 5.1.

Altogether, we can combine abstraction function and structure projection in order to compare the structural characteristics of two architectural configurations. This leads to the following definition of structural refinement as a relation on instance graphs:

**Definition 7.4 (Structural Refinement).** *Given an abstract architectural style  $\mathcal{G}$  with graph schema  $GS$ , a concrete architectural style  $\mathcal{G}'$  with graph schema  $GS'$ , an abstraction function  $abs : Inst(GS') \rightarrow Inst(GS)$ , and two structure projections  $struct_{TG}$  and  $struct_{TG'}$ ; then, structural refinement is defined as a relation  $refines \subseteq Inst(GS') \times Inst(GS)$  with*

$$G' \text{ refines } G \iff abs(struct_{TG'}(G')) \cong struct_{TG}(G)$$

Obviously, it is only possible to compute this refinement criterion for *finite* graphs. This, however, is no real restriction because software architectures and, hence, models thereof are always of limited size.

The computational complexity of deciding whether a concrete architecture structurally refines an abstract architecture or not, depends on the size

of the graphs and the complexity of the chosen abstraction and projection functions. If we realize these functions by simple type mappings as sketched in Example 7.2, then we can approximately assume linear time for their computation, i. e.,  $O(k)$  with  $k = \|G'_N\| + \|G'_E\|$  denoting the size of the concrete graph (which we assume is greater than the size of the abstract one).

However, what counts more in this context is the isomorphism check between the left- and right-hand sides of the criterion. The underlying *graph isomorphism problem* is one of the few problems which is in  $NP$  but is not known to be in either  $P$  or  $NP$ -complete. Hence, there is currently no polynomial-time algorithm for solving the problem in its most general fashion; but, polynomial-time algorithms do exist for special cases of the graph isomorphism problem [82].

Practical experience with the even worse since definitely  $NP$ -complete *subgraph isomorphism problem* in connection with graph transformation rule matchings has shown that there are suitable heuristics for detecting isomorphisms within tolerable running times [162]. According to [162], the following circumstances (among others) support an efficient solution:

- using singleton entry nodes which occur exactly ones in each graph and are specially marked, e. g., by a certain type,
- using a lot of different node and edge types which reduce the number of possible isomorphism candidates,
- using nodes with key attributes and unique values which predetermine some part of the isomorphism already.

As these properties hold in most cases for our instance graphs, too, there is a high probability that structural refinement can efficiently been checked in terms of the graph size.

Before we come to behavioral refinement, we want to point out that the structural refinement relation can easily be lifted to the level of isomorphism classes:

**Lemma 7.5 (Structural Refinement under Isomorphism).**

$$G' \text{ refines } G \implies \forall I' \in [G'], \forall I \in [G] : I' \text{ refines } I$$

*Proof.* We exploit the fact that *abs* and *struct* are closed under isomorphism per definition and state for all  $I' \in [G']$  and all  $I \in [G]$ :

$$\begin{array}{rcl}
& G' \text{ refines } G \\
\stackrel{7.4}{\Longrightarrow} & \text{abs}(\text{struct}_{TG'}(G')) \cong \text{struct}_{TG}(G) \\
\stackrel{7.1,5.1}{\Longrightarrow} & \text{abs}(\text{struct}_{TG'}(I')) \cong \text{struct}_{TG}(I) \\
\stackrel{7.4}{\Longrightarrow} & I' \text{ refines } I
\end{array}$$

□

Hence, the refinement relation on graphs induces a refinement relation on isomorphism classes with  $[G'] \text{ refines } [G] \iff G' \text{ refines } G$ . We will return to this property when checking behavioral refinement with the help of graph transition systems.

## 7.3 Behavioral refinement

Besides the refinement of structure, we also require criteria to check behavioral refinement in terms of behavior preservation and observational substitutability. However, in contrast to structural refinement, we are not going to translate and compare the behavior-related parts of two given instance graphs, but investigate the behavior at a semantic level using the induced graph transition systems as operational runtime models (see Section 7.1).

### 7.3.1 Behavior preservation

The criterion for behavior preservation (20) has to ensure that every behavior of the abstract architecture is still possible in the concrete architecture. The motivation behind is that the business processes incorporated into the business-oriented model shall be preserved when porting the model to the platform-specific level. Consequently, every abstract scenario of communication and reconfiguration operations requires a corresponding scenario in the concrete architecture, and every state, i. e., instance graph, reachable in the abstract system requires the reachability of a corresponding<sup>2</sup> state in the platform-specific architecture.

In our semantic domain, the execution of a communication or reconfiguration operation is modeled by the application of the corresponding graph transformation rule. Hence, every transformation step  $G \Longrightarrow_{\mathcal{G}} H$  in the abstract style  $\mathcal{G}$  has to be preserved by suitable concrete transformation steps  $G' \xRightarrow{*}_{\mathcal{G}'} H'$  in the concrete style  $\mathcal{G}'$ . We allow for several steps at the concrete level because it might require a number of intermediate, platform-specific steps in order to realize the abstract step.

---

<sup>2</sup>By correspondence we mean structural refinement in this context.



*Example 7.6.* Consider an application of the abstract reconfiguration rule **connect** presented in Fig. 5.4. In a behavior-preserving, service-oriented architecture, it is not always enough to apply the corresponding SOA-specific **connect** rule, because the service description has to be known first. If the description is not known yet, further SOA-specific rules for service publication and discovery have to be applied, as illustrated at the bottom of Fig. 7.4 [14, 63].

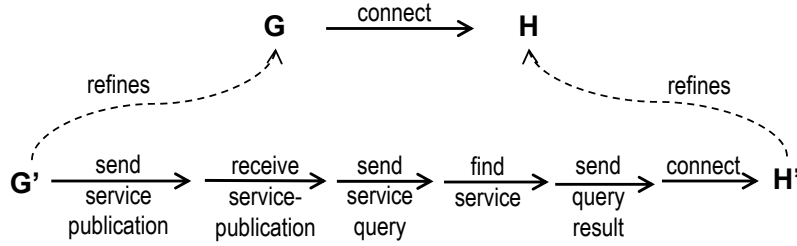


Figure 7.4: Preservation of an abstract transformation step

Extending the above sketched principle to all transformation steps and states reachable in an architecture's behavior results in the following formal definition of behavior preservation:

**Definition 7.7 (Behavior Preservation).** *Let  $M = (\mathcal{G}, G_0)$  be a graph transformation model representing an architecture in an abstract architectural style  $\mathcal{G} = (GS, P)$  with transformation sequences  $Op(M)$ . Let  $M' = (\mathcal{G}', G'_0)$  be a graph transformation model representing an architecture in a concrete architectural style  $\mathcal{G}' = (GS', P')$  with transformation sequences  $Op(M')$ . Given a structural refinement relation  $refines \subseteq Inst(GS') \times Inst(GS)$  according to Definition 7.4, then  $M'$  preserves the behavior of  $M$  ( $M'$  preserves  $M$ , for short), iff*

$$\begin{aligned} &\forall p \in Op(M) \text{ with } p = (G_0 \Rightarrow_{\mathcal{G}} G_1 \Rightarrow_{\mathcal{G}} \dots \Rightarrow_{\mathcal{G}} G_n) \\ &\exists p' \in Op(M') \text{ with } p' = (G'_0 \xRightarrow{*}_{\mathcal{G}'} G'_1 \xRightarrow{*}_{\mathcal{G}'} \dots \xRightarrow{*}_{\mathcal{G}'} G'_n) \\ &\text{such that } \forall i = 0 \dots n : G'_i \text{ refines } G_i \end{aligned}$$

Unfortunately, the definition above cannot directly be put into an algorithm which checks behavior preservation for a given pair of architecture models, because the algorithm would usually have to investigate an infinite number of possible transformation sequences. The problem can only be tackled by skipping recurring states or, in other words, isomorphic graphs.

In Section 4.3.2, we have already pointed out how the reachable states of an architecture model are collected into isomorphism classes which then

induce a *graph transition system*. We will now demonstrate how such graph transition systems can be used to realize an algorithm for checking behavior preservation.

Directly transferring the above definition to graph transition systems would require to prove that all paths in an abstract transition system are also contained in the concrete transition system. However, this problem, also known as *trace inclusion problem*, is difficult to compute because it is PSPACE-complete [147]. Instead, we will check a slightly stronger condition on graph transition systems which entails trace inclusion. It is based on the following, co-inductively defined *simulation relation* between the states of two graph transition systems [63, 64]:

**Definition 7.8 (Simulation Relation  $\text{sim}R^*$ ).** *Given a structural refinement relation  $\text{refines}$  and two graph transition systems  $GTS = (L, S, \Rightarrow, s_0)$  and  $GTS' = (L', S', \Rightarrow, s'_0)$ . Then, a relation  $\text{sim}R^* \subseteq S' \times S$  is a simulation relation between  $GTS'$  and  $GTS$ , if  $(s', s) \in \text{sim}R^*$  implies*

- $s'$  *refines*  $s$
- $\forall \hat{s} \in S$  with  $s \Rightarrow \hat{s}$   $\exists \hat{s}' \in S'$  with  $s' \xRightarrow{*} \hat{s}'$  such that  $(\hat{s}', \hat{s}) \in \text{sim}R^*$

*Example 7.9.* To illustrate this definition, consider the two graph transition systems in Fig. 7.5. The various patterns of the state nodes symbolize the architectural structure of each state from a business-oriented point of view. If a concrete state has the same pattern as an abstract state, then the concrete state structurally refines the abstract one, e.g.,  $s'_6$  *refines*  $s_3$ . Valid simulation relations for these two transition systems are:

$$\begin{aligned}
 \text{sim}R_1^* &= \{(s'_7, s_4)\} \\
 \text{sim}R_2^* &= \{(s'_6, s_3), (s'_7, s_4)\} \\
 \text{sim}R_3^* &= \{(s'_0, s_0), (s'_1, s_1), (s'_4, s_2), (s'_6, s_3), (s'_7, s_4)\} \\
 \text{sim}R_4^* &= \{(s'_0, s_0), (s'_1, s_1), (s'_3, s_2), (s'_6, s_3), (s'_7, s_4)\} \\
 \text{sim}R_5^* &= \{(s'_0, s_0), (s'_2, s_1), (s'_4, s_2), (s'_6, s_3), (s'_7, s_4)\} \\
 \text{sim}R_6^* &= \{(s'_0, s_0), (s'_2, s_1), (s'_3, s_2), (s'_6, s_3), (s'_7, s_4)\} \\
 &\text{and all possible combinations of unions thereof, in particular:} \\
 \text{sim}R_{\cup}^* &= \{(s'_0, s_0), (s'_1, s_1), (s'_2, s_1), (s'_3, s_2), (s'_4, s_2), (s'_6, s_3), (s'_7, s_4)\}
 \end{aligned}$$

Note that none of the simulation relations contains the  $(s'_5, s_1)$ , although  $s'_5$  *refines*  $s_1$  satisfying the first condition of Definition 7.8. The reason is the second, co-inductive condition which this tuple cannot satisfy because none of the (transitive) successors of  $s'_5$  refines  $s_2$ , the successor of  $s_1$ .

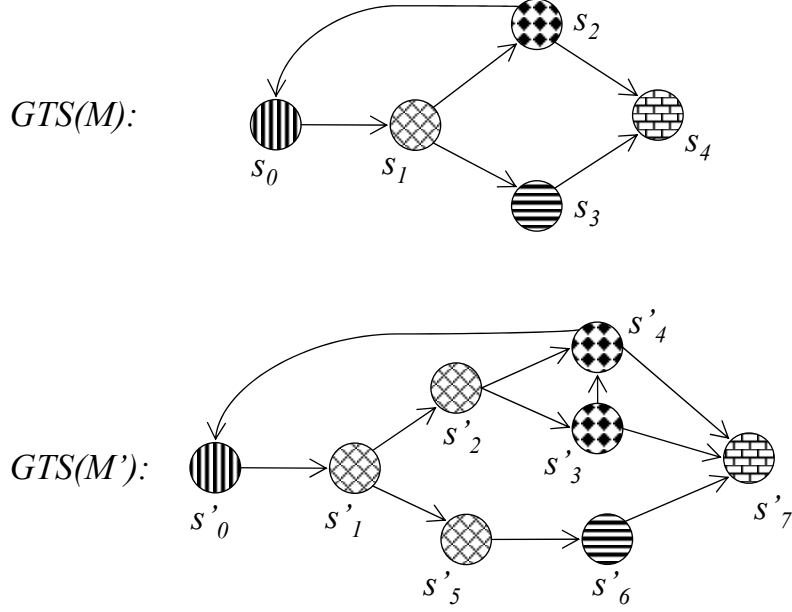


Figure 7.5: Example of abstract and concrete graph transition systems

Nevertheless, this particular tuple does not matter for behavior preservation as long as there is a simulation relation  $simR^*$  containing the two start states  $(s'_0, s_0)$ . The following proposition shows how the existence of such a simulation relation directly entails behavior preservation, i.e.,  $M'$  preserves  $M$ :

**Proposition 7.10 (Behavior Preservation in Graph Transition Systems).** *Let  $M = (\mathcal{G}, G_0)$  be a model over an abstract architectural style  $\mathcal{G} = (GS, P)$  and  $M' = (\mathcal{G}', G'_0)$  be a model over a concrete architectural style  $\mathcal{G}' = (GS', P')$ . Moreover, let  $GTS(M')$  and  $GTS(M)$  be the graph transition systems they induce and  $refines \subseteq Inst(GS') \times Inst(GS)$  a structural refinement relation.*

*Then,  $M'$  preserves  $M$ , if there exists a simulation relation  $simR^*$  between  $GTS(M')$  and  $GTS(M)$  such that  $([G'_0], [G_0]) \in simR^*$ .*

*Proof.* Let  $simR^*$  be the simulation relation between  $GTS(M')$  and  $GTS(M)$  such that  $([G'_0], [G_0]) \in simR^*$ . We show by induction over the path length  $n \in \mathbb{N}_0$  that this entails:

$$\begin{aligned} & \forall p \in Op(M) \text{ with } p = (G_0 \Rightarrow_{\mathcal{G}} G_1 \dots \Rightarrow_{\mathcal{G}} G_n) \\ & \exists p' \in Op(M') \text{ with } p' = (G'_0 \xRightarrow{*}_{\mathcal{G}'} G'_1 \dots \xRightarrow{*}_{\mathcal{G}'} G'_n) \\ & \text{such that } \forall i = 0 \dots n : ([G'_i], [G_i]) \in simR^* \end{aligned}$$

- $n = 0$ :

$$p_0 = (G_0) \text{ and } p'_0 = (G'_0) \text{ and } ([G'_0], [G_0]) \in \text{sim}R^*$$

- $n \rightarrow n + 1$ :

$$\begin{aligned} & \text{Let } p_{n+1} = (G_0 \Rightarrow_{\mathcal{G}} G_1 \dots \Rightarrow_{\mathcal{G}} G_n \Rightarrow_{\mathcal{G}} G_{n+1}) \\ \xRightarrow{\text{Ind.}} & \exists p'_n = (G'_0 \xRightarrow{*}_{\mathcal{G}'} G'_1 \dots \xRightarrow{*}_{\mathcal{G}'} G'_n) \\ & \text{such that } \forall i = 0 \dots n : ([G'_i], [G_i]) \in \text{sim}R^* \\ \xRightarrow{7.8} & \exists [G'_{n+1}] \text{ with } [G'_n] \xRightarrow{*} [G'_{n+1}] \text{ such that } ([G'_{n+1}], [G_{n+1}]) \in \text{sim}R^* \\ \xRightarrow{4.21} & \exists p'_{n+1} = (G'_0 \xRightarrow{*}_{\mathcal{G}'} G'_1 \dots \xRightarrow{*}_{\mathcal{G}'} G'_n \xRightarrow{*}_{\mathcal{G}'} G'_{n+1}) \\ & \text{such that } \forall i = 0 \dots n + 1 : ([G'_i], [G_i]) \in \text{sim}R^* \end{aligned}$$

Altogether, we obtain:

$$\begin{aligned} & \forall p \in \text{Op}(M) \text{ with } p = (G_0 \Rightarrow_{\mathcal{G}} G_1 \dots \Rightarrow_{\mathcal{G}} G_n) \\ & \exists p' \in \text{Op}(M') \text{ with } p' = (G'_0 \xRightarrow{*}_{\mathcal{G}'} G'_1 \dots \xRightarrow{*}_{\mathcal{G}'} G'_n) \\ & \text{such that } \forall i = 0 \dots n : ([G'_i], [G_i]) \in \text{sim}R^* \\ \xRightarrow{7.8} & \forall i = 0 \dots n : [G'_i] \text{ refines } [G_i] \\ \xRightarrow{7.5} & \forall i = 0 \dots n : G'_i \text{ refines } G_i \\ \xRightarrow{7.7} & M' \text{ preserves } M \end{aligned}$$

□

**Computing behavior preservation.** There is a number of similar, co-inductively defined simulation relations in the area of concurrent system verification. For instance, strong and weak bisimulation relations are used to show that an implementation fits to its specification [110]. Independent of the chosen kind of relation, the way of deciding if there is a relation containing the initial system states looks similar in most cases: One computes the greatest relation, which is the union of all possible relations, through a *greatest fixpoint* and checks the membership of the initial states [35, 66].

For computing our behavior preservation relation, we will adopt a fixpoint algorithm presented by Henzinger et al. in [66]. For a given state-labeled transition system, their algorithm computes the greatest fixpoint of a simulation in  $O(mn)$  time, where  $m$  is the number of transitions and  $n$  is the number of states. Adopting a definition by Milner [109], they consider a simulation to be a binary relation  $\leq \subseteq S \times S$  on the state set  $S$  with  $s' \leq s$  implying:<sup>3</sup>

$$\text{label}(s') = \text{label}(s) \tag{7.1}$$

---

<sup>3</sup> $\text{label}(s)$  returns the label of a state  $s$ , and  $\text{post}(s)$  contains all direct successors of  $s$ .

$$\forall \hat{s} \in \text{post}(s) \exists \hat{s}' \in \text{post}(s') \text{ such that } \hat{s}' \leq \hat{s} \quad (7.2)$$

Henzinger et al.'s algorithm [66] takes a state-labeled transition system as input and computes for each state  $s \in S$  the simulator set  $\text{sim}(s)$  containing all states  $s'$  for which there is a simulation relation  $\leq$  such that  $s' \leq s$ . Although the simulator sets are computed for a single transition system only, the algorithm can also be used to compare two transition systems: We just have to construct a joint transition system as the disjoint union of the two single ones.

There are two differences between Henzinger et al.'s definition of a simulation relation and our Definition 7.8, which do not allow to apply their algorithm as it is: (1) we do not compare state labels but check for structural refinement between states; (2) we allow to simulate an abstract step by a *sequence* of concrete steps while equation (7.2) requires a one-to-one correspondence between transitions. Nevertheless, we will show how their algorithm can be adapted to our definition of the behavior preservation problem.

To overcome the first difference, we change the initialization part of their algorithm so that it considers structural refinement instead of state labels. Figure 7.6 shows the modified pseudo code. The only difference to the original version in [66] is line 2 which initializes the simulator sets  $\text{sim}(s)$  with all potential candidates for simulating states. Instead of checking if two states  $s$  and  $s'$  have equal labels, we now call a function `REFINES` which checks if  $s'$  is a structural refinement of  $s$ .

```

PRESERVES ( $TS$ ) : Boolean
1  for all  $s \in S$ 
2     $\text{sim}(s) := \{s' \in S \mid \text{REFINES}(s', s) = \text{TRUE}\}$ 
3
4  while there are three states  $s, \hat{s}, s' \in S$  such that
5     $\hat{s} \in \text{post}(s)$ ,  $s' \in \text{sim}(s)$ , and  $\text{post}(s') \cap \text{sim}(\hat{s}) = \emptyset$  do
6     $\text{sim}(s) := \text{sim}(s) \setminus \{s'\}$ 
7  end while
8
9  if  $s'_0 \in \text{sim}(s_0)$  then
10   return TRUE
11 else
12   return FALSE

```

Figure 7.6: Algorithm for behavior preservation based on [66]

In the second part of the algorithm, the initialized simulator sets are “*sharpened*” by removing those candidates which do not satisfy equation (7.2). Since the number of candidates is finite, the algorithm certainly terminates. And, as pointed out by Henzinger et al., the algorithm can be efficiently implemented such that it computes the fixpoint in  $O(mn)$  time. For details, please refer to their original paper in [66].

To overcome the second difference to Henzinger et al.’s definition of simulation – single steps vs. sequences at the concrete level – we compute the *transitive closure* of the concrete transition system before it is fed into the algorithm. The transitive closure of a transition system  $TS$  is a new transition system  $TS^*$  which contains the same set of states and a transition between any two states  $s_1$  and  $s_2$  whenever there is a path  $s_1 \xrightarrow{*} s_2$  in  $TS$ .

The transitive closure of a directed graph can be computed in  $O(mn)$  time, too [28]. If we do so for the concrete graph transition system  $GTS(M')$  and run our algorithm for the disjoint union of the transitive closure  $GTS(M')^*$  and the abstract transition system  $GTS(M)$  afterwards, then an outcome  $s'_0 \in \text{sim}(s_0)$  means that  $M'$  preserves  $M$  according to Proposition 7.10.

Instead of a formal proof, we argue that starting from  $s'_0$  one can only traverse transitions which belong to the transitive closure  $GTS(M')^*$ . Each of these transitions represents a *path* in the underlying transition system  $GTS(M')$ . Thus, if the outcome of the algorithm satisfies equation (7.2) with respect to  $GTS(M')^*$ , then it satisfies the corresponding equation of Definition 7.8 with respect to  $GTS(M')$ .

*Example 7.11.* Recall the two transition systems  $GTS(M)$  and  $GTS(M')$  in Fig. 7.5. In order to compute the greatest simulation relation  $\text{sim}R^*$  between them, the algorithm is fed with the joint transition system consisting of  $GTS(M)$  and the transitive closure  $GTS(M')^*$  as denoted by the adjacent lists in the left column below. The right column denotes how the algorithm *initializes* the simulator sets according to the structural refinement relationship (cf. Fig. 7.6 ll. 1-2).

In this example, the while loop of the algorithm (Fig. 7.6 ll. 4-7) can only be entered once, namely with  $s = s_1$ ,  $\hat{s} = s_2$ , and  $s' = s'_5$ . After  $s'_5$  has been removed from the simulator set  $\text{sim}(s_1)$ , the loop terminates. Then, the simulator sets represent the greatest simulation relation which we had already constructed as  $\text{sim}R_{\cup}^*$  in Example 7.9. With  $s'_0 \in \text{sim}(s_0)$ , we have shown that  $M'$  preserves  $M$ .

$$\begin{array}{l|l}
post(s_0) = \{s_1\} & sim(s_0) = \{s'_0\} \\
post(s_1) = \{s_2, s_3\} & sim(s_1) = \{s'_1, s'_2, s'_5\} \\
post(s_2) = \{s_0, s_4\} & sim(s_2) = \{s'_3, s'_4\} \\
post(s_3) = \{s_4\} & sim(s_3) = \{s'_6\} \\
post(s_4) = \emptyset & sim(s_4) = \{s'_7\} \\
post(s'_0) = \{s'_0, s'_1, s'_2, s'_3, s'_4, s'_5, s'_6, s'_7\} & sim(s'_0) = \emptyset \\
post(s'_1) = \{s'_0, s'_1, s'_2, s'_3, s'_4, s'_5, s'_6, s'_7\} & sim(s'_1) = \emptyset \\
post(s'_2) = \{s'_0, s'_1, s'_2, s'_3, s'_4, s'_5, s'_6, s'_7\} & sim(s'_2) = \emptyset \\
post(s'_3) = \{s'_0, s'_1, s'_2, s'_3, s'_4, s'_5, s'_6, s'_7\} & sim(s'_3) = \emptyset \\
post(s'_4) = \{s'_0, s'_1, s'_2, s'_3, s'_4, s'_5, s'_6, s'_7\} & sim(s'_4) = \emptyset \\
post(s'_5) = \{s'_6, s'_7\} & sim(s'_5) = \emptyset \\
post(s'_6) = \{s'_7\} & sim(s'_6) = \emptyset \\
post(s'_7) = \emptyset & sim(s'_7) = \emptyset
\end{array}$$

**Running time analysis.** The running time of the algorithm depends on the size of the input graph transition system  $TS$ . Let  $n$  be its number of states and  $m$  be its number of transitions. Then, the efficient implementation of the algorithm presented in [66] computes the greatest fixpoint in  $O(mn)$  time. We only have to consider the time of the modified initialization part.

Obviously, the running time of the modified initialization part (Fig. 7.6 ll. 1-2) depends on the time  $T_{Refines}$  of the **REFINES** function. As this function is computed for every pair of states, the initialization consumes  $O(n^2 \cdot T_{refines})$  time in total. A precise statement about  $T_{refines}$  is not possible, because we have deliberately left open how the underlying structural refinement relation is defined (see Section 7.2). We can only say that  $T_{Refines}$  will depend on the size of the individual instance graphs associated with the states. If this size can be limited by a constant upper bound, say  $K$ , then this factor vanishes from the  $O$ -calculus expression and we obtain  $O(n^2)$ , or  $O(mn)$  assuming that  $n \leq m$ . The upper bound  $K$  could be ensured, e.g., by stopping the exploration of the graph transition system where states are reached with size  $> K$ . The same way, an infinite transition system can be approximated by a finite one.

Eventually, we have to consider the preparation time of the joint input transition system. As the input transition system is naturally greater than either constituent transition system, we can state  $O(mn)$  as an upper bound for the construction time of the transitive closure of  $GTS(M')$  in terms of  $m$  and  $n$ , too [28].

Altogether, we achieve an  $O(mn)$  algorithm for checking behavior preservation. As we can assume that  $n \leq m \leq n^2$ , we obtain  $O(n^3)$  as an upper

bound in terms of the number of abstract and concrete states. Although this means that the algorithm is efficient from a theoretical point of view, we have to be aware of the state explosion problem [26]: As combinatorial combinations of local states of all concurrent threads have to be considered, the state space grows exponentially with the introduction of new threads.

Needless to say, the state explosion problem also affects the time required to check behavior preservation. Thus, we might run into performance problems when architecture models become too large. Some concrete figures about practical experience are given in Section 8.3.

### 7.3.2 Observational substitutability

Observational substitutability (21) is dual to behavior preservation, because we have to check if every behavior of the concrete architecture can also occur in the abstract model. In other words, nothing shall be allowed at the concrete level which is not contained in abstract behavior, except for solely platform-specific operations which do not affect business-relevant entities. This condition ensures that all safety properties satisfied at the abstract level are preserved during the refinement process. Hence, somebody observing the system from outside will not recognize that the abstract architecture is substituted by the concrete one.

Conversely to behavior preservation, every concrete scenario of communication and reconfiguration operations requires an equivalent scenario in the abstract architecture, and every state reachable in the concrete system requires the reachability of an equivalent state (in terms of structural refinement) in the business-oriented architecture.

In our semantic domain, this means that every transformation step  $G' \Rightarrow_{\mathcal{G}'} H'$  in the concrete style  $\mathcal{G}'$  has to be mapped to a suitable transformation step  $G \Rightarrow_{\mathcal{G}} H$  in the abstract style  $\mathcal{G}$ . However, in order to allow to refine abstract operations by multiple steps at the platform-specific level, we have to relax the above condition and include sequences of platform-specific steps as substitutions for single abstract steps. The first state and all intermediate states of the concrete sequence have to be structural refinements of the abstract source state, and only the last state of the sequence refines the abstract target state.

*Example 7.12.* Reconsider the creation of a new connection as sketched in Fig. 7.7. The service discovery actions at the lower level are solely platform-specific and have no effect on business-relevant aspects. Thus, all the intermediate states refine the abstract source state  $G$ , until the connector is actually created by the last action.



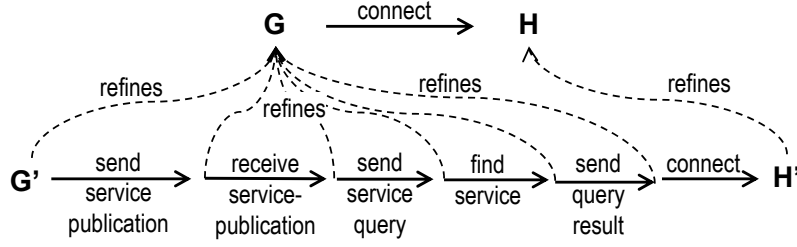


Figure 7.7: Substitution of an abstract transformation step

Extending the above sketched principle to all transformation steps and states reachable in the concrete architecture results in the following definition of observational substitutability:

**Definition 7.13 (Observational Substitutability).** *Let  $M = (\mathcal{G}, G_0)$  be a graph transformation model representing an architecture in an abstract architectural style  $\mathcal{G} = (GS, P)$  with transformation sequences  $Op(M)$ . Let  $M' = (\mathcal{G}', G'_0)$  be a graph transformation model representing an architecture in a concrete architectural style  $\mathcal{G}' = (GS', P')$  with transformation sequences  $Op(M')$ . Given a structural refinement relation  $refines \subseteq Inst(GS') \times Inst(GS)$  according to Definition 7.4, then  $M'$  observationally substitutes  $M$  ( $M'$  substitutes  $M$  for short), iff*

$$\begin{aligned} &\forall p' \in Op(M') \text{ with } p' = (G'_0 \Rightarrow_{\mathcal{G}'} G'_1 \dots \Rightarrow_{\mathcal{G}'} G'_n) \\ &\exists p \in Op(M) \text{ with } p = (G_0 \Rightarrow_{\mathcal{G}} G_1 \dots \Rightarrow_{\mathcal{G}} G_m), \ m \leq n \\ &\text{and a surjective, monotonic function } f : \{0 \dots n\} \rightarrow \{0 \dots m\} \\ &\text{such that } \forall i = 0 \dots n : G'_i \text{ refines } G_{f(i)} \end{aligned}$$

The function  $f$  assigns concrete states to the abstract states they refine. As  $m \leq n$ , the function is in general not injective, i.e., several concrete states may refine the same abstract state. On the opposite,  $f$  has to be surjective because the concrete scenario must not skip any business-relevant state of the abstract scenario it substitutes. Eventually, the assignment has to preserve the order of states, which is guaranteed by the monotonicity of  $f$ , i.e.,  $f(i) \leq f(i+1) \ \forall i \in \{0 \dots n-1\}$ .

Similar to behavior preservation in Section 7.3.1, the above definition of observational substitutability can only be computed if it is transferred to finite graph transition systems. The corresponding trace inclusion problem can again be approximated by a co-inductively defined simulation relation between the states of two graph transition systems. This time, however, the direction of the relation is inverted as follows:

**Definition 7.14 (Simulation Relation  $\text{sim}R$ ).** Given a structural refinement relation  $\text{refines}$  and two graph transition systems  $GTS = (L, S, \Rightarrow, s_0)$  and  $GTS' = (L', S', \Rightarrow, s'_0)$ . Then, a relation  $\text{sim}R \subseteq S \times S'$  is a simulation relation between  $GTS$  and  $GTS'$ , if  $(s, s') \in \text{sim}R$  implies

- $s'$  refines  $s$
- $\forall \hat{s}' \in S'$  with  $s' \Rightarrow \hat{s}' : (s, \hat{s}') \in \text{sim}R \quad \vee$   
 $\exists \hat{s} \in S$  with  $s \Rightarrow \hat{s}$  such that  $(\hat{s}, \hat{s}') \in \text{sim}R$

*Example 7.15.* To illustrate this definition, remember the two graph transition systems in Fig. 7.5. Valid simulation relations in the sense of the preceding definition for these two transition systems are:

$$\begin{aligned} \text{sim}R_1 &= \{(s'_7, s_4)\} \\ \text{sim}R_2 &= \{(s'_6, s_3), (s'_7, s_4)\} \\ \text{sim}R_3 &= \{(s'_5, s_1), (s'_6, s_3), (s'_7, s_4)\} \\ \text{sim}R_4 &= \{(s'_0, s_0), (s'_1, s_1), (s'_5, s_1), (s'_2, s_1), (s'_3, s_2), (s'_4, s_2), (s'_6, s_3), (s'_7, s_4)\} \end{aligned}$$

The following proposition shows how the existence of such a simulation relation  $\text{sim}R$  entails observational substitutability.

**Proposition 7.16 (Observational Substitutability in Graph Transition Systems).** Let  $M = (\mathcal{G}, G_0)$  be a model over an abstract architectural style  $\mathcal{G} = (GS, P)$  and  $M' = (\mathcal{G}', G'_0)$  be a model over a concrete architectural style  $\mathcal{G}' = (GS', P')$ . Moreover, let  $GTS(M')$  and  $GTS(M)$  be the graph transition systems they induce and  $\text{refines} \subseteq \text{Inst}(GS') \times \text{Inst}(GS)$  a structural refinement relation. Then,  $M'$  substitutes  $M$ , if there exists a simulation relation  $\text{sim}R$  between  $GTS(M)$  and  $GTS(M')$  such that  $([G_0], [G'_0]) \in \text{sim}R$ .

*Proof.* The proof works analogously to that of Proposition 7.10. Let  $\text{sim}R$  be the simulation relation between  $GTS(M)$  and  $GTS(M')$  such that  $([G_0], [G'_0]) \in \text{sim}R$ . We show by induction over the path length  $n \in \mathbb{N}_0$  that this entails:

$$\begin{aligned} &\forall p' \in \text{Op}(M') \text{ with } p' = (G'_0 \Rightarrow_{\mathcal{G}'} G'_1 \Rightarrow_{\mathcal{G}'} \dots \Rightarrow_{\mathcal{G}'} G'_n) \\ &\exists p \in \text{Op}(M) \text{ with } p = (G_0 \Rightarrow_{\mathcal{G}} G_1 \Rightarrow_{\mathcal{G}} \dots \Rightarrow_{\mathcal{G}} G_m), \quad m \leq n \\ &\text{and a surjective, monotonic function } f : \{0 \dots n\} \rightarrow \{0 \dots m\} \\ &\text{such that } \forall i = 0 \dots n : ([G_{f(i)}], [G'_i]) \in \text{sim}R \end{aligned}$$

- $n = 0$ :

$$p'_0 = (G'_0) \text{ and } p_0 = (G_0) \text{ and } f_0 : 0 \mapsto 0 \text{ and } ([G_0], [G'_0]) \in \text{sim}R$$

- $n \rightarrow n + 1$ :

$$\begin{aligned} & \text{Let } p'_{n+1} = (G'_0 \Rightarrow_{\mathcal{G}'} G'_1 \Rightarrow \dots \Rightarrow_{\mathcal{G}'} G'_n \Rightarrow_{\mathcal{G}'} G'_{n+1}) \\ \xRightarrow{\text{Ind.}} & \exists p_m = (G_0 \Rightarrow_{\mathcal{G}} G_1 \Rightarrow \dots \Rightarrow_{\mathcal{G}} G_m), \ m \leq n \\ & \text{and a surjective, monotonic function } f_n : \{0 \dots n\} \rightarrow \{0 \dots m\} \\ & \text{such that } \forall i = 0 \dots n : ([G_{f_n(i)}], [G'_i]) \in \text{sim}R \end{aligned}$$

As  $f$  is surjective and monotonic, we achieve  $f_n : n \mapsto m$  and, thus,  $([G_m], [G'_n]) \in \text{sim}R$ . According to Definition 7.14, the transition  $[G'_n] \Rightarrow [G'_{n+1}]$  entails one of the following two cases:

1.  $([G_m], [G'_{n+1}]) \in \text{sim}R$

$$\text{Then we set } f_{n+1}(i) := \begin{cases} f_n(i) & \text{if } i \leq n \\ m & \text{if } i = n + 1 \end{cases}$$

$f_{n+1} : \{0 \dots n + 1\} \rightarrow \{0 \dots m\}$  remains surjective and monotonic, and  $m \leq n + 1$  holds.

2.  $\exists [G_{m+1}]$  with  $[G_m] \Rightarrow [G_{m+1}]$  such that  $([G_{m+1}], [G'_{n+1}]) \in \text{sim}R$

$$\xRightarrow{4.21} \exists p_{m+1} = (G_0 \Rightarrow_{\mathcal{G}} G_1 \Rightarrow \dots \Rightarrow_{\mathcal{G}} G_m \Rightarrow_{\mathcal{G}} G_{m+1})$$

$$\text{and we set } f_{n+1}(i) := \begin{cases} f_n(i) & \text{if } i \leq n \\ m + 1 & \text{if } i = n + 1 \end{cases}$$

$f_{n+1} : \{0 \dots n + 1\} \rightarrow \{0 \dots m + 1\}$  remains surjective and monotonic, and  $m + 1 \leq n + 1$  holds.

In both cases, it follows:  $\forall i = 0 \dots n + 1 : ([G_{f_{n+1}(i)}], [G'_i]) \in \text{sim}R$ .

Altogether, we obtain:

$$\begin{aligned} & \forall p' \in \text{Op}(M') \text{ with } p' = (G'_0 \Rightarrow_{\mathcal{G}'} G'_1 \Rightarrow_{\mathcal{G}'} \dots \Rightarrow_{\mathcal{G}'} G'_n) \\ & \exists p \in \text{Op}(M) \text{ with } p = (G_0 \Rightarrow_{\mathcal{G}} G_1 \Rightarrow_{\mathcal{G}} \dots \Rightarrow_{\mathcal{G}} G_m), \ m \leq n \\ & \exists \text{ a surjective, monotonic function } f : \{0 \dots n\} \rightarrow \{0 \dots m\} \\ & \text{such that } \forall i = 0 \dots n : ([G_{f(i)}], [G'_i]) \in \text{sim}R \\ \xRightarrow{7.14} & \forall i = 0 \dots n : [G'_i] \text{ refines } [G_{f(i)}] \\ \xRightarrow{7.5} & \forall i = 0 \dots n : G'_i \text{ refines } G_{f(i)} \\ \xRightarrow{7.7} & M' \text{ substitutes } M \end{aligned}$$

□

**Computing observational substitutability.** For computing the greatest substitutability relation  $simR$ , we will again adopt Henzinger et al.'s algorithm [66]. However, in terms of the original notion of simulation the algorithm is based on, we now have to decide whether the abstract start state  $s_0$  simulates the concrete start state  $s'_0$ . For this purpose, we have to check if  $s_0$  is in the simulator set of  $s'_0$ , i. e.,  $s_0 \in sim(s'_0)$ .

Due to the inverted simulation direction, we also have to change the initialization part of the algorithm because the underlying check for structural refinement is not symmetric. Otherwise, the simulator sets of concrete states  $s' \in S'$  would never contain any abstract state  $s \in S$ . But, if we replace  $REFINES(s', s)$  by  $REFINES(s, s')$  in line 2 of algorithm PRESERVES in Fig. 7.6, then we can derive from equations (7.1) and (7.2) that the algorithm now computes the greatest simulation relation  $\leq$  such that  $s \leq s'$  implies

$$s' \text{ refines } s \quad (7.3)$$

$$\forall s' \in post(s') \exists \hat{s} \in post(s) \text{ such that } \hat{s} \leq s' \quad (7.4)$$

The modified algorithm is shown in Fig. 7.8. If we directly applied it to the disjoint union of the abstract and concrete transition systems, then the outcome TRUE ( $s_0 \in sim(s'_0)$ ) would mean that for every concrete path starting in  $s'_0$  there is a corresponding abstract path starting in  $s_0$  of exactly the same length! However, our intention was to also allow for shorter abstract paths which subsume the additional platform-specific actions.

SUBSTITUTES ( $TS$ ) : *Boolean*

```

1  for all  $s \in S$ 
2       $sim(s) := \{s' \in S \mid REFINES(s, s') = \text{TRUE}\}$ 
3
4  while there are three states  $s, \hat{s}, s' \in S$  such that
5       $\hat{s} \in post(s)$ ,  $s' \in sim(s)$ , and  $post(s') \cap sim(\hat{s}) = \emptyset$  do
6       $sim(s) := sim(s) \setminus \{s'\}$ 
7  end while
8
9  if  $s_0 \in sim(s'_0)$  then
10     return TRUE
11 else
12     return FALSE
```

Figure 7.8: Algorithm for observational substitutability based on [66]

Similar to the transitive closure trick applied in Section 7.3.1, we solve this discrepancy without modifying the algorithm but by a modification of the input transition system. In this case, we take the abstract transition system  $GTS(M)$  and insert a loop transition  $s \Rightarrow s$  for all states  $s \in S$ . It obviously holds that for every transition  $s_1 \Rightarrow s_2$  in the extended transition system  $GTS(M)^\infty$  there is either the same transition in  $GTS(M)$  or  $s_1 = s_2$ .

If we apply the modified algorithm to the disjoint union of the concrete transition system  $GTS(M')$  and the extended abstract transition system  $GTS(M)^\infty$ , then an outcome  $s_0 \in \text{sim}(s'_0)$  means that  $M'$  substitutes  $M$  according to Proposition 7.16. We can argue that, starting from  $s_0$ , one can only traverse transitions which belong to  $GTS(M)^\infty$ . Each of these transitions represents a transition or loop in the original transition system  $GTS(M)$ . Thus, if the outcome of the algorithm satisfies equation (7.4) with respect to  $GTS(M)^\infty$ , then it satisfies the corresponding equation of Definition 7.14 with respect to  $GTS(M)$ .

*Example 7.17.* Recall the two transition systems  $GTS(M)$  and  $GTS(M')$  in Fig. 7.5. In order to compute the greatest simulation relation  $\text{sim}R$  between them, the algorithm is fed with the joint transition system consisting of  $GTS(M')$  and the extended transition system  $GTS(M)^\infty$  as denoted by the adjacent lists in the left column below. The right column denotes how the algorithm *initializes* the simulator sets according to the inverted structural refinement relationship.

$\text{post}(s_0) = \{s_0, s_1\}$	$\text{sim}(s_0) = \emptyset$
$\text{post}(s_1) = \{s_1, s_2, s_3\}$	$\text{sim}(s_1) = \emptyset$
$\text{post}(s_2) = \{s_0, s_2, s_4\}$	$\text{sim}(s_2) = \emptyset$
$\text{post}(s_3) = \{s_3, s_4\}$	$\text{sim}(s_3) = \emptyset$
$\text{post}(s_4) = \{s_4\}$	$\text{sim}(s_4) = \emptyset$
$\text{post}(s'_0) = \{s'_1\}$	$\text{sim}(s'_0) = \{s_0\}$
$\text{post}(s'_1) = \{s'_2, s'_5\}$	$\text{sim}(s'_1) = \{s_1\}$
$\text{post}(s'_2) = \{s'_3, s'_4\}$	$\text{sim}(s'_2) = \{s_1\}$
$\text{post}(s'_3) = \{s'_4, s'_7\}$	$\text{sim}(s'_3) = \{s_2\}$
$\text{post}(s'_4) = \{s'_0, s'_7\}$	$\text{sim}(s'_4) = \{s_2\}$
$\text{post}(s'_5) = \{s'_6\}$	$\text{sim}(s'_5) = \{s_1\}$
$\text{post}(s'_6) = \{s'_7\}$	$\text{sim}(s'_6) = \{s_3\}$
$\text{post}(s'_7) = \emptyset$	$\text{sim}(s'_7) = \{s_4\}$

In this example, the initial simulation relation does not have to be sharpened any more, the while loop of the algorithm (Fig. 7.8 ll. 4-7) is not entered, and the simulator sets represent the greatest simulation relation which we

had already constructed as  $\text{sim}R_4$  in Example 7.15. With  $s_0 \in \text{sim}(s'_0)$ , we have shown that  $M'$  substitutes  $M$  according to Proposition 7.16.

Note that this example also reveals the necessity of the loop transitions. If we had not inserted them,  $\text{post}(s_1)$  would not contain  $s_1$  and the algorithm would enter the while loop with  $s = s'_1$ ,  $\hat{s} = s'_5$ , and  $s' = s_1$  because of  $\text{post}(s_1) \cap \text{sim}(s'_5) = \emptyset$ . Then,  $s_1$  would be removed from  $\text{sim}(s'_1)$  and, in a second iteration, also  $s_0$  from  $\text{sim}(s'_0)$ . Thus, the algorithm would return a wrong result.

The computational complexity of the algorithm remains the same as for behavior preservation. Altogether, we achieve an  $O(mn)$  algorithm for checking observational substitutability.

## 7.4 Testing behavioral refinement

In this section, we show how behavioral refinement can also be *tested* in cases where complete refinement checks become too time-consuming due to the state explosion problem. As usual, such tests can only show the correctness for the test cases but never substitute a formal verification like that described in the previous section.

In principle, both testing behavior preservation and observational substitutability means to test trace inclusion between an abstract and a concrete transition system, apart from different directions. For this purpose, we have to select a limited number of traces from one transition system as *test cases* [62] and check if these traces are also contained in the other transition system. The following part explains how to automate these checks with the help of suitable analysis tools like model checkers.

### 7.4.1 Testing behavior preservation

The rationale behind behavior preservation was to guarantee that no business scenario or other possible behavior of an abstract architecture is lost during the refinement to a more platform-specific level. Let  $GTS$  be the graph transition system induced by the abstract architecture model, and  $GTS'$  be the concrete, platform-specific graph transition system. Then, a behavior preservation test has to check if for a given path  $p = (G_0 \Rightarrow G_1 \dots \Rightarrow G_n)$  in  $GTS$ , there is a corresponding path  $p' = (G'_0 \xRightarrow{*} G'_1 \dots \xRightarrow{*} G'_n)$  in  $GTS'$  such that  $G'_i$  refines  $G_i$  for all  $i = 0 \dots n$  (cf. Definition 7.7).

The difficulty of such a test is that we cannot simply explore the concrete transition system and look for a path  $p'$  containing all  $G'_i$ , because we do not know in advance how the  $G'_i$  actually look like. However, if we were able

to formulate *predicates*  $P_{G_i}$  such that  $G'_i$  satisfying  $P_{G_i}$  ( $G'_i \models P_{G_i}$ ) entails  $G'_i$  *refines*  $G_i$ , then we could perform the test by searching the concrete transition system for a path which satisfies all  $P_{G_i}$ .

**Definition 7.18 (Refinement Predicate).** *Given an abstract architectural style  $\mathcal{G}$  with graph schema  $GS$ , a concrete architectural style  $\mathcal{G}'$  with graph schema  $GS'$ , and a structural refinement relation  $\text{refines} \subseteq \text{Inst}(GS') \times \text{Inst}(GS)$ , then a predicate  $P_G$  is called *refinement predicate* of  $G \in \text{Inst}(GS)$  if for all  $G' \in \text{Inst}(GS')$ :*

$$G' \models P_G \implies G' \text{ refines } G$$

In some sense, refinement predicates invert the underlying abstraction function of the structural refinement relation, because they describe under which conditions a concrete graph structurally refines a given abstract graph. In the definition above, we have deliberately left open in which language the predicates are expressed and by which reasoning logic the satisfaction relation  $\models$  is decided. Both issues depend on the way the abstraction function is defined and on the analysis tool which has to perform the tests and to interpret the predicates.

For example, if the abstraction function is based on a straight-forward type graph mapping as proposed in Section 7.2.1, then the refinement predicate has to invert that mapping, enumerate all valid refinements under the concrete typing, and demand that the concrete graph comprises one of these refinements.

In the following paragraphs, we suggest two different ways in which the concrete transition system can be searched for a trace which satisfies the refinement predicates  $P_{G_0} \dots P_{G_n}$ .

**Model checking with LTL formulae.** If the concrete graph transition system can be encoded into the format of an off-the-shelf model checker like *SPIN* [74], then we can describe the desired path refinement of  $p$  with the help of temporal logics and employ the model checker for searching  $p'$ . For instance, let  $p = (G_0 \Rightarrow G_1 \dots \Rightarrow G_n)$  be expressed by a sequence of refinement predicates  $P_{G_0}, P_{G_1}, \dots, P_{G_n}$ . Then we can formulate the refinement of  $p$  by a formula in *linear-time temporal logic* (LTL), which reads as follows:<sup>4</sup>

$$P_{G_0} \wedge \Diamond(P_{G_1} \wedge \Diamond(P_{G_2} \wedge \dots \Diamond(P_{G_n}) \dots))$$

Since we require only one path to satisfy the above formula, while an LTL formula always refers to all paths of the transition system, we have to negate

---

<sup>4</sup>The LTL operator  $\Diamond$  means “at some time in the future”

the above formula and let the model checker look for a *counter example*. A counter example that violates the negated formula serves as a witness for the original formula [63].

The advantage of this proposal is that we do not have to change anything of the graph transition system and can directly apply standard model checking tools which incorporate several sophisticated strategies and heuristics to optimize the evaluation of temporal logic formulae. On the other hand, it is difficult to find a model checker which can operate on graph transition systems with graphs as states and evaluate the refinement predicates  $P_{G_0}, P_{G_1}, \dots, P_{G_n}$  over these graphs.

Varró et al. [137] work on filling this gap by their translation tool *Check-VML*. It takes a graph transformation model and translates it, together with the LTL property to be verified, into the input language of the model checker SPIN, which is *Promela*. More about this approach is revealed in Section 8.1.5.

**Reachability analysis.** Testing trace inclusion can also be realized without full support of a temporal logic like LTL: Our second testing strategy only requires a tool which supports a simple kind of *reachability analysis*, namely the generation and exploration of a graph transition system until a state is reached in which a certain graph transformation rule becomes applicable.

Such a feature can be exploited in order to find some state  $G'$  in the platform-specific architecture model which satisfies a given refinement predicate  $P_G$ . For this purpose, we rewrite the refinement predicate as a graph transformation rule  $r(P_G)$  whose left-hand side contains positive and negative application conditions which represent the requirements that  $G'$  must satisfy in order to structurally refine  $G$ . The right-hand side of  $r(P_G)$  is identical to its left-hand side, i. e., the rule does not modify the host graph.

If we add the new *predicate rule* to the graph transformation system and restart the generation of the corresponding transition system, then the required state  $G'$  is reached as soon as  $r(P_G)$  becomes applicable.

Extending this idea from single states to paths, we can also check if there is a path  $p' = (G'_0 \xRightarrow{*} G'_1 \dots \xRightarrow{*} G'_n)$  containing states  $G'_i$  which satisfy  $P_{G_i}$  for all  $i = 0 \dots n$  as follows:

- For  $i = 0$ , we can explicitly check whether the concrete start state  $G'_0$  satisfies predicate  $P_{G_0}$ , i. e., whether  $G'_0$  *refines*  $G_0$ .
- For every other  $P_{G_i}$ , we create a new predicate rule  $r(P_{G_i})$  which is added to the graph transformation system. Thus, when generating the graph transition system, the application of these rules are mixed with



ordinary rule applications. If we can enforce that the predicate rules are applied in ascending order only, i. e.,  $r(P_{G_i})$  some time after  $r(P_{G_{i-1}})$ , then a path  $p'$  exists iff  $r(P_{G_n})$  can be applied at least once (see the bold path in Fig. 7.9).

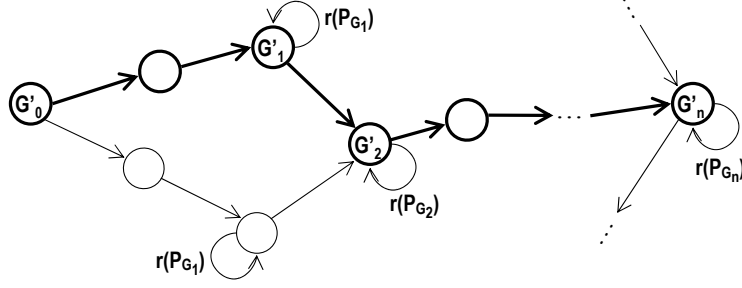


Figure 7.9: Reachability analysis with special predicate rules

In order to enforce an ascending application order of the predicate rules, we insert a separate *counter* node into the initial instance graph  $G'_0$  which represents the history of already applied predicate rules. For this purpose, the counter node gets a numerical attribute with is initially set to 1. While the counter is completely ignored by ordinary graph transformation rules, the predicate rules get an additional application condition such that rule  $r(P_{G_i})$  can only be applied if the value of the counter attribute equals  $i$ . Moreover, the right-hand side of a predicate rule  $r(P_{G_i})$  is extended such that it increases the value of the counter from  $i$  to  $i + 1$ . This effect enables the application of the next predicate rule  $r(P_{G_{i+1}})$  as soon as its other application conditions related to the refinement predicate are fulfilled, too.

In comparison to the first testing strategy, this reachability analysis requires less capabilities of the applied analysis tool. In Section 8.1.6, we will introduce Rensink's graph transformation tool *GROOVE* [130] which supports state space generation and such a reachability analysis for graph transition systems.

And, although some extra rules have to be added to the graph transformation system, these extensions remain local and do not affect the existing transformation rules. However, due to the newly introduced counter node and its different values representing the history of already applied predicate rules, the growing size of the state space to be explored might neutralize the advantages of this testing strategy. Fortunately, this disadvantage does not occur when testing observational substitutability as described in the next section.

### 7.4.2 Testing observational substitutability

Testing observational substitutability works almost symmetrically to testing behavior preservation, because we now have to check whether a given platform-specific path  $p' = (G'_0 \Rightarrow G'_1 \dots \Rightarrow G'_n)$  is included – under the necessary abstractions – in the potential behavior of the more abstract, business-oriented architecture model. However, there are three important differences:

1. On the one hand, preparing test cases for observational substitutability is easier because, when expressing the concrete path in terms of *abstraction predicates* over the abstract architectural style, we can directly apply the available abstraction function *abs* (cf. Section 7.2.1).
2. On the other hand, our definition of observational substitutability (cf. Definition 7.13) restricts the allowed trace inclusions in such a way that a suitable abstraction  $p = (G_0 \Rightarrow G_1 \dots \Rightarrow G_m)$  must not contain any intermediate state which does not correspond to one of the concrete states (the function  $f$  assigning concrete to abstract states has to be surjective). For this reason, we cannot simply apply the reachability analysis proposed for behavior preservation tests in the previous section, but we have to adapt that proposal to the characteristics of observational substitutability first.
3. Due to the requirement that the desired abstract sequence  $p = (G_0 \Rightarrow G_1 \dots \Rightarrow G_m)$  has to be dense, i.e., without any intermediate states and  $m \leq n$ , we can restrict a reachability analysis starting in  $G_0$  to paths of length  $n$ .

Dual to refinement predicates, we define abstraction predicate as follows:

**Definition 7.19 (Abstraction Predicate).** *Given an abstract architectural style  $\mathcal{G}$  with graph schema  $GS$ , a concrete architectural style  $\mathcal{G}'$  with graph schema  $GS'$ , and a structural refinement relation  $\text{refines} \subseteq \text{Inst}(GS') \times \text{Inst}(GS)$ , then a predicate  $P_{G'}$  is called abstraction predicate of  $G' \in \text{Inst}(GS')$  if for all  $G \in \text{Inst}(GS)$ :*

$$G \models P_{G'} \implies G' \text{ refines } G$$

According to the first issue mentioned above, we can state as abstraction predicate  $P_{G'}$  that an abstract graph  $G$  satisfies the predicate if its structural part equals to  $\text{abs}(G')$  and, consequently,  $G' \text{ refines } G$  (cf. Definition 7.4). As all  $G'_i$  of the concrete path  $p'$  are given, we can again create special predicate rules  $r(P_{G'_i})$ . This time, their positive and negative application conditions

require that the host graph contains  $abs(G'_i)$  and no other structure-related elements. Then, an abstract state  $G_i$  in which  $r(P_{G'_i})$  becomes applicable is structurally refined by  $G'_i$ .

Analogously to behavior preservation tests, we have to enforce that the new predicate rules can only be applied in the predefined order. Thus, we apply the “counter trick” from the previous section again. Accordingly, a path  $p$  containing abstractions of all  $G'_i$  is reached as soon as  $r(P_{G'_n})$  becomes applicable.

However, the counter does not prevent that the resulting path  $p$  contains any intermediate states which are not related to the states of the given concrete path  $p'$  (see the second issue mentioned above). To satisfy this requirement of observational substitutability, we have to ensure that at least one of the predicate rules is applied after every “ordinary” rule application. In other words, not more than one ordinary transition rule must be applied between two predicate rules.

We can guarantee this condition by adding a new boolean attribute **ordinaryEnabled** to the counter node. Predicate rules always set the value of this flag to **TRUE**, and ordinary transformation rules of the architectural style are extended such that they require a positive value before application and change the value to **FALSE** after completion. This way, a subsequent transformation rule can only be applied after another predicate rule has set the value to **TRUE** again.

In comparison to behavior preservation tests, the reachability analysis for observational substitutability requires some more rule extensions for handling the **ordinaryEnabled** flag. On the other hand, using this flag as described above dramatically reduces the state space to be explored during the reachability analysis.

## 7.5 Summary

In this chapter, we have introduced a new, semantical notion of refinement for graph transformation models. Graph transition systems with instance graphs as states have been used as semantic domain. Since the approach does not require any fixed mapping between the transformation rules, it remains applicable even if abstract and concrete transformation rules are very different. Instead, we rather base the refinement criteria for behavior preservation and observational substitutability on a comparison of the different states. This way, the approach becomes context-dependent and supports changing refinements of the same abstract operation.

We have borrowed concepts from the domain of concurrent system verifi-

cation, which provides a broad range of different refinement and simulation relations for comparing state transition systems. The existent approaches mostly define state-based simulation relations by simple mappings between available state labels. However, as our states reflect complete instance graphs, we apply a more complex matching of abstract and concrete states based on structural refinement.

Simulation relations differ in whether they distinguish *internal*, invisible actions from externally observable ones or not (*weak* versus *strong* simulations). In state-based approaches, the equivalent to internal actions are *stuttering steps* whose source and target states are equal from an external point of view. In [1] for instance, a stuttering step means a state change of an internal component only. The research literature contains a number of different proposals on how to handle such stuttering steps. In our case, stuttering steps correspond to purely platform-specific steps at the concrete level. They do not change the business-relevant parts of the system and are, thus, stuttering steps from the business level point of view. Our simulation relations reflect this fact by relating several consecutive platform-specific states to a single abstract state if necessary (cf. Fig. 7.7).

Due to the semantical state space analysis, we can check behavioral refinement entirely without syntactically matching the graph transformation rules of the underlying abstract and concrete architectural styles. This is different to existing syntactical refinement proposals for graph transformation systems which all require such a rule matching [58, 55].

On the other hand, the semantical refinement criteria cause additional complexity because they cannot simply be decided by looking at the model specification but require a simulation of the system in terms of exploring the corresponding graph transition system. Though we have introduced an  $O(mn)$  time algorithm for checking behavior preservation and observational substitutability, i. e., one that is efficient at least from a theoretical point of view, the state explosion problem might hinder a full analysis in practice. For this reason, we also spent a section on testing the refinement properties.

Another weakness of the proposed algorithms is their restriction to a yes/no answer. In the case of a positive outcome everything is all right and the mutual consistency of the two models has been verified. But, in the negative case, the application architect does not get any information about the cause of the failure. Provided that the two architectural styles and the corresponding transformation rules have been correctly specified, then there are two potential obstacles to a valid refinement: It could either be that the capabilities of the two architectural styles are so different that the abstract model cannot be realized on the chosen target platform, or the platform-specific architecture encoded into the concrete start graph is not consistent with the

abstract one. Future work should aim at an extension of the approach which helps the architect to detect the actual cause for a failed refinement check.

While the presented refinement concepts are intended to support the development of dynamic software architectures, they can very likely be generalized to many other application domains of graph transformation models, too.

So far, we have shown how to model dynamic software architectures in platform-specific architectural styles and how to check if a model at a lower level of platform abstraction is a valid refinement of one in a more abstract style. We consider these to be the two main ingredients for a platform-consistent and step-wise development of dynamic architectures. Next, we will complete the proposal by discussing suitable tool support.



## Chapter 8

# Existing tool support for modeling and refinement

In order to enable the practical application of the concepts presented so far, software architects need support by adequate *CASE tools* (short for “Computer-Aided Software Engineering”). In this chapter, we will discuss to which extent this can be accomplished by existing tools and how these tools have to be integrated.

Figure 8.1 summarizes the various activities of style and application architects in a UML activity diagram and identifies the kind of tool which is required for each of the tasks. Note that the depicted process does not describe a complete software development process but only the activities which belong to the platform-consistent architecture development as proposed in the preceding chapters.

The left swim lane of the activity diagram comprises the tasks the style architect is responsible for. At first, he requires appropriate tools to define the graph transformation system and the UML profile of a platform-independent architectural style, as well as a translation relation for conversions between the two model representations (cf. Chapters 5 and 6). Then he repeats the same procedure for the platform-specific architecture. Finally, he defines an abstraction function which forms the basis of future refinement checks between instances of the two styles (cf. Section 7.2.1).

After the style architect has provided the style definitions, the application architect can design the business-level and platform-specific architectures as instances of the styles using some graphical UML editor. Before he can check the mutual consistency of these models, they need to be translated into instance graphs of our semantic domain, which should at best be done by some automatic translator in the background.

Having translated the two architecture models into the graph-based se-

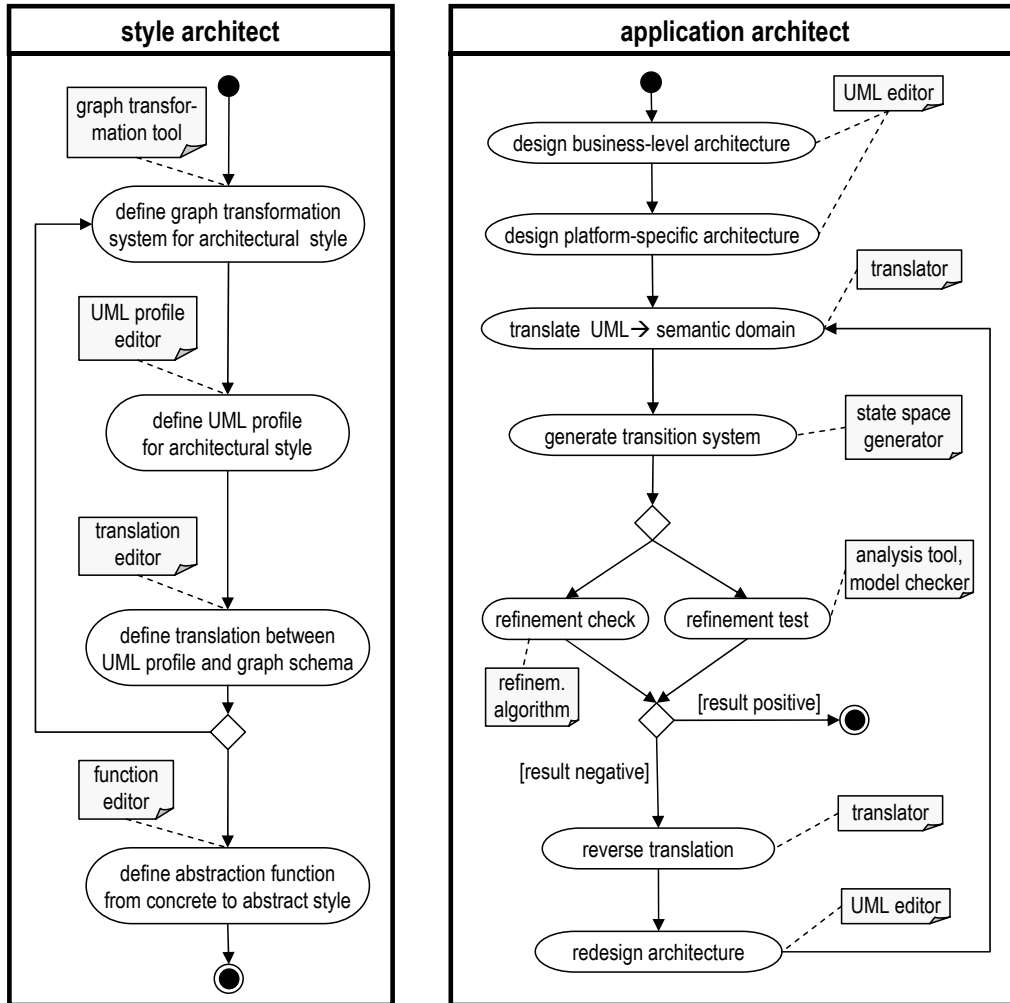


Figure 8.1: Activities of style and application architects and required tools

semantic domain, the application architect requires a state space generator in order to derive the transition systems induced by the two graph transformation models (cf. Section 4.3). Based on these transition systems, he can then either check or test the mutual consistency of the two models according to the refinement criteria defined in Chapter 7. If the outcome is negative, then the architect might apply a reverse translator to translate some crucial states of the transition system back into their UML representation before he starts to redesign the architectures.

Due to the high effort required to develop a new, comprehensive CASE tool for all these tasks, we rather favor to survey available specialized tools



and how they could be extended and combined to an integrated tool environment for our approach. In this context, graph transformation tools play a key role because graphs and graph transformation systems form the conceptual foundation of our modeling and refinement approach. Thus, Section 8.1 is especially dedicated to existent graph transformation tools, the requirements we have, and the features they provide so far.

Section 8.2 briefly touches on UML tools and basic concepts for tool integration. Section 8.3 presents a practical example using one of the most promising tools for a refinement test of our travel agency architecture. Eventually, Section 8.4 concludes the chapter.

## 8.1 Graph transformation tools

There is a limited number of CASE tools for graph transformation-based modeling. In order to judge their suitability for the formal architecture modeling and refinement method presented in this thesis, we at first want to collect a list of required features.

### 8.1.1 Required features

1. Most basic of all, every graph transformation tool should allow to enter and edit graphs, e. g., the start graph of a graph transformation model. Due to the notion of graph schema and instance graphs we have introduced in Section 4.1, we are particularly interested in support for
  - (a) *typed graphs* whose conformance is checked against an underlying type graph as discussed in Section 4.1.1,
  - (b) *attributed graphs* with attributes declared in the type graph as discussed in Section 4.1.2,
  - (c) node type *inheritance* including the correct resolution of inherited edge types and attributes,
  - (d) specifying *cardinalities* for edge types as discussed in Section 4.1.3.
2. In order to specify platform mechanism by graph transformation rules, the tool should provide a *rule editor* which allows a style architect
  - (a) to enter graph transformation rules by their *left- and right-hand sides* including a correspondence mapping between the elements of the two sides so that their intersection is well-defined and we can omit the gluing graph as discussed in Section 4.2,

- (b) to extend transformation rules by *negative application conditions* as discussed in the same section.
- 3. Based on their operational semantics, graph transformation models can be *simulated* according to the recognize-select-execute paradigm (see Section 4.2.1). In order to support these three simulation steps, the tool should
  - (a) determine the set of enabled transformation rules and *recognize all occurrences* of their left-hand sides in the current host graph,
  - (b) respect the *DPO gluing condition* stated in Definition 4.14 and exclude those occurrences which do not satisfy the dangling or identification condition,
  - (c) perform *cardinality checks* and exclude those occurrences where a rule application would lead to a violation of cardinality restrictions,
  - (d) allow the architect to test certain transformation sequences by *user-specific rule selections* for every transformation step,
  - (e) be able to automatically simulate arbitrary transformation sequences based on *random rule selections*,
  - (f) *execute selected rules* by modifying the current host graph.
- 4. In order to enable *refinement checks* between two transformation models as described in Chapter 7, we require a tool which
  - (a) allows to specify an *abstraction function* between two graph schemas by some means or other as discussed in Section 7.2.1,
  - (b) enables *structural refinement checks* for a given pair of instance graphs based on the abstraction function as discussed in Section 7.2,
  - (c) is able to generate a *graph transition system* as runtime model of a given graph transformation model (see Section 4.3.2),
  - (d) has a built-in technique for *isomorphism detection* in order to minimize the state space of the transition system and to exploit the symmetries of isomorphic states,
  - (e) implements the algorithm for *behavior preservation checks* introduced in Section 7.3.1,
  - (f) implements the algorithm for *observational substitutability checks* introduced in Section 7.3.2.

5. In order to support the *refinement tests* proposed in Section 7.4, we require a tool which provides a mechanism to check
  - (a) *reachability properties* over the induced graph transition system,
  - (b) more complex *temporal logic properties* formulated over the induced graph transition system.
6. Eventually, we prefer a *graphical user interface* for user-friendly editing of graphs and graph transformation rules.

The above list covers only those features we require for our particular purposes; other important aspects of graph transformation theory such as graph parsing etc. are not considered. For this reason, the following survey of available tools does only reflect their suitability for our specific requirements and is not a general ranking. We rather acknowledge that all the tools are well-done implementations usually emphasizing different aspects of graph transformation theory by a number of useful and sophisticated features.

### 8.1.2 PROGRES

The **PRO**grammed **G**raph **RE**placement System PROGRES<sup>1</sup> is a graph transformation tool which emerged from the IPSEN project in the 1980s. The main goal of this project was to mechanize the development of *Integrated Programming Support ENvironments* by graph transformation systems. Such an environment provides tools for various phases of a software life cycle from analysis over design to programming together with accompanying task like configuration and project management [43].

While in the beginning of the IPSEN project pure graph transformation systems with rather simple forms of rules were used [115], the transformation systems were soon extended by additional means to control the application of transformation rules. This led to the development of PROGRES [141, 140], a very powerful example of programmed graph replacement systems as introduced in Section 5.2.3.

The graphical user interface of PROGRES combines visual specification of transformation rules with textual programming of control flows. Its data model is based on typed, attributed graphs, also with node type inheritance and abstract node types, but not with cardinality restrictions for edge types.

In PROGRES, graph transformations are based on the set-theoretic approach mentioned at the beginning of Section 4.1. In principle, this does not affect the specification of transformation rules with left-hand side, right-hand

---

<sup>1</sup>[www-i3.informatik.rwth-aachen.de/research/projects/progres/](http://www-i3.informatik.rwth-aachen.de/research/projects/progres/)

side, and negative application conditions. However, due to the set theoretic foundation, PROGRES does not respect the DPO dangling condition (see Definition 4.14) when applying a transformation rule to an instance graph. Thus, a rule which removes a certain node can be applied even if this causes dangling edges. In such cases, the dangling edges are simply removed together with the node.

This derivation from the DPO semantics can be managed by equipping transformation rules with additional negative application conditions [56]: For each node  $n$  of a rule to be removed, consider all node types  $T$  which are connected to the type of  $n$  in the graph schema. Moreover, consider an occurrence  $o_L$  of the left-hand side in the current host graph. If a  $T$ -node  $n_T$  is linked to  $o_L(n)$  but there is no pre-image of  $n_T$  in the rule's left-hand side, then the DPO dangling condition excludes the application of the rule. In PROGRES, we have to ensure this exclusion manually by a negative application condition stating that there must not be such a node  $n_T$  of type  $T$  linked to  $n$ .

For example, consider the reconfiguration rule **disconnect** in Fig. 8.2 which is typed over the known graph schema of Fig. 5.1 and Fig. 5.6. Its purpose is to remove a **Connector** node  $con$  ( $= n$ ) between two ports. Naturally, this should only happen when the connector is currently not used for sending messages. According to the DPO semantics, the upper variant (a) of the rule perfectly fulfills this purpose because it cannot be applied as long as some node of type **Message** ( $= T$ ) is still linked to the connector. This statement does not hold for the SPO semantics, because a dangling edge between the removed **Connector** node and the **Message** node would simply be deleted in SPO, too. Thus, we have to add a negative application condition to the **disconnect** rule as shown in the lower variant (b) of Fig. 8.2.

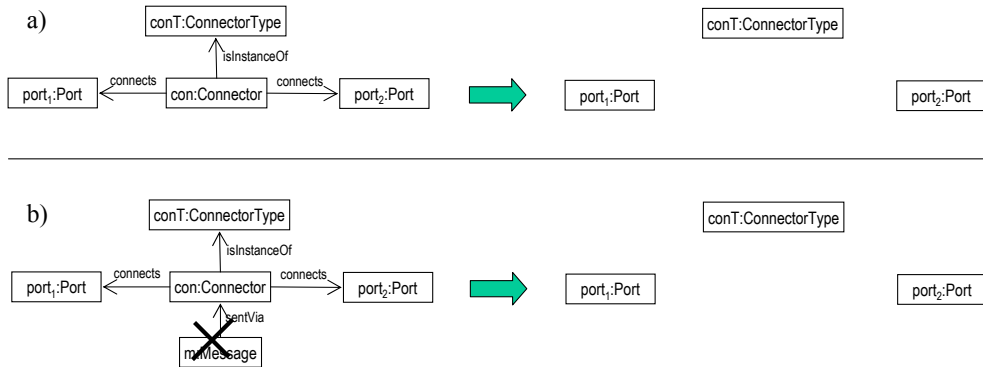


Figure 8.2: Rule extension to ensure DPO dangling condition in SPO tools

The PROGRES language provides explicit control flow constructs to define a user-specific order in which transformation rules are executed by the built-in program interpreter. These constructs also include a non-deterministic choice operator which can be used to realize random rule selections. When the program execution runs into a dead end where none of the rules can be applied any more, the interpreter initiates a back-tracking mechanism which undoes previous rule applications until an alternative execution path has been found.

Unfortunately, PROGRES does not support the generation, exploration and analysis of graph transition systems induced by graph transformation models.

### 8.1.3 AGG

The **A**tttributed **G**raph **G**rammar system AGG<sup>2</sup> is being developed at the TU Berlin. Similar to PROGRES, it aims at the specification and prototypical implementation of applications with complex graph-structured data. For this purpose, graph transformations are combined with the object-oriented programming language Java such that AGG may be used as a general purpose graph transformation engine in high-level Java applications.

Besides an AGG version with a Java API for embedding the tool into other software systems, there is a second, stand-alone version without API for using AGG via its own graphical user interface. This environment supports the visual administration of graphs and rules, and it integrates tools for editing, interpreting, and debugging graph transformation systems.

AGG operates on directed, labeled, attributed graphs. Since release 1.2.0, AGG's data model is extended by type graphs. The type graph does not allow inheritance of node types, but one can prescribe cardinalities for edge types. A built-in type checker ensures that all instance graphs satisfy the typing and multiplicity constraints of the type graph.

Although the name of the tool hints at graph *grammars* only, in fact, arbitrary graph transformation systems can be specified. Transformation rules consist of left-hand side, right-hand side, and forbidden graphs as negative application conditions. Having a graph transformation system at hand, it may be simulated and validated using AGG's analysis techniques, namely critical pair analysis [60] and consistency checking [65].

The execution semantics of transformation rules in AGG is based on the algebraic formalization, but it follows the single pushout approach (SPO [38]) which is more general than the double pushout approach we apply. In par-

---

<sup>2</sup>[tfs.cs.tu-berlin.de/agg/](https://tfs.cs.tu-berlin.de/agg/)

ticular, the SPO approach does not require an occurrence of a left-hand side to satisfy the DPO gluing conditions. However, AGG turns out to be very flexible with respect to these semantical differences, because the user can adapt the setting of the tool in such a way that it respects the dangling and identification conditions when interpreting the transformation rules.

Graph transformations can be executed by the rule interpreter in three different modes. The *interactive transformation mode* allows the user to select a certain rule and to determine a certain matching of its left-hand side. Before the selected rule is executed, the interpreter checks if the matching is a valid occurrence of the left-hand side, if it does not violate a negative application condition, and if the resulting graph does not break any cardinality constraint. In the *automatic transformation mode*, the interpreter computes possible matchings and randomly applies transformation rules to the current host graph until the user stops the interpretation or no rule is applicable any more. Different to this, the *parsing mode* initiates back-tracking when the interpreter runs into a dead end as it tries to find a transformation sequence to a certain stop graph. To ensure termination of the parsing process, AGG requires layered graph grammars as discussed in Section 4.3.1.

Like PROGRES, AGG does not support the generation of graph transition systems and related refinement analysis.

#### 8.1.4 FUJABA

FUJABA (“**F**rom **U**ML to **J**ava **A**nd **B**ack **A**gain”)<sup>3</sup> is a software development tool originating from the University of Paderborn which, since its first release in 1998, has been equipped with an increasing number of useful modules for, e. g., software design, code generation, reverse engineering, real-time support, etc. The original core of the tool is based on *story diagrams* [46], a behavior modeling language which is intended to combine the advantages of both UML and graph transformations.

Story diagrams adopt main features from PROGRES (see Section 8.1.2), e. g., directed, attributed, typed graphs with explicit graph schemas, programmed graph transformations with parameterized rules, and negative application conditions. However, story diagrams extend the PROGRES graph model by direct support for ordered, sorted, and qualified associations and aggregations. The underlying graph schema is modeled as an UML class diagram. Cardinalities can be used to specify restrictions on association multiplicities. Thus, the data model of story diagrams corresponds to the object-oriented data model [46].

---

<sup>3</sup>[www.fujaba.de](http://www.fujaba.de)

In FUJABA, the control flow aspect of programmed graph transformations is specified using UML activity diagrams. The individual activities contain either small pieces of program code (like in UML) or a graph transformation rule. The graph transformation rules are denoted in a UML collaboration diagram-like notation. This way, the paradigm of visual programming is realized to a greater extent than in PROGRES.

Since story diagrams are intended to be used in object-oriented software design, they usually model the body of an operation that belongs to a certain class. Due to the precise semantics of story diagrams which is based on a mapping to the PROGRES language, FUJABA allows to generate Java code implementing the specified operations. For simulation purposes, the FUJABA environment also provides a *dynamic object browser* which visualizes the current object graph of the designed application and the manipulation of this object graph due to the graph transformation-based operations.

Since the execution semantics for transformation rules is borrowed from PROGRES, dangling edges are not prohibited as required by the DPO approach, but they are implicitly deleted as done by PROGRES. Thus, additional negative application conditions should be used as described in Section 8.1.2, in order to achieve the required DPO semantics.

The transformation rules are executed according to the control flow the user has specified in the underlying story diagram. Since the rules are parameterized, the user can also use parameter values to determine a specific matching for the rule. If, in spite of the parameters, there are several possible matchings, the interpreter selects one of them non-deterministically.

Although cardinalities can be specified in the graph schema, they are not necessarily respected during rule applications. In detail, the interpreter can only distinguish to-one from to-many associations but does not respect other upper bounds. And, if a cardinality allows a single link to a certain node type only, one can link a new object of that type even if there already exists another link. In this case, the creation of the new link simply overwrites the existing link.

Similar to PROGRES and AGG, FUJABA does not support the concept of explicit graph transition systems and related refinement analysis. First attempts to fill this gap have been made with the following two tools, CheckVML and GROOVE.

### 8.1.5 CheckVML

CheckVML is a tool by Varró et al. [137] that targets at the *verification* of graph transformation models. In an ideal case, verification means to ensure that a model fulfills all its requirements. For this purpose, CheckVML



combines graph transformation with recent *model checking* approaches.

Model checking is a verification technique that has shown practical relevance for hardware verification. It operates on so-called *Kripke structures* which are state transition systems where a state consists of a subset of a finite universe of logical propositions. A model checker evaluates temporal logic formulas over such transition systems in order to check liveness and safety properties.

Recently, model checking has gained increasing attention in the area of software verification although the dynamic and concurrent nature of software systems makes the problem inherently harder. The main difficulty comes with the *state explosion problem* which arises when modeling concurrent processes. Since the efficiency of model checking algorithms depends on the size of the state space, sophisticated means of abstraction and state reduction are required in order to keep the state space reasonably small.

Existing model checkers like SPIN [74] already incorporate a large number of research results on efficiently tackling model checking problems. For this reason, the authors of CheckVML decided to build their approach on top of these solutions and to provide a translation of graph transformation models into the input domain of existing model checkers like SPIN.

CheckVML accepts graph transformation models with attributed graphs that are typed over a certain type graph. The type graph concept supports inheritance but no edge cardinalities. Each transformation rule is specified by graphs for the left-hand side, the right-hand side, and optional negative application conditions. The tool has no graphical user interface but imports the specifications from XML files provided by the user.

The imported graph transformation model is translated into Promela which is the input language of the SPIN model checker. Since Promela uses propositional logic to define system states and transitions, CheckVML expresses all reachable graphs in terms of propositions and generates the transitions by recursively applying the transformation rules. Concerning its rule application strategy, CheckVML removes all dangling edges when deleting a node. Since this contradicts the DPO dangling condition, we have to modify our rules the same way as described for PROGRES in Section 8.1.2.

The logical propositions are represented by boolean variables as described by Varró in [154]. The truth value of such a variable indicates the existence or non-existence of a certain node or edge. Thus, the conjunction of all variables represents a specific graph. In order to minimize the required memory space, CheckVML distinguishes between *static* and *dynamic* node and edge types. Static types are those that are never changed by a transformation rule. Since their instances remain the same in all states of the transition system, it is sufficient to represent only the dynamic elements by boolean variables.



For every dynamic node and edge type, CheckVML generates the declaration of an *array* of booleans in Promela. The individual fields of the array can be “switched” on or off to indicate the presence of an instance of that type. If a rule application deletes one instance and another rule application creates a new instance of the same type, the corresponding field of the array can be reused for the new instance. As a result, certain but not all isomorphic graphs are aggregated into a single state. This incomplete isomorphism detection might let the state space grow unnecessarily large.

Model checkers like SPIN can only tackle *finite* state transition systems. Due to this fact, the CheckVML user has to fix an *upper bound* in advance, which limits the number of instances of a dynamic type that can be present in each state. Technically, this upper bound determines the length of the instance arrays. If in a state all fields of a certain array are already occupied, CheckVML does not consider rule applications creating further instance of that type in this state. These “cuts” implicitly restrict the size of the state space to the combinatorial product of all array sizes. On the other hand, they lead to a Promela specification which does not represent the complete behavior of the original transformation model.

Despite this limitation, the advantage of the CheckVML approach lies in the ability to check transformation models against arbitrary temporal logic properties with optimized off-the-shelf model checkers like SPIN. The CheckVML tool itself is used as a *preprocessor* which translates both the model and the property into a specification for the model checker. After this preprocessing, we can feed the Promela code into SPIN and let it check the LTL formula.

### 8.1.6 GROOVE

GROOVE (“**GR**aphs for **Ob**ject-**O**riented **VE**rification”)<sup>4</sup> is a toolset for graph transformation-based software verification recently developed by A. Rensink from the University of Twente [130]. Similar to our approach, he uses graphs to represent a system’s runtime states and graph transformation rules to model transitions between these states during a system execution. Concerning verification, his aims are close to those of CheckVML; but instead of applying off-the-shelf model checkers like SPIN, he plans to develop a new model checker which is especially suited for checking temporal properties of graph transition systems with a dynamic number of propositions (see Section 4.3.2) [131]. A more detailed comparison between GROOVE and CheckVML is provided by their authors in [132].

---

<sup>4</sup>[groove.sourceforge.net](http://groove.sourceforge.net)

At the current stage, the GROOVE project has yielded an editor for graphs and graph transformation rules, as well as a simulator which allows to generate a partial or full transition system from a graph transformation model. The graph model Rensink applies is very simple: it only knows edge labels, but no higher level concepts such as type graphs, attributes or cardinalities. Node labels can be simulated by adding a loop edge to a node which carries the intended label. With such node and edge labels, one can restrict the applicability of graph transformation rules, but they are not as powerful as real types (cf. Section 4.1.1). As a node might own several loop edges with different labels, these node labels can also be used as a workaround for representing node attributes.

The rule editor supports graph transformation rules, also with negative application conditions. The simulation component allows to execute these rules in various modes either manually or automatically. During each simulation, the reached graphs are stored and a graph transition system is generated. Thus, a full exploration of the state space results in a complete graph transition system. One of the most valuable features of this simulator is its capability to identify isomorphic states during the state space generation. This way, the size of the transition system is not unnecessarily blown up.

Although GROOVE does not support any similarity analysis between two transition systems as required for our behavioral refinement checks, the explicit generation of graph transition systems is a necessary prerequisite for an efficient implementation of the algorithms introduced in Section 7.3. Moreover, the exploration of the transition system can be terminated with the application of a dedicated transformation rule as required for refinement tests based on reachability analysis (see Section 7.4).

### 8.1.7 Summary

Table 8.1 summarizes the results of the preceding survey. All tools allow to enter and edit – in the bounds of their respective data model – meaningful graph transformation rules, even with negative application conditions. FUJABA, PROGRES, AGG, and CheckVML support typed, attributed graphs. In the case of GROOVE, we can only approximate typing concepts and attributes by special edge labels. Full support of cardinality constraints, including automatic constraint checking, is only provided by AGG.

Except for CheckVML, which is intended to translate graph transformation models into the input language of model checkers, all tools allow the execution of transformation rules, i.e., the simulation of graph transformation models. As only AGG supports the DPO gluing condition, a user of the other tools has to ensure this condition himself using the workaround

Table 8.1: Comparison of graph transformation tools

Requirement	PROGRES	AGG	FUJABA	CheckVML	GROOVE
1. graphs and graph schema					
(a) typed graphs	x	x	x	x	-
(b) attributed graphs	x	x	x	x	-
(c) inheritance	x	-	x	x	-
(d) cardinalities	-	x	x	-	-
2. graph transformation rules					
(a) left- and right-hand sides	x	x	x	x	x
(b) negative application conditions	x	x	x	x	x
3. simulation of transformation models					
(a) recognize LHS occurrences	x	x	x	-	x
(b) DPO gluing condition	-	x	-	-	-
(c) cardinality checks	-	x	-	-	-
(d) user-specific rule selections	x	x	x	-	x
(e) random rule selections	x	x	-	-	x
(f) execute selected rules	x	x	x	-	x
4. refinement checks					
(a) abstraction function	-	-	-	-	-
(b) structural refinement checks	-	-	-	-	-
(c) transition system generation	-	-	-	x	x
(d) isomorphism detection	-	-	-	-	x
(e) behavior preservation checking	-	-	-	-	-
(f) observational substitutability checking	-	-	-	-	-
5. refinement tests					
(a) reachability properties	-	-	-	x	x
(b) temporal logic properties	-	-	-	x	-
6. graphical user interface	x	x	x	-	x

described in Section 8.1.2.

According to expectations, none of the available tools support the new refinement concepts for graph transformation models presented in this thesis. Neither the definition of abstraction functions and structural refinement checks nor behavioral refinement checks can be found. Only the two young approaches GROOVE and CheckVML come closer to our refinement-related requirements because they aim at analyzing induced graph transition systems. While both tools generate the space of reachable states, only GROOVE provides complete isomorphism detection. For this reason, GROOVE would be our favorite candidate for implementing the refinement checking algorithm of Section 7.3 on top. This way, we could do automated refinement checks between abstract and concrete architectural models. However, such implementation is beyond the scope of this thesis.

Although neither GROOVE nor CheckVML enable complete refinement checks so far, we can at least do refinement tests exploiting their capability to search the state space for a state with certain properties or a transition with a certain transformation rule. Thanks to the connection with the model checker SPIN, CheckVML also allows to check more complex LTL formulas like those we have used for behavior preservation tests in Section 7.4.1.

## 8.2 UML tools and tool integration

As outlined in Chapter 6, the application architect should be able to design software architectures in a high-level modeling language such as UML. The creation of UML models is a standard task, and there are many commercial UML tools with varying degree of language support. Almost all of them provide a graphical editor for the diagram types relevant to architectural modeling, i. e., class diagrams, component diagrams, and activity diagrams or state charts.

As a consequence of our style-based modeling approach for dynamic architectures, we also require support for style-specific UML profiles: The style architect needs to create new UML profiles with stereotypes and other extensions of the UML meta-model; and the application architect requires an editor which allows to select one of the profiles for describing an architecture in a certain architectural style.

A detailed survey of existing off-the-shelf UML CASE tools is beyond the scope of this thesis. One example that supports the application and definition of new UML profiles is *Poseidon* by Gentleware<sup>5</sup>.

---

<sup>5</sup>[www.gentleware.com](http://www.gentleware.com)

Continuing our argument from the beginning of this chapter, we propose to assemble a development environment for platform-consistent architectures out of several specialized modeling and graph transformation tools. In contrast to a single, monolithic software, such an environment allows to apply the best-fitting CASE tool for each of the tasks defined in Fig. 8.1 (*best-of-breed* approach). For this purpose, the tools need to exchange their intermediate results. Hence, *data flow* becomes a crucial issue for tool integration.

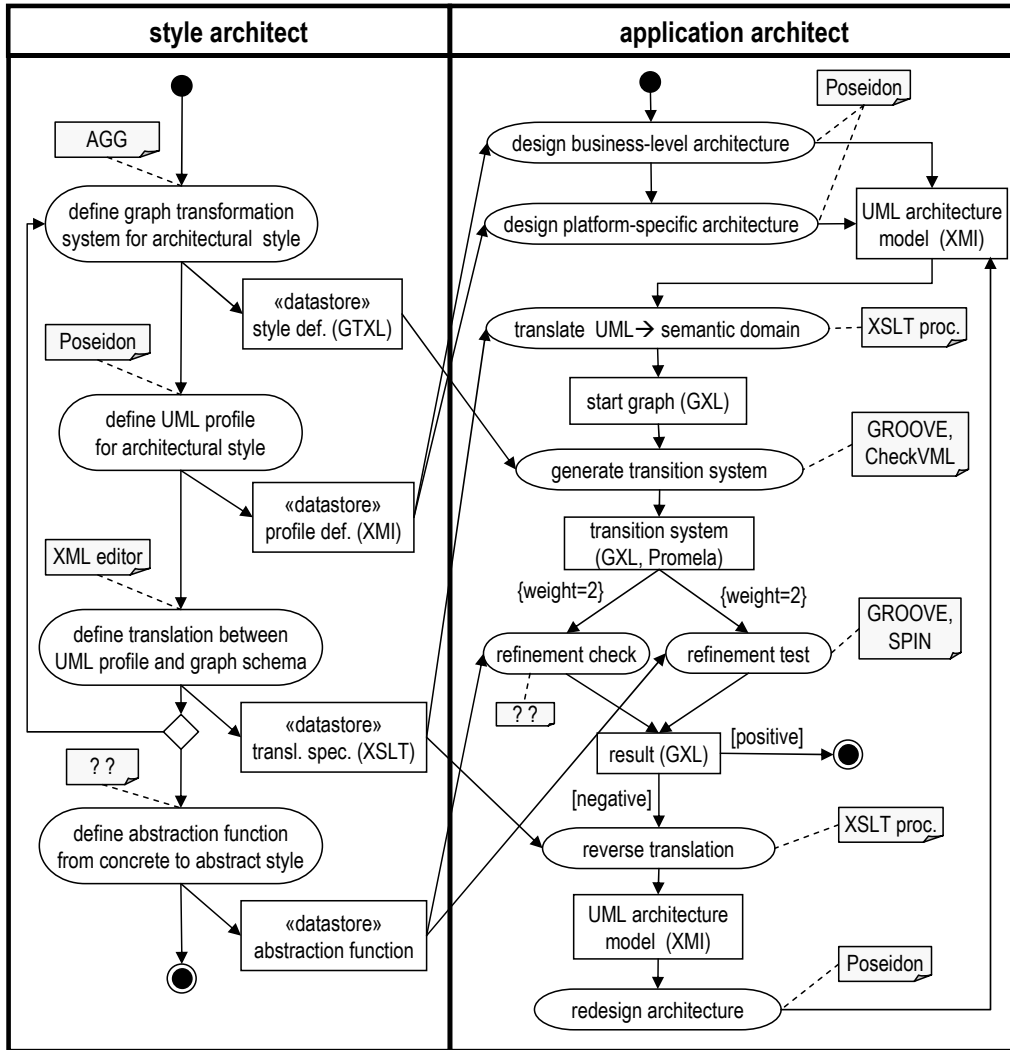


Figure 8.3: Data flow in an integrated tool environment

In order to illustrate the necessary data flow for our approach, Figure 8.3 supplements the activity diagram from the beginning of the chapter with ob-

ject flow states (rectangular boxes) representing data objects and adjacent object flow edges indicating source and target of each data flow. As every data flow also implies a control flow constraint, the two processes of style and application architect can now be synchronized by the data flow dependencies. For instance, the application architect can only start designing his architectures after the style architect has delivered the profile definitions for the corresponding architectural styles.<sup>6</sup>

In order to exchange the various intermediate results, dependent tools require appropriate import and export interfaces as well as common exchange formats which capture, e. g., models, graphs, or transformation systems. In recent years, the *eXtensible Markup Language* (XML)<sup>7</sup> has become a de facto standard for the definition of such exchange formats, and there is already a number of XML-based languages which can be used in our domain:

- The *XML Metadata Interchange* (XMI) format [119] is a vendor-independent standard which allows to store and exchange UML models. Nowadays, nearly all UML tools support the import and export of XMI files. As the definition of profiles is a built-in feature of UML, one can save and exchange UML profile definitions using the XMI format, too.
- The *Graph eXchange Language* (GXL)<sup>8</sup> allows to describe typed, attributed graphs which can be extended to represent hypergraphs and hierarchical graphs. An advantage of GXL is that it can be used to exchange instance graphs together with their corresponding graph schema in a uniform format [159]. AGG, CheckVML, and GROOVE support GXL and use it, e. g., for the start graph of a transformation model.
- The *Graph Transformation eXchange Language* (GTXL)<sup>9</sup> is being developed as an extension of GXL in order to share entire graph transformation rules between tools. GTXL is already supported by AGG and CheckVML, and this number will certainly grow in the near future.

Another advantage of XML is the existence of a related standard, called *XSL Transformations* (XSLT)<sup>10</sup>, which allows to describe transformations of XML documents from one format into another format. So-called *XSLT processors* are available which execute these format transformations and,

---

<sup>6</sup>Readers not familiar with the UML constructs in Fig. 8.3 and the underlying token flow semantics are referred to [41] for further explanations of UML activity diagrams.

<sup>7</sup>[www.w3.org/XML/](http://www.w3.org/XML/)

<sup>8</sup>[www.gupro.de/GXL](http://www.gupro.de/GXL)

<sup>9</sup>[tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html](http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html)

<sup>10</sup>[www.w3.org/TR/xslt](http://www.w3.org/TR/xslt)

thus, help to overcome the gap between different input and output formats. For instance, the style architect could specify translations for his architectural styles as XSLT documents which enable the application architect to apply an XSLT processor for translating UML models from XMI into GXL graphs and back again (see Fig. 8.3). More on XML-based integration of software components in general can be found in [32].

In Fig. 8.3, we have already assigned a selection of possible data formats to the object flow states (printed in brackets at the end of each label). Moreover, the UML note elements attached to the various tasks show potential candidates out of the above considered CASE tools which support the selected import and export formats. Naturally, the tasks belonging to the new refinement technique introduced in Chapter 7, i. e., the definition of an abstraction function and the refinement checking, are not supported by existent tools yet. While this part remains future work, the next section shows that at least refinement *tests* can already be done today using, e. g., GROOVE.

## 8.3 Practical example with GROOVE

We complete this chapter by a practical example using the GROOVE tool. Our intention is not only to show how the tool works, but also to provide some practical impression on the size of the induced graph transition systems and how GROOVE supports refinement tests.

The electronic travel agency serves us as application example again: We entered two different variants of its architecture into GROOVE, the first one in the component-based architectural style (see Sections 5.1 and 5.2) and the second one in the service-oriented style (see Section 5.3).

As GROOVE does not support the concept of graph schemas, we can represent the architectural styles by their graph transformation rules only. The two models of the travel agency architecture are represented by suitable start graphs, respectively, which the transformation rules can be applied to. In order to achieve approximate effects as through type graphs, we use node and edge labels in both the transformation rules and the start graphs as explained in Section 8.1.6.

Figure 8.4 shows a screenshot of the GROOVE simulator after having loaded the graph transformation rules of the component-based style (see left area). The transformation rule **callOperation**, earlier introduced in Fig. 5.14, has been selected and is displayed in the central area. GROOVE depicts transformation rules as a single graph; elements printed with bold, green lines represent nodes and edges to be added, and elements printed with dashed, blue lines are to be deleted. The other elements are kept unchanged.

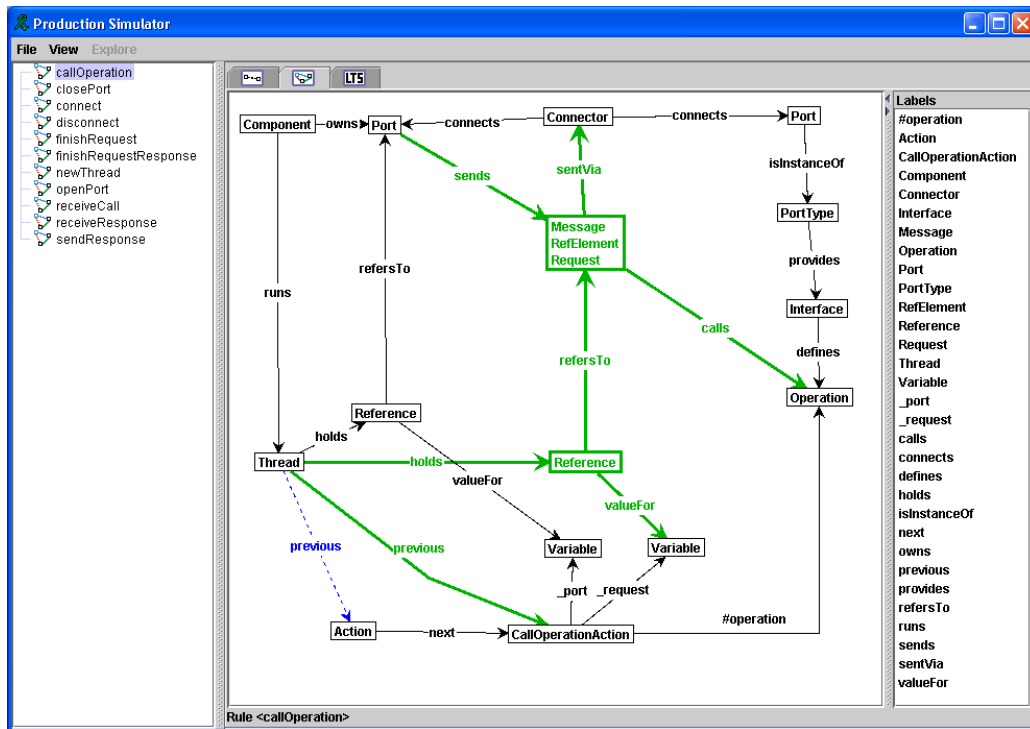


Figure 8.4: Screenshot of a transformation rule in the GROOVE simulator

After loading the transformation rules of the architectural style, the GROOVE simulator requires the start graph modeling the architecture in its initial configuration. Table 8.2 gives some basic figures about the size of the two start graphs. Although the concrete model in the service-oriented style contains only one more component than the abstract model, namely the discovery service for retrieving service descriptions, the total number of nodes and edges grows from 206 to 381. This is caused by platform-specific details which are also added to the business-relevant components.

Due to their sizes, we do not want to depict the start graphs of the

Table 8.2: Start graph sizes of the two travel agency models

	abstract model in the component-based style	concrete model in the service-oriented style
Component nodes	3	4
all start graph elements	206	381



two model variants here. Instead, complete UML models of the architecture both in the component-based and the service-oriented style (based on the corresponding UML profile) can be found in Appendices C and D.

A loaded start graph together with the style's graph transformation rules forms a valid graph transformation model from which the GROOVE simulator generates the corresponding graph transition system. The screenshot in Fig. 8.5 shows some part of the transition system of the abstract model. The nodes represent states, and the edges represent transitions. Final states are colored in red. By selecting one of the states, one can zoom into the state and watch the underlying instance graph.

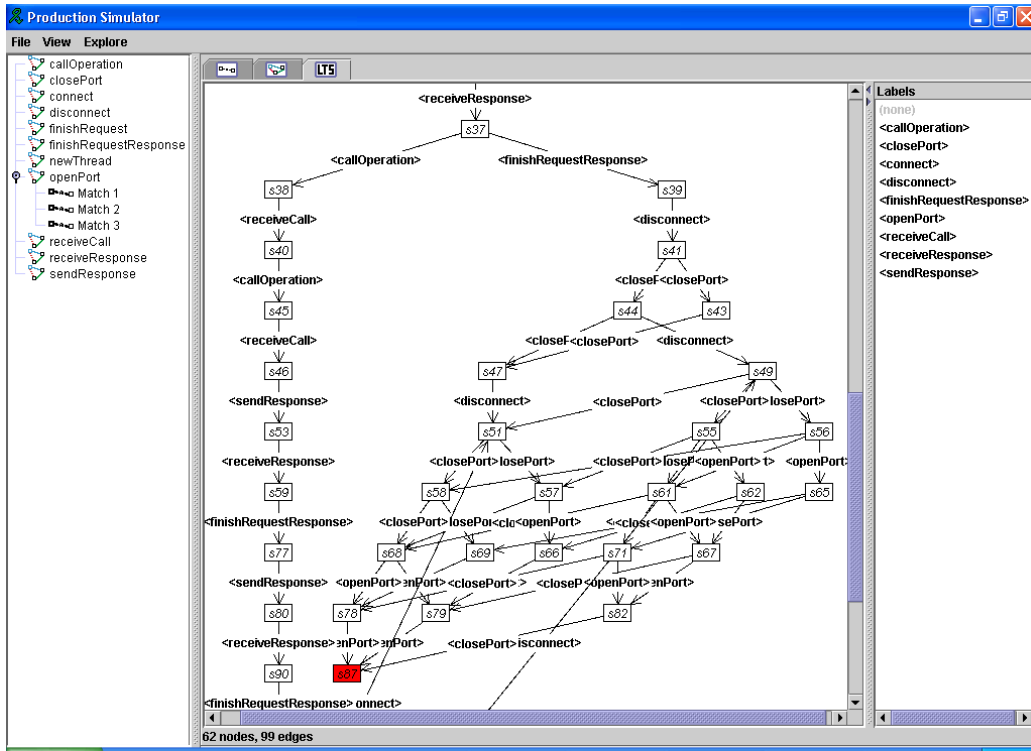


Figure 8.5: Screenshot of a graph transition system generated by GROOVE

Table 8.3 gives some basic figures about the two transition systems generated by GROOVE. Although both transition systems have in common that they converge into a single final state, the difference in the number of states and transitions is even far greater than the above reported difference between the two start graph sizes. This is an effect of the already mentioned state explosion problem, as the concrete model embodies a larger number of and more fine-grained concurrent threads. Nevertheless, the generation time

Table 8.3: Graph transition systems of the two travel agency models

	abstract model in the component-based style	concrete model in the service-oriented style
states	62	3248
final states	1	1
transitions	100	10316
generation time on a 1 GHz PC	2s	112s

of the concrete transition system is still less than 2 minutes on a low-end PC, which is acceptable and leaves room for even larger models.

**Refinement test.** With GROOVE, we are not able to do complete refinement checks. However, the tool is applicable for refinement tests as introduced in Section 7.4. For example, we can test observational substitutability using GROOVE in the following way: (1) select a certain path from the concrete graph transition system, (2) derive an *abstraction proposition* rule from each state of this path, (3) add these proposition rules to the abstract graph transformation model, (4) let GROOVE generate the resulting graph transition system, and (5) check, whether the transition system contains at least one application of the *last* proposition rule. In the following paragraphs, we will detail these steps.

At first, we generate the graph transition system for the concrete architecture model and select a path from the start state to the final state as test case. For this path, we then intend to find a corresponding abstract path in the abstract transition system. As we select the test case randomly, we could also use GROOVE’s *linear* exploration mode. In this mode, GROOVE follows only a single outgoing transition from every state, i. e., it produces a random path instead of a complete transition system.

The actual test case we have selected from the concrete transition system (characterized in the right column of Table 8.3) consisted of 70 states including the initial and the final ones. According to Section 7.4.2, we then had to derive a suitable proposition rule for each of these states. In a first step, we saved the 70 involved instance graphs in the GXL format using GROOVE’s export function. Then, we converted the GLX files into GROOVE XML files containing a proposition rule each. For this purpose, we defined an XSLT specification which removes all behavior-related and platform-specific elements from the instance graph, declares the remaining elements as parts of

both the left-hand and right-hand sides, and adds a counter node whose value is increased by the rule application. This way, an XSLT processor can automatically create all proposition rule files.

The proposition rules have to be added to the abstract graph transformation model. Besides, the existing abstract graph transformation rules have to be extended by the counter node (see Section 7.4.2). Figure 8.6 shows a screenshot of GROOVE after loading the so-modified abstract graph transformation model. One of the 70 proposition rules has been selected and is displayed in GROOVE's characteristic single-graph representation. We can see that all platform-specific elements belonging to the service-oriented style and all behavior-related elements have been removed.

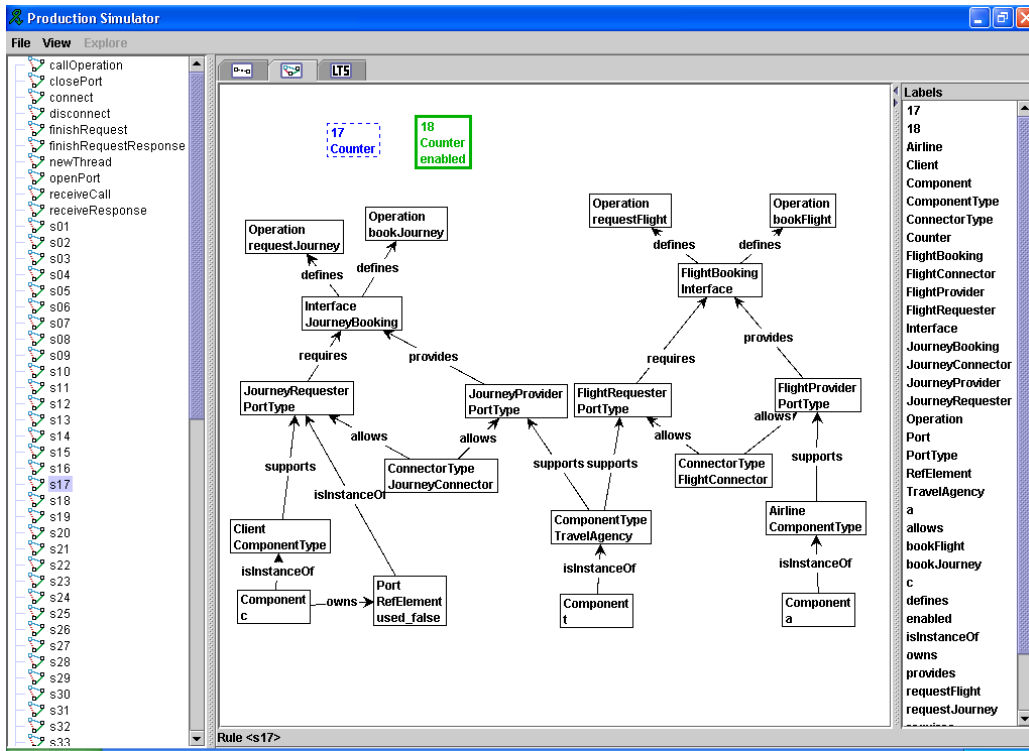


Figure 8.6: Screenshot of a proposition rule in GROOVE

Now, we can start the GROOVE simulator with the modified abstract transformation rules, the added proposition rules, and the start graph for the business-level travel agency architecture. The full exploration of the induced graph transition system by the GROOVE simulator generates 453 different states. If the last proposition rule *s70* becomes applicable in one of these states, then this implies the existence of a path which starts in the start

state and contains applications of all 70 proposition rules in increasing order (see Section 7.4.2).

The screenshot in Fig. 8.7, which shows some part of the generated transition system, contains even several states in which the simulator could apply `s70`. A corresponding path containing all abstraction propositions can easily be found by tracing back a path from one of these states to the start state. This path represents a valid abstraction of the concrete path we have selected as test case. Hence, this observational substitutability test is passed.

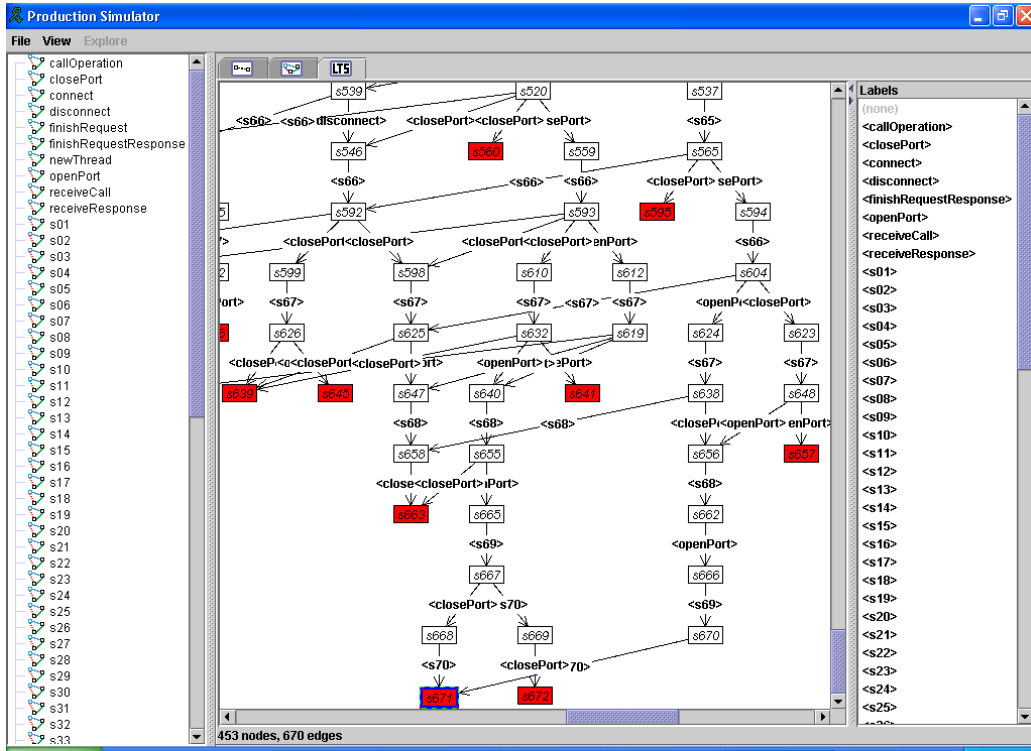


Figure 8.7: Screenshot of the test transition system in GROOVE

Doing an analogous test for behavior preservation would require the following steps: (1) select a certain path from the abstract graph transition system, (2) derive a *refinement proposition* rule from each state of this path, (3) add these proposition rules to the concrete graph transformation model, (4) let GROOVE generate the resulting graph transition system, and (5) check, whether the transition system contains at least one application of the *last* proposition rule.

However, when trying out this procedure for our travel agency example, the fourth step failed on our low-end test PC. The failure is caused by the

state explosion effect of the counter node, which we have already predicted for such behavior preservation tests in Section 7.4.1. Hence, instead of the reachability analysis, we should better apply the CheckVML tool in conjunction with a model checker, as also proposed in Section 7.4.1. CheckVML does not encode any refinement propositions like the counter node into the model blowing up the state space, but it generates a separate temporal logic formula which is then evaluated by the model checker.

## 8.4 Summary

The discussion in this chapter has shown that the existent tools are useful but not completely satisfactory for practical work with our approach. Future efforts are required to build up an integrated tool environment and, especially, to implement the necessary modules for automated refinement checks.

The practical example in Section 8.3 has recalled to our mind the risk of state explosion we have to cope with due to the semantically defined refinement criteria. On the other hand, the example also shows the feasibility of state space generation – even on relatively slow machines – for smaller models, which we usually encounter at the architectural level of design.



# Chapter 9

## Conclusion

The research goal of this work was to develop a modeling and refinement technique for dynamic software architectures which enables the software architect to consider functional and business requirements separate from platform-specific requirements. The separation of these two kinds of requirements promises

- less design errors because a stepwise approach subdivides the overall complexity into several pieces,
- more flexibility because the decision for a certain target platform can be postponed to a later phase of the development process, and
- better reusability because platform-independent models can be reused when porting an architecture to another platform.

Altogether, we believe that the expected productivity gain justifies the higher effort caused by the creation and consistency management of several architecture models. Moreover, the technique proposed in this thesis reduces the required effort substantially: The style-based modeling approach with its systematic platform descriptions ensures the consistency of architecture models to platform capabilities, and the refinement approach on top ensures the mutual consistency between models at different levels of platform abstraction.

In this final chapter, we will at first revisit the requirements we started with in Chapter 2 and discuss how they are satisfied by our approach. Then, we will draw our conclusions and point out some of the issues which remain open for future work.

## 9.1 Evaluation against the requirements

In Section 2.2, we identified 25 requirements – subdivided into the three categories *architecture description*, *platform description*, and *architecture refinement* – whose satisfaction we consider necessary and desirable for a step-wise, platform-consistent development technique of dynamic software architectures. As these requirements characterized the objectives of our work, it is now time to review our results and to evaluate them against these objectives<sup>1</sup>.

**Architecture and platform description.** The requirements of the first two categories are satisfied by our style-based modeling approach: Every architecture is designed as an instance of a certain architectural style. The style functions as platform model representing the capabilities and restrictions of a certain target platform. This way, conformance to the architectural style entails platform consistency of the architectural model.

Formally, architectural styles are specified as graph transformation systems consisting of a graph schema and a set of graph transformation rules, and instances of an architectural style are specified as instance graphs over the graph schema to which the transformation rules can be applied (cf. Chapter 5).

Graphs as formal architecture descriptions are very variable and allow to represent all required modeling constructs using different kinds of nodes. As demonstrated in Section 5.1, this includes component and connector types (1) as well as instances of these types building a certain runtime configuration (3). The required node and edge types for these modeling elements are defined in the style’s graph schema which captures the platform vocabulary (12). Hence, we can distinguish two different typing relations: one between model (graph) and style (graph schema), and the other one within models, e. g., between component types and components.

As a graph schema consists of a type graph, cardinalities, and other constraints, it can easily be used to reflect the topological constraints imposed by the underlying middleware platform (13). The formal typing morphisms from instance graphs to their graph schema relate model elements to the platform vocabulary, which was required as structural instantiation (17).

Besides structural aspects, we also required the incorporation of platform-specific communication (14) and reconfiguration mechanisms (15) into the platform model. As we use graphs to describe architectural configurations, graph transformation rules are ideally suited to describe reconfiguration

---

<sup>1</sup>Like in previous chapters, references to the corresponding requirement in Section 2.2 are put in parentheses.



mechanisms from simple up to more complex operations (4) (Section 5.2.1).

By also including communication objects such as messages, events, and so forth in the graph models, we are able to describe communication mechanisms by graph transformation rules, too (Section 5.2.2). As required for a platform model, all transformation rules are defined over the style vocabulary and kept independent of a concrete application (14)(15).

In Section 5.2.3, we have shown how to also encode the communication behavior of components into our graph models (2). For the sake of reconciliation between communication and reconfiguration actions (6), our process definitions combine both kinds of operations and determine their mutual control flow dependencies. Every action refers to one of the mechanisms of the architectural style (18). For this reason, we extended the graph transformation rules so that their application order conforms to the defined processes. This way, our approach also supports programmed reconfigurations (5).

The operational semantics of graph transformation systems explained in Chapter 4 allows us to simulate the concurrent communication and reconfiguration processes of the involved components (8). Based on this operational semantics, one can apply various analysis techniques (9), e. g., the derivation of a graph transition system which can then be analyzed or model-checked (cf. Section 4.3).

By their nature, graph transformation rules support modeling asynchronous behavior. However, by capturing the state of two or more components in the precondition of a rule, we are also able to define synchronized actions (7).

Both the graph schema and the set of transformation rules can easily be augmented by new elements, which satisfies our extensibility requirement for architectural styles (16).

Since we managed to capture all modeling constructs by typed graphs and graph transformation rules, the overall formal underpinning of the approach remains homogeneous and quite simple (10). On the other hand, the graph models can grow very large in size. For this reason, we proposed to add a high-level modeling language like UML on top of the formal representation in order to make the approach more user-friendly (11). In Chapter 6, we explained how to adapt UML to specific architectural styles using profiles and how to automatically convert UML models into the formal semantic domain.

**Architecture refinement.** The third category of requirements in Section 2.2 dealt with architectural refinement. Again, we proposed a style-based solution, building on an abstraction function which maps the vocabulary of

a concrete to that of an abstract architectural style (Section 7.2.1). This abstraction relationship can be reused for any two instances of the architectural styles in order to check whether a concrete architecture preserves the structure of an abstract architecture (19)(24)(Section 7.2).

Applying the structural refinement criterion to runtime configurations allows us to compare the behavior of two architectures at a semantic level (22): By a comparison of all states reachable in the induced graph transition systems, we can check behavioral refinement without any fixed mapping between the available operations or platform mechanisms (Section 7.3). Doing without this mapping, the approach allows for different refinements of a certain operation depending on the current context (23).

Similar to refinement concepts in the concurrent systems domain, we defined suitable simulation relations between the states of transition systems whose existence entails behavior preservation (20) and observational substitutability (21). The former proves that every abstract behavior is preserved at the platform-specific level, and the latter proves that every concrete behavior is also allowed at the business-oriented level.

Eventually, we aimed at appropriate tool support for automating the refinement checks (25). Although we did not realize a new CASE tool, we provided an efficient algorithm for checking behavior preservation and observational substitutability (Section 7.3) and surveyed existent tools for modeling and analysis (Chapter 8). Some of these tools can, e.g., be used to generate the transition systems required for refinement checks.

Since the resulting state space is subject to the state explosion problem, which affects the performance of a subsequent refinement check, we also introduced strategies for testing refinement (Section 7.4). In contrast to the refinement algorithm, refinement tests can also be applied to infinite state spaces.

In summary, the presented approach satisfies all requirements we considered objectives of our work. Nevertheless, there is still a number of open issues for future work as discussed in the following section.

## 9.2 Conclusions and future work

In this thesis, we have proposed the first comprehensive approach to platform-consistent development of dynamic software architectures. It is founded on a stepwise refinement of architecture descriptions from business-level models to platform-specific ones. In this sense, it fits into the conceptual modeling framework of the MDA initiative. However, while MDA's long-term vision

stresses the need for automated model transformations, the focus of this work has been put on ensuring consistency between models at different levels of platform abstraction.

To summarize the main contributions of this work, we elaborated the usage of architectural styles as platform models which classify the set of architectural models conforming to a certain target platform. We showed how to formally define these architectural styles as graph transformation systems yielding architecture descriptions with operational semantics for dynamic, reconfigurable systems. And, to evaluate and illustrate the proposal, we specified two architectural styles, an abstract one for component-based architectures and a concrete one for service-oriented architectures.

In the refinement part of the thesis, we introduced an abstraction relation at the style level which allows us to check any given pair of style instances for structural refinement. And, concerning the refinement of behavior, we transferred refinement concepts from the concurrent systems world to the graph transformation domain. The resulting semantic refinement checks enable the application architect to ensure that the platform-specific model still supports all business-relevant behavior and does not violate any safety condition hold by the abstract behavior.

The presented notion of refinement is not restricted to descriptions of software architectures, but it can in principle be applied to any graph transformation models. However, facing the state explosion problem, its high level of abstraction makes software architecture a well-suited application domain for such semantic approaches.

In spite of the above summarized achievements, one can easily identify a number of issues left open for future work. For instance, the discussion about tool support has revealed that current tools cannot completely realize the proposed development technique. In particular, we need an implementation of the new refinement algorithms for better experiments and a future application of the approach in practice.

Another extension of the presented approach could be a component for tracing uncovered inconsistencies. At the moment, the refinement checks can only verify whether a valid refinement relation exists or not; they cannot tell the reason when the outcome of a check is negative. Hence, future research should work on identifying the model parts which prevent the existence of a valid refinement and suggesting corrections which remove these obstacles.

Concerning the computational complexity of the semantic refinement criteria, future work should try to circumvent the state explosion problem. In this context, a promising research direction could be the combination with syntactic criteria which impose certain restrictions on the model but then

allow to decide refinement without generating the entire state space [55, 58]. Besides, there are promising approaches in the domain of model checking, from exploiting symmetries up to compositional model checking [52].

The modeling part of this thesis leaves room for future improvements, too. For instance, one could consider the application of hierarchical graphs and hierarchical transformation rules in order to represent architectures as a hierarchy of components, splitting one component into several subcomponents (also called *horizontal refinement*). Moreover, one could consider extending the component behavior descriptions by additional concepts such as branching conditions, transactions, or exceptional behavior. The challenge here is to stick to a formalization which allows to preserve the related refinement concepts.

The long-term continuation of this kind of research will try to further approach the vision of completely automated refinement constructions. There is still a long way to go, and – in our opinion – it will not be possible to entirely eliminate the need for human experience and intervention. However, we dare to regard this thesis as a substantial contribution to the computer-aided development of dynamic software architectures.

# Bibliography

- [1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
- [2] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architecture. In *Proc. of the ACM SIGSOFT '93 Symposium on Foundations of Software Engineering*, volume 18(5) of *ACM Software Engineering Notes*, pages 9–20, 1993.
- [3] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.
- [4] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Proc. of the 1st Int. Conference on Fundamental Approaches to Software Engineering, FASE 98*, volume 1382 of *LNCS*, pages 21–37. Springer, 1998.
- [5] R. Allen and D. Garlan. Formalizing architectural connection. In *Proc. of the 16th Int. Conference on Software Engineering*, pages 71–80. IEEE Computer Society Press, 1994.
- [6] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.
- [7] S. Balsamo, M. Bernardo, and M. Simeoni. Performance evaluation at the software architecture level. In M. Bernado and P. Inverardi, editors, *Formal Methods for Software Architecture*, volume 2804 of *LNCS*, pages 207–258. Springer, 2003.
- [8] G. S. Banavar, W. Zwaenepoel, D. Terry, and R. Want, editors. *Proc. of the 2nd Int. Conference on Mobile Systems, Applications, and Services, MobiSys'04*. ACM Press, 2004.

- [9] R. Bardohl, H. Ehrig, J. de Lara, O. Runge, G. Taentzer, and I. Weinhold. Node type inheritance concept for typed graph transformation. Technical Report 2003-19, Technical University of Berlin, 2003.
- [10] R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. In M. Wermelinger and T. Margaria-Steffen, editors, *Proc. 7<sup>th</sup> Int. Conference on Fundamental Approaches to Software Engineering, FASE 2004*, volume 2984 of *LNCIS*, pages 214–228. Springer, 2004.
- [11] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and analysis of architectural styles based on graph transformation. In *Proc. 6th ICSE Workshop on Component-Based Software Engineering (CBSE6): Automated Reasoning and Prediction*, pages 67–72, 2003.
- [12] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and validation of service-oriented architectures: Application vs. style. In *Proc. European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 03*, pages 68–77. ACM Press, 2003.
- [13] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based refinement of dynamic software architectures. In *Proc. of the 4th Working IEEE/IFIP Conference on Software Architecture, WICSA 2004*, pages 155–164. IEEE Computer Society, 2004.
- [14] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based modeling and refinement of service-oriented architectures. *Int. Journal on Software and Systems Modeling, SoSyM*, 2005. to appear.
- [15] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proc. ICSE 2003 – Int. Conference on Software Engineering*, pages 187–197. IEEE, 2003.
- [16] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. Elsevier Science, 2001.
- [17] A. Bertolino, F. Corradini, P. Inverardi, and H. Miccini. Deriving test plans from architectural descriptions. In *Proc. of the International Conference on Software Engineering, ICSE 00*, pages 220–229. ACM Press, 2000.

- [18] T. Bolusset and F. Oquendo. Formal refinement of software architectures based on rewriting logic. In *Proc. of the Int. Workshop on Refinement of Critical Systems, RCS'02*, 2002. [www-lsr.imag.fr/zb2002/](http://www-lsr.imag.fr/zb2002/).
- [19] P. Bottoni, G. Taentzer, and A. Schürr. Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In *Proc. of the 2000 IEEE International Symposium on Visual Languages, VL'00*, page 59. IEEE Computer Society, 2000. Long version available as technical report SI-2000-06, University of Rome.
- [20] J. S. Bradbury. Organizing definitions and formalisms for dynamic software architectures. Technical Report 2004-477, Queen's University, Kingston, Canada, 2004.
- [21] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. A survey of self management in dynamic software architecture specifications. In *Proceedings of the ACM SIGSOFT International Workshop on Self-Managed Systems, WOSS'04*. ACM Press, 2004.
- [22] C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In *Proc. WICSA1, First Working IFIP Conference on Software Architecture*, volume 140 of *IFIP Conference Proceedings*, pages 107–126. Kluwer, 1999.
- [23] M. Champion, C. Ferris, E. Newcomer, and D. Orchard. *Web Service Architecture, W3C Working Draft*. World Wide Web Consortium, 2002. <http://www.w3.org/TR/2002/WD-ws-arch-20021114/>.
- [24] P. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, Vol. 1, No. 1:9–36, 1976.
- [25] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.
- [26] E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking algorithms. In *Proc. of the 6<sup>th</sup> ACM Symposium on Principles of Distributed Computing*, pages 294–303, 1987.
- [27] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

- [28] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series. McGraw-Hill, 1990.
- [29] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–265, 1996.
- [30] 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 [134], *Chapter 3*. 1997.
- [31] E. M. Dashofy, N. Medvidovic, and R. N. Taylor. Using off-the-shelf middleware to implement connectors in distributed software architectures. In *Proc. of the 21st Int. Conference on Software Engineering, ICSE'99*, pages 3–12. IEEE Computer Society Press, 1999.
- [32] R. Depke, G. Engels, S. Thöne, M. Langham, and B. Lütke-meier. Process-Oriented, Consistent Integration of Software Components. In *Proc. 26th International Computer Software and Applications Conference (COMPSAC'02)*. IEEE, 2002.
- [33] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- [34] D. Distefano, A. Rensink, and J.-P. Katoen. Model checking birth and death. In R. Baeza-Yates, U. Montanari, and N. Santoro, editors, *Foundations of Information Technology in the Era of Network and Mobile Computing*, volume 223 of *IFIP Conference Proceedings*, pages 435–447. Kluwer Academic Publishers, 2002.
- [35] A. Dovier, C. Piazza, and A. Policriti. A fast bisimulation algorithm. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of the 13th Int. Conference on Computer-Aided Verification (CAV01)*, volume 2102 of *LNCS*, pages 79–90. Springer, 2001.
- [36] J. Ebert and G. Engels. Specialization of object life cycle definitions. In *Fachberichte Informatik 19/95*. Universität Koblenz-Landau, Germany, 1997.
- [37] 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, 1999.



- [38] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In *[134], Chapter 4*. 1997.
- [39] H. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.
- [40] M. Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In *Proc. of the 12th Brazilian Symposium on Computer Networks*, pages 175–187, 1994.
- [41] G. Engels, A. Förster, R. Heckel, and S. Thöne. Process Modeling Using UML. In *Process-Aware Information Systems*. Wiley, 2005. to appear.
- [42] G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *Proc. UML 2000 - The Unified Modeling Language*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.
- [43] G. Engels and W. Schäfer. *Programmentwicklungsumgebungen: Konzepte und Realisierung*. Teubner, Stuttgart, 1989.
- [44] J. L. Fiadeiro and T. S. E. Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming*, 28(2-3):111–138, 1997.
- [45] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
- [46] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In G. Engels and G. Rozenberg, editors, *Proc. of the 6<sup>th</sup> International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, volume 1764 of *LNCS*. Springer, 1998.
- [47] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1994.
- [48] D. Garlan. Style-based refinement for software architecture. In *Proc. ISAW-2, 2nd Int. Software Architecture Workshop on SIGSOFT '96*, pages 72–75. ACM Press, 1996.

- [49] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proc. of the ACM SIGSOFT '94 Symposium on Foundations of Software Engineering*, ACM Software Engineering Notes, 1994.
- [50] D. Garlan, J. Kramer, and A. Wolf, editors. *Proc. of the 1st Workshop on Self-Healing Systems, WOSS'02*. ACM Press, 2002.
- [51] D. Garlan and M. Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, vol. II:1–39, 1993.
- [52] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the compositional verification of real-time UML designs. In *Proc. of the European Software Engineering Conference, ESEC*, pages 38–47. ACM Press, September 2003.
- [53] R. Gorrieri and A. Rensink. Action refinement. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 1047–1147. Elsevier, 2001.
- [54] H. Göttler, J. Günther, and G. Nieskens. Use graph grammars to design CAD systems. In *Proc. 4<sup>th</sup> Int. Workshop on Graph Grammars and Their Application to Computer Science*, volume 532 of *LNCS*, pages 396–410. Springer, 1991.
- [55] M. Große-Rhode, F. Parisi Presicce, and M. Simeoni. Spatial and temporal refinement of typed graph transformation systems. In *Proc. Math. Foundations of Comp. Science 1998*, volume 1450 of *LNCS*, pages 553–561. Springer, 1998.
- [56] A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996.
- [57] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [58] R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Math. Struct. in Computer Science*, 6:613–648, 1996.

- [59] R. Heckel and G. Engels. Relating functional requirements and software architecture: separation and consistency of concerns. *Journal of Software Maintenance and Evolution: Research and Practice*, 14:371–388, 2002.
- [60] R. Heckel, J.M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 1st Int. Conference on Graph Transformation, ICGT 02*, volume 2505 of *LNCS*, pages 161–176. Springer, 2002.
- [61] R. Heckel, M. Lohmann, and S. Thöne. Towards a UML profile for service-oriented architectures. In *Proc. of Workshop on Model Driven Architecture: Foundations and Applications (MDAFA)*, CTIT Technical Report TR-CTIT-03-27. University of Twente, Enschede, The Netherlands, 2003.
- [62] R. Heckel and L. Mariani. Automatic conformance testing of web services. In *Proc. of the 8th Int. Conference on Fundamental Approaches to Software Engineering (FASE 2005)*, volume 3442 of *LNCS*, pages 34–48. Springer, 2005.
- [63] R. Heckel and S. Thöne. Behavior-preserving refinement relations between dynamic software architectures. In *Proc. of the 17th Int. Workshop on Algebraic Development Techniques, WADT 2004*, volume 3423 of *LNCS*, pages 1–27. Springer, 2004.
- [64] R. Heckel and S. Thöne. Behavioral refinement of graph transformation-based models. In *Proc. of the ICGT 2004 Workshop on Software Evolution through Transformations, SETra 04*, volume 127(3) of *Electronic Notes in Theoretical Computer Science*, pages 101–111. Elsevier, 2005.
- [65] R. Heckel and A. Wagner. Ensuring consistency of conditional graph grammars - a constructive approach. In *Joint COMPU-GRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation, SEGRAGRA 1995*, volume 2 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.
- [66] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. of the 36th IEEE Symposium on Foundations of Computer Science, FOCS 1995*, pages 453–462. IEEE, 1995.

- [67] D. Hirsch. *Graph transformation models for software architecture styles*. PhD thesis, Departamento de Computación, Universidad de Buenos Aires, 2003.
- [68] D. Hirsch, P. Inverardi, and U. Montanari. Graph grammars and constraint solving for software architecture styles. In *Proc. of the 3rd Int. Software Architecture Workshop, ISAW3*, pages 96–72. ACM, 1998.
- [69] D. Hirsch, P. Inverardi, and U. Montanari. Modeling software architecture and styles with graph grammars and constraint solving. In *Proc. of the 1st Working IFIP Conference on Software Architecture, WICSA1*, pages 127–142. Kluwer, B.V., 1999.
- [70] D. Hirsch and U. Montanari. Synchronized hyperedge replacement with name mobility. In *Proc. CONCUR 2001 - Concurrency Theory*, volume 2154 of *LNCS*, pages 121–136. Springer, 2001.
- [71] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.
- [72] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., 1985.
- [73] C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, 2000.
- [74] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
- [75] P. Inverardi and M. Tivoli. Software architecture for correct components assembly. In M. Bernado and P. Inverardi, editors, *Formal Methods for Software Architecture*, volume 2804 of *LNCS*, pages 92–121. Springer, 2003.
- [76] V. Issarny and A. Zarras. Software architecture and dependability. In M. Bernado and P. Inverardi, editors, *Formal Methods for Software Architecture*, volume 2804 of *LNCS*, pages 259–286. Springer, 2003.
- [77] ITU-TS, Geneva. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*, 1996.
- [78] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering – A use case driven approach*. Addison Wesley, 1992.

- [79] J. Jahnke, W. Schäfer, and A. Zündorf. A design environment for migrating relational to object oriented database systems. In *Proc. of the Int. Conference on Software Maintenance*, pages 163–170. IEEE Computer Society Press, 1996.
- [80] N. Kaveh and W. Emmerich. Validating distributed object and component design. In M. Bernado and P. Inverardi, editors, *Formal Methods for Software Architecture*, volume 2804 of *LNCS*, pages 63–91. Springer, 2003.
- [81] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained - The Model Driven Architecture: Practice and Promise*. Addison Wesley, 2003.
- [82] J. Köbler, U. Schöning, and J. Torán. *The graph isomorphism problem: its structural complexity*. Birkhauser Verlag, 1993.
- [83] B. König. A graph rewriting semantics for the polyadic pi-calculus. In *Workshop on Graph Transformation and Visual Modeling Techniques, Proc. of ICALP Workshops 2000*, pages 451–458. Carleton Scientific, 2000.
- [84] J. Kramer. Configuration programming – a framework for the development of distributable systems. In *Proc. of the IEEE Int. Conference on Computer Systems and Software Engineering, COMPEURO’90*, pages 374–384, 1990.
- [85] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.
- [86] J. Kramer and J. Magee. Exposing the skeleton in the coordination closet. In *Proc. of the 2nd IEEE Int. Conference on Coordination Models and Languages, Coord’97*, volume 1282 of *LNCS*, pages 18–31. Springer, 1997.
- [87] J. Kramer and J. Magee. Analysing dynamic change in software architectures: A case study. In *Proc. of the 4th IEEE Int. Conference on Configurable Distributed Systems*, pages 91–100. IEEE Computer Society, 1998.
- [88] J. Kramer, J. Magee, and S. Uchitel. Software architecture modeling and analysis: A rigorous approach. In M. Bernado and P. Inverardi, editors, *Formal Methods for Software Architecture*, volume 2804 of *LNCS*, pages 44–51. Springer, 2003.

- [89] J. M. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn, Germany, 2004.
- [90] S. Mac Lane. *Categories for the Working Mathematician*. Springer New York-Berlin, 1971.
- [91] D. Le Métayer. Software architecture styles as graph grammars. In *Proc. 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 216 of *ACM Software Engineering Notes*, pages 15–23. ACM Press, 1996.
- [92] M. Lefering and A. Schürr. Specification of integration tools. In M. Nagl, editor, *Building Tightly-Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *LNCS*, pages 440–456. Springer, 1996.
- [93] M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M. Sleep, M. Plasmeijer, and M. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 185–199. Wiley, 1993.
- [94] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
- [95] D. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.
- [96] N. Lynch and F. Vaandrager. Forward and backward simulations – part i: Untimed systems. *Information and Computation*, 121(2):214–233, 1995.
- [97] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proc. of the 5th European Software Engineering Conference, ESEC’95*, volume 989 of *LNCS*, pages 137–153. Springer, 1995.
- [98] J. Magee, N. Dulay, and J. Kramer. A constructive development environment for parallel and distributed programs. *Distributed Systems Engineering*, 1(5):304–312, 1994.
- [99] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proc. of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering, FSE’96*, pages 3–14. ACM Press, 1996.

- [100] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.
- [101] J. Magee, C. Szyperski, and J. Bosch, editors. *Proc. of the 4th Working IEEE / IFIP Conference on Software Architecture, WICSA 2004*. IEEE Computer Society, 2004.
- [102] T. De Marco. *Structured Systems Analysis and Specification*. Yourdon Press, 1979.
- [103] N. Medvidovic. On the role of middleware in architecture-based software development. In *Proc. of the 14th Int. Conference on Software Engineering and Knowledge Engineering, SEKE'02*, pages 299–306. ACM Press, 2002.
- [104] N. Medvidovic, E. M. Dashofy, and R. N. Taylor. The role of middleware in architecture-based software development. *International Journal of Software Engineering and Knowledge Engineering*, 13(4):367–393, 2003.
- [105] N. Medvidovic, D. Rosenblum, and R. N. Taylor. A language and environment for architecture-based software development and evolution. In *Proc. of the 21st Int. Conference on Software Engineering (ICSE'1999)*, pages 44–53. IEEE Computer Society Press, 1999.
- [106] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J.E. Robbins. Modeling software architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(1):2–57, 2002.
- [107] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.
- [108] J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. Object Management Group, 2003. [www.omg.org/docs/omg/03-06-01.pdf](http://www.omg.org/docs/omg/03-06-01.pdf).
- [109] R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd Int. Joint Conference on Artificial Intelligence*, pages 481–489. British Computer Society, 1971.
- [110] R. Milner. A calculus of communication systems. *LNCS*, 92, 1980.
- [111] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.

- [112] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural styles, design patterns, and objects. *IEEE Software*, 14(1):43–52, 1997.
- [113] M. Moriconi and X. Qian. Correctness and composition of software architectures. In *Proc. of the ACM SIGSOFT '94 Symposium on Foundations of Software Engineering*, ACM Software Engineering Notes, pages 164–174. ACM Press, 1994.
- [114] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, 1995.
- [115] M. Nagl, G. Engels, R. Gall, and W. Schäfer. Software specification by graph grammars. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Graph Grammars and their Application to Computer Science*, volume 153 of *LNCS*, pages 267–287. Springer, 1983.
- [116] P. Naur and B. Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968*. Scientific Affairs Division, NATO, 1969. [homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF](http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF).
- [117] H. Penny Nii. Blackboard systems. *AI Magazine*, 7(3):38–53, 1986.
- [118] E. Di Nitto and D. Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In *Proc. of the 21st Int. Conference on Software Engineering, ICSE'1999*, pages 13–22. IEEE Computer Society Press, 1999.
- [119] Object Management Group. *XMI: XML Metadata Interchange*. <http://www.omg.org/>.
- [120] Object Management Group. *UML 2.0 OCL Final Adopted Specification*, 2003. [www.omg.org/cgi-bin/doc?ptc/2003-10-14](http://www.omg.org/cgi-bin/doc?ptc/2003-10-14).
- [121] Object Management Group. *UML 2.0 Superstructure Revised Final Adopted Specification*, 2004. [www.omg.org/cgi-bin/doc?ptc/2004-10-02](http://www.omg.org/cgi-bin/doc?ptc/2004-10-02).
- [122] C. Oliveira and M. Wermelinger. The community workbench. In *Proc. of the 26th Int. Conference on Software Engineering, ICSE'04*, pages 709–710. IEEE Computer Society, 2004.



- [123] P. Oreizy. Issues in the runtime modification of software architectures. Technical Report UCI-ICS-TR-96-35, Department of Information and Computer Science, University of California, Irvine, 1996.
- [124] M. P. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Communications of the ACM*, 46(10):25–28, 2003.
- [125] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.
- [126] T. Pilioura and A. Tsalgatidou. E-services: Current technology and open issues. In *Proc. of the 1st Workshop on Technologies for e-Services, TES 2001*, volume 2193 of *LNCIS*, pages 1–15. Springer, 2001.
- [127] T. W. Pratt. Pair grammars, graph languages, and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.
- [128] J. Rekers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.
- [129] F. De Remer and H. H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(2):80–86, 1976.
- [130] A. Rensink. The GROOVE simulator: A tool for state space generation. In M. Nagl, J. Pfalz, and B. Böhlen, editors, *Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE '03)*, volume 3062 of *LNCIS*, pages 479–485. Springer, 2003.
- [131] A. Rensink. Towards model checking graph grammars. In M. Leuschel, S. Gruner, and L. Lo Presti, editors, *Proc. 3<sup>rd</sup> Workshop on Automated Verification of Critical Systems*, Tech. Report DSSE-TR-2003-2, pages 150–160. University of Southampton, 2003.
- [132] A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proc. 2<sup>nd</sup> International Conference on Graph Transformation, ICGT 2004*, volume 3256 of *LNCIS*, pages 226–241. Springer, 2004.

- [133] J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating architecture description languages with a standard design method. In *Proc. of the 20 th International Conference on Software Engineering, ICSE 98*, pages 209–218. IEEE Computer Society, 1998.
- [134] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, 1997.
- [135] J. E. Rumbaugh, M. Blaha, W. J. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1990.
- [136] A. Salomaa. *Formal Languages*. ACM Monograph Series. Academic Press, New York, 1973.
- [137] Á. Schmidt and D. Varró. CheckVML: A tool for model checking visual modeling languages. In *Proc. UML 2003 - The Unified Modeling Language*, volume 2863 of *LNCS*, pages 92–95, 2003.
- [138] A. Schürr. Specification of graph translators with triple graph grammars. In *Proc. WG'94 Int. Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.
- [139] A. Schürr. Programmed graph replacement systems. In [\[134\]](#), Chapter 7. 1997.
- [140] A. Schürr, A.J. Winter, and A. Zündorf. Graph Grammar Engineering with PROGRES. In W. Schäfer, editor, *Software Engineering, ESEC '95*, volume 989 of *LNCS*, pages 219–234. Springer, 1995.
- [141] A. Schürr, A.J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In [\[37\]](#). 1999.
- [142] M. Shaw. Toward higher-level abstractions for software systems. *Data and Knowledge Engineering*, 5:119–128, 1990.
- [143] M. Shaw. The coming-of-age of software architecture research. In *Proc. of the 23rd International Conference on Software Engineering, ICSE 01*, pages 656–664. IEEE Computer Society, 2001.
- [144] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

- [145] M. Shaw, R. De Line, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, 1995.
- [146] D. Soni, R. L. Nord, and C. Hofmeister. Software architecture in industrial applications. In *Proc. of the 17th International Conference on Software Engineering, ICSE'95*, pages 196–207. ACM Press, 1995.
- [147] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *5th STOC*, pages 1–9. ACM, 1973.
- [148] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley, 1998.
- [149] G. Taentzer. Distributed graphs and graph transformation. *Applied Categorical Structures*, Special Issue on Graph Transformation, 7(4):431–462, 1999.
- [150] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *Proc. TAGT'98 - Theory and Application of Graph Transformations*, volume 1764 of *LNCS*, pages 179–193. Springer, 2000.
- [151] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6):390–406, June 1996.
- [152] R. van Glabbeek and U. Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4-5):229–327, 2000.
- [153] A. van Lamsweerde. From system goals to software architecture. In M. Bernado and P. Inverardi, editors, *Formal Methods for Software Architecture*, volume 2804 of *LNCS*, pages 25–43. Springer, 2003.
- [154] D. Varró. Towards symbolic analysis of visual modeling languages. In *Proc. GT-VMT 2002 - Int. Workshop on Graph Transformation and Visual Modeling Techniques*, volume 72 of *ENTCS*, pages 57–70. Elsevier, 2002.
- [155] J. Very, L. Perrochon, and D. C. Luckham. Event-based execution architectures for dynamic software systems. In *Proc. of the 1st Working*

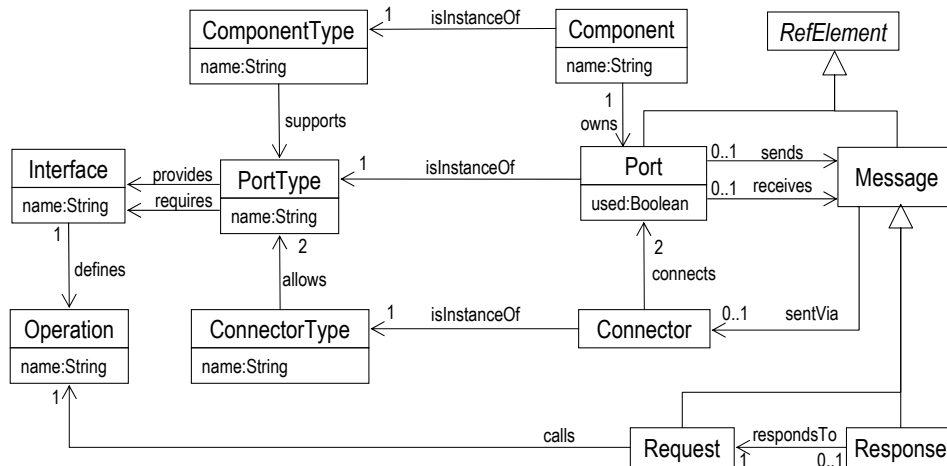
- IFIP Conference on Software Architecture, WICSA1*, pages 303–318. Kluwer, B.V., 1999.
- [156] M. Wermelinger and J. L. Fiadeiro. Algebraic software architecture reconfiguration. In *Proc. of the 7th European Software Engineering Conference and 7th Symposium on Foundations of Software Engineering, ESEC/FSE'99*, pages 393–409, 1999.
- [157] M. Wermelinger and J. L. Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2):133–155, 2002.
- [158] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A graph-based architectural (re-)configuration language. In *Proc. of the 8th European Software Engineering Conference and 9th Symposium on Foundations of Software Engineering, ESEC/FSE'01*, pages 21–32. ACM Press, 2001.
- [159] A. Winter, B. Kullbach, and V. Riediger. An overview of the GXL graph exchange language. In S. Diehl, editor, *Software Visualization: International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001. Revised Papers*, volume 2269 of *LNCS*, pages 324–336. Springer, 2002.
- [160] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.
- [161] L. Zanolin, C. Ghezzi, and L. Baresi. An approach to model and validate publish/subscribe architectures. In *Proc. of the ESEC 2003 Workshop on Specification and Verification of Component-Based Systems, SAVCBS 2003*, Tech. Report 03-11, pages 35–41. Department of Computer Science, Iowa State University, 2003.
- [162] A. Zündorf. *PROgrammierte GRaphErsetzungsSysteme*. PhD thesis, Technical University, Aachen, 1996. Vieweg (in German).

# Appendix A

## Architectural style for component-based architectures

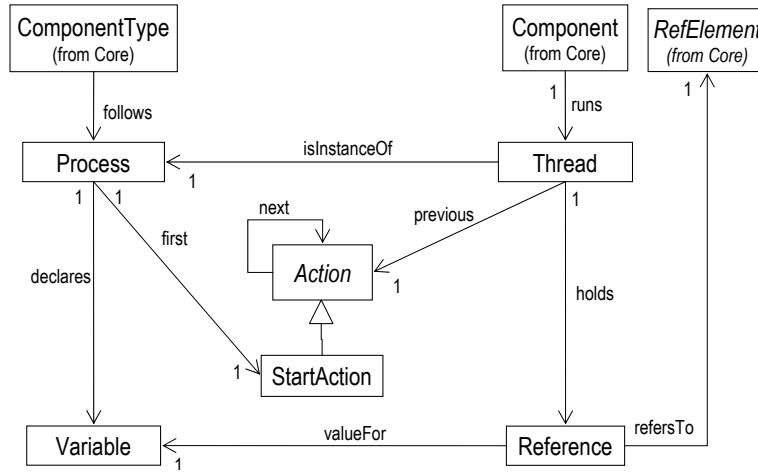
### A.1 Graph schema

Due to its size, the graph schema of the style is subdivided into three *packages*, similar to the UML packaging concept.

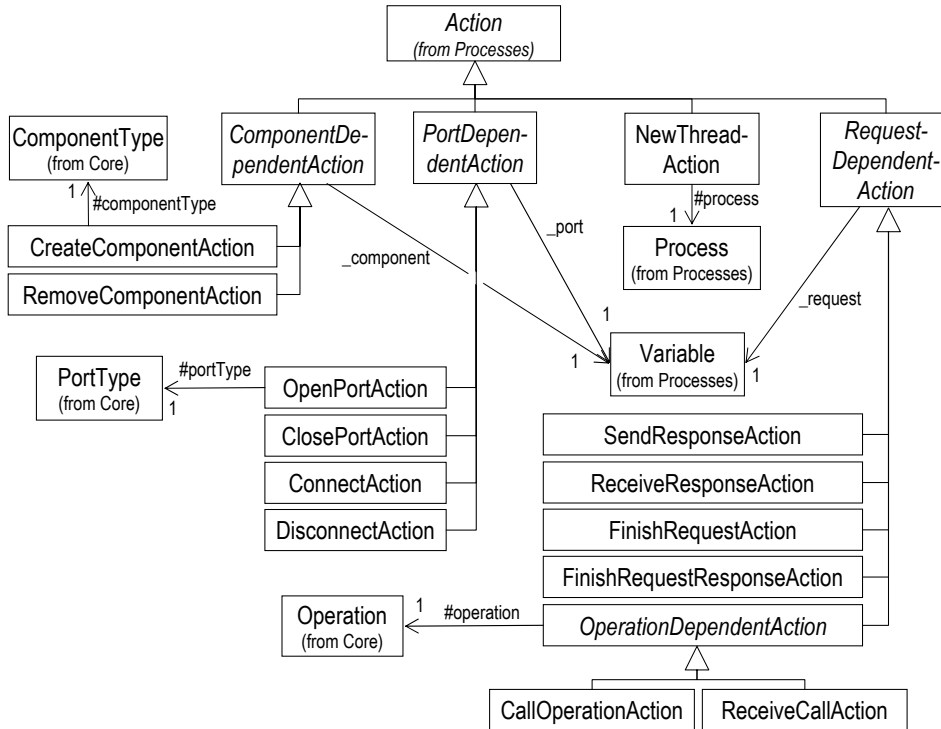


package: **Core**

description: This package contains the essential constructs for describing architectural configurations.

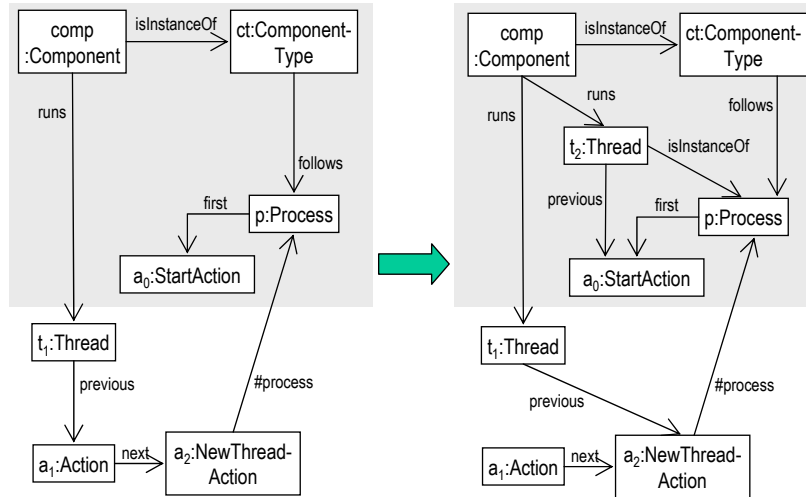


package: **Processes**  
 description: This package contains constructs for describing component behavior.



package: **Actions**  
 description: This package contains all action node types and their parameters. A detailed description is provided with the corresponding graph transformation rules.

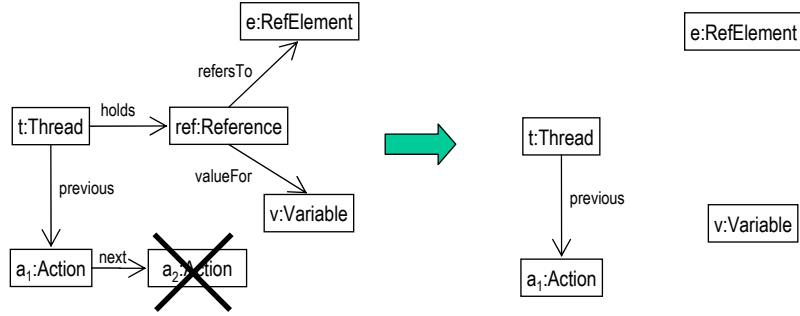
## A.2 Graph transformation rules



rule: **newThread**

description: Starts a new execution of a process by creating a new thread.

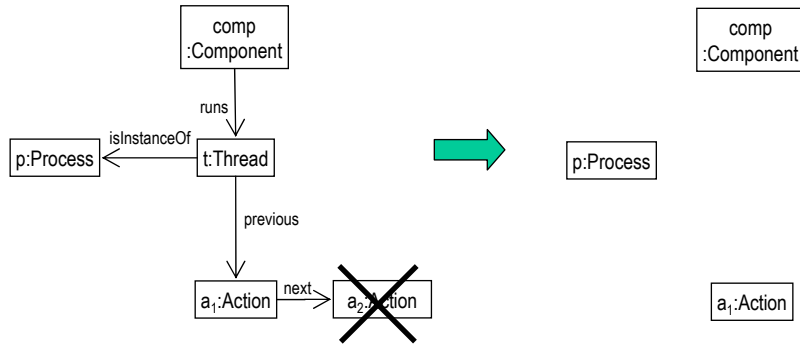
parameters: in **#process** (the process to be started)



rule: **clearReference**

description: Removes references (variable values) hold by finished threads.

parameters: none

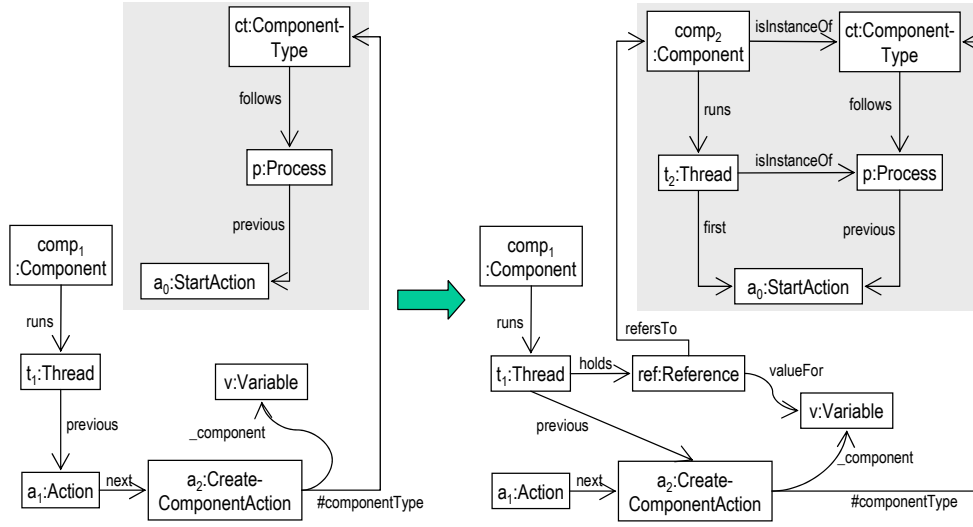


rule: **clearThread**

description: Removes remaining nodes of a finished thread.

parameters: none





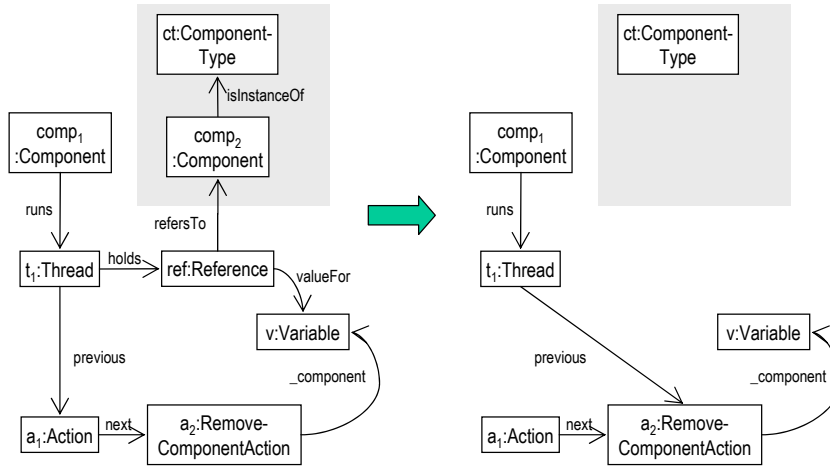
rule:

### createComponent

description: Creates a new instance of a certain component type.

parameters: in #componentType (the desired type of the new component)

out \_component (the newly created component)

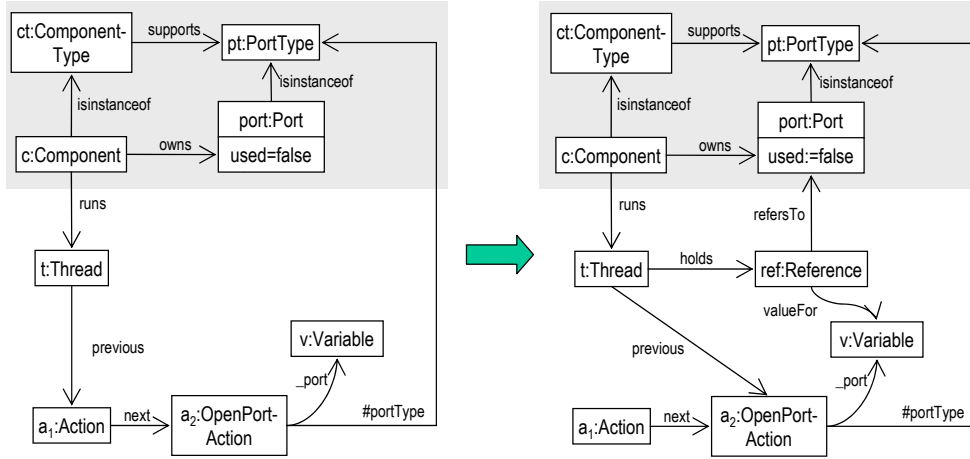


rule:

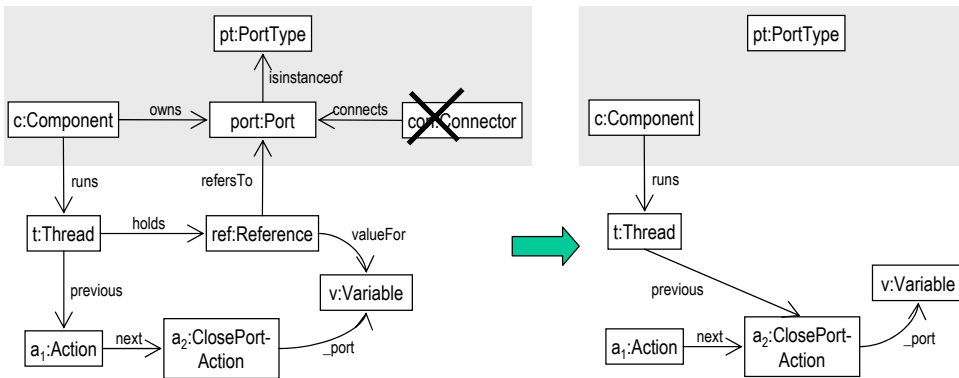
### removeComponent

description: Removes a certain component instance from the current configuration. The DPO dangling condition ensures that the rule is applied only after all ports of the component have been closed and all its threads have been finished.

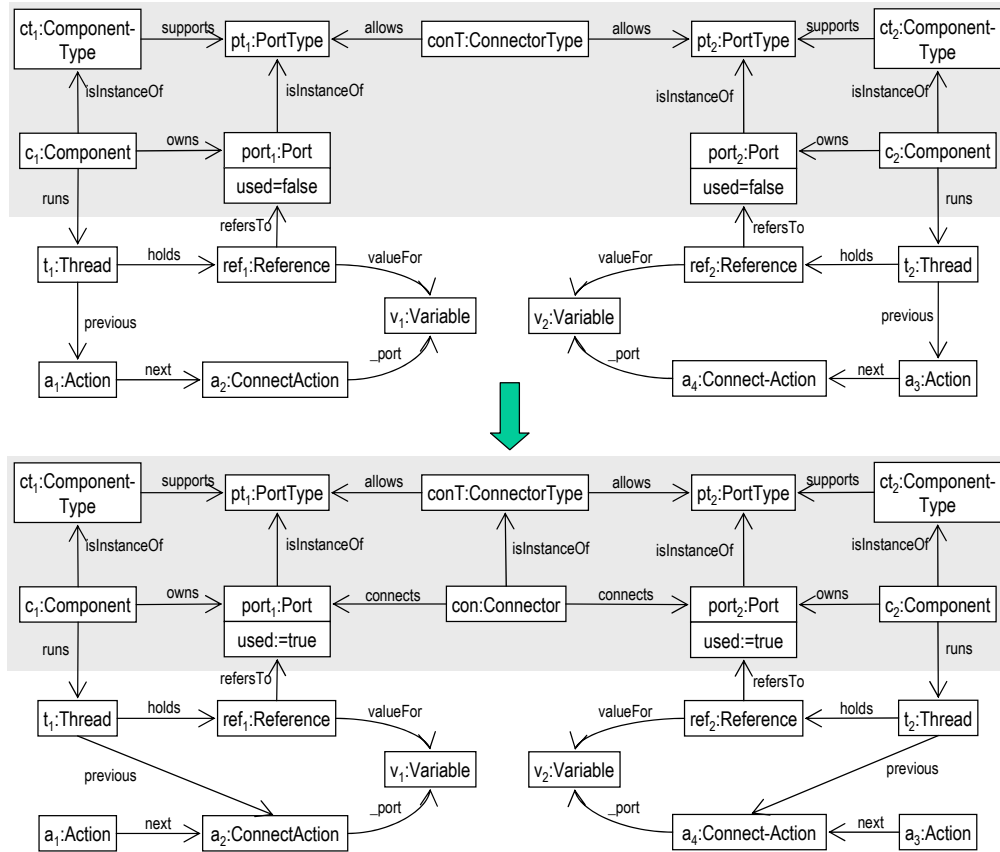
parameters: in \_component (the component to be removed)



rule: **openPort**  
 description: Opens a new port if there is no other free port left.  
 parameters: in **#portType** (the desired type of the new port)  
 out **\_port** (the newly created port)



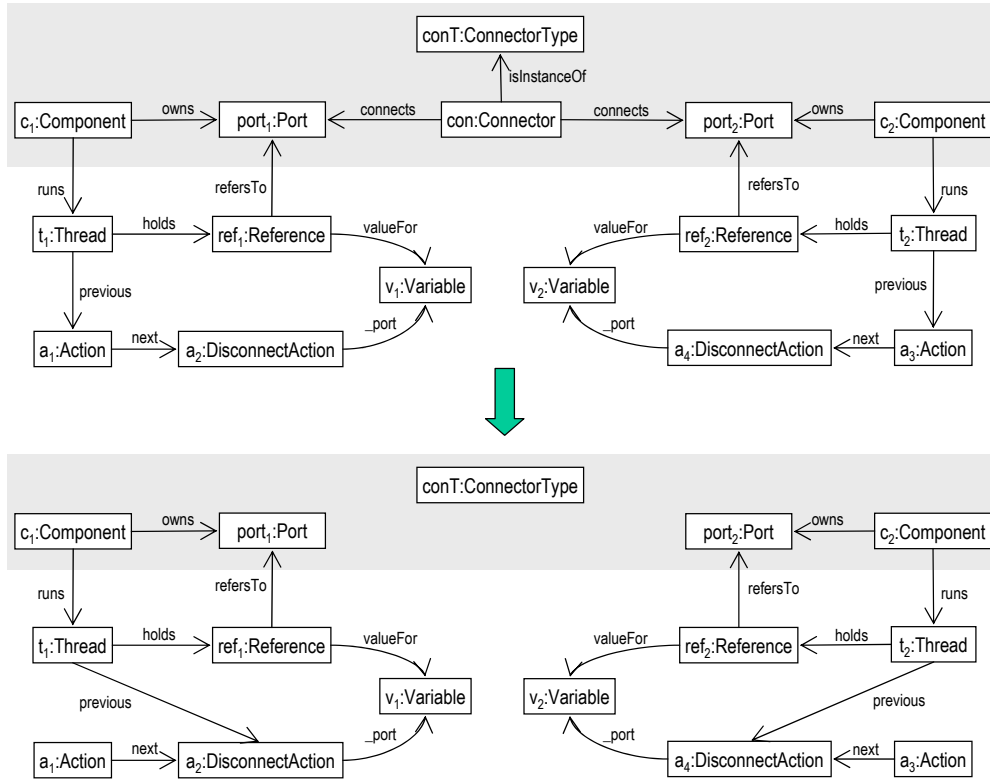
rule: **closePort**  
 description: Closes a port that is not in use any more.  
 parameters: in **\_port** (the port to be closed)



rule: **connect**

description: Places a connector between a port and the port of another component.

parameters: in `_port` (the port to be connected)



rule:

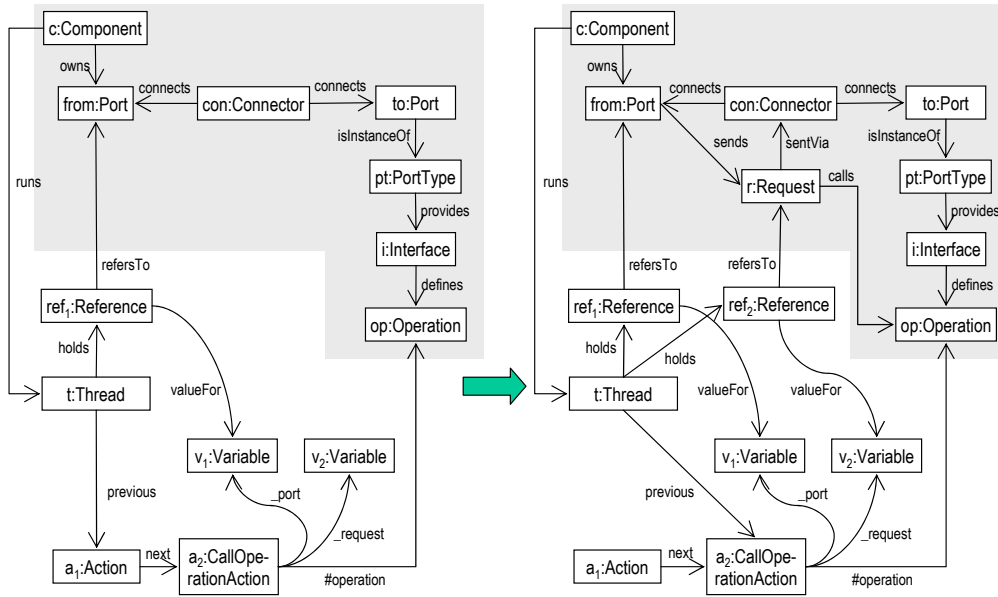
**disconnect**

description:

Removes a connector between two ports. The DPO dangling condition ensures that the rule is applied when no messages are currently sent via the connector.

parameters:

in  $port$  (the port to be disconnected)

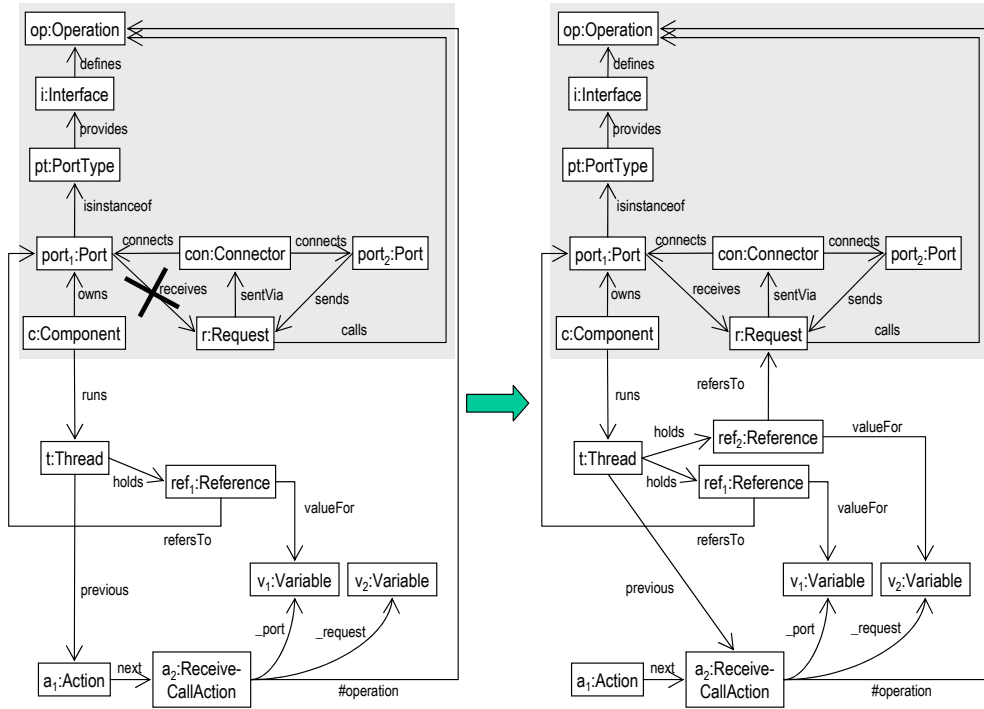


rule:

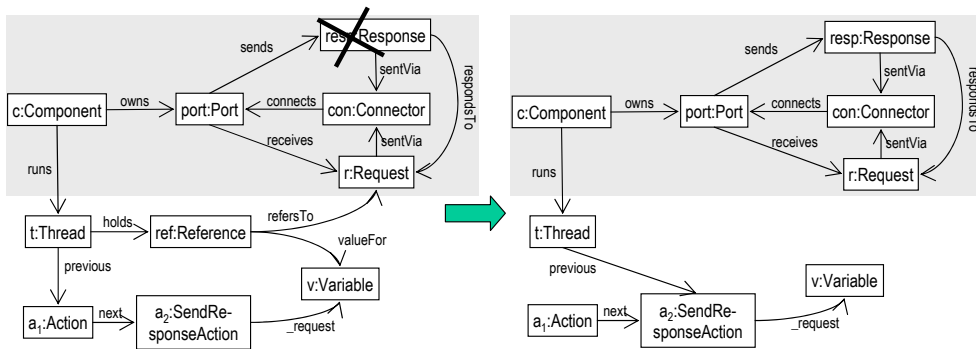
### callOperation

description: Sends a request to a connected port which provides a required operation.

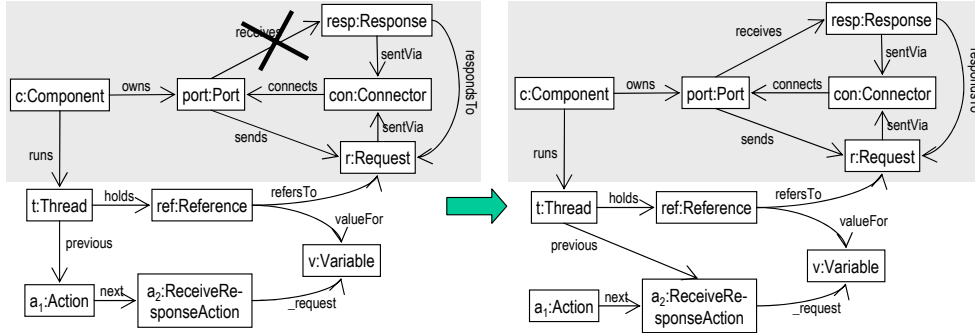
parameters: in `_port` (the port which shall send the request message)  
in `#operation` (the operation to be called)  
out `_request` (the resulting request message)



rule: **receiveCall**  
description: Delivers a request message to the target port.  
parameters: in `_port` (the port which expects an incoming request)  
in `#operation` (the operation which may be called)  
out `_request` (the received request message)



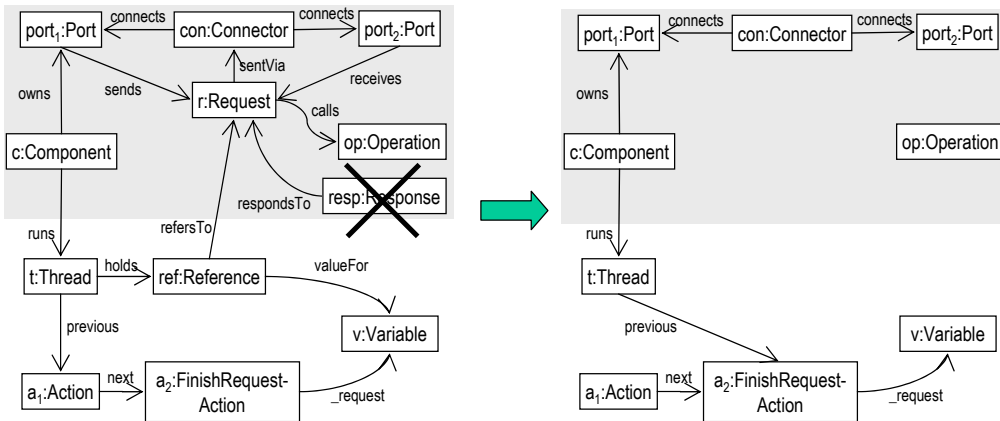
rule: **sendResponse**  
description: Sends the response message for a previous request.  
parameters: in `_request` (the previously received request message)



rule: **receiveResponse**

description: Delivers a response message to the requesting port.

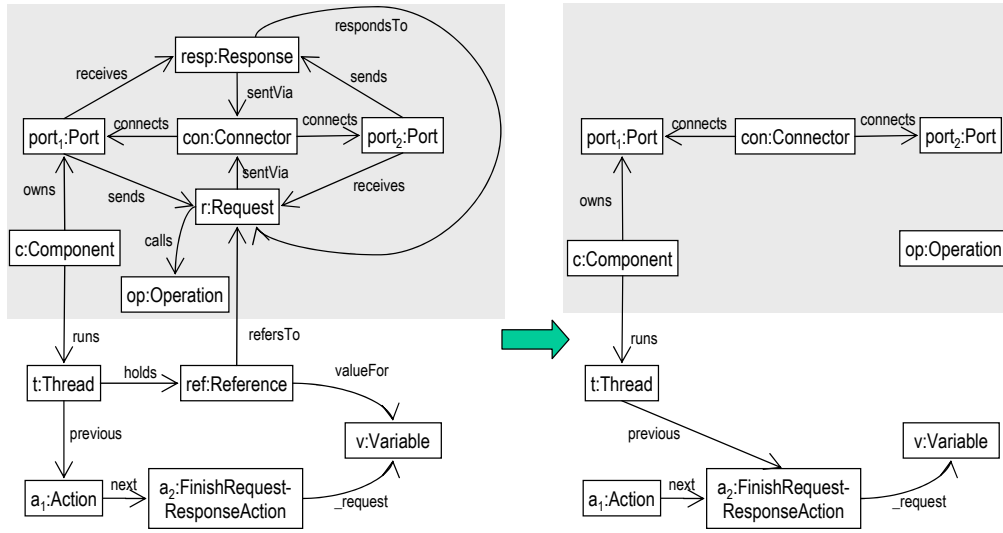
parameters: in `_request` (the request message the response refers to)



rule: **finishRequest**

description: Removes a request message after it has been processed without a response.

parameters: in `_request` (the request which been processed)



rule: **finishRequestResponse**

description: Removes a request message and the corresponding response after they have been processed.

parameters: in `_request` (the request which been processed)

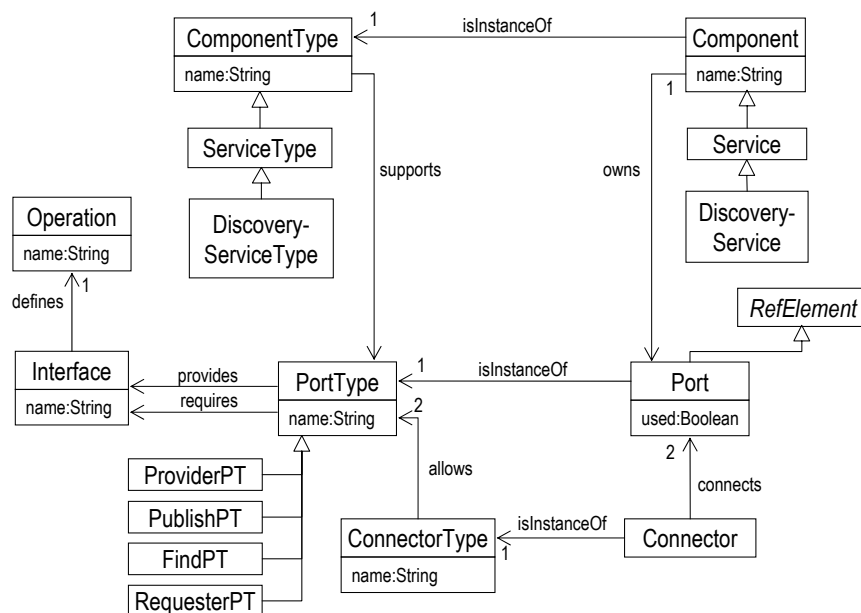


## Appendix B

# Architectural style for service-oriented architectures

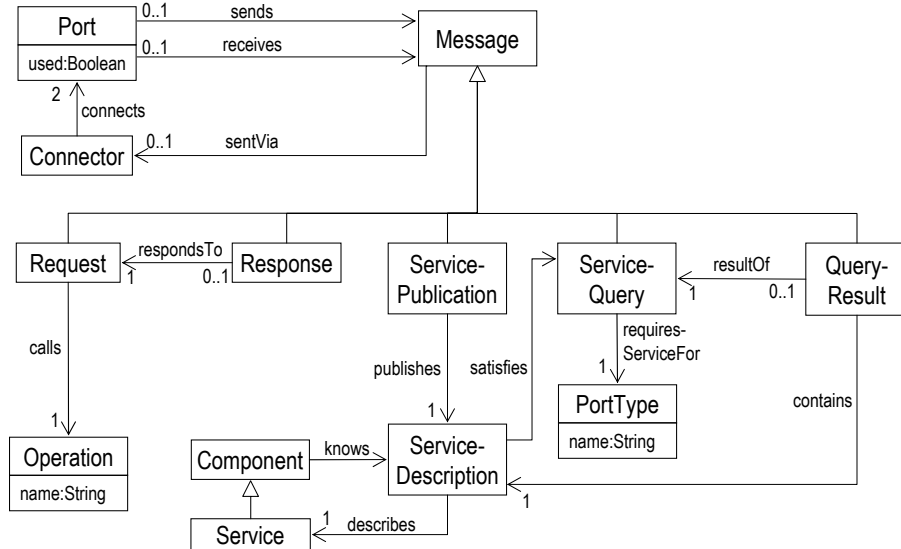
## B.1 Graph schema

Due to its size, the graph schema of the style is subdivided into the following four packages.



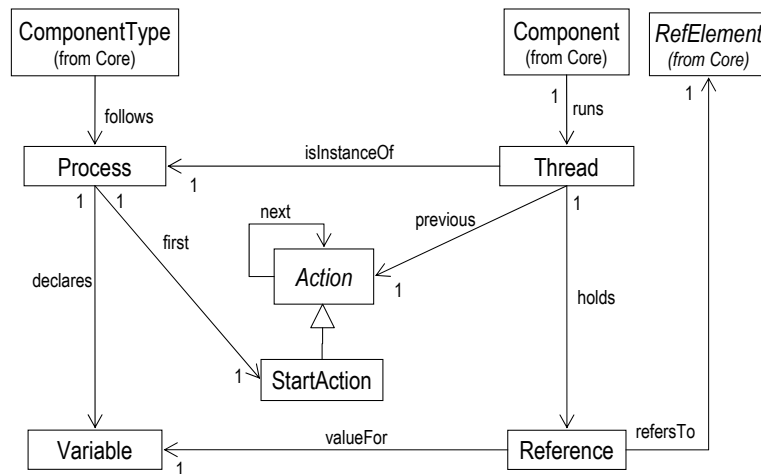
package: **Core**

description: This package contains the essential constructs for describing SOA-based configurations.



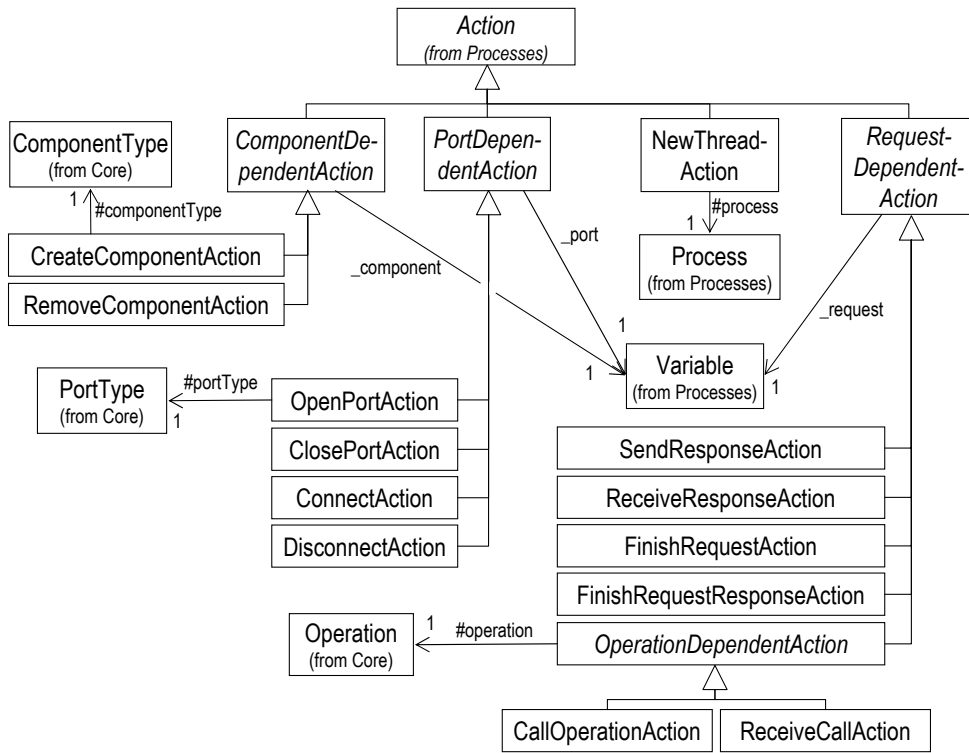
package: **Messages**

description: This package defines SOA-specific message types and their relationship to service descriptions.



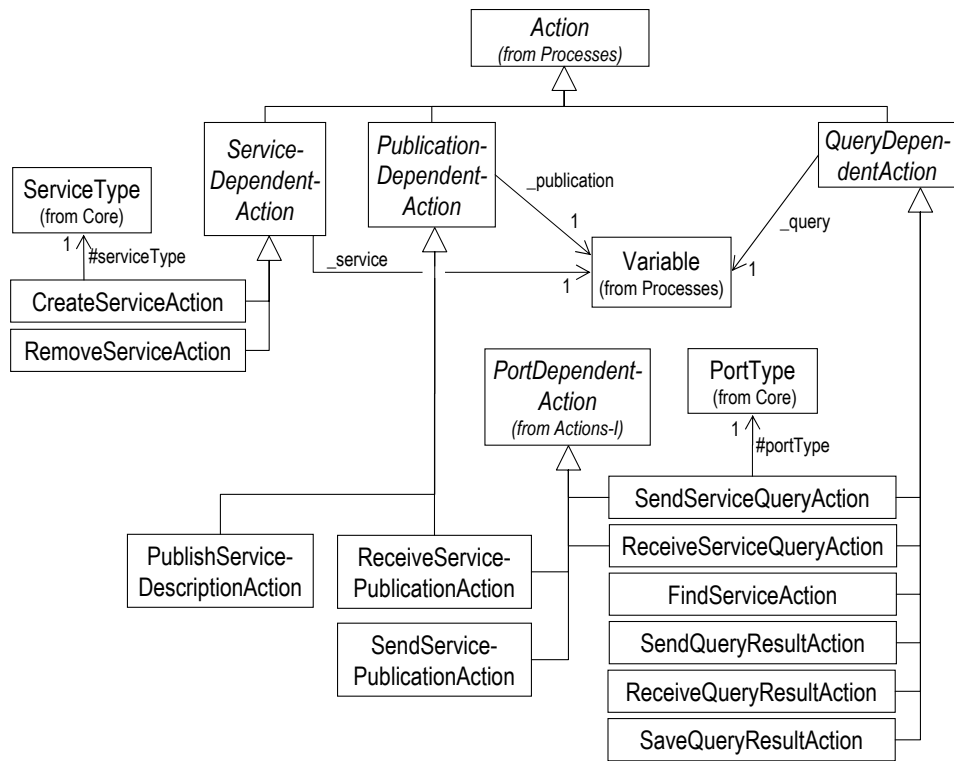
package: **Processes**

description: This package contains constructs for describing component behavior.



package: **Actions** (part I)

description: This package repeats the action node types and parameter edges inherited from the component-based style. A detailed description is provided with the corresponding graph transformation rules in [Appendix A](#).

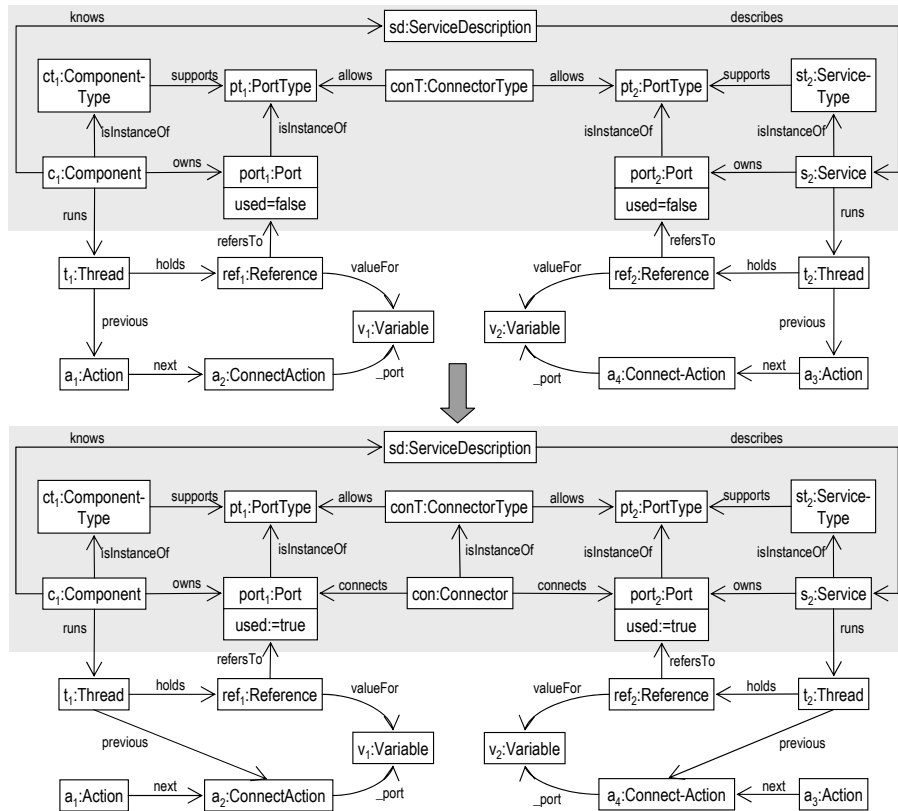


package: **Actions** (part II)

description: This second part of the package defines all SOA-specific action node types and their parameter edges. A detailed description is provided with the corresponding graph transformation rules in the following section.

## B.2 Graph transformation rules

The service-oriented style inherits all graph transformation rules from Appendix A, except for reconfiguration rule **connect** which is replaced by the following SOA-variant. In addition, the SOA style comprises all other SOA-specific rules presented thereafter.



rule:

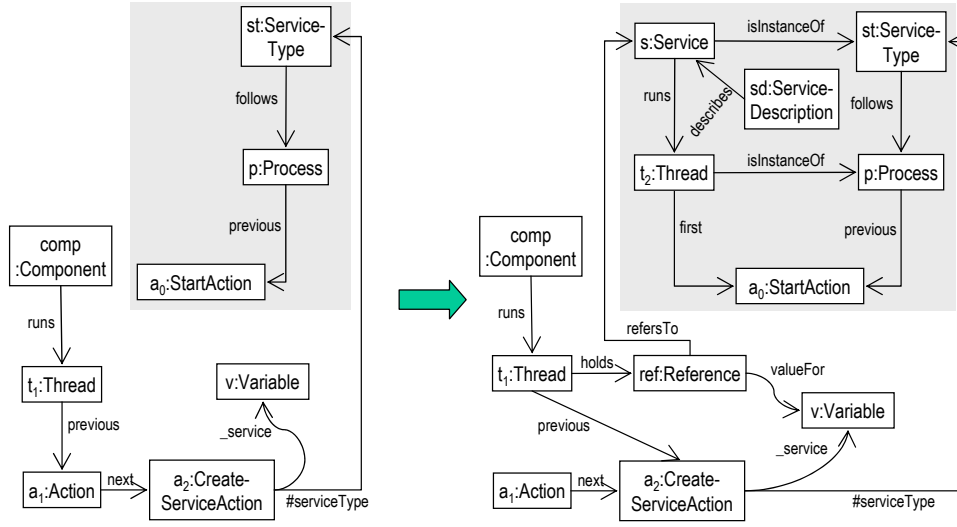
**connect**

description:

Places a connector between the ports of a **Component** and of a **Service**. This enables the component to use the service. The Component has to **know** the **ServiceDescription** beforehand. As **Service** is a subtype of **Component**, it is also possible that a service connects to another service. The requested service has to agree to the connection (synchronizing operation).

parameters:

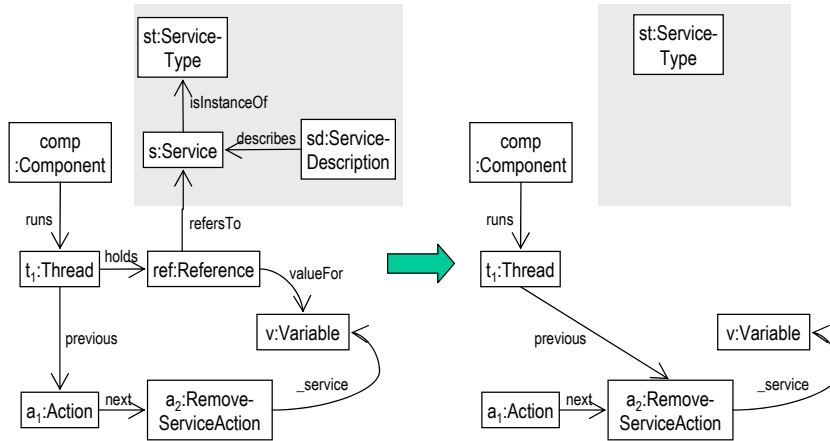
in `_port` (the port to be connected)



rule: **createService**

description: Creates a new instance of a certain service type including a service description.

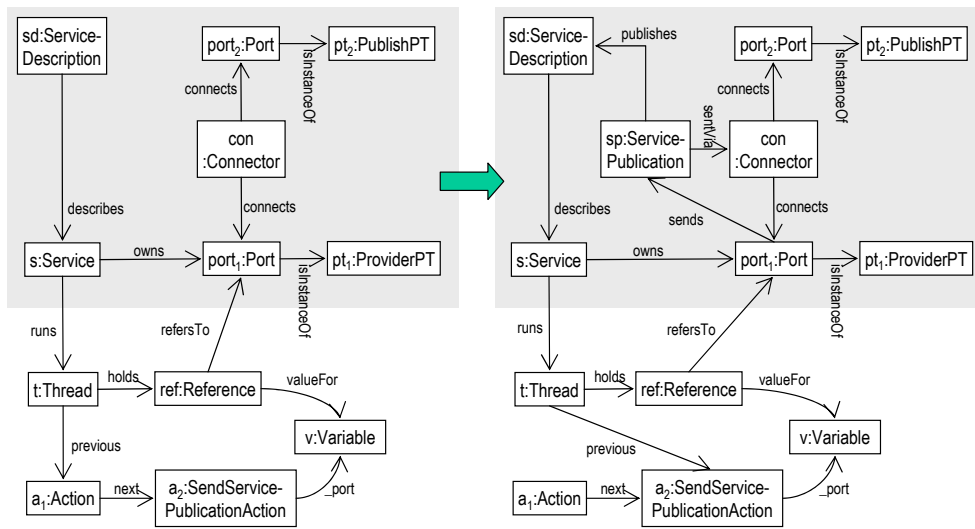
parameters: in **#serviceType** (the desired type of the new service)  
out **\_service** (the newly created service)



rule: **removeService**

description: Removes a certain service instance and its service description from the current configuration. The DPO dangling condition ensures that the rule is applied only after all ports of the service have been closed and all its threads have been finished.

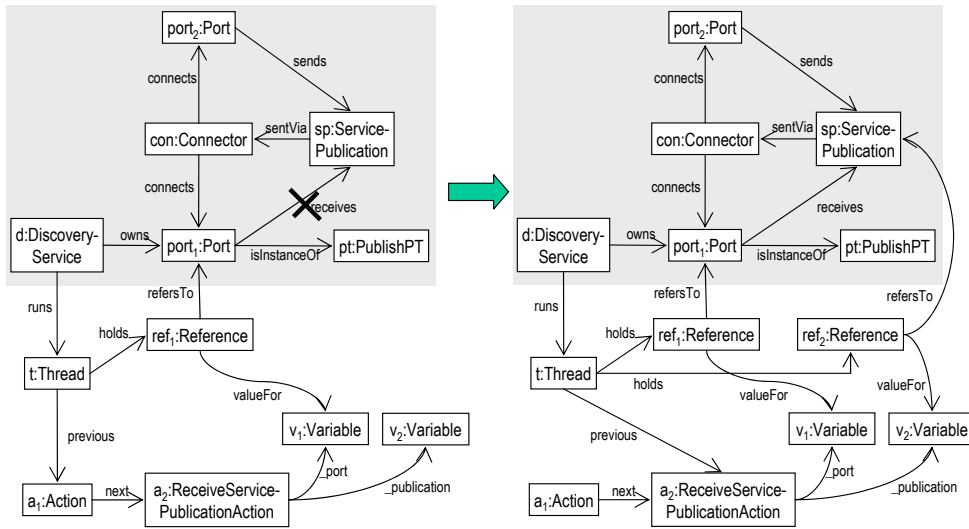
parameters: in **\_service** (the service to be removed)



rule: **sendServicePublication**

description: Sends a **ServicePublication** message to a discovery service as indicated by the **PublishPT** port. The message node is linked to the **ServiceDescription** of the responsible service.

parameters: in **\_port** (the port which shall send the message)



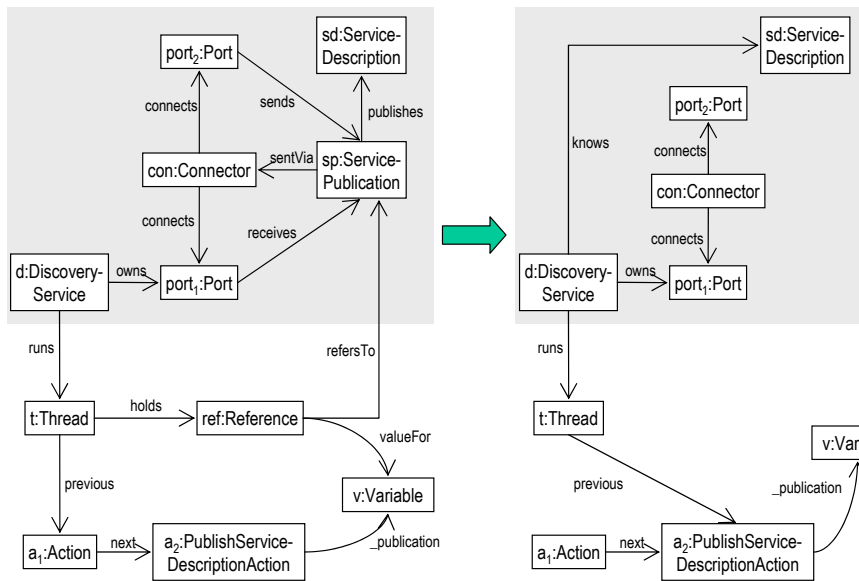
rule: **receiveServicePublication**

description: Models the receipt of a **ServicePublication** message by a discovery service.

parameters: in `_port` (the port which expects the incoming publication message)

out `_publication` (the received publication message)

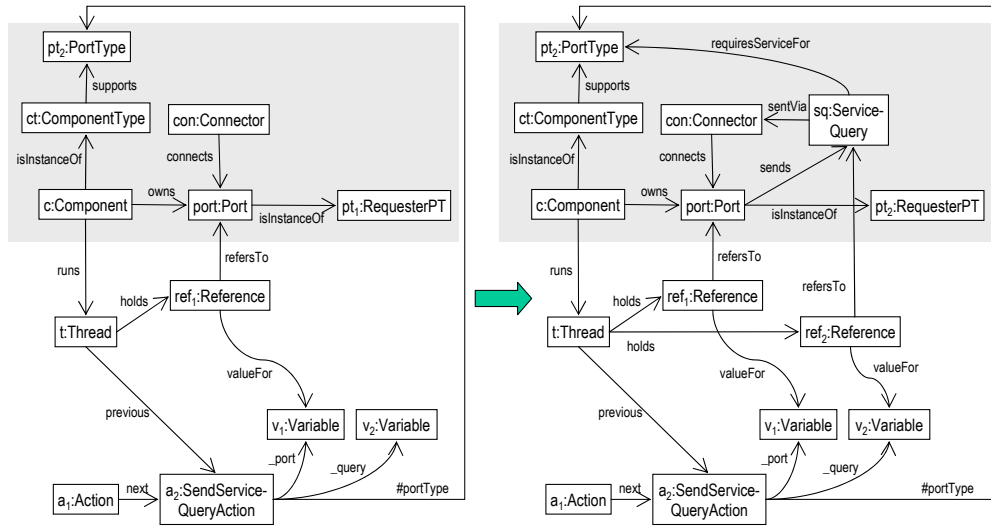




rule: **publishServiceDescription**

description: Safes a received service description at the discovery service and finishes the preceding communication for publication.

parameters: in `_publication` (the publication message which contains the service description)

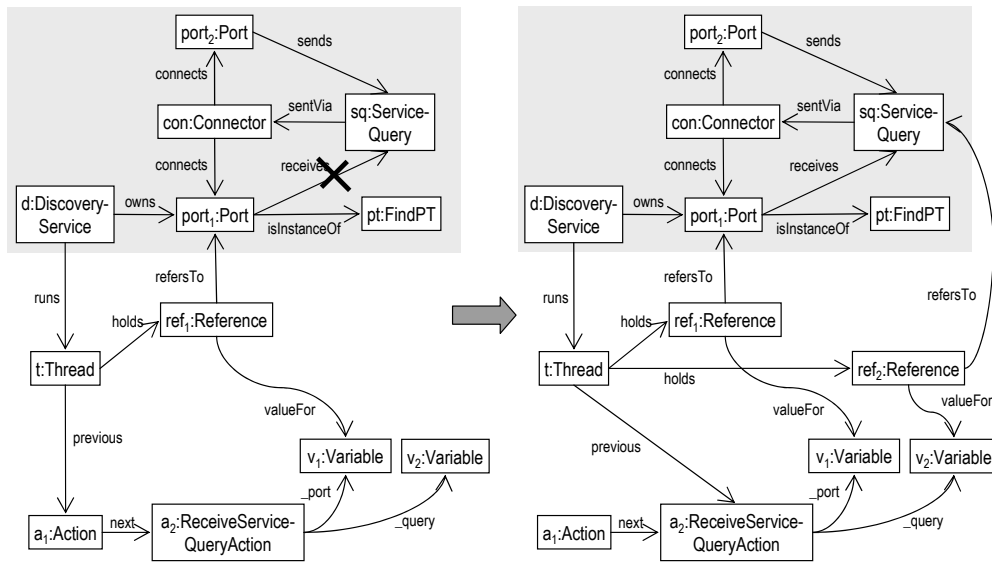


rule:

### sendServiceQuery

description: A Component sends a ServiceQuery message from its RequesterPT port. The query message points to a PortType which the component requires a suitable service for.

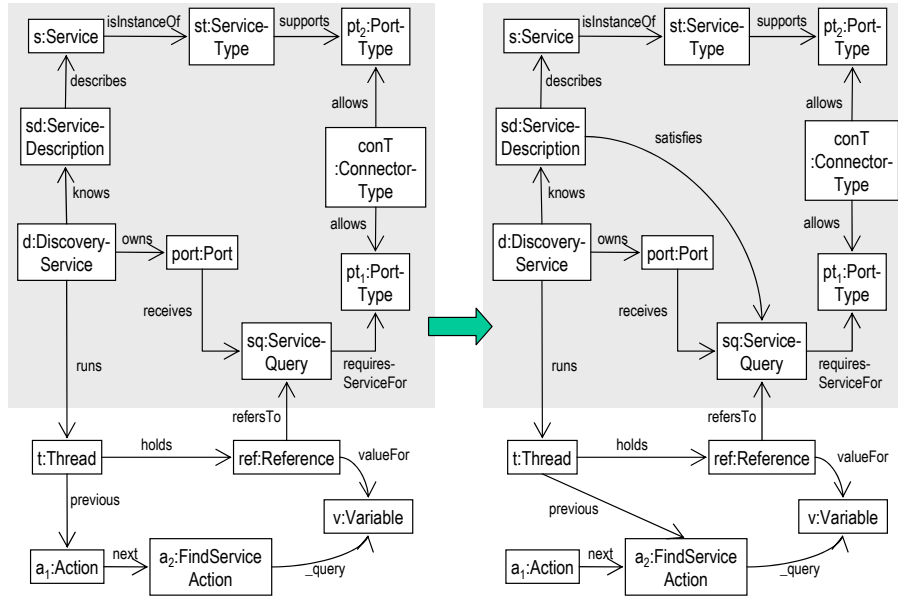
parameters: in `_port` (the port which shall send the query message)  
 in `#portType` (the port type which the component requires a service for)  
 out `_query` (the resulting query message)



rule: **receiveServiceQuery**

description: Models the receipt of a **ServiceQuery** message by a **DiscoveryService** through its **FindPT** port.

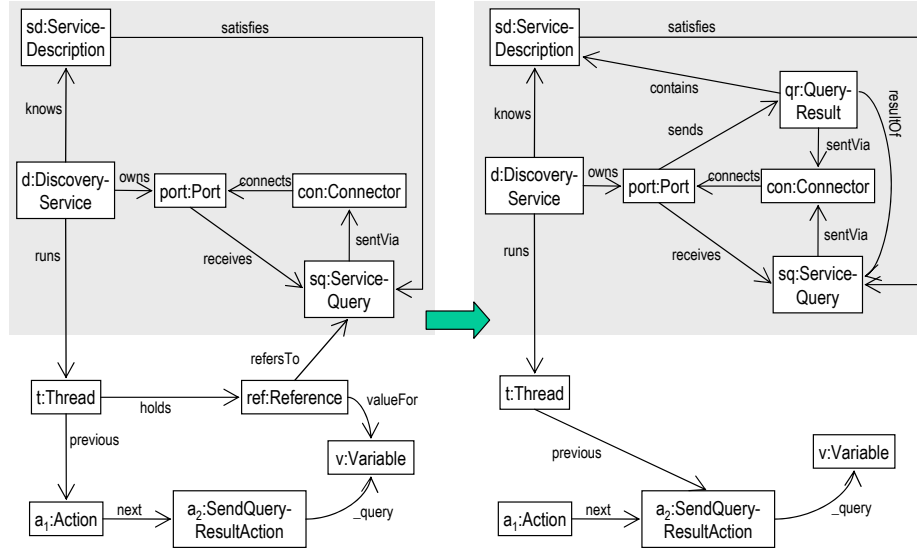
parameters: in **\_port** (the port which expects the incoming query)  
out **\_query** (the received query message)



rule: **findService**

description: Selects an appropriate **ServiceDescription**. A service satisfies a **ServiceQuery**, if there is a **ConnectorType** which allows to connect the port types of requester and service.

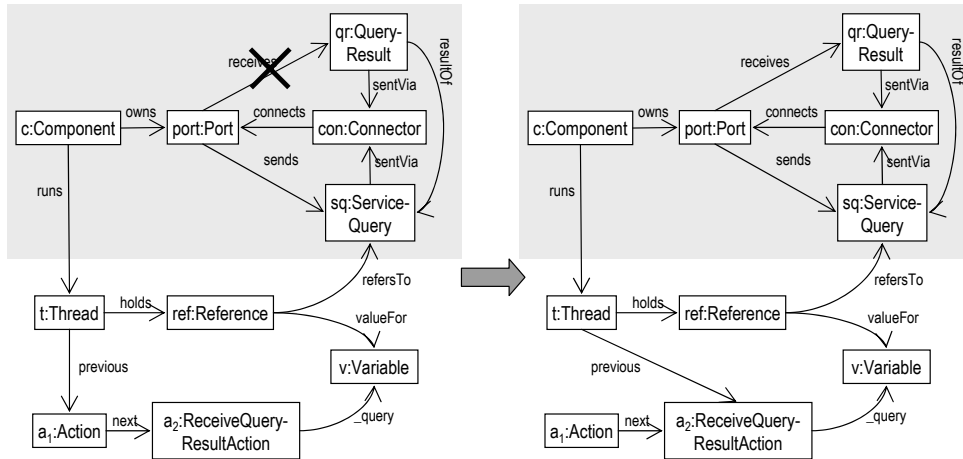
parameters: in `_query` (the query message which is linked to the requester port type)



rule: **sendQueryResult**

description: Sends a response message of type **QueryResult** from the **DiscoveryService** to the query originator.

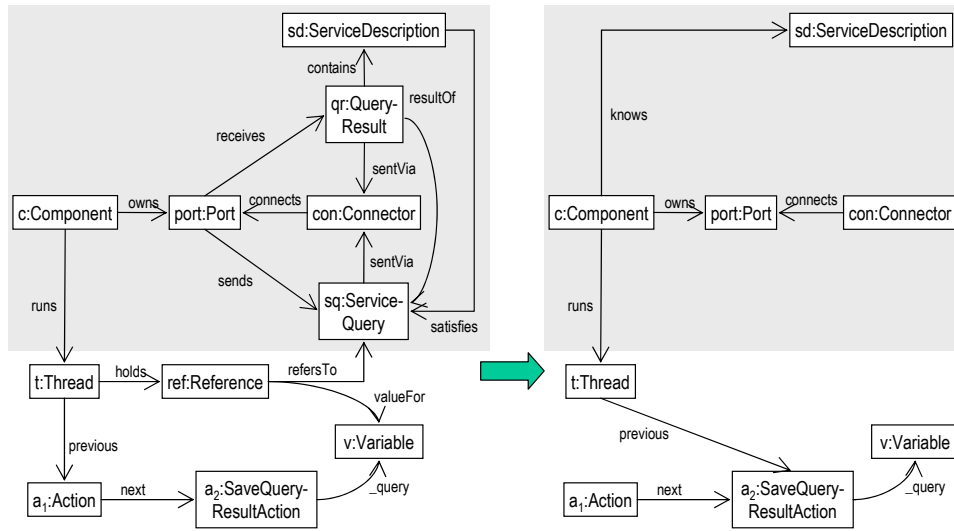
parameters: in `_query` (the previously received query message)



rule: **receiveQueryResult**

description: The service requester receives a **QueryResult** message.

parameters: in `_query` (the previously sent query message the result refers to)



rule: **saveQueryResult**

description: Finishes the query communication and saves the received **ServiceDescription** for the requesting **Component**.

parameters: in `_query` (the previously sent query message)

# Appendix C

## Abstract model of the travel agency architecture

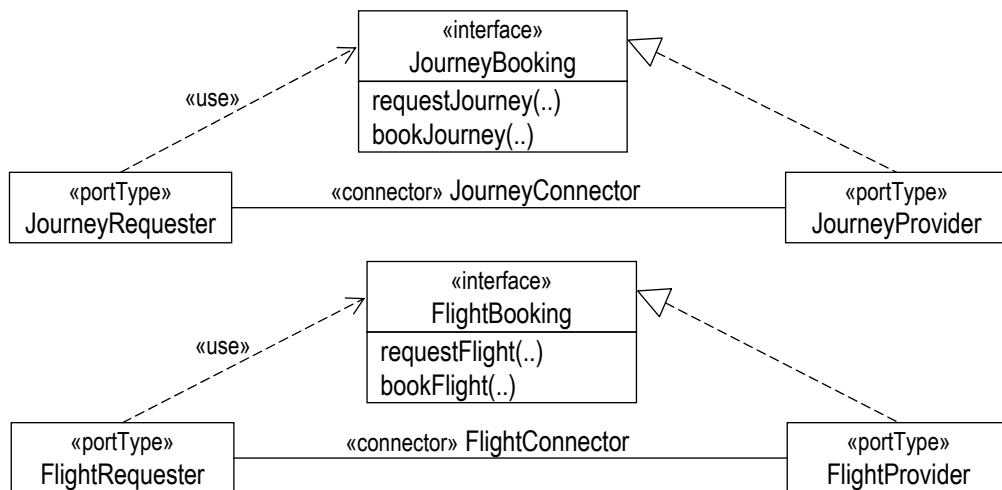


Figure C.1: Class diagram with interfaces, port types, and connector types

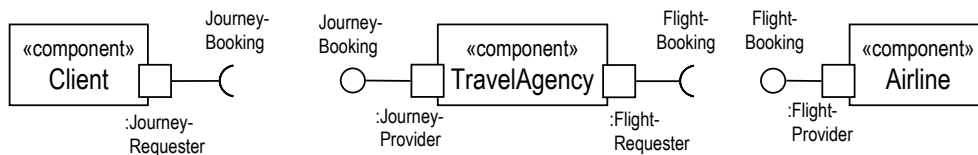


Figure C.2: Component diagram with component types

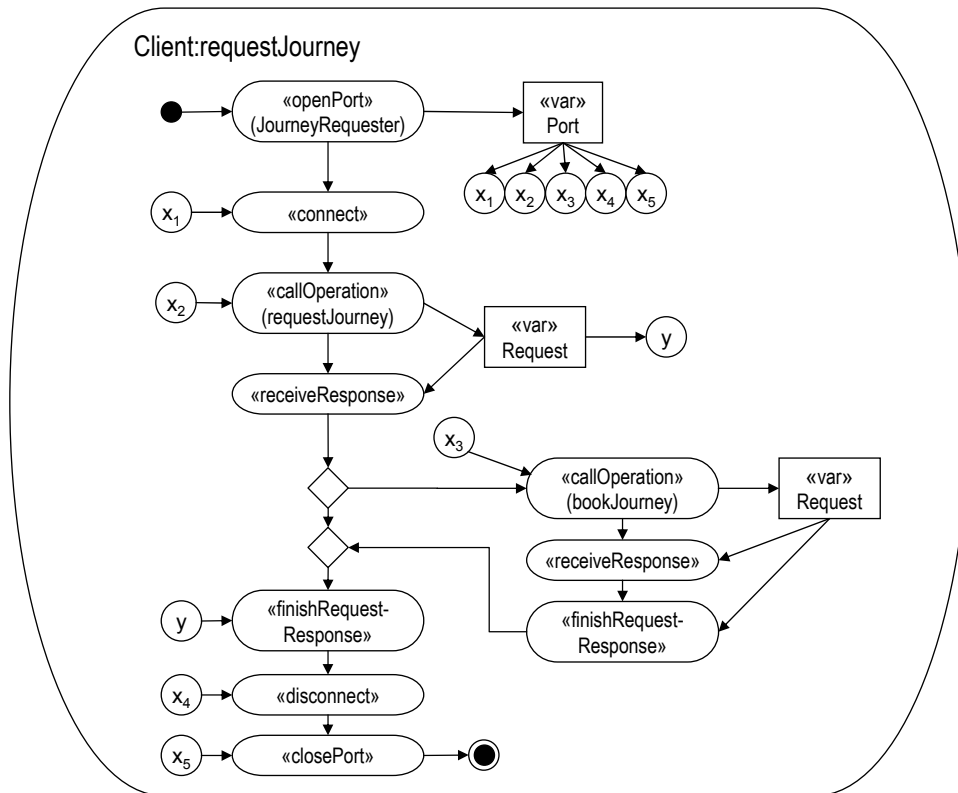


Figure C.3: Activity diagram for the client's process



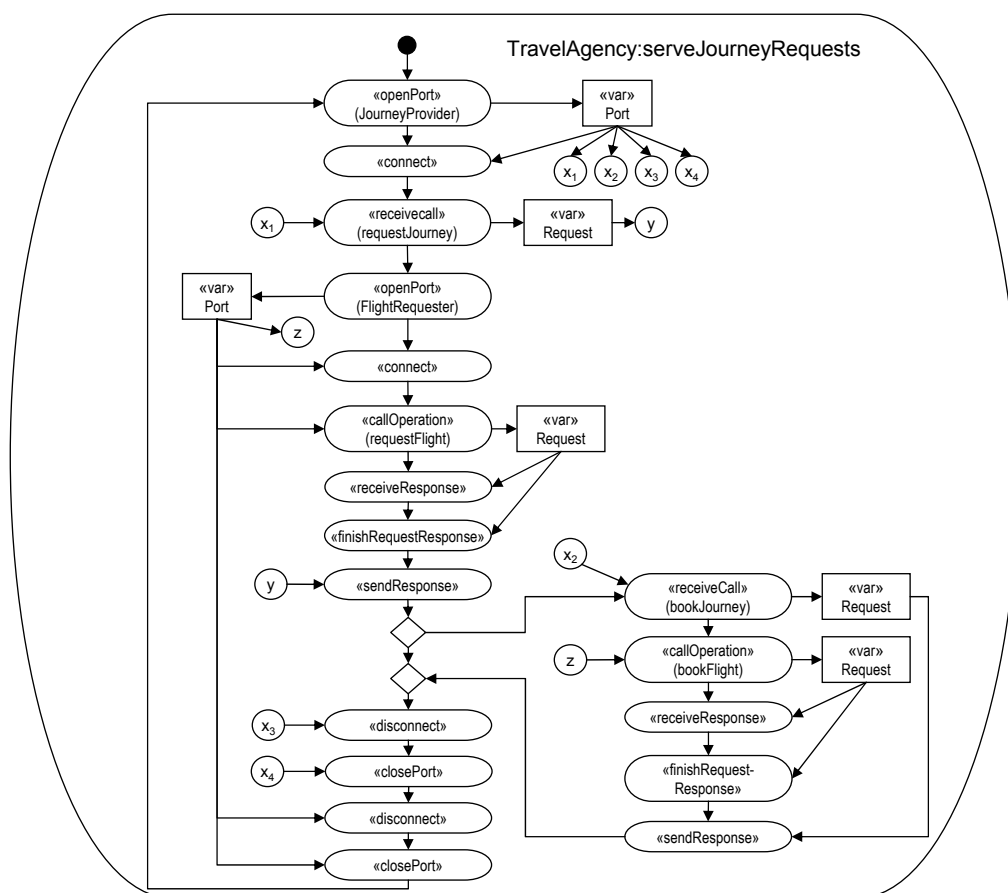


Figure C.4: Activity diagram for the travel agency's process

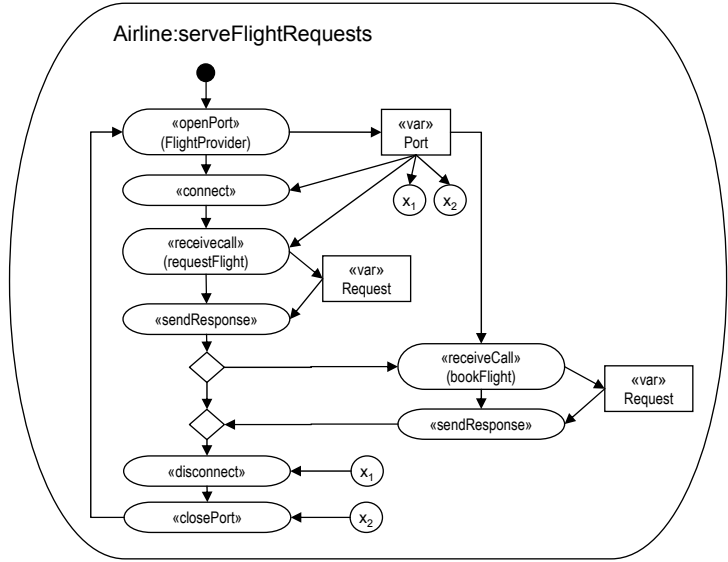


Figure C.5: Activity diagram for the airline’s process

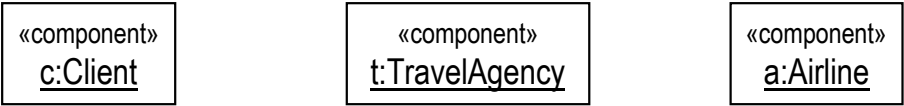


Figure C.6: Component diagram with the initial configuration

## Appendix D

### Concrete model of the travel agency architecture

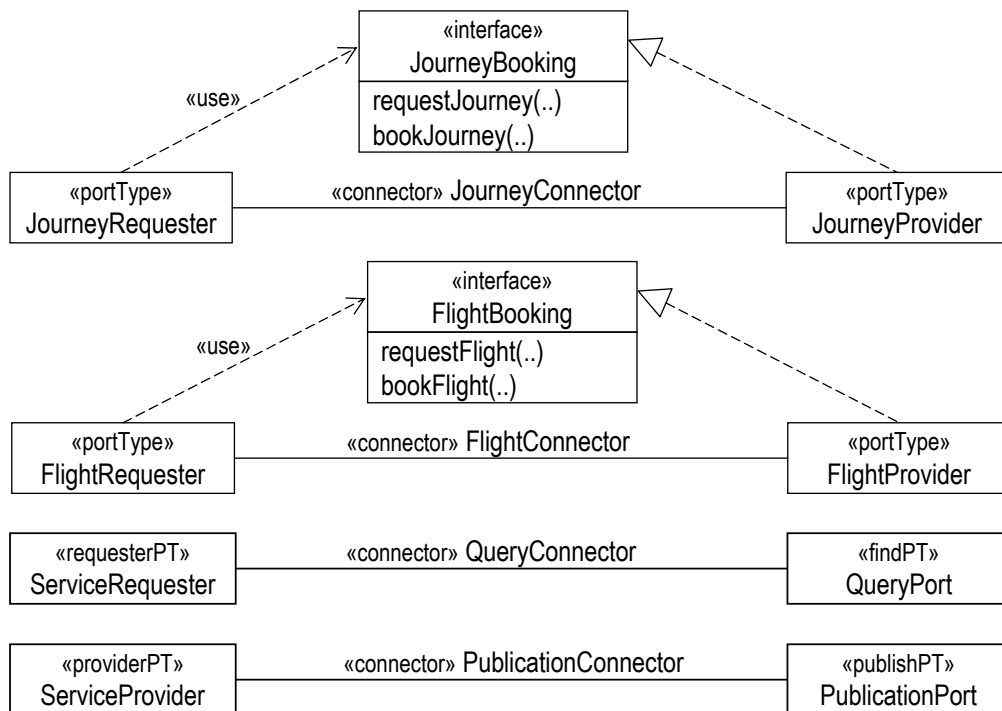


Figure D.1: Class diagram with interfaces, port types, and connector types

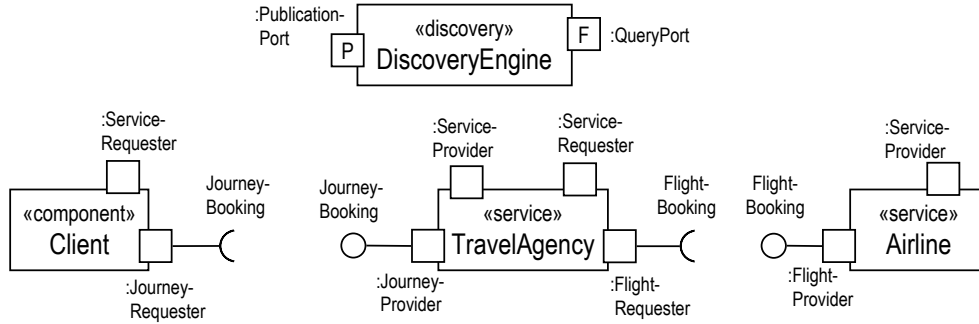


Figure D.2: Component diagram with component types

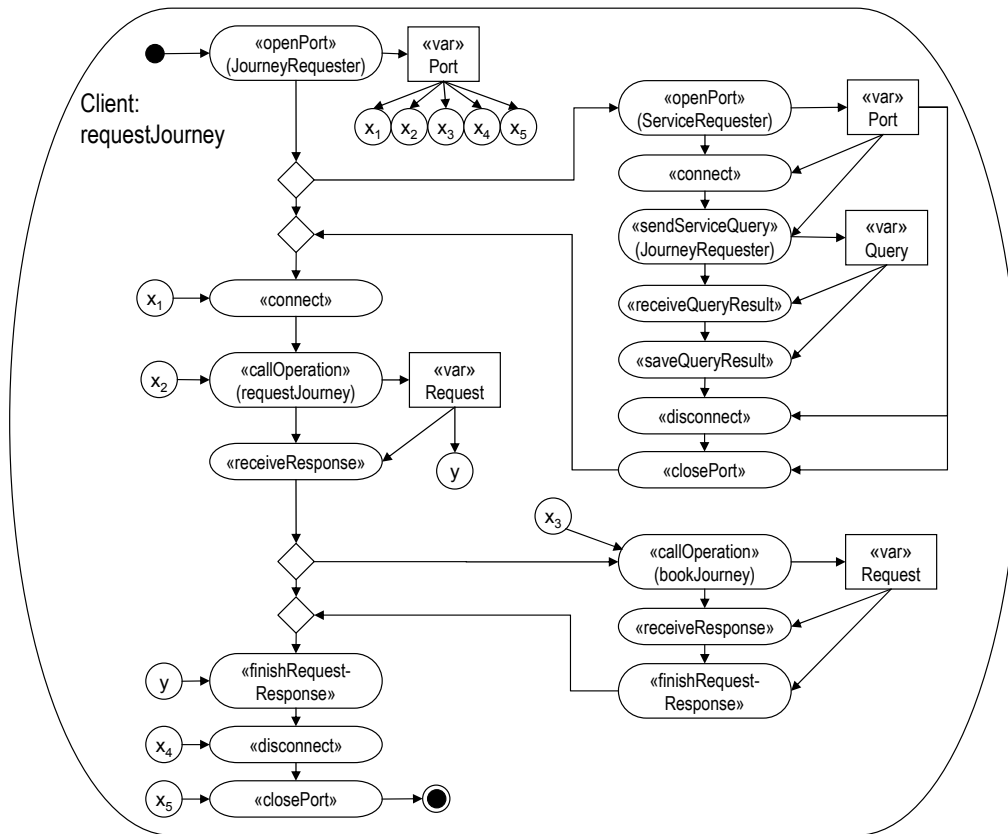


Figure D.3: Activity diagram for the client's process

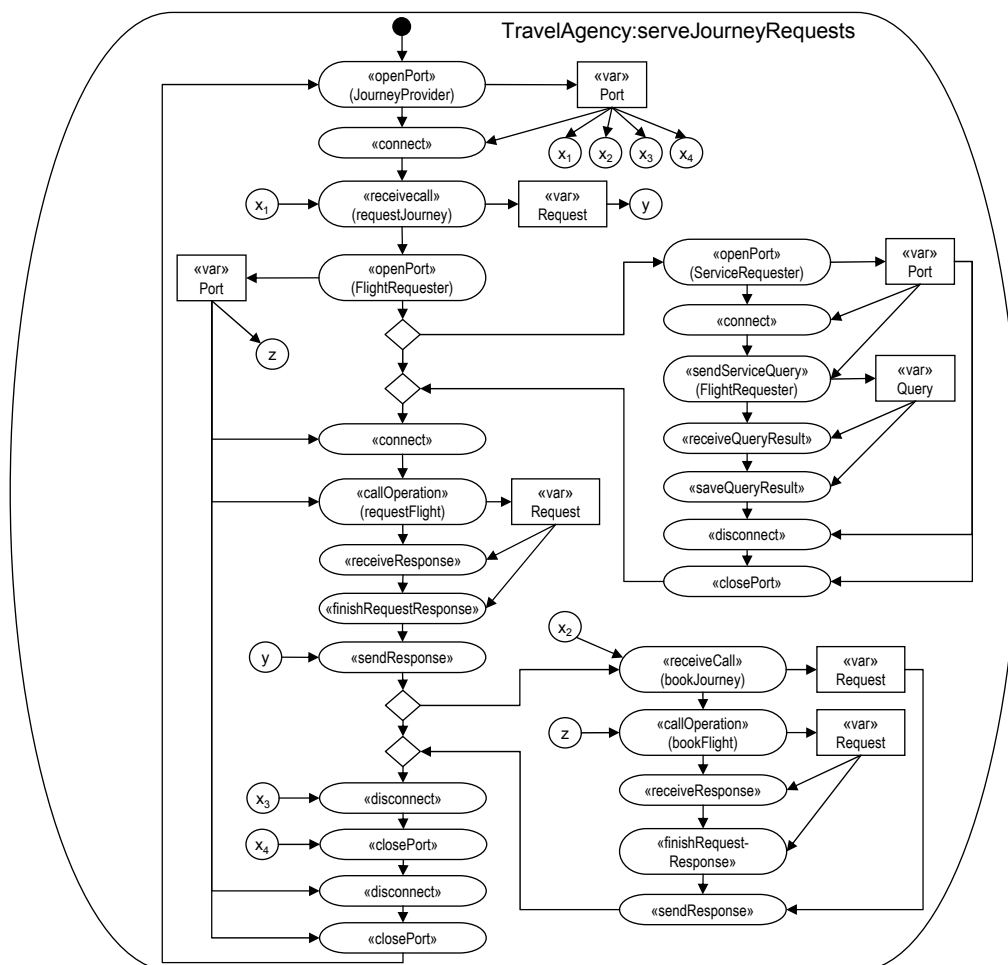


Figure D.4: Activity diagram for the travel agency's process

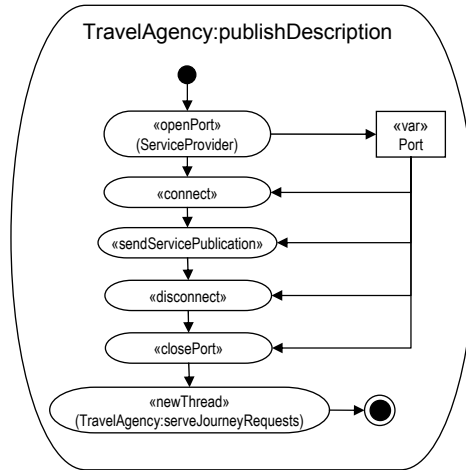


Figure D.5: Activity diagram for the travel agency's publication process

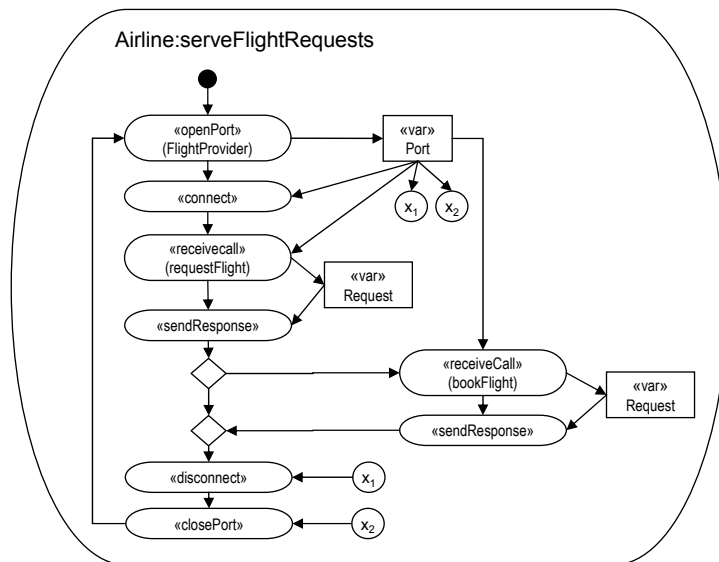


Figure D.6: Activity diagram for the airline's process

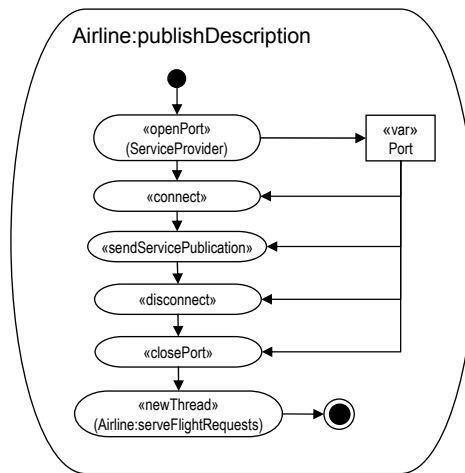


Figure D.7: Activity diagram for the airline's publication process

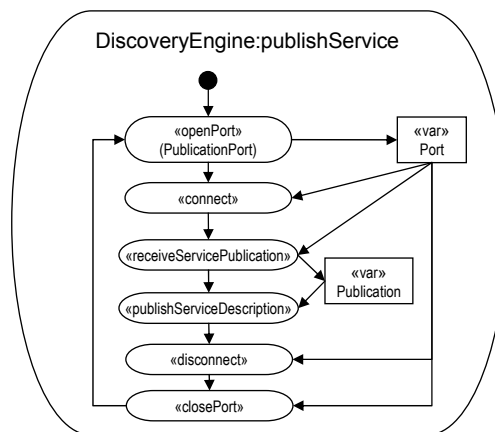


Figure D.8: Activity diagram for the discovery engine's publication process

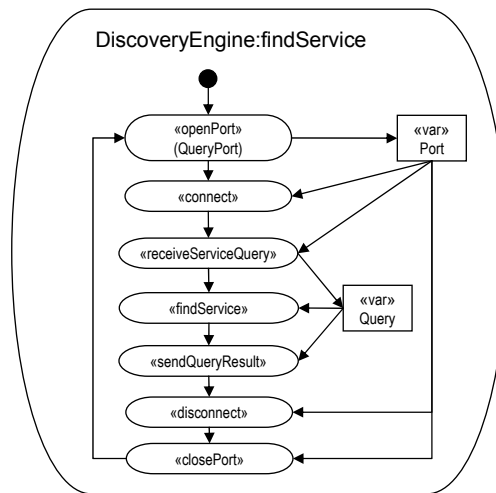


Figure D.9: Activity diagram for the discovery engine's query process

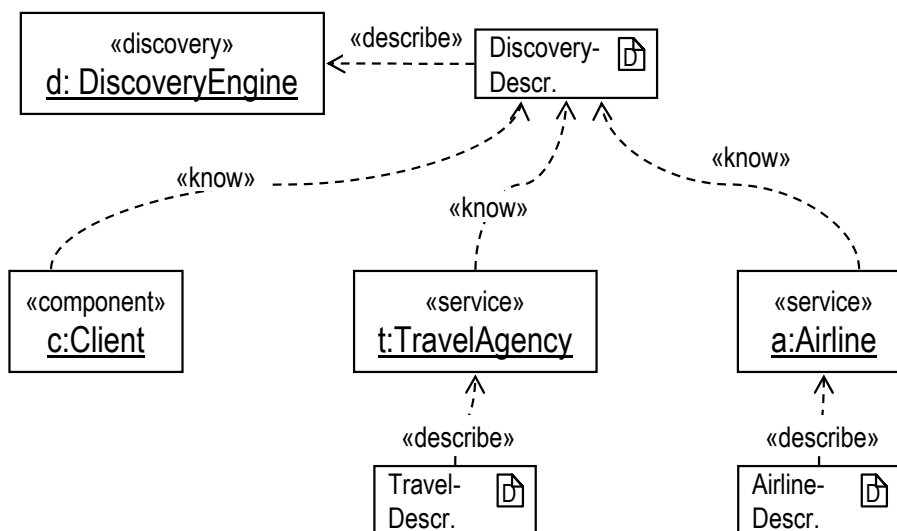


Figure D.10: Component diagram with the initial configuration



# List of Figures

1.1	Components involved in the electronic travel agency . . . . .	2
1.2	The electronic travel agency as dynamic architecture . . . . .	8
1.3	Complete specification of dynamic software architectures [20] .	10
1.4	A platform model and its instantiation . . . . .	13
1.5	Hierarchy of platform models . . . . .	15
1.6	Reusable refinement relationship between platform models . .	19
2.1	Style-based refinement relationships . . . . .	45
3.1	Style-based modeling – an overview . . . . .	49
3.2	Transformation rule examples for component (dis-)connection	51
3.3	Transition system as operational architecture model . . . . .	53
3.4	UML component diagram . . . . .	54
3.5	UML component diagram in a service-oriented style . . . . .	55
3.6	UML activity diagrams . . . . .	57
3.7	Refinement vs. abstraction . . . . .	59
3.8	Effect of an abstraction function . . . . .	60
3.9	State-based refinement check of transition system paths . . . .	62
4.1	Exemplary type graph as UML class diagram . . . . .	65
4.2	Instance graph as UML object diagram . . . . .	66
4.3	Type graph with inheritance as UML class diagram . . . . .	67
4.4	Instance graphs typed by clan morphisms . . . . .	68
4.5	Attributed type graph . . . . .	70
4.6	Attributed instance graph in formal and UML-like syntax . . .	72
4.7	Type graph with cardinalities . . . . .	73
4.8	Graph transformation rule and its application . . . . .	76
4.9	Negative application condition . . . . .	78
4.10	Algebraic constituents of a graph transformation rule . . . . .	79
4.11	Double Pushout Diagram o . . . . .	81
4.12	Pushout in the category <b>Graph</b> . . . . .	82

4.13	Construction of a pushout . . . . .	83
4.14	Two exemplary graph transformation rules . . . . .	90
4.15	Transformation sequence . . . . .	90
4.16	Induced graph transition system . . . . .	90
5.1	Graph schema of the component-based style . . . . .	95
5.2	Instance graph for the travel system architecture . . . . .	95
5.3	Reconfiguration rule <b>openPort</b> . . . . .	98
5.4	Reconfiguration rule <b>connect</b> . . . . .	98
5.5	Reconfiguration scenario as transformation sequence . . . . .	99
5.6	Graph schema extensions for component communication . . . . .	100
5.7	Communication rule <b>callOperation</b> . . . . .	101
5.8	Communication rule <b>receiveCall</b> . . . . .	102
5.9	Control flow realized by processes and action counter . . . . .	104
5.10	Graph schema extensions concerning control flow . . . . .	104
5.11	Control flow-enabled reconfiguration rule <b>openPort</b> . . . . .	105
5.12	Graph schema extensions concerning variable action parameters . . . . .	107
5.13	Parameter definitions for the <b>CallOperationAction</b> . . . . .	108
5.14	Parameter-enabled transformation rule <b>callOperation</b> . . . . .	108
5.15	Rule <b>connect</b> enforcing thread synchronization . . . . .	109
5.16	Thread management rule <b>newThread</b> . . . . .	110
5.17	Thread management rule <b>clearReference</b> . . . . .	111
5.18	Thread management rule <b>clearThread</b> . . . . .	111
5.19	Roles in a service-oriented architecture, cf. [23] . . . . .	113
5.20	Graph schema of the service-oriented style . . . . .	114
5.21	Graph schema part for communication in SOA . . . . .	115
5.22	SOA-specific reconfiguration rule <b>sendServicePublication</b> . . . . .	116
5.23	SOA-specific reconfiguration rule <b>findService</b> . . . . .	117
5.24	SOA-specific reconfiguration rule <b>saveQueryResult</b> . . . . .	117
6.1	Stereotype definitions for the SOA-specific UML profile . . . . .	121
6.2	Class diagram in the SOA-specific profile . . . . .	121
6.3	Behavior-related stereotype definitions . . . . .	125
6.4	Activity diagram for <b>Client</b> in the SOA-specific style . . . . .	126
6.5	Activity diagram for <b>TravelAgency</b> in the SOA-specific style . . . . .	128
6.6	Translation chain from UML diagrams to graph transition systems . . . . .	130
6.7	Triple graph production . . . . .	133
6.8	Unidirectional triple graph production . . . . .	134
7.1	Some part of a type graph mapping [14] . . . . .	143

7.2	Abstraction of an instance graph . . . . .	145
7.3	Triple graph production as part of an abstraction function . .	146
7.4	Preservation of an abstract transformation step . . . . .	149
7.5	Example of abstract and concrete graph transition systems . .	151
7.6	Algorithm for behavior preservation based on [66] . . . . .	153
7.7	Substitution of an abstract transformation step . . . . .	157
7.8	Algorithm for observational substitutability based on [66] . . .	160
7.9	Reachability analysis with special predicate rules . . . . .	165
8.1	Activities of style and application architects and required tools	172
8.2	Rule extension to ensure DPO dangling condition in SPO tools	176
8.3	Data flow in an integrated tool environment . . . . .	185
8.4	Screenshot of a transformation rule in the GROOVE simulator	188
8.5	Screenshot of a graph transition system generated by GROOVE	189
8.6	Screenshot of a proposition rule in GROOVE . . . . .	191
8.7	Screenshot of the test transition system in GROOVE . . . . .	192
C.1	Class diagram with interfaces, port types, and connector types	243
C.2	Component diagram with component types . . . . .	243
C.3	Activity diagram for the client's process . . . . .	244
C.4	Activity diagram for the travel agency's process . . . . .	245
C.5	Activity diagram for the airline's process . . . . .	246
C.6	Component diagram with the initial configuration . . . . .	246
D.1	Class diagram with interfaces, port types, and connector types	247
D.2	Component diagram with component types . . . . .	248
D.3	Activity diagram for the client's process . . . . .	248
D.4	Activity diagram for the travel agency's process . . . . .	249
D.5	Activity diagram for the travel agency's publication process .	250
D.6	Activity diagram for the airline's process . . . . .	250
D.7	Activity diagram for the airline's publication process . . . . .	251
D.8	Activity diagram for the discovery engine's publication process	251
D.9	Activity diagram for the discovery engine's query process . . .	252
D.10	Component diagram with the initial configuration . . . . .	252



# List of Tables

2.1	Comparison of approaches to dynamic architecture description	36
2.2	Comparison of approaches to platform-aware description . . .	40
2.3	Comparison of approaches to architecture refinement . . . . .	46
6.1	Notation guide for SOA-specific stereotypes . . . . .	123
6.2	Mapping between SOA style and UML meta-model . . . . .	132
7.1	Type graph mapping for all nodes of $TG'$ . . . . .	144
8.1	Comparison of graph transformation tools . . . . .	183
8.2	Start graph sizes of the two travel agency models . . . . .	188
8.3	Graph transition systems of the two travel agency models . . .	190