Modeling and Verifying Dynamic Evolving Service-Oriented Architectures

Holger Giese, Basil Becker

Technische Berichte Nr. 75

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 75

Holger Giese | Basil Becker

Modeling and Verifying Dynamic Evolving Service-Oriented Architectures

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.de/ abrufbar.

Universitätsverlag Potsdam 2013
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2013/6511/
URN urn:nbn:de:kobv:517-opus-65112
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-65112

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-246-9

Abstract The service-oriented architecture supports the dynamic assembly and runtime reconfiguration of complex open IT landscapes by means of runtime binding of service contracts, launching of new components and termination of outdated ones. Furthermore, the evolution of these IT landscapes is not restricted to exchanging components with other ones using the same service contracts, as new services contracts can be added as well. However, current approaches for modeling and verification of service-oriented architectures do not support these important capabilities to their full extend. In this report we present an extension of the current OMG proposal for service modeling with UML — *SoaML* — which overcomes these limitations. It permits modeling services and their service contracts at different levels of abstraction, provides a formal semantics for all modeling concepts, and enables verifying critical properties. Our compositional and incremental verification approach allows for complex properties including communication parameters and time and covers besides the dynamic binding of service contracts and the replacement of components also the evolution of the systems by means of new service contracts. The modeling as well as verification capabilities of the presented approach are demonstrated by means of a supply chain example and the verification results of a first prototype are shown.

Keywords Service-Oriented Architecture, Verification, Contracts, Evolution, Infinite State, Invariants, Modeling, SoaML, Runtime Binding, Runtime Reconfiguration

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Service-oriented architecture enables more flexible IT solutions. At the level of the architecture the runtime binding of service contracts, starting new component instances and terminating components result in a dynamic assembly and runtime reconfiguration of complex open IT landscapes. As new services contracts can be added at runtime as well, the dynamics goes even further and permit that the IT landscapes evolves at the level of its components as well as service contracts. In the service-oriented approach *orchestration* describes a collaboration with a single dedicated coordinator that enacts the collaboration between the other parties. The *choreography* interaction scheme in contrast support the free interplay of different roles within a collaboration.

The service-oriented approach in contrast to standard component-based architectural models employs collaborations describing the interaction of multiple roles in form of service contracts (cf. [1, 2]). Current approaches for modeling service-oriented architectures, however, do either only support scenarios where the dynamics and evolution are restricted to static collaborations with fixed service contracts [3, 4, 5] or an appropriate rigorous formal underpinning for the model for the conceptually supported dynamics is missing [6]. While orchestration is often described by business processes or activity diagrams [7, 8], for choreography it is less clear which kind of behavioral description is best suited [9].

1.1 State-of-the-Art

Proposals for the formal verification of service-oriented architectures are even more restricted and do only support scenarios where the evolution has been so restricted that checking a bounded formal model is sufficient [10]. A number of approaches support orchestration [11, 12] looking only into a single collaboration, while others [13] also take into account how a fixed number of collaborations can interfere. Approaches to verify also choreography also require a fixed number of participants [14]. Approaches that are not dedicated to service-oriented systems, either have not the required expressiveness [15], do not support verification [16, 17] or do not scale with the system size [18].

Therefore, the current proposals for modeling and verification do not support the beforehand outlined dynamics and evolution of IT landscapes with orchestration and choreography but only checking specific configurations.

Given a fixed system configuration you can of course use tests to detect compatibility problems. While in case of in-house service-oriented architectures testing can thus provide some coverage in case the dynamics and evolution of IT landscapes is restricted to the tested cases, for more dynamic scenarios the high if not unbounded number of possible configurations results in a low coverage. Furthermore,

in case of more advanced scenarios such as cross-organizational service-oriented architectures, digital ecosystems [19] or ultra-large-scale systems [20], no overarching governance exists and thus the open and dynamic character of these systems prevent that all possible combination of components and service contracts can be systematically tested before they could become active.

In the literature several aspects are discussed, that we will also have to investigate. Summarizing we can say, that most of the modeling approaches lack the capability to model dynamic collaborations. The verification techniques, targeting service-oriented architectures, mostly only address the verification of orchestrations, whereas approaches for the verification of dynamic systems are rare or assume a bounded, previously known number of reconfigurations. Decompositional approaches exist, but to the best of our knowledge, none of them provides techniques for the decomposition at type-level.

1.2 Contribution

In this report we present our approach for modeling services and their service contracts at different levels of abstraction. It is an extension of *SoaML* the current OMG proposal for service modeling with UML [1], provides a formal semantics for all used and newly introduced modeling concepts and enables compositional verification of critical properties. The approach allows for complex properties including communication parameters and time and therefore covers besides the dynamic binding of service contracts and the replacement of components also the evolution of the systems by means of new services.

This work conceptually continues earlier work to guarantee crucial safety properties for patterns/collaborations with a fixed number of roles using model checking [21], where the behavior of the interaction of a collaboration and its roles could be verified separately. It uses and extends our own automatic formal verification approach for systems with structural dynamics [22, 23] covering rules to join and leave a collaboration. Also a first combination of the former two approaches to verify the coordination for one collaboration and the outlined pure structural rules have been presented in [24]. However, this former approach is limited to solutions where a collaboration is instantiated or terminated as a whole, only a small, finite number of static roles per collaboration exist, the reactive behavior of the roles itself has to be fully decoupled from the structural dynamics and the collaboration does not support parameter passing.

1.3 Supply Chain Case Study

In this report we want to introduce the concepts of *SoaML* together with our running example. We have chosen a service-oriented supply-chain network as application example. In a supply-chain network arbitrary factories can participate as long as they fulfill some minimal requirements. A supply-chain network can be recursively built from any factory delivering a product. The supply-chain network of a factory C is the union of the supply-chain networks of all factories, C buys products from, with C being the new root element. Factories that sell raw materials have a supply-chain network that contains only themselves. In domains such as the automotive or avionic industry these supply-chain network become easily very large. Also the business relationships among factories may change often, depending on several constraints such as required product quality, production costs, delivery deadlines to only name a few. However, despite all this it has to be ensured that each factory within the supply-chain network delivers the requested product. The contract negotiation between two factories is established through a service contract. The exact form of the contract is not specified as this might be dependent on the different domains and products.

Figure 1.1 depicts an example for a small supply chain system or a snippet of a supply chain system.

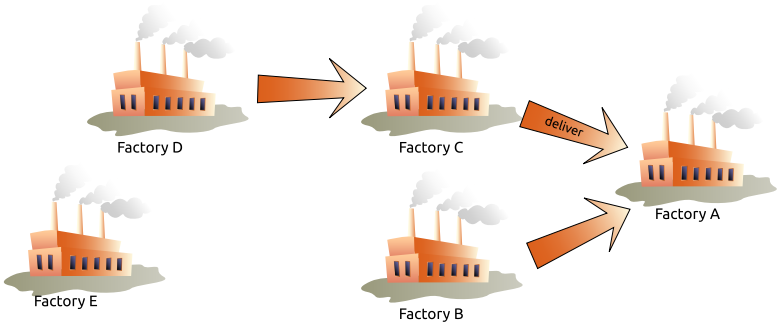


Figure 1.1: Sketch of a small supply chain system instance

The system consists of four factories. The arrow between two factories indicates that the two factories are in a contract relation with each other, where the supplier is marked by the arrow's origin and the customer is marked by the arrow's target. The fifth factory ("Factory E") of the example is currently not connected with any of the other factories.

1.4 Outline

The report is organized as follows: In Chapter 2 we present a list of challenges that a modeling and verification approach for open service-oriented systems has to fulfill. We introduce the extended OMG proposal for service modeling using the supply chain example and discuss its limitations in Section 3. The proposed extensions for modeling services to support abstraction as well as evolution is then outlined in Section 4. Thereafter, we explain the employed compositional verification technique and demonstrates its applicability using our running example in Section 5. The state-of-the-art for modeling and verification of service-oriented systems is reviewed in Section 6. The paper closes with a final conclusion and outlook on planned future work.

Chapter 2

General Challenges

The outlined supply chain case study also permits us to explain in more detail the challenges that result for the modeling and analysis of SOA referring to the concrete case study where helpful. We will use this list of challenges to discuss the shortcomings of *SoaML* in the succeeding Section 3.2 as well as the benefits of the proposed approach concerning modeling in Section 4.4, concerning analysis in Section 5.4, and concerning modeling and analysis of evolution in Section 2.3.

2.1 Modeling: Coverage & Dynamics

Modeling SOA (M1) The model has to cover the concepts of service-oriented systems such as service contracts, roles, components, architecture and service landscapes.

Modeling Dynamics (M2) The model has to support also the dynamics of service-oriented systems such as joining/leaving service contracts dynamically and adding components dynamically.

Transferred to our application example, challenge Modeling Dynamics (M2) requires that the modeling language supports that factory E can establish a contract with any of the other factories and that the existing contract relations can be terminated. Further, additional factories, not yet depicted in Figure 1.1, could be added to the supply chain system.

2.2 Analysis: Scalability & Applicability

Scalable Analysis (A1) The service landscapes are potentially very large and thus checking all possible configurations may not scale.

Analysis of Reconfiguration (A2) The service landscapes change their configuration at runtime. Therefore we can in principle not check each configuration in isolation but have to consider the interplay between the regular behavior and reconfiguration behavior.

Analysis under restricted knowledge (A3) The analysis has to work also in the face of 1) no global view and separated responsibilities, 2) IP constraints for component details and maybe even 3) IP constraints for contract details.

Although, the exemplary sketch of an supply chain network, depicted in Figure 1.1, shows a small instance situation, only, a supply chain network easily grows very large. This is especially the case for the automotive or the avionic industry. Thus the checking of these large supply chain networks has to scale with the network's size and probably even a linear increase in computational complexity would excess available computational resources. Challenge Analysis of Reconfiguration (A2) points out the fact, that no fixed configuration generally exists. Last, challenge Analysis under restricted knowledge (A3) mentions the restricted knowledge of the overall system each of the participants has. Thus in a supply chain network it is unlikely that a customer knows all business relations of each of it's suppliers. For our application example this means that we cannot argue that Factory A knows for sure whether or not Factory B is supplied by Factory D.

2.3 Evolution: Modeling & Analysis

Modeling Evolution (E1) The modeling has to cope with the uncoordinated introduction of new types for service contracts and components at runtime.

Analyzing Evolution (E2) Also the analysis has to cope with the uncoordinated introduction of new types for service contracts and components at runtime.

Expressed in the terms of our application example, challenge Modeling Evolution (E1) can be understood in a way that the number of factories, that participate in the supply chain network is not fixed. Also, the service contracts, i. e. the types, that are to be established between the different factories might change, this could be the case if legal regulations require these changes or if new factories, following a different business model, participate in the supply chain network. Challenge Analyzing Evolution (E2) can be compared to challenge Analysis under restricted knowledge (A3), whereas Analyzing Evolution (E2) targets the introduction of types and Analysis under restricted knowledge (A3) targets the introduction of new instances. Thus, for Analysis under restricted knowledge (A3) the system's set of types does not change but the instance situation does, in contrast to Analyzing Evolution (E2), which involves a change to the set of types, but not necessarily a change at the instance situation.

Chapter 3

Modeling with SoaML

The modeling language for Service-Oriented Architecture proposed by the Object Management Group (OMG) is called *SoaML*[1][25]. Although *SoaML* is not yet an official OMG standard major tool vendors such as International Business Machines provide direct tool support for *SoaML* within their modeling tools.

3.1 Modeling Concepts

SoaML is a meta-model and profile for the modeling of service-oriented systems using the UML. *SoaML* mainly uses collaborations and components to describe the system's structure, UML-behaviors are used for the modeling of the behavior of the different parts. Further, *SoaML* defines different views on the system, such as the service- and the participant-architecture. *SoaML* relies on UML collaborations as their basic building blocks for modeling a services architecture as well as a single service. Services are defined as collaboration among roles, components can participate in a service if they fulfill the requirements of at least one of the roles.

3.1.1 Service Architecture

The most abstract service related view, available in a *SoaML* model is the ServicesArchitecture. The intent of the ServicesArchitecture collaboration is to point out, which services exist and how the different entities work together within those services. A service is modeled as a service contract collaboration. Typically a service contract comprises roles and a behavior – the service's choreography. The choreography can be modeled using any UML behavior specification such as e. g. interaction and activity diagrams. The roles that are defined in a service contract can be bound to components, which provide a matching interface.

Figure 3.1 depicts an exemplary ServicesArchitecture for our supply chain system application example.

[1] http://www.soaml.org

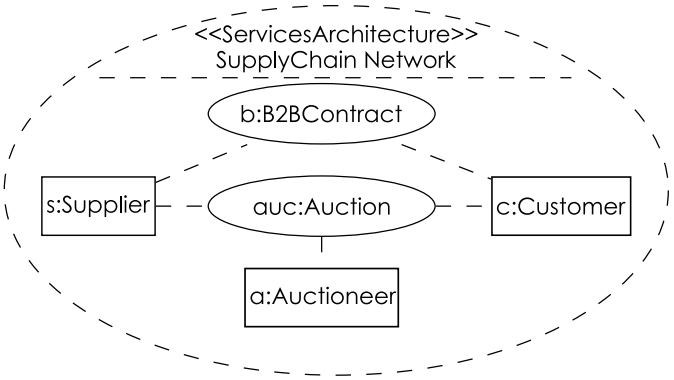


Figure 3.1: The ≪ServicesArchitecture≫ for the SupplyChain application example

3.1.2 Participant Architecture

Components participating in a service contract instance are called Participants and their internal structure is described in a ParticipantArchitecture diagram. Especially in a situation where a participant participates in multiple service contract instances at a time it is required that the participant provides a behavior that coordinates the participant's action within the different service contract. This behavior is called orchestration behavior. Figure 3.2 depicts the ParticipantArchitecture for a factory from our application example. The shown Factory component provides the two interfaces Customer and Supplier via its two ports. Internally the component relies on an entry and an exit store and a production unit to fulfill the request it receives via the supplier port. The internal component controller orchestrates the interplay between the three other internal components. The service contracts used in the Factory participant do not differ completely from the ones used between participants. The main difference to the contract service contract is the fact that now a Controller component initiates behavior and tells the others which action they actually have to execute.

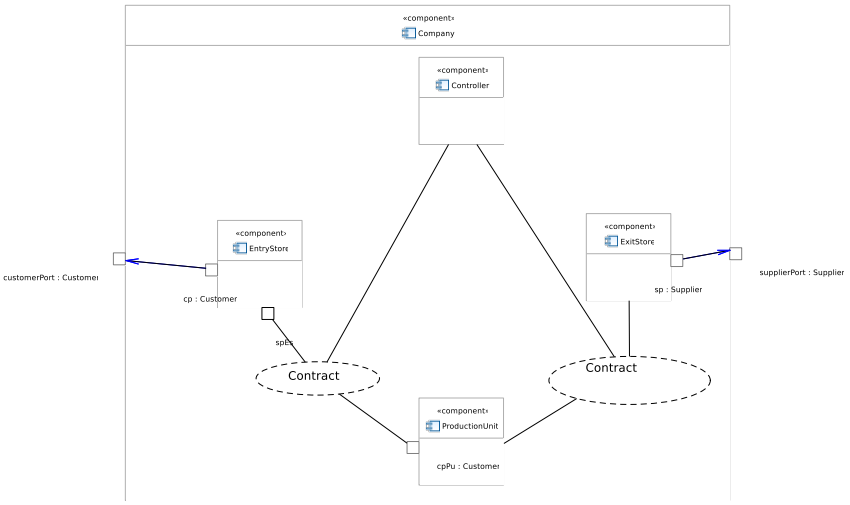


Figure 3.2: ParticipantArchitecture for the Factory participant

The complete supply chain network can be modeled as a collaboration consisting of the roles Supplier,Customer and Auctioneer (cf. Figure 3.1). The services that exist in this architecture are Auction and BusinessToBusinessContract. The Auction service contract requires all three different roles to be bound in order to be established, the BusinessToBusinessContract requires only that

the SUPPLIER and the CUSTOMER roles are bound. In our application example componnents can play different roles, also more than one. The supply chain application example we describe the FACTORY component. The FACTORY component specifies any type of factory that exists within the supply chain network. For a fully functional supply chain system we would also need additional start- and end-points. These could be for example special factory-components that do only use either their SUPPLIER or CUSTOMER roles.

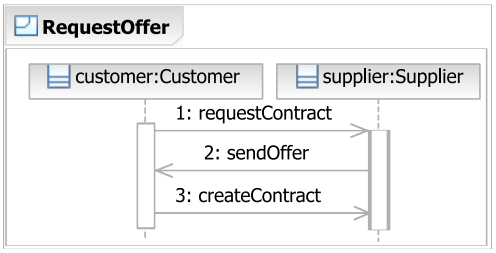


Figure 3.3: An UML interaction diagram showing the BUSINESSTOBUSINESS service contract

Each service is also connected to a specific behavior, the service choreography. For the BUSINESSTOBUSINESSCONTRACT service contract the choreography is depicted in Figure 3.3. The behavior describes a simple request-response protocol that runs between SUPPLIER and CUSTOMER. The CUSTOMER sends the SUPPLIER a CONTRACTREQUEST and the SUPPLIER eventually replies to this request with a CONTRACTOFFER. The CUSTOMER can accept the proposed offer and a CONTRACT will be instantiated between the two roles. Each CONTRACT is equipped with a recall date. A CONTRACT becomes invalid once the recall date has passed and it has not been updated by one of the involved participants. The sequence diagram depicted in Figure 3.3 only depicts the scenario for a successful negotiation between SUPPLIER and CUSTOMER. The AUCTION service contract also aims at establishing a valid CONTRACT between a SUPPLIER and a CUSTOMER but also owns a negotiation phase – in form of a classical auction – where the highest price for the offer good is determined among a possibly varying set of bidders. Bidders are allowed to join the AUCTION service contract as long as the service is in the auction stage. A bidder also may leave the service contract at any time except he is the current leader in the auction.

Although, we only have introduced some different components and service contracts we can not be sure that those are the only ones that are acutally involved in real supply-chain network. The current set of collaboration and component types is referred to as a service landscape. For a fixed service landscape multiple intstantiations can be created. An instantiation of a service landscape is called a serivce landscape configuration. If we also refer to systems where the set of component and collaboration types can change over time we call this a complex service landscape.

3.2 Discussion of SoaML

SoaML addresses challenge Modeling SOA (M1) by providing concepts of service-oriented systems such as service contracts, roles, components, Services architecture and participant architecture. It does not explicitly support to model also the dynamics of service-oriented systems such as joining/leaving service contracts dynamically and adding components dynamically required for challenge Modeling Dynamics (M2).

Also evolution is not directly supported and thus neither the modeling nor analysis of evolution of the service landscapes as raised by challenge Modeling Evolution (E1) is covered.

Challenges	Coverage
Modeling SOA (M1)	✓
Modeling Dynamics (M2)	○
Modeling Evolution (E1)	○

Table 3.1: Coverage of the challenges for modeling with *SoaML*. Legend: ✓ means the challenge is fulfilled, ∼ means the challenge is partly fulfilled and ○ means the challenge isn't fulfilled.

Chapter 4

Modeling with rigSoaML

The previous chapter yields that *SoaML* has the capabilities to model the basic constituents of a service-oriented system but does falls short of modeling the structural dynamism (cf. Table 3.1). In this chapter we will introduce our modification of *SoaML*, called *rigSoaML*, that explicitly addresses these issues.

4.1 Prerequisites

We need some formal concepts and clarifications to describe our approach *rigSoaML*. We will first introduce the formal model and then connect the formal model with our modelling notations.

4.1.1 Semantic Model

The formal model we are going to use for *rigSoaML* can be roughly described as graphs are states and graph transformation rules are the behavior. Further, we will introduce some methodologies to separate system parts at the type and the instance-level, in order to be able to transfer the findings of our formal reasoning to the running system. Graphs are defined in Definition B.2 and extended to attributed and typed graphs in Definition B.10. We will refer to the empty graph as $G_\emptyset$. Graph transformation rules are introduced in Definition B.11. However, we will use a simplified notation as we are not going to add the control modes every time they should occur. In our application example, we only use clocks, i. e. attributes that have a constant derivation of 1, and constants.

4.1.2 Results for Composing Pseudo-Type Separated GTS

We will further required some results concerning the combination of different GTS with respect to the guaranteed properties and their interference with each other. We will use the follwoing terms, throughout the paper to characterize possible interference:

type separated means that due to types two rules sets cannot interfere as they have no nodes with the same type in common

pseudo-type describes a special node t at instance-level that serves to identify a set of other nodes. Therefore each node, that is supposed to be pseudo-typed by t has to be also connected to t by a link. A node must not be pseudo-typed by two nodes. Pseudo-typing can be applied to rules and properties.

pseudo-type separated means that due to pseudo-types two rules sets cannot interfere as they have no nodes with the same type or pseudo-type in common; it also requires that all rules including the two rule sets preserve the pseudo-typing.

The use of pseudo-typing is expressed in the following corollary, which is a more informal variant of Corollary C.11

Corollary 4.1
The union of rule sets separated by pseudo-typing that both preserve their pseudo-typing also preserve the pseudo-typed properties of each of the rule sets.

The complete proof and additional background information is contained in Appendix C. However, in order for the lemma to be correct and to preserve the pseudo-typing it is of tremendous importance, to forbid to specialize concrete collaborations or components. This ensures that all concrete collaborations/components are separated and only be linked through abstract collaborations used by ports of concrete components. However, here the separation of the components is sufficient as all possible concrete collaborations can in fact be linked to the port.

To make this work, we employ inheritance of types. Hence, a component with a role implementing a role of an abstract collaboration will preserve the pseudo-typing for a concrete collaboration type, the implements the same abstract collaboration type, even though it does *not* know the concrete collaboration's actual type, as it simply links it to the collaboration type node, which – by construction – is a subtype of the collaboration type node of the abstract collaboration the component's role refers to.

4.1.3 Employed Notations for GTS

We will employ *Class diagrams* to specify the types and their attributes including clocks for the node of the graphs. Thus the set of lables that are available for nodes and edges in graphs can be directly derived from the modelled UML-class diagrams.

For graph transformation rules we use *StoryPatterns* as a modeling notation. StoryPatterns are an extension of the UML instance diagrams, that allow the developer to also model side-effects, such as creation and deletion of objects and links, within one diagram. Therefore two special stereotypes ≪CREATE≫ and ≪DESTROY≫ are used. Elements augmented with the create (delete respectively) stereotype will be created (deleted) by the application of the StoryPattern. The applicability of StoryPatterns could be restricted by the use of negative application conditions (NAC), which describe elements that must not exist in the current instance situation, and constraints above the attributes. The translation of a StoryPattern into a graph transformation rule is an easy to accomplish task. All elements that are to be deleted or remain unchanged specify the rule's left hand side and all elements that are to be created or remain unchanged specify the rule's right hand side. The NACs are directly translated, as they do not contain any side-effects. The labeling for nodes and edges is given by the links' and objects' types.

StoryPatterns are also employed to specify graph pattern. However, in a StoryPattern that specifies a graph pattern side-effects must not occur. I. e. only a situation is described, but no change to the situation.

Throughout the report we will refer to the global set of all rules as $\mathcal{R}$.

Example 1: To illustrate the StoryPattern notation have a look at Figure 4.3. The figure shows a StoryPattern that specifies the Contract *creation between a* Supplier *and a* Customer *role. The* Contract *node is marked with a ≪create≫ stereotype and thus created by the rule. The StoryPattern is only applicable is both* Customer *and* Supplier *belong to the same* ContractCollaboration *instance and no* Contract *has been created, yet.*

4.2 Modeling Concepts

Amongst all the capabilities of *SoaML*, which we have roughly described in the above paragraphs, the modeling concept of *SoaML* also has some limitations. The two service contracts BusinessToBusinessContract and Auction could both be seen as specialized types of a abstract Contract service contract. Service contracts in *SoaML* subtype the UML concept of collaborations and thus they also support inheritance. Unfortunately the UML only defines the specialization of collaborations at the role-level. *SoaML* reuses this definition but misses to clearly define what inheritance means to the choreography of the specialized service contract. Further *SoaML* does not contain any concepts to model that participants join or leave a running service contract as it is required for the Auction service contract. If the modeled system should be verified the developer needs some guidance, that makes clear what kind of specification is required in which modeling stage. *SoaML* is missing this guidance, what is obvious as *SoaML* is a multi-purpose modeling language. Lastly, the notion of structural changes, which naturally occur in a service-oriented system, can hardly be expressed with the concepts offered by *SoaML*.

4.2.1 Service Roles

As *SoaML* does, we also make use of Roles to decouple collaborations and components. However, we distinguish between abstract and concrete roles, what results in mainly three typical cases how roles are used in *rigSoaML*:

abstract roles only define the concept but without behavior; are used to describe assumed/guaranteed properties of participants even when their behavior is not defined

concrete roles (in abstract collaborations): definer role behavior such that components can refine it; leave behavior of other roles and the network open

concrete roles (in concrete collaborations): definer role behavior such that components can refine it; behavior of other roles and the network can also only be refined

Formally we can define a service role as follows:

Definition 4.2
A role type $ro^i = (ro^i)$ *consists of a role type node* ro_i. *The role type is* concrete *if it has an assigned concrete behavior and otherwise* abstract. *It is further* refined *if role types exist that are subtypes.*

A role instance of a role type ro^i is represented by a node of type ro^i. The rules that are assigned to the role type ro^i have to have an instance of that type in their precondition. Thus, the rules are only

applicable if an instance of that type exists. Further, the rules have to preserve a pseudo-typing over role type ro^i (cf. Section 4.1.2) by linking all nodes, occurring in the rule, to ro^i.

Example 2:[Abstract Role] In the supply chain application example exist two abstract *role types. For the role types* CUSTOMER *and* SUPPLIER *no concrete behavior is specified. These two role types get* refined *through further role types that will be introduced within the supply chain example.*

4.2.2 Service Contracts/Collaboration

rigSoaML uses collaborations to specify the different service-contracts that are available in a service-oriented system. However, the basic notation as UML-collaborations, that is used in *SoaML*, is not sufficient for our purposes, as we will need more information for a collaboration to be specified.

Definition 4.3
A collaboration type $Col_i = (col_i, (ro_i^1, \ldots, ro_i^{n_i}), CD_i, R_i, \Phi_i)$ *consists of a collaboration type node* col_i*, a number of roles* ro_i^j*, an UML class diagram* CD_i*, a function* $R_i : \{col_i, ro_i^1, \ldots, ro_i^{n_i}\} \mapsto 2^{\mathcal{R}}$ *assigning rules to roles, and guaranteed properties* Φ_i*. The collaboration type is* concrete *if all roles are concrete (have an assigned concrete behavior) and otherwise* abstract*. It is further* refined *if collaboration types exist that are subtypes, with the restriction that only* abstract *collaboration types can be refined.*

A collaboration instance of collaboration type Col_i is represented by a node of type col_i. All rules of Col_i also preserve a pseudo-typing linking of all nodes to col_i. For two different roles ro_i^k and ro_i^l the set of assigned rules has to be disjoint $R_i(ro_i^k) \cap R_i(ro_i^l) = \emptyset$. The creation of collaboration instances of collaboration type Col_i is only possible through the collaboration type's roles ro_i^k and their assigned behavior $R_i(ro_i^k)$. The collaboration type Col_i's property have to be pseudo-type separated by the collaboration type node col_i.

The relation amongst the collaboration Col_i's roles $ro_i^1, \ldots, ro_i^{n_i}$ and any additional data types that are used within the collaboration are specified within the class-diagram CD_i. The class-diagrams of different collaborations have to be separated by different name-spaces. However, a collaboration Col_i being a subtype of collaboration Col_j, is allowed to enhance the class diagram CD_j with it's own types. Obviously, this is required as otherwise collaboration subtypes have no possibility to use the super collaboration's roles and data-types. Nevertheless, the new elements have to be defined in a separated name-space.

The rules for creation of new role-instances and the connection of role-instances with collaboration-instances are part of the roles' behavior. For the creation of new role-instances we can distinguish two different cases. First, a role owns a rule that specifies the creation of a new instance of an other role instance (which is not necessarily of the same type as the creating role). Second, for a role-type a rule exists, that allows instances of that type to connect with an existing collaboration-instance. Finally, the combination of the two cases is allowed, too. However, then the first case has to be restricted to the creation of roles of the same type as the creating role.

Within a collaboration both, synchronous and asynchronous, communication styles can be specified. For asynchronous communication message passing schemes could be employed. I. e. an instance of role A creates a new message and links/sends it to a role B. Later, role B can process the new message. For a synchronous communication role A has to directly modify role B, e. g. setting a mode-flag of role B.

The collaboration's choreography will be modeled through Story Patterns. In Figure 4.3 an example for a StoryPattern is depicted. The StoryPattern is applicable if both roles SUPPLIER and CUSTOMER

could be matched and they were connected by an ContractCollaboration instance. The result of the StoryPattern is, that a new Contract instance becomes created, whose attribute recallDate has to be greater than the ContractCollaboration's now attribute.

For the modeling of the collaboration's properties Φ_i we facilitate StoryPatterns, too, but we restrict them to be side-effect free. I. e. it is forbidden that StoryPatterns for properties create or delete elements. This restriction is possible as they are only used to identify sets of states that satisfy a certain condition, the condition that is expressed through the StoryPattern. We say that a state – i. e. an UML object diagram – satisfies a StoryPattern iff we can find a match for the StoryPattern in the instance diagram. For the frequent case that we want to explicitly forbid certain situations we can prefix the pattern P with the temporal logic expression $AG\neg\exists P$, meaning that the pattern P must never occur in the instance graph. If P is always used in this way, we call P a forbidden pattern. An example for a forbidden pattern is shown in Figure 4.2. This StoryPattern matches all states where two Contracts are established between the same Customer and Supplier roles. This StoryPattern is only used in combination with the temporal logic prefix $AG\neg\exists$ and is hence called a forbidden pattern.

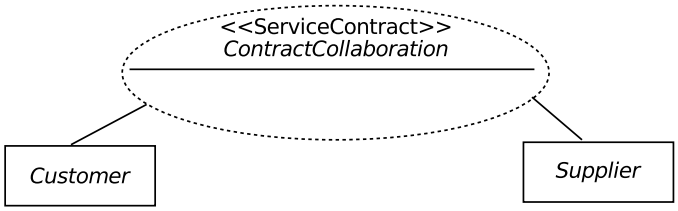


Figure 4.1: Contract Collaboration Structure

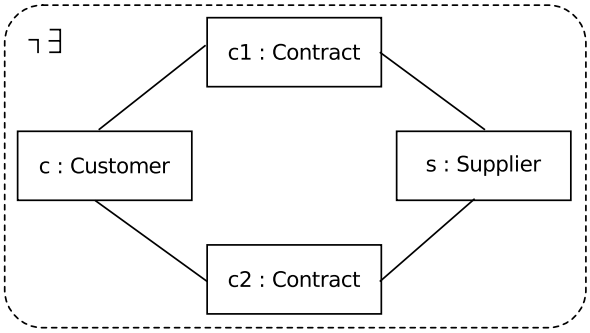


Figure 4.2: Property: Not two Contracts between Customer and Supplier role.

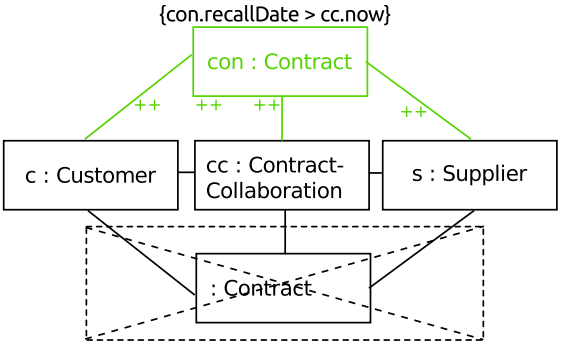


Figure 4.3: The ContractCollaboration's createContract rule

Example 3:[abstract collaboration] An example for an abstract collaboration is the contract collaboration, whose structural diagram is depicted in Figure 4.1. The contract collaboration *is specified as*

$Col_{CC} = (ContractCollaboration, (Customer_{CC}, Supplier_{CC}), CD_{CC}, R_{CC}, \Phi_{CC})$. *Although the collaboration is abstract, there is behavior specified for the* Customer *role. The* Customer *role can create a* Contract *between the* Customer *and the* Supplier *role (see Figure 4.3). The collaboration network, does not have any assigned behavior. The collaboration's properties specify that not two* Contracts *exists between the* Customer *and the* Supplier *role. This is formally expressed as* $\Phi_{CC} = AG\neg\exists$ *twoContracts with twoContracts being the graph constraint depicted in Figure 4.2.*

$$R_{CC}(Customer_{CC}) = \{createContract\}$$
$$R_{CC}(Supplier_{CC}) = \emptyset$$

The rules for the collaboration's Customer *role can be used to exemplify pseudo-typing as introduced in Section 4.1.2. The role's* CreateContract *rule (cf. Figure 4.3) contains the collaboration node of type* ContractCollaboration*, which is connected to all other nodes – also the created one – contained in the rule. The rule is* pseudo-typed *over the node of type* Contract-Collaboration.

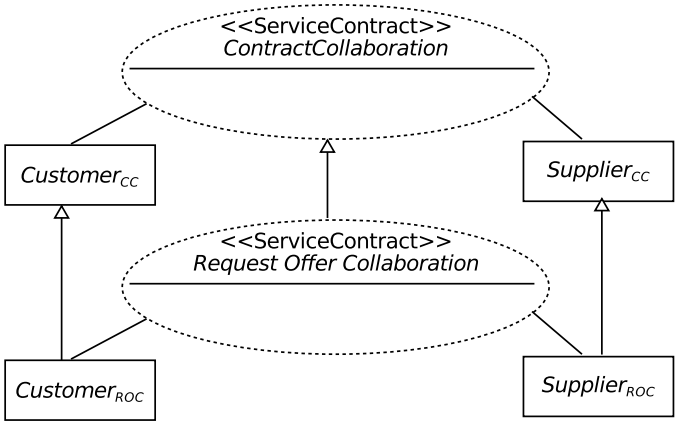


Figure 4.4: Request Offer Collaboration Structure

Example 4:[Concrete collaboration] The abstract contract collaboration *introduced in the previous example is refined by the* concrete collaboration Request Offer Collaboration *(ROC)* $Col_{ROC} = (RequestOfferCollaboration, (Customer_{ROC}, Supplier_{ROC}), CD_{ROC}, Net_{ROC}, R_{ROC}, \Phi_{ROC})$. *The ROC's structure is depicted in Figure 4.4. The concrete role types* $\text{Customer}_{\text{ROC}}$ *and* $\text{Supplier}_{\text{ROC}}$ *refine the abstract role types* $\text{Customer}_{\text{CC}}$ *and* $\text{Supplier}_{\text{CC}}$*, respectively. The roles' behavior is specified through two sets of story-pattern, which allow the* $\text{Customer}_{\text{ROC}}$ *to send a* Request *to the* $\text{Supplier}_{\text{ROC}}$*, who in turn can answer by sending an* Offer *and finally if the* Offer *is acceptable to both a* Contract *can be created. These rules are depicted in Figure 4.7. For the collaboration's two roles* Customer *and* Supplier *we get the following assignment of rules:*

$$R_{ROC}(Customer_{ROC}) = \{sendRequest,\ createContract\}$$
$$R_{ROC}(Supplier_{ROC}) = \{sendOffer\}$$

The rules describing the Customer *role's behavior are shown in Figures 4.7(a) and 4.7(c). The* Supplier *role's behavior is depicted in Figure 4.7(b).*

In comparison with the Contract-Collaboration *introduced in Example 3 the collaboration's properties have been extended by three more graph constraints, depicted in Figure 4.6. The collaboration type's class diagram is depicted in Figure 4.5.*

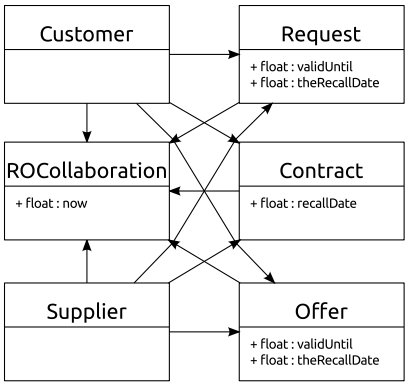


Figure 4.5: Class Diagram CD_{ROC} for the Request Offer Collaboration

4.2.3 Service Provider/Components

In conformance with *SoaML*, *rigSoaML* employs components to implement the collaborations' roles. *SoaML*, however, does only support a syntactical refinement between roles and components, i. e. the interfaces should look the same, whereas we further require a semantical refinement. Therefore it is necessary to specify additional relations between roles and components. Our specification of components will comprise safety-properties, that have to be fulfilled by the component's implementation, too.

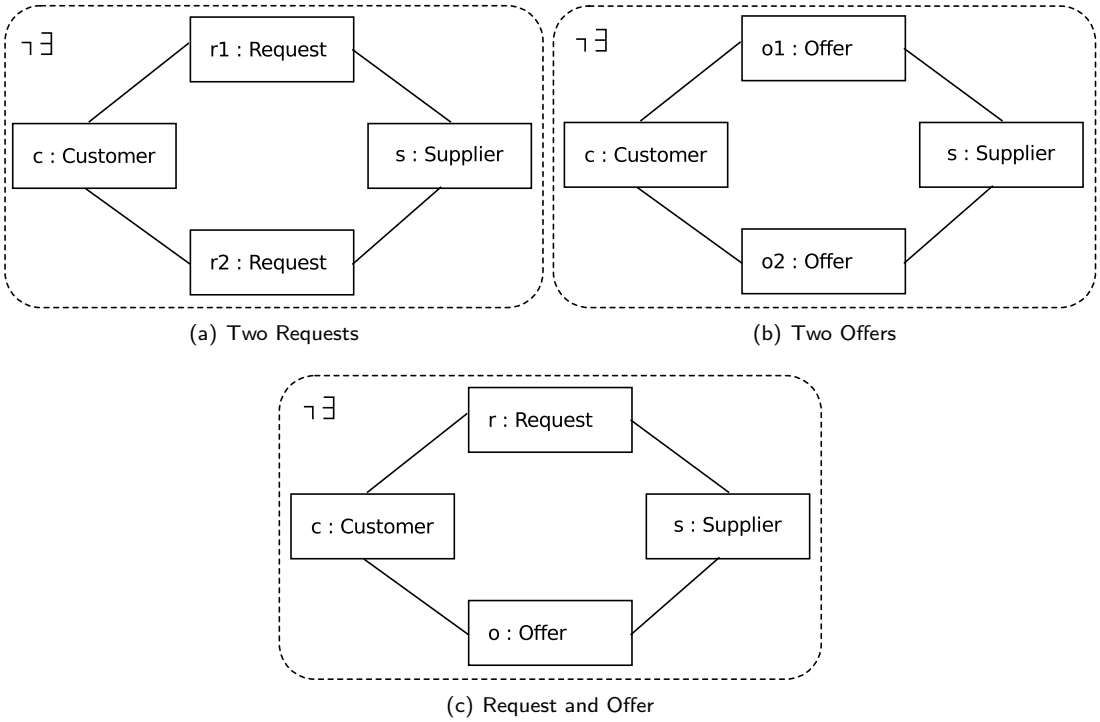
Definition 4.4
A component type $Com_i = (com_i, (ro_i^1, \ldots, ro_i^{m_i}), CD_i, R_i, I_i, \Psi_i)$ *consists of a component type node* com_i*, a number of roles* ro_i^j*, a class-diagram* CD_i*, a function* $R_i : \{com_i, ro_i^1, \ldots, ro_i^{m_i}\} \mapsto 2^{\mathcal{R}}$ *assigning rules to roles, a set of initial rules* $I_i \subseteq R_i(com_i)$*, properties* Ψ_i*. The component type is* concrete *if all roles and the implementation are concrete (have an assigned concrete behavior) and otherwise* abstract*. It is further* refined *if component types exist that are subtypes. An initial rule* $i \in I_i$ *has to have an empty pre-condition and must only create elements, that are pseudo-typed to* com_i*.*

A component instance of component type Com_i is represented by a node of type com_i, which also fulfills the pseudo-typing requirements and thus separates elements from each other that belong to different component instances. All rules of Com_i preserve a pseudo-typing linking all nodes to com_i. The function R_i is defined as for collaboration types (see Definition 4.3). The only way a component instance of type Com_i can be created is through the execution of any of the creation rules in I_i. The creation rules I_i may be refined through a component type Com_j that has a create relation $Com_j \xrightarrow{create} Com_i$ to Com_i. As for collaboration types the component types' properties have to be pseudo-typed over the component type node.

The component type's class diagram CD_i contains all class diagrams of the collaboration types that are used by the component type.[1] Additionally, the component itself, represented by a class com_i (node type), and all data-types required by the component are contained in CD_i. If the component type Com_i refines the component type Com_j parts of CD_j might be contained in CD_i, too. Again the types, defined by different components, must be located in different (and disjoint) name-spaces.

We write $R_i(ro_i^k) \subseteq R_i(com_i)$ to refer to the set of all rules that belong to the component Com_i's implementation of role ro_i^k.

[1] A component type uses a collaboration type, if the component type implements a role that has been defined for this collaboration type.



(a) Two Requests (b) Two Offers

(c) Request and Offer

Figure 4.6: Properties Φ_{ROC} for the Request Offer Collaboration

Example 5:[Factory component] The two role types Customer *and* Supplier *are both implemented by a single component* Factory *(see Figure 4.8). The* Factory *component is formally given as:* $Com_{Fac} = \left(com_{Fac}, (Customer^{CC}_{Fac}, Supplier^{CC}_{Fac}), CD_{Fac}, I_{Fac}, \Psi_{Fac}\right)$ *The company's behavior is again specified through a set of rules, that is depicted in Figure 4.10. We have the following assignment of the component's rules to it's roles:*

$$R_{Fac}(Customer^{CC}_{Fac}, Com_{Fac}) = \{\texttt{createRequest}, \texttt{createContract}\}$$
$$R_{Fac}(Supplier^{CC}_{Fac}, Com_{Fac}) = \{\texttt{createOffer}\}$$

In contrast to the collaboration's rules (see Figure 4.7) the factory's rules clearly allow to distinguish which rule is assigned to which role. At the level of collaboration this assignment is not necessarily directly visible, without having a look at the collaboration's specification.

Figure 4.8 shows that the Factory *component extends the* AbstrFactory *component, which is specified as abstract. However, for the* AbstrFactory *component a property is specified:* $\Psi_{AFac} = AG\neg\exists\texttt{earlyRecall} \wedge AG\neg\exists\texttt{custNoCon}$. *The corresponding graph constraints are depicted in Figure 4.11(a) and 4.11(b), respectively. The factory's class diagram* CD_{AFac} *is depicted in Figure 4.9. The component's complete specification can be given as:* $Com_{AFac} = (com_{AFac}, (Customer_{AFac}, Supplier_{AFac}), CD_{AFac}, R_{AFac}, \Psi_{AFac})$ *with* $R(Supplier_{AFac}) = R(Customer_{AFac}) = R(com_{AFac}) = \emptyset$.

4.2.4 Service Landscapes / Architecture / System

The *rigSoaML* counterpart to *SoaML*'s service landscapes are system types and systems. System combine collaboration- and component-types to a conceptual unit.

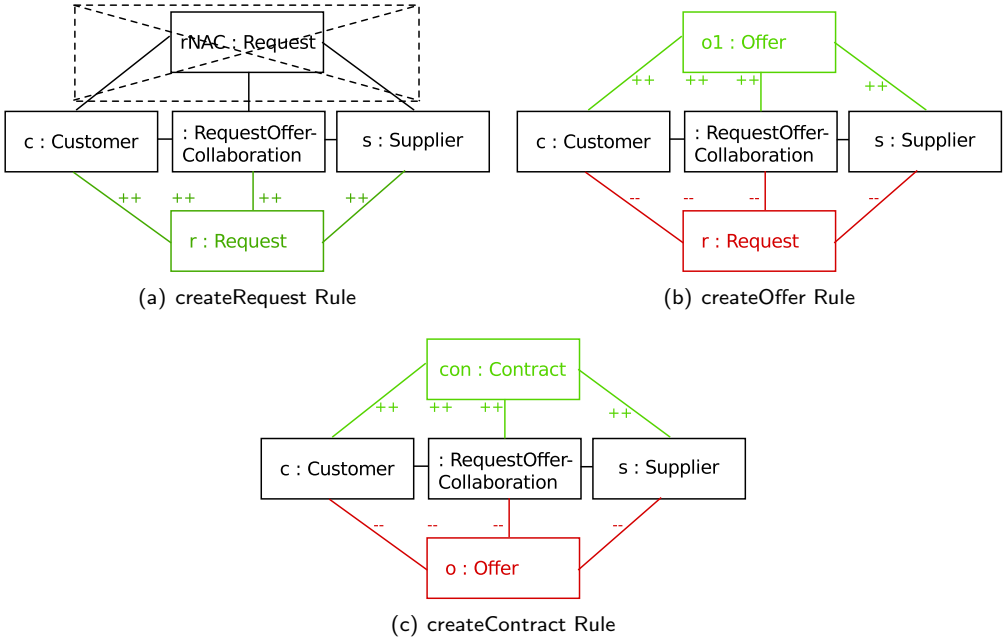


Figure 4.7: Behavioral rules for the ROC's roles

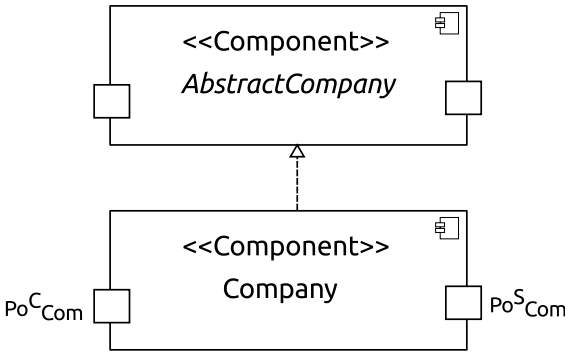


Figure 4.8: Structural overview of the FACTORY component

Definition 4.5
A system type $Sys = ((Col_1, \ldots, Col_n), (Com_1, \ldots, Com_m))$ *consists of a number of collaborations types* Col_i *and a number of component types* Com_j.

Example 6: Using the previous Examples 5 and 4 we can define a first system type $Sys_1 = ((Col_{ROC}), (Com_{Fac}))$.

Definition 4.6
A system *is a pair* $S = (Sys, G_S)$ *with system type* $Sys = ((Col_1, \ldots, Col_n), (Com_1, \ldots, Com_m))$ *and an initial configuration* G_S *that is type conform for* Sys. *A system is* concrete *iff all collaborations* $Col_1, \ldots, Col_n$ *and components* $Com_1, \ldots, Com_m$ *are concrete.*

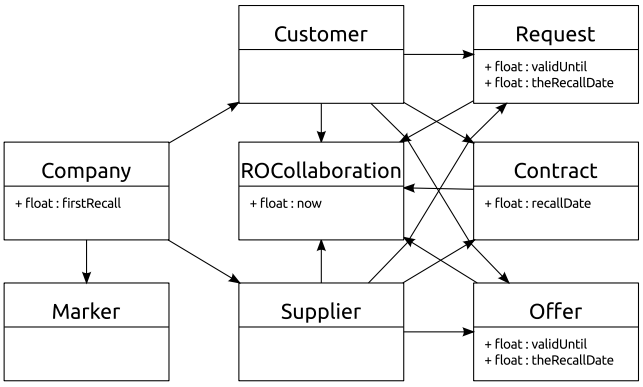


Figure 4.9: Class Diagram CD_{Fac} for the Factory component type

Complex Landscapes

A complex landscape describes a computing evnvironment that is formally given by a system type (cf. Definition 4.5). Hence, for a fixed set of collaboration- and component types, a complex landscape summarizes the structural changes that can happen at the instance level. A snapshot at a certain point in time shows a landscape configuration, i. e. the components and collaborations togehter with their connections between each other. In a landscape configuration only instances of components and collaborations may occur whose type is defined in the corresponding system type.

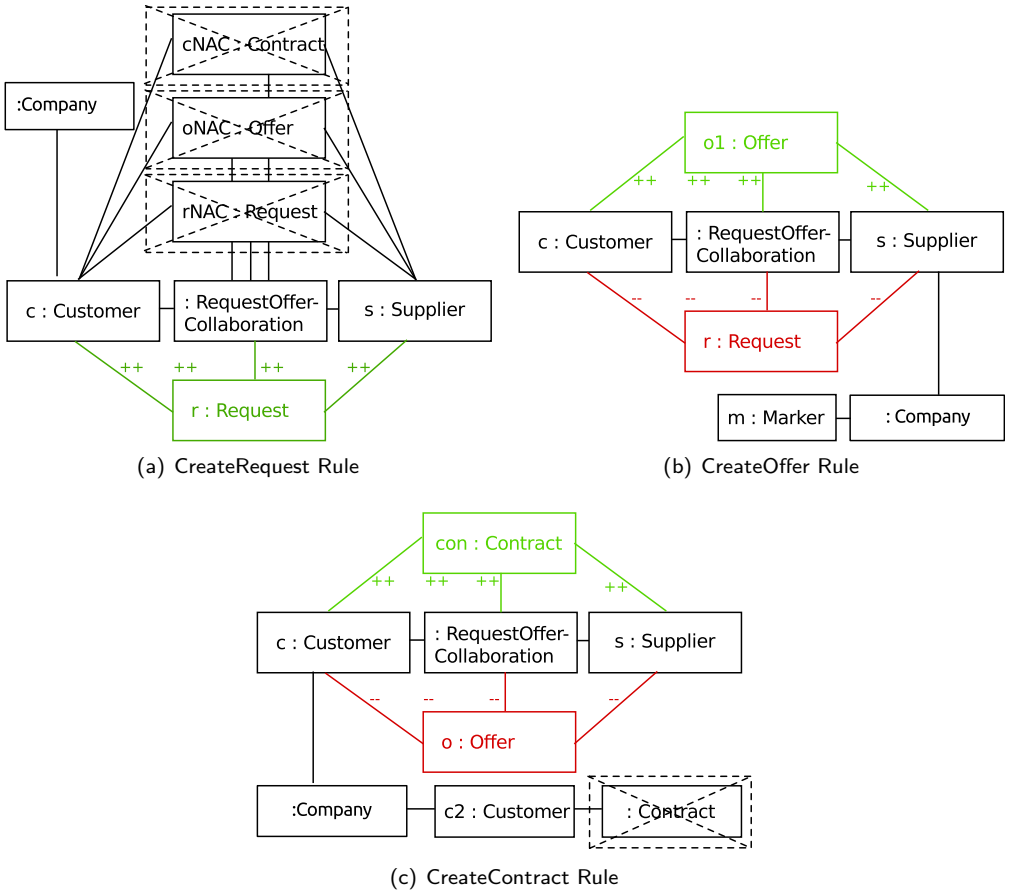
From the developer point of view it is interesting to see, that it is sufficient to have information about the relevant super-types for components and collaborations in order to develop a new constituent. The required information is publicly available and, depending on the current landscape, no even the complete system type specification has to be known to the developer. As a consequence, developers only have to have knowledge of the occurring types within the system and not of the current instance situation.

For a complex landscape, a combined class diagram can be given, that is derived by the combination of the different class diagrams that are specified for components and collaborations. The joints for the different class diagrams are the classes that the different class diagrams have in common, as they use them. The pseudo-type separation and distinction between classes is guaranteed through unique name-spaces. This is, each component and collaboration has to use an unique name-space such that their own classes does not interfere with other components' or collaborations' classes.

The term complex landscape is closely related to the *SoaML* ServicesArchitecture, which basically is a huge collaboration containing all specified service contracts and roles. However, the components are not necessarily part of the *SoaML* ServicesArchitecture but are constituents of the complex landscape. The other view *SoaML* specifies is the ParticipantArchitecture, which does only show the internal architecture of one single participant, i. e. a component in our terms.

Landscape Evolution

One aspect of our motivation for this work is that complex landscape are subject to change and thus the software engineering and verification methodologies have to be aware of these changes. We will use the term landscape evolution to describe that a complex landscape changes. Especially this term will be used, whenever the corresponding system type changes. The reasons, why such changes happen, are manifold but the way the change looks can be roughly categorized in the following cases.



(a) CreateRequest Rule (b) CreateOffer Rule

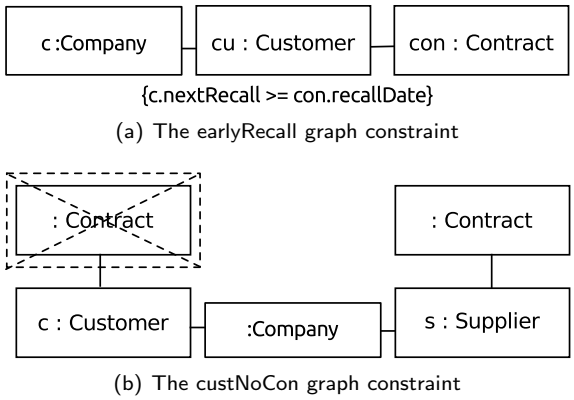
(c) CreateContract Rule

Figure 4.10: Behavioral rules for the ROC's roles

Top-level changes Changes to the type system can be made at the top-most level. Mostly the types that are added at this level are abstract types that do not necessarily provide any new behavior, but are used to describe new concepts, that are required to further develop the current system. It is possible to add component types as well as collaboration types. The removal of types is not supported yet. For our application example it could be the case that at some point in time it is necessary to add transport agents to the system that take care of delivering the goods being dealt between some of the factories.

Implementation-level changes The more frequent case, however, will be changes at the implementation-level. Thus, whenever a new party will participate in the running system it is likely that they provide an own component or collaboration type, depending on the required needs. Anyway, it is important to note that adding a new type at the implementation level is beneficial, as the new type is compatible to the running system and inherits the already proven safety properties.

SoaML does not contain any suitable equivalent to express landscape evolution. Nevertheless, it could be added to *SoaML* as this is, what we have done in this work. In Figure 4.12 we show an exemplary evolution diagram. The diagram can be horizontally split into two main parts, the abstract specification at top and the implementation part below. The implementation part however can further be divided into "swim-lanes", one for each organization that provides an implementation for any of the abstract concepts. The vertical order of the entities in the implementation part indicates a partial order, when



(a) The earlyRecall graph constraint

(b) The custNoCon graph constraint

Figure 4.11: Properties for AbstrFactory component

they have been introduced. To support better discernibility, the evolution diagram separates different versions with alternating background colors (light grey and white).

Definition 4.7
An extended evolution sequence *is a sequence of systems* $(Sys_1, G_S^1), \ldots, (Sys_n, G_S^n)$ *such that (1)* Sys_{i+1} *only extends* Sys_i *by additional collaboration and component types, (2)* G_S^{i+1} *is also type conform to* Sys_i*, and (2)* G_S^{i+1} *can be reached from* G_S^i *in the system* Sys_i*.*

An evolution sequence *is a sequence of system types* $Sys_1, \ldots, Sys_n$ *such that at least one related extended evolution sequence* $(Sys_1, G_S^1), \ldots, (Sys_n, G_S^n)$ *exists.*

4.2.5 Type Conformance

In order to have a scalable solution, we cannot directly approach correctness at the level of system types as the number of collaborations and components in such systems can be already very high. Instead we will only look at type conformance which is easily covered also for large system types. A system type is then type conform if the collaboration and component types are consistently referring to each other.

Definition 4.8
A concrete system type $Sys = ((Col_1, \ldots, Col_n), (Com_1, \ldots, Com_m))$ *is* type conform *iff for all roles references in any component also a collaboration defining that role exists.*

The overall class diagram CD (and thus the related node type set $\mathcal{T}$) is the union of the class diagrams of the collaborations and components and it must hold for any type for a node or edge that it only defined exclusively once, only used in subtype collaborations or components such that they are pseudo-type separated there.

We now have to define what type conformance means for a system that also includes abstract types by extending Definition 4.8.

Definition 4.9
A system type $Sys = ((Col_1, \ldots, Col_n), (Com_1, \ldots, Com_m))$ *is* type conform *if for all roles references in any component also a collaboration defining that role exists and all subtype relations of collaborations and components are type conform.*

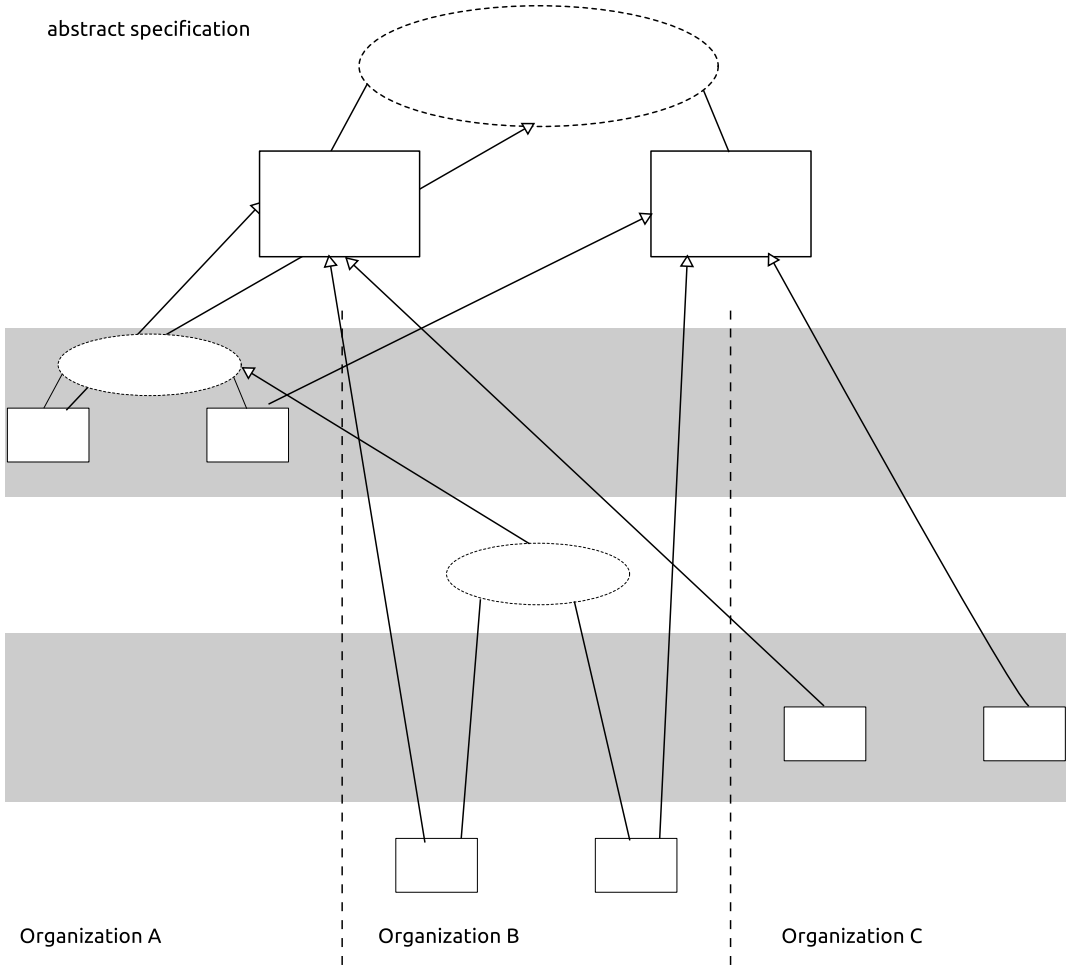


Figure 4.12: Evolution Diagram for an abstract specification and three independent organizations developing implementations

An obvious property of type conform systems is, that the application of type conform rules does not invalidate the systems' type conformance. This is property is straightly inherited from typed graph transformation systems, that have exactly this property. Thus, in our modeling approach it is impossible that a collaboration type instance is connect with a role, which it doesn't know.

4.3 Mapping SoaML to rigSoaML

We will use this chapter to point out the direct expressions of *SoaML* modeling elements and *rigSoaML* modeling elements. This chapter will also show where our modeling approach *rigSoaML* is more detailed than the original *SoaML*.

4.3.1 Services Architecture

A service oriented system, that is modeled using the *SoaML*, as proposed by the OMG, necessarily consists of at least one ≪ServicesArchitecture≫ collaboration diagram. This diagram contains services

and roles, that are used within these services. Figure 4.13 depicts the ≪ServicesArchitecture≫ for the Supply Chain application example.

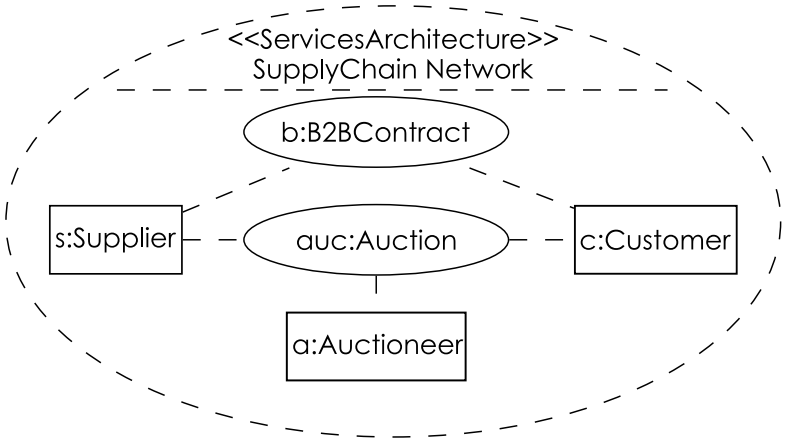


Figure 4.13: Services Architecture for the SupplyChain application example

If we transfer the depicted diagram to our modeling language *rigSoaML*, we get a set of collaboration type definitions (cf. Definition 4.3)

$$Col_{ROC} = (RequestOfferCollaboration, (Supplier, Customer), \emptyset, \emptyset, \emptyset)$$

and

$$Col_{Auction} = (AuctionCollaboration, (Supplier, Customer, Auctioneer), \emptyset, \emptyset, \emptyset)$$

In the above collaboration types, several constituents are not yet defined and we wrote $\emptyset$ to make this clear. Thus, a ≪ServicesArchitecture≫ collaboration diagram only gives us knowledge of the collaboration types and the roles that are available.

The inheritance between service contracts is not directly reflected within our collaboration type definition, but we use this information for the verification. However, the conditions for a correct inheritance are much stronger in *rigSoaML* than in *SoaML*, which mainly uses syntactic substitutability, whereas in *rigSoaML* also behavioral refinement is required (as it will be defined in Definition 5.10).

4.3.2 Behavior specification

We have shown in the previous section that the ≪ServicesArchitecture≫ does not provide a behavior specification for the modeled service contracts. However, in *SoaML* each collaboration has a behavior specification that can be specified using any UML behavior – i. e. sequence diagrams, activity diagrams, state machines. For the Request Offer Collaboration we have already given an exemplary sequence diagram. We show it again in Figure 4.14.

In our modeling approach the sequence diagram has been translated in a set of graph transformation rules. Each message in the sequence diagram, resulted in one graph transformation rule. The assignment to the roles of the collaboration was straight forward: the sender of the message got the corresponding rule assigned. However, although the approach looks very intuitive it is impossible to generalize it to suit an arbitrary system. The important information, we miss here, is the rules' "internal" structure. The sequence diagram does not specify what the exact pre-conditions for sending a request are. Thus, it is up to the modeler to give a solid translation of an UML behavior specification to a set of graph transformation rules. Further, an UML sequence diagram does not necessarily specify the collaboration

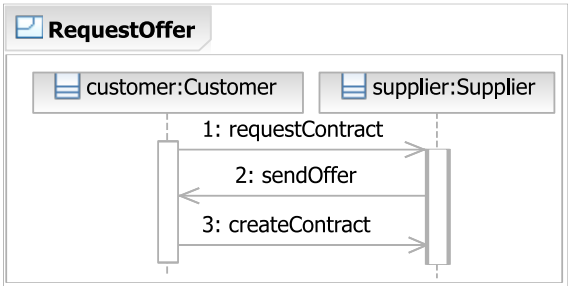


Figure 4.14: Sequence Diagram for the Request Offer Collaboration

type's complete behavior but only a part of the behavior. Our specification approach, however, always captures the collaboration and component types' complete behavior.

4.3.3 Participant Architecture

The ≪ParticipantArchitecture≫ describes in *SoaML* the internal design of the participants in the service-oriented systems. In our terms, a participant is a component and consequently a ≪ParticipantArchitecture≫ becomes translated in a component type declaration. However, as for the ≪ServicesArchitecture≫, the ≪ParticipantArchitecture≫ only specifies the structural constituents of a component type. The rules that declare the component type's behavior have to translated from the corresponding UML behavior specification and the same restrictions and difficulties the were already discussed in Section 4.3.2 apply here, too.

4.3.4 Safety Properties

The previous sections clearly showed, that *SoaML* models do only specify a part of the information we need for complete *rigSoaML* model. Especially the safety properties that are required for collaboration and component types are missing. The problem here is, that *SoaML* wasn't designed to also declare safety properties within the system model. Hence, no defined way to specify them exists. However, it might be possible to specify them using OCL constraints that restrict the allowed instance situations, one could use UML sequence diagrams to specify invalid behavior or explicitly point out those behavior sequences that are allowed.

In consequence we can not give a precise and automatic way to derive the safety properties for collaboration and component types.

4.4 Discussion

In this section we want to review how well *rigSoaML* covers the challenges that result for the modeling as introduced in Chapter 2.

Challenge Modeling SOA (M1) is as outlined in Section 3.2 already mostley covered by *SoaML* and thus *rigSoaML* inherit this coverage. The model has to cover the concepts of service-oriented systems such as service contracts, roles, components, architecture and service landscapes.

However, *rigSoaML* in contrast to *SoaML* (see Section 3.2) also covers challenge Modeling Dynamics

(M2). *rigSoaML* supports dynamics of service-oriented systems such as joining/leaving service contracts dynamically and adding components dynamically by means of the rules for roles and components.

The evolution can be modeled by step-wise adding types to the system type. Thus, also challenge Modeling Evolution (E1) concerning the modeling of the uncoordinated introduction of new types for service contracts and components at runtime is covered. Further, the newly introduced evolution diagram illustrates the different changes, that occurred to the system during its lifetime.

Challenges	*SoaML*	*rigSoaML*
Modeling SOA (M1)	✓	✓
Modeling Dynamics (M2)	○	✓
Modeling Evolution (E1)	○	✓

Table 4.1: Comparison of the coverage of the challenges for modeling with *SoaML* and *rigSoaML*

Chapter 5

Verification of Complex Landscape rigSoaML Models

The problem of verifying the correctness of *complex landscapes* is that existing formal approaches do not scale, are not applicable to the specific settings of SOA with dynamic binding, and do not support evolution. Formal verification usually operates at the level of instances and does only work for rather small configurations with a fixed upper bound of elements and a fixed number of initially defined element types, while complex landscapes may contain unbounded many elements and even the defined element types may evolve. Therefore, instance-based formal verification approaches that look at a particular configuration are in principle not applicable. For the same reasons also testing a particular configuration, that does provide an even low coverage than formal verification, is not sufficient as well. Furthermore, due to the dynamic nature of *complex landscapes* it cannot be assumed that any of the involved organization has all relevant details of the concrete service implementations at hand to apply formal verification techniques or testing techniques looking at the complete configuration.

Therefore, we propose to instead establish the required guarantees for the correctness for the collaboration and component types as introduced for the suggested modeling approach. We will at first simplify the problem by only considering landscapes with concrete type definitions (Section 5.1). Then, we will in an additional step also consider abstract service contracts as a means to bind independently developed components to each other and refine collaborations (Section 5.2). Finally, also the very demanding case of landscape evolution where new collaboration and component types enter the scene is considered (Section 5.3).

5.1 Concrete Service Landscapes

The simplification to approach the problem of service landscape verification followed in this section is that any concrete system only instantiates concrete types and thus we will in a first step omit the abstract types and evolution.

As a formal verification at the instance level seems impossible, we will instead approach the problem at the type level. For the verification at the type level we will then show that the correctness proven for the collaboration and component types and only type conformance for the system type will by construction imply that the related correctness also holds at the instance level for any possible configurations of related systems. The general idea of our verification approach is sketched in Figure 5.1. The figure is virtually separated into two layers. The bottom layer shows the actual instance situation, for which

we want to come up with a correctness-proof. The top layer shows the types that are instantiated at the instance level – illustrated through the dashed vertical arrows. At the top-level grey boxes indicate verification obligations, i. e. we have to verify that the RequestOfferCollaboration behaves correctly, and the "Check Role Refinement" label indicates that we have ensure that the component correctly refines the collaboration's roles. The scalability of our approach comes from the fact, that the type view is to some degree independent of the instance view.[1] What we have to show as a general property of our approach is that the results we yield for the type-level are valid for the instance-level, too.

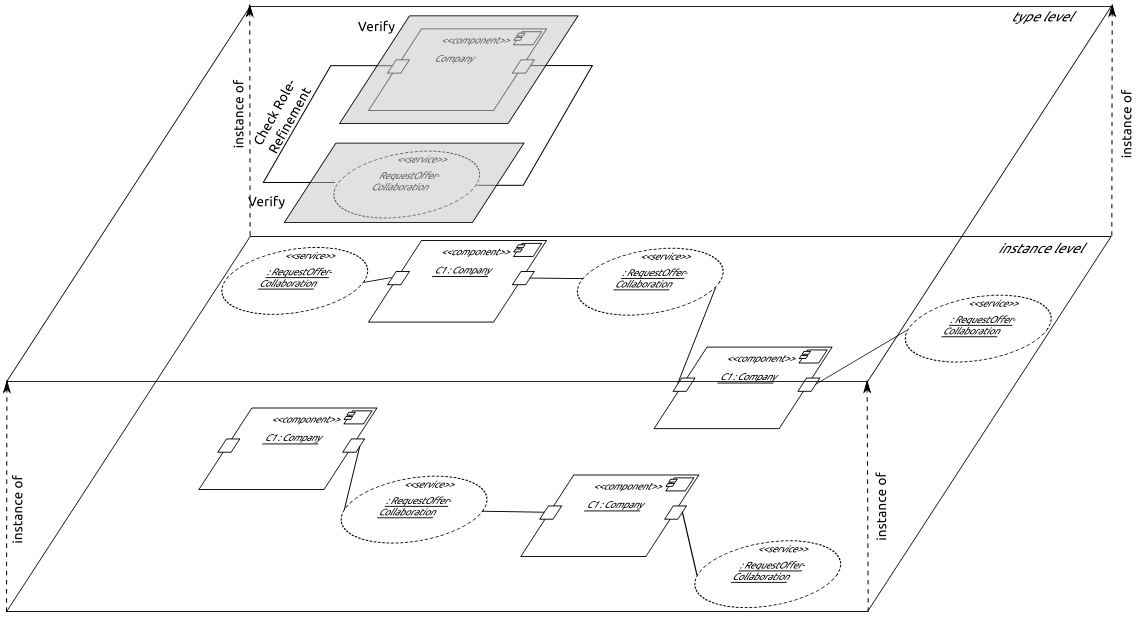


Figure 5.1: Sketch of the general proof scheme

5.1.1 Correct Collaboration Types

We start our considerations with defining what we mean by correct types for collaborations and components. A correct collaboration type requires that the resulting behavior ensures the guarantees if the assumptions are guaranteed.

Definition 5.1
A concrete collaboration type $Col_i = (col_i, (ro_i^1, \ldots, ro_i^{n_i}), CD_i, R_i\Phi_i)$ *is* correct *if for the empty configuration* $G_\emptyset$ *holds that the reachable collaboration configurations are correct*

$$G_\emptyset, R_i(Col_i) \models \Phi_i$$

for $R_i(Col_i) = R_i(ro_i^1) \cup \cdots \cup R_i(ro_i^n) \cup R_i(col_i))$ *the overall behavior of the collaboration.*

Please note that looking only at the behavior of all roles and to consider only the initial empty $G_\emptyset$ is sufficient to cover all possible behavior as only the behavior of the roles can create or delete roles or other exclusive elements.

Example 7: To exemplify a correct collaboration type we will use the concrete collaboration type Request Offer Collaboration (ROC), that has been introduced in Example 4. The ROC has two concrete role types $\text{CUSTOMER}_{\text{ROC}}$ *and* $\text{SUPPLIER}_{\text{ROC}}$*, a network behavior is not specified. The properties the ROC*

[1] As long as the types, that are instantiated at the instance level, do not change, the type level does not change.

has to fulfill are those that are specified for the ROC. Consequently the property that has to be satisfied by the ROC is $AG\neg\exists \mathtt{noSupplier} \wedge AG\neg\exists \mathtt{earlyRecall} \wedge AG\neg\exists \mathtt{earlyRequest} \wedge AG\neg\exists \mathtt{earlyOffer}$. *This property has to be satisfied by every graph transformation system that can be build using any valid configuration as initial graph and the collaboration's rules.*

The following table summarizes the checks that are necessary to show that the ROC is a correct collaboration.

Task	*Required?*	*Time*	*Memory*
Verify Φ_{CC}	*yes*		
- Check $G_\emptyset, R_{ROC}(Col_{roc}) \models \Phi_{CC}$	*yes*	*1668 ms*	*≤512 MB*
Verify Φ_{ROC}	*yes*		
- Check $G_\emptyset, R_{ROC}(Col_{roc}) \models \Phi_{ROC}$	*yes*	*5703ms*	*≤512 MB*

The set of rules $R_{ROC}(Col_{ROC})$ *is given as the combination of all rules of the RequestOfferCollaboration's roles:* $R_{ROC}(Col_{ROC}) = R_{ROC}(\mathit{Supplier}_{ROC}) \cup R_{ROC}(\mathit{Customer}_{ROC})$.

5.1.2 Correct Components Types

A correct component type requires that the resulting behavior ensures that the guarantees if the assumptions are guaranteed and that the component's implementation refines the combined role behavior.

Definition 5.2
A concrete component type $Com_i = (com_i, (ro_i^1, \ldots, ro_i^{m_i}), CD_i, I_i, \Psi_i)$ *is* correct *iff for the empty configuration* $G_\emptyset$ *holds that the reachable component configurations are correct*

$$G_\emptyset, R_i(com_i) \cup COMP(Com_i) \cup I_i \models \Psi_i \qquad (1)$$

and the component behavior $R_i(com_i)$ *refines the orthogonally combined role behavior and creation behavior*

$$R_i(com_i) \sqsubseteq R_i(ro_i^1) \cup \cdots \cup R_i(ro_i^{m_i}) \quad \cup \quad I_i \quad \cup \bigcup_{Com_i \to_{create} Com_j} I_j. \qquad (2)$$

We employ here $COMP(Com_i) = \bigcup_{1 \le l \le m_i} COMP(Com_i, ro_i^l)$ *with* $COMP(Com_i, ro_i^l) = R_j(Col_j)$ *to add the collaboration behavior for each role without the role itself which is covered by* $R_i(com_i)$ *to the component to derive a related closed behavior. To further differentiate the two elements of correctness we refer to the first condition as* correct concerning guarantees *and for the second condition as* correct concerning refinement.

Example 8: The concrete component Factory *can be proved correct. The* Factory *component owns an implementation and thus can be verified. The* Factory *has to satisfy it's own guaranteed properties* Ψ^g_{Fac} *– it's assumed properties* Ψ^a_{Fac} *are empty by definition.*

Task	*Required*	*Time*	*Memory*
Verify $G_\emptyset, R_{Fac}(Com_{Fac}) \models \Psi_{Fac}$	*Yes*	*12995ms*	*≤512MB*
Check role refinement	*yes*	*syntactically guaranteed*	

The rule set $R_{Fac}(Com_{Fac})$ *is given as defined in Definition 5.2.* $R_{Fac}(Com_{Fac}) = R_{Fac}(\mathit{Supplier}_{Fac}) \cup R_{Fac}(\mathit{Customer}_{Fac})$. *The correct role-refinement is syntactically guaranteed as the rules for the Factory component do only enhance the rules for the RequestOffer Collaboration with new types, defined in* CD_{Fac}.

5.1.3 Correct Collaboration Instances

After defining our notion of correctness for the types, we have to define what the related notion of correctness at the instance level means.

Definition 5.3
All collaboration instances of a concrete system $S = (Sys, G_S)$ *with system type* $Sys = ((Col_1, \ldots, Col_n), (Com_1, \ldots, Com_m))$ *are* correct *if it holds*

$$G_S, R(com_1) \cup \cdots \cup R(com_m) \cup R(col_1) \cup \cdots \cup R(col_n) \models \Phi_1 \wedge \cdots \wedge \Phi_n.$$

We can then show in the following Lemma that the correctness of the collaboration types ensures also the notion of correctness at the instance level.

Lemma 5.4
All collaborations of a concrete system $S = (Sys, G_\emptyset)$ *with system type* $Sys = ((Col_1, \ldots, Col_n), (Com_1, \ldots, Com_m))$ *and rule function* R *are* correct *if (1) the system type* Sys *is* type conform*, (2) all collaboration types* $Col_1, \ldots, Col_n$ *of* Sys *are* correct*, and (3) all component types* $Com_1, \ldots, Com_m$ *of* Sys *are* correct concerning refinement.
Proof. *At first we can conclude that due to the fact that the concrete collaboration types and their rules and properties are by definition* separated *by pseudo-types for* col_i *and due to (2) we further know that for all* i *holds* $(R(ro_i^1) \cup \cdots \cup R(ro_i^{n_i}) \cup R(col_i)) \models \Phi_i$*. Due to Corollary C.9 we know that* $R \cup R'$ *preserves the properties of* R *and* R' *when* R *and* R' *and the properties are pseudo-type separated and thus we get as the collaborations are pseudo-type separated that*

$$G_\emptyset, (R(ro_1^1) \cup \cdots \cup R(ro_1^{n_1}) \cup R(col_1)) \cup \\ \cdots \cup R(ro_n^1) \cup \cdots \cup R(ro_n^{n_n}) \cup R(col_n)) \models \Phi_1 \wedge \cdots \wedge \Phi_n.$$

Also for the creation rules $I_1 \cup \cdots \cup I_m$ *holds that they are pseudo-type separated and thus we get*

$$G_\emptyset, (R(ro_1^1) \cup \cdots \cup R(ro_1^{n_1}) \cup R(col_1)) \cup \\ \cdots \cup R(ro_n^1) \cup \cdots \cup R(ro_n^{n_n}) \cup R(col_n)) \cup I_1 \cup \cdots \cup I_m \models \Phi_1 \wedge \cdots \wedge \Phi_n.$$

Due to (1) we have type conformance which guarantees that the role types are only properly connected to collaboration types. Thus by replicating them as well as the creation rules for each occurrence and reordering them according to the concrete components types involved we get

$$G_\emptyset, (R(ro_1^1) \cup \cdots \cup R(ro_1^{m_1}) \cup I_1 \cup \cdots \cup I_m) \cup \cdots \cup (R(ro_m^1) \cup \\ \cdots \cup R(ro_m^{m_m}) \cup I_1 \cup \cdots \cup I_m) \cup R(col_1) \cup \cdots \cup R(col_n) \models \Phi_1 \wedge \cdots \wedge \Phi_n,$$

as replication of rules preserves the properties. We further know due to (3) that the implementation of a component refines the combined role behavior $(R(com_i) \sqsubseteq (R(ro_i^1) \cup \cdots \cup R(ro_i^{m_i}) \cup I_1 \cup \cdots \cup I_m)$ *and thus can derive the required condition for correctness of the system for all collaborations of Definition 5.3 by substitute* $R(com_i)$ *for* $(R(ro_i^1) \cup \cdots \cup R(ro_i^{m_i}) \cup I_1 \cup \cdots \cup I_m$ *as due to Corollary C.11 it is ensured that refinement preserves the property* $\Phi_1 \wedge \cdots \wedge \Phi_n$*:*

$$G_\emptyset, R(com_1) \cup \cdots \cup R(com_m) \cup R(col_1) \cup \cdots \cup R(col_n) \models \Phi_1 \wedge \cdots \wedge \Phi_n.$$

□

Example 9: To exemplify Lemma 5.4 let us consider the system type Sys*, that has been introduced in Example 6. The first condition we have to check is (1) (see Lemma 5.4). This condition is satisfied, as the component types, that occur in* Sys*, only use roles, that are introduced by collaborations, that are*

part of Sys, too. Condition (2) – the correctness of all collaboration types – has already been shown in Example 7. The remaining condition of the Lemma we have to show is condition (3), which enforces a correct refinement between the component's roles and the collaboration's roles. In detail we have to show that the roles $\text{SUPPLIER}_{\text{COM}}$ and $\text{CUSTOMER}_{\text{COM}}$ refine the roles $\text{SUPPLIER}_{\text{ROC}}$ and $\text{CUSTOMER}_{\text{ROC}}$, respectively. The role refinement for Sys is satisfied, as the component's rules (see Figure 4.10) do not alter the collaboration's rules (see Figure 4.7) other than strengthening the rules' preconditions. In consequence the component's rules do not introduce new situations, where the rules are applicable, but reduce the amount of rule applications.

Summarizing, we have shown that any concrete system $S = (G_\emptyset, Sys)$ where $G_\emptyset$ is the initial empty configuration contains only correct collaboration instances.

5.1.4 Correct Component Instances

As done for collaborations, we now define what the related notion of correctness at the instance level for components means.

Definition 5.5
All components of a system $S = (Sys, G_S)$ with system type $Sys = (\mathcal{T}, R, (Col_1, \ldots, Col_n), (Com_1, \ldots, Com_m))$ are correct *if it holds:*

$$G_S, R(com_1) \cup \cdots \cup R(com_m) \cup R(col_1) \cup \cdots \cup R(col_n) \models \Psi_1 \wedge \cdots \wedge \Psi_m.$$

We can show in the following Lemma that the correctness of the component types ensures also the notion of correctness at the instance level.

Lemma 5.6
All components of a system $S = (Sys, G_\emptyset)$ with system type $Sys = ((Col_1, \ldots, Col_n), (Com_1, \ldots, Com_m))$ are correct *if (1) the system type Sys is* type conform*, (2) all collaboration types $Col_1, \ldots, Col_n$ are* correct*, and (3) all component types $Com_1, \ldots, Com_m$ are* correct*.*
Proof. *As for all i with $1 \leq i \leq m$ holds due to (2) that $R(com_i) \cup COMP(Com_i) \cup I_i \models \Psi_i$ and all $R(com_i) \cup COMP(Com_i) \cup I_i$ are included in a refined manner in $R(com_1) \cup \cdots \cup R(com_m) \cup R(col_1) \cup \cdots \cup R(col_n)$, we finally get due to Corollary C.9 the required result*

$$G_\emptyset, R(com_1) \cup \cdots \cup R(com_m) \cup R(col_1) \cup \cdots \cup R(col_n) \models \Psi_1 \wedge \cdots \wedge \Psi_m.$$

The disjoint type-graphs, required by Corollary C.9 are achieved through pseudo-typing and pseudo-types preserving rules. Pseudo-typing guarantees that not two component instances can influence each other directly. □

Example 10: For the system type Sys (cf. Example 6) we have only shown, yet, that the collaboration instances in any system configuration are correct (cf. Example 9). The remaining proof, that the component instances are correct, too, is a combination of our previous results. According to Lemma 5.6 we have to show, that the system type is correct (cf. Example 9), the collaborations types are correct (cf. Example 7) and that the component types are correct (cf. Example 8). Following, the above lemma yields that the system type contains only correct component instances, if the system started from a correct configuration.

Example 11: For the system type Sys (cf. Example 6) we have only shown, yet, that the collaboration instances in any system configuration are correct (cf. Example 9). The remaining proof, that the component instances are correct, too, is a combination of our previous results. According to Lemma 5.6 we have to show, that the system type is correct (cf. Example 9), the collaborations types are correct

(cf. Example 7) and that the component types are correct (cf. Example 8). Following, the above lemma yields that the system type contains only correct component instances, if the system started from a correct configuration.

5.1.5 Correct Systems

As done for collaborations and components, we now define what the related notion of correctness at the instance level for systems means.

Definition 5.7
A concrete system $S = (Sys, G_S)$ *with system type* $Sys = ((Col_1, \ldots, Col_n), (Com_1, \ldots, Com_m))$ *is* correct *if it holds:*

$$G_S, R(com_1) \cup \cdots \cup R(com_m) \cup R(col_1) \cup \cdots \cup R(col_n) \models \Phi_1 \wedge \cdots \wedge \Phi_n \wedge \Psi_1 \wedge \cdots \wedge \Psi_m.$$

We can then show in the following Theorem 5.8 that the type conformance of the system type and the correctness of collaboration types and component types ensures correctness at the instance level for the system.

Theorem 5.8
An system $S = (Sys, G_\emptyset)$ *with system type* $Sys = ((Col_1, \ldots, Col_n), (Com_1, \ldots, Com_m))$ *is* correct *if (1) the system type* Sys *is* type conform*, (2) all collaboration types* $Col_1, \ldots, Col_n$ *are* correct*, and (3) all component types* $Com_1, \ldots, Com_m$ *are* correct*.*
Proof. *The required result follows directly from combining the results of Lemma 5.4 and 5.6 as both require the same or a weaker conditions and their composed results are equal to the required conclusion.* □

The presented Theorem 5.8 provides sufficient but not necessary conditions to ensure the correctness. It permits to straight forward establish the required correctness of the types by checking refinement and the guarantees for the properties using the rule sets as employed in condition (2) and (3).

Figure 5.1 (see Page 42) visualizes that, according to Theorem 5.8, the required guarantees for the instance level can be established by only do checks at the type level. Therefore, we can conclude that the complexity of checking the guarantees is only depending on the number of types and the complexity of the checking problems of the collaboration and component types.

Example 12: Let us assume, that the System $S = (Sys, G_\emptyset)$ *is a concrete system of system type* Sys *(cf. Example 6) and empty starting configuration* $G_\emptyset$*. In the previous Examples 12, 11, and 9 we have shown that the conditions for Theorem 5.8 hold. We can thus conclude, that the System* S *is correct.*

Task	*Required?*	*Time*	*Memory*
Verify Φ_{ROC}	*yes*		
- Check $G_\emptyset, R_{ROC}(Col_{roc}) \models \Phi_{ROC}$	*yes*	*5703ms*	*≤512 MB*
Verify $G_\emptyset, R_{Fac}(Com_{Fac}) \models \Psi_{Fac}$	*Yes*	*12995ms*	*≤512MB*
Check role refinement	*yes*	*syntactically guaranteed*	
Sum		*18698ms*	*≤512MB*

5.2 Service Landscapes and Abstraction

In the preceding section we have shown that the correctness of concrete component and collaboration types by construction implies the correctness of all instances in a concrete system if it is type conform. However, in complex landscapes the cooperation is usually not only defined using concrete collaborations but also using abstract ones which allow that the concrete service contract participants can refine the roles as suitable for their specific needs while still protecting their own IP.

To extend the results also to the abstract collaborations and components, we can exploit the fact that no instances of abstract collaborations or components can exist, as the abstract concepts themselves are never manifested in a system.

Furthermore, the required refinement relation between abstract collaborations and abstract components and more concrete counterparts will ensures the required guarantees implied by the abstract collaborations or components are also implied by all concrete collaborations and components refining them.

5.2.1 Refining Role Types

The refinement resp. subtyping of role requires that the resulting behavior is a refinement.

Definition 5.9
For concrete role types ro_i and ro_j holds that ro_i refines *ro_j (written $ro_i \sqsubseteq ro_j$) if holds*

$$R(ro_i) \sqsubseteq R(ro_j),$$

5.2.2 Refining Collaboration Types

The refinement resp. subtyping of collaboration requires that with respect to all roles of the refined collaboration the resulting behavior is still a refinement.

Definition 5.10
For a collaboration type $Col_i = (col_i, (ro_i^1, \ldots, ro_i^{n_i}), CD_i, \Phi_i)$ and an abstract collaboration type $Col_j = (col_j, (ro_j^1, \ldots, ro_j^{n_j}), CD_j, \Phi_j)$ holds that Col_i refines *Col_j (written $Col_i \sqsubseteq Col_j$) if holds $n_j \leq n_i$ and the refinement results in stronger guarantees and that the refined assumptions plus guarantees imply the assumptions*

$$\Phi_i \Rightarrow \Phi_j \qquad (1).$$

For all Col_i and its super type Col_j holds that the subtype relation is correct *if $Col_i \sqsubseteq Col_j$.*

For subtyping of collaborations we can show in the following Lemma that the guarantees of the correct refined collaboration are preserved by refinement resp. subtyping.

Lemma 5.11
For a correct, concrete collaboration type $Col_i = (col_i, (ro_i^1, \ldots, ro_i^{n_i}), CD_i, \Phi_i)$ and any of its abstract collaboration super types $Col_j = (col_j, (ro_j^1, \ldots, ro_j^{n_j}), CD_j, \Phi_j)$ ($Col_i \sqsubseteq Col_j$) holds

$$R(Col_i) \models \Phi_j.$$

Proof. *For any collaboration type holds via induction that its local assumption and guarantees imply the guarantees of any super type Col_j ($\Phi_i \Rightarrow \Phi_j$). As for a correct, concrete collaboration holds by definition $R(Col_i) \models \Phi_i$. Consequently, we can conclude $R(Col_i) \models \Phi_j$.* □

Example 13: The Request Offer Collaboration *(ROC, for specification see Example 4) refines the abstract collaboration* Contract *(Con, cf. Example 3). For the* Contract *Collaboration we had to run the following checks:*

Task	*Required*	
Verify $R(Con) \models \Phi_{Con}$	*yes*	*see Example 7*

Obviously, verification for the Contract *Collaboration is comparatively easy as this collaboration only specifies a few properties and does not inherit properties from super-collaborations. The* Request Offer Collaboration*, however, inherits from the* Contract *Collaboration (see Figure 4.1) and thus has to satisfy* $\Phi^g_{Con} \wedge \Phi^g_{ROC}$.

Task	*Required*	
Verify $R(Con) \models \Phi_{Con}$	*No*	
Verify $R(ROC) \models \Phi_{ROC}$	*Yes*	*see Example 7*
check refinement	*Yes*	*syntactically checked*

For the Request Offer Collaboration *it is not required to verify the property* Φ^a_{Com} *again, as the collaboration's rules refine the* Contract *collaboration's rules. The rule refinement holds by construction, as for the* Request Offer Collaboration*'s* createContract *rule only the precondition of the* Contract *collaboration's* createContract *rule had to be strengthened. Therefore, according to Lemma 5.11, the verification results for the property* Φ^a_{Com} *and the* Contract *collaboration also hold for the* Request Offer Collaboration.

5.2.3 Refining Component Types

The refinement resp. subtyping of components also requires that the resulting behavior is a refinement.

Definition 5.12
For a component type $Com_i = (com_i, (ro_i^1, \ldots, ro_i^{m_i}), CD_i, I_i, \Psi_i)$ *and an abstract component type* $Com_j = (com_j, (ro_j^1, \ldots, ro_j^{m_j}), CD_j, I_j, \Psi_j)$ *holds that* Com_i refines Com_j *(written* $Com_i \sqsubseteq Com_j$*) if it holds* $m_j \leq m_i$ *and the refinement results in stronger guarantees and weaker assumptions*

$$\Psi_i \Rightarrow \Psi_j \qquad (1).$$

For all Com_i *and its super type* Com_j *holds that the subtype relation is* correct *if* $Com_i \sqsubseteq Com_j$.

For subtyping of components we can show in the following Lemma that the guarantees of the refined components are preserved by refinement resp. subtyping.

Lemma 5.13
For a correct, concrete component type $Com_i = (com_i, (ro_i^1, \ldots, ro_i^{m_i}), CD_i, I_i, \Psi_i)$ *and any of its correct component super types* $Com_j = (com_j, (ro_j^1, \ldots, ro_j^{m_j}), CD_j, I_j, \Psi_j)$ *(*$Com_i \sqsubseteq Com_j$*) holds*

$$R(Com_i) \models \Psi_j.$$

Proof. *For any component type holds via induction that its local assumption and guarantees imply the guarantees of any super type* Com_j *(*$\Psi_i \Rightarrow \Psi_j$*). As for a correct, concrete component holds by definition* $R(Com_i) \models \Psi_i$*. Consequently, we can conclude* $R(Com_i) \models \Psi_j$*. In case of an abstract refined component type and a correct refining component type it also holds due to condition (3).* □

Example 14: Let us take a look at the FACTORY *component (cf. Example 5) to exemplify Lemma 5.13. The* FACTORY *component's super type is the abstract component* ABSTRCOMP *(cf. Example 5). In order to safely omit the check, that the* FACTORY *also satisfies the* ABSTRCOMP*'s safety properties we have, according to the above lemma, to show that the rule sets assigned to the components' role are in a valid refinement relation (see Definition 5.12). More concrete we have to show the following:*

$$R_{Com}(Customer^{CC}) \sqsubseteq R_{ACom}(Customer) R_{Com}(Supplier^{CC}) \sqsubseteq R_{ACom}(Supplier)$$

The conditions for a correct refinement between rule sets can be informally described as: the refining rule set must not allow more behavior than the refined rule-set. The difference between the two rule sets is that the rules of the FACTORY*'s roles have a stronger pre-condition than the rules specified for the* ABSTRCOMP*. Hence, an easy syntactical check yield the required property.*

5.2.4 Correct Systems with Abstraction

It now remains to show that the refinement ensure correctness for a system including the guarantees for the abstract concepts. The following Theorem 5.14 then demonstrates that this correctness criteria is met by construction if all types are correct and the refinement conditions for subtypes are fulfilled.

Theorem 5.14
An system $S = (Sys, G_S)$ with system type $Sys = ((Col_1, \ldots, Col_n), (Com_1, \ldots, Com_m))$ is correct *if (1) the system type Sys is* type conform*, (2) all collaboration types $Col_1, \ldots, Col_n$ are* correct*, and (3) all component types $Com_1, \ldots, Com_m$ are* correct*.*
Proof. *Due to Theorem 5.8 we can conclude that all properties of the concrete collaborations and components are preserved. Based on the type conformance of the system type Sys, Lemma 5.13 and Lemma 5.13 further guarantee that also all properties of the abstract collaborations and components are preserved.* □

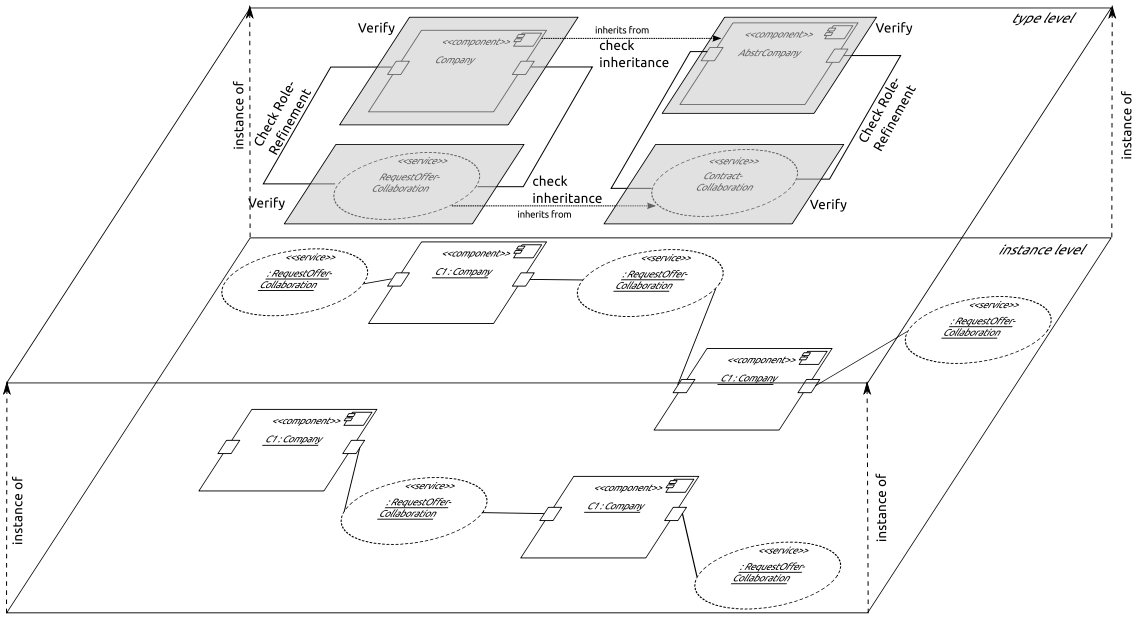


Figure 5.2: Verification scheme for the verification of complex landscapes with abstraction

The general proof scheme including abstraction is depicted in Figure 5.2. The sketch differs from the one in Figure 5.1 in such way as the type-level now contains abstract collaborations and components,

too. Beside the already known verification obligation, we now also have to ensure that the inheritance between the abstract and concrete entities is correct.

Example 15: Combining the system definition we gave in Example 6 and the examples on correct collaboration and component types, 7 and 8, respectively, we can conclude that the system is correct, too. The necessary arguments for Theorem 5.14 are all given in the respective examples.

The overall effort to verify the complete system can be seen in the following table. Note, that Con and ROC are the shorthands for the Contract *and the* RequestOfferCollaboration*, respectively.*

Task	*Required?*	*Time*	*Memory*
Verify Φ_{CC}	*yes*		
- Check $G_\emptyset, R_{ROC}(col_{roc}) \models \Phi_{CC}$	*yes*	*1668 ms*	*≤512 MB*
Verify Φ_{ROC}	*yes*		
- Check $G_\emptyset, R_{ROC}(col_{roc}) \models \Phi_{ROC}$	*yes*	*5703ms*	*≤512 MB*
Task	*Required*	*Time*	*Memory*
Verify $G_\emptyset, R_{Fac}(Com_{Fac}) \models \Psi_{Fac}$	*Yes*	*12995ms*	*≤512MB*
Check role refinement	*yes*	*syntactically guaranteed*	
Summarized		*20366ms*	*≤512MB*

Hence, the compositional capabilities of our approach allows us to sum up the times for the different verification steps and avoid any overhead, that might be introduced through the composition of collaboration-types and component-types. We in sum only need a little longer than 20 seconds to verify a system or arbitrary size.

5.3 Service Landscape and Evolution

So far the presented results do not cover evolution. Therefore, in this section we will extend the former results to also cover typical evolution scenarios such as adding new collaboration or component types.

If we look at our former result in more detail, we can see that the assumptions have been made that all types are known at verification time. Furthermore, the transitive nature of the refinement required for subtyping has been employed to also support abstraction along the static subtyping relation spanning essentially a fixed finite tree of types.

These assumptions are not true for a steadily evolving system where type definition are added over time and where the subtyping tree is thus not necessarily fixed. Furthermore, the different organizations involved will only have partial view on the subtyping tree and the types they want to add and thus all types cannot be not known.

For a given *extended evolution sequence* as defined in Definition 4.7 we can define correctness as follows:

Definition 5.15
An extended evolution sequence $(Sys_1, G_S^1), \ldots, (Sys_n, G_S^n)$ *with* $Sys_n = ((Col_1, \ldots, Col_p), (Com_1, \ldots, Com_q))$ *is* correct *if for any combined path* $\pi_1 \circ \cdots \circ \pi_n$ *such that* π_i *is a path in* Sys_i *leading from* G_S^i *to* G_S^{i+1} *for* $i < n$ *and that* π_n *is a path in* Sys_n *starting from* G_S^n *holds*

$$\pi_1 \circ \cdots \circ \pi_n \models \Phi_1 \wedge \cdots \wedge \Phi_p \wedge \Psi_1 \wedge \cdots \wedge \Psi_q.$$

An evolution sequence $Sys_1, \ldots, Sys_n$ *is* correct *iff all possible related extended evolution sequence* $(Sys_1, G_S^1), \ldots, (Sys_n, G_S^n)$ *are correct.*

A first observation is that Sys_n contains all types defined in any Sys_i. However, for a combined path $\pi_1 \circ \cdots \circ \pi_n$ such that π_i is a path in Sys_i leading from G_S^i to G_S^{i+1} for $i < n$ holds not in general that an equal path π in Sys_n exists that goes through all G_S^i, as the rules added by later added types may influence the outcome. E. g., they may be urgent and thus have to be executed or may block other rules due to a higher priority.

However, we can exploit the above observation and construct a related system type that includes all possible combined paths of any possible extended evolution sequences for a given evolution sequence. We further abstract from the concrete ordering and only distinguish types that are defined already in Sys_1 or added later.

Definition 5.16
An dynamically evolving collaboration type $E(Col_i) = (col_i, Ro_i^1, \ldots, Ro_i^{n_i}), \Phi_i)$ for a collaboration type $Col_i = (col_i, (Ro_i^1, \ldots, Ro_i^{n_i}), \Phi_i)$ results by adding a special collaboration node type t_i^{Col}, extending all rules of $Ro_i^1, \ldots, Ro_i^{n_i}$, and col_i such that one node of type t_i^{Col} is an additional condition to be enabled, and add a special rule r_i^{Col} to $R(col_i)$ that creates at most one node of type t_i^{Col} using a NAC and has only the additional pre-condition that all types it depends on have been activated already (their respective node exists).

Example 16: Let us take the collaboration type for the RequestOffer-Collaboration, we have defined in Example 4, and change it into an evolving collaboration type. In accordance with Definition 5.16 we have to change the collaboration's rule set by adding the special rule r_{ROC}, which is shown in Figure 5.3.

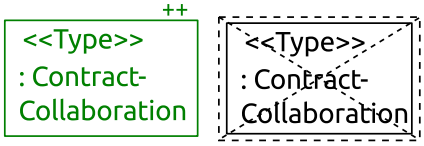


Figure 5.3: Special rule r_{ROC} for the RequestOfferCollaboration

The Contract Collaboration *does not depend on any other component or collaboration type. Therefore the precondition of rule r_{ROC} does only contain the NAC that prevents the rule from being applied more than once. However if the* Contract Collaboration *would depend on other collaboration types they would occur in the rule's precondition.*

Definition 5.17
Analogously, a dynamically evolving component type $E(Com_i) = (com_i, (ro_i^1, \ldots, ro_i^{m_i}), CD_i, R_i, I_i, \Psi_i)$ for a component type $Com_i = (com_i, (ro_i^1, \ldots, ro_i^{m_i}), CD_i, R_i, I_i, \Psi_j)$ results by adding a special component node type t_i^{Com}, extending all rules of com_i such that one node of type t_i^{Com} is an additional condition to be enabled, and adding a special rule r_{com_i} to $R(com_i')$ that creates at most one node of type t_i^{Com} using a NAC and having only the additional pre-condition that all types it depends on have been activated already (their respective node exists).

Example 17: As we have done it for the collaboration type RequestOffer-Collaboration in the previous example, let us change the Factory component type, introduced in example 5, into an dynamically evolving component type as defined in Definition 5.17. The required addition rule is depicted in Figure 5.4. The Factory *component type depends on the* ContractCollaboration *collaboration type. The dependency is due to the fact that the* Factory *component type implements roles, that are defined in the* ContractCollaboration *collaboration type. Thus, it is important that the* ContractCollaboration *type is present in the system, before the* Factory *type gets introduced. This is specified in the above rule by adding the* ContractCollaboration *type to the rule's precondition.*

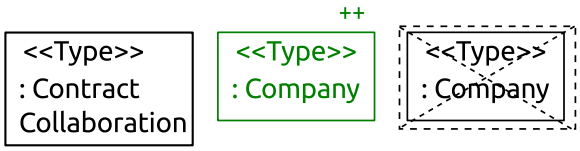


Figure 5.4: Special rule r_{Comp} for the Factory component

Definition 5.18
Given a system type $Sys_1 = ((Col_1, \ldots, Col_p), (Com_1, \ldots, Com_q))$ *and given another system type* $Sys_n = ((Col_1, \ldots, Col_p, \ldots, Col_{p+r}), (Com_1, \ldots, Com_q, \ldots, Com_{q+s}))$ *extending the first one the related dynamically evolving system type is given as*

$$E(Sys_1, Sys_n) = ((Col_1, \ldots, Col_p, E(Col_{p+1}), \ldots, E(Col_{p+r})), \\ (Com_1, \ldots, Com_q, E(Com_{q+1}), \ldots, E(Com_{q+s})))$$

.

Example 18: To exemplify Definition 5.18 we can construct two small systems $S_1 = ((Col_{ROC}), \emptyset)$ *and* $S_2 = ((Col_{ROC}), (Com_{Comp}))$*. The corresponding evolving system type* $E(S_1, S_2)$ *can then be specified as:*

$$E(S_1, S_2) = ((Col_{ROC}), (E(Com_{Comp})))$$

Hence, as the collaborations types do not change within the evolutionary step from S_1 *to* S_2 *we do not have to alter the set of collaboration types in the dynamically evolving system type* $E(S_1, S_2)$*. However, the set of component types changes, i. e. the Factory component is added, and thus we have to add this component's dynamically evolving component type (see Example 17) to* $E(S_1, S_2)$*.*

We can now exploit the fact that the related dynamically evolving system type includes all possible extended evolution sequences to check also the correctness of an evolution sequence.

Theorem 5.19
An evolution sequence of systems $Sys_1, \ldots, Sys_n$ *is* correct *if the related dynamic evolving system type* $E(Sys_1, Sys_n)$ *is correct.*
Proof. *For any extended evolution sequence* $(Sys_1, G_S^1), \ldots, (Sys_n, G_S^n)$ *for* $Sys_1, \ldots, Sys_n$ *and any combined path* $\pi_1 \circ \cdots \circ \pi_n$ *such that* π_i *is a path in* Sys_i *leading from* G_S^i *to* G_S^{i+1} *holds that a related path* $\pi_1' \circ \cdots \circ \pi_n'$ *such that* π_i' *is a path in* $E(Sys_1, Sys_n)$ *leading from* G_S^i *to* G_S^{i+1} *exists such that* $\pi_i' = E(\pi, Sys_1, Sys_n) \circ \pi_r$*. The rule* π_r *is an arbitrary sequential combination of all* r_{col_i} *and* r_{com_j} *for collaborations and components that are in* Sys_{i+1} *but not* Sys_i*. Consequently, if* $E(Sys_1, Sys_n)$ *has been proven correct, we can conclude that also all extended evolution sequence* $(Sys_1, G_S^1), \ldots, (Sys_n, G_S^n)$ *have to be correct and thus* $Sys_1, \ldots, Sys_n$ *must be correct.* □

In the following Corollary 5.20 we can characterize what is required when a evolution sequence is extended by adding new types for collaborations and components.

Corollary 5.20
An evolution sequence of systems $Sys_1, \ldots, Sys_n$ *with* $Sys_{n-1} = ((Col_1, \ldots, Col_p), (Com_1, \ldots, Com_q))$ *and* $Sys_n = ((Col_1, \ldots, Col_p, \ldots, Col_{p+r}), (Com_1, \ldots, Com_q, \ldots, Com_{q+s}))$ *is* correct *if (1)* $Sys_1, \ldots, Sys_{n-1}$ *is correct (using the conditions of Theorem 5.19) and (2) if all* $p < i \leq p + r$ $E(Col_i)$ *are correct and all* $q < j \leq q + s$ $E(Com_j)$ *are correct, and (3) if all subtype relations for any* Col_i *with* $p < i \leq p + r$ *and any* Com_j *with* $q < j \leq q + s$ *are correct.*
Proof. *From (1), (2) and (3) we can directly construction the conditions to prove* $E(Sys_1, Sys_n)$ *when all conditions for* $E(Sys_1, Sys_{n-1})$ *have been proven already.* □

Corollary 5.20 provides a direct guideline what has to be done when you want to add a new type for a collaboration or component. Note that an organization which wants to extend the system type accordingly does not require to know all other types besides those which are refined. Furthermore, if two independent extensions are done which do not refer to each other, the concrete order does not matter as the checks remain the same. Therefore, each organization can simply check its own extension and the ordering how they are enacted does not matter.

Lemma 5.21
For a correct collaboration type Col holds also that its dynamic extension $E(Col)$ is correct. For a correct component type Com holds also that its dynamic extension $E(Com)$ is correct.
Proof. *Due to its construction the additional rule does not affect the correctness as for any trace of $E(X)$ holds that it must start with an initial delay and then the additional rule while the rest equals a trace for X. As the additional rule has an arbitrary timing, when eliminating the additional rule we simply get traces that equal those of X and we can conclude that if a property is violated in $E(X)$ it would also be violated in X and vice versa.* □

Consequently, it is thus sufficient due to Lemma 5.21 to simply check the collaboration and component types and this already guarantees that any extended evolution sequence will aslo show correct behavior.

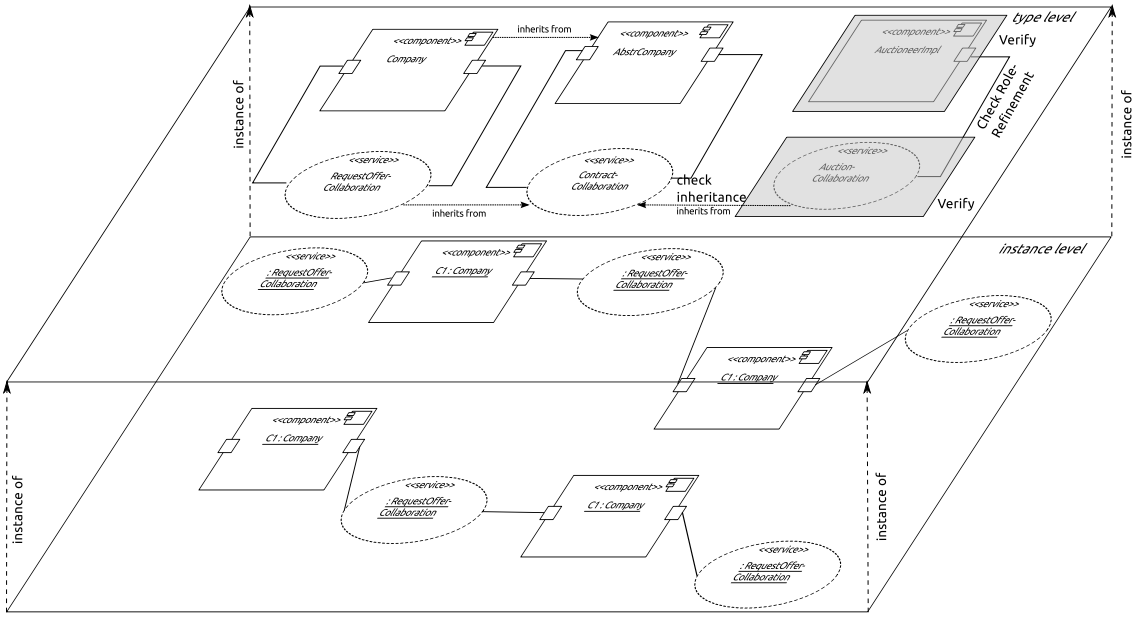


Figure 5.5: Incremental verification scheme for the verification of complex landscapes with evolution

Figure 5.5 depicts the incremental verification scheme for our verification approach. In the figure we assume that the evolutionary step consists of adding the component-type AuctioneerImpl and the collaboration-type Auction. The necessary verification steps are mentioned within the figure.

Example 19: We exemplify the incremental verification of evolving complex landscapes with the introduction of a new implementation of the Contract *collaboration. The* Auction *collaboration requires a new role* Auctioneer *and changes the negotiation pattern between* Supplier *and* Customer. Suppliers *create an auction together with an* Auctioneer *and as long as the auction is running* Suppliers *can send bids to the* Auctioneer. *The* Auctioneer *checks the bid, whether or not it is higher than the currently leading bid and marks the bid as leading or discards it. The* Auction *collaboration requires to introduce a new role* Auctioneer, *a collaboration* Auction *that refines the* Contract *collaboration and the two specialized roles* Supplier *and* Customer.

The Auction *collaboration type is newly introduced and thus we have to verify that it satisfies it's own safety properties. Further the* Auction *collaboration type refines the* Contract *collaboration type and thus has to satisfy the properties ϕ_{CC}. The correct refinement between* Auction *and* Contract *collaboration type has to be checked, too. As well as the refinement between roles and new component types. However, if we can show that the new component types and the* Auction *collaboration type are correct, we can use Corollary 5.20 to conclude that the evolution sequence of systems Sys_1, Sys_2, Sys_3 where Sys_1 is our application example as explained prior to this example, Sys_2 is Sys_1 and the* Auction *collaboration type and Sys_3 is Sys_2 extended by the* AuctioneerImpl *component type, is correct. The following table gives an overview of the necessary checks:*

Task	*Required?*
Verify Φ_{Auc}	*yes*
- Check $G_\emptyset, R_{Auc}(col_{Auc}) \models \Phi_{Auc}$	*yes*
Verify Φ_{CC}	*yes*
- Check $g^\emptyset, R_{Auc}(col_{Auc}) \models \Phi_{CC}$	*no*
- Check correct refinement	*yes*
Task	*Required?*
Verify $G_\emptyset, R_{Auctioneer}(Com_{Auctioneer}) \models \Psi_{Auctioneer}$	*yes*
Check role refinement	*yes*

5.4 Discussion

While for *SoaML* no proper analysis support exists, for *rigSoaML* we have outlined how even complex service landscapes can be analyzed. In detail we have provided lemmata that explicitly state which pre-conditions have to be met in order to verify service-oriented systems. The basic idea, we follow, is to start with the verification of small entities – i. e. collaboration- and component-types – and compose these results into an argument for the overall system's correctness. The compositionality of our approach is expressed in Theorem 5.8, which uses the results of correct collaboration- and component instances within a system, that have been introduced in Lemma 5.4 and Lemma 5.6, respectively. Further, we presented a more generalized variant of Theorem 5.8 that also covers abstraction. These findings accumulate in Theorem 5.14. In a last step we showed that our approach can be facilitated to verify evolution of service-oriented systems, see Theorem 5.19 and Corollary 5.20.

5.4.1 Analysis: Scalability

As visualized in Figure 5.1, 5.2, and 5.5 the verification scheme only considers the types, supports subtyping, and permits to address evolution by means of an incremental scheme where only new types and their relations to existing types have to be checked.

Furthermore, as exemplified in the examples for checking a concrete system in Example 12, a system with subtypes in Example 15, and an evolution step in Example 19 only requires to check the types. In addition, the building blocks of checking that the collaborations types are correct in Example 7 and that the component types are correct in Example 8 all require only moderate efforts. Therefore, *rigSoaML* can be checked for arbitrary large service landscapes and thus scale as demanded by challenge Scalable Analysis (A1).

5.4.2 Analysis: Applicability

The rules in *rigSoaML* permit to also model that service landscapes change their configuration at runtime and thus the model captures the regular behavior and reconfiguration behavior at once. Consequently, the outlined analysis approach fulfills challenge Analysis of Reconfiguration (A2).

For our overall example of the supply chain case study this means, that the within the specifications of components and collaborations – e. g. Factory and RequestOfferCollaboration – the rules for terminating a relation are already included. Thus we do not only consider the pure business rules of sending and receiving Request, Offer and Contract messages, but also describe exactly the conditions, which allow participants, Factory-components in this case, to leave the collaboration again.

The type based compositional approach for analysis works even if no no global view exists and there are fully separated responsibilities. The abstraction concepts further permit that IP related to component details and to some extent even the IP of service contract details can be protected. Therefore, *rigSoaML* also fulfills challenge Analysis under restricted knowledge (A3).

In terms of our supply chain example this means that we do not have to know the exact business logic of a particular component X-Factory as long as the component developer can prove that the component conforms to the AbstrFactory and it's implemented roles. Even other participants that are involved in a service contract with an instance of X-Factory do not see more of the component than the role allows and they do not need more information.

5.4.3 Evolution: Analysis

Due to the fact that the compositional analysis approach also works for evolution on the basis of the types only, the analysis can be done incrementally as required for each added type in isolation covering its local guarantees as well as the linkage to concepts it inherits guarantees or constraints from. Thus also challenge Analyzing Evolution (E2) concerning the analysis of the uncoordinated introduction of new types for service contracts and components at runtime is covered. Here it is particular important that even for the evolution an incremental checking is possible such that even evolution sequences with very large sets of defined types can be handled as long as in each evolution step only a small number of additional types are introduced.

Let us recapitulate this again by mean of our application example. One possible evolutionary step, that we have sketched, is the introduction of an Auction service contract, which differs from the RequestOfferCollaboration in the way the negotiation between Producer and Consumer is done. Further, this new service contract type required that we also introduce a new role Auctioneer and a component, implementing this role. The introduction of these two new constituents made it necessary to check for the Auction that it is safe, with respect to its own safety properties, and that is a valid sub-type of the ContractCollaboration. The same holds for the new components that implement the new roles. They have to satisfy their own safety properties, i. e. they have to be correct, and they have to be correct role refinements.

5.4.4 Summary

As outlined our approach does scale due to compositional approach and fulfills the related analysis challenges Scalable Analysis (A1)-Analysis under restricted knowledge (A3) by using abstraction to decouple the different concrete elements via abstract ones. Furthermore, the evolution challenge Analyzing Evolution (E2) is supported by an incremental and decentralized verification scheme.

Challenges	Coverage by *rigSoaML*
Scalable Analysis (A1)	✓
Analysis of Reconfiguration (A2)	✓
Analysis under restricted knowledge (A3)	✓
Analyzing Evolution (E2)	✓

Table 5.1: Coverage of the challenges for analysis with *rigSoaML*

Chapter 6

Related Work

The approach, we have presented in this report, touches several field of computer science that have already been widely investigated. In the following paragraphs we will outline the distinctions between our approach and the related pieces of work.

6.1 Modeling

Modeling using roles and focusing on collaborations rather than components is not new: Since the 1970s the OOram Software Engineering method [26] has been developed which provides a clear distinction between roles and objects and separates different collaborations in form of role models. The idea of contracts, which has been introduced in [27], also already supports a number of participants and in addition results in some *contract obligations* the classes that take over the role of the participants have to fulfill. Lohmann et al. extend in [28, 4, 29] the concept of contracts to visual contracts that also use StoryPatterns as notation. However, their technique is used for run-time checks and less for the specification of the behavior. Also a less clear historical connection between roles/collaborations and design pattern [30] exists, which is reflected today by the fact that design patterns can be modeled in UML using collaborations. The modeling of design patterns in UML is advocated in [31, 32]. The authors in [31], however, do not use UML collaborations for the modeling of design patterns, but develop a own meta-model and use UML sequence diagrams, which potentially describe partial behavior. Kent and Lauder [32] instead propose an own visual notation, but use sequence diagrams for the behavior modeling, too.

A more formal approach to the modeling of patterns and behavior is presented by Kim and Carrington in [33]. They use Object Z for modeling design patterns and their behavior. Although Object Z is a very versatile formal language, the approach of Kim and Carrington does not support dynamic collaborations.

The use of collaborations for the modeling of services has been proposed by several authors (cf. [34, 2]) as well as all proposals for a UML Profile and Meta-Model for Services [1, 35]. In [34] static but hierarchic UML collaborations and the distinction between the collaboration and the collaboration use are presented. However, the authors omit the definition of the roles' behavior. An approach not using UML that overcomes this limitation is presented in [2] which uses sequence diagrams for potentially incomplete early behavior specifications. The UML Profile [1] is conceptually similar to [34]. It further extends [34] also supporting behavior specifications for the different roles.

UML class diagrams for the structure and graph transformations for the behavior modeling are also employed in [36] to model service-oriented architectures, but in contrast to our approach services are

not modeled as collaborations. This approach has been extended to also capture "behavior-preserving architecture refinement" [37]. The refinement, however, is reduced to a reachability problem, which is then tackled with graph-transformation and model-checking tools. Our notion of refinement differs from the one in [37] as we rely on static checks for refinement.

To some extent the systems, we describe in this report, can be seen as ensembles, as introduced by Hölzl and Wirsing [38]. The formal model that is presented in [38] is very expressive, but lacks the possibility to encapsulate the services' behavior into collaboration - or similar constructs. Further, no analysis techniques exist, yet.

We can conclude that none of the modeling concepts supports dynamic collaborations as addressed in this work.

For the formal modeling of concurrent and distributed systems often process calculi are used. The best suited calculus to model service-oriented self-adaptive systems is the π-calculus [39], as it specifically allows the modeler to dynamically create new communication channels. Although, the π-calculus has been used to formally describe service-oriented systems [40, 41] and business processes, it lacks the expressiveness of attributed and typed graph-transformation systems. Nevertheless, the π-calculus is well suited to check systems for bi-similarity and refinement [42]. Approaches, that allow the model checking of π-calculus specifications are available, but typically only allow to verify a restricted subset of the π-calculus. Yang et al. describe an approach to model check π-calculus specifications with logic programming [43], but they had to forbid processes that do not contain finite replication or the parallel composition of processes. The Mobility Workbench [44] allows only a finite number of processes, too.

In the context of dynamic software updates for controllers, evolving system specifications are discussed by Ghezzi et al. [45]. The authors allow system specifications, which are a variant of life sequence charts (LSC), to be changed by adding new LSC to them. The authors then give some arguments, under which conditions a controller is in a state, such that it can be safely updated. However, this approach can not be used to our class of system as we do not have local updates, that are only valid for one controller, exclusively, but we update component and collaboration types, which influence the complete system. For the complete system, however, the current state is not known.

6.2 Verification

6.2.1 Service specific

To our best knowledge no work exists which especially addresses the problem to verify dynamic collaborations, however, a number of related approaches for the verification of service-oriented systems exist. Model checking has been employed to check business process models with varying number of active process instances. In [46, 14, 47], for example, standard BPEL models are enriched by resource allocation behavior to ensure the correct detection of deadlocks and safety violations for web services compositions under resource constraints. The same underlying analysis technique – LTSA - Labeled Transition System Analyser – is used by the authors of [48] for the verification of service compositions. This approach lacks the functionality to verify dynamic systems, as the compositions have to be known a priori. A transformation based verification technique is presented in [49]. Web-service compositions become transformed into a equivalent model, that is based on coloured petri nets and then verification tools dedicated to the verification of CP-nets are employed. This approach does not support the dynamic structural changes, that are present in our systems. The work of Cheng et al. [50] follows a similar approach. In [51] an approach dedicated to the compositional verification of middleware based software architectures is presented. The verification of a software architecture is divided into the verifi-

cation of properties, which hold for the middleware and those, which hold for the complete architecture. However the approach does not cover structural dynamics and is restricted to finite state systems.

For systems with structural dynamics like our earlier work [22] some work has been published, which does not cover dynamic collaborations to their full extent: An approach which has been successfully applied to verify service-oriented systems [36] is the one of Varró et al. It transforms visual models based on graph theory into a model-checker specific input [52]. A more direct approach is GROOVE [53] by Rensink where the checking works directly with the graphs and graph transformations. DynAlloy [54] extends Alloy [55] in such a way that changing structures can be modeled and analyzed. For operations and required properties in form of logical formulae it can be checked whether given properties are operational invariants of the system. Real-Time Maude [56], which is based on rewriting logics, is the only approach we are aware of covering structural changes as well as time. The tool supports the simulation of a single behavior of the system as well as bounded model checking of the complete state space, if it is finite. An approach that pre-computes all possible reconfigurations of a system and then applies model-checking for the verification is described by Zhang et al. [10]. This approach is not able to verify systems that don't have a finite set of possible reconfigurations, as our systems generally have.

However, all these approaches do not fully cover the problem as they require an initial configuration and only support finite state systems (or systems for which an abstracted finite state model of moderate size exist).

There are only first attempts that address the verification of infinite state systems with dynamic structure: In [57] graph transformation systems are transformed into a finite structure, called Petri graph which consists of a graph and a Petri net, each of which can be analyzed with existing tools for the analysis of Petri nets. For infinite systems, the authors suggest an approximation. The approach is not appropriate for the verification of the coordination of autonomous vehicles even without time, because it requires an initial configuration and the formalism is rather restricted, e. g., rules must not delete anything. Partner graph grammars are employed in [58] to check topological properties of the platoon building. The partner abstraction is employed to compute over approximations of the set of reachable configurations using abstract interpretation. However, the supported partner graph grammars restrict not only the model but also the properties, which can be addressed a priori.

Niebuhr and Rausch [59] advocate an approach that uses run-time testing to guarantee the correctness of dynamic adaptive systems. The authors argue that at design time a check of the correct bindings between components is not possible and hence introduce run-time testing at the binding-time of components. In our opinion this could only be an additional task to build correct systems, as situations where the binding has to be executed in order to guarantee the correctness can only be addressed through an approach that uses formal verification.

6.2.2 Compositional approaches

In [18] Gradara et al. presented an approach for the decompositional verification of Calculus of Communicating Systems (CCS) processes. The systems therefore have to be decomposed into modules, which are specified as CCS processes. The modules can be separately verified and the verification results can be combined as long as the system follows some structural constraints. System evolution is supported through the possibility to update modules. In comparison to our work they follow an bottom-up approach for the verification. Hence, the system has to be known in advance, whereas our approach is – concerning the verification – more like a top-down approach. In our approach the verification is performed at the type-level, whereas in [18] the instance-level is checked. Further, reconfiguration is not addressed.

In [17] Dam and Fredlund present an approach to verify open and distributed systems. Their approach is mainly based on the π-calculus as formal language and does not directly provide a tool for automatic

verification, however, the authors state that certain steps of the verification could be automated. Dam and Fredlund describe process networks with changing communication structures. But although their approach allows for compositional reasoning, the evolution of the system and reuse of already verified system parts is not part of their work.

6.2.3 Type-centric approaches

Types are a standard element of modern programming languages, consequently they have also been used and reflected concerning the verification of programs. Interesting for our work are especially so called behavioral type systems, which not only subsume data elements but also behavior. In [15] Igarashi and Kobayashi present a framework for the specification of systems of behavioral types for the π-calculus. The basic idea of their approach is to express types as abstract processes. Together with a less expressive calculus that is used for the abstract processes this allows them to verify more complex π-calculus specifications. As the process calculus for the abstract processes does not support an operator for the creation of new channels, the approach does not reach the expressiveness of our approach. The use of behavioral types for service contracts is advocated by Meredith et al. in [60], but without any contributions for their verification.

Liskov and Wing [16] give a definition of types and subtypes that is not purely syntactical but also implies behavioral compatibility. Mainly, they describe a set of constraints that have to be satisfied for a correct subtype relation. However, they define their theory at a very abstract level of programming languages, where methods are abstracted to pre- and post-conditions and invariants for types. Further, they do not provide an automatic proof for the subtype relation.

6.2.4 Discussion

The previous sections have shown that for almost each of our requirements approaches exist, that provide sufficient capabilities to satisfy them. Only the "Analysis under restricted knowledge (A3)" requirement is, if at all, only partially fulfilled. However, an approach that is able to model and verify self-adaptive service-oriented systems, is not described in the literature. A comparison of all presented related work is shown in Table 6.1.

Challenges	*SoaML*	*rigSoaML*	[26]	[27]	[28, 4, 29]	[30]	[33]	[34, 2, 1, 35]	[36]	[38]	[40, 41]
Modeling SOA (M1)	✓	✓	✓	∼	∼	∼	∼	✓	✓	∼	✓
Modeling Dynamics (M2)	✓	✓	○	○	∼	○	○	○	○	∼	∼
Challenges	*SoaML*	*rigSoaML*	[43]	[44]	[45]	[48]	[51]	[46, 14, 47]	[36]	[53]	[54, 55]
Scalable Analysis (A1)	○	✓	∼	○	∼	∼	∼	∼	∼	○	○
Analysis of Reconfiguration (A2)	○	✓	∼	✓	∼	○	○	○	○	○	○
Analysis under restricted knowledge (A3)	○	✓	∼	∼	○	○	∼	○	○	○	○
Challenges	*SoaML*	*rigSoaML*	[56]	[10]	[57]	[58]					
Scalable Analysis (A1)	○	✓	○	∼	✓	✓					
Analysis of Reconfiguration (A2)	○	✓	○	○	∼	∼					
Analysis under restricted knowledge (A3)	○	✓	○	○	○	○					
Challenges	*SoaML*	*rigSoaML*	[59]	[18]	[17]						
Modeling Evolution (E1)	○	✓	○	∼	∼						
Analyzing Evolution (E2)	○	✓	✓	∼	○						

Table 6.1: Coverage of the challenges for *SoaML*, *rigSoaML*, and the related work. Please note that the strong simplification in the table (✓,○, and ∼) makes approaches look similar that are very different. For more detailed description please refer to the approaches descriptions.

Chapter 7

Conclusion

In this report we have presented a combination of a modeling together with a verification technique, that allows to tackle some problems that arise in the context of self-adaptive and service-oriented systems. The origin for these problems is the potentially open nature of those systems that confronts the developer with a situation, where only little knowledge of the current system is available. To overcome these problems we presented a modular modeling technique that builds atop of *SoaML* and thus collaborations and components as first class citizens. The approach we have presented in this report allows to model no service-contracts or participant based on a relatively small amount of knowledge concerning the system's other consituents. It is only required that the constituents that are reachable, when traversing the inheritance relations to the root, are known. This allows to easily introduce new components and service-contracts into the system. Concerning the verification our approach makes heavy use of behavior refinement. The strict use of refinement eases the verification of new consituents in two points. First, for a newly developed consituent only those properties have to be verified, that are specific for this new constituents. All properties that are inherited and have been verified before, remain valid if the consitutent follows the guidelines for refinement. Second, refinement is used to show substitutability, which is required to prove correctness of complex service landscapes.

The presented approach scales very well in the size of system. For an increasing number of instances of a verified system type, no additional verification is necessary. For a changed system type the required modeling and verification effort is small, as depicted above.

7.1 Future Work

At the current state our approach only supports the addition of new types into the system. The removal of types is not covered. In the future we plan to overcome this limiation which bears some difficulties. A type can only be removed after all of it's instances have been removed. The detection that there are no remaining instances is difficult for a highly distributed and de-centralized scenario, that we are investigating. Therefore, we plan to introduce a lease mechanism. Components and service-contract instances have to lease their type. Once the lease period is finished the lease has to be renewed. For a type that is to be deleted the renewal of the lease will not be granted. Removal of types that are no leaves in the inheritance tree is difficult to handle, too.

It is to be noted that also weaker conditions for the correctness of a system at the instance level are possible where besides the types and rules the behavior constraints only visible at the instance level are taken into account. We can, for example, establish guarantees or refinement using state-space based

techniques in case of finite state models of a collaboration or component, if we can establish in addition to the pseudo-type separation guarantees that the collaborations and components are separated at the instance-level. A separation at the instance level would mean that by construction the behavior related to different collaboration or component instances can never directly effect each other. The rules applied for one instance can never effect the elements employed by another collaboration or component. What is still possible also in case of instance-level separation would be that the behavior related to two roles within a single component instance effects each other. However, this is covered by the compositional verification scheme.

A further direction of research is to use testing to check the properties. Of course, testing can not provide the same comprehensive support as formal verification can do, but testing is an alternative for situation, where formal verification can not be applied. Testing can be applied at two different stages of development. First, one could use classical testing approach during the design stage. These approach would have to ensure the same properties, we checked using formal verification, i. e. safety properties and refinement checks. The second approach is to use run-time testing. Using run-time testing one would test the compatibility of roles and components each time, a link is newly established.

From the practical perspective we also have to consider the task of going from a purely theoretical model to a running system. Although, we can use the capabilities of the Story-Diagram-Interpreter to directly execute the components' behavioral rules, we still have to cope with the decentralized nature of the developed systems. Thus, not all entities that appear in a Story-Pattern rule are physically located at the machine, that executes the rule.

Bibliography

[1] Jim Amsden, Pete Rivett, Kolk Henk, Fred Cummins, Jishnu Mukerji, Antoine Lonjon, Cory Casanave, and Irv Badr. *UML Profile and Metamodel for Services*, June 2007. http://www.omg.org/docs/ad/07-06-03.pdf.

[2] Manfred Broy, Ingolf H. Krüger, and Michael Meisinger. A formal model of services. *ACM Trans. Softw. Eng. Methodol.*, 16(1):5, 2007.

[3] Lai Xu. A multi-party contract model. *ACM SIGecom Exchange*, 5(1):13–23, July 2004.

[4] Marco Lohmann, Stefan Sauer, and Gregor Engels. Executable Visual Contracts. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), 21-24 September 2005, Dallas, TX, USA*, pages 63–70. IEEE Computer Society, 2005.

[5] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 261–272, New York, NY, USA, 2008. ACM.

[6] Ingolf Krüger and Vina Ermagan. A UML2 Profile for Service Modeling. In Gregor Engels, Bill Opdyke, Douglas Schmidt, and Frank Weil, editors, *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 4735 of *Lecture Notes in Computer Science*, Nashville, TN, USA, 2007. Springer Berlin / Heidelberg.

[7] Chris Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, 2003.

[8] Web Services Business Process Execution Language Version 2.0. OASIS Standard, 2007.

[9] Gero Decker, Oliver Kopp, Frank Leymann, Kerstin Pfitzner, and Mathias Weske. Modeling Service Choreographies Using BPMN and BPEL4Chor. In *Proceedings of the 20th international conference on Advanced Information Systems Engineering*, pages 79–93, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] Jian Zhang, Heather J. Goldsby, and Betty H.C. Cheng. Modular verification of dynamically adaptive systems. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 161–172, New York, NY, USA, 2009. ACM.

[11] Tian Tan, Yang Liu, Jun Sun, and Jing Dong. Verification of Orchestration Systems Using Compositional Partial Order Reduction. In Shengchao Qin and Zongyan Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 98–114. Springer Berlin / Heidelberg, 2011.

[12] Jing Dong, Yang Liu, Jun Sun, and Xian Zhang. Verification of Computation Orchestration Via Timed Automata. In Zhiming Liu and Jifeng He, editors, *Formal Methods and Software Engineering*, volume 4260 of *Lecture Notes in Computer Science*, pages 226–245. Springer Berlin / Heidelberg, 2006.

[13] Alessio Lomuscio, Hongyang Qu, Marek Sergot, and Monika Solanki. Verifying Temporal and Epistemic Properties of Web Service Compositions. In Bernd Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Proceedings of Service-Oriented Computing – ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 456–461. Springer Berlin / Heidelberg, 2007.

[14] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Compatibility Verification for Web Service Choreography. In *Web Services, IEEE International Conference on*, pages 738–741, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[15] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the Pi-calculus. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 128–141, New York, NY, USA, 2001. ACM.

[16] Barbara Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, pages 1811–1841, November 1994.

[17] Mads Dam and Lars-åke Fredlund. On the Verification of Open Distributed Systems. In *Proceedings of the 1998 ACM Symposium on Applied Computing (SAC '98)*, pages 532–540, New York, NY, USA, 1998. ACM.

[18] Sara Gradara, Antonella Santone, Gigliola Vaglini, and Maria Luisa Villani. Modular formal verification of specifications of concurrent systems. *Software Testing, Verification and Reliability*, 18(1):5–28, 2008.

[19] David G. Messerschmitt and Clemens Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. The MIT Press, 2005.

[20] Linda Northrop, Peter H. Feiler, Richard P. Gabriel, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, and Douglas Schmidt. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006.

[21] Holger Giese, Matthias Tichy, Sven Burmester, Wilhelm Schäfer, and Stephan Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. ESEC/FSE*, pages 38–47. ACM, September 2003.

[22] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *Proc. ICSE*. ACM, 2006.

[23] Basil Becker and Holger Giese. On Safe Service-Oriented Real-Time Coordination for Autonomous Vehicles. In *Proc. ISORC*. IEEE Computer Society Press, 2008.

[24] Holger Giese. Modeling and verification of cooperative self-adaptive mechatronic systems. In *Reliable Systems on Unreliable Networked Platforms*, volume 4322 of *LNCS*. Springer Verlag, 2007.

[25] Arne J. Berre. *Service oriented architecture Modeling Language (SoaML) - Specification for the UML Profile and Metamodel for Services (UPMS)*, November 2008. http://www.omg.org/docs/ad/2008-11-01.pdf.

[26] Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working With Objects - The OOram Software Engineering Method*. Manning Publications Co., Greenwhich, CT 06830, UK, 1996.

[27] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proc. OOPSLA/ECOOP '90*, pages 169–180, New York, NY, USA, 1990. ACM.

[28] Gregor Engels, Marco Lohmann, Stefan Sauer, and Reiko Heckel. Model-Driven Monitoring: An Application of Graph Transformation for Design by Contract. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, volume 4178 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2006.

[29] Marco Lohmann, Gregor Engels, and Stefan Sauer. Model-driven Monitoring: Generating Assertions from Visual Contracts. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pages 355–356. IEEE Computer Society, 2006.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[31] Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering*, 30(3):193–206, March 2004.

[32] Anthony Lauder and Stuart Kent. Precise Visual Specification of Design Patterns. In *ECOOP'98 - Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 114–134. Springer Berlin / Heidelberg, 1998.

[33] Soon-Kyeong Kim and David Carrington. A formalism to describe design patterns based on role concepts. *Formal Aspects of Computing*, 21(5):397–420, October 2009.

[34] Richard Torbjorn Sanders, Humberto Nicolas Castejon, Frank Alexander Kraemer, and Rolv Braek. Using UML 2.0 collaborations for compositional service specification. In *Proc. MoDELS*, volume 3713 of *LNCS*, pages 460–475. Springer Berlin / Heidelberg, 2005.

[35] *UML Profile and Metamodel for Services - for Heterogeneous Architectures (UPMS-HA)*, June 2007. http://www.omg.org/cgi-bin/doc?ad/2007-06-02.

[36] Luciano Baresi, Reiko Heckel, Sebastian Thöne, and Daniel Varró. Modeling and validation of service-oriented architectures: Application vs. style. In *Proc. ESEC/FSE*, pages 68–77. ACM, 2003.

[37] Luciano Baresi, Reiko Heckel, Sebastian Thöne, and Daniel Varró. Style-based modeling and refinement of service-oriented architectures. *Software and Systems Modeling*, 5(2):187–207, 2006.

[38] Matthias Hölzl and Martin Wirsing. Towards a System Model for Ensembles. In Gul Agha, Olivier Danvy, and José Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 241–261. Springer Berlin / Heidelberg, 2011.

[39] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[40] Shuiguang Deng, Zhaohui Wu, Mengchu Zhou, Ying Li, and Jian Wu. Modeling Service Compatibility with Pi-calculus for Choreography. In David Embley, Antoni Olivé, and Sudha Ram, editors, *Conceptual Modeling - ER 2006*, volume 4215 of *Lecture Notes in Computer Science*, pages 26–39. Springer Berlin / Heidelberg, 2006.

[41] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 70(1):96–118, 2007.

[42] Mads Dam. On the decidability of process equivalences for the pi-calculus. *Theoretical Computer Science*, 183(2):215–228, 1997.

[43] Ping Yang, C.R. Ramakrishnan, and Scott A. Smolka. A Logical Encoding of the pi-Calculus: Model Checking Mobile Processes Using Tabled Resolution. *International Journal on Software Tools for Technology Transfer (STTT)*, 2004.

[44] Bjorn Victor and Faron Moller. The Mobility Workbench — A Tool for the pi-Calculus. In David Dill, editor, *Computer Aided Verification (Proc. of CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer Berlin / Heidelberg, 1994.

[45] Carlo Ghezzi, Joel Greenyer, and Valerio Panzica La Manna. Synthesizing Dynamically Updating Controllers from Changes in Scenario-based Specifications. In *Proceeding of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2012) (to appear)*, pages 145 – 154. IEEE Computer Society, 2012.

[46] Howard Foster, Wolfgang Emmerich, Jeff Kramer, Jeff Magee, David S. Rosenblum, and Sebastián Uchitel. Model checking service compositions under resource constraints. In *ESEC/SIGSOFT FSE*, pages 225–234. ACM, 2007.

[47] Howard Foster, Sebasti'an Uchitel, Jeff Magee, and Jeff Kramer. Model-based verification of Web service compositions. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 152–161, October 2003.

[48] Junqing Chen and Linpeng Huang. Formal verification of service composition in pervasive computing environments. In *Internetware '09: Proceedings of the First Asia-Pacific Symposium on Internetware*, pages 1–5, New York, NY, USA, 2009. ACM.

[49] YanPing Yang, QingPing Tan, and Yong Xiao. Model Transformation Based Verification of Web Services Composition. In *Grid and Cooperative Computing - GCC 2005*, volume 3795 of *Lecture Notes in Computer Science*, pages 71–76. Springer Berlin / Heidelberg, 2005.

[50] Yong-shang Cheng, Zhikui Wang, and Xiao-feng Zhou. Research on Web Service Composition and Verification. *Semantics, Knowledge and Grid, International Conference on*, pages 467–470, 2007.

[51] Mauro Caporuscio, Paola Inverardi, and Patrizio Pelliccione. Compositional Verification of Middleware-Based Software Architecture Descriptions. In *Proc. ICSE*, pages 221–230, 2004.

[52] Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Software and System Modeling*, 3(2):85–113, 2004.

[53] Arend Rensink. Towards model checking graph grammars. In *Proc. AVoCS*, pages 150–160. University of Southampton, 2003.

[54] Marcelo F. Frias, Juan P. Galeotti, Carlos Lopez Pombo, and Nazareno Aguirre. DynAlloy: Upgrading Alloy with actions. In *Proc. ICSE*. ACM, 2005.

[55] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[56] P.C. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In *Proc. FASE*, LNCS 2984, pages 354–358. Springer, 2004.

[57] Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis technique for graph transformation systems. In *Proc. CONCUR*, volume 2154 of *LNCS*, pages 381–395. Springer, 2001.

[58] Jörg Bauer and Reinhard Wilhelm. Static Analysis of Dynamic Communication Systems by Partner Abstraction. In *Proc. of the 14th International Symposium, SAS*, volume 4634 of *LNCS*, pages 249–264. Springer Berlin / Heidelberg, 2007.

[59] Dirk Niebuhr and Andreas Rausch. Guaranteeing Correctness of Component Bindings in Dynamic Adaptive Systems based on Runtime Testing. In *Proceedings of the 4th Workshop on Services Integration in Pervasive Environments (SIPE 09) at the International Conference on Pervasive Services 2009 (ICSP 2009)*, pages 7–12, 2009.

[60] L. G. Meredith and Steve Bjorg. Contracts and types. *Communications of the ACM*, 46(10):41–47, October 2003.

[61] Hartmut Ehrig, Annegret Habel, Leen Lambers, Fernando Orejas, and Ulrike Golas. Local Confluence for Rules with Nested Application Conditions. In *Proceedings of Intern. Conf. on Graph Transformation (ICGT' 10)*, volume 6372 of *LNCS*, pages 330–345. Springer, 2010.

[62] Claudia Ermel, Jürgen Gall, Leen Lambers, and Gabriele Taentzer. Modeling with Plausibility Checking: Inspecting Favorable and Critical Signs for Consistency between Control Flow and Functional Behavior. Technical Report 2011/2, TU Berlin, 2011.

[63] Rajeev Alur, Costas Courcoubetis, N. Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, X. Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, February 1995.

Appendix A

Complete Case Study

A.1 abstract contract collaboration interface

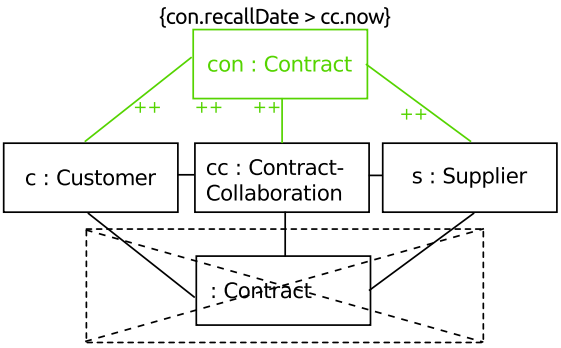


Figure A.1: Abstract contract collaboration: create contract rule (see also Figure 4.3)

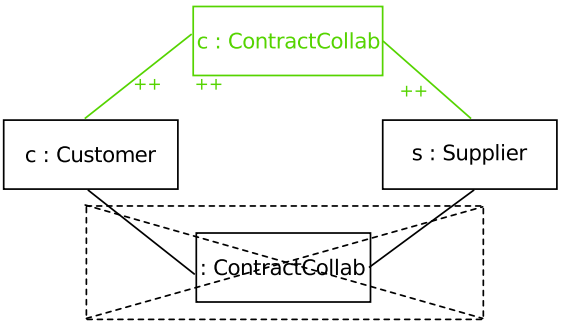


Figure A.2: Abstract contract collaboration: create collaboration rule

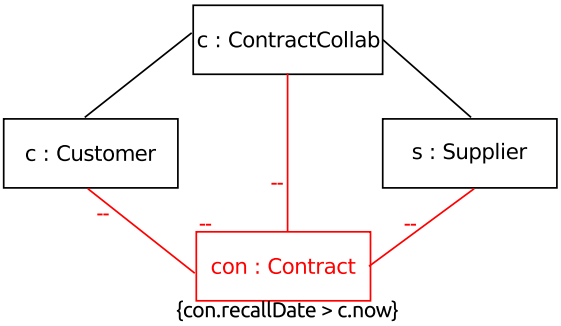


Figure A.3: Abstract contract collaboration: delete contract urgent rule

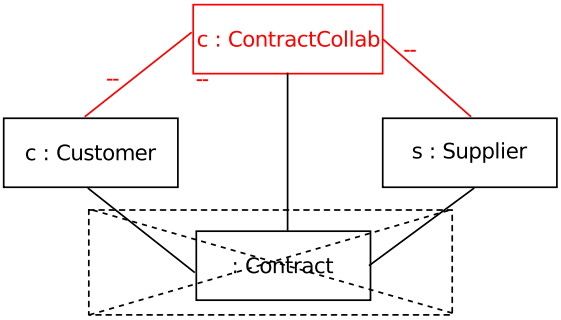


Figure A.4: Abstract contract collaboration: destroy collab rule

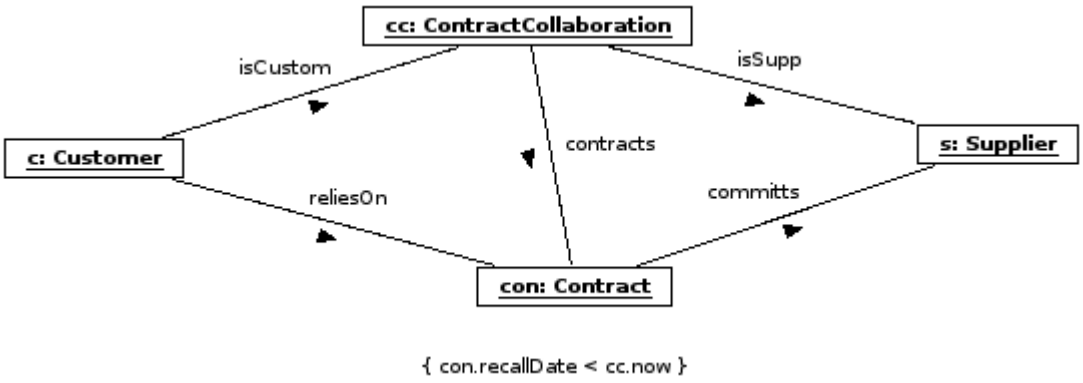


Figure A.5: Abstract contract collaboration: unrecalled contract forbidden

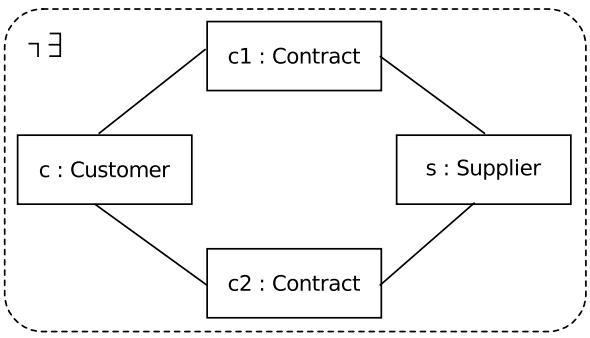


Figure A.6: Abstract contract collaboration: two contracts forbidden (see also Figure 4.2)

A.2 abstract factory interface

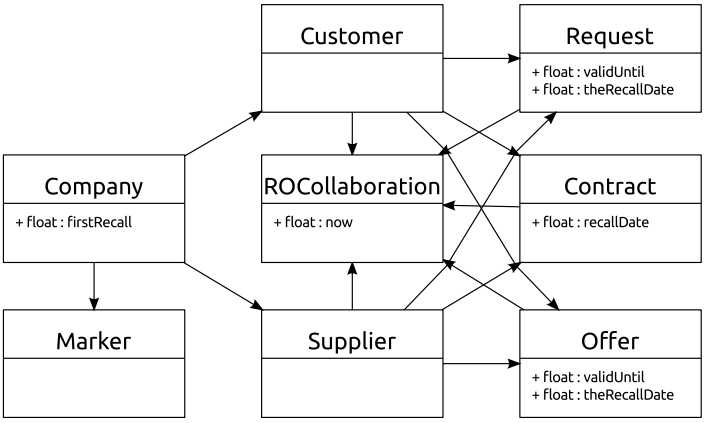


Figure A.7: Abstract factory interface: Class diagram (see also Figure 4.9)

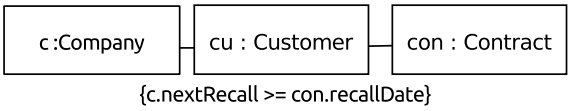


Figure A.8: Abstract factory interface: customer recall guarantee

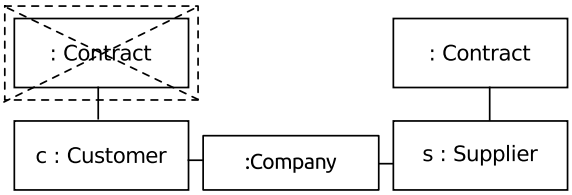


Figure A.9: Abstract factory interface: The property the system has to fulfill (see also Figure 4.11(b))

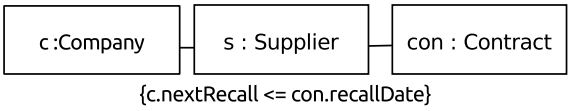


Figure A.10: Abstract factory interface: supplier recall guarantee

A.3 Factory implementation

The factory implementation is still incomplete. To make it work the contract update has to be divided into two steps. 1) propose update 2) accept and perform update

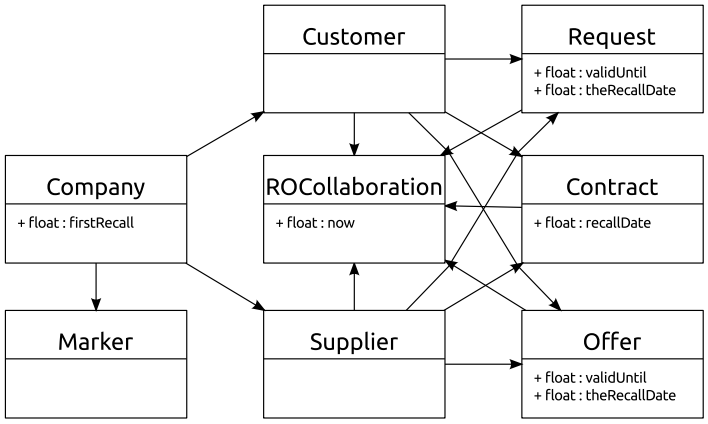


Figure A.11: ClassDiagram for the Factory implementation (see also Figure 4.9)

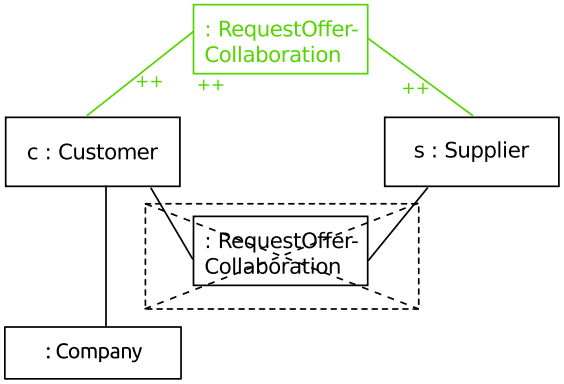


Figure A.12: Factory implementation: create ROCollab

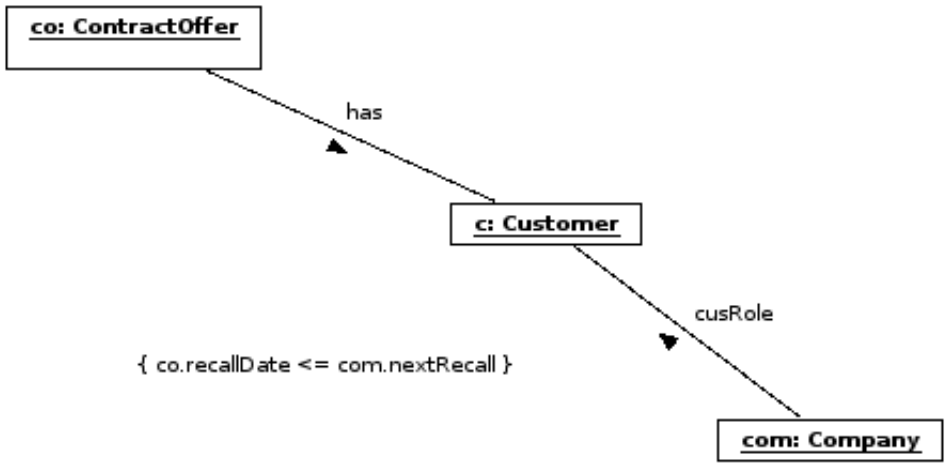


Figure A.13: Factory implementation: constrain contract offer - forbidden

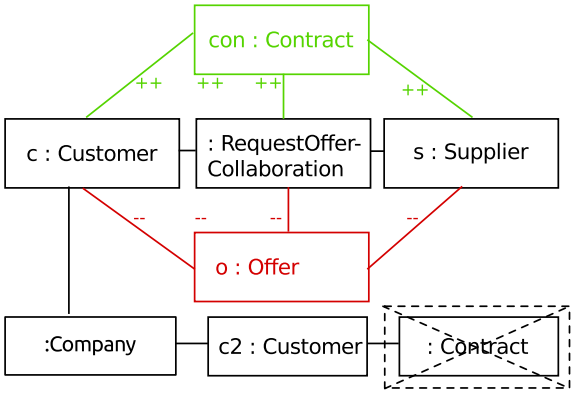


Figure A.14: Factory implementation: customer create contract (see also Figure 4.10(c))

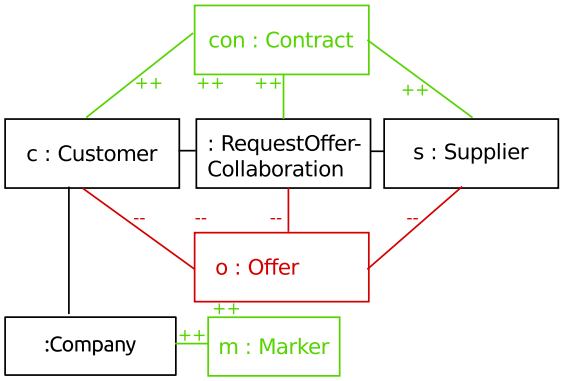


Figure A.15: Factory implementation: Customer create last Contract. This rule preempts the other createContract rule.

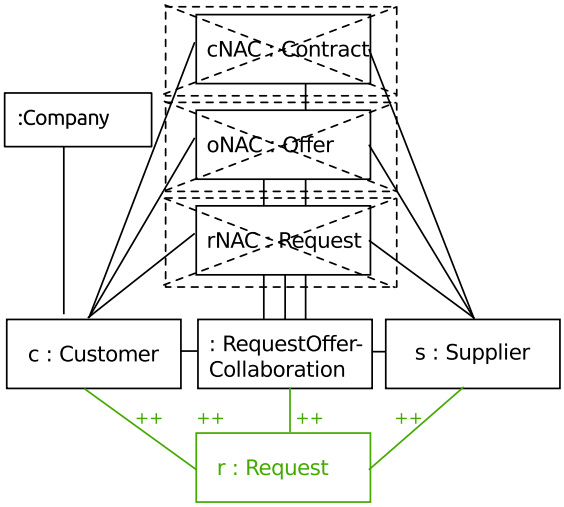


Figure A.16: Factory implementation: customer create REQUEST (see also Figure 4.10(a))

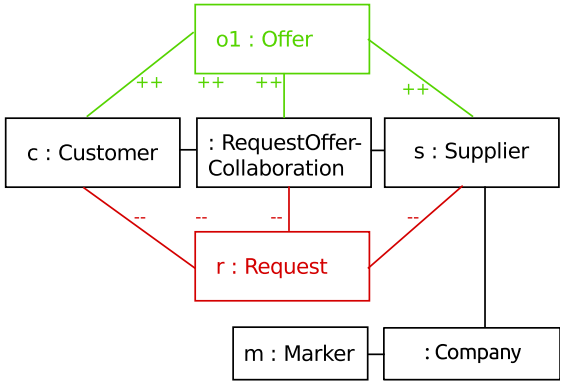


Figure A.17: Factory implementation: SUPPLIER create OFFER (see also Figure 4.10(b))

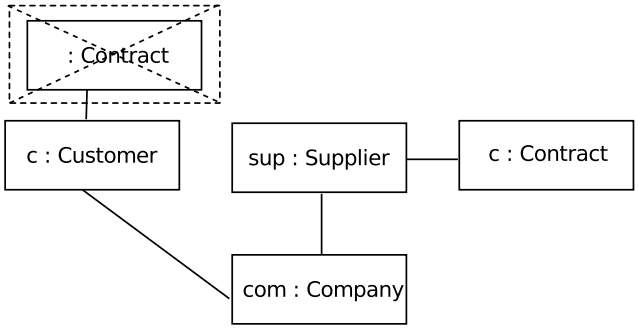


Figure A.18: Factory implementation: guaranteed property

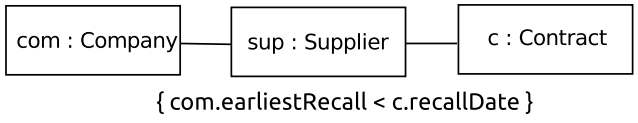


Figure A.19: Factory implementation: guaranteed property supEarlyRecall

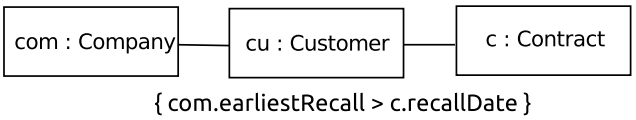


Figure A.20: Factory implementation: guaranteed property CustLateRecall

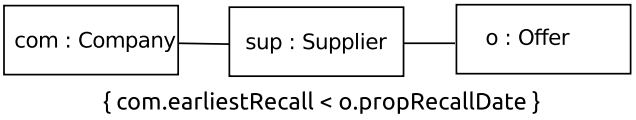


Figure A.21: Factory implementation: guaranteed property supEarlyPropRecall

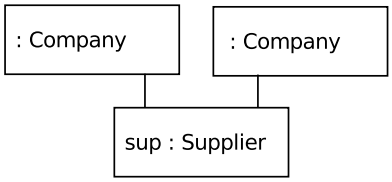


Figure A.22: Factory implementation: guaranteed property sup2Comp

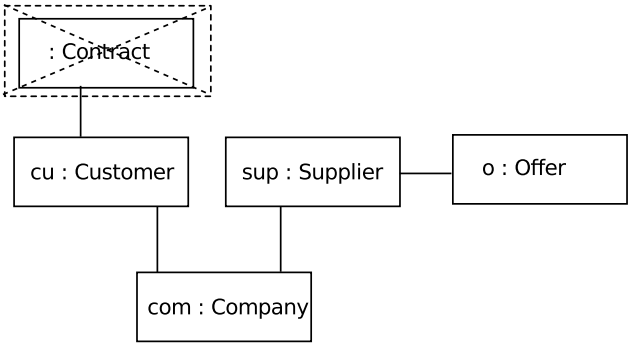


Figure A.23: Factory implementation: guaranteed property OfferButNoCustCon

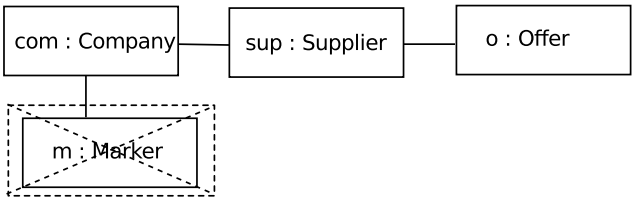


Figure A.24: Factory implementation: guaranteed property OfferButNoMarker

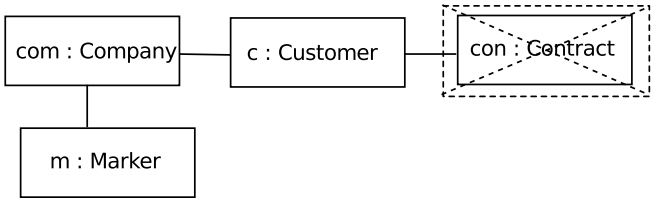


Figure A.25: Factory implementation: guaranteed property MarkerButNoCon

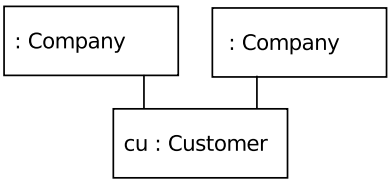


Figure A.26: Factory implementation: guaranteed property Cust2Comp

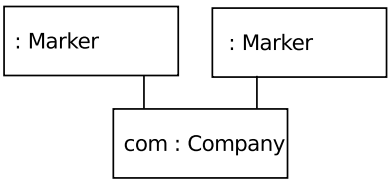


Figure A.27: Factory implementation: guaranteed property Comp2Marker

A.4 Request offer contract collaboration

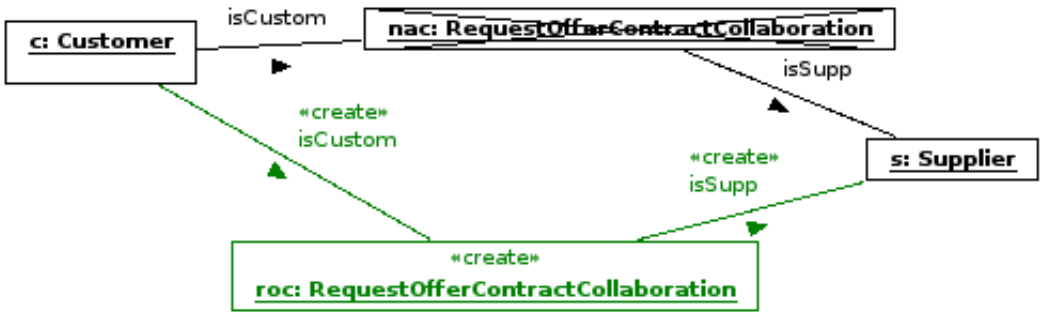


Figure A.28: Request offer contract collaboration: create collaboration rule

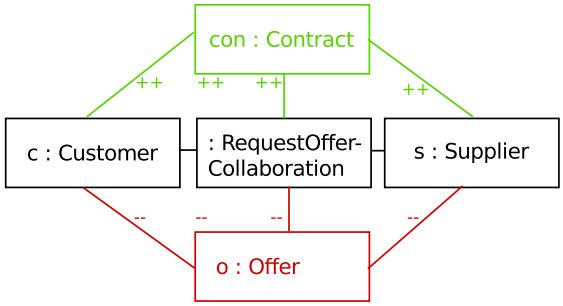


Figure A.29: Request offer contract collaboration: create contract rule

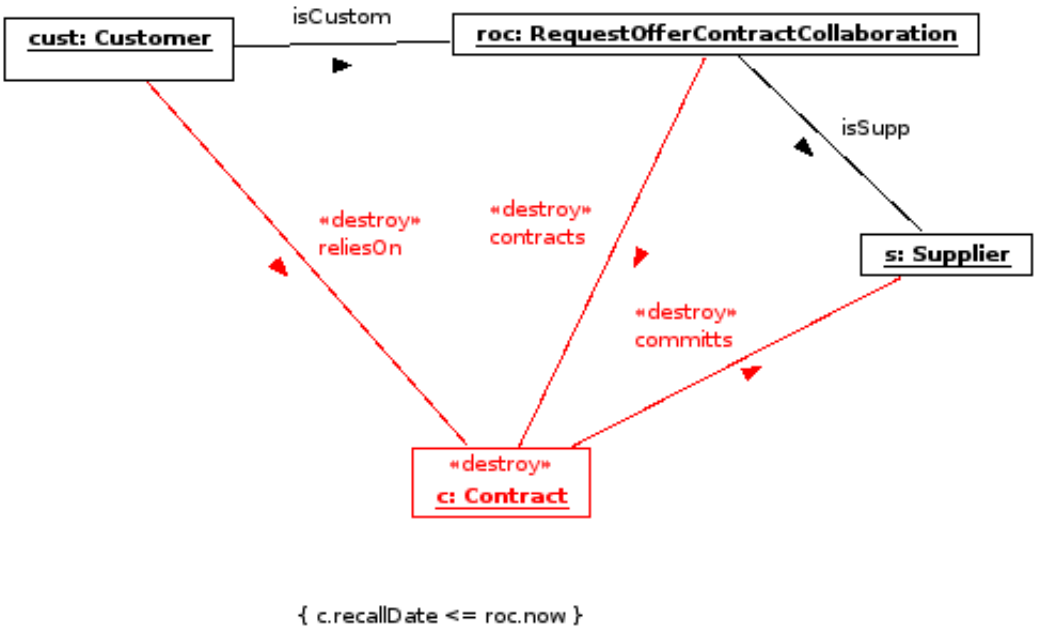


Figure A.30: Request offer contract collaboration: delete contract urgent rule

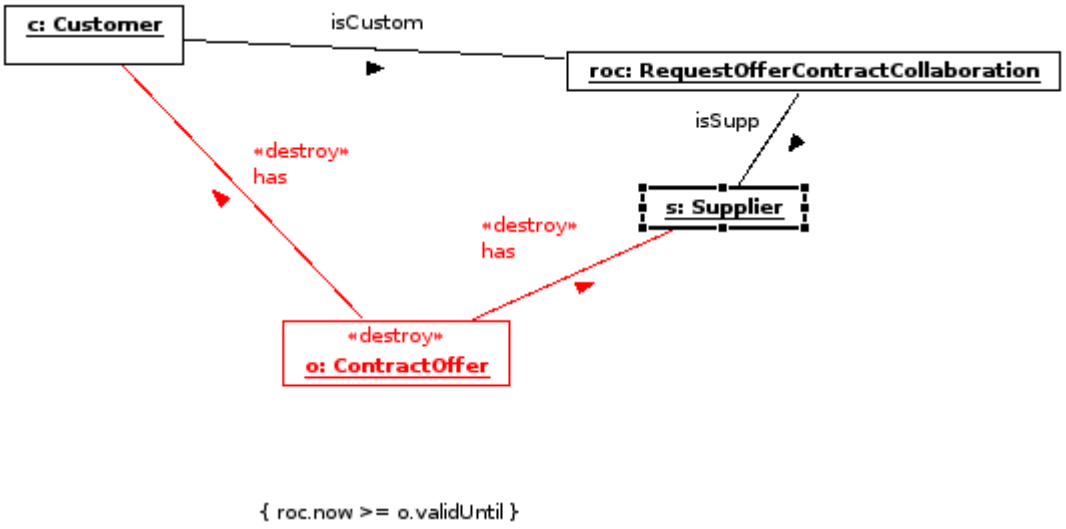


Figure A.31: Request offer contract collaboration: delete invalid offer urgent rule

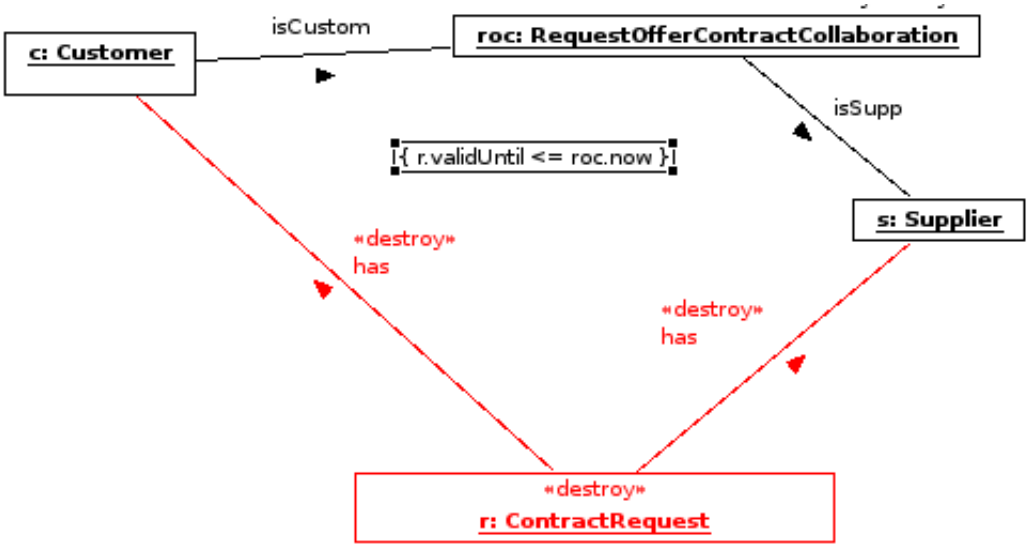


Figure A.32: Request offer contract collaboration: delete invalid request urgent rule

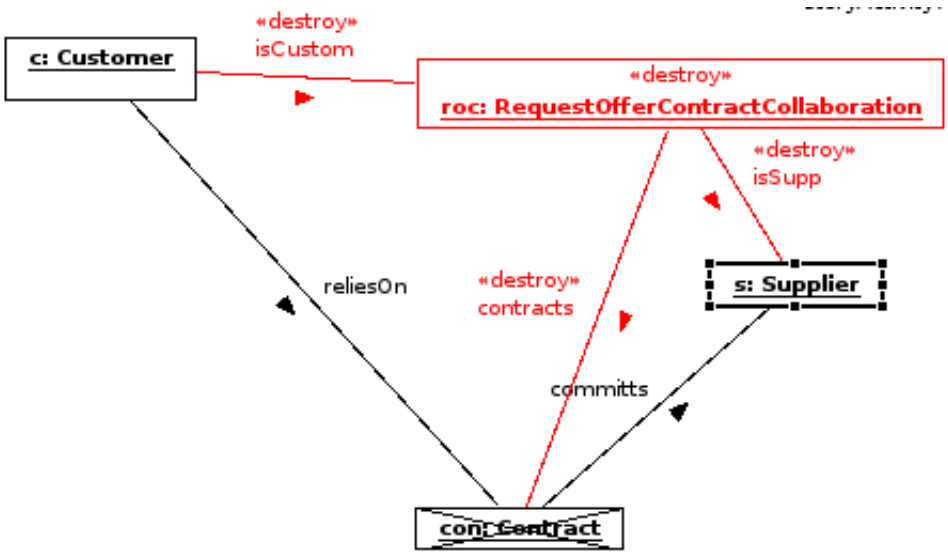


Figure A.33: Request offer contract collaboration: destroy collaboration rule

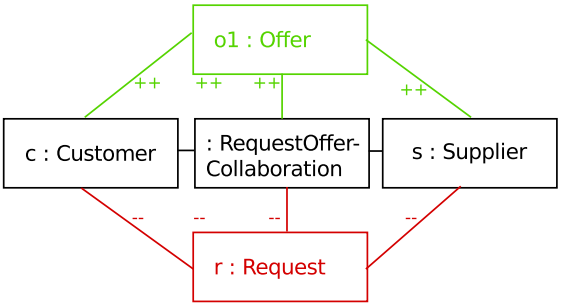


Figure A.34: Request offer contract collaboration: send offer rule

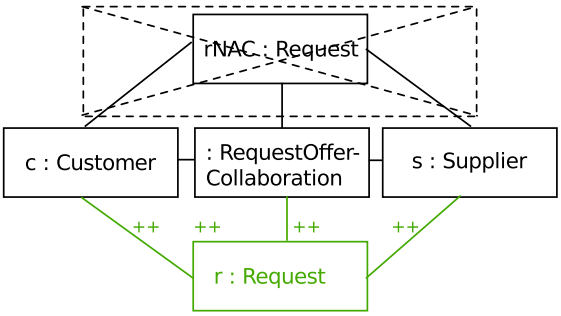


Figure A.35: Request offer contract collaboration: send request rule

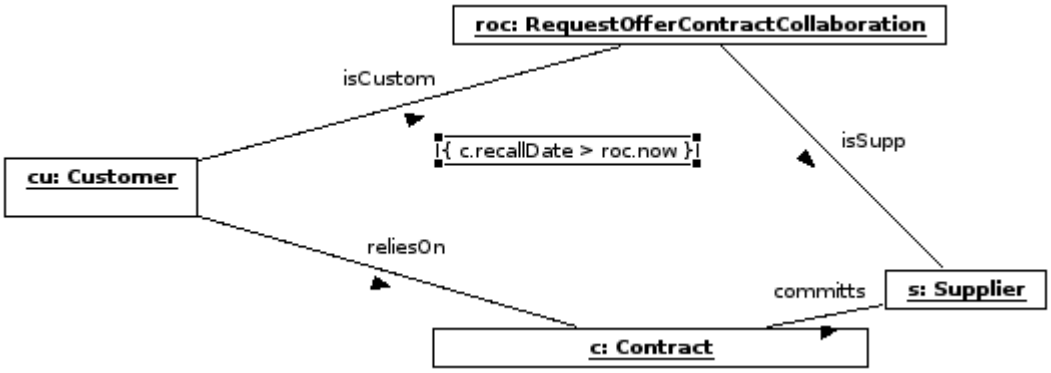


Figure A.36: Request offer contract collaboration: unrecalled contract, forbidden

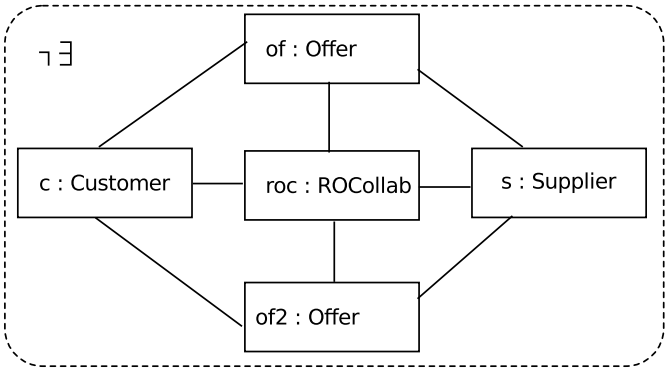


Figure A.37: Request Offer contract collaboration: guaranteed property notTwoOffers

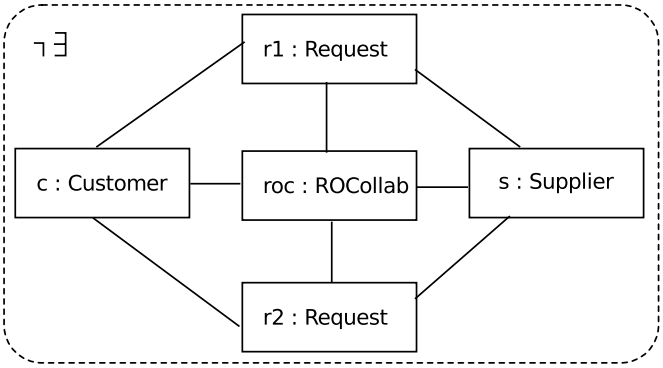


Figure A.38: Request Offer contract collaboration: guaranteed property notTwoRequest

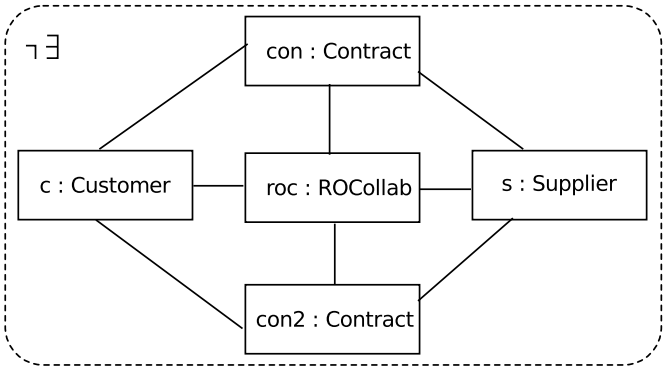


Figure A.39: Request Offer contract collaboration: guaranteed property notTwoContracts

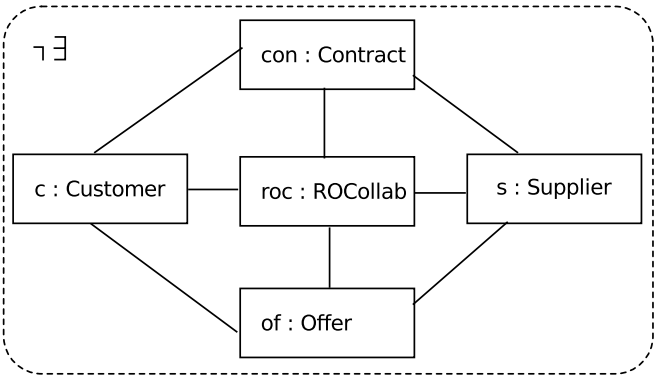


Figure A.40: Request Offer contract collaboration: guaranteed property notOfferandContract

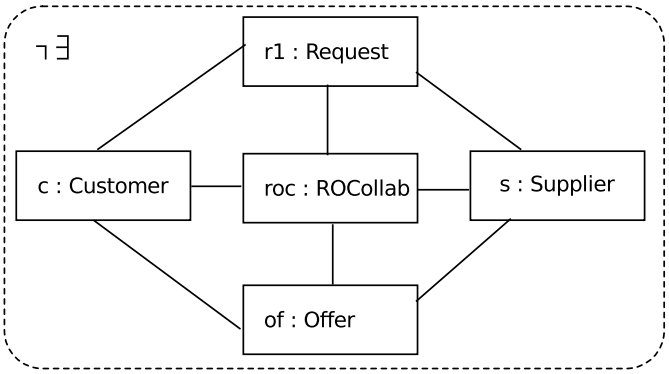


Figure A.41: Request Offer contract collaboration: guaranteed property noOfandRequest

Appendix B

Formal Foundations

B.1 Graphs

Definition B.1 (Type-graph)
A type-graph T is a labeled and directed graph, given as a tupel $T = (V, E, l_V, l_E, s, t)$ where V is a set of vertexes,$E \subseteq V \times V$ is a set of directed edges, $l_V : V \mapsto \mathcal{A}$ is a vertex labeling function, $l_E : E \mapsto \mathcal{A}$ is an edge labeling function, $s, t : E \mapsto V$ returns for each edge it's source and target vertex, respectively. $\mathcal{A}$ is a globally defined alphabet holding all possible types for nodes and edges.

A type-graph T is said to be well-defined if the types assigned to the vertexes and edges are pair-wise disjoint.

Definition B.2 (Typed Graph)
A typed graph G is given as a tupel $G = (V, E, l_V, l_E, s, t, T)$ where V is a set of vertexes, $E \subseteq V \times V$ is a set of directed edges, $l_V : V \mapsto V_T$ assigns each vertex in V a type in the type-graph T,$L_E : E \mapsto V_E$ assigns each edge in E a type in the type-graph T,$s, t : E \mapsto V$ returns for each edge it's source and target vertex, respectively, and T is a type-graph.

A graph G conforms to it's type-graph T iff:

$$\forall e : \quad e \in E \quad \Longrightarrow \quad l_V(s(e)) = s(l_E(e)) \wedge l_V(t(e)) = t(l_E(e))$$

A graph G' is called a subgraph of G iff, $V' \subseteq V$ and $E' \subseteq E$ and is denoted as $G' \leq G$. We write $G' < G$ ff $V' \subset V$ or $E' \subset E$.

The set of all graphs is denoted as $\mathcal{G}$

In some cases it is useful to restrict a graph G, typed over type-graph T, to a set of nodes and edges. This can be done by defining a type-graph $T' \leq T$ and removing all elements from G that are not typed over elements contained in T'. We write this as $G' = G_{|T'}$. Where G' is given as follows: $V' = \{v | v \in V \wedge l_V(v) \in V_{T'}\}$ and $E' = \{e | e \in E \wedge l_E(e) \in E_{T'}\}$

Definition B.3 (Graph morphism)
A graph morphism $m = (m_v, m_e)$ is a structure and type preserving mapping between two graphs

$G, H \in \mathcal{G}$ with $m_v : V_G \mapsto V_H$ and $m_e : E_G \mapsto E_H$ such that:

$$\begin{aligned} \forall(v,v') : (v,v') \in m_v \quad &\rightarrow \quad l_v(v) = l_v(v') \\ \forall(e,e') : (e,e') \in m_e \quad &\rightarrow \quad l_e(e) = l_e(e') \wedge (s(e), s(e')) \in m_v \\ &\qquad \wedge (t(e), t(e')) \in m_v \end{aligned}$$

A morphism between G and H is denoted as $G \xrightarrow{m} H$ or $G \to H$ for short. If the functions m_v, m_e are injective functions we say that m is an injective morphism and denote this as $G \hookrightarrow H$ or $G \overset{m}{\hookrightarrow} H$ if we want to explicitly name the morphism. If the morphism m consists of two bijective functions, we say that G and H are isomorphic to each other. We denote two isomorphic graphs G and H as $G \approx H$, if we further want to stress the isomorphism we make it explicitly as $G \approx_m H$. m is then called an isomorphism.

A subgraph isomorphism between two graph $G, H \in \mathcal{G}$ exists if there is a subgraph $G' \leq G$ and a isomorphism m, such that $G' \approx_m H$. A subgraph isomorphism is denoted as $G \lesssim_m H$

Definition B.4 (Graph constraint)
A graph constraint C is given as $C = (\exists P, \bigwedge_{i \in I} N_i)$ where P and each N_i for $i \in I$ is a graph with $P < N_i$ and $P \xrightarrow{n_i} N_i$. A graph G satisfies a constraint C iff

$$\exists q : P \xrightarrow{q} G$$
$$\nexists q_i' : N_i \xrightarrow{q_i'} G \text{ such that } q_i' \circ n_i = q \; \forall i \in I$$

We shall denote $G \models C$ if a graph G satisfies a constraint C.

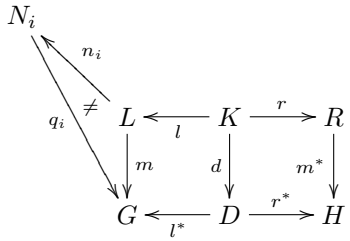
The above definition of graph constraints conforms to the widely used definition that is given in [61] for application conditions with the difference that our definition does not allow for nesting of graph constraints.

B.2 Graph Transformations

Definition B.5 (Graph Transformation Rule)
A graph transformation rule r is defined as $r = (L, R, K, l, r, A^-)$ where: $L, R, K \in \mathcal{G}$ are three graphs with $V_L \cap V_R = V_L \cap V_K = V_K \cap V_R = \emptyset$, that are typed over the same type graph, $l = (l_v, l_e), r = (r_v, r_e)$ are total and injective graph isomorphism, with $L \xleftarrow{l} K$ and $K \xrightarrow{r} R$ and A^- is a set of graphs encoding negative application conditions (NACs) with $N_i \in A^-$, $L \overset{n_i}{\hookrightarrow} N_i$ and $L \leq N_i$. L and R are called left- and right-hand-side, respectively, and K is called interface graph. We also denote a graph transformation rule as $L \leftarrow K \rightarrow R$. The rule deletes all elements that are in L but not in $ran(l)$ and creates elements that are in R but not in $ran(r)$.

Definition B.6 (Applicability of Graph Rules)
A graph rule $L \xleftarrow{l} K \xrightarrow{r} R$ is applicable to a graph G if we can find a match $L \xrightarrow{m} G$ which satisfies the dangling condition: all edges e adjacent to the image $m_v(v)$ of a deleted node v are also part of the image of m, $e \in \mathrm{ran}(m)$. The identification condition meaning $\forall e_1, e_2 : e_1, e_2 \in L \wedge e_1 \neq e_2 \rightarrow$

$m(e_1) \neq m(e_2)$, *i. e.* m *has to be injective.*



And for each $N_i \in A^-$ *we must not find an injective morphism* q_i *such that* $n_i \circ q_i = m$*. The applicability of a graph transformation rule* r *can be encoded as a graph constraint* $Appl(r)$ *such that the rule* r *is applicable in graph* G *iff* $G \models Appl(r)$[1].

Definition B.7 (Graph Transformation Systems)
A graph transformation system (GTS) S *is given as a tuple* $S = (G_0, R, p, T)$ *where* $G_0 \in \mathcal{G}$ *is the initial graph,* R *is a set of graph transformation rules,* $p : R \mapsto R$ *is a transitive function that indicates preempting rules, and* T *is a type-graph such that* G_0 *and all rules in* R *are typed over* T*.*

Definition B.8 (Semantics of GTS)
The semantics of a GTS S *is given through a labeled transition system (LTS)* $L_S = (S_0, S, \delta)$ *where* S *is a set of states,* $S_0 \in S$ *is the system's initial state and* $\delta : S \times S$ *is the LTS' transition relation. Each state in* S *we assign a graph, such that* G_0 *is assigned to* S_0 *and the pair* $(S, T) \in \delta$ *if there exists a rule* $r \in R$ *and a morphism* m *such that* $G_S \overset{r,m}{\Rightarrow} G_T$ *and it does not exists a rule* r' *and morphism* m' *with* $(r', r) \in p$ *and* $G_S \overset{r',m'}{\Rightarrow} G_{T'}$*. Where* G_S *and* G_T *refer to the graphs that are assigned the states* S *and* T*, respectively.*

B.3 Hybrid Graph Transformations

Hybrid graph transformations differ from the variant that has been introduced above in that they not only specify discrete behavior but also have a continuous part. The continuous part is typically modelled through a set of attributes and laws that describe the change of the attributes' valuations over time. At the level of type-graphs we will specify the attributes that are assigned to any node of a given type. At graph level we assign a valuation to all pairs of nodes and attributes. Graph transformation rules will be enriched with a jump condition, that constrains the attribute valuations for which the graph transformation rule will be enabled and allows us to encode an update of the attributes' valuation.

Before we can define attributed type graphs and attributed graphs we have to introduce $\mathcal{A}$ a global set of attributes and $\dot{\mathcal{A}}$ the first derivation over the time of $\mathcal{A}$. The attribute's derivation is controlled by laws. These laws can change depending on the overall system's behavior. Therefore we will model the laws through special nodes that are called *control modes*. Control modes can be written, i. e. created and deleted, and read, i. e. matched and forbidden, by hybrid graph transformation rules. An alternative the introduction of control modes, would have been the use of a special mode attribute but then the laws had to be also dependent of the mode attribute.

Definition B.9 (Attributed Type Graphs)
An attributed type graph TG_A *is given as a tuple* $TG_A = (V, E, l_v, l_e, s, t, Attr, CM)$ *where* V, E, l_v, l_e, s, t *are defined as in Definition B.1 and* $Attr : \mathcal{A} \mapsto V$ *is a partial function that assigns the global set of attributes to the nodes of the type graph and* $CM \subset V$ *is a special set of nodes*

[1] The applicability constraint for a graph transformation rule $R : L \leftarrow K \rightarrow R$ is basically given through the rule's left-hand-side L, the rule's negative application conditions and additional NACs that encode the dangling condition (cf. [62]).

called control modes. Each control mode is only allowed to be adjacent to exactly one other node. Further, each control mode $cm \in CM$ has a function $f_{cm} : \mathbb{R}^{\geq 0} \mapsto (A_v \mapsto \mathbb{R})$ where v is the node adjacent to cm and $A_v = \{a | a \in \mathcal{A} \wedge Attr(a) = v\}$.

Obviously, in the above definition the function f_{cm} is a compact representation for a differential equation. For any positive real number $r \in \mathbb{R}^{\geq 0}$ the function has a mapping to an attribute valuation $A_v \mapsto \mathbb{R}$. The above definition of attributed type graphs is somehow related to the definition of hybrid automata by Alur et al. [63]. The difference clearly is that the number of attributes or variables, as [63] calls them, is not fixed but depends on the nodes that are available in the actual graph. The fact that in attributed type graphs a node can have different control modes (also only one at a time) is not an advantage compared to hybrid automata as this is supported through different *activities* (cf. [63]).

Definition B.10 (Attributed Graph)
An attributed graph is a graph that is additionally equipped with a valuation β. The attributed and typed graph $G = (V, E, l_V, l_E, s, t, T, \beta)$ is defined as a graph introduced in Definition B.2 and has a valuation $\beta : \mathcal{A} \times V \mapsto \mathbb{R}$ such that

$$\forall a, v, r : ((a, v), r) \in \beta \wedge a \in Attr \wedge v \in V \wedge r \in \mathbb{R} \implies Attr_T(a) = l_V(v)$$

and

$$\forall a, v : a \in \mathcal{A} \wedge v \in V \wedge Attr_t(a) = l_V(v) \implies \exists r : r \in \mathbb{R} \wedge \beta((a, v)) = r$$

We say an attributed graph is well-formed if each node $v \in V$ with $l_V(v)$ being adjacent to a set of nodes C with $C \subset CM_T$ is adjacent to exactly one node v' with $l_V(v') \in C$.

From an attributed graph G we can derive the attributed graph $G' = G \oplus t$ with $t \in \mathbb{R}$ which differs from G only in the valuation β'. In β' for each node $v \in V$ being adjacent to a control mode c the valuation for the variable subset A_v is replaced by the valuation $f_c(t)$.We use (G, β) as a shorthand notation for an attributed graph if it's single constituents are not important for understanding.

The above definition of attributed graphs gives us graphs that are enriched with a valuation function that assigns each valid pair of nodes and attributes exactly one value. A pair of attributed and nodes is valid if and only if the graph's type-graph connects the node's type with the attribute. Further the valuation has to be defined for all valid pairs of nodes and attributes that occur in the attributed graph. In the further we will use $v.a$ as a shorthand reference to the attribute of type $a \in \mathcal{A}$ that is connected to the node v.

Definition B.11 (Hybrid Graph Transformation Rule)
A hybrid graph transformation rule $P = (L, R, K, l, r, A^-, \phi)$ where the first constituents of the tuple are defined as in Definition B.5 and $\phi : (\mathcal{A} \times (V_L \cup V_R) \mapsto \mathbb{R}) \mapsto \mathbb{B}$ assigns the valuation pairs for the left- and right-hand side of the rule a boolean value.

The application of a hybrid graph transformation rule is defined as follows:

Definition B.12 (Hybrid Graph Transformation Rule Application)
A hybrid graph transformation rule $P = (L, R, K, l, r, A^-, \phi)$ is applicable to the attributed graphs G, H iff the (discrete) graph transformation rule $P' = (L, R, K, l, r, A^-)$ is applicable to $G \overset{P',m}{\Longrightarrow} H$ and the graphs valuations in the image of m and m^* satisfy ϕ

$$\phi(\beta_G^m \cup \beta_H^{m^*}) \equiv true$$

Where β^m denotes the valuations of the attributed graphs G and H, respectively that are translated over the morphism m. The application of a hybrid graph transformation rule is denoted as $G, \beta_G \overset{P,m}{\Longrightarrow} H, \beta_H$.

Given the constructs we have defined above we can now introduce in complete analogy to the discrete scenario hybrid graph transformation systems and define their semantics. A hybrid graph transformation system (HGTS) is defined as a (discrete) GTS, but now the initial graph is an attributed graph, the rules are hybrid graph transformation rules and in addition to the priorities we also introduce the concept of urgent rules. Urgent rules in contrast to non-urgent rules have to be applied, once they are enabled.

Definition B.13 (Hybrid Graph Transformation Systems)
A hybrid graph transformation systems (HGTS) $S = (G_0, R, prio, T, \mathcal{R}_u)$ *is given as an initial attributed graph* G_0*, a set of hybrid graph transformation rules* R*, a priority function* $prio : R \mapsto \mathbb{N}$*, an attributed type graph* T *and a set of urgent rules* $\mathcal{R}_u \subseteq R$*.*

The semantics of an HGTS has to be defined differently compared to a discrete GTS, as we now have to consider the continuous changes of the attributes' valuations over the time.

Definition B.14 (Semantics of HGTS)
The semantics of a hybrid graph transformation systems S *are given through a labeled transition systems* $L = (S_0, S, \delta)$ *where* S *is a set of states, each of them corresponding to a graph* G_S*,* $S_0 \in S$ *is the LTS' initial state and corresponds to* G_0*. The transition relation* $\delta \subseteq S \times R \times S \cup S \times \mathbb{R} \times S$ *is given as* $(S_P, r, S_Q) \in \delta$ *with* $S_T, S_Q \in S$ *and* $r \in R$ *iff* $\exists m$ *such that* $G_T, \beta_T \overset{r,m}{\Longrightarrow} G_Q, \beta_Q$ *and* $(S_P, t, S_Q) \in \delta$ *with* $S_P, S_Q \in S$ *and* $t \in \mathbb{R}$ *iff* $G_Q = G_S \oplus t$ *and if* $\nexists 0 \leq t' < t$ *and* $r_u \in \mathcal{R}_u$ *such that* r_u *is applicable in* $G_s \oplus t'$*.*

Appendix C

Traces & Refinement

In this chapter we will introduce the formal underpinning of our approach. We will use traces and refinement of traces as semantic model and use this model to describe the refinement of rule sets.

C.1 Traces

Definition C.1 (Trace)
A trace t is given as $t = (G_0, S)$ where G_0 is the initial graph and S is a sequence of morphism / graph transformation rule pairs or positive real numbers $r \in \mathbb{R}^{\geq 0}$, such that:

$$G_0 \to G_1 \to \dots G_{i-1} \to G_i \to \dots$$

where each step $\to$ is either a rule application $G_i \xrightarrow{r_i}_{m_i} G_{i+1}$ with r_i a graph-rule and m_i a matching morphism or a continous step with $G_i \xrightarrow{r} G_{i+1}$. A continuous step has to follow the restrictions that are mentioned in Definition B.14. For a given set of rules R we denote the set of all traces starting with initial graph G_0 as $T(G_0, R)$. The set of all traces that can be constructed using the set R from any graph is denoted as $T(R)$. A sub-trace of trace t starting at the i-th position is denoted as t_i for $i \in \mathbb{N}^+$

A trace t is a possible path through a GTS' reachability graph starting at the GTS' initial graph. Following, $T(G_0, R)$ is an equivalent notation for the GTS' reachability graph given through G_0 and R[1]. We can restrict any trace $t \in T(G_0, R)$, where G_0 and the rules in R are typed over type graph T to a rule-set R' typed over $T' \leq T$, denoted as $t|_{T',R'}$ by restricting each graph $G_i \in t$ to $G_{i|T'}$ and changing each step $G_i \xrightarrow{r_i} G_{i+1}$ to $G_{i|T'} \xrightarrow{r_i'} G_{i+1|T'}$ iff $r_i \prec r_i'$ and $G_{i|T'} \xrightarrow{\tau} G_{i+1|T'}$ otherwise. Valuations are dependent on the available control-modes, and each control-mode has the exclusive control over a set of attributes. Following in a restricted graph the valuations becomes restricted, too, but for the attributes and control-modes, that are not removed, the valuation after a continuous step is the same in the restricted graph as in the unrestricted graph. The fact that jump-conditions of refined graph-rules only strengthen the refined graph-rule's jump-condition ensures that corresponding graph-rules in the restricted trace are correctly applied.

Note that in a restricted trace containing a τ-step $G_i \xrightarrow{\tau} G_{i+1}$ it is not necessarily the case that $G_i \approx G_{i+1}$

[1] Under the assumption that all rules have the same priority. If different priorities are present the set of valid traces in the HGTS's reachability graph is smaller.

In a restricted trace it is not unlikely that isomorphic graphs are repeated several times. The *stutter*-operator $\natural$ can be used to remove the repetition of finite sequences of identical states from a trace t. Given a trace t, $\natural t$ is given as the trace where each sequence

$$G_i \to G_{i+1} \to \ldots \to G_j \to G_{j+1}$$

with $G_{i+1} \approx G_k$ for $i+1 < k \leq j$ gets replaced by the sequence

$$G_i \to G_{i+1} \to G_{j+1}.$$

Note that the stutter-operator does not combine sequences of continuous steps into one longer continuous step. If the stutter-operator were defined this way, the traces would lose information and in consequence it could happen that the trace t satisfies a property ϕ but $\natural t$ does not. Let us assume we have a trace $t = \ldots G_{i-1} \xrightarrow{r_1} G_i \xrightarrow{r_2} G_{i+1}$ where the graph G_i is essentially required for $t \models \phi$, then the trace $t' = \ldots G_{i-1} \xrightarrow{r_1+r_2} G_{i+1}$ does not satisfy ϕ as the graph G_i is missing in t'.

C.2 Properties

In order to be able to specify what correct behavior means, we have to be able to express properties and have to define what it means, if a trace satisfies a properties. The properties are given as a word of the language $\mathcal{L}$. The properties language $\mathcal{L}$ is based on the LTL temporal logic without the next operator. Thus, it is possible to use globally, future and holds until operators. AP defines a set of atomic properties, that could be satisfied by system states. The language $\mathcal{L}$ of LTL formula as used in our approach is recursively given as:

$$\begin{aligned} a &\in \mathcal{L} \quad \forall a \in AP \\ \phi \wedge \psi &\in \mathcal{L} \quad \forall \phi, \psi \in \mathcal{L} \\ \neg\phi &\in \mathcal{L} \quad \forall \phi \in \mathcal{L} \\ F\phi &\in \mathcal{L} \quad \forall \phi \in \mathcal{L} \\ G\phi &\in \mathcal{L} \quad \forall \phi \in \mathcal{L} \\ \phi\, U\, \psi &\in \mathcal{L} \quad \forall \phi, \psi \in \mathcal{L} \end{aligned}$$

Given two atomic properties $\phi, \psi \in AP$ we assume that they both can be evaluated independent of each other and we do not have any interference between them (i. e. no binding of elements to variables is allowed).

For the case of traces that stem from graph transformation systems the set of atomic properties AP is given through graph constraints (cf. Definition B.4). Satisfaction of properties for a trace t can formally be expressed as:

$$\begin{aligned} &t \models \phi \text{ with } \phi \in AP \text{ iff } G_0 \lesssim \phi \\ &t \models \phi \wedge \psi \text{ with } \phi, \psi \in \mathcal{L} \text{ iff } t \models \phi \ \wedge\ t \models \psi \\ &t \models \neg\phi \text{ with } \phi \in \mathcal{L} \text{ iff } t \not\models \phi \\ &t \models F\phi \text{ with } \phi \in \mathcal{L} \text{ iff } \exists i :\ i \in \mathbb{N} \wedge i \geq 0 \wedge t_i \models \phi \\ &t \models G\phi \text{ with } \phi \in \mathcal{L} \text{ iff } \forall i : i \in \mathbb{N} \to t_i \models \phi \\ &t \models \phi U \psi \text{ with } \phi, \psi \in \mathcal{L} \text{ iff } \exists i : i \in \mathbb{N} \wedge \forall j : 0 \leq j < i \to t_j \models \phi \wedge t_i \models \psi \end{aligned}$$

Lemma C.2 (Stutter invariant properties)
Given two traces t, t' with $t' = \natural t$ and a property $\phi \in \mathcal{L}$ then $t \models \phi \implies t' \models \phi$

Proof. *The properties that could be specified with our property language $\mathcal{L}$ only consider existence of states and do not consider the properties of the next state. The stutter operator $\natural$ removes repeating states from the trace. Obviously if the trace t contains a state s the trace $\natural t$ does contains this state, too. Only the ordinal number of the state within the trace did change. Said this it can be easily followed that the lemma holds.* □

C.3 Refinement

Definition C.3 (Trace Refinement)
Given a trace $t \in T(G_0, R)$ where G_0 and the rules in R are all typed over the type-graph T and a trace $t' \in T(H_0, R')$ where H_0 and R' are all typed over T' with $T < T'$ we say that t' refines t iff we can find a mapping $\mathrm{ord} : \mathbb{N} \mapsto \mathbb{N}$ such that the restricted trace

$$t'_r = t'_{|T,R} = H_{0|T} \to H_{1|T} \to H_{i|T} \to$$

satisfies the following conditions:

$$\mathrm{ord}(0) = 0$$
$$\forall i, j : i, j \in \mathbb{N} \wedge i > j \to \mathrm{ord}(i) > \mathrm{ord}(j)$$
$$H_{\mathrm{ord}(i)|T} \approx G_i$$
$$H_{\mathrm{ord}(i)|T} \xrightarrow{r'_{\mathrm{ord}(i)}} H_{\mathrm{ord}(i)+1|T} \in t'_{|T,R} \quad \to \quad r'_{\mathrm{ord}(i)} \prec r_i \wedge H_{\mathrm{ord}(i)+1|T} \approx G_{i+1}$$
$$\forall i, j : i \in \mathbb{N} \wedge \mathrm{ord}(i) < j \leq \mathrm{ord}(i+1) \to H_{j|T} \approx G_{i+1}$$

We denote the refinement relation between traces t and t' as $t \prec_{\mathrm{ord}} t'$ or simply $t \prec t'$ if the mapping ord is not necessary.

Lemma C.4 (Refinement invariant trace properties)
Given two traces t and p with type-graphs T and P with $T < P$ and $t \prec p$ and a property $\phi \in \mathcal{L}$, where the atomic properties of ϕ are typed over T, then $t \models \phi \implies p \models \phi$ holds.

Proof. *The property ϕ can only be defined atop of types given in T, as otherwise $t \models \phi$ could not hold. Thus, it is sufficient to have a look at the restricted trace $p_{|T,R_t}$. Definition C.3 gives us that $p_{|T,R_t}$ is equivalent to t modulo the stuttering[2]. Together with Lemma C.2 we get that the lemma holds.* □

C.4 Syntactical Refinement

A concept we will often need in the following sections is the concept of rule-sets.

Definition C.5 (Rule sets)
A rule-set $S = (R, p, \mathcal{R}_u)$ is given through a set of rules R, a transitive preemption relation $p : R \times R$[3] and a set of urgent rules $\mathcal{R}_u$. The rules in $\mathcal{R}_u$ have to be applied once they are enabled the other rules can be applied. A rule $r \in R$ can be applied to a graph G if $G \xrightarrow{r} G'$ is a valid graph-rule application and if not exists a rule $r' \in R$ with $G \xrightarrow{r'} G''$ and $(r', r) \in p$, i. e. if no rule applicable rule r' exits that preempts the rule r.

[2] In p_{lvertT,R_t} occur only states that occur in t, too. But some states, that stem from rule applications, which have no correspondent in R_t become repeated.

[3] Note that the preemption relation p, can sometimes be expressed through a total mapping $R \mapsto \mathbb{N}$.

The compositional verification scheme that is presented in this thesis strongly relies on the notion of rule refinement. In this section we will introduce how refinement is formally defined and how this could be algorithmically checked, within our tool.

Definition C.6 (Rule Refinement)
Given two graph transformation rules R_1 and R_2, that are typed over the type-graphs TG_1 and TG_2 with $TG_1 < TG_2$, we say that R_2 refines R_1, if there exists a graph morphism m with $L_{R_1} \mapsto_m L_{R_2}$ and if the elements in $L_{R_2} \setminus ran(m)$ are typed over $TG_2 \setminus TG_1$.

For hybrid graph transformation rules having jump-conditions ϕ_1 and ϕ_2, respectively, we require that $\phi_2 \equiv \phi_1 \wedge \phi_2'$. We denote the refinement relation between R_1 and R_2 as $R_1 \prec R_2$.

Informally spoken, two rules are in a refinement relation if the more concrete one enhances the more abstract one's precondition without removing any elements. Hence, the applicability of the rule gets decreased through refinement. The above definition also takes care that the original rule's negative application conditions do not become violated through the refinement. This is due to the required existence of an isomorphism between the original and the refined rule.

Definition C.7 (Rule-set Refinement)
Given two sets of graph-transformation rules $R = \{R_1, R_2, \ldots, R_n\}$ and $R' = \{R_1', R_2', \ldots, R_m'\}$ with $n \leq m$. We say that set R' refines set R if for each trace $t' \in T(R')$ there exists a trace $t \in T(R)$ such that $t' \prec t$. We write this as $R' \prec R$.

Refining rule-sets faces several difficulties. In general rule sets might be augmented with priorities, such that the applicability of a rule depends on the non-applicability of all other rules having a higher priority. In the refined rule set it has to be assured that rules, being preempted in the abstract variant are also preempted in the refined variant. In Figure C.1 a sketch of this situation is depicted. Above the dashed line the LTS of the abstract rule-set is shown and the refined one below. In the abstract LTS rule r_2 is applicable but preempted by the also applicable rule r_1 due to r_1's higher priority. The refinement of r_1 and r_2 to r_1' and r_2', respectively, may end in a situation where r_1' is not applicable (remember: refinement means strengthening the rule's precondition), but r_2' still is. Hence, in the refined system exists a trace, which is impossible to exist in the abstract one and following the refined system does not simulate the abstract one any longer.

In consequence we have to ensure that whenever a graph-rule preempts the application of a lower priority rule, this preemption does also occur in the refined set of graph-rules. The easy way to do this is to prohibit the refinement of any rule that has not the lowest priority. Obviously this restriction would severely limit the applicability of our approach. A more versatile solution is to allow refinement for such rules, but require that for each possible situation an applicable refined rule exists in the refined rule-set. Being more precise the rules r_1 and r_2 will generally be refined by a two sets of rules $r_{1,1}' \ldots r_{1,n}'$ and $r_{2,1}' \ldots r_{2,m}'$. For the refined rules it has to hold, that whenever the rule r_1 would be applicable and additionally any of the lower-priority rules $r_{2,1}' \ldots r_{2,m}'$ is applicable, then we require also at least one of the rules $r_{1,1}' \ldots r_{1,n}'$ is applicable, which then preempts the lower priority rule refining r_2. In the following we will denote this characteristics by a predicate $Preempt(R, Q)$ where R is a set of graph-transformation-rules and Q is a set of graph-transformation-rules that refine R. $Preempt(R, Q)$ is true whenever the refining rules in Q preserve the preempting behavior of the rules in R.

For timed- and hybrid graph-rules we also have to consider the urgent rules. These rules are comparable to higher priority rules and thus the arguments made above also hold for them. If urgent rules are refined, the set of refined rules has to capture all possible application situations. Meaning that whenever an urgent rule r_u would be applicable we have to have at least one rule in the set of refining rules $r_{u,1}' \ldots r_{u,n}'$ that is applicable, too. The difference here compared to the preemption of rules, which is discussed in the paragraph above, is that urgent rules have to be applied and behavior that is captured

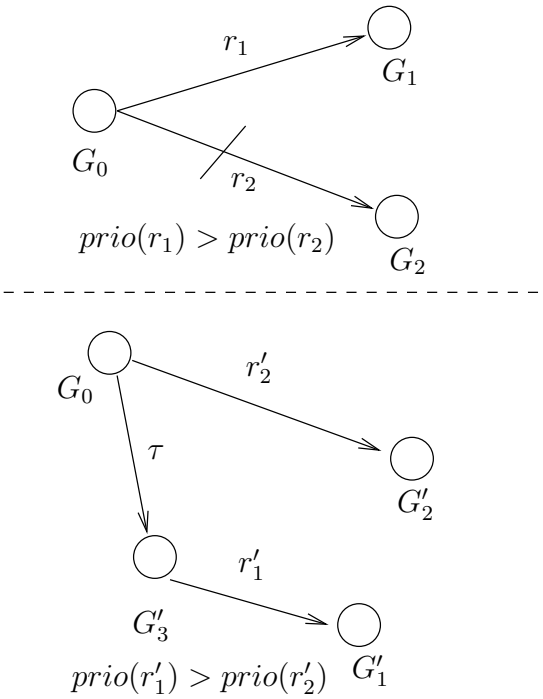


Figure C.1: Rule Refinement Sketch

by urgent rules has to happen. Otherwise safety criterions often could not be met by the system. In analogy to the $Preempt$-predicate we define a predicate $Urgent(R, Q)$.

Lemma C.8 (Rule-set Refinement)
Given two sets of graph transformation rules $RS = (R, p, \mathcal{R}_u)$ with $R = \{r_1, \ldots, r_n\}$ and $RS' = (R', p', \mathcal{R}'_u)$ with $R' = \{r_1, \ldots, r_m\}$. $RS' \prec RS$ if

$$\begin{aligned} \forall r_i, r_j, r'_i, r'_j : r_i, r_j \in R \wedge r'_i, r'_j \in R' \wedge r'_i \prec r_i \wedge r'_j \prec r_j \wedge (r_j, r_i) \in p \\ \implies (r'_j, r'_i) \in p' \\ Preempt(R, R') \wedge Urgent(R, R') \end{aligned}$$

With the set $S(r)$ for $r \in R$ being given as $S = \{r'_k | r'_k \in R' \wedge r'_k \prec r_i\}$

Proof. *According to Definition C.7 the rule-set R' refines the rule-set R iff for all traces $t' \in T(R')$ we can find a trace $t \in T(R)$ such that $t' \prec t$. We will prove the lemma by contradiction and thus we assume that a trace $t' \in T(R')$ exists such that no trace corresponding trace in $T(R)$ can be found. Without loss of generality we further assume that we have two traces $t' \in T(R')$ and $t \in T(R)$ that are in a valid refinement relation for the first i steps in the abstract system and then diverge. By construction the divergent behavior can only originate from rules that are present in both the abstract and the refining rule-set. Rules that only occur in R' and do not refine a rule in R are only allowed to read elements of R's type-graph but not to write them. Thus, the steps $G_i \xrightarrow{r_i} G_{i+1}$ and $G'_j \xrightarrow{r'_j} G'_{j+1}$ with $j \geq i$ must be performed through rules $r_i \not\prec r'_j$. We have three different possibilities for this situation to occur: (i) $r'_j \prec r_k$ with $(r_k, r_i) \in p$, (ii) $r'_j \prec r_k$ with $(r_i, r_k) \in p$, and (iii) $r'_j \prec r_k$ with $(r_k, r_i) \notin p \wedge (r_i, r_k) \notin p$[4].*

[4] This depicts a situation were both rules can not preempt each other.

For (i) we have $r'_j \to G'_j$ and $r_i \to G_i \not\leftarrow r_k$ with $(r_k, r_i) \in p$. If r_k would be applicable to G_i the application $r_i \to G_i$ would be preempted due to $(r_k, r_i) \in p$. However, if $r'_j \to G'_j$ and $G_i \approx G'_j$ and $r'_j \prec r_k$ then by construction $r_k \to G_i$, what contradicts our assumption of divergent behavior.

Case (ii) describes a situation where a preempting rule is not applicable in the concrete trace and thus a preempted rule is applied. Hence, we have $r'_j \to G'_j$, $r_i \to G_i$, $r_k \to G_i$ and $(r_i, r_k) \in p$. Further, it must exist at least one rule $r'_i \in R'$ that refines r_i. For this rule r'_i we know that $(r'_i, r'_j) \in p'$, but as $r'_j \to G'_j$ is applied $r'_i \not\to G'_j$ must hold. However, the rule r'_i cannot exist in our refined rule-set R' as $Preempt(R, R')$ and $Urgent(R, R')$ holds.

Finally, the third case remains. We have $r'_j \to G'_j$, $r_i \to G_i \leftarrow r_k$ and $(r_k, r_i) \notin p \wedge (r_i, r_k) \notin p$. Thus r_k is applicable in G_i and following a trace of the rule-set R has to exist, that applies rule r_k instead of r_i in this case, we do not have a divergent behavior a position i. The applicability of r_k to G_i is given through the applicability of r'_j to G'_j. Hence in this case, the behavior does not diverge, as we only need one trace in the abstract rule-set that conforms to the refined rule-set.

We have shown that either the divergent behavior does not exist or that the assumptions we made earlier were wrong. Following we can conclude that the assumed existence of a trace, showing divergent behavior, was wrong and thus the lemma holds. □

The lemma above is necessary for the general case, however, in some situations it is possible to ease the checking obligations due to the rules' properties. This is the case for the combination of two rule-sets, which are defined on disjoint type-graphs.

Corollary C.9 (Combination of rule-sets)
Given two rule-set $RS_1 = (R_1, p_1, \mathcal{R}^1_u)$ and $RS_2 = (R_2, p_2, \mathcal{R}^2_u)$ that rules in R_1 and R_2 are defined above two disjoint type-graphs TG_1 and TG_2, we can construct a combined rule-set $RS = (R, p, \mathcal{R}_u)$ with $R = R_1 \cup R_2$, $p = p_1 \cup p_2$ and $\mathcal{R}_u = \mathcal{R}^1_u \cup \mathcal{R}^2_u$. For the new rule-set holds, that $RS \prec RS_1$ and $RS \prec RS_2$

Proof. *The disjoint type-graphs TG_1 and TG_2 imply that all graphs that are reachable for RS can be partitioned into two unconnected parts, one being typed over each type graph. We know further, that applications of graph-transformation-rules do not consume time. Consequently, the predicates $Urgent(RS, RS_1)$, $Urgent(RS, RS_2)$, $Preempt(RS, RS_1)$ and $Preempt(RS, RS_2)$ hold. Further, each rule r refines itself and hence we have the necessary conditions for Lemma C.8.* □

As a consequence of Corollary C.9 we get that, if $RS_1 \models \phi_1$ and $RS_2 \models \phi_2$ with ϕ_1 being typed over T_1 and ϕ_2 being typed over T_2 then $RS \models \phi_1 \wedge \phi_2$. This follows directly from the fact that the two type-graphs are strictly disjoint and the $Urgent$ and $Preempt$ predicates are satisfied.

In the case that the rule-set we want to combine are not defined over strictly disjoint type-graphs, but are "pseudo-type separated" Corollary C.9 can not be applied.

Definition C.10 (Pseudo-typed graphs)
Let P be a set of nodes called pseudo-types. A graph G is pseudo-typed iff each node in $v \in V_G \setminus P$ with $l_v(v) \notin CM$ is adjacent to exactly one node $n_P \in P$. A graph transformation rule R is pseudo-typed if the graphs specifying R's left- and right-hand-side, respectively, are pseudo-typed.

The definition of pseudo-typed clearly excludes control-mode instances from the need to be adjacent to a pseudo-type. But as control-modes have to adjacent to exactly one node, which has to be pseudo-typed, of course, this exception will not violate any of the further results.Pseudo-type separated rule-sets, are rule-sets whose rules are pseudo-typed as in Definition C.10. The pseudo-types have the same effect on the rule-sets as oridnary types have and thus the combination of pseudo-type separated rule-sets has the same properties as the combination of rule-sets defined above disjoint type-graphs has.

Corollary C.11 (Combination of pseudo-typed rule-sets)
Given two rule-set $RS_1 = (R_1, p_1, \mathcal{R}_u^1)$ and $RS_2 = (R_2, p_2, \mathcal{R}_u^2)$ that rules in R_1 and R_2 are pseudo-typed over disjoint pseudo-types P_1 and P_2, we can construct a combined rule-set $RS = (R, p, \mathcal{R}_u)$ with $R = R_1 \cup R_2$, $p = p_1 \cup p_2$ and $\mathcal{R}_u = \mathcal{R}_u^1 \cup \mathcal{R}_u^2$. For the new rule-set holds, that $R \prec RS_1$ and $R \prec RS_2$.

Proof. *The two rule-sets RS_1 and RS_2 are pseudo-typed. A pseudo-typed graph-transformation rule can only be applied to a graph, that is pseudo-typed over the same pseudo-type-node, but we know that $P_1 \cap P_2 = \emptyset$. Followingly, in the combined rule-set RS the rules stemming from RS_1 do not interfere with the rules stemming from RS_2. Thus, we can recall the arguments from the proof of Corollary C.9 and get the desired results.* □

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
74	978-3-86956-245-2	**Modeling and Enacting Complex Data Dependencies in Business Processes**	Andreas Meyer, Luise Pufahl, Dirk Fahland, Mathias Weske
73	978-3-86956-241-4	**Enriching Raw Events to Enable Process Intelligence**	Nico Herzberg, Mathias Weske
72	978-3-86956-232-2	**Explorative Authoring of ActiveWeb Content in a Mobile Environment**	Conrad Calmez, Hubert Hesse, Benjamin Siegmund, Sebastian Stamm, Astrid Thomschke, Robert Hirschfeld, Dan Ingalls, Jens Lincke
71	978-3-86956-231-5	**Vereinfachung der Entwicklung von Geschäftsanwendungen durch Konsolidierung von Programmier-konzepten und -technologien**	Lenoi Berov, Johannes Henning, Toni Mattis, Patrick Rein, Robin Schreiber, Eric Seckler, Bastian Steinert, Robert Hirschfeld
70	978-3-86956-230-8	**HPI Future SOC Lab - Proceedings 2011**	Christoph Meinel, Andreas Polze, Gerhard Oswald, Rolf Strotmann, Ulrich Seibold, Doc D'Errico
69	978-3-86956-229-2	**Akzeptanz und Nutzerfreundlichkeit der AusweisApp: Eine qualitative Untersuchung**	Susanne Asheuer, Joy Belgassem, Wiete Eichorn, Rio Leipold, Lucas Licht, Christoph Meinel, Anne Schanz, Maxim Schnjakin
68	978-3-86956-225-4	**Fünfter Deutscher IPv6 Gipfel 2012**	Christoph Meinel, Harald Sack (Hrsg.)
67	978-3-86956-228-5	**Cache Conscious Column Organization in In-Memory Column Stores**	David Schalb, Jens Krüger, Hasso Plattner
66	978-3-86956-227-8	**Model-Driven Engineering of Adaptation Engines for Self-Adaptive Software**	Thomas Vogel, Holger Giese
65	978-3-86956-226-1	**Scalable Compatibility for Embedded Real-Time components via Language Progressive Timed Automata**	Stefan Neumann, Holger Giese
64	978-3-86956-217-9	**Cyber-Physical Systems with Dynamic Structure: Towards Modeling and Verification of Inductive Invariants**	Basil Becker, Holger Giese
63	978-3-86956-204-9	**Theories and Intricacies of Information Security Problems**	Anne V. D. M. Kayem, Christoph Meinel (Eds.)
62	978-3-86956-212-4	**Covering or Complete? Discovering Conditional Inclusion Dependencies**	Jana Bauckmann, Ziawasch Abedjan, Ulf Leser, Heiko Müller, Felix Naumann
61	978-3-86956-194-3	**Vierter Deutscher IPv6 Gipfel 2011**	Christoph Meinel, Harald Sack (Hrsg.)
60	978-3-86956-201-8	**Understanding Cryptic Schemata in Large Extract-Transform-Load Systems**	Alexander Albrecht, Felix Naumann
59	978-3-86956-193-6	**The JCop Language Specification**	Malte Appeltauer, Robert Hirschfeld

ISBN 978-3-86956-246-9
ISSN 1613-5652