Architectural Modelling and Verification of Open Service-Oriented Systems of Systems

Dissertation
zur Erlangung des akademischen Grades
“doctor rerum naturalium”
(Dr.rer.nat.)
in der Wissenschaftsdisziplin “Informatik”

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Basil Becker

Potsdam, den 06. August 2013

Published online at the
Institutional Repository of the University of Potsdam:
URL http://opus.kobv.de/ubp/volltexte/2014/7015/
URN urn:nbn:de:kobv:517-opus-70158
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-70158

Abstract

Systems of Systems (SoS) have received a lot of attention recently. In this thesis we will focus on SoS that are built atop the techniques of Service-Oriented Architectures and thus combine the benefits and challenges of both paradigms. For this thesis we will understand SoS as ensembles of single autonomous systems that are integrated to a larger system, the SoS. The interesting fact about these systems is that the previously isolated systems are still maintained, improved and developed on their own. Structural dynamics is an issue in SoS, as at every point in time systems can join and leave the ensemble. This and the fact that the cooperation among the constituent systems is not necessarily observable means that we will consider these systems as open systems. Of course, the system has a clear boundary at each point in time, but this can only be identified by halting the complete SoS. However, halting a system of that size is practically impossible. Often SoS are combinations of software systems and physical systems. Hence a failure in the software system can have a serious physical impact what makes an SoS of this kind easily a safety-critical system.

The contribution of this thesis is a modelling approach that extends OMG's *SoaML* and basically relies on collaborations and roles as an abstraction layer above the components. This will allow us to describe SoS at an architectural level. We will also give a formal semantics for our modelling approach which employs hybrid graph-transformation systems. The modelling approach is accompanied by a modular verification scheme that will be able to cope with the complexity constraints implied by the SoS' structural dynamics and size. Building such autonomous systems as SoS without evolution at the architectural level — i.e. adding and removing of components and services — is inadequate. Therefore our approach directly supports the modelling and verification of evolution.

Zusammenfassung

Systems of Systems (SoS) sind ein seit längerem bekanntes Konzept, das jedoch in letzter Zeit vermehrt Aufmerksamkeit erhielt. Das Hauptaugenmerk dieser Arbeit wird auf SoS liegen, die mit Hilfe von Techniken aus Service-Orientierten Architekturen erstellt werden. Somit vereinen die hier betrachteten SoS die Vorteile und Herausforderungen beider Paradigmen. SoS können definiert werden als Zusammenschlüsse einzelner, autonomer Systeme, die zu einem größeren System integriert werden. In diesem Zusammenhang interessant ist, dass die ehemals isolierten Systeme nach wie vor isoliert voneinander weiterentwickelt und gewartet werden. Desweiteren kommt der Strukturdynamik innerhalb des SoS eine beachtliche Bedeutung zu, da jederzeit Systeme dem SoS beitreten und es verlassen können. Zusammen mit der Tatsache, dass die Kooperationen zwischen den konstituierenden Systemen nicht immer beobachtbar sind, führt dies dazu, dass wir diese Systeme als offene Systeme bezeichnen. Wobei das System natürlich jederzeit eine klar definierte Grenze besitzt, diese aber nur durch ein Anhalten des Systems zu bestimmen ist. Dies jedoch ist, von einer praktischen Perspektive aus betrachtet, unmöglich. Häufig stellen SoS eine Kombination aus Softwaresystemen und pyhsikalischen Systemen dar mit der Folge, dass ein Fehler in der Software eine SoS schnell eine immense physikalische Wirkung entwickeln kann. Von daher fallen SoS leicht in die Klasse der sicherheitskritischen Systeme.

In dieser Arbeit werden wir einen Modellierungsansatz vorstellen, der die Sprache *SoaML* der OMG erweitert. Die grundlegenden Konzepte dieses Ansatzes sind die Modellierung mit Kollaborationen und Rollen als Abstraktionsebene über Komponenten. Der vorgestellte Ansatz erlaubt es uns SoS auf einer architekturellen Ebene zu betrachten. Die formale Semantik unseres Modellierungsansatzes ist durch hybride Graphtransformationssysteme gegeben. Abgestimmt auf die Modellierung werden wir ebenfalls ein Verfahren zu Verifikation von SoS vorstellen, welches trotz der inhärenten Komplexität von SoS, diese zu verifizieren. Die Modellierung und Verifikation von Evolution wird von unserem Ansatz direkt unterstützt.

Acknowledgements

I would like to express my deep gratitude to Professor Dr. Holger Giese, my research supervisor, for giving me the opportunity to carry out my dissertation in his research group, for his patient guidance, enthusiastic encouragement and useful critiques of this research work. My thanks also go to the HPI's Research School on Service-oriented Computing for partially supporting me with a scholarship that allowed me to carry out this research.

I would also like to thank Kerstin Miers for her support in all kinds of administrative matters.

My special thanks go to Dr. Leen Lambers and Thomas Vogel for much good advice they gave me, interesting discussions, as well as their enjoyable company during lunch- and coffee-breaks. I am particularly grateful to Johannes Dyck, who helped me with the maintenance and improvement of the Invariant Checker implementation. Dr. Andreas Seibel, Dr. Christian Krause and Johannes Dyck deserve a special mention for being congenial office colleagues.

Finally, I wish to thank Anja, my parents and friends for their continuous support and encouragement during this time.

Contents

Chapter 1

Introduction

The idea of Systems of Systems (SoS) has been around for a while [103] but has received a lot of attention recently. The available definitions for SoS range from *"network-enabled synergistic collaborations between systems that are operationally and managerially independent, distributed, evolve dynamically and exhibit emergence"* [68] through simply *"collaborative systems"* and *"systems that are built from components which are large-scale systems in their own right"* [103] to a synonym for ultra-large-scale systems (ULSS) [107], which are defined as *"a dynamic community of interdependent and competing organisms (in this case, people, computing devices, and organizations) in a complex and changing environment"*. For this thesis, we will use Valerdi et al.'s [127] understanding of SoS, which defines SoS as ensembles of single autonomous systems that are integrated into a larger system, the SoS. The interesting fact about these systems is that the systems isolated beforehand are still maintained, improved and developed on their own. Further, structural dynamics is an issue in SoS as at every point in time systems can join and leave the ensemble. The relevance of SoS is emphasized through several EU-funded projects, which investigate different aspects of SoS. The Compass project (`http://www.compass-research.eu`) addresses the development and maintenance of SoS. The evolution and adaptation of lifecycle-models for SoS is the research area of the DANSE project (`http://www.danse-ip.eu`). Certification issues of SoS are the main interest of the OPENCOSS project (`http://opencoss-project.eu`).

SoS are not necessarily pure software systems. Often they are combinations of software systems and physical systems. A different term, that is often used for this class of systems, is Cyber-Physical-Systems [98, 80] (CPS) or Networked-Cyber-Physical-Systems [124] (NCPS). NCPS are dynamic, distributed systems that are tightly integrated with their environment. Generally, CPS require capabilities to sense and change the environment through sensors and actors, respectively. Typical application domains for NCPS are traffic control, train, vehicular and avionic systems. E.g. for a vehicular system (cf. [30]), the system's state is determined not only by its software parts but also by the values of the vehicle's physical properties. For a motor car, the decisions that have to be made might depend, among other properties, on the car's velocity and acceleration.

An important point, which is discussed by Ghezzi et al. [34] for SOA, but which holds also for SoS, is that they are so-called open-systems. For an open system, it is impossible to determine the system's boundaries without halting the system. This argument is easy to follow if one recalls that the only information available is that stating which service one currently uses. The services that are invoked by the providing component in order to fulfill one's service request are hidden from the user.

Evolution is a classic challenge in software engineering. Lehman [99] and Parnas [111] made a clear point that evolution is an inevitable phenomenon as software otherwise is decreasing in its utility. The task of establishing a profound evolution strategy is already hard for a monolithic system. In the context of SoS with different managerial authorities and the uncoordinated evolution of constituent systems [103], the problem becomes even harder. Consequently, support for evolution has to be a requirement during the SoS's design phase.

So far, we have introduced the system types, SoS, ULSS and NCPS, which all share the commonality that they contain an inherent need for adaptation, are of unknown size, are hard to halt and, lastly, are often used in safety-critical environments. The safety-critical environment makes it directly necessary that the system's behaviour is well known or at least that it is guaranteed that the system stays within the boundaries of predefined safety properties. Thus, a sound automatic verification mechanism for these systems is required.[1] Verification can always be used to verify different properties of a system. So, beside the differentiation between liveness- and safety-properties, probabilistic methods also allow us to check the system's expected availability and other non-functional properties. Structural safety properties, however, are to some extent characteristic for SoS. The systems' high flexibility directly raises the questions of which structural combinations — i.e. connections of constituent systems — are possible within the SoS and which aren't. Thus, for an automatic verification technique to be applicable, several challenges have to be solved. The system's size is not certain. Each single constituent system has a different managing authority, and thus the constituent system's exact behaviour specification is probably the authority's intellectual property and not publicly available. New systems with previously unknown behaviour can be introduced to the SoS at any point in time. Established and well-known analysis techniques, such as model-checking — independent of the underlying mechanism to build the state space — are, due to the extremely large and probably infinite state space, not applicable out of the box. Testing, although very useful at the implementation level, can only be used if the constituent systems are known. Further, the large state space makes it complicated to reach a good test-coverage. However, for self-adaptive systems [14], which to some extent have similar characteristics, such as architectural reconfiguration and incomplete knowledge, currently no suitable verification technique exists (cf. [19]). That reconfiguration or self-adaptation can, in principle, be facilitated to design a dependable system is pointed out by several authors [50].

An implementation technique that is often used for the realisation of SoS and

[1] Verification can be understood in several different ways. In this thesis, we will use the term verification in the sense of formal verification and we will only consider automatic verification techniques.

ULSS (cf. [107]) is Service-oriented Architectures [64] (SOA). SOA are widely used to build distributed and loosely coupled systems and are proposed as a solution to build open and organisationally separated systems (cf. [128]). Further, they offer great capabilities to integrate legacy systems into a new environment. Within an SOA, the question of how the overall system is controlled arises directly. Two main techniques have emerged and are used in practice: orchestration and choreography. In an orchestrated SOA, one or more central controllers coordinate the behaviour of the components involved. A choreography, however, is the decentralised counterpart to an orchestration. If we want to use SOA as an implementation paradigm for a SoS, both coordination mechanisms could be employed; however, as it is inevitable that within an SoS organisational borders are crossed, a choreography is more likely to be applied. An orchestrated SOA is likely to be found within organisational bounds. Orchestration and choreography are not exclusive choices, but can also be used in combination.

Beside this flexibility, service-oriented systems also introduce a further layer of abstraction atop of components. Service-oriented systems describe the behaviour that occurs between the components involved, at a more abstract level, than components do. The behaviour is assigned to roles that can be implemented by components. This allows us to specify and analyse the behaviour of the services, independent of the components.

Verification requires a rigorous modelling approach that has a clear formal semantics. For the modelling of SoS, often a role-based modelling approach using contracts between roles is proposed [49]. This directly matches the implementation paradigm of service-oriented architectures, which are also specified at a similar level of abstraction. *SoaML* [28] is a modelling language, i.e. a profile and meta-model for UML 2, that puts strong emphasis on the definition of roles and contracts, but *SoaML* does not have the formal underpinning that is required to run a full formal verification. Therefore, we will extend *SoaML* in a suitable way and augment it with the necessary formal background. Our extension of *SoaML* will be called *rigSoaML*.

1.1 Running Example

The application example we will use throughout this thesis is a supply chain system. We will use this example to illustrate the modelling and verification aspects. Basically, the system consists of two different roles: customers and suppliers. A customer has the capability to buy goods at the market and the supplier can sell goods. Selling and buying goods is achieved through a coordination of supplier and customer in a service contract called “Contract Collaboration”. Components can implement both of the two roles, and the abstract super type of such a component is called a “Factory”. The “contract collaboration” service contract has an attribute “recall date” that states when the service contract can be recalled. We want each supply chain system to follow this restriction: A company can only deliver its goods to a customer if all of its own customer roles are equipped with an active service contract. Hence, the supply chain system’s rules have to ensure that only such contracts are signed – i.e. instantiated – that satisfy this restriction. Figure 1.1 shows an exemplary sketch of a supply chain

system. The components are depicted by a factory pictogram and the service contracts are indicated through the arrows. The supplier role is at the start of the arrow and the customer role at the end.

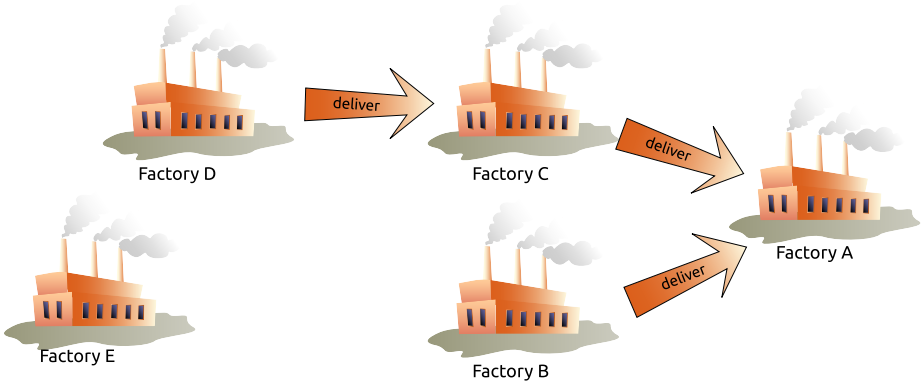


Figure 1.1: Sketch of the Supply-Chain running example

Depending on the business domain, different flavours of negotiation processes can be used. In the application example, this is modelled through different service contracts. The abstract service contract only specifies that a contract element can be created between consumer and producer, without considering any negotiation. An extension of this basic service contract is the introduction of a negotiation stage that comprises a customer request, followed by a producer's offer and concludes with the creation of a contract. An alternative extension would be, e.g. , an auction like a negotiation stage, where the consumer role is extended to also make bids. After a given period of time, the consumer with the highest bid would be selected and the contract created.

The supply chain system makes it very obvious that the systems we are investigating are open and that it is a very hard, yet unsolvable task to determine the system's current structure, firstly, due to the system's size and secondly, because companies do not have any interest in making their deliverer public, as this could undermine their market position.

1.2 Goals of the Thesis

In this thesis, we want to address two goals that are relevant in the context of SoS. These are *verification* and *modelling* of the architectural properties of SoS.

Goal Verification The main goal we want to achieve in this thesis is to develop a verification scheme for compositional reasoning about SoS. The verification approach should be suitable to verify discrete as well as hybrid systems. Further, the approach should scale to the size of SoS and be generally applicable.

Goal Modelling In order to achieve our main goal, we also have to provide a modelling language that is able to describe SoS in a sufficient degree of abstrac-

tion. We will therefore extend OMG's *SoaML*. The modelling approach should be able to express the structural dynamics within the SoS as well as having a precise understanding of inheritance and the preconditions and consequences. Further, the approach has to have a formal underpinning. We will therefore present a mapping to graph-transformation systems.

This thesis is organised as follows. In Section 1.1 we have introduced our running example. In Chapter 2 we discuss the requirements that a modelling and verification approach for SoS has to fulfil. Chapter 3 introduces the current state of the art of modelling with *SoaML* and in Chapter 4 we describe how we overcome the weaknesses of *SoaML* with our modelling extension *rigSoaML*. The verification scheme we have developed to allow the verification of SoS is explained in Chapter 5 and the detailed techniques will be presented in Chapter 6. In Chapter 7 we present our verification tool the Invariant Checker. We evaluate the tool and the modelling approach in Chapter 8. A comprehensive discussion of related work is contained in Chapter 9. Chapter 10 concludes the thesis with a summary and an outlook to future work.

Chapter 2

Requirements

In this chapter we will discuss the different origins of service-oriented systems of systems and deduce a set of requirements that have to be fulfilled by a suitable modelling and verification approach for these systems.

2.1 System of Systems

For systems as complex as SoS, a magnitude of requirements can be defined. Beside the classical expectations of a software system, for an SoS several other requirements have to be met that are implied by the systems' size and different organisations developing constituent systems of the SoS. However, for this thesis we want to focus on requirements that arise in the context of SoS and that are relevant for our aim of verifying these systems. Therefore, the requirements can be separated into two parts: self-adaptation as one characteristic of SoS and service-oriented architectures as the implementation paradigm we follow.

One important aspect of SoS is that the constituent systems remain independent of each other and each of these systems has it's own managing authority [100]. Although these different managing authorities have to cooperate with each other to make the overall SoS run, they probably have no interest in publishing the complete implementation of their systems. They want to keep their intellectual property (IP) and publish only those details that are necessary to cooperate within the SoS. Additionally, as a result of the openness (cf. [34]) the boundaries of the SoS and hence all its constituents are unknown.

2.1.1 Self-Adaptation

Self-adaptation and self-adaptive systems are used whenever the system has to deal with contradictory objectives [132]. These lead to the observation that more than one optimal configuration of the system exist and the best one cannot be determined at the design time. The optimal configuration of a self-adaptive system depends on the system's current configuration, the goals it is to achieve

and the environment the system is operating in. However, self-adaptive systems are also well suited to capturing the problem of system evolution, as they are built to deal with contradictory but also with changing objectives and requirements [29].

Independent of the technique that is used for adaptation, a self-adaptive system requires several parts that allow us to recognise its own and the environment's state, to analyse these, and to make decisions based on the previous analysis and finally implement the decision. A very prominent way to represent these four parts is the MAPE-K loop [93], which first appeared in the context of Autonomic Computing. However, the three layers architecture [96] introduced to the SEAMS community by Kramer and Magee contains the artifacts of the MAPE-K loop as well. In self-adaptive systems, which have a single or few nodes that are responsible for analysing and deciding which adaptation / reconfiguration should be applied, we say that these are top-down self-adaptive systems. Bottom-up self-adaptive systems, in contrast, consist of lots of independent units that decide on their own which local adaptation / reconfiguration is to be applied (cf. [14]).

The techniques that are typically applied to enable self-adaptation range from dynamic architectures, which allow for high-level reconfigurations, to aspect-oriented programming, which allows for changing systems at a very fine-grained level. In this thesis, we will focus on self-adaptation at an architectural level, which is widely accepted (cf. [109]). At this coarse-grained level, most of the changes that occur are addition and removal of components, together with changing associations between components.

2.1.2 Service-Oriented Architecture

Service-oriented architectures are an approach to coping with current software systems' increasing complexity. Service-oriented architectures raise the level of abstraction by making services first-class citizens. In a complete service-oriented world, the developer's task is to combine the services in a meaningful way. Typically, process languages such as e.g. BPEL (Business Process Execution Language) are employed for this. Hence, in service-oriented architecture the interplay of different components is of great importance. Further, service-oriented architectures are used whenever a fixed binding of components is not desirable. Scenarios in which a flexible architecture is required range from mobile systems that have to deal with alternating availability of components to business applications that often have to change their business partners to reduce costs. For a service-oriented system, this flexibility is often described as "loosely coupled" and requires that components can be looked up and bound at run-time.

For service-oriented architectures, two different kinds of coordination are mainly used, called orchestration and choreography. In the orchestration-based service-oriented architecture one or a few central components orchestrate the architecture's behaviour. They trigger exactly which component performs which action at which point in time. Orchestration is mainly used within organisational units, as it requires control to be given to a central component. If this is not

applicable, a different approach is used: choreography. In a choreography, the responsibility for the fulfilment of the service is spread among the participating components. A choreography is often used for the communication between different organisational units, i.e. different companies. In one service-oriented architecture, both orchestration and choreography can exist in parallel.

Each participant in a service-oriented architecture only knows and recognises the interfaces that are specified by the choreography, but fails to identify what happens behind the interface. Obviously, this is necessary if two or more companies are involved in a common SOA. Each company only wants to provide the information that is already known, but wants to protect its internal structure. Therefore, it is impossible to decide whether a component provides a task on its own or if the component relies on other services or internal components.

Service-oriented systems in the flavour we are investigating in this thesis are so-called open systems. The term open-systems describes systems which have no pre-defined or easily detectable boundaries.[1] Consequently, the size, i.e. number of participating components, of open-systems can range from only a few to an ultra-large-scale system (ULSS). Even if it were possible to determine the actual system configuration for a given point in time, it cannot be said that this configuration will not change again. One of the service-oriented architecture's key characteristics is flexible bindings, meaning that for each instantiation of a service contract the participating component might differ and for each of the service contract's roles different component types might be available. Consequently, a verification approach for these systems is required to scale to the system sizes and to be robust enough against the uncertainty that is involved with the system's current configuration.

2.2 Modelling

Our goal of "verification" can only be reached if we provide a sufficiently exact modelling approach. Therefore, we have to provide a modelling technique that is suitable to model open service-oriented systems of systems, having all the capabilities and requirements that we have introduced in the previous sections. Obviously, a modelling language for open service-oriented systems of systems needs first-class concepts to express service contracts and components. Service-oriented architectures became popular as a means to keep track of software's increasing complexity. In service-oriented architectures, complexity is managed by abstraction and separation. That is, definitions of service contracts are independent of the implementing components. Hence, a modelling approach for service-oriented architecture should support the separation at the interfaces.

The modelling approach should directly support the expression of structural changes to the system. By structural change, we mean the instantiation and destruction of service contracts and components as well as the bindings between them.

[1] For service-oriented systems that are mainly used to ease the problem of system-integration, this does not necessarily hold true.

Complexity typically implies that software engineers reuse existing solutions with well-known – and, in our context, verified – properties to develop a system. Therefore, for a modelling approach it is required that existing parts can be reused in a new system. Reuse, however, can occur at two different levels. First is at the modelling level, if existing parts — i.e. service contracts and components — are incorporated into a new system. The second use-case is verification, where it will be beneficial if parts only have to be verified once.[2] Further, reuse is an appropriate technique to adapt an existing solution to one's special needs.

2.3 Verification

Concerning the verification of self-adaptive service-oriented systems that show all of the aspects we have sketched in the above paragraphs, verification becomes an issue. Generally, verification is a hard task and it almost always requires good software engineering and modelling skills to be feasible. However, in self-adaptive service-oriented systems things get worse, as the system size is unknown [34], systems, in general, cannot be stopped [107], and evolutionary changes cannot be foreseen or planned [34]. Therefore scalability is a key issue for this class of systems. The second issue is to find a technique or a set of techniques that can be applied in such a demanding environment. Even if we do not consider the problem of state space explosion, it is hardly thinkable that a monolithic verification approach could be successfully applied to SoS. Testing has the drawback that we do not know in advance what should be tested. Thus, we can test the correct interplay of some components, but we do not know the actual combination of components that interact at run-time.

2.4 Evolution

The necessity of evolution for any kind of system has already been discussed by Parnas [111] and Lehman [99]. An important characteristic property of Systems of Systems is that the single constituent parts evolve separately and in an uncoordinated way. Evolution, as we discuss it in this thesis, ranges from the introduction of completely new service contracts and components to the incremental improvement of existing ones. We discussed in Section 2.1.2 that systems of systems are so-called open systems; thus, a comprehensive view of the system does not exist and following the modelling approach should not require us to have such a comprehensive view.

2.5 Scenarios

In the previous sections we have characterised open service-oriented systems of systems as systems that are very flexible with regard to their structure. This

[2] Of course, this can only be achieved if the specification that the reused models have to fulfil, does not change

flexibility can be expressed in several scenarios that are common to open service-oriented systems of systems. Our modelling and verification approach will have to support them.

Introduction of new services In an open service-oriented system of systems it has to be assumed that, after the initial system deployment, new services are added to the system to adapt the system's capabilities to changed requirements. If we add a new service, the new service can (i) be a specialisation of an already existing service, or (ii) implement a completely new concept. In (i) we can derive the new service from the existing one through inheritance. For case (ii), the new service is placed at the system's topmost specification level. From the verification point of view, we have to ensure in both (i) and (ii) that the newly added service specification invalidates neither the system's nor the service specification's safety properties.

In case (ii) the modelling is very straight forward. We cannot reuse any of the existing service specifications, so the modelling can be done independently of the already deployed service specifications. In (i), this is fundamentally different. We have to know at least the exact specification of the service we want to extend and of all its predecessors in the inheritance hierarchy.

Removal of a service The counterpart to the addition of a new service specification is its removal. Again we can decide between two layers, the specification layer and the instance layer. Whereas the removal of a service instance is easy and the decision to remove a service instance can be made locally by the involved roles, the removal of a service specification is a global task. It is necessary to avoid having instances without specification remaining in the system. The complexity here clearly arises from the open and distributed system setting we consider.

Addition of new component Components are those parts of the system that can actually be executed – i.e. components have a behaviour. Therefore it is crucial that not only new services but also new components can be added. Moreover, each newly added service specification most often requires new components that implement the service specification's behaviour. As for service specifications, we have two possibilities for components. Either they were added at the top level of the system specification or they were added as specialisations of existing components. The implications concerning verification and modelling are the same as for newly added service specifications.

Removal of Components For the removal of components the same arguments and reasons as for the removal of a service holds.

Update of Components Instead of removing and adding a component, it can also be the case that the components become updated dynamically during run-time. In this scenario, the update can be either immediately applied to all running instances of the component or applied to newly created instances only.

2.6 Summary

The previous sections can be summarised to a list of requirements or challenges that a modelling language and a verification approach for systems of systems has to fulfil. The list does not contain requirements that are specific to the run-time situation of such systems, as we will not develop a run-time environment.

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

Modelling 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.

Modelling Evolution (M3) The modelling has to cope with the uncoordinated introduction of new types of service contracts and components at run-time.

Scalable Analysis (A1) Verification has to scale with system size.

Applicable Analysis (A2) The verification methodology has to be applicable.

Analysis of Reconfiguration (A3) The systems, in general, change their configuration at run-time. Therefore we cannot check each configuration in isolation but have to consider the interplay between the regular behaviour and reconfiguration behaviour

Analysis under restricted knowledge (A4) The analysis of open service-oriented systems of systems has also to work 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.

Analysing Evolution (A5) Also, the analysis has to cope with the uncoordinated introduction of new types of service contracts and components at run-time.

Chapter 3

Modelling with *SoaML*

The modelling language for service-oriented architecture proposed by the Object Management Group (OMG) is called *SoaML*.[1] *SoaML* fulfils most of the above requirements (cf. Section 2) and is an implementation of the *OMG UML Metamodel and Profile for Services* [28]. *SoaML* builds atop of the UML 2.x specification. 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 modelling tools.

3.1 Modelling Concepts

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

3.1.1 Services 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 modelled as a service contract collaboration. Typically, a service contract comprises roles and a behaviour – the service's choreography. The choreography can be modelled using any UML behaviour specification, such

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

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. In Figure 3.2 and Figure 3.3 we show the two service contracts ContractCollaboration and RequestOfferCollaboration in a single ≪ServiceArchitecture≫ collaboration, as *SoaML* uses the ≪ServiceArchitecture≫ stereotype for both the specification of single service contracts and the specification of the complete system.

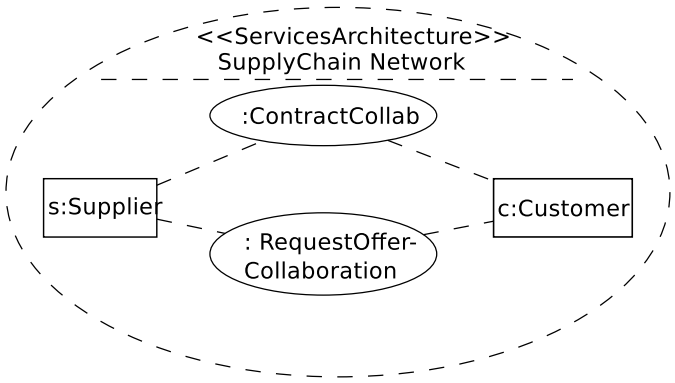


Figure 3.1: The ≪ServicesArchitecture≫ for the SupplyChain application example

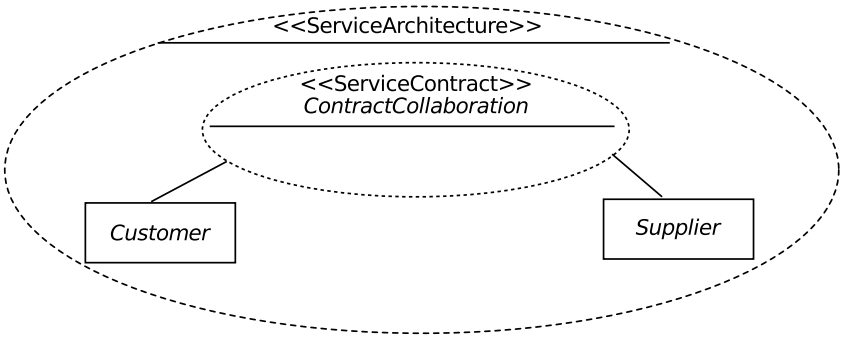


Figure 3.2: Contract Collaboration Structure

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 behaviour. Figure 3.4 depicts the ParticipantArchitecture for a factory from our application example. The Factory

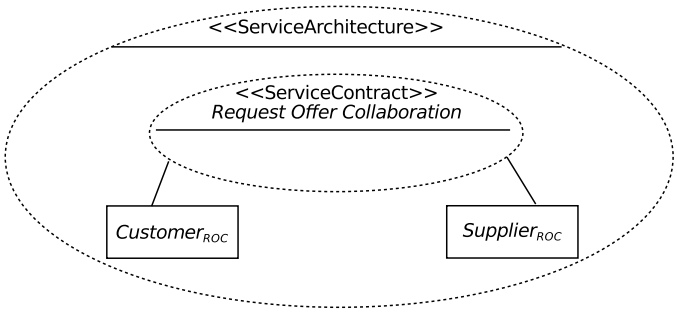


Figure 3.3: Services Architecture for the RequestOfferCollaboration collaboration type

component shown 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 fulfil 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 from the contract service contract is the fact that now a Controller component initiates behaviour and tells the others which action they actually have to execute.

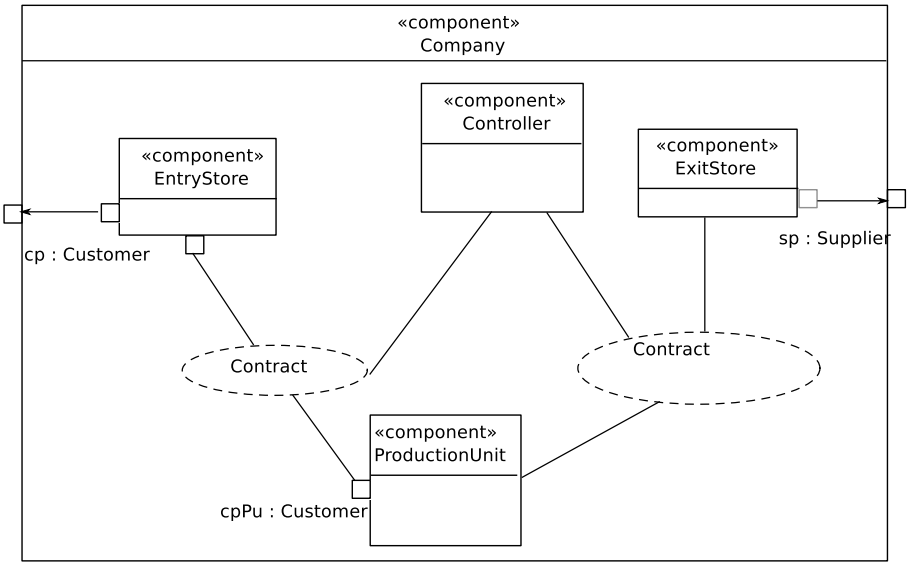


Figure 3.4: ParticipantArchitecture for the Factory participant.

The complete supply chain network can be modelled as a collaboration consisting of the roles Supplier,Customer (cf. Figure 3.1). The services that exist in this architecture are ContractCollaboration and RequestOfferCollaboration. The ContractCollaboration service contract requires all roles, i.e. Customer and Supplier to be bound in order to be established. The same holds for the RequestOffer-

Collaboration. In our application example, components can play different roles, and also more than one. The supply chain application example we describe as 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, for example, be special factory components that only use either their Supplier or Customer roles.

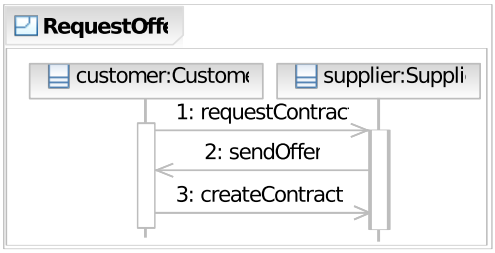


Figure 3.5: A UML interaction diagram showing the RequestOfferCollaboration service contract

Each service is also connected to a specific behaviour, the service choreography. For the RequestOfferCollaboration service contract the choreography is depicted in Figure 3.5. The behaviour 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 participants involved. The sequence diagram depicted in Figure 3.5 only depicts the scenario for a successful negotiation between Supplier and Customer. The Auction service contract also aims to establish a valid Contract between a Supplier and a Customer, but also owns a negotiation phase – in the form of a classical auction – where the highest price for the offered 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 may also leave the service contract at any time unless he is the current leader in the auction.

3.2 Discussion of *SoaML*

In Chapter 2 we articulated several requirements a modeling approach has to fulfill. For *SoaML* obviously only the requirements related to modeling are of interest, as verification is not in *SoaML*'s scope. In the following we will shortly discuss, whether or not *SoaML* fulfills our requirements.

SoaML supports reuse due to its relation to the UML. Collaborations and components are allowed to inherit from each other, but unfortunately only a syntactic conformance is required by UML and *SoaML*. By syntactic conformance,

we want to describe how at least the same roles and interfaces have to be available. The exact meaning of inheritance with respect to the collaborations' and components' behaviours is undefined. Of course, the intention is that the inherited collaboration or component should behave similarly to the super-type, but *SoaML* does not provides any technique that enforces this. For the verification, this leads to a situation where no previous verification results can be reused and the complexity of the verification tasks increases with the system size.

Although *SoaML* supports modelling of composite service contracts, it is unable to express the creation and deletion of service contracts and the connection of components, implementing roles, according to the service contract specifications. Basically, *SoaML* supports the modelling of the desired structure (requirement: Modelling SOA (M1)) only, but provides no way to intuitively specify the modifications that are necessary to achieve this structure (requirement: Modelling Dynamics (M2)). Of course, it is possible to facilitate some OCL pre- and post-conditions for this task, but this solution is not very intuitive, is consequently error-prone, and opens up the possibility of using the full expressive power of OCL, which in turn prevents verification.

Also, evolution is not directly supported and thus neither the modelling nor the analysis of the evolution of systems as raised by challenge Modelling Evolution (M3) is covered.

Requirements	Coverage
Modelling SOA (M1)	✓
Modelling Dynamics (M2)	∘
Modelling Evolution (M3)	∘

Table 3.1: Coverage of the challenges for modelling with *SoaML*. Legend: ✓ means the challenge is fulfilled, ∼ means the challenge is partly fulfilled and ∘ means the challenge is not fulfilled.

Chapter 4

Modelling with *rigSoaML*

The previous chapter yields that *SoaML* has the capabilities to model the basic constituents of a service-oriented system, but falls short of modelling the structural dynamism (cf. Table 3.1). In this chapter we will introduce our modification of *SoaML*, called *rigSoaML*, which 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 that model states and graph-transformation rules that model the behaviour. 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. As we have to capture continuous behaviour, too, we will use attributed graphs. For reasons of brevity, we will here only roughly introduce the two main concepts, typed and attributed graphs and hybrid graph-transformation rules. The complete definition of the employed formal model can be found in the appendix in Chapter B.

Definition 4.1 (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 its single constituents are not important for understanding.

Basically, an attributed graph is a typed graph that is enriched with a valuation function. The attributes that are available for each node are determined by the node's type and defined in the graph's type-graph (type-graphs are formally defined in Definition B.9). The derivation of the attributes' values over time is captured in so-called control modes (CM_T in the above definition). We will refer to the empty graph as $G_\emptyset$ and to the set of all type-graphs as $\mathcal{G}$. If a graph S is a sub-graph of graph G, we denote this as $S < G$. Isomorphic graphs are denoted as $S \approx G$.

Definition 4.2 (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 4.3 (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 $G \overset{P',m}{\Rightarrow} H$ 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.

Hence, a hybrid graph transformation rule can be applied to an attributed graph, if we find a structural match and the graph's valuation satisfies the hybrid graph transformation rule's jump condition ϕ. For our application example we will use a simplified notation that does not use 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. Therefore we will omit the control-modes and increase the figure's readability.

The semantic domain we will use throughout this thesis are traces of hybrid graph-transformation systems. Formally a trace is given as follows:

Definition 4.4 (Trace)**.** *A trace $t \in \mathcal{G}^*$ is a possibly infinite sequence of graphs. The i^{th} graph of t is accessed as $t(i)$ or G_i if it is not ambiguous. 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 trace can always contain states,*

that are reached via continuous steps. A sub-trace of trace t starting at the i-th position is denoted as t_i for $i \in \mathbb{N}^+$. A sub-trace up to position k is denoted as $t^{\leq k}$ if the state G_k is contained in the sub-trace and $t^{<k}$ otherwise.

A trace t is a possible path through a graph transformation system's (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].

Example 1: Figure 4.1 shows an exemplary trace. The trace consists of six states $G_0, \ldots, G_5$. The initial state G_0 contains only the two nodes Customer and Supplier. The first step in the trace creates the ContractCollab node. In the next steps the nodes Request, Offer and Contract between the two roles become instantiated and deleted, again. The trace contains no continuous steps.

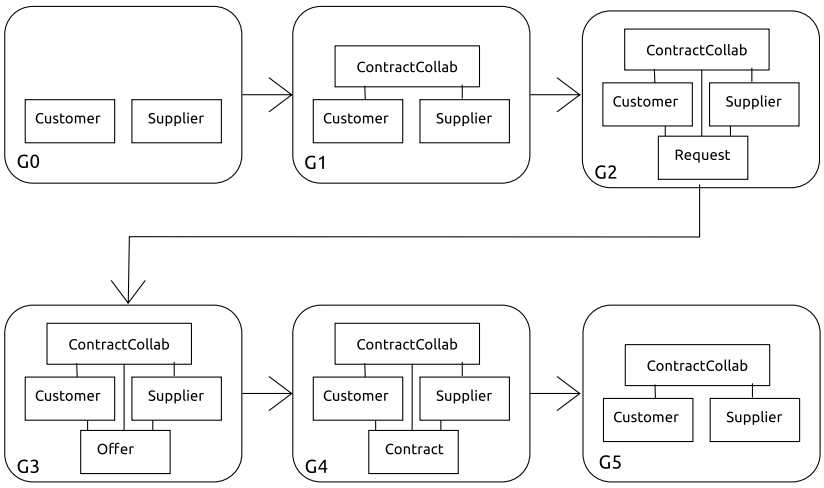


Figure 4.1: An exemplary trace for the Contract Collaboration collaboration type

Traces are used in a multitude of formalisms and in general a trace characterizes the behavior that is externally observable, without knowing any internal details. In this way traces are defined, but differently named, for Petri Nets [60] (transition occurrence sequence) or runs [79], finite automata (accepted words) and process calculi. The trace definition we used in this thesis is different as our traces contain the state, too (cf. Definition 4.4). Therefore, they are more related to a path through a Kripke structure, where the locations' atomic properties encode the internal state (cf. [40]). We decided us to use this special notion of traces, as we want to be able to reason about the structural changes that occur within a SoS.

4.1.2 Safety 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

[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.

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 can be formally defined as follows.

Definition 4.5 (Property Language $\mathcal{L}$)**.** *Let AP be a set of graph constraints, the atomic properties of $\mathcal{L}$, then we can define $\mathcal{L}$ recursively as:*

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

The symbols $\Box$ and $\Diamond$ stand for globally and finally, respectively. A property ϕ that is prefixed with $\Box$ has to be hold in each state of the trace, whereas a for a property being prefixed with $\Diamond$has is satisfied if one of the trace's states fulfills it. 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). The above definition differs from the standard definition of linear temporal logic [114] by the absence of the next-operator. We omitted the next-operator, as we will require the capability that the specified properties are stutter invariant (cf. [112]).

The properties specified in $\mathcal{L}$ are evaluated over traces of HGTS.

Definition 4.6 (Satisfaction of $\mathcal{L}$)**.** *Given a trace t and a property $\phi \in \mathcal{L}$, t satisfies ϕ if the following recursive definition applies:*

$t \models \phi$ *with* $\phi \in AP$ *iff* $G_0 \lesssim \phi$

$t \models \phi \wedge \psi$ *with* $\phi, \psi \in \mathcal{L}$ *iff* $t \models \phi \ \wedge\ t \models \psi$

$t \models \neg\phi$ *with* $\phi \in \mathcal{L}$ *iff* $t \not\models \phi$

$t \models \Diamond\phi$ *with* $\phi \in \mathcal{L}$ *iff* $\exists i \in \mathbb{N} : i \geq 0 \wedge t_i \models \phi$

$t \models \Box\phi$ *with* $\phi \in \mathcal{L}$ *iff* $\forall i \in \mathbb{N} : t_i \models \phi$

$t \models \phi \mathcal{U} \psi$ *with* $\phi, \psi \in \mathcal{L}$ *iff* $\exists i \in \mathbb{N} \wedge \forall j \in \mathbb{N} : 0 \leq j < i \implies t_j \models \phi \wedge t_i \models \psi$

For S being a set of traces, we define that $S \models \phi$ iff $s \models \phi$ holds for all $s \in S$.

Although we use traces to evaluate the properties of $\mathcal{L}$ the origin of the traces are HGTS (see Definition B.13). Therefore it is reasonable to require that, if a property ϕ is to be evaluated for a set of traces, the property ϕ has to hold for each trace.

Example 2: Let us consider the trace t, we have introduced in Example 1. For the property $\Box\phi$ with ϕ being the `notTwoContracts` property (see Figure 4.3) it can easily seen that, $t \models \Box\phi$. This is due to the fact that we cannot find any i with $0 \leq i \leq 3$ such that the state G_i violates the `notTwoContracts` property, i.e. contains two Contract instances between the Supplier and Customer roles.

4.1.3 Abstract Operators

The introduction of a semantics for collaborations, components and systems further allows us to introduce abstract operators, which we will need to define the verification scheme for *rigSoaML*. In the following we will define two operators. One to describe, that a set of rules satisfies a safety property and one to describe that a rule-set refine another one. We call these two operators abstract as we will not give any arguments why the desired properties actually hold.

Models Operator The models operator $\models$ is defined of the satisfaction of a property $\phi \in \mathcal{L}$ as we have introduced it in Section 4.1.2.

Refinement Operator For two rule-sets R, S and a property $\phi \in \mathcal{L}$ the refinement operator $\sqsubseteq_A$is specified as:

$$R \models \phi \wedge S \sqsubseteq_A R \implies S \models \phi$$

4.1.4 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 6.21

Corollary 4.7. *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 Chapter 6. The idea behind pseudo-typing is that all elements that belong to a component or a collaboration instance only are connected to elements that belong to the same component or collaboration, respectively. This ensures that all concrete collaborations/components are separated and only be linked through collaborations used by ports of components.

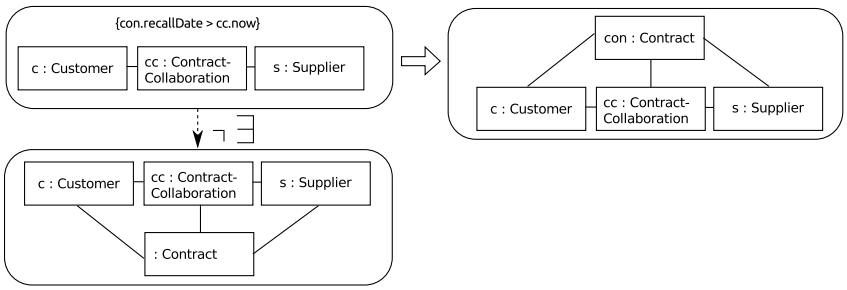


Figure 4.2: The graph-transformation-rule for the `createContract` rule

4.1.5 Employed Notations for GTS

We will employ *Class diagrams* to specify the structure of components, collaborations and systems. Thus the set of lables that are available for typing nodes and edges in graphs can be directly derived from the modelled UML-class diagrams.

We use *StoryPatterns* [95] as a modeling notation for the behavior specification. 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≫ (in diagrams often abbreviated to "++" and "–") 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 can be directly mapped to graph transformation rules.

Example 3: To illustrate the StoryPattern notation have a look at Figure 4.4. 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.

The corresponding graph-transformation rule is shown in Figure 4.2. In the top-left part of the rule the rule's left-hand-side is shown. Below the LHS the rule's negative application condition is depicted and the result of the rule application is right of the big arrow. The morphism between the LHS and the RHS is indicated by nodes having the same name. The same holds for the LHS and the NAC. The procedure to translate a Stroy Pattern to a graph transformation rule is as follows:

Nodes and edges having no ++ or −− marker attached to them are part of the graph transformation rule's LHS and RHS. Nodes and edge having a ++ marker are part of the rule's RHS only and those having a −− marker belong to the rule's LHS. For each dashed box and crossed out box we add a NAC to the rule's LHS containing the elements of the LHS and additionally the elements of the dashed box.

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

4.2 Modelling Concepts

In chapter 3 we have seen that *SoaML* only supports the requirement Modelling SOA (M1). The service contract RequestOfferCollaboration could be seen as a specialised type of an abstract ContractCollaboration service contract. Service contracts in *SoaML* subtype the UML concept of collaborations and thus they also support inheritance. Unfortunately, the UML only defines the specialisation of collaborations at the role level. *SoaML* reuses this definition but fails to clearly define what inheritance means to the behaviour of the specialised service contract. Further, *SoaML* does not contain any concepts to model the fact that participants join or leave a running service contract. If the modelled system needs to be verified, the developer needs some guidance that makes clear what kind of specification is required in which modelling stage. *SoaML* lacks this guidance, which is obvious as *SoaML* is a multi-purpose modelling language. Lastly, the notion of structural changes, which occur naturally occur in a service-oriented system, cannot easily 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. Formally, we can define a service role as follows:

Definition 4.8. *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 behaviour* $R(Ro^i) \neq \emptyset$ *and otherwise* abstract*, where* $R(Ro^i)$ *is a function returning the role's rules. It is further* refined *if role types exist that are subtypes.*

An 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 4.1.4) by linking all nodes occurring in the rule to ro^i.

Example 4:[Concrete Role] In the supply chain application example, two *concrete* role types exist. For the role types Customer and Supplier,

a rudimentary behaviour is specified. These two role types get *refined* through further role types that will be introduced within the supply chain example.

Refining Role Types

The refinement or subtyping of a role requires that the resulting behaviour is a refinement.

Definition 4.9. *For concrete role types Ro_i and Ro_j, it holds that Ro_i* refines *Ro_j (written $Ro_i \sqsubseteq_{Ro} Ro_j$) iff*

$$R(Ro_i) \sqsubseteq_A R(Ro_j)$$

holds,

4.2.2 Collaboration Types

rigSoaML uses collaborations to specify the different service contracts that are available in a service-oriented system. However, the basic notation as UML collaborations, which is used in *SoaML*, is not sufficient for our purposes, as we will need more information for a collaboration to be specified. For example, we will have to specify safety properties for each collaboration type. *SoaML* does not directly support this.

Definition 4.10. *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, a 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 safety properties $\Phi_i \in \mathcal{L}$. Either all roles are concrete (having an assigned concrete behaviour) or all roles are abstract. The collaboration type is* concrete *if all roles are concrete (having an assigned concrete behaviour) and otherwise* abstract. *It is further* refined *if collaboration types exist that are subtypes.*

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-type linking of all nodes to col_i. 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 behaviour $R_i(ro_i^k)$. The collaboration type Col_i's property has 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 that is a subtype of collaboration Col_j is allowed to enhance the class diagram CD_j with its own types. Obviously, this is required because otherwise collaboration subtypes have no possibility of using the super- collaboration's roles and data-types. Nevertheless, the new elements have to be defined in a separated and unique name-space.

The rules for creation of new role-instances and the connection of role-instances with collaboration-instances are part of the roles' behaviour. 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 another 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 also allowed. 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. by setting a mode-flag of role B.

The collaboration's choreography will be modelled through StoryPatterns. In Figure 4.4 an example of a StoryPattern is depicted. The StoryPattern is applicable if both roles Supplier and Customer could be matched and they were connected by a ContractCollaboration instance. The result of the StoryPattern is that a new Contract instance is created, whose attribute recallDate has to be greater than the ContractCollaboration's now attribute.

For the modelling 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 for StoryPatterns for properties to 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. a UML object diagram — satisfies a StoryPattern if 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 $\Box\neg P$, 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 of a forbidden pattern is shown in Figure 4.3. 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 "$\Box\neg$" and is hence called a forbidden pattern.

Example 5:[Concrete Collaboration] An example of a concrete collaboration is the ContractCollaboration, whose structural diagram is depicted in Figure 3.2. The ContractCollaboration is specified as

$$Col_{CC} = (ContractCollaboration, (Customer_{CC}, Supplier_{CC}), CD_{CC}, R_{CC}, \Phi_{CC})$$

The Customer role can create a Contract between the Customer and the Supplier role (see Figure 4.4). The collaboration network does not have any assigned behaviour. The collaboration's properties specify that no two Contracts exist between the Customer and the Supplier role. This is formally expressed as $\Phi_{CC} = \Box\neg\exists$`twoContracts`, with `twoContracts`

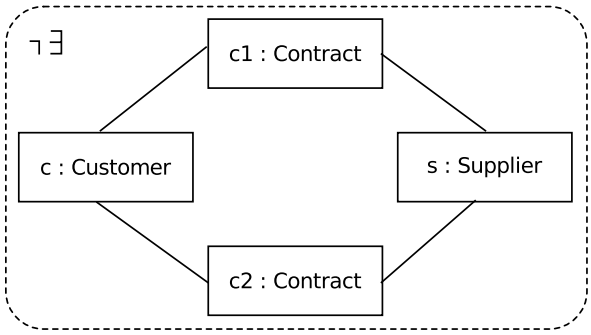


Figure 4.3: Property: Not two Contracts between Customer and Supplier role.

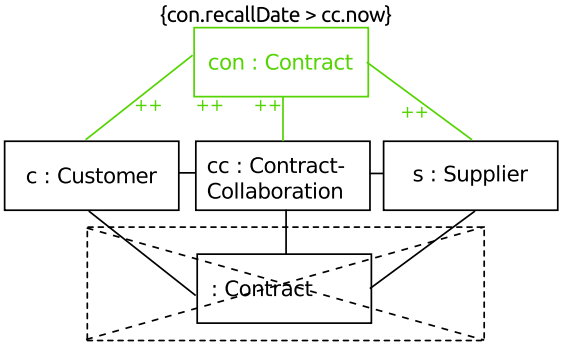


Figure 4.4: The ContractCollaboration's createContract rule

being the graph constraint depicted in Figure 4.3.

$$R_{CC}(\text{Customer}_{\text{CC}}) = \{\texttt{createContract}, \texttt{deleteContract}, \texttt{destroyCollab}\}$$
$$R_{CC}(\text{Supplier}_{\text{CC}}) = \{\texttt{deleteContract}, \texttt{destroyCollab}\}$$

The rules for the collaboration's Customer role can be used to exemplify pseudo-typing, as introduced in Section 4.1.4. The role's CreateContract rule (cf. Figure 4.4) 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 ContractCollaboration.

In Section A.2.1 we depict all the rules and graph constraints for the ContractCollaboration.

Refining Collaboration Types

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

Definition 4.11. *For an abstract or concrete collaboration type* $Col_i = (col_i, \{ro_i^1, \ldots, ro_i^{n_i}\}, CD_i, R_i, \Phi_i)$ *and an abstract collaboration type* $Col_j = (col_j,$

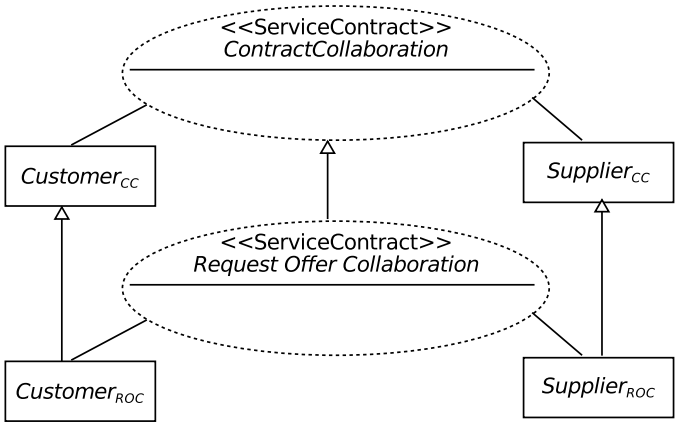


Figure 4.5: RequestOfferCollaboration Structure

$\{ro_j^1, \ldots, ro_j^{n_j}\}, CD_j, R_j, \Phi_j)$*, it holds that* Col_i refines Col_j *(written* $Col_i \sqsubseteq_{Col} Col_j$*) iff* $n_j \leq n_i$ *holds and the refinement results in stronger properties*

$$\Phi_i \implies \Phi_j$$

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

The refinement of collaboration types has some implications concerning the compatibility of the refined roles. The intuitive understanding is that wherever a role R is required, it is possible to use the refined role R'. The refinement is assumed to be covariant. The refinement of collaboration types, however, is not covariant. To explain this, let us consider the two collaboration types ContractCollaboration and RequestOfferCollaboration. The RequestOfferCollaboration introduces new refined variants of the roles Supplier and Customer, named Supplier$_{\mathsf{ROC}}$ and Customer$_{\mathsf{ROC}}$. Of course, it is impossible to bind the Supplier role to a RequestOfferCollaboration. The role does not know the RequestOfferCollaboration's protocol and thus could arbitrarily create a Contract without awaiting the Offer being sent. This is still the expected behaviour, as the RequestOfferCollaboration requires a role of type Supplier$_{\mathsf{ROC}}$. The role Supplier is of the wrong type. In the reverse situation — i.e. using the Supplier$_{\mathsf{ROC}}$ role in the ContractCollaboration — it cannot be assumed that the refined role behaves as desired. In the concrete example given, the role expects that, before a Contract can be created, a Offer has to be made.

Nevertheless, a role can be used in more than one collaboration type without changing it. The collaboration type's other roles then have to be compatible. Summarising, with regard to the roles collaboration type, refinement is neither covariant nor contravariant, but is invariant. A substitution of roles by their super- or sub-types is not supported.

Example 6:[Collaboration Refinement] The concrete ContractCollaboration introduced in the previous example is refined by the *concrete collab-*

oration RequestOfferCollaboration (ROC)

$$Col_{ROC} = (RequestOfferCollaboration, (Customer_{ROC}, Supplier_{ROC}), CD_{ROC}, R_{ROC}, \Phi_{ROC})$$

The ROC's structure is depicted in Figure 4.5. The concrete role types $\mathsf{Customer}_{\mathsf{ROC}}$ and $\mathsf{Supplier}_{\mathsf{ROC}}$ refine the role types $\mathsf{Customer}_{\mathsf{CC}}$ and $\mathsf{Supplier}_{\mathsf{CC}}$, respectively. The roles' behaviour is specified through two sets of StoryPatterns, which allow the $\mathsf{Customer}_{\mathsf{ROC}}$ to send a Request to the $\mathsf{Supplier}_{\mathsf{ROC}}$, 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 4.10. For the collaboration's two roles Customer and Supplier we get the following assignment of rules:

$$R_{ROC}(\text{Customer}_{\text{ROC}}) = \{\texttt{sendRequest}, \texttt{createContract}, \texttt{deleteContract}, \texttt{destroyCollab}\}$$
$$R_{ROC}(\text{Supplier}_{\text{ROC}}) = \{\texttt{sendOffer}, \texttt{deleteContract}, \texttt{destroyCollab}\}$$

Some of the rules describing the Customer role's behaviour are shown in Figures 4.8 and 4.10. The Supplier role's behaviour is depicted in Figure 4.9.

In comparison with the ContractCollaboration introduced in Example 5, the collaboration's properties have been extended by three more graph constraints, depicted in Figure 4.7. We could thus write the RequestOfferCollaboration's properties Φ_{ROC} as $\Phi_{ROC} \equiv \Phi_{CC} \wedge \Phi'_{ROC}$, with Φ'_{ROC} being the additional forbidden patterns. Hence, the required implication $\Phi_{ROC} \implies that\Phi_{CC}$ holds.

The collaboration type's class diagram is depicted in Figure 4.6.

A complete set of rules and graph constraints for the RequestOfferCollaboration is shown in Section A.2.3.

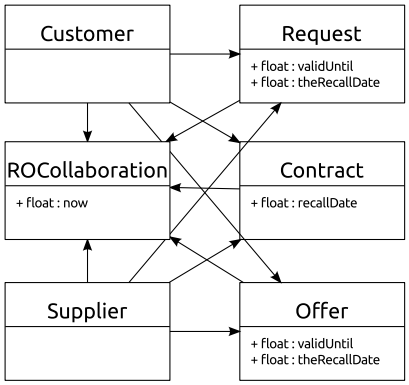
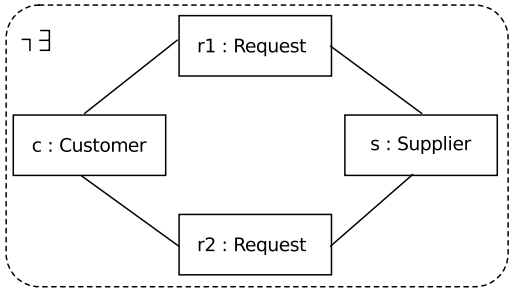
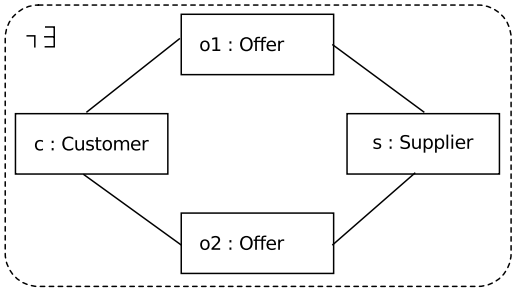


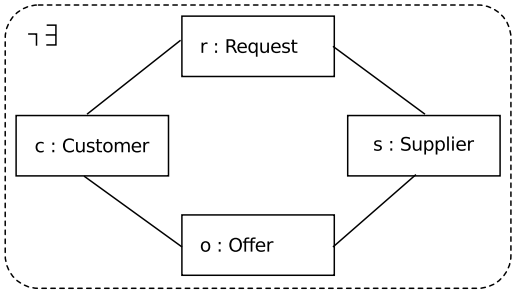
Figure 4.6: Class Diagram CD_{ROC} for the RequestOfferCollaboration



(a) `noTwoRequests`



(b) `noTwoOffers`



(c) `noOfandReq`

Figure 4.7: Properties Φ_{ROC} for the RequestOfferCollaboration

Semantics of Collaborations

We will give the semantics of a collaboration C through the set of all possible traces that conform with the collaboration's properties and rules. We differentiate the definition of the semantics between concrete and abstract collaboration types. Remember that a concrete collaboration type has a fully specified operational behaviour – i.e. all roles have at least a rudimentary set of rules – whereas abstract collaboration types are still lacking their final behaviour specification. The semantic domain that we have chosen for collaborations, components and systems are sets of traces. In the following, we will define the semantics of collaborations, components and systems and relate the dependencies in the model to refinement relations among the corresponding sets of traces.

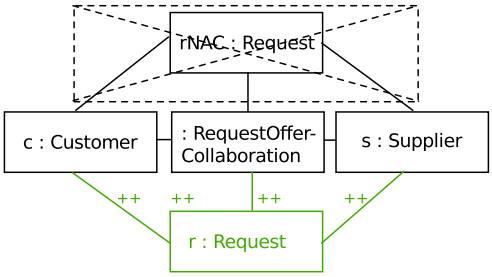


Figure 4.8: `createRequest` Rule

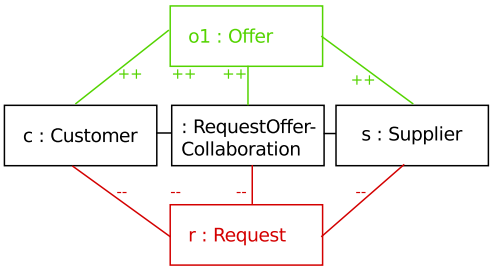


Figure 4.9: `makeOffer` Rule

Definition 4.12 (Semantics of Collaborations)**.** *Given a concrete collaboration* $Col_i = (col_i, (ro_i^1, \ldots, ro_i^{n_i}), CD_i, R_i, \Phi_i)$ *the collaboration's semantics* $[\![Col_i]\!]$ *is given as*

$$[\![Col_i]\!] = T(G_\emptyset, R)$$

where R *is given as* $R = R_{Col_i}(ro_i^1) \cup \ldots \cup R_{Col_i}(ro_i^{n_i})$.

For an abstract collaboration $Col_a = (col_a, (ro_a^1, \ldots, ro_a^{n_a}), CD_a, R_a, \Phi_a)$, *the collaboration's semantics* $[\![Col_a]\!]$ *is given as*

$$[\![Col_a]\!] = \{t | t \in T(CD_a) \wedge t \models \Phi_a\}$$

where $T(CD_a)$ *is the set of all traces that can be built from graphs that are typed over* CD_a.

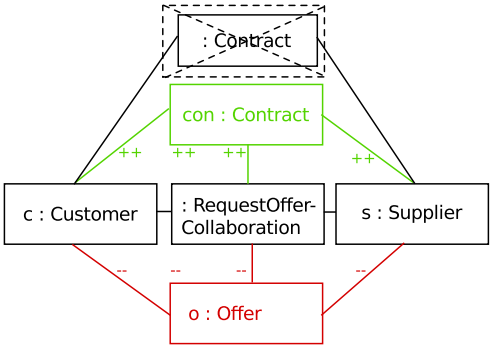


Figure 4.10: createContract Rule

Mapping to *SoaML*

A collaboration type that has been modelled in *rigSoaML* as defined above, can also be partially expressed in terms of *SoaML*. The corresponding concept in *SoaML* is the ≪ServicesArchitecture≫ collaboration diagram. For our example collaboration type, RequestOfferCollaboration (cf. Example 6) can be mapped to a ≪ServicesArchitecture≫ as depicted in Figure 3.3.

The mapping to *SoaML* does not preserve all information that is available in our *rigSoaML* collaboration types. The collaboration type's safety properties have to be added to the ≪ServicesArchitecture≫ collaboration diagram using OCL constraints.[2] For the behaviour, we can use any UML behaviour. For our example collaboration type ROC, we could use a sequence diagram to specify a possible application of the rules. The resulting diagram is depicted in Figure 3.5. However, a sequence diagram is only a partial specification of the behaviour. The problem in general is that the available UML behaviour specifications, i.e. activity diagrams, state machines and sequence diagrams, cannot directly express the structural preconditions and changes that can be specified using StoryPatterns.

4.2.3 Component Types

In conformance with *SoaML*, *rigSoaML* employs components to implement the collaborations' roles. *SoaML*, however, only supports 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 4.13. *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 $I_i \subseteq R_i(com_i)$, and properties $\Psi_i \in \mathcal{L}$. The component assigns a concrete behaviour $R_i(ro_i^j) \neq \emptyset$ for $1 \leq j \leq m_i$ either to each of its roles or to none of them $R_i(ro_i^j) = \emptyset$. In the first case, the component is considered concrete*, in the second* abstract*. A concrete component type must only implement roles of concrete collaboration types. 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 fulfils the pseudo-typing requirements and thus separates elements that belong to different component instances from each other. All

[2] As we also allow specification of temporal properties, the expressiveness of OCL might not be strong enough (cf. [136])

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 4.10). 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.[3] 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.

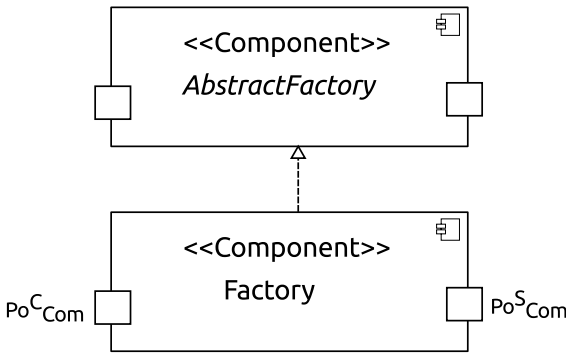


Figure 4.11: Structural overview of the Factory component

Example 7:[Factory component] The two role types $\mathsf{Customer}_{\mathsf{ROC}}$ and $\mathsf{Supplier}_{\mathsf{ROC}}$ are both implemented by a single component Factory (cf. Figure 4.11). The Factory component is formally given as: $Com_{Fac} = \big(com_{Fac}, (Customer_{Fac}^{ROC}, Supplier_{Fac}^{ROC}), CD_{Fac}, I_{Fac}, \Psi_{Fac}\big)$. The component's behaviour is again specified through a set of rules, which are depicted in Figure 4.13. We have the following assignment of the component's rules to its roles:

$$R_{Fac}(Customer_{Fac}^{ROC}, Com_{Fac}) = \{\texttt{createRequest}, \texttt{createContract}\}$$
$$R_{Fac}(Supplier_{Fac}^{ROC}, Com_{Fac}) = \{\texttt{makeOffer}\}$$

The Factory's class diagram CD_{Fac} is depicted in Figure 4.12.

In contrast to the collaboration's rules (see Figure 4.10), the component's rules clearly allow us 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.

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

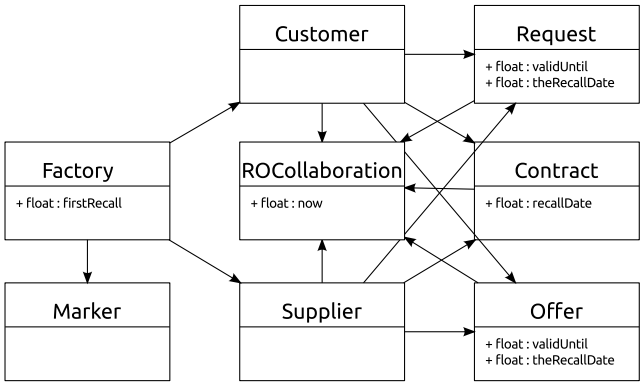


Figure 4.12: Class Diagram CD_{Fac} for the Factory component type

Figure 4.11 shows that the Factory component extends the AbstrFactory component, which is specified as abstract. However, for the AbstrFactory component a property is specified: $\Psi_{AFac} = \Box\neg\texttt{earlyRecall} \wedge \Box\neg\texttt{custNoCon}$. The corresponding graph constraints are depicted in Figure 4.14(a) and 4.14(b), respectively. The abstract component's complete specification can be given as:

$$Com_{AFac} = (com_{AFac}, (Customer_{AFac}, Supplier_{AFac}), CD_{AFac}, R_{AFac}, \Psi_{AFac})$$

with $R(Supplier_{AFac}) = R(Customer_{AFac}) = R(com_{AFac}) = \emptyset$.

Refining Component Types

The refinement or subtyping of components also requires that the resulting behaviour is a refinement.

Definition 4.14. *For a component type*

$$Com_i = (com_i, (ro_i^1, \ldots, ro_i^{m_i}), CD_i, R_i, I_i, \Psi_i)$$

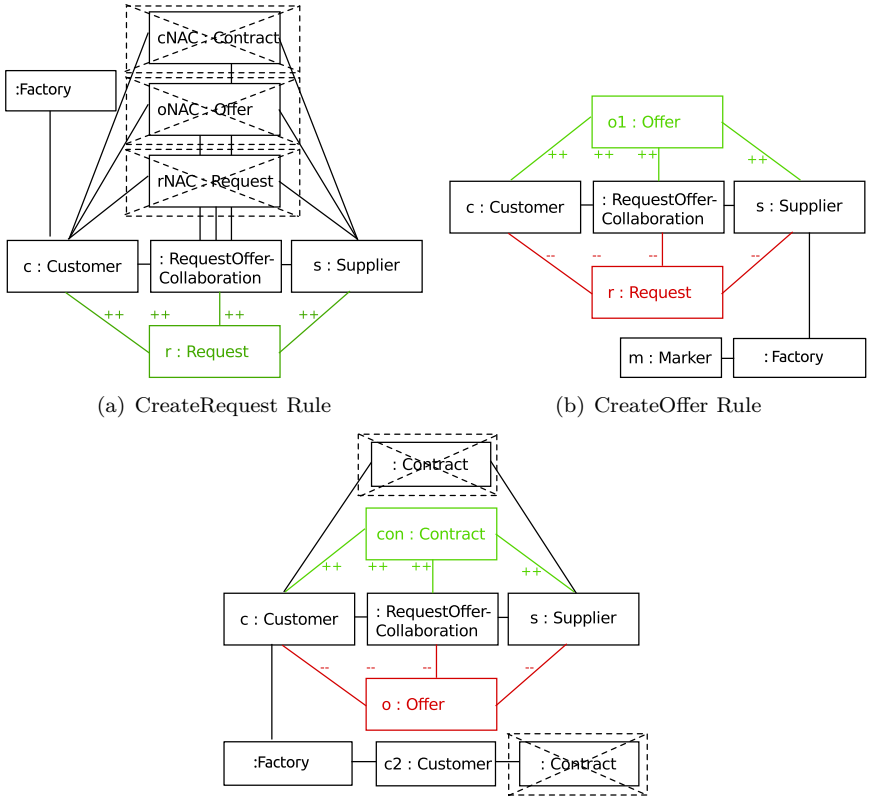
and an abstract component type $Com_j = (com_j, (ro_j^1, \ldots, ro_j^{m_j}), CD_j, I_j, \Psi_j)$ *it holds that* Com_i refines Com_j *(written* $Com_i \sqsubseteq_{Com} Com_j$*) if it holds that* $m_j \leq m_i$ *and the refinement results in stronger properties*

$$\Psi_i \Rightarrow \Psi_j$$

For all Com_i *and its super-type* Com_j *it holds that the subtype relation is* correct *if* $Com_i \sqsubseteq_{Com} Com_j$.

Semantics of Components

While collaboration types are more important during design time, component types are the system constituents that are actually present in the final system and executed. Thus, it is important to also give a semantic definition for



(a) CreateRequest Rule

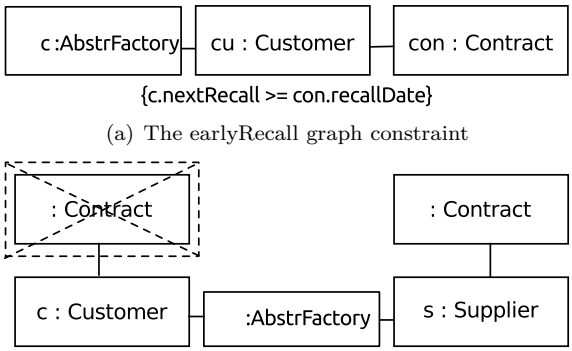
(b) CreateOffer Rule

(c) CreateContract Rule

Figure 4.13: Behavioural rules for the Factory's roles

component types. The semantics of a component type cannot be given without taking into account the collaborations that a component can participate in. If we were only to consider the component's rules for the definition of the component's semantics, we would miss important aspects of the component's behaviour. Therefore, we also add the rules of the collaborations belonging to the component to the definition of the component's semantics. This allows us to achieve a closed set of rules.

In Definition 4.13 we have defined a component type as being abstract if it does not specify its own behaviour for the implemented roles. However, this does not necessarily imply that no behaviour for the component has been specified yet, as the collaboration types can be concrete and hence have a complete specified behaviour. A concrete component type, however, only implements roles of concrete collaboration types. For the definition of the semantics, we thus have to consider these situations.



(a) The earlyRecall graph constraint

(b) The custNoCon graph constraint

Figure 4.14: Properties for AbstrFactory component

Definition 4.15 (Semantics of Components)**.** *The semantics of a concrete component* $Com_i = (com_i, \left(ro_i^1, \ldots, ro_i^{m_i}\right), CD_i, R_i, I_i, \psi_i)$ *is defined as*

$$[\![C]\!] = \left\{ t | t \in T\left(G_\emptyset, \hat{P}\right) \right\}$$

, where $\hat{P}$ *is given as:*

$$\hat{P} = R(com_i) \bigcup_{col \in Cols} R(col) \setminus R(cr(col))$$

where $Cols$ *is a set of collaboration types* $Cols = \{col_1, \ldots, col_k\}$ *with* $k \leq m_i$ *and* $cr : Cols \mapsto 2^{\left\{ro_i^1, \ldots, ro_i^{m_i}\right\}}$ *is a mapping of roles to the collaboration types that defined the roles, such that for all* $s, t \leq k$ *and* $s \neq t$ *it holds that* $cr(col_s) \cap cr(col_t) = \emptyset$ *and* $\bigcup_{col \in Cols} cr(col) = \left\{ro_i^1, \ldots, ro_i^{m_i}\right\}$.

The semantics of an abstract component type $Com_a = (com_a, \left(ro_a^1, \ldots, ro_a^{m_a}\right), CD_a, R_a, I_a, \psi_a)$ *is defined as*

$$\begin{aligned} [\![Com_a]\!] = \{ t | t \models \psi_a \wedge \\ \forall Col_a \in Cols_A \quad t \models \phi_{Col_a} \wedge \\ \forall Col_c \in Cols \setminus Cols_A \quad t_{|CD_{Col_c}} \in T(R_{Col_c}) \} \end{aligned}$$

In the above definition, we use a few auxiliary constructs such as the set $Cols$ and the mapping function cr. $Cols$ is the set of collaboration types that the component type com_i can participate in, as it has at least one of the collaboration type's roles, with $Cols_A \subseteq Cols$ being the subset of abstract collaboration types. Of course, a component type can implement more than one role from a specific collaboration type and thus the size of $Cols$ is potentially smaller than the size of the collaboration type's roles. The mapping function cr is used to express the connection between the roles and the defining collaboration types.

For an abstract component type, the semantics is given as all the traces that satisfy the component type's properties ψ_a and that additionally satisfy the abstract collaboration types' properties, and for all concrete collaboration types

the trace has to be of such a form that the parts relevant to the concrete collaboration type could be created using the concrete collaboration type's rules[4].

Mapping to *SoaML*

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 into 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 behaviour have to be translated into a UML behaviour specification. As for collaboration types, we do not have a direct representation of the safety properties.

4.2.4 System Types

The *rigSoaML* counterpart to *SoaML*'s service landscapes are system types and systems. Systems combine collaboration- and component-types into a conceptual unit. However, the

Definition 4.16. *A* system type $Sys = ((Col_1, \ldots, Col_n), (Com_1, \ldots, Com_m))$ *consists of a number of collaboration types* Col_i *and a number of component types* Com_j*.*

> **Example 8:** Using the previous examples 7 and 6, we can define a first system type $Sys_1 = ((Col_{ROC}), (Com_{Fac}))$.

The instances of a system type are called systems and consist of collaboration and component instances.

Definition 4.17. *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* Sys*. A system is* concrete *if all collaborations* $Col_1, \ldots, Col_n$ *and components* $Com_1, \ldots, Com_m$ *are concrete.*

Semantics of Systems

Systems consist of components that are connected over roles and collaborations. However, at the system level collaborations are not visible, as they are solely a theoretical concept to abstract and encapsulate a desired behaviour. Further, a system can only consist of concrete components, as abstract components cannot be instantiated — they are not yet fully specified. However, we will use the power of abstraction that a collaboration gives us to show the correctness of systems. But before being able to do this, we have to define the semantics of systems.

If we think of systems as conglomerates of components, immediately the question of concurrency and how concurrency is handled in the formal model arises. To

[4]For details on the trace restriction $t_{|CD_{Col_c}}$, refer to Definition 6.2.

be honest, this problem is not specific to systems but also has to be considered for collaborations. For collaborations we assumed, and defined it accordingly, that no two rules can be applied in parallel. However, as it is further specified that the application of rules does not consume time, the continuous behaviour of our systems should not be influenced, and especially not if the two components executing the two rules are completely separated from each other. If two rule applications are not independent of each other — i.e. the order of the rule application influences the outcome — the developer has to make sure either that only one order could occur or that both orders are safe. Thus, we can define the semantics of systems as the union of the components' rule-sets included in the system.

Definition 4.18 (Semantics of systems)**.** *The semantics of a system* $Sys = ((Col_1, \ldots, Col_n), (Com_1, \ldots, Com_m))$ *is defined as follows:*

$$[\![Sys]\!] = \{t | t \in T(G_\emptyset, R_{Sys})\}$$

where $R_{Sys} = \bigcup_{1 \leq i \leq m} R(Com_i)$ *is the combined rule-set of the system's components.*

Type Conformance

An important property of a system type and the corresponding systems is type conformance. A system type is then type conform if the collaboration and component types are consistently referring to each other.

Definition 4.19. *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, a collaboration defining that role also exists and all subtype relations of collaborations and components are type conform, i.e. all components are only used in collaborations that offer exactly the concrete role that is implemented by the component.*

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 has only defined exclusively once, only used in subtype collaborations or components such that they are pseudo-type separated there.

An obvious property of type conform systems is that the application of type conform rules does not invalidate the systems' type conformance. This property is inherited directly from typed graph- transformation systems, which have exactly this property. Thus, in our modelling approach it is impossible for a collaboration type instance to connect with a role which it doesn't know. Hence, type conformance is guaranteed for any system configuration if the system type is type conform.

Mapping to *SoaML*

A system type as defined above can be expressed in terms of *SoaML* by a ≪ServicesArchitecture≫ collaboration diagram that comprises all collaboration

types that occur in the system. In order also to give an overview of the available component types, a comprehensive set of ≪ParticipantArchitecture≫ component diagrams has to be given too.

4.2.5 System Evolution

One aspect of our motivation for this work is that system types are subject to change and thus the software engineering and verification methodologies have to be aware of these changes. We will use the term system evolution to describe the way that a system type changes. The reasons for which such changes happen are manifold, but the way the change looks can be roughly categorised into the following cases.

Top-level changes Changes to the type system can be made at the topmost level. Mostly the types that are added at this level are abstract types that do not necessarily provide any new behaviour, 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. E.g. , 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 participates in the running system it is likely that they provide their own component or collaboration type, depending on their requirements.

SoaML does not contain any suitable equivalent to express system evolution. Nevertheless, it could be added to *SoaML*, as this is what we have done in this work. In Figure 4.15 we show an exemplary evolution diagram. The diagram can be horizontally split into two main parts, the abstract specification at the top and the implementation part below. The implementation part, however, can further be vertically divided into "swim-lanes", one for each organisation that provides an implementation for any of the abstract concepts. The vertical order of the entities in the implementation part indicates a partial order of when they have been introduced. Note that again a total order cannot be given, due to the SoS' open and distributed nature. To support better discernibility, the evolution diagram separates different versions with alternating background colours (light grey and white). In the given example diagram, the abstract specification level consists of the known collaboration type ContractCollaboration, its two roles Customer and Supplier, and the abstract AbstrFactory component type. "Organisation A" publishes a specialisation of the ContractCollaboration the RequestOfferCollaboration collaboration type (in the figure we used the abbreviation ROC) as well as the two refined roles (abbreviated to Cust and Sup). The concrete component type Factory has been introduced by "Organisation B". "Organisation C" introduces some differently refined roles for Customer and Supplier, that are not further considered here. The diagram does not allow us to infer the time order in which a concept has been introduced. The diagram only states that "Organisation A" introduced the RequestOfferCollabo-

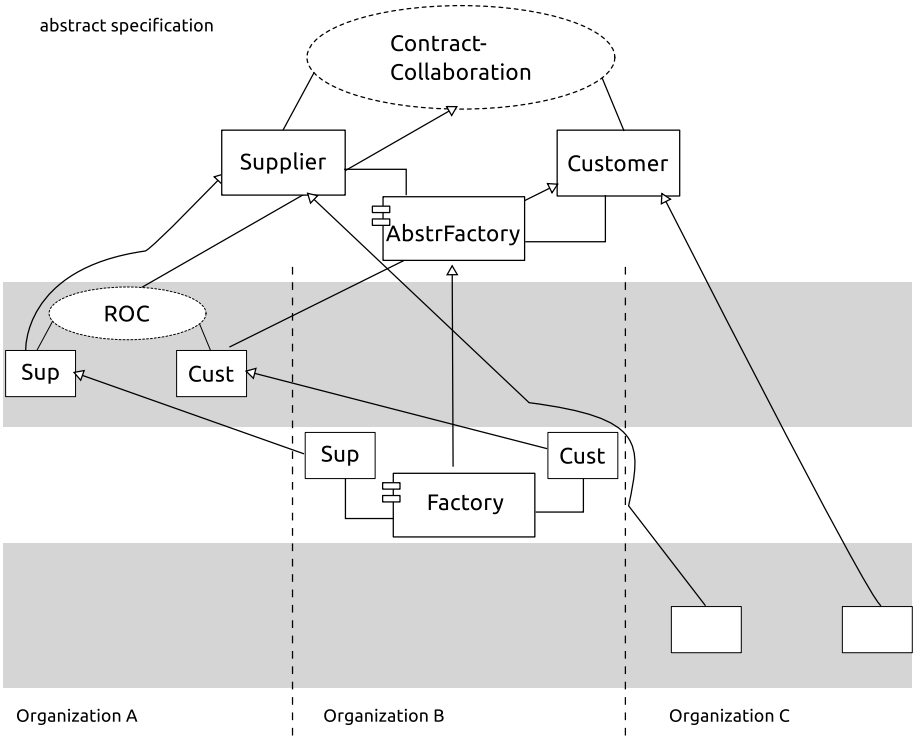


Figure 4.15: Evolution Diagram for an abstract specification and three independent organisations developing implementations

ration before "Organisation B" introduced the Factory component. Whether or not "Organisation C" introduced the two roles before "Organisation A", before "Organisation B" or last of all cannot be decided based on the diagram.

Formally, we define system evolution as follows:

Definition 4.20. *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.*

4.3 Discussion

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

Requirement Modelling SOA (M1) is, as outlined in Section 3.2, already mostly covered by *SoaML* and thus *rigSoaML* inherits this coverage. However, *rigSoaML*,

in contrast to *SoaML* (see Section 3.2), also covers challenge Modelling Dynamics (M2). *rigSoaML* supports the dynamics of service-oriented systems such as joining/leaving collaboration instances dynamically and adding components dynamically by means of the rules for roles and components.

The evolution of SoS can be modelled by step-wise addition of types to the system type. Thus, also the challenge Modelling Evolution (M3) concerning the modelling of the uncoordinated introduction of new types for service contracts and components at run-time is covered. Further, the newly introduced evolution diagram illustrates the different changes that have occurred to the system during its lifetime.

Requirements	*SoaML*	*rigSoaML*
Modelling SOA (M1)	✓	✓
Modelling Dynamics (M2)	○	✓
Modelling Evolution (M3)	○	✓

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

Further, our approach *rigSoaML* provides the capabilities to specify the system at an abstract level. This is useful to guide the development without the need to provide behaviour specifications. We have used this, e.g. , in the supply chain system application example, when we introduced the AbstrFactory specification (cf. Example 7). The AbstrFactory abstract component type does not have any rules for its two specified roles but a safety property that relates the recall dates of the Contract instances that are available at the component's roles.

In Section 2.5 we presented a list of scenarios that are typical of SoS. The scenarios that describe the introduction of new collaboration and component types are already supported by *SoaML* and hence, also by *rigSoaML*. The creation of new collaboration and component instances, however, is not directly addressed in *SoaML*, but *rigSoaML* does support this through StoryPatterns. The newly introduced evolution diagram allows us to model the evolution steps occurring in an SoS.

Chapter 5

Verification Schemes for *rigSoaML* Models

The problem of verifying the correctness of systems 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 only works for rather small configurations with a fixed upper bound of elements and a fixed number of initially defined element types, while systems may contain many unbounded 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 provides an even lower coverage than formal verification is not sufficient either. Furthermore, due to the dynamic nature of systems, it cannot be assumed that any of the involved organisations has all relevant details of the concrete service implementations at hand to apply formal verification techniques or testing techniques looking at the complete configuration in detail.

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

We will describe the complete approach to verify *rigSoaML* models with the abstract operators that we have introduced in Section 4.1.3. This allows us to decouple the general verification scheme from the concretely used formal techniques.

5.1 Concrete Systems

The simplification used to approach the problem of system 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 by 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 to 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.

5.1.1 Correct Collaboration Types

We start our considerations by defining what we mean by correct types for collaborations and components. A correct collaboration type requires that the resulting behaviour ensures the collaboration type's properties.

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* $G_\emptyset$ *it 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 behaviour of the collaboration.*

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

> **Example 9:** To exemplify a correct collaboration type, we will use the concrete collaboration type RequestOfferCollaboration (ROC), which has been introduced in Example 6. The ROC has two concrete role types $\mathsf{Customer}_{\mathsf{ROC}}$ and $\mathsf{Supplier}_{\mathsf{ROC}}$. The properties the ROC has to fulfil are

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

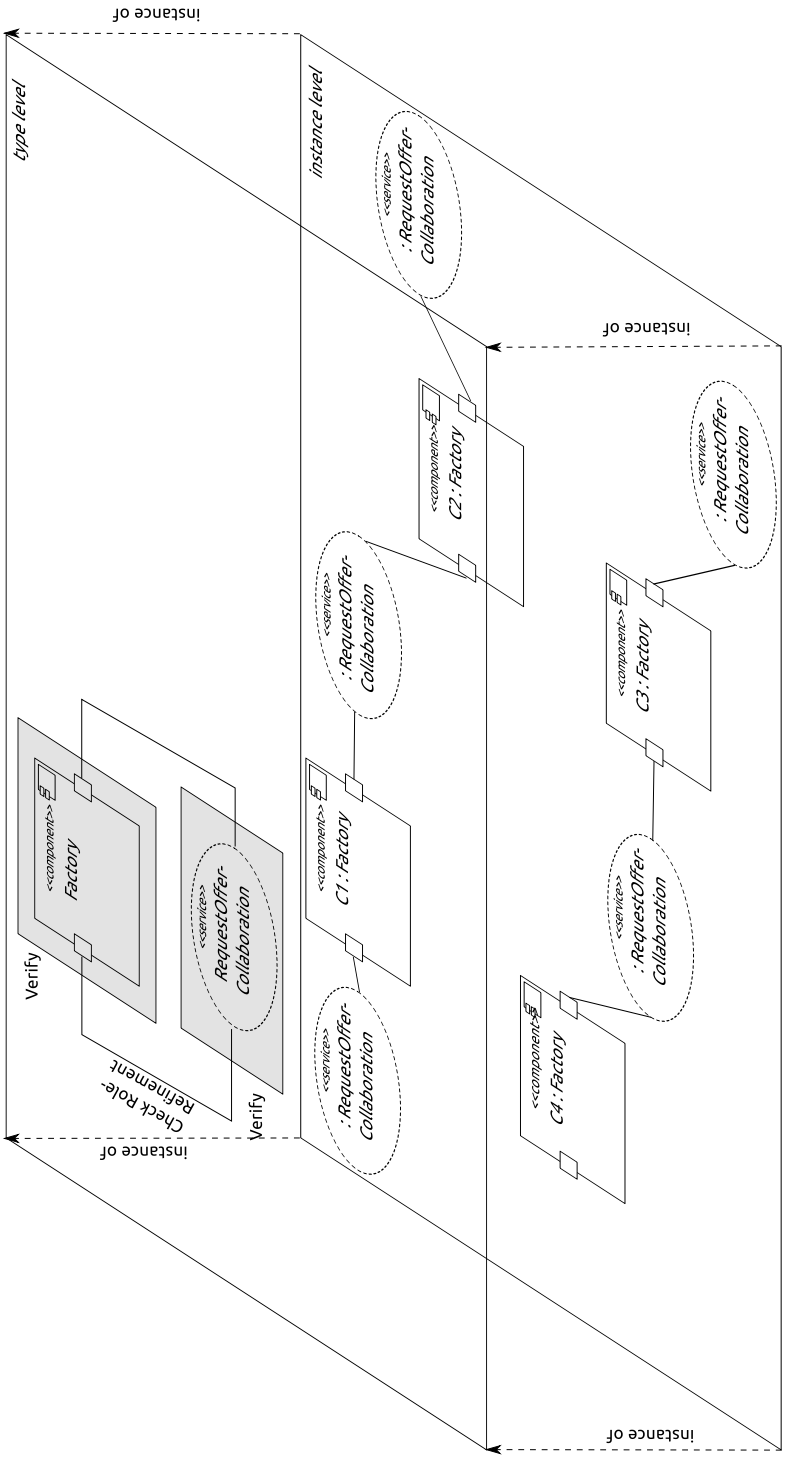


Figure 5.1: Sketch of the general proof scheme

those that are specified for the ROC. Consequently, the property that has to be satisfied by the ROC is

$$\phi_{ROC} \equiv \Box\neg\exists\texttt{noSupplier} \wedge \Box\neg\exists\texttt{earlyRecall} \wedge \Box\neg\exists\texttt{earlyRequest} \wedge \Box\neg\exists\texttt{earlyOffer}$$

. This property has to be satisfied by every graph-transformation system that can be built using empty configuration $G_\emptyset$ as the initial graph and the collaboration's rules.

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

Task	Required
Verify Φ_{CC}	yes
- Check $G_\emptyset, R_{ROC}(Col_{roc}) \models \Phi_{CC}$	yes
Verify Φ_{ROC}	yes
- Check $G_\emptyset, R_{ROC}(Col_{roc}) \models \Phi_{ROC}$	yes

The set of rules $R_{ROC}(Col_{ROC})$ is given as the combination of all rules of the RequestOfferCollaboration's roles:

$$R_{ROC}(Col_{ROC}) = R_{ROC}(\text{Supplier}_{ROC}) \cup R_{ROC}(\text{Customer}_{ROC})$$

5.1.2 Correct Component Types

A correct component type requires that the resulting behaviour ensures that the component type's properties are satisfied and that the component's implementation refines the combined role behaviour.

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* $G_\emptyset$ *it 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 behaviour $R_i(com_i)$ *refines the orthogonally combined role behaviour and creation behaviour*

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

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 behaviour for each role without the role itself, which is covered by* $R_i(com_i)$ *to the component to derive a related closed behaviour. To further differentiate the two elements of correctness, we refer to the first condition as* correct concerning guarantees *and to the second condition as* correct concerning refinement*.*

Example 10: The concrete component Factory can be proved correct. The Factory component owns an implementation and thus can be verified. The Factory has to satisfy the properties Ψ_{Fac}

Task	Required
Verify $G_\emptyset, R_{Fac}(Com_{Fac}) \models \Psi_{Fac}$	Yes
Check role refinement	yes

The rule set $R_{Fac}(Com_{Fac})$ is given as defined in Definition 5.2. $R_{Fac}(Com_{Fac}) = R_{Fac}(\text{Supplier}_{Fac}) \cup R_{Fac}(\text{Customer}_{Fac})$. The correct role refinement is syntactically guaranteed as the rules for the Factory component 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 that*

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

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. For the proof of the lemma, we will use the fact that the union of two pseudo-type separated rule-sets preserves the properties of both rule-sets. For details, see Corollary 6.21.

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. 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 (2), we further know that for all i it holds that $(R(ro_i^1) \cup \cdots \cup R(ro_i^{n_i}) \cup R(col_i)) \models \Phi_i$. Due to Corollary 6.19, 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, as the collaborations are pseudo-type separated, we obtain 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 $I_1 \cup \cdots \cup I_m$ it 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 behaviour $(R(com_i) \sqsubseteq_A (R(ro_i^1) \cup \cdots \cup R(ro_i^{m_i}) \cup I_1 \cup \cdots \cup I_m)$ and thus we can derive the required condition for correctness of the system for all collaborations of Definition 5.3 by substituting $R(com_i)$ 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 6.21, 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 11: To exemplify Lemma 5.4 let us consider the system type Sys, which has been introduced in Example 8. 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 9. 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 $\mathsf{Supplier}_{\mathsf{Com}}$ and $\mathsf{Customer}_{\mathsf{Com}}$ refine the roles $\mathsf{Supplier}_{\mathsf{ROC}}$ and $\mathsf{Customer}_{\mathsf{ROC}}$, respectively.

In summary, 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 was 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 that:*

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

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 $1 \leq i \leq m$, it 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 obtain, due to Corollary 6.19, 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 6.19 are achieved through pseudo-typing and pseudo-types preserving rules. Pseudo-typing guarantees that no two component instances can influence each other directly. □

Example 12: For the system type *Sys* (cf. Example 8), we have only shown so far that the collaboration instances in any system configuration are correct (cf. Example 11). 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 11, the collaboration types are correct (cf. Example 9 and that the component types are correct (cf. Example 10). It follows that 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 that:*

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

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. *Given a system type* $Sys = ((Col_1, \ldots, Col_n), (Com_1, \ldots, Com_m))$ *and a corresponding system* $S = (Sys, G_\emptyset)$*, the system* S *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 two Lemmas 5.4 and 5.6 directly yield the required result, as both require the same or a weaker condition 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 us to straightforwardly establish the required correctness of the types by checking the refinement and the guarantees for the properties using the rule-sets employed in conditions (2) and (3).

Figure 5.1 (see Page 45) visualizes that, according to Theorem 5.8, the required guarantees for the instance level can be established by only doing checks at the type level. Therefore, we can conclude that the complexity of checking the guarantees depends only on the number of types and the complexity of the checking problems of the collaboration and component types.

Example 13: Let us assume that the System $S = (Sys, G_\emptyset)$ is a concrete system of system type Sys (cf. Example 8) and an empty starting configuration $G_\emptyset$. In the previous examples 13, 12, and 11 we have shown that the conditions for the Theorem 5.8 hold. We can thus conclude that the System S is correct.

Task	Required
Verify Φ_{ROC}	yes
- Check $G_\emptyset, R_{ROC}(Col_{roc}) \models \Phi_{ROC}$	yes
Verify $G_\emptyset, R_{Fac}(Com_{Fac}) \models \Psi_{Fac}$	Yes
Check role refinement	yes

5.2 Systems 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 system types the cooperation is usually defined not only using concrete collaborations but also using abstract ones which allow the concrete service contract participants to refine the roles as suits 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 ensure that the required guarantees implied by the abstract collaborations or components are also implied by all concrete collaborations and components refining them.

5.2.1 Correct refining Collaboration Types

For subtyping of collaborations, we can show in the following lemma that the properties of the correctly refined collaboration are preserved by refinement or subtyping.

Lemma 5.9. *For a correct, concrete collaboration type* $Col_i = (col_i, (ro_i^1, \ldots, ro_i^{n_i}), CD_i, R_i, \Phi_i)$ *and any of its abstract collaboration super-types* $Col_j = (col_j, (ro_j^1, \ldots, ro_j^{n_j}), CD_j, R_j, \Phi_j)$ *(*$Col_i \sqsubseteq_{Col} Col_j$*) holds*

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

Proof. For any collaboration type, it holds via induction that its local properties imply the properties of any super-type Col_j ($\Phi_i \implies \Phi_j$). As for a correct, concrete collaboration it holds by definition that $R_i(Col_i) \models \Phi_i$. Consequently, we can conclude $R_i(Col_i) \models \Phi_j$. □

Example 14: The RequestOfferCollaboration (ROC, for specification see Example 6) refines the collaboration ContractCollaboration (Con, cf. Example 5). For the ContractCollaboration we had to run the following checks:

Task	Required
Verify $R(Con) \models \Phi_{Con}$	yes

Obviously, verification of the ContractCollaboration is comparatively easy, as this collaboration only specifies a few properties and does not inherit properties from super-collaborations. The RequestOfferCollaboration, however, inherits from the ContractCollaboration (see Figure 3.2) 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
check refinement	Yes

For the RequestOfferCollaboration it is not required to verify the property Φ_{Com} again, as the collaboration's rules refine the ContractCollaboration's rules.

5.2.2 Correct Refining of Component Types

For subtyping of components, we can show in the following lemma that the guarantees of the refined components are preserved by refinement or subtyping.

Lemma 5.10. *For a correct, concrete component type* $Com_i = (com_i, (ro_i^1, \ldots, ro_i^{m_i}), CD_i, R_i, I_i, \Psi_i)$ *and any of its correct component super-types* $Com_j = (com_j, (ro_j^1, \ldots, ro_j^{m_j}), CD_j, R_j, I_j, \Psi_j)$ *(*$Com_i \sqsubseteq_{Com} Com_j$*) it holds that*

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

Proof. For any component type it holds via induction that its local properties imply the properties of any super-type Com_j ($\Psi_i \implies \Psi_j$). As for a correct, concrete component, it holds by definition that $R_i(Com_i) \models \Psi_i$. Consequently, we can conclude that $R_i(Com_i) \models \Psi_j$. □

Example 15: Let us take a look at the Factory component (cf. Example 7) to exemplify Lemma 5.10. The Factory component's super-type is the abstract component AbstrFactory (cf. Example 7). In order to safely

omit checking that the Factory also satisfies the AbstrFactory'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 4.14). More concretely, we have to show the following:

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

5.2.3 Correct Systems with Abstraction

It now remains to show that the refinement ensures correctness for a System, including the guarantees for the abstract concepts. The following Theorem 5.11 then demonstrates that this correctness criterion is met by construction if all types are correct and the refinement conditions for subtypes are fulfilled.

Theorem 5.11. *A 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 5.9 and Lemma 5.10 further guarantee that also all properties of the abstract collaborations and components are preserved. □

The general proof scheme including abstraction is depicted in Figure 5.2. The sketch differs from the one in Figure 5.1 in that 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 16: Combining the system definition we gave in Example 8 and the examples of correct collaboration and component types, 9 and 10, respectively, we can conclude that the system is correct too. The necessary arguments for Theorem 5.11 are all given in the respective examples.

The necessary steps to verify the complete system can be seen in the following table. Note that Con and ROC are shorthand for the Contract-Collaboration and the RequestOfferCollaboration, respectively.

Task	Required
Verify Φ_{CC}	yes
- Check $G_\emptyset, R_{ROC}(col_{roc}) \models \Phi_{CC}$	yes
Verify Φ_{ROC}	yes
- Check $G_\emptyset, R_{ROC}(col_{roc}) \models \Phi_{ROC}$	yes
Task	Required
Verify $G_\emptyset, R_{Fac}(Com_{Fac}) \models \Psi_{Fac}$	yes
Check role refinement	yes

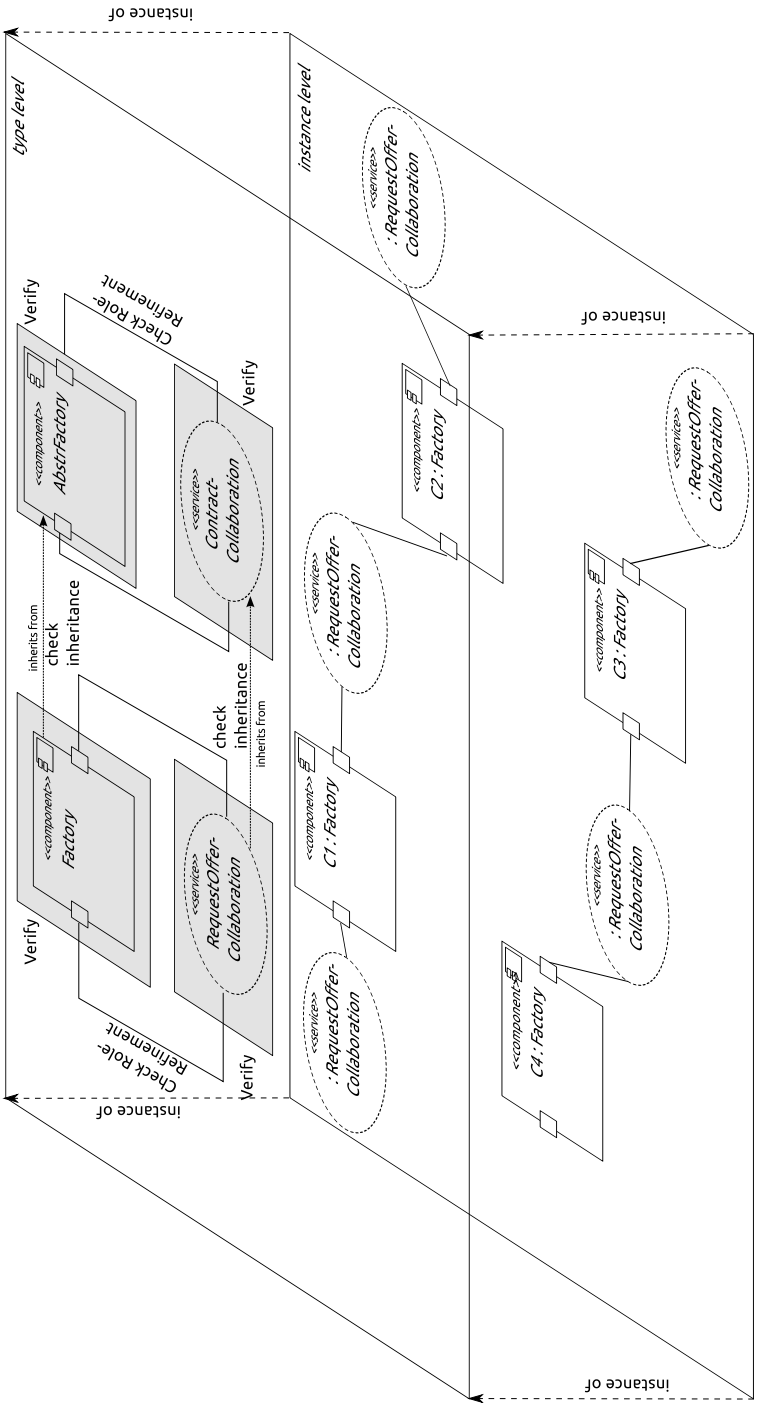


Figure 5.2: Verification scheme for the verification of system types with abstraction

Hence, the compositional capabilities of our approach allow us to reduce the necessary steps for the verification of any system that conforms to the system type to three comparatively simple verification tasks. The verification tasks are independent of the actual system's size, as they work on the component and collaboration types and not the actual component and collaboration instances.

5.3 Evolution

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

If we look at our former results 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 definitions are added over time and where the subtyping tree is thus not necessarily fixed. Furthermore, the different organisations involved will only have a partial view of 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 4.20 we can define correctness as follows:

Definition 5.12. *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 trace* $t_1 \circ \cdots \circ t_n$ *such that* t_i *is a trace in* Sys_i *leading from* G_S^i *to* G_S^{i+1} *for* $i < n$ *and that* t_n *is a trace in* Sys_n *starting from* G_S^n *it holds that*

$$t_1 \circ \cdots \circ t_n \models \Phi_1 \wedge \cdots \wedge \Phi_p \wedge \Psi_1 \wedge \cdots \wedge \Psi_q.$$

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

A first observation is that Sys_n contains all types defined in any Sys_i. However, for a combined trace $t_1 \circ \cdots \circ t_n$ such that t_i is a trace in Sys_i leading from G_S^i to G_S^{i+1} for $i < n$, it does not in general hold that an equal trace t 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 5.13. *A 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*, …,* $Ro_i^{n_i}$*, and* col_i *such that one node of type* t_i^{Col} *is an additional condition to be enabled, and adding 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 17: Let us take the collaboration type for the ContractCollaboration we have defined in Example 5, and change it into an evolving collaboration type. In accordance with Definition 5.13 we have to change the collaboration's rule set by adding the special rule r_{CC}, which is shown in Figure 5.3.

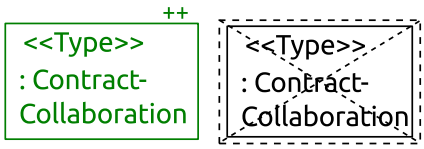


Figure 5.3: Special rule r_{CC} for the ContractCollaboration

The ContractCollaboration does not depend on any other component or collaboration type. Therefore the precondition of rule r_{CC} only contains the NAC that prevents the rule from being applied more than once. However, if the ContractCollaboration were to depend on other collaboration types, they would occur in the rule's precondition.

Definition 5.14. *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 an NAC and having only the additional precondition that all types it depends on have been activated already (their respective node exists).*

Example 18: As we have done for the collaboration type ContractCollaboration in the previous example, let us change the Factory component type, introduced in example 7, into a dynamically evolving component type as defined in Definition 5.14. The required additional rule is depicted in Figure 5.4. The Factory component type depends on the RequestOfferCollaboration collaboration type. The dependency is due to the fact that the Factory component type implements roles that are defined in the RequestOfferCollaboration collaboration type. Thus, it is important that the RequestOfferCollaboration type is present in the system before the Factory type gets introduced. This is specified in the above rule by adding the RequestOfferCollaboration type to the rule's precondition. The same holds true for the AbstrFactory abstract component type, which is the Factory's super-type.

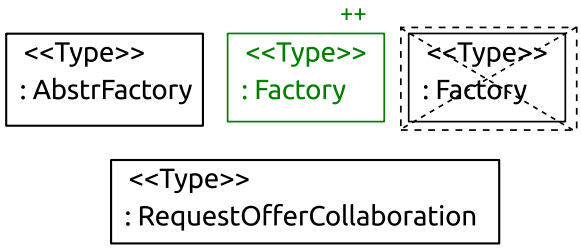


Figure 5.4: Special rule r_{Comp} for the Factory component

Definition 5.15. *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 19: To exemplify Definition 5.15, 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 collaboration 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 18) 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 5.16. *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 trace $t_1 \circ \cdots \circ t_n$ such that t_i is a trace in Sys_i leading from G_S^i to G_S^{i+1}, it holds that a related trace $t'_1 \circ \cdots \circ t'_n$ such that t'_i is a trace in $E(Sys_1, Sys_n)$ leading from G_S^i to G_S^{i+1} exists, such that $t'_i = E(t, Sys_1, Sys_n) \circ t_r$. The rule t_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 5.17, we can characterise what is required when an evolution sequence is extended by adding new types for collaborations and components.

Corollary 5.17. *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)* $Sys_1, \ldots, Sys_{n-1}$ *are correct (using the conditions of Theorem 5.16) 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 construct the conditions to prove $E(Sys_1, Sys_n)$ when all conditions for $E(Sys_1, Sys_{n-1})$ have been proven already. □

Corollary 5.17 provides a direct guideline for what has to be done when you want to add a new type for a collaboration or component. Note that an organisation which wants to extend the system type accordingly does not need to know all other types besides those which are refined. Furthermore, if two independent extensions are done which do not refer to each other, in the presented construction the concrete order does not matter as the checks remain the same. Therefore, each organisation can simply check its own extension, and the order in which they are enacted does not matter.

Lemma 5.18. *For a correct collaboration type* Col*, it holds also that its dynamic extension* $E(Col)$ *is correct. For a correct component type* Com*, it 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 $E(X)$ it 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 5.18 to simply check the collaboration and component types, and this already guarantees that any extended evolution sequence will also show correct behaviour.

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 20: 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 whether or not the bid is higher than the currently

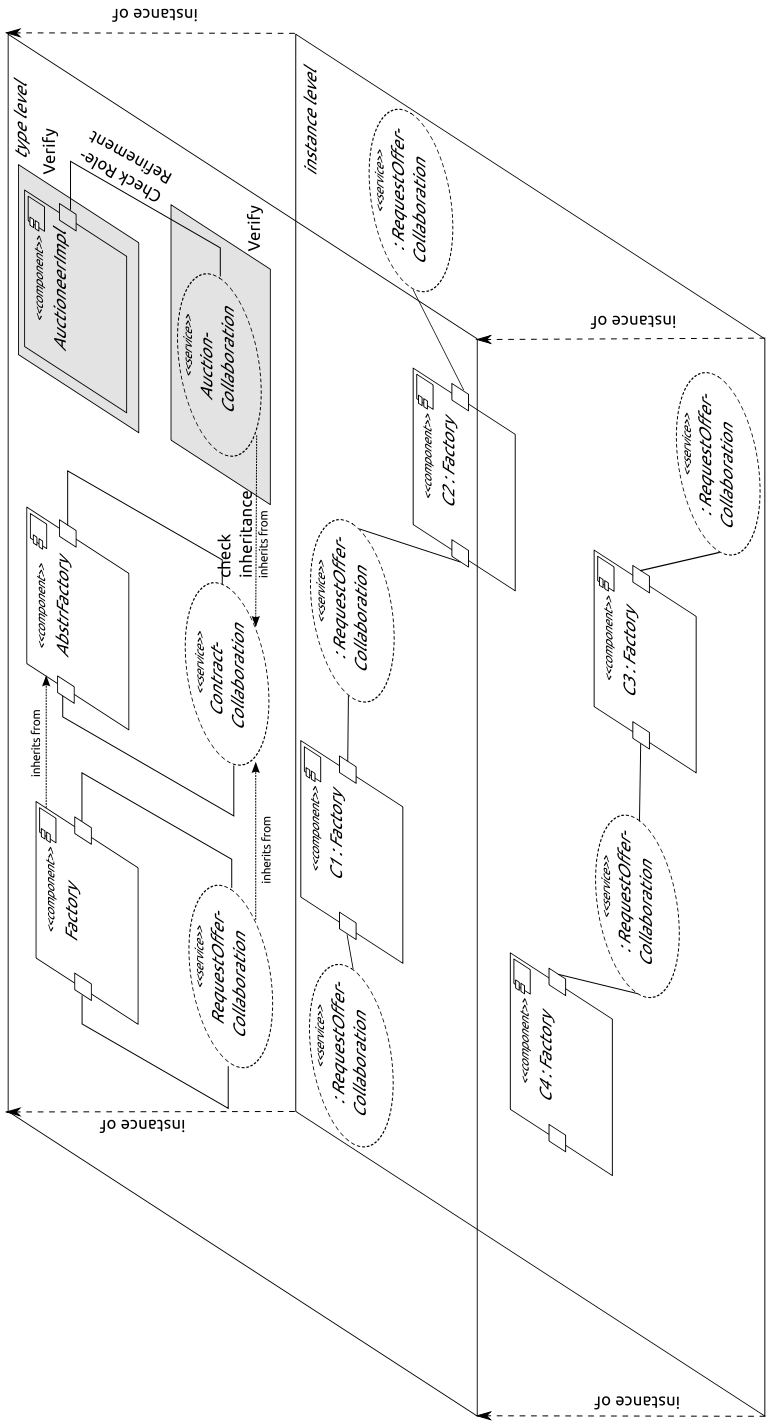


Figure 5.5: Incremental verification scheme for the verification of system types with evolution

leading bid and marks the bid as leading or discards it. The Auction collaboration requires us to introduce a new role Auctioneer, a collaboration Auction that refines the Contract collaboration, and the two specialised roles Supplier and Customer.

The Auction collaboration type is newly introduced and thus we have to verify that it satisfies its 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 types 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 5.17 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 system types can be analysed. 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 generalised variant of Theorem 5.8 that also covers abstraction. These findings accumulate in Theorem 5.11. In a last step, we showed that our approach can be facilitated to verify evolution of service-oriented systems; see Theorem 5.16 and Corollary 5.17.

As visualised in Figures 5.1, 5.2, and 5.5 the verification scheme only considers the types, supports subtyping, and permits us 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 13, a system with subtypes in Example 16, and an evolution step in Example 20 only needs to check the types. In addition, the building blocks of checking that the collaboration types are correct in Example 9 and that the component types are correct in Example 10 all require only moderate efforts. Therefore, *rigSoaML* can be checked for arbitrary large service landscapes and thus scale, as demanded by requirement Scalable Analysis (A1).

The rules in *rigSoaML* permit us to also model the way that systems change their configuration at run-time and thus the model captures the regular behaviour and reconfiguration behaviour at the same time. Consequently, the outlined analysis approach fulfils requirement Analysis of Reconfiguration (A3).

For our overall example of the supply chain case study, this means that within the specifications of components and collaborations — e.g. Factory and Request-OfferCollaboration — the rules for terminating a relation are already included. Thus, we 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 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 fulfils requirement Analysis under restricted knowledge (A4) .

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 its 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.

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 properties as well as the linkage to concepts it inherits properties from. Thus also requirement Analysing Evolution (A5), concerning the analysis of the uncoordinated introduction of new types for service contracts and components at run-time, is covered. Here, it is particularly 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 means 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 subtype 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.

5.4.1 Summary

As outlined, our approach does scale due to the compositional approach and fulfils the related analysis requirements A1–A4 by using abstraction to decouple the different concrete elements via abstract ones. Furthermore, requirement Analysing Evolution (A5) is supported by an incremental and decentralised verification scheme.

Requirements	Coverage by *rigSoaML*
Scalable Analysis (A1)	✓
Applicable Analysis (A2)	✓
Analysis of Reconfiguration (A3)	✓
Analysis under restricted knowledge (A4)	✓
Analysing Evolution (A5)	✓

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

At the beginning of this thesis we introduced typical scenarios for changes that can occur in SoS (see Section 2.5). We will now briefly recall these scenarios and explain the necessary proof obligations we get from the findings of this chapter. Our approach does not yet support the removal of component and collaboration types. The removal of instances is supported through the rules that can delete instances. The removal of types is difficult. Before a type (whether a collaboration or a component type) can be removed from the system type, first all its instances have to be removed from the system and all of its subtypes have to be removed from the system type. However, we don't have a comprehensive and consistent view of the system and thus cannot guarantee that these conditions are satisfied. The update of existing component or collaboration types cannot be achieved for similar reasons. For the update, either it has to be assured that the type could be updated independent of its instances' state or all instances have to be in a certain defined state that allows the update of the type. Again, the restricted knowledge of the system prohibits the update of types. However, the update of types could be partially avoided by the introduction of new types that have the updated behaviour. This solution, of course, does not update existing instances of the previous type.

For a newly added collaboration type, we have to check different obligations depending on whether or not the collaboration type refines an existing collaboration type. In the case of a new collaboration type that does not refine any existing collaboration type, we have, according to Definition 5.1, to ensure that the collaboration type's rules satisfy the collaboration type's safety property for all reachable states, starting in $G_\emptyset$. In Figure 5.1, this is depicted as the grey rectangle around the collaboration type. Should the collaboration type, however, refine an existing abstract collaboration type, we have to ensure that the refinement is correct and that the refining collaboration type's rules guarantee

its own and the refined collaboration type's safety properties (see Definition 4.11 and Figure 5.2). The verification of new concrete component types requires two things. First, we have to verify that the component type's rules guarantee the component type's safety properties (see Definition 5.2) and we have to check that the component type correctly refines the behaviour of the roles it implements. Should the component type refine an abstract component type, we additionally have to check that the component type satisfies the abstract component type's safety guarantees too, and that it correctly refines the abstract component type's behaviour.

Chapter 6

Verification Techniques

The previous chapter introduced the verification scheme we apply in order to verify architectural models of service-oriented systems of systems. The verification scheme gives some obligations we have to fulfil that were introduced by the use of the abstract operators (see Section 4.1.3). We will have to give a definition for refinement of rules and rule-sets that satisfies the abstract refinement operator's properties and we will have to present a verification technique that can verify properties over our *rigSoaML* models. We will start with a technique for the verification of inductive invariants, the Invariant Checker, in Section 6.1. Then we will explain the definition of refinement in Section 6.2 and finally show how the refinement relation can be automatically checked, which is done in Section 6.3.

6.1 Verifying Inductive Invariants

In the previous chapter we have explained what is necessary for a system type to be correct. What we have omitted so far is to present a concrete technique which allows for the verification that a rule-set satisfies a property. In this section, we will describe a way to statically verify a rule-set with respect to a given set of properties. The approach we use is called Invariant Checking and has already been presented in [2] and [5]. An extension for the incremental verification discrete GTS (i.e. such as using structural dynamics only) is described in [3].

6.1.1 Restricted *rigSoaML*

Our Invariant Checker approach currently does not support the property language's full range. The Invariant Checker is restricted to the verification of safety properties [110]. Liveness properties including progress and fairness properties are not supported by the Invariant Checker approach, but to the best of our knowledge currently a static verification of liveness properties has not yet been described in the literature. We therefore restrict the property language $\mathcal{L}$

(see Definition 4.1.2) to only allow conjunctions of negated atomic properties (graph constraints) that have to be globally (symbol $\Box$) satisfied.

Definition 6.1 (Property Language $\mathcal{L}^-$)**.** *Let AP be a set of atomic graph constraints. $\mathcal{L}^-$ can then be recursively defined as:*

$$\phi \wedge \psi \in \mathcal{L}^- \quad \forall \phi, \psi \in \mathcal{L}^-$$
$$\Box\neg\phi \in \mathcal{L}^- \quad \forall \phi \in AP$$

Each property $\phi \in \mathcal{L}^-$ is obviously a property of $\mathcal{L}$ too. Thus, the evaluation of the property language $\mathcal{L}^-$ does not change in comparison with $\mathcal{L}$.

In our approach, a set of forbidden graph patterns $\mathcal{F} = \{F_1, \ldots, F_n\}$ (see Definition B.4) are employed to represent possible safety violations of our system. We can derive a related property $\Phi_\mathcal{F} = \Box\neg F_1 \wedge \Box\neg F_2 \wedge \ldots \wedge \Box\neg F_n$. A graph G satisfies the property $\Phi_\mathcal{F}$ iff G satisfies each of the sub-properties (i.e. $\Box\neg F_i$).

The property $\Phi_\mathcal{F}$ is an *operational invariant* of the GTS S iff for a given initial graph G^0 for all $G \in \mathsf{REACH}(S, G^0)$ it holds that $G \models \Phi_\mathcal{F}$ (cf. [51]). However, due to the Turing-completeness of graph-transformation systems with types, checking them is restricted to finite models and thus does not fit the considered class of problems. We therefore instead tackle the problem of whether the property $\Phi_\mathcal{F}$ is an *inductive invariant*. This is the case if for all graphs G and for all rules $r \in \mathcal{R}$ it holds that $G \models \Phi_\mathcal{F} \ \wedge \ G \to_r G'$ implies $G' \models \Phi_\mathcal{F}$. If we have an inductive invariant and the initial graph G^0 fulfils the property, then $\Phi_\mathcal{F}$ is also an *operational invariant*, as inductive invariants are stronger than their operational counterparts.

We can reformulate the definition of an *inductive invariant* as follows, to have a falsifiable form: a property $\Phi_\mathcal{F}$ is an inductive invariant of a GTS $S = (G_0, R, p, T)$ (cf. Definition B.7) if and only if there exists no pair (G, r) of a graph G and a rule $r \in R$ such that $G \models \Phi_\mathcal{F}$, $G \to_r G'$ and $G' \not\models \Phi_\mathcal{F}$. Such a pair (G, r), which witnesses the violation of property $\Phi_\mathcal{F}$ by rule r, is then a *counterexample* for the initial hypothesis.

6.1.2 Checking of discrete GTS

As explained in detail in [2], we can exploit the fact that the application of a rule can only have a local effect to verify whether a counterexample exists. A counterexample (G, r) can only exist when the local modification of G by rule r is necessarily responsible for transforming the correct graph G into a graph that violates the property. In addition, to have a possible counterexample we require that the rule r is not preempted by a rule r'.

As we can represent the infinitely many possible counterexamples by only a finite set of representative patterns $\Theta(R_l, F_i)$ of graph patterns P' that are combinations of a RHS R_l of a rule r_l and a forbidden graph pattern $F_i \in \mathcal{F}$ (cf. [2]), we can check that no counterexample exists (and $\Phi_\mathcal{F}$ is thus an inductive invariant) only considering this finite set.

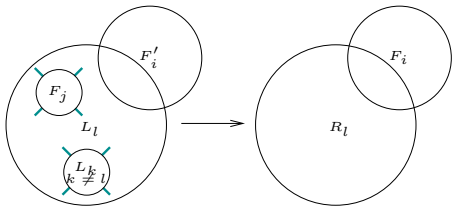


Figure 6.1: Schema to check a potential counterexample (P, r_l) with resulting graph pattern P' that is a combination of a RHS R_l of a rule r_l and a forbidden graph pattern $F_i \in \mathcal{F}$

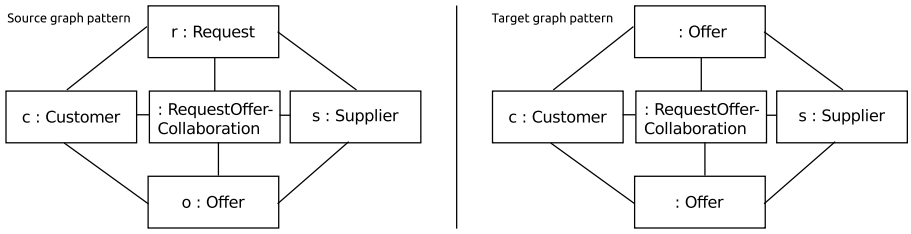


Figure 6.2: Pair of source-(left) and target-graph-pattern (right) as constructed by the Invariant Checker

As depicted in Figure 6.1, we have to check for any graph pattern $P' \in \Theta(F_i, R_l)$ for some $F_i \in \mathcal{F}$ and $r_l \in \mathcal{R}$ whether the pair (P, r_l) with P defined by $P \rightarrow_r P'$ is a counterexample for $\Phi_{\mathcal{F}}$ or not, as follows:

1. Check that the rule r_l can be applied to graph pattern P and that the resulting graph pattern is P' (this implies that no $r_k \in \mathcal{R} \setminus \{r_l\}$ exists with $prio(r_k) > prio(r_l)$ that matches P, due to the definition of rule application).

2. Check that there exists no $F_j \in \mathcal{F}$ with $F_j \sqsubseteq P$ (otherwise P is already invalid).

We use the above conditions in our algorithm to check if a counterexample exists. The algorithm performs this check for any given rule $(L, R)_r \in \mathcal{R}$ and forbidden graph pattern $F \in \mathcal{F}$. The algorithm therefore computes the set of all possible target graph patterns for R and the forbidden graph pattern F ($\Theta(R, F)$) and then derives the related source graph patterns. The above sketched conditions are then checked for all source graph patterns to decide whether the source graph pattern P represents potentially safe graphs that can be transformed into unsafe graphs by applying r. If so, the pair (P, r) is a valid counterexample.

Example 21: To exemplify the verification algorithm let us consider the supply chain system. The rule L_k is the RequestOfferCollaboration's `makeOffer` rule (cf. Figure 4.9), and the forbidden pattern F_i is `noTwoOffers` (cf. Figure 4.7(b)). In Figure 6.2 a possible pair of source and target graph patterns is shown. It can easily be seen that the depicted target graph pattern (the figure's right-hand side) is a combination of `makeOffer` and `noTwoOffers`. The source graph pattern contains the

`noOfandReq` forbidden pattern (cf. Figure 4.7(c)) as a subgraph and thus the constructed pair is no witness against the GTS' correctness.

6.1.3 Extension for Hybrid Systems

In our previous work [5], we have shown how to extend the Invariant Checker to also capture time-dependent behaviour. In the following, we will build atop of these findings and extend the Invariant Checker to verify hybrid behaviour too. The main difference in the verification is the fact that in [5] we have introduced a system of linear inequalities that modelled the timed behaviour and for arbitrary hybrid behaviour we are going to use hybrid automata [25, 85], as linear inequalities are not sufficient to express arbitrary continuous behaviour.

The verification of hybrid systems differs significantly from the verification of discrete graph-transformation systems as presented in the foregoing Section 6.1.2. In the employed hybrid model, the behaviour is described by rule applications and continuous steps. Therefore, reaching a forbidden graph pattern in principle could involve a rule application as well as a continuous step. The basic idea to approach the checking is to extend the discrete case by mapping the continuous behaviour-related aspects on a hybrid automaton, which can then be checked by a hybrid model-checker. Thereby, the hybrid model checker used limits the usable class of differential equations for the encoding of continuous behaviour.

Analogously to the discrete case, we can formulate the definition of an *inductive invariant* for the hybrid case in a falsifiable form: a property $\Phi_{\mathcal{F}}$ with forbidden attributed graph pattern $(F_i, \psi_i) \in \mathcal{F}$ is an inductive invariant of a HGTS $S = (G_0, R, p, T, \mathcal{R}_u)$ (cf. Definition B.13) if and only if there exists no pair $((G, \beta), r)$ of an attributed graph (G, β) (see Definition 4.1) and a hybrid graph-transformation rule $r \in R$ such that $(G, \beta) \models \Phi_{\mathcal{F}}$, $(G, \beta) \rightarrow_r \rightarrow_\delta (G', \beta)$, and $(G', \beta) \not\models \Phi_{\mathcal{F}}$. Such a pair $((G, \beta), r)$ which witnesses the violation of property $\Phi_{\mathcal{F}}$ by rule r is then a *counterexample* for the hybrid case.[1] Using the same idea as for the discrete case, we can lift this problem to an attributed graph pattern. Again, only a finite set of representative patterns $\Theta((F_i, \psi_i), R_l, \mu_l$ of graph patterns P' that are combinations of a RHS R_l of a rule $r_l = (L_l, R_l, K_l, l_l, r_l, A_l^-, \phi_l)$ and a forbidden graph pattern $(F_i, \psi_i) \in \mathcal{F}$ have to be considered.

As depicted in Figure 6.3, we have to check for any graph pattern $(P', \phi_{P'}) \in \Theta((F_i, \psi_i), R_l, \mu_l)$ for some $(F_i, \psi_i) \in \mathcal{F}$ and $r_l \in \mathcal{R}$ whether the pair $((P, \phi_P), r_l)$ with (P, ϕ_P) defined by $(P, \phi_P) \rightarrow_r \rightarrow_\delta (P', \phi_{P'})$ is a counterexample for $\Phi_{\mathcal{F}}$ or not, as follows:

1. Check that the rule r_l can be applied to attributed graph pattern (P, ϕ_P) and that the $(P', \phi_{P'})$ results from this application plus a time step of length $\delta \geq 0$ (this implies that no $r_k \in \mathcal{R}_u \setminus \{r_l\}$ exists with $prio(r_k) > prio(r_l)$ that matches (P, ϕ_P) and that for all $x \leq \delta$ it holds that $(P', \phi_{P'} \ominus x)$ is matched by no $r_m \in \mathcal{R}_u$, due to the definition of rule application).

[1] This condition in fact requires that for an initial state (G, β) we check not only that $(G, \beta) \models \Phi_{\mathcal{F}}$ but also that $(G, \beta \oplus x) \models \Phi_{\mathcal{F}}$ for all x with $(G, \beta) \rightarrow_x (G, \beta \oplus x)$.

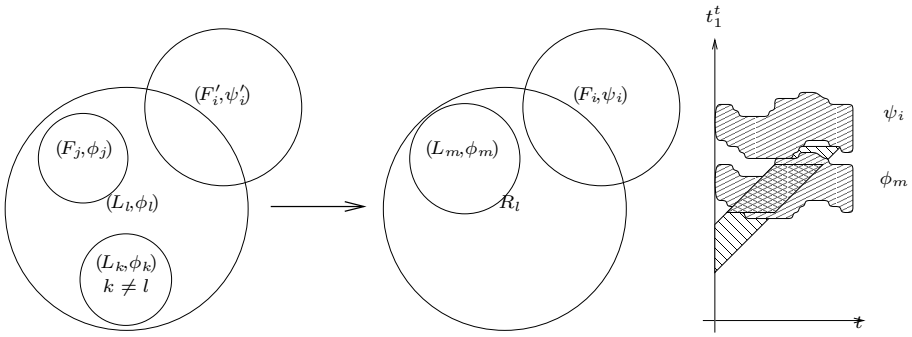


Figure 6.3: Schema to check a potential counterexample $((P, \phi_P), r_l)$ with resulting graph pattern $(P', \phi_{P'})$ that is a combination of a RHS R_l of a rule r_l and a forbidden graph pattern $(F_i, \psi_i) \in \mathcal{F}$ in the hybrid case

2. Check that there exists no $(F_j, \phi_j) \in \mathcal{F}$ with $(F_j, \phi_j) \sqsubseteq (P, \phi_P)$ (otherwise (P, ϕ_P) is already invalid).

Therefore, the extended checking algorithm employs in its first step a slightly adjusted version of the discrete algorithm to derive potential counterexamples (see Figure 6.3). In a second step, the algorithm constructs an instance of a hybrid automaton to also verify the continuous behaviour. The generic automaton we use for the hybrid checking is depicted in Figure 6.4. We follow the notation and semantics of the hybrid automaton as described by Henzinger [86].

For preempting rules $((L_k, \phi_k))$ as well as forbidden attributed graph patterns $((F_j, \psi_j))$, it holds in the case that they also contain attribute constraints that a match found in the source graph pattern does not directly invalidate the counterexample but rather restricts the possible attribute values. We derive a system of constraints ϕ_{Init} to encode which values for the source graph pattern's attributes are not excluded either by preempting rules or forbidden graph patterns combining the conditions $iso(\psi_j)$[2] for all matches iso of F_j in (P, ϕ_P) and $iso'(\phi_k)$ for all matches iso' of L_k in (P, ϕ_P). The initial condition additionally resets a special timer variable $timer$ to zero.

Concerning the application of the rule r, we have to take into account that the rule may either not affect variables (which are therefore in their possible values determined by ϕ_{Init}) or update them to a given constant. While in the discrete case the match of the forbidden graph pattern in the target graph pattern is sufficient to ensure that $\Phi_{\mathcal{F}}$ is not valid any more, in the HGTS case the related forbidden attributed graph pattern (F_i, ψ_i) may also require that the variables' valuations fulfil the additionally specified conditions (see the ψ_i area in the constraint space on the right-hand side of Figure 6.3). We therefore encode the continuous evolution after the discrete step in an additional system of differential equations DE_t that stem from the target graph patterns' control modes. The differential equations DE_t form the state invariant for the state "target pattern" in the generic automaton.

In the discrete case, it was sufficient to check whether the target graph pattern

[2]with $iso(\psi_j)$ we denote the projection of the condition ψ_j over the isomorphism iso

can be reached to judge whether the forbidden graph can be reached. In the hybrid case, urgent rules may in fact prevent us from reaching an attribute evaluation which fulfils the constraints of the structurally embedded forbidden graph patterns. This effect is encoded in another system of constraints ϕu (see the ϕ_m area in the constraint space on the right-hand side of Figure 6.3).

After having identified the different constraint systems and systems of differential equations that determine the automaton's behaviour, we will now explain the automaton's detailed construction. The state "source pattern" has the state invariant $timer \leq 0$ and the flow condition $\dot{timer} = 1$. The only outgoing transition from this state is equipped with the jump condition $timer \geq 0 \wedge \phi_r$, and ensures that the state "source pattern" is left immediately and the effect of the application of rule r is covered by the automaton. The "target pattern" state's flow condition is given through the combination of control modes present in the target pattern. It is encoded through the set of differential equations DE_t, as it has been computed by the algorithm (for details on control modes see Section B.3). The "target pattern" state's invariants ensure that neither the currently checked forbidden pattern's attribute condition is fulfilled $\neg\phi_{F_i}$ nor that any urgent rule is activated $\neg\phi_U$. The constraint ϕ_U is a disjunction of the urgent rule's jump conditions that are structurally embedded in the target graph pattern. The "target pattern" state has two outgoing transitions, one to the "failure" state that is equipped with a guard $\phi_{F_i} \wedge \neg\phi_u$ and one to the "urgent state", that is guarded by ϕ_U.

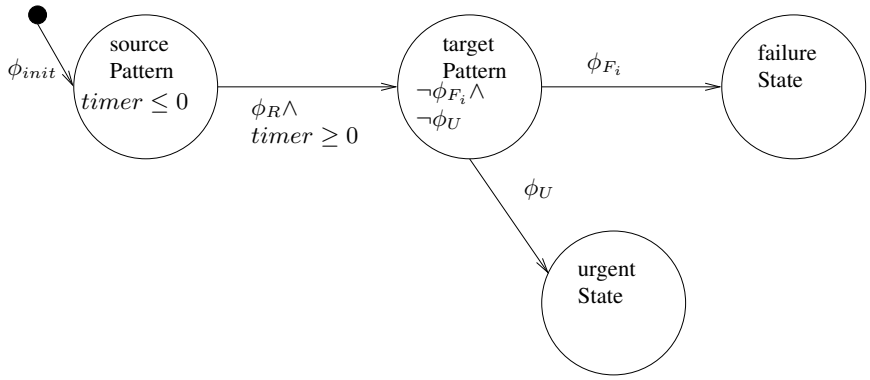


Figure 6.4: The generic automaton, which is used for the hybrid model checking

For the automaton, only two different paths are possible, or the automaton runs into a deadlock. The initial condition ϕ_{init} ensures that the attribute conditions of all forbidden patterns found in the source graph pattern are not satisfied and that no graph- transformation rule that preempts the current rule r can be applied. Hence, we start from a correct state. The state invariant, however, requires that the "source pattern" location has to be left immediately. This is only possible if the jump condition ϕ_r can be satisfied. Otherwise, the automaton runs into a deadlock and cannot advance and the current pair is not a counterexample. If the location "target pattern" can be reached, we have successfully simulated the application of rule r from a safe source graph pattern to a target graph pattern. The "target pattern" location only has to be left if its invariant is violated. This is the case if either the attribute condition ϕ_{F_i} of the

current forbidden pattern is satisfied or any of the urgent rules that match in the target graph pattern is enabled. The location "failure" can only be reached if ϕ_{F_i} is satisfied and ϕ_u is not. The location "urgent" can only be reached if the guard ϕ_U is satisfied.

We can conclude that, if the location "failure" can be reached, we have found a transition from a safe source graph pattern to an unsafe target graph pattern without any urgent rule being fired. In this case, we have found a valid counterexample against the HGTS correctness. Otherwise, the automaton either reached the "urgent" location and then any of the urgent rules prevented an unsafe situation occurring, or the automaton ran into a deadlock due to unsatisfiable guards when transitioning from the "source pattern" to the "target pattern" location. In both cases, we found no counterexample.

Summarising, we map the verification problem for HGTS to a discrete part that is very similar to the verification of GTS, and a hybrid-model-checking problem. For the latter, we make use of a generic hybrid automaton (cf. Figure 6.4) that describes the steps of the HGTS according to the possible attribute valuations. The hybrid automaton's initial condition ensures that the source graph patterns contain no forbidden pattern (F_j, ψ_j), and the attribute update when transitioning from the "source pattern" state to the "target pattern" state models the graph-transformation rule's jump condition. In the "target pattern" state the state-invariant ensures that the continuous evolution of the attribute values conforms to the target graph pattern's control modes and that neither an urgent graph-transformation rule is applicable nor is the attribute constraint ψ_i of the forbidden pattern (F_i, ψ_i) fulfilled. If one of these two cases occurs, the "target pattern" state has to be left. Depending on the reason for which the automaton leaves the "target pattern" state, either the "failure" state or the "urgent" state is reached. The system is unsafe if the "failure" state is reachable. Concerning the differential equations that are to be built from the different control modes available in the graph patterns, it has to be ensured by the modeller that the differential equations are always closed. We consider a differential equation as closed if no dependencies on the values of adjacent nodes exist. If this restriction is violated, it could happen that, due to the minimal context the Invariant Checker algorithm produces, the differential equations are not well-formed.

6.2 Refinement

Before we can start to formally define our variant of the abstract refinement operator, we will have to discuss some additional properties of traces. Afterwards, we will give a semantical definition of refinement and show that the syntactical constraints, together with some additional checks, can guarantee the semantical refinement.

6.2.1 Traces

Traces have been introduced in Definition 4.4 in Section 4.1.1. They are used to describe a possible execution path of HGTS and are sequences of graphs.

What we will need when we want to reason about refinement is the possibility to express a trace in a restricted type-graph and by only using a smaller set of rules. This step is called restriction and can be formally defined as:

Definition 6.2 (Restricted Trace)**.** *Given a trace t, where all states $t(i)$ are typed over type-graph TG, we can restrict t to a type-graph $TG' \leq TG$, denoted as $t|_{T'}$ by restricting each graph $G_i \in t$ to $G_{i|T'}$.*

Valuations are dependent on the available control modes, and each control mode has the exclusive control over a set of attributes. It follows that, in a restricted graph, the valuations become 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.

Example 22: To exemplify the notion of a restricted trace, we will restrict trace t introduced in Example 1 to the type-graph T_{Con} of the ContractCollaboration collaboration type. To make the differences more explicit, we did not remove the deleted elements but displayed them in light grey. Hence, the trace $t|_{T_{Con}}$ consists only of the black elements. The restricted trace $t|_{T_{Con}}$ is depicted in Figure 6.5.

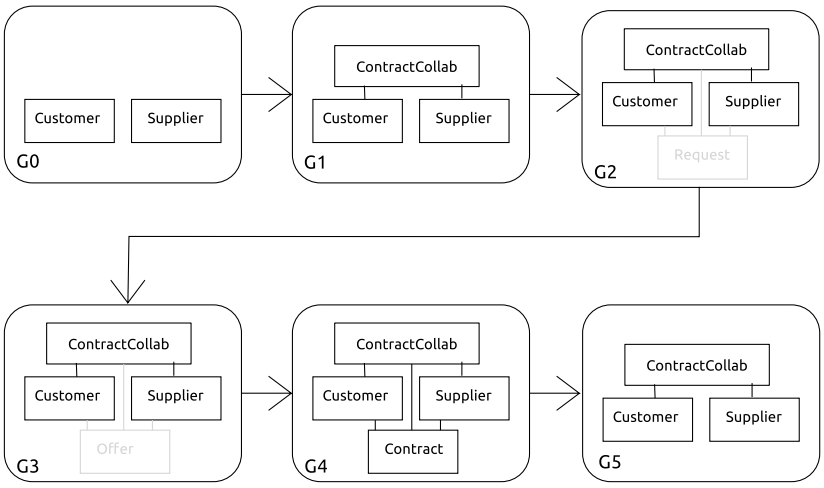


Figure 6.5: The trace shown in Figure 4.1 restricted to the ContractCollaboration's type-graph

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.

Definition 6.3 (Stutter Operator)**.** *Given a trace t, the condensed trace $\natural t$ 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 $i+1 < k \leq j$ is 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 $t = \ldots G_{i-1}, G_i, G_{i+1}, \ldots$ with G_i and G_{i+1} being the resulting graphs of two continuous steps $G_{i-1} \xrightarrow{r_1} G_i$ and $G_i \xrightarrow{r_2} G_{i+1}$ and $r_1, r_2 \in \mathbb{R}^{>0}$ and where the graph G_i is essentially required for $t \models \phi$, then the trace $t' = \ldots G_{i-1}, G_{i+1}$ does not satisfy ϕ as the graph G_i is missing in t'.

Example 23: We can use the trace $t|_{T_{Con}}$ shown in Example 22 to show the result of the $\natural$ Operator. The trace $t|_{T_{Con}}$ contains the consecutive states G_1, G_2 and G_3 which are all isomorphic to each other. We can thus condense trace $t|_{T_{Con}}$ to $\natural\,(t|_{T_{Con}})$ by removing the states G_2, G_3. The resulting trace is depicted in Figure 6.6; note that the states' ordinal numbers changed too.

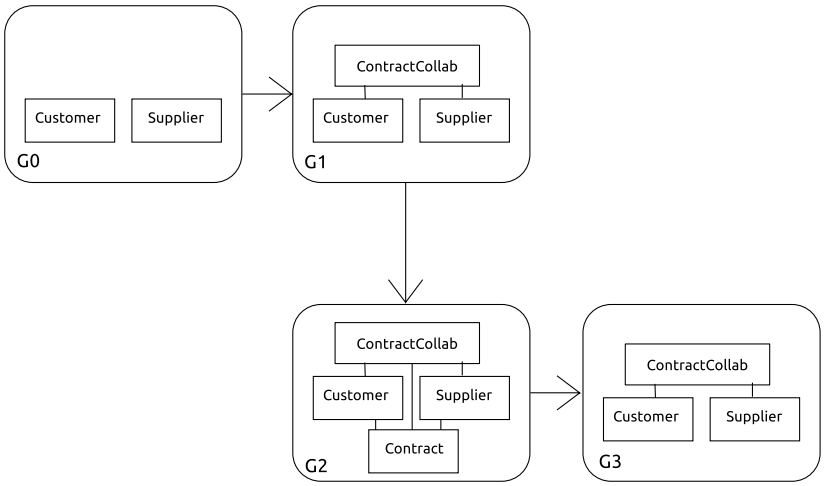


Figure 6.6: The trace from Figure 6.5 restricted by the $\natural$ operator

6.2.2 Properties

The restriction of traces (see Definition 6.2) and the removal of replicated states using the $\natural$ operator (see Definition 6.3) directly raises the question of the impact of these operations on the properties that still hold for the traces. The properties we allow for *rigSoaML* do not contain any operator that directly addresses the next state[3] and therefore it is acceptable to remove states from the trace. The following lemma makes the arguments more explicit.

Lemma 6.4 (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 the existence of states and do not consider the properties of the

[3]In CTL and LTL typically a next-operator exists

next state. The stutter operator $\natural$ removes repeating states from the trace. Obviously, if the trace t contains a state s, the trace $\natural t$ also contains this state. Only the ordinal number of the state within the trace has changed. This said, it can be easily followed that the lemma holds. □

Example 24: Using the same arguments as in Example 2, we can show that the property $\Box\neg\phi$, with ϕ being similar to Example 2, also holds for the restricted trace, depicted in Figure 6.5. It can be easily seen that the removal of the states G_2 and G_3 does not invalidate the argument for the satisfaction of $\Box\neg\phi$ through t. Hence, the trace $\natural t \models \Box\neg\phi$ holds too.

6.2.3 Semantical Refinement

In this section we will introduce our exact notion of refinement, which is used to fill in the abstract refinement operator. As our semantical domain are traces, we will first define refinement of traces. However, in order to be able to do so, we have to formally define what refinement of rules is.

Definition 6.5 (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 $t' = H_0, H_1, \ldots$ refines $t = G_0, G_1, \ldots$ if we can find a mapping* ord $: \mathbb{N} \mapsto \mathbb{N}$ *such that the restricted trace*

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

satisfies the following conditions:

$$\begin{gathered}
\text{ord}(0) = 0 \\
\forall i, j \in \mathbb{N} : i > j \implies \text{ord}(i) > \text{ord}(j) \\
H_{\text{ord}(i)|T} \approx G_i \\
\forall i, j \in \mathbb{N} : \text{ord}(i) < j \leq \text{ord}(i+1) \implies H_{j|T} \approx G_{i+1}
\end{gathered}$$

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

In the above definition, we use the function ord to map relative states to each other. We consider two states to be relative to each other if the restricted graph of the refining state is isomorphic to the graph of the refined state.

Example 25: To exemplify the semantical refinement of traces as it has been defined in Definition 6.5, let us investigate the traces we used in the Examples 23 and 1. We want to show that the trace t from Example 23 is refined by trace s, used in Example 1. Definition 6.5 requires that we construct a trace s' that is restricted to the type-graph used for trace t. We have already constructed such a restricted trace in Example 22. The next step is to create the mapping function ord. For our example, this function could be given as:

$$\text{ord} = \{(0,0); (1,1); (2,4); (3,5)\}$$

The mapping function ord is visualised in Figure 6.7. It remains to show

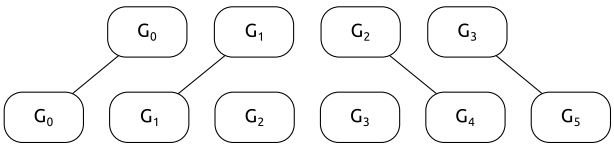


Figure 6.7: Visualisation of the mapping function ord

that the states being matched by the function ord are isomorphic and that the states G_1, G_2 and G_3 of trace s' are isomorphic to each other. This can be easily seen in the respective figures.

Lemma 6.6 (Refinement invariant trace properties)**.** *Given two traces t and p with type-graphs T and P with $T < P$ and $p \prec t$ and a property $\phi \in \mathcal{L}$, where the atomic properties of ϕ are typed over T, then $t \models \phi \implies thatp \models \phi$ 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 $p_{|T}$. Definition 6.5 gives us that $p_{|T}$ is equivalent to t modulo the stuttering[4]. Together with Lemma 6.4 we obtain that the lemma holds. □

After having defined the refinement of single traces and having shown which properties could be preserved through the refinement relation, we will now extend our definition to the refinement of a whole set of traces.

Definition 6.7 (Trace-set Refinement)**.** *Given two sets of traces S and Q, the set S refines set Q, iff*

$$\forall s \in S : \exists q \in Q : s \prec q$$

We denote this refinement relationship as $S \overset{*}{\prec} Q$.

The definition of trace-set refinement does not require that each trace in the refined trace-set has a counterpart in the refining set. Thus, a trace-set refinement is valid also if not the complete behaviour is preserved by the refining trace-set. Using the above definition of trace-set refinement, we can finally formulate the missing argument, as to why our notion of refinement satisfies the properties that the abstract refinement operator requires.

Corollary 6.8 (Trace-Set Refinement preserves Properties)**.** *Given two sets of traces S and Q with $S \overset{*}{\prec} Q$ and a property $\phi \in \mathcal{L}$ with $Q \models \phi$, it then also holds that $S \models \phi$.*

Proof. We can safely assume that the trace-sets S and Q were defined atop the two type-graphs TG_S and TG_Q respectively. Further, the atomic properties contained in ϕ are expressed in terms of TG_Q. From the assumption that $Q \models \phi$, we can conclude that any trace $t_Q \in Q$ satisfies ϕ. Definition 6.7 gives us that for each trace $t_S \in S$ a trace $t_Q \in Q$ exists such that $t_S \prec t_Q$. Together with Lemma 6.6, we can then conclude that also $t_S \models \phi$ holds. Hence, we have the required result: $S \models \phi$. □

[4]In $p_{|T}$ occur only those states that also occur in t. But some states that contain only changes in the elements that are exclusively defined in type graph P are repeated.

6.2.4 Syntactical Refinement

After the definition of the semantical refinement, we will now derive a notion of a syntactical refinement. The focus in this section is more on sets of rules rather than on traces and sets of traces. In contrast to Section 6.2.3, we will now develop the necessary arguments under which two sets of rules refine each other.

Definition 6.9 (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 can say that R_2 refines R_1, if there exist two graph morphisms m_l and m_r with $L_{R_1} \mapsto_{m_l} L_{R_2}$, $R_{R_1} \mapsto_{m_r} R_{R_2}$ and if the elements in $L_{R_2} \setminus ran(m_l)$ and $R_{R_2} \setminus ran(m_r)$ 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 $R_2 \lesssim R_1$.

Informally speaking, 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 decreases 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.

> **Example 26:**[Rule Refinement] Let us revisit the two `createContract` rules of the ContractCollaboration and the RequestOfferCollaboration (cf. Figure 4.4 and Figure 4.10, respectively). The difference between the two rules is that the refining `createContract` rule of the RequestOfferCollaboration additionally deletes a previously made Offer object. Thus, we can create the required isomorphism between the two rules and, as the Offer node is defined in the RequestOfferCollaboration's class diagram only, we have a valid refinement.

Definition 6.10 (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 $R' \sqsubseteq_C R$.*

In order to further elaborate refinement of the rule-set, we need additional constructs, which we will introduce in the following.

Definition 6.11 (Path)**.** *A path π for a start graph G_0 and a set of rules R is given as $\pi = (G_0, S)$, where S is a sequence of pairs of graph-morphisms and rules or a continuous step of duration $\delta \in \mathbb{R}^{>0}$, such that either $G_i \xrightarrow{r_i}_{m_i} G_{i+1}$ for $(m_i, r_i) \in S$ and $r_i \in R$ or $G_i \xrightarrow{\delta_i} G_{i+1}$ holds. The i^{th} state of a path π is accessed by $\pi(i)$. For each path π we can construct a corresponding trace t^π by iterating the path's states. It trivially holds that $t^\pi(i) = \pi(i)$.*

The above definition suggests that a path is a construct that is closely related to a trace. In fact the only difference is that a path holds the additional information on how the state has been reached. Therefore, we can analogously define what

a restricted path is. However, the restriction now has to consider the different rule-sets too.

Definition 6.12 (Restricted Path)**.** *A path $\pi = (G_0, S)$ for a set of rules R with G_0 and the rules of R being defined over a type-graph T can be restricted to a path π' defined over a rule-set R' and type-graph T' with $T' < T$ by replacing each graph G_i occurring in π by its restricted pendant $G_{i|T'}$ and changing each step $G_i \xrightarrow{r_i} G_{i+1}$ to $G_{i|T'} \xrightarrow{r'_i} G_{i+1|t'}$ iff $r'_i \lesssim r_i$ and $G_{i|T'} \xrightarrow{\tau} G_{i+1|T'}$ otherwise. Continuous steps are not changed. The set of all paths for fixed start graph G_0 and rule-set R is given as $\Pi(G_0, R)$. The set of all paths for any start graph and rule-set R is given as $\Pi(R)$.*

After having defined restricted paths, we can now continue with the definition of path refinement.

Definition 6.13 (Path Refinement)**.** *Given two paths $\pi = (G_0, R)$ and $\pi' = (G'_0, R')$ being defined over the type graphs T and T' with $T < T'$, respectively, the path π' refines the path π iff the two corresponding traces t^π and $t^{\pi'}$ refine each other with $t^{\pi'} \prec_{\text{ord}} t^\pi$.*

The introduced definitions of paths, restricted paths and path refinement allow us to give a constructive lemma that guarantees rule-set refinement purely relying on syntactical properties of the rule-sets.

Lemma 6.14 (Rule-set Refinement)**.** *Given two sets of graph-transformation rules $R = \{r_1, \ldots, r_n\}$ and $R' = \{r_1, \ldots, r_m\}$ with $m \geq n$, the rules of R can be defined over type-graph T and those of R' over T' with $T < T'$. $RS' \sqsubseteq_C RS$ if*

$$\forall r_i \in R : \exists r_j \in R' : r_j \lesssim r_i$$

and, all rules $r_j \in R'$ for which no rule $r_i \in R$ exists, such that $r_j \lesssim r_i$ must not write (i.e. create or delete) any nodes or edges that are defined in type-graph T.

Proof. According to Definition 6.10, 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 $t' \prec t$. Obviously, for each trace t there must exist a corresponding path π such that $t = t^\pi$. We will prove the lemma by contradiction and thus we assume that a path $\pi' \in \Pi(R')$ exists such that no corresponding path π in $\Pi(R)$ can be found. Without loss of generality, we further assume that we have two paths $\pi' \in \Pi(R')$ and $\pi \in \Pi(R)$ that are in a valid refinement relation for the first i steps in the refined path and then diverge. Hence, if we construct the traces t^π and $t^{\pi'}$ we can find a mapping function ord such that $t^{\pi \leq i} \prec_{\text{ord}} t^{\pi \leq j}$ for $j \geq i$ and $\text{ord}(i) = j$ holds. In the following, we will refer to the states $\pi(i)$ as G_i and to $\pi'(j)$ as G'_j.

By construction, the divergent behaviour can only originate from rules that are present in both rule-sets. 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 $G'_j \xrightarrow{r'_j} G'_{j+1}$ must be performed through rules $r'_j \not\lesssim r_i$. Thus, there must be a third rule r_k with $r'_j \lesssim r_k$ that is applicable in G_i. It follows that a path of the rule-set R has to exist, that applies rule r_k instead of r_i in this case, and we do not have a divergent behaviour in position

i. The applicability of r_k to G_i is given through the applicability of r'_j to G'_j. Hence, a path $\pi^* \in \Pi(R)$ must exist that does not show the divergent behaviour. This gives us, according to Definition 6.10, the required condition for a correct rule-set. □

Complex Rule-Set Refinement

For some situations the above used rule-sets do not provide sufficient strength of expression. It might sometimes be useful to have the ability to express that one rule is prioritised over another or that a rule has to be applied once it is applicable (the so-called urgent rule). For these situations we need a different notion of rule-sets and in consequence a different notion of rule-set refinement.

Definition 6.15 (Complex Rule sets)**.** *A complex rule-set $S = (R, p, \mathcal{R}_u)$ is given through a set of rules R, a transitive preemption relation $p : R \times R$ and a set of urgent rules $\mathcal{R}_u \subseteq R$. The rules in $\mathcal{R}_u$ have to be applied once they are enabled, while the other rules can optionally 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 no rule $r' \in R$ exists with $G \xrightarrow{r'} G''$ and $(r', r) \in p$, i.e. if no applicable rule r' exists that preempts the rule r.*[5]

In the definition of roles, component types and collaboration types, for brevity reasons, we have only used rule-sets without preemption and urgent behaviour. However, complex rule-sets can be simply used as a replacement if the extra expression strength is required. For a complex rule-set S, we can also give a set of paths $\Pi(S)$ in the same way that we did in Definition 6.11, but we have to consider the restrictions concerning the rule applicability given in Definition 6.15.

Refining complex rule-sets raises several difficulties. In refining a complex rule-set, it has to be assured that rules that are being preempted in the refined complex rule-set are also preempted in the refined variant. In Figure 6.8 a sketch of this situation is depicted. Above the dashed line, a snippet of a path for the refined complex rule-set is shown, with one for the refining complex rule-set below. In the refined path, rule r_2 is applicable but preempted by the also applicable rule r_1 due to $(r_1, r_2) \in p$. The refinement $r'_1 \lesssim r_1$ and $r'_2 \lesssim r_2$ may end in a situation where r'_1 is not applicable (remember: rule refinement means strengthening the rule's precondition), but r'_2 still is. The transition labelled with τ should indicate that the missing precondition for rule r'_1 can be created by a different rule that only exists in the refining complex rule-set, which does not preempt rule r'_2. Hence, in the refining system there exists a path, which is impossible to exist in the refined one and it follows that the refining system no longer simulates the abstract one.

In consequence we have to ensure that whenever a graph-rule preempts the application of a lower priority rule, this preemption also occurs in the refined set of graph-rules. The easy way to do this is to prohibit the refinement of any rule that preempts at least one other rule. Obviously, this restriction would severely limit the applicability of our approach. A more versatile solution is

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

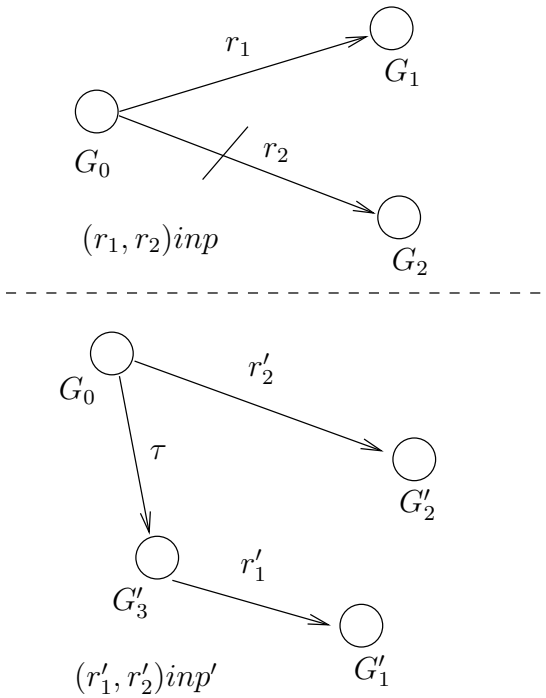


Figure 6.8: Rule Refinement Sketch

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 two sets of rules $r'_{1,1} \dots r'_{1,n}$ and $r'_{2,1} \dots r'_{2,m}$. For the refined rules, it has to hold that whenever the rule r_1 is applicable and additionally any of the lower-priority rules $r'_{2,1} \dots r'_{2,m}$ is applicable, then we require also that at least one of the rules $r'_{1,1} \dots r'_{1,n}$ is applicable, which then preempts the lower priority rule refining r_2. In the following, we will denote this characteristic by a predicate $Preempt(R, Q)$ where R and Q are two complex rule-sets. $Preempt(R, Q)$ is true whenever the refining rules in Q preserve the preempting behaviour of the rules in R.

Definition 6.16 (Preempt predicate)**.** *Given two rule-sets $RS_1 = (R_1, p_1, \mathcal{R}_u)$ and $RS_2 = (R_2, p_2, \mathcal{R}_u)$, these satisfy the predicate $Preempt(RS_1, RS_2)$ iff no two paths $t \in \Pi(RS_1)$ and $s \in \Pi(RS_2)$ exist such that*

$$\exists r_i^1, r_j^1 \in R_1 \wedge (r_i^1, r_j^1) \in p_1 \wedge$$
$$\exists r_i^2, r_j^2 \in R_2 \wedge (r_i^2, r_j^2) \in p_2 \wedge r_i^2 \lesssim r_i^1 \wedge r_j^2 \lesssim r_j^1 \wedge$$
$$\exists G_k^1 \in t, G_m^2 \in s \wedge G_k^1 \approx G_m^2 \wedge \mathrm{ord}(k) = m \wedge s^{\leq m} \prec_{\mathrm{ord}} t^{\leq k} \wedge$$
$$G_k \xrightarrow{r_i^1} G_{k+1} \in t \wedge G_m \xrightarrow{r_j^2} G_{m+1} \in s$$

For timed and hybrid graph-rules, we also have to consider the urgent rules. These rules are comparable to preempting 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. This means that whenever

an urgent rule r_u is applicable, we have to have at least one rule in the set of refining rules $r'_{u,1} \dots r'_{u,n}$ that is applicable too. The difference here, compared to the preemption of rules, is that urgent rules have to be applied and behaviour that is captured by urgent rules has to happen. Otherwise, safety criteria could often not be met by the system. By analogy to the *Preempt*-predicate, we define a predicate $Urgent(R, Q)$.

Definition 6.17 (Urgent predicate)**.** *Given two rule-sets $RS_1 = (R_1, p_1, \mathcal{R}_u^1)$ and $RS_2 = (R_2, p_2, \mathcal{R}_u^2)$, these satisfy the predicate $Urgent(RS_1, RS_2)$ iff no two paths $t \in \Pi(RS_1)$ and $s \in \Pi(RS_2)$ exist such that*

$$\begin{gathered}\exists r_i^1 \in \mathcal{R}_u^1 \wedge \exists k \in \mathbb{N} \wedge G_k \xrightarrow{r_i^1} G_{k+1} \in t \wedge \\ \exists m \in \mathbb{N} \wedge s^{\preceq m} \prec t^{\preceq k} \wedge \mathrm{ord}(k) = m \wedge \\ \nexists r_j^2 \in \mathcal{R}_u^2 \wedge r_j^2 \lesssim r_i^1 \wedge S_m \xrightarrow{r_j^2} S_{m+1} \in s\end{gathered}$$

The above definition states that the *Urgent* predicate holds if, whenever two paths t and s are in a perfect refinement relation for a certain number of steps ($s^{\preceq m} \prec t^{\preceq k}$) and then in the refined path an urgent rule has to be applied ($G_k \xrightarrow{r_i^1} G_{k+1}$), then there must be an applicable urgent rule in the refining rule-set that is applicable in the refining path.

Lemma 6.18 (Complex Rule-set Refinement)**.** *Given two sets of graph-transformation rules $RS = (R, p, \mathcal{R}_u)$ with $R = \{r_1, \dots, r_n\}$ and $RS' = (R', p', \mathcal{R}'_u)$ with $R' = \{r_1, \dots, r_m\}$ with $m \geq n$ and the rule-sets being defined over type-graphs T and T', respectively, with $T \leq T'$, then $RS' \sqsubseteq_C RS$ iff*

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

and, all rules $r_j \in R'$ for which no rule $r_i \in R$ exists, such that $r_j \lesssim r_i$ must not write (i.e. create or delete) any nodes or edges that are defined in type-graph T.

Proof. The above lemma is clearly an extension of Lemma 6.14. Hence we also prove the lemma by contradiction and following the same rationale we can safely assume that we have two paths $\pi \in \Pi(RS)$ and $\pi' \in \Pi(RS')$ and two states G_i and G'_j with $j \geq i$ where the paths diverge for the first time. Again, the steps $G_i \xrightarrow{r_i} G_{i+1}$ and $G'_j \xrightarrow{r'_j} G'_{j+1}$ must be performed through rules $r'_j \not\lesssim r_i$. We have three different possibilities for this situation to occur: (i) $r'_j \lesssim r_k$ with $(r_k, r_i) \in p$, (ii) $r'_j \lesssim r_k$ with $(r_i, r_k) \in p$, and (iii) $r'_j \lesssim r_k$ with $(r_k, r_i) \notin p \wedge (r_i, r_k) \notin p$[6].

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 is applicable to G_i, the application $r_i \to G_i$ will be preempted due to $(r_k, r_i) \in p$. However,

[6]This depicts a situation where both rules cannot preempt each other and is already discussed in the proof of Lemma 6.14

if $r'_j \to G'_j$ and $G_i \approx G'_j$ and $r_k \lesssim r'_j$ then by construction $r_k \to G_i$, which contradicts our assumption of divergent behaviour.

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, there 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, but this case has already been discussed in the proof for Lemma 6.14.

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

6.2.5 Application to *rigSoaML*

We will now apply the theoretical findings in the previous sections to the constituents of our modelling approach *rigSoaML*— i.e. to collaboration types, component types and system types. We will provide results for the combination of two rule-sets into one and show that the trace-sets that define the semantics of collaboration and component types are in a valid refinement relation if the collaboration or component types, respectively, are.

Combining rule-sets

In the previous subsection, we explained the semantical definition of refinement and gave sufficient syntactical criteria to guarantee the refinement. However, in some situations it is possible to further simplify the necessary checks. This is the case for the combination of two rule-sets, which are defined on disjoint type-graphs.

Corollary 6.19 (Combination of rule-sets)**.** *Given two complex rule-sets RS_1 and RS_2 with $RS_1 = (R_1, p_1, \mathcal{R}^1_u)$ and $RS_2 = (R_2, p_2, \mathcal{R}^2_u)$, which 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, it holds that $RS_1 \sqsubseteq_C RS$ and $RS_2 \sqsubseteq_C RS$.*

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. Hence, 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 6.18. □

As a consequence of Corollary 6.19 we get that, if $RS_1 \models \phi_1$ and $RS_2 \models \phi_2$, with ϕ_1 being typed over TG_1 and ϕ_2 being typed over TG_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-sets we want to combine are not defined over strictly disjoint type-graphs, but are "pseudo-type separated", Corollary 6.19 cannot be applied.

Definition 6.20 (Pseudo-typed graphs)**.** *Let P be a set of nodes called pseudo-types. A graph G is pseudo-typed iff each node in $V_G \setminus P$ 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.*

Pseudo-type separated rule-sets are rule-sets whose rules are pseudo-typed as in Definition 6.20. The pseudo-types have the same effect on the rule-sets as ordinary types have, and thus the combination of pseudo-type separated rule-sets has the same properties as those of the combination of rule-sets defined above disjoint type-graphs.

Corollary 6.21 (Combination of pseudo-typed rule-sets)**.** *Given two complex rule-sets $RS_1 = (R_1, p_1, \mathcal{R}_u^1)$ and $RS_2 = (R_2, p_2, \mathcal{R}_u^2)$, which rules in R_1 and R_2 are pseudo-typed over disjoint pseudo-types P_1 and P_2 with $P_1 \cap P_2 = \emptyset$, 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 complex rule-set RS, it holds that $R \sqsubseteq_C RS_1$ and $R \sqsubseteq_C RS_2$*

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 $P_1 \cap P_2 = \emptyset$. Therefore, 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 6.19 and get the desired results. □

Refinement of Collaboration and Component types

For two collaboration types C and C' where C' refines C, we have to translate this refinement relationship into our semantic domain. The informal understanding of the refinement relation between collaboration types is that the refining collaboration type C' is allowed to do anything the refined collaboration type C does, but nothing more in terms of the refined collaboration type's type-graph. The corresponding informal translation into our semantic domain would be: for each trace $t \in [\![C']\!]$ we can find a trace $t' \in [\![C]\!]$ such that t' refines t.

Corollary 6.22 (Collaboration type refinement)**.** *Given two collaboration types C and C', if C' refines C then $[\![C']\!] \overset{*}{\prec} [\![C]\!]$ holds.*

Proof. The collaboration types C and C' can be either concrete or abstract. If both are abstract, we do not have any rules for either of the two collaboration types, but we know from Definition 4.11 that the property $\phi_{C'}$ of collaboration

type C' implies property ϕ_C of collaboration type C. For abstract collaboration types, the semantics $[\![C]\!]$ and $[\![C']\!]$ are given as all traces that satisfy the property ϕ_C or $\phi_{C'}$, respectively. For the type-graphs of the collaboration types, it holds that $T_c \leq T_{C'}$. Thus, for each trace $t' \in [\![C']\!]$, it holds that $t'_{|T} \in [\![C]\!]$. This gives us the condition as required by Definition 6.7.

For C being an abstract and C' being a correct concrete collaboration type, again $\phi_{C'} \implies that \phi_C$ has to hold, and we have verified that $R'(C') \models \phi_{C'}$ holds. The trace set $[\![C']\!]$ is given as $T(R')$ and thus for each trace $t' \in T(R')$ it holds that $t' \models \phi_{C'}$. The restricted trace $t'_{|T}$ is again a trace of the abstract collaboration type C and thus we have a valid refinement.

If the correct concrete collaboration type C' refines the correct concrete collaboration type C, then for the two rule-sets R_C and $R_{C'}$ $R_{C'} \sqsubseteq_C R_C$ holds. Then Lemma 6.14 directly gives us the necessary conditions of Definition 6.7 such that $[\![C']\!] \overset{*}{\prec} [\![C]\!]$ holds. □

Due to the nature of abstract component types, which can implement abstract as well as concrete roles, even if the component itself does not yet specify any concrete behaviour, some partial aspects of the behaviour can already be defined.

Corollary 6.23 (Component type refinement)**.** *Given two component types C and C' such that C' refines C, then $[\![C']\!] \overset{*}{\prec} [\![C]\!]$ holds.*

Proof. Only abstract component types can be refined. Therefore C has to be an abstract component type. C' can be either abstract or concrete. In general, each component type implements roles of multiple collaboration types. Let $Cols = (Col_1, Col_2, hdots, Col_n)$ and $Cols' = (Col'_1, Col'_2, \ldots, Col'_k)$ be two sets of collaboration types with $k \geq n$, that define roles which are implemented by C and C', respectively. Without loss of generality, we further assume that each of the first n collaboration types is either the same $Col = Col'$ or that $Col' \sqsubseteq Col$ holds.

For each trace $t' \in [\![C']\!]$, we can find a mapping function $I'_{Col'_j} : \mathbb{N} \mapsto \mathbb{N}$ with $1 \leq j \leq n$ and $\forall l, m \in \mathbb{N} : l < m \implies I'_{Col'_j}(l) < I'_{Col'_j}(m)$, such that the trace $t'_{Col'_j}$ with $t'_{Col'_j}(i) = t'(I'_{Col'_j}(i))$ refines a trace $c'_{Col'_j} \in [\![Col'_j]\!]$. In the case that $Col'_j \sqsubseteq Col_j$, we know from Corollary 6.22 that a trace $c_{Col_j} \in [\![Col_j]\!]$ has to exist such that $t'_{Col'_j} \prec c'_{Col'_j} \prec_{\mathrm{ord_{Col_j}}} c_{Col_j}$. If $Col'_j = Col_j$,this trace trivially exists. As the component type C is abstract and thus only describes in a declarative way which traces are in $[\![C]\!]$ (cf. Definition 4.15), and although the traces have to conform with the behaviour specified by the collaboration types in $Cols$, there must exist a trace $t \in [\![C]\!]$ such that $t \prec t'_{Col'_j}$ for all $Col'_j \in Cols'$.

The same technique as for the collaboration types can be applied to the components' internal behaviour and we can thus construct a trace $t'_{C'}$ using a mapping function $I'_{C'}$ for refining component type C' and a trace t_C for the refined abstract component type C. Recalling the arguments from Corollary 6.22 yields that $t'_{C'} \prec t_C$. The trace t_C can also be found in a trace $t \in [\![C]\!]$.

In a last step, we combine the mapping functions $I'_{C'}$ and $I'_{Col'_j}$ or $I'_{Col'_j} \circ \mathrm{ord_{Col_j}}$, respectively, to a mapping function ord that relates each state $t(i)$ to a state

$t'(ord(i))$. By construction, $t'(ord(i))_{|T_C}$ is isomorphic to $t(i)$. According to Definition 6.5, this yields $t' \prec_{\text{ord}} t$. We can construct the mapping function ord for each trace $t' \in [\![C']\!]$ and thus find a refined trace $t \in [\![C]\!]$. Hence we have shown that $[\![C']\!] \overset{*}{\prec} [\![C]\!]$ holds. □

6.3 Automatic Refinement Checking

So far we have given a clear syntactical definition of rule-set refinement and the conditions that have to additionally hold for refinement of complex rule-sets. The necessary algorithms are still missing. In this section we will introduce the required steps to algorithmically decide whether or not two rule-sets refine each other. With Lemma 6.14 and Lemma 6.18 we have three clear criteria that are sufficient to show rule-set refinement. These are (i) syntactical rule refinement (ii) preservation of preemption and (iii) satisfaction of the preempt and the urgent predicate.

Rule refinement (i) between two rules r and p with $p \lesssim r$ can be shown by computing two matches $m_l : L_r \to L_p$ and $m_r : R_r \to R_p$. Further, we have to show that the elements of p which are not in the co-domain of m_l and m_r and are written by p are typed over a type-graph TG' that extends the type-graph TG, which defines the types used in rule r. For a refined rule-set without preemption and urgent predicates, we are done after this check (see Lemma 6.14). However, for complex rule-sets that are to be refined, this is not sufficient.

To check the preservation of the rules' preemption relation (ii) in two complex rule-sets (cf. Definition 6.15) $P = (R_P, p_P, \mathcal{R}_u^P)$ and $Q = (R_Q, p_Q, \mathcal{R}_u^Q)$, we iterate through the entries of p_P and check that for each entry $(r_i, r_j) \in p_P$ there exists an entry $(r'_i, r'_j) \in p_Q$ for all $r'_i \lesssim r_i$ and $r'_j \lesssim r_j$.

The remaining check is to check that the preempt and the urgent predicates hold. In the following, we will present a technique that works on graph-transformation rules directly and reuses capabilities of the Invariant Checker.

Checking the preempt predicate

The two predicates *Preempt* (see Definition 6.16) and *Urgent* (see Definition 6.17) state that two refining rule-sets (cf. Definition 6.10) ensure that whenever a preempting / urgent rule r in the refined rule-set is applicable there must be a refining rule r' in the refining rule-set that is applicable too, if a preempted rule r'_p is also applicable. The last restriction can be safely added for the preempt case, as there the rules don't have to be applied. In the following, let us assume that $S = (R, p, \mathcal{R}_u)$ and $S' = (R', p', \mathcal{R}'_u)$ are two rule-sets, such that $S' \sqsubseteq_C S$ and $r_1, r_2 \in R$ are graph-transformation rules with $(r_1, r_2) \in p$. Each of the rules $r_1^1 \dots r_1^n \in R'$ refines r_1 and each of the rules $r_2^1 \dots r_2^m$ refines r_2 and all rules $r_1^1 \dots r_1^n$ preempt the rules $r_2^1 \dots r_2^m$.

For the predicate preempt to be satisfied, we have to check that, in all situations where the rule r_1 is applicable and any of the preempted rules r_2^i with $1 \leq i \leq m$ is applicable too, there has to exist a rule r_1^j with $1 \leq j \leq n$ that can be

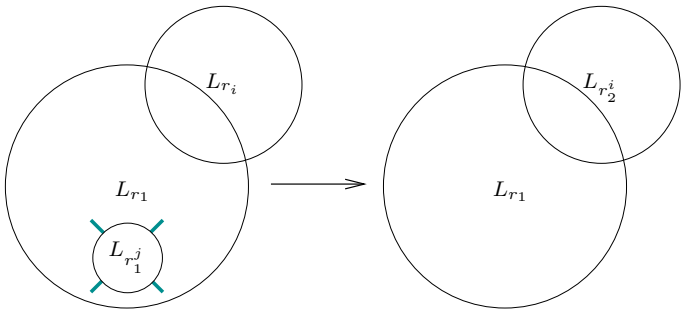


Figure 6.9: Sketch for the verification of predicates

applied. For the Invariant Checker we have already described in Section 6.1.2 how patterns can be constructed that represent specific situations. We use this technique again to construct patterns that stand for graphs in which the graph- transformation rule r_1 and any of the graph-transformation rules r_2^i with $1 \leq i \leq m$ are enabled. Obviously, we therefore have to build the overlap between two left-hand sides and not, as we did for the Invariant Checker, a right-hand side and a forbidden pattern. The next step is to check whether any of the rules r_1^j with $1 \leq j \leq n$ is applicable in the constructed pattern. This would effectively prevent the graph-transformation rule r_2^i from being applied, as $(r_1^j, r_2^i) \in p'$ holds. The check for applicable rules is also implemented for the Invariant Checker but only for the source graph pattern, which is constructed through reverse application of the graph-transformation rule. Therefore, we do the following: i) we remove all side-effects from the graph-transformation rule r_1, and this rule is referred to as r_1^{id}; ii) for each graph-transformation rule r_2^i with $1 \leq i \leq m$ we introduce a forbidden pattern $f_2^i \equiv Appl(r_2^i)$[7] that is equivalent to the graph constraint, encoding r_2^i's applicability. Now we can use the Invariant Checker to verify that the graph-transformation rule r_1^{id} satisfies the inductive invariant $\Phi_{\mathcal{F}}$, with $\mathcal{F}$ being given as the set $\mathcal{F} = \{f_2^i | 1 \leq i \leq m\}$. The rules r_1^j with $1 \leq j \leq n$ are used to check the applicability of the rule r_1^{id} in the source graph pattern. Figure 6.9 shows a sketch of the modified Invariant Checker.

If the Invariant Checker can construct a transition from the source to the target graph pattern, the *Preempt* predicate is not fulfilled. The reason for this is that in the source graph pattern none of the rules that refine r_1 is applicable but in the target graph pattern the rule r_2^i is applicable. It follows that, in the refining rule-set R' the rule r_2^i is not always preempted if it is required. However, the Invariant Checker might construct the counterexample due to the source graph pattern's incompleteness. In some cases, a graph-transformation rule r_1^j might only partially match the source graph pattern. In this situation, the Invariant Checker is obviously unable to find a satisfactory match and thus reports a counterexample. But, given the cardinality constraints in the type graph and corresponding meta-model, we are able to extend the source graph pattern. This extension of the source graph pattern is always possible if the minimal cardinality constraints are not satisfied. I.e. when a RequestOfferCol-

[7]Please recall that *Appl*() maps a graph-transformation rule to a graph constraint that exactly matches all graphs where the graph-transformation rule is applicable

laboration is required to be connected to a Supplier, if this constraint is not fulfilled, we can enhance the source graph pattern accordingly. Further, we can use knowledge we have gained from the verification of inductive invariants on the rule-set R'. Let us assume we have previously verified that the graph constraint $C = (\exists P, \bigwedge_{i \in I} N_i)$ (cf. Definition B.4) is an inductive invariant of the rule-set R'. Then we can conclude that it is impossible that a correct graph – i.e. the source graph pattern has to be a valid graph – contains a match for P without at least one match for the negative application conditions N_i. It follows that it is safe to enhance the source graph pattern with any of the N_i. To do this systematically, we can introduce some enhancement rules that create any of the N_i whenever the graph constraint matches in the source graph pattern.[8] For each constraint C we can construct a $|I|$ enhancement rule, where rule R_{C_i} is given as $R_{C_i} = (P, N_i, P, id_P, n_i, A_i^-)$ (cf. Definition B.5) with P being defined as in C, N_i is a negative application condition of C with $P \xrightarrow{n_i} N_i$, and A_i^- is the set of C's negative application conditions. Note that this set of rules is too small. The rules currently create the constraint's NACs completely from scratch, without reusing parts from the SGP. For a comprehensive and fully automated approach, these completions have to be considered too. However, for the understanding of the approach, it is sufficient to know that we can create a set of rules R_C for any graph constraint C. Each of the rules R_{C_i} is applicable simultaneously. This can be easily seen, as they all share — by construction — the same left-hand side. And all of the rules in R_C are in conflict with each other, as for one match of rules' left-hand side, only one of the rules is applicable. To enhance the given source graph pattern algorithmically, we

- Build the rule-set R_C for each constraint C, as described above
- Let the source graph pattern be the initial graph G_0. We select an applicable rule-set R_C and apply each of its rules to G_0 yielding $|R_C|$ new graphs.
- For each of the new graphs we repeat the second step

Figure 6.10 shows a sketch of the labelled transition system that is created by the enhancement algorithm. All transitions starting in the same graph belong to the same rule-set R_C. The predicate is satisfied if we can find on each path, starting in G_0, a graph where a graph-transformation rule r_1^j is applicable. Therefore, it is important to note that whenever one rule belonging to a rule-set is applied, the remaining rules contained in that rule-set have to be applied too.

If the Invariant Checker, however, cannot construct a transition from the source to the target graph pattern, we know for sure that the *Preempt* predicate is satisfied. The Invariant Checker in the modified variant does not check whether or not the source graph pattern contains any other forbidden pattern (cf. Figure 6.9), and the transition does not exist due to the applicability of any of the rules r_1^i. This is exactly the behaviour we require for the rule-sets R and R' to satisfy $Preempt(R, R')$.

[8] Note that there is a difference between whether we can find a match in the source graph pattern if we interpret the source graph pattern as a graph or if we can a find a match for a graph constraint if we interpret the source graph pattern as a graph constraint.

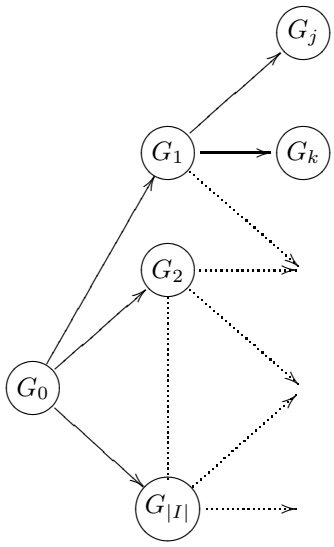


Figure 6.10: Sketch for the enhancement algorithm

Urgent predicate

The $Urgent$ predicate differs slightly from the $Preempt$ predicate. The $Preempt$ predicate, as discussed above, requires the rules r_1^i to be applicable only if any of the rules r_2^j is applicable too. For rules that are defined as urgent rules, this is not satisfactory, as urgent rules are used to specify mandatory behaviour, hence any of the rules r_1^i has to be applicable whenever the rule r_1 is applicable. The applicability of r_1 implies the applicability of any of the rules r_1^i. We achieve this by simply omitting the overlap operation and applying the enhancement algorithm to generate possibly missing context for the source graph pattern.

Discussion

In this section we presented a way to use a slightly modified variant of the Invariant Checker to verify the predicates $Preempt$ and $Urgent$. The necessary modifications are minimal and can be implemented without difficulty in the Invariant Checker algorithm, as only checks were omitted and no additional checks were required. However, the presented approach does not finally decide whether or not the predicates are satisfied. This happens due to the generated minimal context. Therefore we presented a way to enhance the generated source graph pattern and hence increase the likelihood of finding a match.

Alternatively, we could have used a mapping of graph constraints and graph-transformation rules to boolean logic and then used a SAT solver to find a satsifactory valuation. However, even SAT solvers suffer from a similar problem to that of our Invariant Checker. If the universe's size is too small, the SAT solver cannot find a suitable match for a graph-transformation rule.

6.4 Summary

In this section we showed a way in which the abstract operators $\sqsubseteq_A$ and $\models$ could be filled out with a concrete meaning. For the refinement, we first presented a semantical notion of refinement, which is built on the characteristics of our semantic domain, traces of HGTS. The definitions we gave in Section 6.2.3 were sufficient to show that our notion of refinement fulfils the properties of the abstract refinement operator $\sqsubseteq_A$. In the following section, we gave some additional arguments, mainly based on the syntactical properties of StoryPattern and hence graph-transformation rules that could be used to simplify the check on whether a refinement relation holds. However, due to the expressiveness of our modelling approach, the pure syntactical check is not sufficient and we had to accompany it with two predicates *Preempt* and *Urgent* that have to be satisfied by the refined and the refining rule-set. These predicates, however, could be checked using our Invariant Checker verification approach for HGTS, that we presented in Section 6.1. Having syntactical properties that guarantee refinement is beneficial for the software engineer, who has clear guidelines as to which modification is supported and which possibly violates refinement.[9]

The second big part of this chapter addressed the verification of safety properties. We presented our verification technique Invariant Checker,which is able to statically verify HGTS. The whole expressiveness of $\mathcal{L}$ is too strong to verify with the Invariant Checker approach and therefore we presented a restricted, but therefore verifiable, subset $\mathcal{L}^-$. For this subset, we showed how we could verify the inductive state invariants of a HGTS.

[9] We only showed that the syntactical refinement implies the semantical refinement. Nevertheless, it is possible that despite having a rule-set that violates our syntactical refinement rules, a refinement may be correct.

Chapter 7

Tool Support

Throughout the chapters 5 and 6 we have explained in detail how the verification scheme we are using looks and what the formal backgrounds are. The thesis' practical part comprises the implementation of a tool that allows us to perform the described verification steps. In this chapter we will describe the tool's implementation and the necessary extensions to verify the different types of supported graph-transformation systems.

The Invariant Checker is implemented as a plug-in for the Eclipse[1] IDE. In its current state, the Invariant Checker can automatically verify inductive invariants for discrete GTS and linear hybrid GTS and semi-automatically for hybrid GTS. For hybrid GTS, the Invariant Checker returns all possible counterexamples, which then have to be transformed into an appropriate input language for the hybrid model checker that is used.

7.1 General Architecture

The Invariant Checker is implemented as a Pipe and Filter architecture [121]. This architectural style uses filters, which are the parts that actually perform the computation, and pipes, which are used as buffers and communication channels between the filters. For the Invariant Checker we decided that each filter is a thread of computation and the pipes are used to synchronise the filters. The pipes ensure that a filter that wants to read from an empty pipe or write into a full pipe is blocked until the operation can be safely executed. This architecture thus effectively constrains the amount of memory that can be consumed by the Invariant Checker[2]. For the special case of filters that either read or write elements from pipes, we use the special terms Consumer and Producer. Each filter only knows the pipes it reads from and writes to. Which filter is responsible for the computation of another filter's input is irrelevant for the filter. Hence, it is easily possible to change the pipe and filter architecture's structure, e.g. by

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

[2] Obviously, it is still possible that a single filter consumes a lot of memory space and thus we cannot give an upper bound on the memory demands of the algorithm.

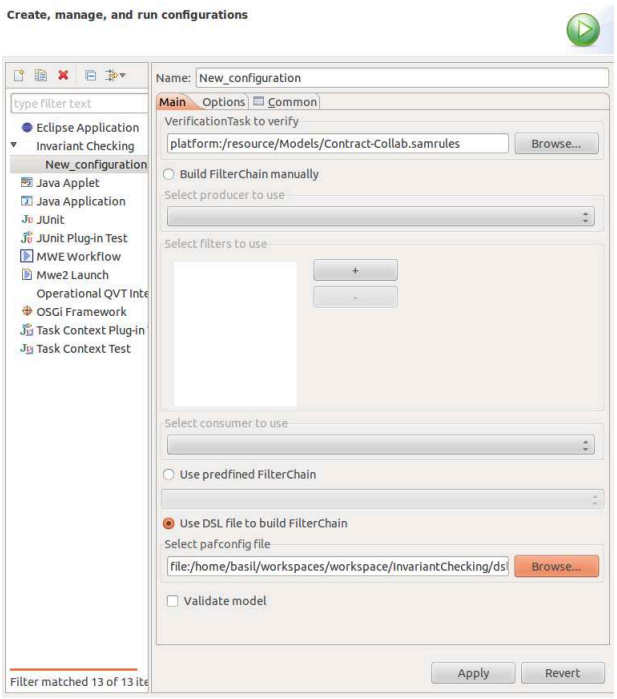


Figure 7.1: Screenshot of the Invariant Checker's launch dialogue

adding an additional filter, without the need to change existing filters, as long as the data elements that were written to the pipes did not change. Hence the only coupling that exist between two consecutive filters is of the data type of the exchanged data-elements.

In Figure 7.1 a screenshot of the Invariant Checker's launch dialogue is depicted. The dialogue mainly requires the user to specify the GTS to be verified and the pipe and filter configuration that should be used for the verification. Therefore the user has three possibilities, either to build a configuration just for this launch directly in the dialogue, or to use a predefined configuration, or - for more complex situations – to specify the configuration via a DSL. For details on the DSL, please refer to [61].

7.2 Verifying discrete GTS

In this section we will shed some light on the algorithm for the verification of discrete GTS. The algorithm is introduced in Section 6.1.2. In our publication [2], we presented a first implementation, which was not built using the pipe and filter architectural style. The algorithm behind Invariant Checker has been split up into different filters that are all executed in parallel. Put simply, the implemented filters solve the following tasks:

Generation of pairs The first filter in the pipe and filter architecture creates all required pairs of graph-transformation rules and forbidden sub-graphs.

Generate sub-graphs For the reverse application of the graph-transformation rule r, we have to know all matching sub-graphs between r and the sub-graph. Therefore it is essential that we get all sub-graphs.

Match sub-graph This filter performs the matching computations between the rule r's sub-graph, which has been previously generated, and the current forbidden sub-graph. The result of this filter is a set of matchings.

Merge graphs The result of this filter is the target graph pattern that is the outcome of the merge-operation with the graph-transformation rule, the forbidden sub-graph and the found match.

Rule application This filter applies the rule and checks whether or not the rule application is correct, i.e. whether all NACs are satisfied. If the rule application was not correct

Check Properties this filter investigates whether the source graph pattern contains any forbidden pattern or not. If no match is found, the current element is passed on.

Check Rules This filter checks for preempting rules that are applicable to the source graph pattern. If no match exists, the current element is passed on.

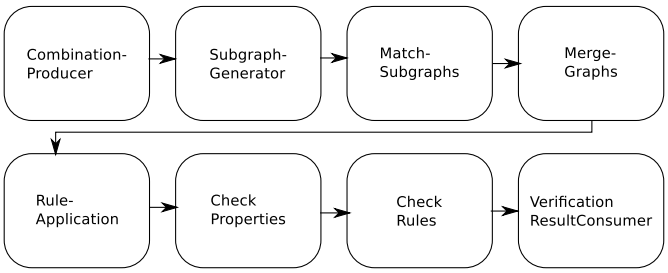


Figure 7.2: The configuration for the Invariant Checker for the verification of discrete GTS

The configuration of the Invariant Checker for the verification of discrete GTS is depicted in Figure 7.2. The input is a GTS including the safety properties as forbidden patterns. Each box stands for one filter, and the arrows stand for the pipes between the filters.

The implementation of the Invariant Checker algorithm for discrete GTS was part of my Bachelor's Thesis at the University of Paderborn.

7.3 Verifying hybrid GTS

For the case of a hybrid GTS, the above filters have to be modified, as we now have to check that a match of a forbidden pattern or a hybrid graph-transformation rule exists and that the attribute conditions are satisfied. Therefore, in the hybrid case we use slightly modified filters that compute all possible matches and pass the result on to the next filter. We will further distinguish

between linear hybrid systems, which are hybrid systems as described in Section B.3, but the continuous attributes only have fixed linear derivations and non-linear hybrid systems.

7.3.1 Linear hybrid GTS

Linear hybrid GTS are a subclass of hybrid GTS for which we have shown in [5] how they can be checked with the use of a solver for linear inequalities. The verification of linear hybrid GTS can obviously be used to verify timed GTS too. Timed GTS are a subclass of linear hybrid GTS, where the derivation of each continuous attribute is 1. Consequently, the continuous attributes are then called clocks. The necessary changes in comparison with the verification of discrete GTS have already been discussed in Section 6.1.3. In summary, we cannot omit a possible counterexample because we have found a matching forbidden pattern or a matching rule that preempts the counterexample's rule. We have to collect this information and use it to build a suitable system of linear inequalities and try to solve it. Therefore the ordered list of filters for the checking of linear hybrid GTS looks as follows:

Hybrid Check Properties This filter investigates whether or not the source graph pattern contains any forbidden pattern. The current element, together with all found matches, is passed on.

Hybrid Check Rules This filter checks for preempting rules that are applicable to the source graph pattern. The current element, together with all found matches, is passed on.

Check Urgent Rules This filter checks whether urgent rules are applicable in the target graph pattern. The current element with all found matches is passed on.

Check Constraints This filter builds a system of linear inequalities based on the results of the previous filters and the current element. If the inequalities can be solved, the current element is passed on as a counterexample. To solve the linear inequalities, we use the solver CPLEX.[3]

Figure 7.3 shows the change we made to the Invariant Checker's configuration in order to be able to verify hybrid linear GTS. The filters that are depicted in a doubly framed box have been newly added to the filter configuration.

7.3.2 Non-linear hybrid GTS

The verification of non-linear hybrid GTS differs only slightly from the verification of linear hybrid GTS, as described in the previous section. The main difference is that the solver we used for linear hybrid GTS is not able to verify constraint systems other than linear inequalities. However, in Section 6.1.3 we have already shown that the verification task can be mapped to a hybrid automaton. For this automaton, we can ask a suitable model checker whether the state "failure" (cf. Figure 6.4) is reachable. Thus, we only have to replace the

[3] www.ibm.com/software/products/us/en/ibmilogcpleoptistud/

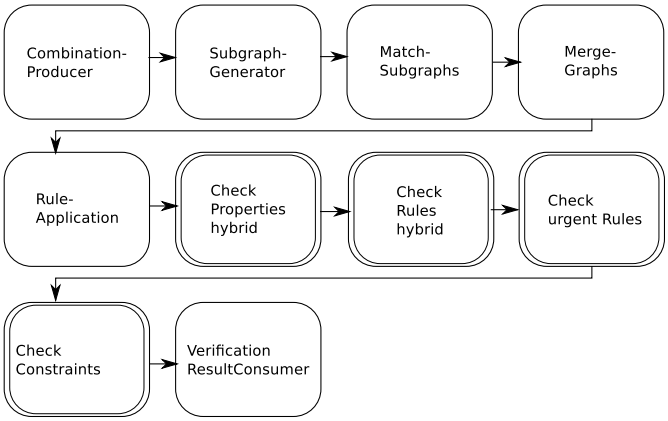


Figure 7.3: Configuration of the Invariant Checker for the verification of linear hybrid GTS. Doubly rounded filters are new.

last filter from the chain of filters we used for the verification of linear hybrid GTS.

Check non-linear Constraints This filter builds a hybrid automaton based on the results of the previous filters and the current element and invokes a hybrid model checker to check whether the state "failure" is reachable. If the state can be reached, the current element is passed on as a counterexample.

In Figure 7.4 we depict the Invariant Checker's configuration for the verification of hybrid GTS. The invocation of the hybrid model-checker is part of the "Check non-linear Constraints" filter.

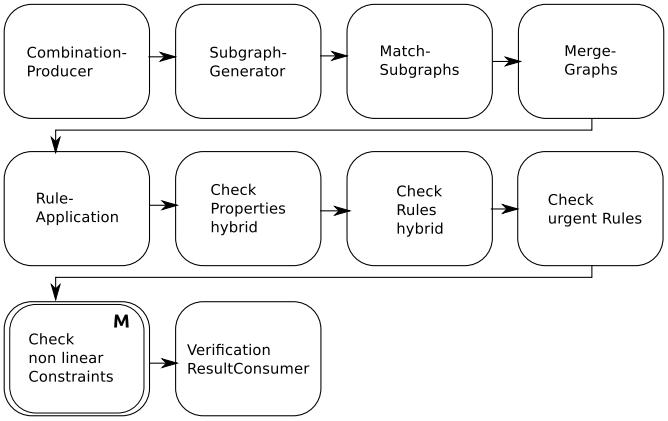


Figure 7.4: Configuration of the Invariant Checker for the verification of non-linear hybrid GTS

For the verification of the hybrid automata, we use the hybrid model-checker PHAVer [74]. The automatic invocation of PHAVer from the Invariant Checker, including the generation of the required automata using the guards, state-

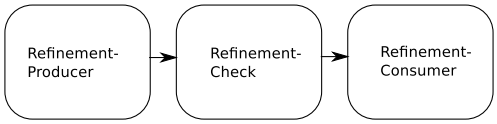


Figure 7.5: Configuration for the syntactical refinement check

invariants and flow-conditions that have been computed by the Invariant Checker has not yet been implemented. Therefore, we marked the "Check non-linear constraints" filter in Figure 7.4 with an "M" to indicate that this step currently has to be done manually.

7.4 Refinement Checks

Depending on the rule-sets used, it is either sufficient to check the syntactical refinement only or the predicates *Preempt* and *Urgent* have to be satisfied too. In the following, we will show how these checks could be implemented.

7.4.1 Syntactical Refinement Check

The rule r_2 syntactically refines rule r_1 if we can find a match for r_1's left- and right-hand side in r_2's left- and right-hand side, respectively (cf. Definition 6.9). The "Refinement Producer" looks for pairs of refining rules and passes them on. The algorithm for computing a match is already part of the Invariant Checker and had thus been reused in the "Refinement Check Filter" (see Figure 7.5). The "Refinement Consumer" returns pairs of rules that violate the syntactical refinement definition.

Refinement Producer Searches for pairs of refining rules and passes them on.

Refinement Check Tries to compute a match between the rules' left-hand sides and right- hand sides. If no matches can be found, the current pair is passed on.

Refinement Consumer Returns those rule pairs that violate the syntactical refinement definition.

7.4.2 Checking Urgent and Preempt Predicates

In Section 6.3 we showed that the checking of the predicates *Urgent* and *Pre empt* could be done using the Invariant Checker. However, the two involved GTS, i.e. the refined and the refining one, need some pre-processing. Let $S = (R^S, p^S, \mathcal{R}_u^S)$ and $P = (R^P, p^P, \mathcal{R}_u^P)$ be two rule-sets for which we want to show that $Preempt(S, R)$ holds. For every rule pair $(r_1, r_2) \in p^S$ — i.e. rule r_1 preempts rule r_2 — we will have to build a dedicated GTS G_{r_1,r_2}. The GTS's rule-set R_{r_1,r_2} contains the rule r_1^{id}, which is rule r_1 without deleting and creating any nodes or edges, and all rules $r_1^1, \ldots, r_1^n \in R^P$ that all refine

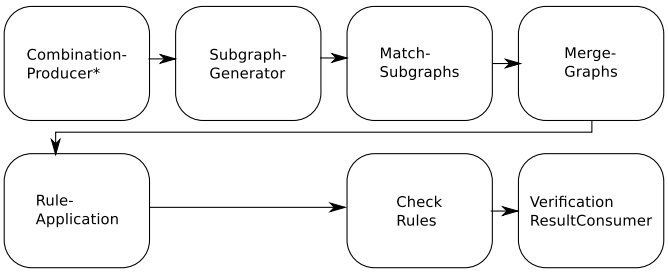


Figure 7.6: Configuration for the check of the *Preempt* and *Urgent* predicates

rule r_1. The set of forbidden patterns $\mathcal{F}$ is given through the set of graph constraints $Appl(r_2^1), \ldots, Appl(r_2^m)$ with $r_2^1, \ldots, r_2^m \in R^P$. All rules r_2^i refine rule r_2, and $(r_1^j, r_2^i) \in p^P$ holds for all $1 \leq j \leq n$ and $1 \leq i \leq m$. $Appl(r_2^i)$ is the translation of the rule r_2^i into a graph constraint that encodes the rule's applicability. To check the *Urgent* predicate, the construction of the set $\mathcal{F}$ is not necessary and thus can be omitted.

After the pre-processing, we yield a number of GTS, which we have to verify separately. However, we can use a more or less standard Invariant Checker configuration for either GTS or HGTS without the "Check Properties" or "Hybrid Check Properties" filter. Further, the only rule that has to be applied by the Invariant Checker is the rule r_1^{id}, and thus the "Combination Producer" that builds the pairs of graph-transformation rules and forbidden patterns is only allowed to combine this rule with the forbidden pattern. The other rules must only be used in the "Check Rules" filter or its hybrid counterpart. The "Check Properties" filter has to be removed as the rule r_1^{id} does not change the constructed graph pattern. Hence, the currently investigated forbidden pattern is also present in the source graph pattern and in consequence the "Check Properties" filter will remove the potential counterexample.

Summarising, with some pre-processing and minor modifications to the existing Invariant Checker configurations, we are able to reduce the checking of the *Preempt* and the *Urgent* predicate to the verification of inductive invariants using the Invariant Checker. In Figure 7.6, the modified configuration of the Invariant Checker is shown. As the "Combination Producer" had to be slightly modified, it is named "Combination Producer*". The pre-processing, however, is currently not implemented, although we have shown that it is conceptually feasible.

Chapter 8

Evaluation

In this chapter we will evaluate our modelling and verification approach with our running application examples, the Supply Chain System, and an additional example, the RailCab system.

8.1 Supply Chain System

The supply chain application example has been used throughout the course of this thesis. Therefore we will refrain from an explanation of the system, and start directly with some of the more complicated aspects concerning the modelling and analysis of the supply chain system.

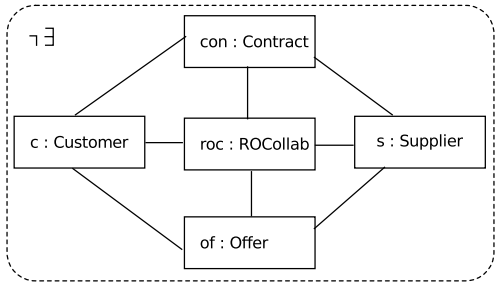
8.1.1 Modelling

The supply chain system we use as application examples consists — as already mentioned — of an abstract component AbstrFactory, the abstract service contract Contract, the concrete service contracts Auction and Request Offer Collaboration, and the concrete component Factory.

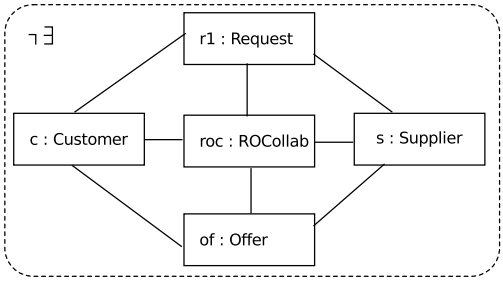
The abstract service contract Contract specifies a meta-model that consists of the elements Customer, Supplier, Contract and its own representative ContractCollaboration. Although the service contract is abstract, the Customer role already owns a behaviour specification, the createContract StoryPattern. At the same level of detail, we have specified the abstract component Factory, which mainly declares the properties we assume that a Factory should fulfil.

We have modelled two more service contracts that refine the abstract service contract Contract. These are the Auction and the RequestOfferCollaboration service contracts. The RequestOfferCollaboration service contract implements a basic challenge response protocol between the two service roles Customer and Supplier. The whole rule-set for the RequestOfferCollaboration is shown in Appendix A.2.3. For a correct and safe RequestOfferCollaboration, we require some safety properties to be fulfilled. Besides the safety properties that have already

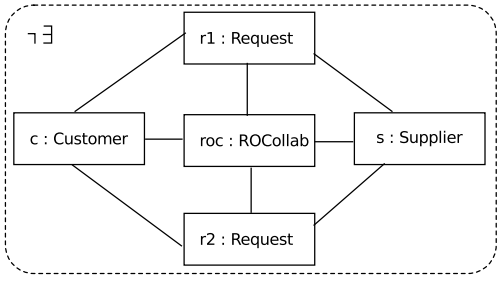
been specified for the abstract Contract collaboration, the RequestOfferCollaboration additionally requires several safety properties that had to be added due to our applied verification technique Invariant Checker. As we have explained in Section 6.1.2, the Invariant Checker approach mainly checks one step of GTS. However, dependencies between different rules, e.g. the `makeOffer` rule, can only be applied after the `sendRequest` rule, making it impossible for several situations to happen. If it is forbidden to create more than one Requests (cf. Figure 8.1(c)) between the same Customer and Supplier roles, it is impossible that a Request and an Offer (cf. Figure 8.1(b)) exist in parallel too. But this information is not available to the Invariant Checker and thus we had to add several forbidden patterns to express these dependencies between the rules. Figure 8.1 shows an excerpt from the additional forbidden properties that had to be added to the RequestOfferCollaboration specification. Note that the need to add these additional forbidden properties was not introduced by our verification approach in general, but by the specific technique, i.e. the Invariant Checker that we used to verify the collaboration and component types.



(a) noOfAndContract



(b) noReqAndOffer



(c) noTwoRequest

Figure 8.1: Additional forbidden properties for the RequestOfferCollaboration

Factory For the concrete component Factory, we want to specify that the component can only sign a Contract with her Supplier service role, if for all its Customer service roles valid Contracts have been signed. We can express this requirement through a graph constraint that forbids the Supplier role to sign a Contract if there still exists a Customer service role, which has not signed a Contract (see Figure 8.2). Obviously, we have to capture such situations in the component's implementation in order to avoid them. Thus, before allowing the component's Supplier service role to sign a Contract, we have to ensure that all Customer service roles have signed a valid Contract. The difficulty with this check is that we don't know in advance how many Customer service roles have been created for a Factory component. Consequently, we cannot add all Customer service roles to the rule's precondition.

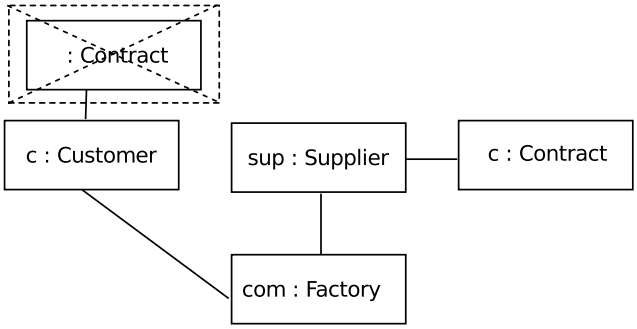


Figure 8.2: `supConwithoutCustCon` safety property

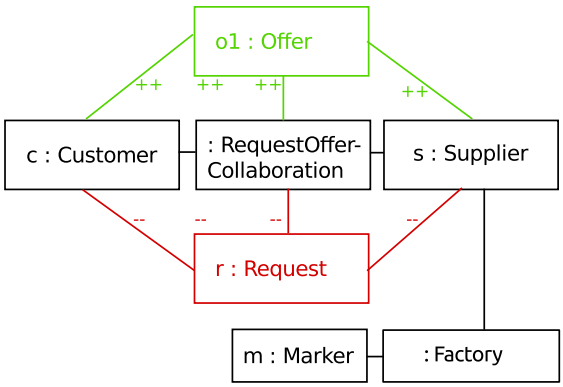


Figure 8.3: The `makeOffer` rule

A possible way to specify the necessary precondition is to use an NAC. We have to specify that no Customer exists that does not have a Contract signed. Unfortunately, this requires the capabilities of nested application conditions, which we currently do not support.[1] The solution we came up with for modelling the required behaviour, despite the mentioned limited expressiveness, is to use an

[1] Our modelling language for graph-transformation rules and graph constraints SaMiGra actually does support arbitrarily nested application conditions, but neither our Invariant Checker verification tool nor our Story Diagram Interpreter execution environment supports them at the moment.

auxiliary flag — internal to the Factory component — to mark states where all Customer service roles have a valid Contract and two rules for `createContract` together with preemption. The Supplier service role's `makeOffer` rule (see Figure 8.3) only creates the Offer if the marker flag is set. The marker flag is set by the rule shown in Figure 8.5. This rule is preempted by the rule shown in Figure 8.4. In consequence, as long as two Customer service roles exist that are still missing a Contract, the rule shown in Figure 8.4 is applicable. Otherwise, the rule in Figure 8.5 is applied, which then creates the marker flag that enables the `makeOffer` rule. The removal of the marker flag is done in the reverse way and again using two rules. One rule requires the existence of the marker flag and deletes it, while the second one forbids the existence of the marker flag. An alternative solution would have been to specify an urgent rule that is part of the Factory's implementation, and which immediately deletes the marker flag if one Customer service role exists without a Contract.

In Figure 8.6 we show an evolution diagram for the supply chain example that recapitulates all the concepts introduced and the dependencies between them.

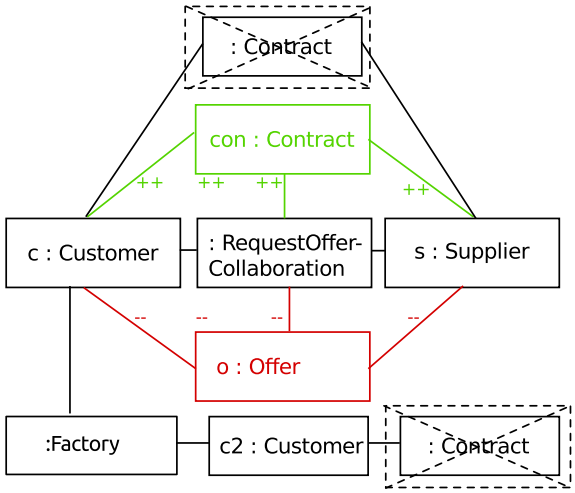


Figure 8.4: The Factory's preempting `createContract` rule

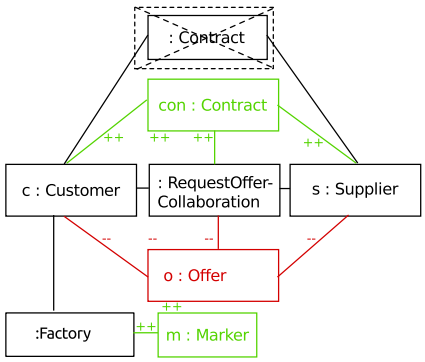


Figure 8.5: The Factory's preempted createContract rule

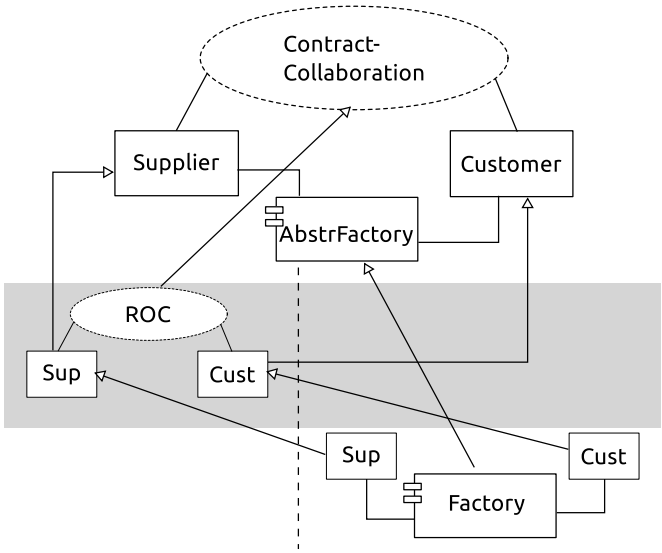


Figure 8.6: Evolution Diagram for the Supply Chain System

Refinement	Duration[ms]	successful?
$R_{ROC} \sqsubseteq_C R_{CC}$	49	✓
$R_{Fac} \sqsubseteq_C R_{ROC}$	75	✓

Table 8.1: Overview of the refinement checks' durations

8.1.2 Analysis

In the supply chain application example, refinement occurs at several positions. We have the abstract Contract service contract specification, which is refined multiple times. First, the abstract component AbstrFactory refines the roles Customer and Supplier, second the concrete collaboration type RequestOffer-Collaboration refines the ContractCollaboration collaboration type, third Factory refines both of the RequestOfferCollaboration's roles. We have introduced two different notions of refinement, one for complex rule-sets (see Lemma 6.18) and one for the rule-set (see Lemma 6.14). The refined rule-sets of the supply chain system are rule-sets without the use of urgent rules or preemption. Thus the pure syntactical conditions given in Lemma 6.14 are sufficient to guarantee a valid refinement. A complex rule-set is used in the Factory's implementation, where we need to be able to express preemption between the `createContract` rules (see Figure 8.4 and Figure 8.5). But this rule-set is not refined within our application example. For a valid refinement, it remains for us to show that the refining rules themselves refine the refined rules. This check has to be done two times.

In Table 8.1 we show the results of the two necessary refinement checks. We had to check that $R_{ROC}(Col_{Roc}) \sqsubseteq_C R_{Con}(Col_{Con})$ and $R_{Fac} \sqsubseteq_C R_{ROC}$ hold. A check between the rule-sets of the abstract component AbstrFactory and the concrete component Factory is not necessary, as for an abstract component no

rules have to be specified.

The supply chain system required several applications of the Invariant Checker. In Chapter 5 we have already pointed out which verification steps are necessary to show the system's safety. We first showed that we have to verify that the instances of the collaboration types are correct (cf. Example 11), then we had to check that the component types are correct (cf. Example 12), and finally we had to verify that the complete system consisting of collaboration and component type instances is correct (cf. Example 13). The mentioned examples only listed the necessary checks, but showed no data on the actual verification. In the following table 8.2, we present the run-time data such as the number of rules (#R) and forbidden patterns (#P) that had to be checked, the duration of the verification run (d. [ms]), generated sub-graphs (#Sub), the number of pairs (#Pairs) and the number of hybrid checks (#LIE), together with the duration. For all checks, the memory consumption of the Invariant Checker was less than $512MB$.

Task	#R	#P	d. [ms]	#Sub	#Pairs	#LIE
$R_{CC}(Col_{CC}) \models \Phi_{CC}$	4	1	1113	30	34	0
$R_{ROC}(Col_{ROC}) \models \Phi_{ROC}$	6	5	5703	758	905	0
$R_{Fac}(Com_{Fac}) \models \Psi_{Fac}$	8	15	12995	2053	1689	95

Table 8.2: Detailed evaluation results for the concrete component and collaboration types

During the verification run of the Factory component type, the Invariant Checker constructed an interesting counterexample, which it wasn't able to discard. The counterexample was constructed by overlapping the `deleteOtherContract` rule (cf. Figure A.57) and the `offerButNoCustCon` forbidden property (cf. Figure A.66) with only the Supplier node. The resulting source graph pattern is depicted in Figure 8.7 and it can be seen that the Supplier role has an adjacent Offer and Contract object. However, the RequestOfferCollaboration forbids the existence of an Offer and a Contract in parallel for one Supplier role (cf. Figure A.45). The Supplier is either part of two RequestOfferCollaborations, which is not allowed, or the Offer belongs to the RequestOfferCollaboration present in the source graph pattern. In this case, the aforementioned safety property would be violated. Hence, it is safe to discard the witness. The reason for which the Invariant Checker was unable to discard the witness automatically is that the source graph pattern contains too little context around the Offer node. But, as we have explained above, any safe completion of the source graph pattern yields a violation of at least one of the specified safety properties.

Comparison with GROOVE A tool that is commonly used to verify and investigate the nature of graph-transformation systems is GROOVE [117]. We used GROOVE to build up the state space of various supply chain networks consisting only of the Factory implementations. We were not able to build a GTS in GROOVE that is as expressive as the model we presented in this thesis, due to the fact that the standard implementation of GROOVE does not support

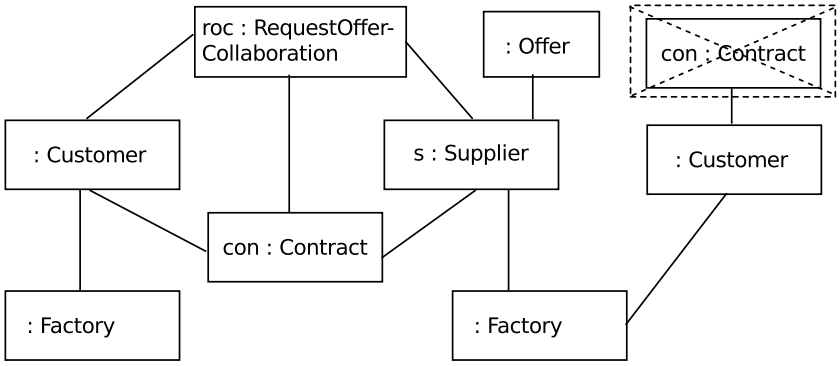


Figure 8.7: Source graph pattern of the witness found by the Invariant Checker

time.[2] Thus, the model's behaviour we checked was an over-approximation of the behaviour we specified in this thesis. The size of the computed state space is shown in Table 8.3.

Factories in initial graph	# states	# transitions
2	5	6
4	198	678
8	136928	1156925
9	672883	6510041
10	≥2097733	≥22021912

Table 8.3: Size of the state space as computed by GROOVE

The initial graphs were all based on the same schema. They consisted of one Factory node that only had a Supplier role and the Marker node set, but no Customer role, and — depending on the initial graph's size — several copies of a triple consisting of Factory, Customer role and Supplier role. The rule-set we used was complete in the sense that we specified the structural part of our rules in GROOVE but omitted the jump conditions. We further simplified the model for GROOVE by omitting the rules that can create new Factory instances. For an initial graph with 10 Factory instances, GROOVE ran for 2064 minutes and consumed 8GB of memory without finishing the state space exploration. We thus stopped GROOVE and gave lower bounds for the number of the found states and transitions.

The simplifications we had to make in order to adapt our model to GROOVE's expressiveness yielded a system that could not be verified. I.e. the Invariant Checker could verify the supply chain system due to the interplay of the jump conditions and the attribute conditions of the safety properties. But as we had to omit these for GROOVE, the verified model no longer satisfies the safety properties. Therefore, we only generated the state space instead of using GROOVE's verification capabilities also. The state space generation gives a sufficiently exact impression of a verification in GROOVE, as a system's complete state space has

[2] Attributes of different types are supported, but their value is only changed through rule applications and does not change continuously.

to be built up in order to decide whether or not the system is correct. Obviously, the initial states we used were highly symmetric, and probably graph-shaped abstractions (see [37]) could have solved this problem for the initial states we used, but in a more realistic scenario it is unlikely that only factories of the same type would be used. Consequently, abstraction cannot play off their capabilities to the full extent.

8.2 RailCab System

The RailCab system has been developed at the University of Paderborn as a research prototype. The project aims at establishing a new railway system that is built atop of small autonomous shuttles instead of long and tightly scheduled trains. Users can book a shuttle or a seat in a shuttle for exactly their desired connection. The shuttles then challenge each other to provide the best (with respect to duration and costs) offer for the demanded connection. As shuttles are autonomous vehicles, they are bound to a schedule and can freely decide where to go and which route to take. Compared with traditional trains, the energy of consumption of a single shuttle is enormous in relation to the passengers aboard. The reason for the high energy consumption is the wind resistance. However, the wind resistance can be effectively reduced if the shuttles build convoys. For these convoys, it has to hold that they do not require a mechanical connection, as this would contradict the shuttles' flexibility; the shuttles can travel at high speeds; and lastly, travel at close proximity to each other, otherwise the wind resistance will again be too high. For two or more shuttles driving in a convoy, strong and reliable coordination is necessary to avoid collisions. In this thesis, we will model the coordination protocol as a service contract that can be established between two shuttles.

In this thesis, we will introduce two different variants of the RailCab system, which differ in the degree of the detail in which the coordination protocol between shuttles is specified. The first variant is a purely timed system, i.e. all continuous behaviour is specified through clocks. In the timed variant, we assume that the system is safe if the coordination protocol is instantiated in time. The second variant also models the shuttles' physical movement, which is modelled through variables for acceleration, position and speed. This variant could be used to show that, using the coordination protocol, physical collisions could be avoided. For the timed variant, we again omit the control modes as we did for the supply chain system, as they do not change. However, for the hybrid variant the control modes are subject to change by the rules and thus we depict them. In [2] we showed a completely discrete version of the RailCab system, but this version will not be used in this thesis.

8.2.1 Modelling

The RailCab system is a system from a completely different domain than the supply chain system. The RailCab system is a safety critical and cyber-physical system, where the timely execution of the correct rules is necessary to guarantee a proper functioning of the system. In the RailCab system, the need to use

urgent rules is much stronger, as we have to enforce that a certain behaviour takes place. On the other hand, the variety of possibly existing different service contracts is probably not as great as in the supply chain example. In a first step, we will introduce the timed variant of the RailCab system and then the hybrid one.

Timed RailCab System

The RailCab system's structure is described through the ontology depicted in Figure 8.8. The ontology comprises the service role Shuttle which is used two times in the DistanceCoordination service contract, distinguished by the two associations front and rear. A Shuttle is placed at exactly one Track. The Shuttle service role has a clock timeAtTrack. This clock is reset each time the Shuttle moves to the succeeding Track.

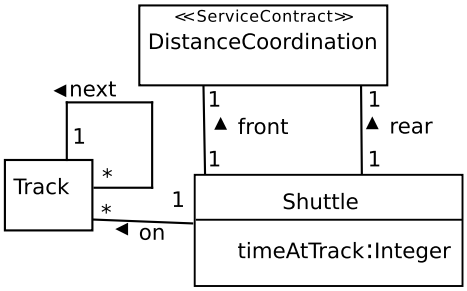
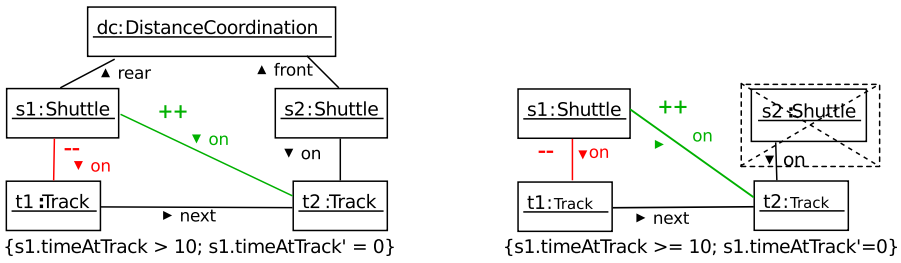


Figure 8.8: Ontology for the timed RailCab example

TimedDC collaboration type The Shuttle role's movement is described through two rules `moveDC` (see Figure 8.9(a)) and `moveSimple` (see Figure 8.9(b)). Both rules require that the Shuttle is for at least ten time units at the current Track before moving on. Each application of one of the move rules resets the timeAtTrack clock for the Shuttle. In contrast to the `moveDC` rule, the `moveSimple` rule forbids that there is another Shuttle at the next Track. The rule `moveDC` can only be applied if a DistanceCoordination collaboration has been instantiated. This is done using the `createDC` rule (see Figure 8.10). The `createDC` rule requires that the Shuttle is at the current Track for at least 3 time units. The creation of two DistanceCoordination collaboration instances between two Shuttles is prevented by the use of a NAC.

The timed RailCab System has to satisfy two safety properties. These are `noDC` and `collision`, which make it impossible that a DistanceCoordination collaboration has not been created and that a collision occurs. In the timed variant of the RailCab example, the only possibility of encoding a collision is that two Shuttles are located at the same Track and no DistanceCoordination collaboration has been instantiated (see Figure 8.11(b)). However, we require that the DistanceCoordination collaboration is instantiated at least 5 time units after the rear Shuttle and the front Shuttle are at neighbouring Tracks (see Figure 8.11(a)).



(a) The Shuttle role's `moveDC` rule (b) The Shuttle role's `moveSimple` rule

Figure 8.9: The Shuttle role's move rules

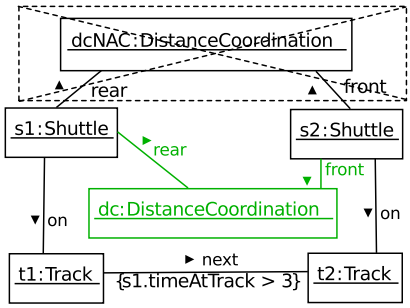


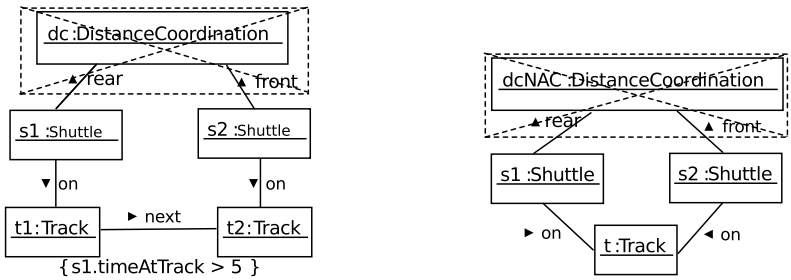
Figure 8.10: The Shuttle role's `createDc` rule

Formally, the timed RailCab System could be specified as follows:

$$Col_{tDC} = (Col_{tDC}, \{\mathsf{Shuttle}\}, CD_{tDC}, \{\texttt{createDC}, \texttt{moveSimple}, \texttt{moveDC}, \texttt{removeDC}\}, \Phi_{tDC})$$

With Φ_{tDC} being given as $\Phi_{tDC} \equiv \Box\neg\texttt{noDC} \wedge \Box\neg\texttt{collision}$.

tShuttleComp component type The tShuttleComp component type implements the DistanceCoordination's role Shuttle. However, there is no additional behaviour given for the tShuttleComp. The rules differ from the DistanceCoor-



(a) The `noDc` property (b) The `collision` property

Figure 8.11: Safety properties for the timed RailCab system

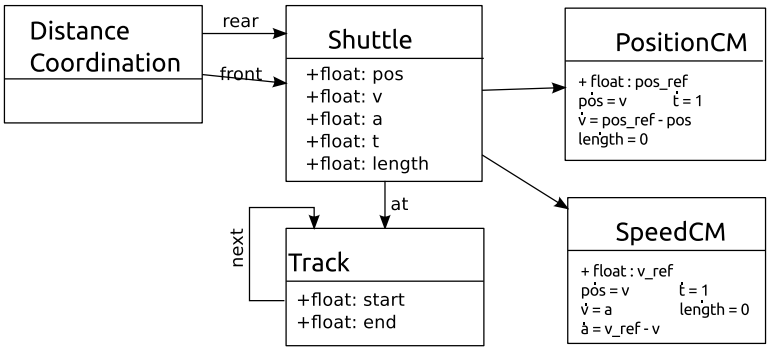


Figure 8.12: Ontology used for the RailCab application example

dination collaboration type's rules in an additional node of type tShuttleComp that is adjacent to the Shuttle role. No additional properties have been specified for tShuttleComp.

Hybrid RailCab System

The ontology for the hybrid variant of the RailCab system (see Figure 8.12) extends the timed RailCab's ontology by additional concepts. The ontology specifies two control modes that describe the Shuttle roles' continuous behaviour. The two control modes are used depending on the instantiation of the DistanceCoordination service contract. If a Shuttle is driving alone — i.e. without being in a convoy — it uses the SpeedControl control mode, whereas otherwise the PositionControl control mode is in place. A convoy's leading Shuttle uses the SpeedControl control mode too. The SpeedControl control mode tries to keep the Shuttle's velocity as close as possible to a given target velocity. The PositionControl mode tries to achieve this for the Shuttle's position, as driving in close proximity is essential for reducing the wind resistance.

For the specification of the system's continuous behaviour, we introduced a set of attributes for Shuttles, PositionControlMode, SpeedControlMode, and Tracks. A Track has two constant attributes start and end. These are used in the move rules to determine whether a Shuttle has physically left a Track. The two attributes start and end are constant, because the type Track does not have any control mode and thus no continuous behaviour is specified. The two control modes own the attributes pos_ref and v_ref, which hold the value of the current reference position and reference velocity. The type Shuttle has five attributes - v for the velocity, a for the acceleration, pos for the position, t for the time since the last move operation, and length for the Shuttle's length. The timely derivation of pos, v and a is determined by the current control mode (cf. laws in Figure 8.12). t is a clock and thus its derivation is constantly 1 and length is a constant which does not change over time.

DistanceCoordination collaboration type Whenever a Shuttle moves into the proximity of another Shuttle — independent of whether or not the ap-

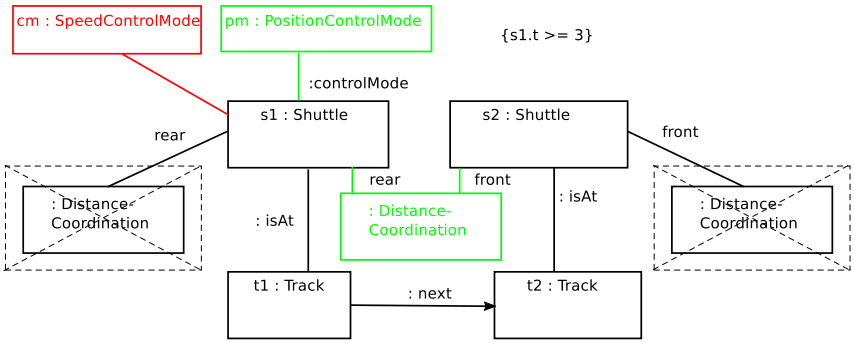
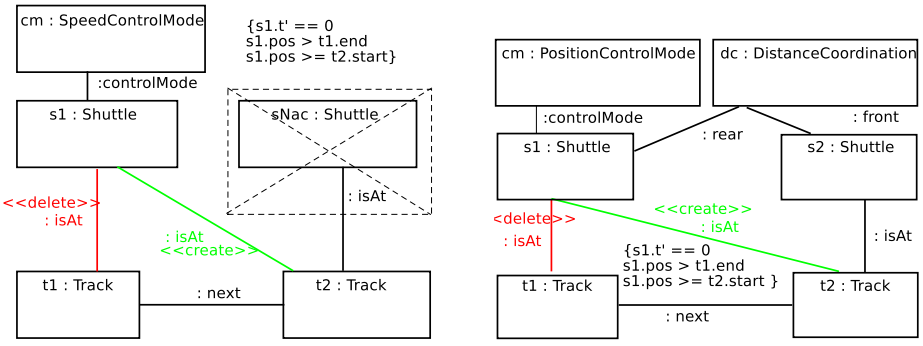


Figure 8.13: The DistanceCoordination collaboration type's createDC rule

proached Shuttle is part of a convoy — the DistanceCoordination collaboration type has to be instantiated between these two Shuttles. Thus, the creation rule createDC for the DistanceCoordination collaboration type (see Figure 8.13) is specified as an urgent rule. The createDC rule changes the rear Shuttle's control mode from SpeedControlMode to PositionControlMode. The front Shuttle's control mode is not changed. The two negative application conditions ensure that neither of the two Shuttles already has a DistanceCoordination collaboration instance instantiated, where it plays the same role. I.e. the front Shuttle is allowed to be the rear role in another DistanceCoordination collaboration instance, but not the leading one. The createDC rule is activated after the rear Shuttle is at the Track for three seconds. The rules for the DistanceCoordination collaboration type further comprise rules for the movement of the Shuttle if it is involved in a DistanceCoordination collaboration instance and if it is moving alone. The corresponding rules are depicted in Figure 8.14. The properties a DistanceCoordination instance has to fulfil are that the service contract is instantiated whenever it is required (see Figure 8.15(a)) and that no collision — i.e. two Shuttles that are located at the same track and the positions overlap — occurs (see Figure 8.15(b)).

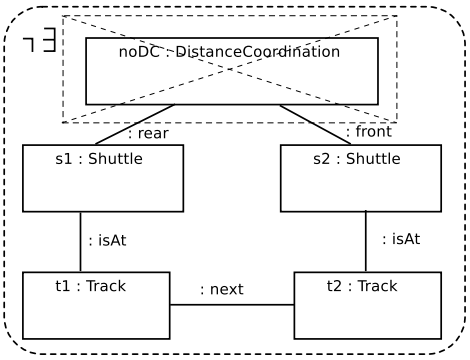


(a) move rule for a singular Shuttle

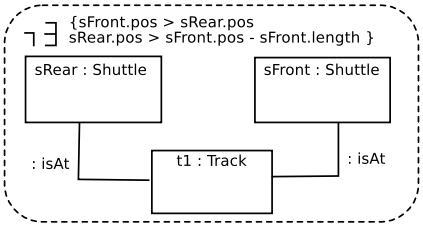
(b) move rule for the rear Shuttle in a DistanceCoordination service contract

Figure 8.14: The Shuttle service role's move rules

In Figure 8.14(b) only the move rule for a Shuttle role is shown, that is the



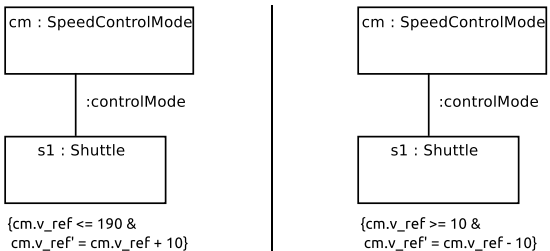
(a) The noDC forbidden property.



(b) The collision forbidden property.

Figure 8.15: Forbidden properties for the RailCab system.

rear part in a DistanceCoordination collaboration instance. Contrary to our first thoughts, together with the rule in Figure 8.14(a), this is sufficient to move all Shuttles of a convoy, the first Shuttle of a convoy and a Shuttle driving alone. This is due to the fact that, within a convoy, i.e. a consecutive chain of Shuttles each connected through a DistanceCoordination collaboration instance, each Shuttle other than the first one is the rear part for one DistanceCoordination collaboration instance. Hence, the moveDC rule (see Figure 8.14(b)) can be applied. The more general move rule (see Figure 8.14(a)), however, is applicable to a singular Shuttle and a convoy's leading Shuttle.



(a) The accel rule

(b) The brake rule

Figure 8.16: The Shuttle role's rules for acceleration and braking

Besides only moving, a Shuttle role can also change its desired speed if it operates in the SpeedControl control mode. Therefore, two rules for accelerating and braking are specified (see Figure 8.16).

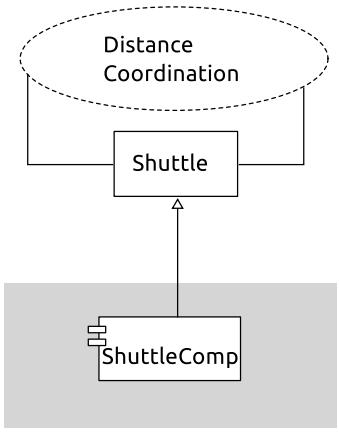


Figure 8.17: Evolution diagram for the RailCab example

The complete DistanceCoordination collaboration type can be formally specified using our notation introduced in Definition 4.10 as:

$$Col_{DC} = (Col_{DC}, \{\mathsf{Shuttle}\}, CD_{DC}, \{\texttt{createDC}, \texttt{move}, \texttt{moveDC}, \texttt{accel}, \texttt{brake}, \texttt{passPos}, \texttt{removeDC}\}, \Phi_{DC})$$

The corresponding class diagram CD_{DC} is given as ontology in Figure 8.12 and the collaboration type's safety properties are given as $\Phi_{DC} \equiv G\neg\exists\,\texttt{noCD} \wedge G\neg\exists\,\texttt{collision}$ with `noDC` and `collision` being the graph constraints depicted in Figures 8.15(a) and 8.15(b), respectively.

ShuttleComp component type The ShuttleComp component type implements the Shuttle role and refines all rules specified for the role in the DistanceCoordination service contract. For the ShuttleComp component type, no additional behaviour has been specified and the corresponding rules only differ from those for the service role in the additional node of type ShuttleComp. In consequence, we also have not defined additional safety guarantees for the component type.

The different parts of the RailCab example are shown in the evolution diagram given in Figure 8.17. As the timed and the hybrid variant of the RailCab example merely differ in the specified behaviour, the evolution diagrams look the same for both variants.

8.2.2 Analysis

We have already mentioned in the modelling Section 8.2.1 that the RailCab system uses complex rule-sets to specify urgent behaviour. Consequently, refinement checking is much more meticulous than for the supply chain example, where the refinement was given by the construction of the rules. However, in the RailCab system we have the special situation that one service role is used twice

Refinement	Duration[ms]	successful
$R_{SComp} \sqsubseteq_C R_{DC}$	47ms	✓

Table 8.4: Overview of the syntactical refinement check's duration

in the same service contract and thus implements all of the service contract's behavioural rules.

The ShuttleComp component type refines the rules that have been specified for the Shuttle service role in the DistanceCoordination collaboration type. The result of the syntactical refinement check for the hybrid RailCab system can be seen in Table 8.4. The important point here is that the `createDC` rule is marked as urgent. Hence, the DistanceCoordination collaboration type uses a complex rule-set (cf. Definition 6.15) and thus all refining rule-sets — i.e. the ShuttleComp's rule-set — have to satisfy the *Urgent* predicate (see Lemma 6.18). The ShuttleComp component type's createDC rule does not differ from the DistanceCoordination collaboration type's rule and thus the *Urgent* predicate is fulfilled for the RailCab example. The same holds for the *Preempt* predicate, which is satisfied, as we did not modify the rules. Due to the similarity between the rule-sets of the timed and the hybrid RailCab variant, we only give the results for the hybrid RailCab system.

Timed RailCab System

For the timed RailCab system, we had to verify that the collaboration type DistanceCoordination satisfies the two safety properties `noDC` and `collision`. As a timed system is always a linear hybrid system, we could use our automatic verification algorithm presented in Section 7.3. The results of the verification are shown in Table 8.5. The column labelled #LIE contains the number of systems of linear inequalities that had to be checked. The results have been computed using an older version of the Invariant Checker and are thus not directly comparable with the current implementation that has been used for the supply chain system.

Task	#R	#P	time [ms]	#Pairs	#LIE
$R_{tDC}(Col_{tDC}) \models \Phi_{tDC}$	4	2	340	68	22

Table 8.5: Detailed evaluation results for the timed RailCab example

Hybrid RailCab System

For the hybrid RailCab System we have modelled the complete behaviour at the level of the DistanceCoordination collaboration type (see Section 8.2.1) and the ShuttleComp component type refines this behaviour without adding any new safety properties. Thus the verification has to be done for the DistanceCoordination collaboration type only. Table 8.6 shows the results of the verification. The last column (#HA) shows the number of times the hybrid automaton had to be checked.

Task	#R	#P	time [ms]	#Sub	#Pairs	#HA
$R_{DC}(Col_{DC}) \models \Phi_{DC}$	7	4	2429	536	643	20

Table 8.6: Detailed evaluation results for the RailCab application example

The structural check of the Invariant Checker was able to reduce the number of hybrid automatons that have to be analysed to a total of 20. Each of the other 643 possible counterexamples against the RailCab system's correctness could be rejected due to violations of pure structural properties. These are merely cardinality constraints we added, such as that it is forbidden that a Shuttle is located at two Tracks at the same time (cf. Section A.1). A typical counterexample that is returned by the Invariant Checker is shown in Figure 8.18. The possible witness stems from a merge of the moveDC-rule (see Figure 8.14(b)) and the collision forbidden pattern (see Figure 8.15(b)). The Shuttle s1 has the rear role and the Shuttle s2 has the front role in the depicted DistanceCoordination collaboration type instance. The application of rule moveDC moves Shuttle s1 at the same Track where Shuttle s2 is located. Hence, after the rule application, the two Shuttles can possibly collide with each other.

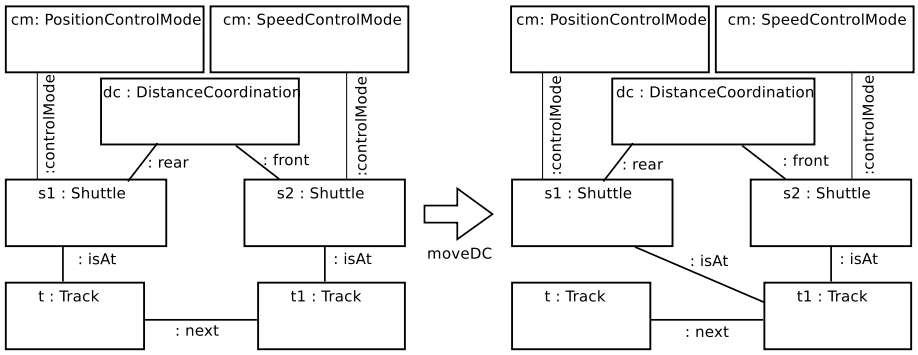


Figure 8.18: Possible witness against the RailCab system's correctness

In order to answer the question whether or not the two Shuttle will actually collide, we can also express this in terms of the reachability of the "failure" state of our generic automaton (see Section 6.1.3). The generic automaton shown in Figure 6.4 has to be instantiated with the situation described by the witness the Invariant Checker computed. The automaton's initial condition ϕ_{init} is built from the jump conditions of preempting rules and forbidden patterns' attribute condition found in the source graph pattern. The rule `moveDC` can only be preempted by the rule `createDC`. But `createDC` does not match in the source graph pattern, due to the existence of a DistanceCoordination instance. The same holds for the forbidden property `noDC` and thus $\phi_{init} \equiv true$. The flow condition and the location invariant of the "source pattern" location is always the same. However, the guard at the outgoing transition to the "target pattern" location uses the `moveDC` rule's jump condition: $s1.t' == 0 \wedge s1.pos > t1.end \wedge s1.pos \geq s2.start$. The "target pattern" location's invariant is determined by structurally embedded urgent rules' jump conditions and the forbidden pattern's attribute condition. In our example, no urgent rule matches and thus we set $\phi_U \equiv false$.

The forbidden pattern is `collision` and hence $\phi_{collision} \equiv s2.pos - s2.length < s1.pos \wedge s1.pos < s2.pos$. It follows that the guard to the "urgent" location is set to $\phi_U \equiv false$ and the guard to the "failure" location is set to $\phi_{collision} \wedge \neg\phi_U \equiv \phi_{collision}$. The flow condition for the "target pattern" location is given through the combination of the control modes. Thus, the SpeedControlMode controls the attributes of Shuttle s2 and the PositionControlMode controls the attributes of Shuttle s1.

We have specified this concrete instantiation of our generic automaton using PHAVer's input language and queried PHAVer whether the location "failure" is reachable. The location was not reachable and thus a collision could not happen. However, PHAVer uses an overapproximation to capture the continuous behaviour. Any attempt to change this by giving additional refinement hints ended in a segmentation fault, probably due to the excessive memory usage by PHAVer[3]

8.3 Summary

In this chapter we have shown that it is possible to successfully model and verify the supply chain and the RailCab system, using our methodology. The restrictions we put on the behaviour modelling — i.e. criteria for refining rules, separation of meta-models — has paid off at the verification stage, as we were able to reuse the verification results and were able to reduce the necessary checks to a small number.

The modelled systems are of two completely different natures. Whereas in the supply chain example the focus is on showing how abstraction and separation can be efficiently used to specify a complex, open and evolving system, we had to specify all behaviour for the RailCab system within the DistanceCoordination service contract. Nevertheless, the RailCab system can still be enhanced with e.g. different component types that vary in the capabilities they offer. The two examples differ further. In the supply chain system, it is not required that an instance of a Contract collaboration type is created. The RailCab's safety, however, relies on the instantiation of the DistanceCoordination collaboration type.

For the supply chain system, we had to cope with the restricted expressiveness of our modelling language (cf. Subsection 8.1.1). For the modelling of the Factory component, it would have been beneficial if our modelling language had had the expressive power of nested application conditions. Finally, we were able to overcome this limitation through the use of a complex rule-set and preempting rules. Using preemption, we were able to express the desired behaviour — which was too complex to be expressed in one rule — in two rules and thus successfully model the Factory component and finally verify that it behaves correctly.

The RailCab example illustrates the usage of different control modes for different modes of operation for the Shuttles. The fact that we model the control modes as nodes allows us to set and read them easily using StoryPatterns. The

[3]PHAVer's memory usage easily reached the 4GB barrier.

Task	Degree of Automation	
	fully	semi
Verify discrete GTS	✓	
Verify linear hybrid GTS	✓	
Verify hybrid GTS		✓
Check Predicates		✓
Check syntactical Refinement	✓	

Table 8.7: Tool suppport of different analysis tasks.

expressiveness of the differential equations that can be used in the control modes only depends on the analysis technique used.

In Chapter 2 we have identified a set of requirements that have to be met by a modelling and verification approach for SoS. The examples in Chapter 4 have already shown that our approach is suitable to model SoS. In this chapter, we have given the proof that our approach is further suited to model physically complex systems such as the behaviour of the RailCab system, including different control modes for the Shuttle role. Additionally, we have shown through the comparison with GROOVE that our approach scales very well with the system size (Requirement Scalable Analysis (A1)). The applicability of our verification approach (Requirement Applicable Analysis (A2)) is given, too, as we were able to verify the required safety properties. The required capability to verify both the system's behaviour and its reconfigurations (Requirement Analysis of Reconfiguration (A3)) is inherently given, as our modelling approach does not distinguish between behaviour and reconfiguration specification. That the last verification requirement Analysis under restricted knowledge (A4) , which requires the analysis to be applicable under intellectual property constraints, is satisfied, can be seen from the information we used in the verification of the different tasks. In each task, we only used the rule-sets that were specified for the collaboration or the component type and did not use any information that might be internal to a component type implementation.

Our approach also satisfies Requirement Analysing Evolution (A5). In Section 5.3 we have shown that the verification of evolving systems can be tackled using the techniques that we used for the verification of SoS without evolution. Hence, in order to verify the evolution step from a system Sys_i to a system Sys_{i+1}, we have to verify that the added component or collaboration type is correct.

Nevertheless, our approach is not yet able to verify a *rigSoaML* model fully automatically, although we have presented the necessary concepts and have shown that they work in principle. In Table 8.7 we give an overview of the degree of automation for the different analysis tasks we have presented throughout this thesis. The verification of hybrid GTS is marked as semi-automatic, as the generation of the required hybrid automata is not yet supported. For the checking of the predicates, the pre-processing step that is described in Section 7.4.2 is not yet implemented.

Chapter 9

Related Work

In this thesis we visited several fields ranging from high-level, visual modelling over formal modelling to analysis and verification, all under the special aspects that have to be satisfied for SoS. We will give an overview of the work which is the most relevant for our thesis. As analysis and verification is unfeasible without formal modelling, we discuss these two aspects together.

9.1 Modelling

SoaML is not the first modelling approach for service-oriented systems, and for other related domains modelling approaches already existed. In this section we will review the most relevant of them. Reconfiguration and adaptation are tightly connected to the research domain of dynamic software architectures. In contrast to a static software architecture — the dependencies between components are determined at design time — a dynamic software architecture features changing connections between components. One approach that addresses the modelling of dynamic software architectures is [42]. The paper mainly focuses on compositions of components, where a component is given as a finite automaton, a set of ports and history variables. What the approach does not support are clear rules that express the change of the components' connections. Further, the creation and deletion of components is not supported yet.

The RailCab example served also as a motivating example for the development of Mechatronic UML [47, 48]. Mechatronic UML uses UML component diagrams to describe a system's structure and real-time statecharts, a real-time extension of UML statecharts, to specify the components' behaviour. Mechatronic UML can then be extended by a reconfiguration capability [46]. Therefore, each location of a real-time statechart is equipped with the required internal configuration of the component. In [87] an approach is presented that also makes use of collaborations and dynamic structural adaptations. For the verification of the structural adaptations, the authors made the assumption that the overall number of roles is small enough, such that they were able to translate reconfiguration behaviour into time-automata, which will then be verified. Concerning

the problems we addressed in this thesis, the Mechatronic UML approach has some shortcomings. Mechatronic UML only addresses the reconfiguration internal to a given component; in order to specify a behaviour similar to our approach, one would have to model a single comprehensive component and the reconfiguration behaviour within a real-time statechart. However, this requires that the exact reconfiguration behaviour is known at the design time of the system. Regarding the verification capabilities, the use of statecharts and standard model-checking might be beneficial for the expressiveness of the safety properties that could be verified. However, the scalability is likely to become an issue due to the state space explosion and in general it is very cumbersome to verify structural properties using model-checking techniques which were developed for the verification of finite automata. Continuous evolution of the systems is also hard to support, as each addition of a new component into the system requires the re-verification of the complete system.

A more formal approach is presented by Fiadeiro et al. [67]. The authors present an algebraic modelling language for the specification of service orchestrations. In contrast to our work, the authors restrict their approach to orchestrations and thus to closed systems. Further, an important part of their work is to prove that the composition of services preserves the services' overall properties. The decentralised development of services and components based on the refinement of services is not discussed in their work. Their formal model also does not cope with structural dynamics as we do in our work.

9.1.1 Role-based modelling and contracts

UML class diagrams for the structure and graph transformations for the behavior modelling are also employed in [35] to model service-oriented architectures, but in contrast to our approach services are not modelled as collaborations.

The use of UML collaborations for the modelling of services has been proposed by several authors and proposals (cf. [28, 39, 43, 119]). In [43] also collaborations have been used, but not UML collaborations. However, none of the three modelling concepts supports dynamic collaborations. In [119] static but hierarchic collaborations and the distinction between the collaboration and the collaboration use are presented. Further, the authors omit the definition of the roles' behaviour. This has been done in [43], but only partially - Broy et al. use sequence diagrams for the behaviour specification. [28] is similar to [119] but extends [119] with behaviour specifications for the different roles. A profile that directly supports reconfiguration at an architectural level is UML4SOA [45]. In contrast to the former profiles, UML4SOA models services a ports. Reconfiguration rules are given at the level of component diagrams that are augmented with special stereotypes to encode preconditions and the effect of the reconfiguration. UML4SOA further provides formal support for the modelled architecture. Therefore, the architecture is translated into graphs and the reconfiguration rules into term-rewrite rules. An LTL model-checker is employed to verify properties of the architecture. Applied to our application domain, UML4SOA has some drawbacks. The fact that services are modelled as ports rather than as collaborations implies that either the complete system specification has to be known or the analysis technique can cope with an unconnected port (unused

service in this case). It follows that changes to the system require a new verification run, as now possibly other reconfiguration rules can be applied. Lastly, the profile does not provide a specification of the service behaviour.

All the presented collaboration concepts could be seen as advances in the idea of contracts, which has been introduced in [84]. A contract consists of a number of participants, each having some *contract obligations* to fulfil. Contracts also support the idea of roles that have to be mapped to classes. But contracts only support a constant number of participants and do not provide support for adding or removing participants to contract instances at run-time.

Also, a less clear historical connection between roles/collaborations and design pattern [77] exists, which is reflected today by the fact that design patterns can be modelled in UML using collaborations. The modelling of design patterns in UML is advocated in [73, 97]. The authors in [73], however, do not use UML collaborations for the modelling of design patterns, but develop their own meta-model and use UML sequence diagrams, which potentially describe partial behaviour. Kent and Lauder [97] instead propose their own visual notation, but use sequence diagrams for the behaviour modelling, too.

A more formal approach to the modelling of patterns and behaviour is presented by Kim and Carrington in [94]. They use Object Z for modelling design patterns and their behaviour. Although Object Z is a very versatile formal language, the approach of Kim and Carrington does not support dynamic collaborations.

The OOram Software Engineering method (cf. [116]), which was developed in the 1970s, already used the distinction between roles and objects. Whereas roles and objects are often counted to the object-oriented paradigm, OOram assigned each role a specific behaviour and used behaviour synthesis to derive the final behaviour. Obviously, the behaviour synthesis is a hard task and can only be generalised for a restricted set of problems.

9.2 Formal Modelling & Verification

Early in this thesis, we pointed out that testing and naive model-checking of open service-oriented SoS are not applicable approaches. Nevertheless, a huge body of work exists that addresses the verification of dynamic and decentralised systems using formal modelling approaches that range from finite state machines, over Petri nets and term rewriting, to graph-transformation systems.

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. [135]. This approach can only verify finite state systems. However, open service-oriented systems are generally infinite state and thus the approach isn't applicable.

Types are a standard element of modern programming languages, and consequently they have also been used and reflected concerning the verification of programs. Interesting for our work are especially so-called behavioural type systems, which subsume not only data elements but also behaviour. Liskov and Wing [101] give a definition of types and subtypes that is not purely syntactical

but also implies behavioural 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.

An approach that directly addresses the formal modelling of SoS is that of Fitzgerald et al. [68]. In [68], a combination of complementary modelling techniques is presented, each of them addressing different aspects of an SoS. An enhanced variant of SysML is used to describe the system's architecture, and CSP describes the constituent systems' behaviour, but no modelling language supports to also model the dynamics that occur in an SoS. Hence, the properties we are able to verify in our approach cannot be expressed in [68].

To some extent, the systems that we describe in this report can be seen as ensembles, as introduced by Hölzl and Wirsing [88]. The formal model that is presented in [88] is very expressive, but lacks the possibility to encapsulate the services' behaviour into collaboration - or similar constructs. Further, no analysis techniques exist yet.

9.2.1 Finite State Machines

Finite state machines are a popular formal model, as they can be verified using well-established techniques such as model-checking. In this section we will present several approaches that use finite state machines as an underlying formal method to describe SOA or SoS.

Beyer et al. [41] argue that web-services should be specified through an interface automaton. Their approach concentrates on the verification of one single automaton, but does not consider the structural dynamism that is present in a SoS.

In [27], an approach is introduced that is not dedicated to the verification of service-oriented systems or SoS directly but addresses the verification of hierarchical systems. The authors show that the flattening of a hierarchical system can be avoided by the use of a hierarchical game. Although the results of the authors look promising, their approach requires knowledge of the complete system to be checked. Therefore it cannot be applied to SoS.

Only closely related is the data-aware approach of Hallé et al. [82]. The authors show a variant of CTL, CTL-FO$^+$, that can be used to verify temporal properties of the data passed in the messages of an SOA. For the verification, a transformation to classic CTL is given.

9.2.2 Petri Nets and Process Technologies

Petri Nets are a common formalism to describe service behaviour. Stahl and Wolf [122] presented an approach to check the compatibility of service compositions. Therefore they make use of so-called open nets, which are basically Petri nets with some unconnected places. In a service composition, these places

are connected to other services and thus the complete behaviour is available. The authors give sufficient conditions to decide whether two services can be successfully combined. Compared to our work, this approach only tackles the refinement aspect, but behaviour preservation cannot be shown. However, it is possible to show the deadlock-freeness of the combined process. Any further verification of the combined processes is not solved; also Petri nets are a comparatively static formalism that it is hard to use to formally describe the structural dynamism of service-oriented systems. Another approach, addressing the compositional verification of Petri nets is that of Juan et al. [91]. This approach allows them to efficiently reduce the state space and verify some important properties of Petri nets, but it does not provide the expressive power to model structural dynamics.

A transformation-based verification technique is presented in [134]. Web-service compositions become transformed into an 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. [53] follows a similar approach. A transformation-based verification technique is presented in [134]. Web-service compositions become transformed into an 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. [53] follows a similar approach.

Model-checking has been employed to check business process models with varying numbers of active process instances. In [70, 72, 71], for example, standard BPEL models are enriched with resource allocation behaviour to ensure the correct detection of deadlocks and safety violations for web-services compositions under resource constraints. The same underlying analysis technique — LTSA - Labelled Transition System Analyser — is used by the authors of [52] for the verification of service compositions. This approach lacks the functionality to verify dynamic systems, as the compositions have to be known a priori. Later, Foster [69] presents an approach that uses finite state machines for the specification of the services' behaviour. Additionally, different modes can be specified, where each mode represents a pre-defined service configuration. An orchestrator is responsible for managing the service invocations and reacting to events that trigger mode changes. The orchestrator, the mode configurations and the services' behaviour become combined into a finite state process, which is then analysed. A verification approach like this can only work if the complete system is known in detail at design time. For SoS, such an approach is therefore not applicable.

9.2.3 Term Rewriting and π-calculus

For the formal modelling of concurrent and distributed systems, process calculi are often used. The best suited calculus to model service-oriented self-adaptive systems is the π-calculus [105], as it specifically allows the modeller to dynamically create new communication channels. Although the π-calculus has been used to formally describe service-oriented systems [59, 102] and business pro-

cesses, 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 [57]. Approaches that allow the model checking of π-calculus specifications are available, but typically only allow the verification of a restricted subset of the π-calculus. Yang et al. describe an approach to model check π-calculus specifications with logic programming [133], but they had to forbid processes that do not contain finite replication or the parallel composition of processes. The Mobility Workbench [130] allows only a finite number of processes too.

Baldan et al. present a semantic framework [31] for the specification of open processes. Their framework is based on the π-calculus and distinguishes two categories of processes: components and coordinators, where a component is a closed process, i.e. it is specified without any unbound variables, and a coordinator is a process that has unbound variables, which are then bound to components. Hence, a coordinator is comparable to a collaboration in our terms. What is missing from the work of Baldan et al. is the decoupling of different coordinators, which is a key element of our approach. Hence, it is not guaranteed that the interplay of two coordinators which are combined over a common process behaves correctly. However, the semantic framework allows us to check bi-similarity for such open systems. Although bisimulation is hard to check and an important feature for process calculi, it does not guarantee any safety properties. Hence, the capability to verify the system is still missing.

In the context of process-based formalisms and the π-calculus, several approaches have been developed that support type-systems for processes and are used to ensure correctness through the correctness of the type-systems. In [89], Igarashi and Kobayashi present a framework for the specification of systems of behavioural types for the π-calculus. The basic idea of their approach is 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 behavioural types for service contracts is advocated by Meredith et al. [104], but without any contributions for their verification.

The problem of deciding safety properties for infinite state π-calculus systems with the help of behavioural types is addressed in [24]. The basic idea in this work is that under certain conditions a process satisfies a property ϕ exactly if the process type satisfies ϕ. The types, however, are CCS-terms but not π-calculus processes, thus the verification of properties for the types is easier. Properties are given through a logic called spatial logic, which makes it possible to encode process typical properties, but which is not suited to directly encode structural properties of systems.

A term rewriting approach for adaptive systems is presented in [44]. The authors use the term rewriting tool robotic systems. The presented approach relies on a layered architecture and term rewriting theories to specify the adaptation. Further, a set of tools for the simulation and static model-checking of MAUDE specifications is shown, that can support the development of adaptive systems. Compared with our approach, [44] does not support evolution and the clear dis-

tinction of types and instances, in addition to which the behavioural subtyping capability is missing.

In [125], the authors present an approach for the formal verification of hierarchical CSP specifications including time. In the input model used, a CSP process is equipped with a clock that starts counting when the process becomes active, and the process has to be terminated before the clock runs out. For the verification, the authors compute finite-state abstractions which they could show were time-abstract bisimilar. However, as with most process-based approaches, [125] fails to model and verify structural dynamics of systems. The evolution of systems isn't captured by this approach either.

In their work, Ramos et al. [115] discuss compatibility notions for components. They use CSP to express that two components are compatible with each other. The problem they solve exists in a different setting than the one we have in our approach. We develop the component types in accordance with the roles of collaboration types. Compatibility is then given through refinement. On the other hand, we do not check any lifeness properties of our specifications and thus cannot guarantee that a desired behaviour emerges.

In [81], Gardara 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 of updating modules. In comparison with our work, they follow a bottom-up approach for the verification. Hence, the system has to be known in advance, whereas our approach — concerning the verification — is more like a top-down approach. In our approach, the verification is performed at the type-level, whereas in [81] the instance-level is checked. Furthermore, reconfiguration is not addressed.

In [58] 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.

Recently Fantechi et al. [66] presented a verification technique based on branching time logic. The authors used a process calculus dedicated to the orchestration of web-services as underlying formalism. However, although the approach provides a good formal support for the verification of service-oriented systems, the structural dynamics and evolution are not covered.

Real-Time Maude [108], 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 behaviour of the system as well as bounded model checking of the complete state space, if it is finite.

9.2.4 Graph transformation Systems

Ciraci et al. [54] describe an approach that simulates the dynamics in the call-graph of object-oriented programs using the graph-transformation tool GROOVE. Based on the state space that is generated by GROOVE, they are able to check simple properties. Obviously, this approach is not suited to verifying the class of self-adaptive service-oriented systems that we are investigating in this thesis.

There are only first attempts that address the verification of infinite state systems with dynamic structure: in [32, 33] graph-transformation systems are transformed into a finite structure, called a Petri graph which consists of a graph and a Petri net, each of which can be analysed 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 and do not support NACs.

An approach for the verification of possibly infinite state graph transformation systems is described by Bauer et al. [37]. They use a technique called neighbourhood abstraction to create a finite set of abstract graphs. The properties they can verify using this technique have to be specified in a special logic, which can mainly express path expressions. The logic formulae are evaluated on single nodes of an abstract graph. The authors state that, due to the abstraction technique, only a restricted variant of negative application conditions can be supported. The authors approach shares with our approach the commonality that graphs are used to represent a potentially infinite number of concretisations. In their approach, these are abstract graphs, whereas our Invariant Checker approach uses graph patterns for this purpose. The approach is further differentiated from ours Invariant Checker as it relies on model-checking, which requires a reachability analysis. Although, the neighbourhood abstraction helps here, the correct degree of abstraction still has to be identified and even if the abstract GTS are finite, they still might be extremely large and even too large. The authors have shown in [118] that their neighbourhood abstraction can be implemented.

An approach which has been successfully applied to verify service-oriented systems [35, 36] is that of Varró et al. It transforms visual models based on graph theory into a model-checker specific input [129]. A more direct approach is GROOVE [117] by Rensink, where the checking works directly with the graphs and graph transformations. DynAlloy [75] extends Alloy [90] in such a way that changing structures can be modelled and analysed. For operations and required properties in the form of logical formulae, it can be checked whether given properties are operational invariants of the system.

Partner graph grammars are employed in [38] 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. A further approach that facilitates abstractions is [76]. Gadducci et al. present a framework

that supports over- and underapproximation of counterpart models. Although, this approach is solving the problem of scalability, it is not applicable to the class of SoS, as evolution is not directly supported and thus the system has to be verified again for each change that is made to it.

In [62] Ehrig et al. present an approach for the analysis of self-healing systems, modelled through algebraic graph transformations. In this approach, the system's operational behaviour is split across normal and repair rules. Further environment rules model the system's environment and can lead to so-called failure states. The authors check that all possible failure states can be tackled with an appropriate repair rule and hence transformed into a normal state. In contrast to our approach, the correct behaviour of the normal rules is not checked. Further, the approach does not allow for the evolution of the self-healing system.

Wehrheim, Steenken and Woinsch present in [131, 123] an approach for modelchecking of a graph-transformation system. The problem of a possible infinite state system is tackled by an abstraction from the actual system state through shape graphs. However, the approach relies on one specific system that could be verified and does not verify a static property of the system's rules, independent of the system's initial state.

An approach that also uses refinement to guarantee behaviour preservation is that of Heckel and Thöne [83]. The notion of refinement they use is similar to our approach, as they also use an abstraction function to map concrete and abstract graphs. The mapping is defined as the removal of all nodes and edges whose type is not contained in the abstract type-graph. In our approach the abstraction function is implicitly given, whereas in that of Heckel and Thöne the abstraction function could be arbitrarily defined. The most prominent difference between the two ways of refining graph- transformation systems lies in the definition and the checking of a behaviour preserving refinement. Our approach makes some syntactical restrictions on the refined rules. Heckel and Thöne don't have such a requirement. On the one hand, this allows them to relate two different GTS with each other without many restrictions. On the other hand, this requires that they have to use a model-checker that verifies for each transition in the abstract system that in the concrete system a correlated path exists. Further, such an approach complicates the check whether a urgent rule is always applied. Preemption of graph-transformation rules isn't supported either.

Kallel et al. provide with MEIDYA [92] an approach to verify invariants in dynamic software architectures. The software engineer has to model the software architecture together with a set of reconfiguration operations, which are graph-transformation rules, and has to provide a set of invariants that the software architecture has to satisfy. They use the formal language Z with an interactive theorem prover to verify that no reconfiguration rule violates the invariants. To enforce correct reconfigurations at run-time, too, they employ aspect-oriented modelling. In contrast to our work, the verification of the invariants is not fully automated; however this might allow them to verify more expressive invariants.

9.3 Summary

From what we have seen, the modelling approach using collaborations and roles is a standard technique that has been used by several other proposals and papers before. However, the fact that our collaboration types are designed in a way that allows their participants — i.e. the components implementing the collaboration types' roles — to dynamically join and leave an instance makes our approach unique.

Concerning the verification, a lot of the graph-transformation based approaches, that are suitable for the verification of infinite-state systems use abstraction to tackle the complexity. Whether abstraction is successful in reducing the state space to a finite one that still allows us to verify the required properties depends on the system under verification. In principle, the π-calculus and its derivates are able to express the structural dynamics occurring in SoS. However, no approach exists that allows us to verify the π-calculus' full expressiveness. The term-rewriting based approaches were all restricted to finite state space systems.

The last critical requirement the presented approaches had to fulfil is evolution of the systems. Whereas most of the modelling approaches we have presented could be easily extended to also support evolution, the verification approaches all require the renewed verification of the complete changed system. This, however, makes the approaches, even if they have the required expressiveness, inapplicable to systems of systems that evolve often and in an uncoordinated way.

Chapter 10

Conclusion

In this thesis we have presented a modelling and verification approach for systems of systems. In a first step we characterised SoS through a set of different requirements comprising the aspects of modelling, verification and evolution. Summarising, our requirements concentrate on the facts that SoS have a high degree of structural dynamics, are open systems without clearly known boundaries and are subject to uncoordinated evolution of their constituent systems. Additionally, we have identified common scenarios that are typical for these systems and that describe the possible changes that can happen within them.

Before presenting our own approach, we first discussed the state-of-the-art modelling standard *SoaML*, which is promoted by the OMG. After having identified *SoaML*'s weaknesses, we were able to overcome these and present our own modelling approach called *rigSoaML*. *rigSoaML* reuses *SoaML*'s basic concepts such as collaborations, roles and components, but enhances them with a new behaviour specification, a well-defined inheritance relationship and evolution diagrams. The behaviour specification is given through StoryPatterns, a compact notation for graph-transformation rules. Inheritance between the different entities of our modelling approach, such as components, collaborations, and roles, has been formally defined as a refinement relationship. Evolution diagrams point out which collaborations, components and roles specialise each other. Formally, these concepts are mapped to typed and attributed graphs and sets of graph-transformation rules. We also made the properties that have to be fulfilled first-class citizens for *rigSoaML*, and hence they have to be given for each collaboration and component type. This approach allows us to give a comprehensive specification of open service-oriented systems of systems.

Our modelling approach *rigSoaML* allows us to verify SoS in a reasonably efficient way. Thereby, the developed verification concept is twofold. At the conceptual level we have described a verification scheme that gives us the obligations we have to check at the concrete level. The verification scheme assumes that two rule-sets refine each other and that the collaboration and component types satisfy their properties. Under these assumptions, we were able to show that the overall system is safe. At the concrete level, we have proved that the rule-sets do refine each other and that they met the safety properties. There-

fore, we presented a notion of refinement for a set of graph-transformation rules and developed a tool — the Invariant Checker — that is capable of verifying inductive invariants for graph-transformation systems.

Finally, we have evaluated our modelling and verification technique using two application examples, showing that our approach satisfies the requirements we identified for self-adaptive service-oriented systems.

10.1 Outlook

This thesis provides answers in a complex field of computer science and at the same time raises new questions. Concerning the modelling aspect, it would be beneficial to raise the modelling language's expressiveness. This could be achieved through the introduction of nested application conditions to StoryPatterns, which would — to some extent — allow us to make the modelling more intuitive. However, a stronger modelling language always has to be supported by a verification technique that can verify the more expressive language. For graph transformation, it has already been shown that arbitrary combinations of nested application conditions are as expressive as first order logic on graphs [113], and hence verification for such languages is generally not possible [56]. The modelling could further be supported by a comprehensive modelling environment that allows the developer to specify abstract and concrete component and collaboration types. The modelling environment then automatically performs the different required checks, such as verification of the safety properties, refinement check and that the preempt and urgent predicates are satisfied.

Concerning the verification, we will complete the filter chain for the verification of non-linear hybrid GTS (see. Section 7.3) such that the manual invocation of PHAVer is not necessary any longer. Additionally, we will implement the missing pre-processing step that is required for the check of the preempt and urgent predicates. The Invariant Checker approach can under some conditions return false negatives, i.e. counterexamples against the system's correctness, which are actually not counterexamples. This, to some extent, happens due to too little context being available in the source and target graph patterns. Actually, only the necessary information for the combination of a graph- transformation rule and a forbidden graph pattern are available for the algorithm. This can yield a situation where the algorithm returns false negatives, as e.g. a preempting graph-transformation rule could not be applied to the source graph pattern. In Section 6.3, we have already argued that it is possible to enrich the source and the target graph pattern with additional information that is available from the system specification. Further, the Invariant Checker algorithm is currently only able to verify the absence of forbidden patterns, but it might be useful as well to have the capabilities to verify that some patterns are preserved by a set of graph-transformation rules. E.g. for a token-ring protocol, we can currently verify that no participants hold two tokens at the same time, but the equally important statement that one token always exists cannot be verified.

From the scenarios we introduced in Section 2.5, a few are not yet supported by our approach. These are the deletion or removal of collaboration and component

types from the system specification and the update of component types. These three scenarios are not yet supported, as we have no possibility of knowing which instances or subtype is still using the specifications. One possible solution to this problem could be the introduction of a lease model for the specifications. Each instance has to lease all the specifications (including the super-types) it uses for a given period of time. If this period has passed, the instance must no longer use the specification but has to renew the lease. If a specification is marked for removal, the lease to use this specification could be renewed. After the last lease has run out, the specification could be safely removed. In principle, updates of component types could be handled similarly; however, at least for refined component types this might not be sufficient as the refinement has to be rechecked after the update. Thus, an update of component types will probably only be possible for concrete component types, which must not be refined.

This thesis discussed the problems that arise when developing service-oriented systems of systems at a conceptual and theoretical level. For the implementation of these systems, a model-driven engineering approach has to be established that does not invalidate the verification results. Especially for hybrid systems, this becomes an issue, as timing constraints that are encoded in the system have to be met by the implementation. First attempts have been made to compute the worst case execution time (WCET) for StoryPatterns [126]. However, this approach mainly targets the local execution of StoryPattern, whereas for our system class at least the network delay has to be considered too.

In this thesis, we have shown a way to verify SoS against a formal specification. However, it still has to be ensured that, at run-time, components which were newly introduced into the system fulfil the required properties. Solutions for this problem include the introduction of certificates, which allow us to easily check that a component type or collaboration type has been successfully verified. Another solution is proof-carrying code — i.e. the newly added artifacts provide their correctness proof upon request. Further, an infrastructure similar to Minsky's law governed interaction [106, 120] could be employed. One possible implementation could be a middleware at which components have to register. The registration comprises, among other things, a behaviour specification based on StoryPatterns. The middleware could then verify that these StoryPatterns conform with the system specification and further ensure that the components' actions all conform with the registered behaviour. Beside the work of Minsky et al. other approaches concerning the run-time checking of service-oriented systems exist. Similar to Minsky, Coronato and De Pietro [55] developed another middleware-based approach. In their approach, the system is constantly monitored and repair strategies are enacted whenever an error occurs. For service-oriented systems, Ghezzi and Guinea [78] advocate the use of pre- and post-conditions for the invocation of external services within an orchestration.

Own Publications

[1] Basil Becker. Towards Safety Guarantees for Service-Oriented Systems. In *ICSE Companion 2009, Companion of the 31th International Conference on Software Engineering*, pages 347 – 350. IEEE Computer Society, 2009.

[2] Basil Becker, Dirk Beyer, Holger Giese, Florian Klein, and Daniela Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *Proc. of the* 28^th *International Conference on Software Engineering (ICSE), Shanghai, China*. ACM Press, 2006.

[3] Basil Becker and Holger Giese. Incremental Verification of Inductive Invariants for the Run-Time Evolution of Self-Adaptive Software-Intensive Systems. In *Proc. 1st International Workshop on Automated engineeRing of Autonomous and run-tiMe evolvIng Systems (ARAMIS)*, pages 33–40. IEEE Computer Society Press, 2008.

[4] Basil Becker and Holger Giese. Modeling of Correct Self-Adaptive Systems: A Graph Transformation System Based Approach. In *CSTST '08: Proc. 5th Intl. Conference on Soft Computing as Transdisciplinary Science and Technology*, pages 508 – 516. ACM Press, 2008.

[5] Basil Becker and Holger Giese. On Safe Service-Oriented Real-Time Coordination for Autonomous Vehicles. In *In Proc. of 11th International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC)*, pages 203–210. IEEE Computer Society Press, 5-7 May 2008.

[6] Basil Becker and Holger Giese. Cyber-Physical Systems with Dynamic Structure: Towards Modeling and Verification of Inductive Invariants. Technical Report 64, Hasso Plattner Institute at the University of Potsdam, Germany, 2012.

[7] Basil Becker and Holger Giese. Modeling and Verifying Dynamic Evolving Service-Oriented Architectures. Technical Report 75, Hasso Plattner Institute at the University of Potsdam, 2013.

[8] Basil Becker, Holger Giese, Stephan Hildebrandt, and Andreas Seibel. Fujaba's Future in the MDA Jungle - Fully Integrating Fujaba and the Eclipse Modeling Framework? In *Proceedings of the 6th International Fujaba Days*, 18-19 September 2008.

[9] Basil Becker, Holger Giese, and Stefan Neumann. Correct Dynamic Service-Oriented Architectures: Modeling and Compositional Verification with Dy-

namic Collaborations. Technical Report 29, Hasso Plattner Institute at the University of Potsdam, 2009.

[10] Basil Becker, Holger Giese, Stefan Neumann, Martin Schenck, and Arian Treffer. Model-Based Extension of AUTOSAR for Architectural Online Reconfiguration. In Stefan Van Baelen, Thomas Weigert, Ileana Ober, and Huascar Espinoza, editors, *Proceedings of the 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB 2009)*, volume 507 of *CEUR Workshop Proceedings*, pages 123–137. CEUR-WS.org, 6-6 October 2009.

[11] Basil Becker, Holger Giese, and Daniela Schilling. A Plugin for Checking Inductive Invariants when Modeling with Class Diagrams and Story Patterns. In *Proc. of the 3rd International Fujaba Days 2005, Paderborn, Germany*, pages 1–4, September 2005.

[12] Basil Becker, Leen Lambers, Johannes Dyck, Stefanie Birth, and Holger Giese. Iterative Development of Consistency-Preserving Rule-Based Refactorings. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations, Fourth International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, volume 6707 of *Lecture Notes in Computer Science*, pages 123–137. Springer / Heidelberg, 2011.

[13] Basil Becker, Stefan Neumann, Martin Schenk, Arian Treffer, and Holger Giese. Model-Based Extension of AUTOSAR for Architectural Online Reconfiguration. In Sudipto Ghosh, editor, *Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers*, volume 6002 of *Lecture Notes in Computer Science (LNCS)*, pages 83–97. Springer-Verlag, 2010.

[14] Betty Cheng, Rogerio Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In Betty Cheng, Rogerio Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2009.

[15] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl Goeschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith, Joao P. Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. Software Engineering for Self-Adaptive Systems: A second Research Roadmap. In Rogério de Lemos, Holger Giese,

Hausi A. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science (LNCS)*, pages 1–32. Springer, January 2013.

[16] Holger Giese, Leen Lambers, Basil Becker, Stephan Hildebrandt, Stefan Neumann, Thomas Vogel, and Sebastian Wätzoldt. Graph Transformations for MDE, Adaptation, and Models at Runtime. In Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors, *Formal Methods for Model-Driven Engineering*, volume 7320 of *Lecture Notes in Computer Science (LNCS)*, pages 137–191. Springer Berlin / Heidelberg, June 2012.

[17] Regina Hebig, Holger Giese, and Basil Becker. Making Control Loops Explicit When Architecting Self-Adaptive Systems. In *SOAR '10: Proceedings of the second international workshop on Self-Organizing Architectures*, pages 21–28, Washington, DC, USA, June 2010. ACM.

[18] Stephan Hildebrandt, Leen Lambers, Basil Becker, and Holger Giese. Integration of Triple Graph Grammars and Constraints. In Christian Krause and Bernhard Westfechtel, editors, *Proceedings of the 7th International Workshop on Graph Based Tools (GraBaTs 2012)*, volume 52, pages 1–12. EC-EASST, 2012.

[19] Gabriel Tamura, Norha M. Villegas, Hausi A. Müller, JoãoPedro Sousa, Basil Becker, Gabor Karsai, Serge Mankovskii, Mauro Pezzè, Wilhelm Schäfer, Ladan Tahvildari, and Kenny Wong. Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems. In Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science (LNCS)*, pages 108–132. Springer, January 2013.

[20] Matthias Tichy, Basil Becker, and Holger Giese. Component Templates for Dependable Real-Time Systems. In Andy Schürr and Albert Zündorf, editors, *Proceedings of the 2nd International Fujaba Days 2004, Darmstadt, Germany*, volume tr-ri-04-253 of *Technical Report*, pages 27–30. University of Paderborn, September 2004.

[21] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. Incremental Model Synchronization for Efficient Run-time Monitoring. In Nelly Bencomo, Gordon Blair, Robert France, Cedric Jeanneret, and Freddy Munoz, editors, *Proceedings of the 4th International Workshop on Models@run.time at the 12th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009), Denver, Colorado, USA*, volume 509 of *CEUR Workshop Proceedings*, pages 1–10. CEUR-WS.org, October 2009. (best paper).

[22] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. Model-Driven Architectural Monitoring and Adaptation for Autonomic Systems. In *Proceedings of the 6th IEEE/ACM International Conference on Autonomic Computing and Communications (ICAC 2009), Barcelona, Spain*, pages 67–68. ACM, 15-19 June 2009.

[23] Thomas Vogel, Stefan Neumann, Stephan Hildebrandt, Holger Giese, and Basil Becker. Incremental Model Synchronization for Efficient Run-Time Monitoring. In Sudipto Ghosh, editor, *Models in Software Engineering,*

Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers, volume 6002 of *Lecture Notes in Computer Science (LNCS)*, pages 124–139. Springer-Verlag, April 2010.

Bibliography

[24] Lucia Acciai and Michele Boreale. Deciding Safety Properties in Infinite-State Pi-Calculus Via Behavioural Types. *Information and Computing*, 212:92–117, 2012.

[25] Rajeev Alur, C. Coucoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid Automata: an algorithmic approach to the specification and verification of hybrid systems. In R.L. Grossmann, A. Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems I*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer Verlag, 1993.

[26] 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.

[27] Benjamin Aminof, Orna Kupferman, and Aniello Murano. Improved Model Checking of Hierarchical Systems. In Gilles Barthe and Manuel Hermenegildo, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *Lecture Notes in Computer Science*, pages 61–77. Springer Berlin / Heidelberg, 2010.

[28] 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.

[29] Jesper Andersson, Rogério de Lemos, Sam Malek, and Danny Weyns. Modeling Dimensions of Self-Adaptive Software Systems. In Betty H. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 27–47. Springer Berlin / Heidelberg, 2009.

[30] Robert Baillargeon. Vehicle System Development: A Challenge of Ultra-Large-Scale Systems. In *ULS ""07: Proceedings of the International Workshop on Software Technologies for Ultra-Large-Scale Systems*, page 5, Washington, DC, USA, 2007. IEEE Computer Society.

[31] Paolo Baldan, Andrea Bracciali, and Roberto Bruni. A semantic framework for open processes. *Theoretical Computer Science*, 389(3):446 – 483, 2007.

[32] 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.

[33] Paolo Baldan, Andrea Corradini, and Barbara König. A framework for the verification of infinite-state graph transformation systems. *Information and Computation*, 206(7):869–907, 2008.

[34] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward Open-World Software: Issue and Challenges. *Computer*, 39(10):36–43, 2006.

[35] Luciano Baresi, Reiko Heckel, Sebastian Thone, and Daniel Varró. Modeling and Validation of Service-Oriented Architectures: Application vs. Style. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 68–77, New York, NY, USA, 2003. ACM.

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

[37] Jörg Bauer, Iovka Boneva, Marcos Kurbán, and Arend Rensink. A Modal-Logic Based Graph Abstraction. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *Graph Transformations*, volume 5214 of *Lecture Notes in Computer Science*, pages 321–335. Springer Berlin / Heidelberg, 2008.

[38] Jörg Bauer and Reinhard Wilhelm. Static Analysis of Dynamic Communication Systems by Partner Abstraction. In *Proceedings of the 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007*, volume 4634 of *Lecture Notes in Computer Science*, pages 249–264. Springer Berlin / Heidelberg, 2007.

[39] Gorka Benguria, Philippe Desfray, Bob Covington, Arne J. Berre, Stephan Roser, Christian Hahn, Michael Pantazoglou, Dumitru Roman, Miahi Moldovan, James Odell, and Andreas Ditze. *UML Profile and Metamodel for Services - for Heterogeneous Architectures (UPMS-HA)*, June 2007. http://www.omg.org/docs/ad/2007-06-02.pdf.

[40] Béatrice Bérard, Michel Bidoit, Alain Finkel, FranÇois Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, and Pierre McKenzie. *Systems and Software Verification*. Springer, Berlin / Heidelberg, 2001.

[41] Dirk Beyer, Arindam Chakrabarti, A. Henzinger Thomas, and Sanjit A. Seshia. An Application of Web-Service Interfaces. In *Proceedings of the 2007 IEEE International Conference on Web Services (ICWS 2007, Salt Lake City, UT, July 9-13)*, pages 831–838. IEEE Computer Society Press, Los Alamitos (CA), 2007.

[42] Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis. Modeling Dynamic Architectures Using Dy-BIP. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Software Composition*, volume 7306 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.

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

[44] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. Modelling and Analyzing Adaptive Self-assembly Strategies with Maude. In *Proceedings of the 9th International Workshop on Rewriting Logic and its Applications (WRLA 2012)*, number 7571 in LNCS, pages 18–138, 2012.

[45] Roberto Bruni, Matthias Hölzl, Nora Koch, Alberto Lluch Lafuente, Philip Mayer, Ugo Montanari, Andreas Schroeder, and Martin Wirsing. A Service-Oriented UML Profile with Formal Support. In Luciano Baresi, Chi-Hung Chi, and Jun Suzuki, editors, *Service-Oriented Computing*, volume 5900 of *Lecture Notes in Computer Science*, pages 455–469. Springer Berlin / Heidelberg, 2009.

[46] Sven Burmester, Holger Giese, and Oliver Oberschelp. Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In J. Braz, H. Araújo, A. Vieira, and B. Encarnacao, editors, *Informatics in Control, Automation and Robotics I*. Springer Verlag, March 2006.

[47] Sven Burmester, Holger Giese, and Matthias Tichy. Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In Uwe Assmann, Arend Rensink, and Mehmet Aksit, editors, *Model Driven Architecture: Foundations and Applications*, volume 3599 of *Lecture Notes in Computer Science (LNCS)*, pages 47–61. Springer Verlag, August 2005.

[48] Sven Burmester, Matthias Tichy, and Holger Giese. Modeling Reconfigurable Mechatronic Systems with Mechatronic UML. In U. Aßmann, editor, *Proc. of Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden*, pages 155–169, June 2004.

[49] Luis M. Camarinha-Matos and Hamideh Afsarmanesh. A comprehensive modeling framework for collaborative networked organizations. *Journal of Intelligent Manufacturing*, 18(5):529–542, 2007.

[50] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Francesco Lo Presti, and Raffaela Mirandola. Towards Self-adaptation for Dependable Service-Oriented Systems. In Rogerio Lemos, Jean-Charles Fabre, Cristina Gacek, Fabio Gadducci, and Maurice Beek, editors, *Architecting Dependable Systems VI*, volume 5835 of *Lecture Notes in Computer Science*, pages 24–48. Springer Berlin / Heidelberg, 2009.

[51] Michel Charpentier. Composing Invariants. In *Proc. of International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 401–421. Springer, 2003.

[52] 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.

[53] 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.

[54] Selim Ciraci, Pim Van den Broek, and Mehmet Akşit. Verifying Runtime Reconfiguration Requirements on UML Models. In Pierre Van de Laar and Teade Punter, editors, *Views on Evolvability of Embedded Systems*, pages 209–225. Springer, 2011.

[55] Antonio Coronato and Giuseppe De Pietro. Tools for the Rapid Prototyping of Provably Correct Ambient Intelligence Applications. *IEEE Transactions on Software Engineering*, 38:975–991, 2012.

[56] Brouno Courcelle. In Jan van Leeuwen, editor, *Handbook of theoretical computer science (vol. B)*, chapter Graph rewriting: an algebraic and logic approach, pages 193–242. MIT Press, Cambridge, MA, USA, 1990.

[57] Mads Dam. On the Decidability of Process Equivalences for the pi-Calculus. *Theoretical Computer Science*, 183(2):215–228, 1997.

[58] 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.

[59] 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.

[60] Jörg Desel and Wolfgang Reisig. Place/transition Petri Nets. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 122–173. Springer Berlin Heidelberg, 1998.

[61] Johannes Dyck. Increasing expressive power of graph rules and conditions and automatic verification with inductive invariants. Master's thesis, Hasso-Plattner-Institut für Softwaresystemtechnik, Universität Potsdam, July 2012.

[62] Hartmut Ehrig, Claudia Ermel, Olga Runge, Antonio Bucchiarone, and Patrizio Pelliccione. Formal Analysis and Verification of Self-Healing Systems. In David S. Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010 Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6013 of *Lecture Notes in Computer Science*, pages 139–153. Springer, 2010.

[63] 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, pages 330–345, 2010.

[64] Thomas Erl. *Service-Oriented Architecture*. Service-Oriented computing series. Prentice Hall, 6 edition, 2005.

[65] 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.

[66] Alessandro Fantechi, Stefania Gnesi, Alessandro Lapadula, Franco Mazzanti, Rosario Pugliese, and Francesco Tiezzi. A Logical Verification Methodology for Service-Oriented Computing. *ACM Transactions on Software Engineering and Methodology*, 21(3):16:1–16:46, July 2012.

[67] José Fiadeiro, Antónia Lopes, and João Abreu. A formal model for service-oriented interactions. *Science of Computer Programming*, 77(5):577 – 608, 2012.

[68] John Fitzgerald, Jeremy Bryans, and Richard Payne. A Formal Model-based Approach to Engineering Systems-of-Systems. In *Collaborative Networks in the Internet of Services - 13th IFIP WG 5.5 Working Conference on Virtual Enterprises, PRO-VE 2012, Bournemouth, UK, October 1-3, 2012. Proceedings*, volume 380 of *IFIP Advances in Information and Communication Technology*, pages 53–62. Springer, 2012.

[69] Howard Foster. Architecture and behaviour analysis for engineering Service Modes. In *PESOS '09: Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[70] Howard Foster, Wolfgang Emmerich, Jeff Kramer, Jeff Magee, David Rosenblum, and Sebastián Uchitel. Model checking service compositions under resource constraints. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007 Dubrovnik, Croatia, September 3-7, 2007*, pages 225–234. ACM, 2007.

[71] 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.

[72] 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.

[73] 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.

[74] Goran Frehse. PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. In *Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 258–273. Springer Berlin / Heidelberg, 2005.

[75] Marcelo Fabian Frias, Juan Pablo Galeotti, Carlos Lopez Pombo, and Nazareno Aguirre. DynAlloy: Upgrading Alloy with actions. In *Proc. of International Conference of Software Engineering*, pages 442–451. ACM, 2005.

[76] Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. Exploiting Over- and Under-Approximations for Infinite-State Counterpart Models. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *ICGT*, volume 7562 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2012.

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

[78] Carlo Ghezzi and Sam Guinea. Run-Time Monitoring in Service-Oriented Architectures. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 237–264. Springer Berlin Heidelberg, 2007.

[79] Holger Giese. *Object-Oriented Design and Architecture of Distributed Systems.* PhD thesis, Westfälische Wilhelms-Universität Münster, Fachbereich Mathematik und Informatik, February 2001.

[80] Holger Giese, Bernhard Rumpe, Bernhard Schätz, and Janos Sztipanovits. Science and Engineering of Cyber-Physical Systems (Dagstuhl Seminar 11441). *Dagstuhl Reports*, 1(11):1–22, 2012.

[81] 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.

[82] Sylvain Halle, Roger Villemaire, and Omar Cherkaoui. Specifying and Validating Data-Aware Temporal Web Service Properties. *IEEE Transactions on Software Engineering*, 35(5):669–683, 2009.

[83] Reiko Heckel and Sebastian Thone. Behavior-Preserving Refinement Relations between Dynamic Software Architectures. In Jose Luiz Fiadeiro, Peter D. Mosses, and Fernando Orejas, editors, *Recent Trends in Algebraic Development Techniques*, volume 3423 of *Lecture Notes in Computer Science*, pages 1–27. Springer Berlin / Heidelberg, 2005.

[84] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In Norman Meyrowitz, editor, *Object-Oriented Programming Systems, Languages and Applications (OOPSLA/ECOOP'90), Ottawa, Canada, October 21-25 1990*, volume 25 of *ACM SIGPLAN notices*, pages 169–180, New York, NY, USA, October 1990. ACM.

[85] Thomas Henzinger. The Theory of Hybrid Automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.

[86] Tom Henzinger. The theory of hybrid automata. In *Proceedings of 11th Symposium on Logic in Computer Science (LICS '96)*, pages 278–292. IEEE Computer Press, 1996.

[87] Martin Hirsch, Stefan Henkler, and Holger Giese. Modeling Collaborations with Dynamic Structural Adaptation in Mechatronic UML. In *Proc. of the ICSE 2008 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS"08), Leipzig, Germany*. ACM Press, May 2008.

[88] 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.

[89] 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.

[90] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[91] Eric Y.T. Juan, Jeffrey J. Pl Tsai, and Tadao Murata. Compositional verification of concurrent systems using Petri-net-based condensation rules. *ACM Transactions on Programming Languages and Systems*, 20(5):917–979, September 1998.

[92] Slim Kallel, Mohamed Hadj Kacem, and Mohamed Jmaiel. Modeling and enforcing invariants of dynamic software architectures. *Software and Systems Modeling*, 11(1):127 – 149, 2012.

[93] Jeffrey O. Kephart and David Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003.

[94] 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.

[95] Hans J. Köhler, Ulrich A. Nickel, Jörg Niere, and Albert Zündorf. Integrating UML Diagrams for Production Control Systems. In *Proc. of the* $22^n d$ *International Conference on Software Engineering (ICSE), Limerick, Ireland*, pages 241–251. ACM Press, 2000.

[96] Jeff Kramer and Jeff Magee. Self-Managed Systems: an Architectural Challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[97] 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.

[98] Edward A. Lee. Cyber Physical Systems: Design Challenges. Technical Report UCB/EECS-2008-8, EECS Department, University of California, Berkeley, January 2008.

[99] Meir Manny Lehman. Software's Future: Managing Evolution. *IEEE Software*, 15(01):40–44, 1998.

[100] Grace A. Lewis, Edwin Morris, Patrick R. H. Place, Soumya Simanta, and Dennis B. Smith. Requirements engineering for systems of systems. In *Systems Conference, 2009 3rd Annual IEEE*, pages 247–252, 2009.

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

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

[103] Mark W. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284, 1998.

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

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

[106] Naftaly H. Minsky and Victoria Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(3):273–305, 2000.

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

[108] Peter Ölveczky and José Meseguer. Specification and Analysis of Real-Time Systems Using Real-time Maude. In Tiziana Margaria and Michel Wermelinger, editors, *Proceedings on Fundamental Approaches to Software Engineering (FASE2004)*, volume 2984 of *Lecture Notes in Computer Science*. Spinger-Verlag Heidelberg, 2004.

[109] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 899–910, New York, NY, USA, 2008. ACM.

[110] Susan Owicki and Leslie Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):455–495, 1982.

[111] David Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th International Conference on Software Engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[112] Doron Peled and Thomas Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243 – 246, 1997.

[113] Karl-Heinz Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Department of Computing Science, University of Oldenburg, Oldenburg, 2009.

[114] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE Computer Press, November 1977.

[115] Rodrigo Ramos, Augusto Sampaio, and Alexandre Mota. Conformance notions for the coordination of interaction components. *Science of Computer Programming*, 75(5):350–373, May 2010.

[116] Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working With Objects - The OOram Software Engineering Method*. Manning Publications, 1996.

[117] Arend Rensink. Towards Model Checking Graph Grammars. In Michael Leuschel, S. Gruner, and S. Lo Presti, editors, *3rd Workshop on Automated Verification of Critical Systems (AVoCS)*, Technical Report DSSE–TR–2003–2, pages 150–160. University of Southampton, 2003.

[118] Arend Rensink and Eduardo Zambon. Neighbourhood Abstraction in GROOVE - Tool Paper. In Juan de Lara and Daniel Varró, editors, *Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)*, Electronic Communications of the EASST, Berlin, 2010. European Association for the Study of Science and Technology.

[119] Richard Torbjørn Sanders, Humberto Nicolás Castejón, Frank Kraemer, and Rolv Bræk. Using UML 2.0 Collaborations for Compositional Service Specification. In *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 460–475. Springer Berlin / Heidelberg, 2005.

[120] Constantin Serban, Wenxuan Zhang, and Naftaly H. Minsky. A decentralized mechanism for application level monitoring of distributed systems. In *Proceedings of 5th International Conference on Collaborative Computing: Networking, Applications and Worksharing, CollaborateCom*, pages 1–10. IEEE, 2009.

[121] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[122] Christian Stahl and Karsten Wolf. Deciding Service Composition and Substitutability Using Extended Operating Guidelines. *Data & Knowledge Engineering*, 68(9):819–833, September 2009.

[123] D. Steenken and D. Wonisch. Using Shape Analysis to verify Graph Transformations in Model Driven Design. In *IEEE 9th International Conference on Industrial Informatics (INDIN 2011)*, July 2011.

[124] Mark-Oliver Stehr, Carolyn Talcott, John Rushby, Pat Lincoln, Minyoung Kim, Steven Cheung, and Andy Poggio. Fractionated Software for

Networked Cyber-Physical Systems: Research Directions and Long-Term Vision. 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 110–143. Springer Berlin / Heidelberg, 2011.

[125] Jun Sun, Yang Liu, Jin Song Dong, Yanhong A. Liu, Ling Shi, and Étienne André. Modeling and verifying hierarchical real-time systems using stateful timed CSP. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):3:1–3:29, 2013.

[126] Matthias Tichy, Holger Giese, and Andreas Seibel. Story Diagrams in Real-Time Software. In Holger Giese and Bernhard Westfechtel, editors, *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany*, volume tr-ri-06-275 of *Technical Report*, pages 15–22. University of Paderborn, September 2006.

[127] Ricardo Valerdi, Elliot Axelband, Thomas Baehren, Barry Boehm, Dave Dorenbos, Scott Jackson, Azad Madni, Gerald Nadler, Paul Robitaille, and Stan Settles. A research agenda for systems of systems architecting. *International Journal of System of Systems Engineering*, 1(1-2):171–188, 2008.

[128] Willem-Jan van den Heuvel, Olaf Zimmermann, Frank Leymann, Patricia Lago, Ina Schieferdecker, Uwe Zdun, and Paris Avgeriou. Software service engineering: Tenets and challenges. In *PESOS '09: Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 26–33, Washington, DC, USA, 2009. IEEE Computer Society.

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

[130] 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.

[131] Heike Wehrheim, D. Steenken, and D. Wonisch. Towards A Shape Analysis for Graph Transformation Systems. *ArXiv e-prints*, October 2010.

[132] Kristopher Welsh and Pete Sawyer. When to Adapt? Identification of Problem Domains for Adaptive Systems. In *Requirements Engineering: Foundation for Software Quality*, volume 5025/2008 of *Lecture Notes in Computer Science*, pages 198–203. Springer Berlin / Heidelberg, 2008.

[133] 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.

[134] 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.

[135] Jian Zhang, Heather J. Goldsby, and Betty H. 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.

[136] Paul Ziemann and Martin Gogolla. OCL Extended with Temporal Logic. In Manfred Broy and Alexandre V. Zamulin, editors, *Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 351–357. Springer Berlin Heidelberg, 2003.

List of Figures

List of Tables

Appendix A

Complete Examples

In this chapter we will give an overview of the complete application examples. Obviously, some of the figures we will show here, have already been shown throughout the course of this thesis.

A.1 RailCab System

A.1.1 Timed DistanceCoordination collaboration type

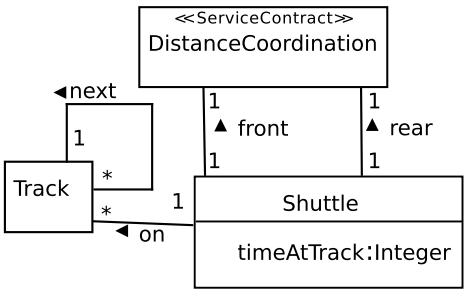


Figure A.1: Ontology for the time RailCab example

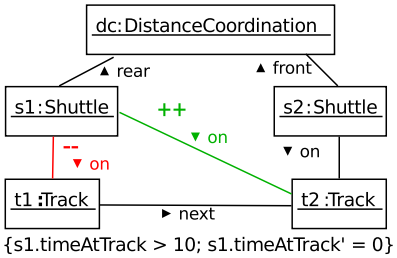


Figure A.2: The Shuttle role's `moveDC` rule

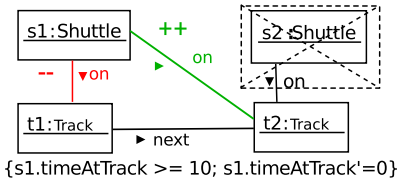


Figure A.3: The Shuttle role's `moveSimple` rule

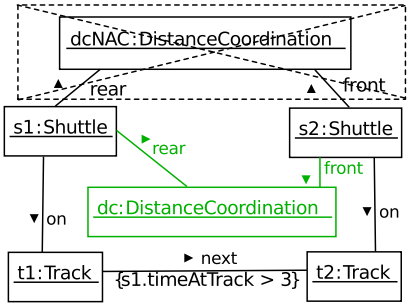


Figure A.4: The Shuttle role's `createDc` rule

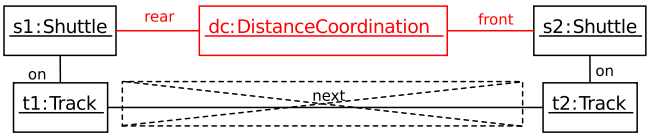


Figure A.5: The Shuttle role's `destroyDC` rule

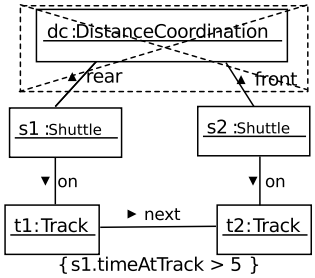


Figure A.6: The `noDc` property

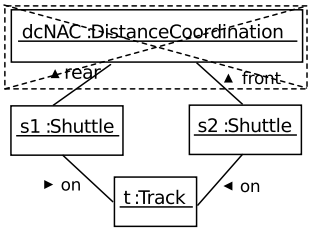


Figure A.7: The `collision` property

A.1.2 DistanceCoordination collaboration type

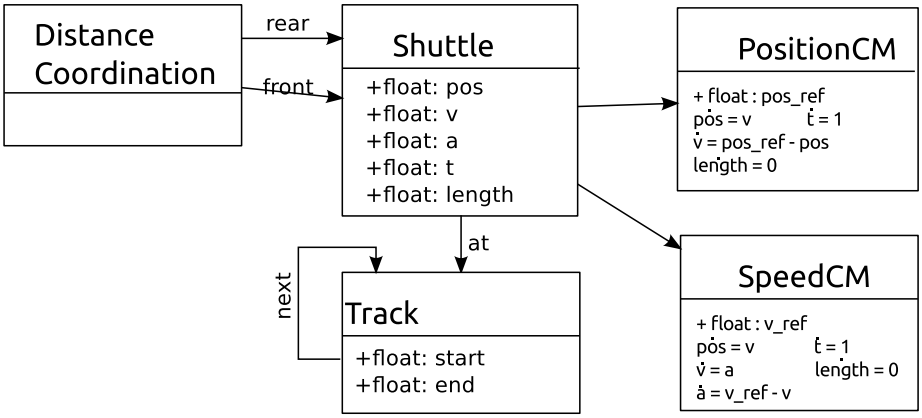


Figure A.8: Ontology used for the RailCab application example - Figure 8.12

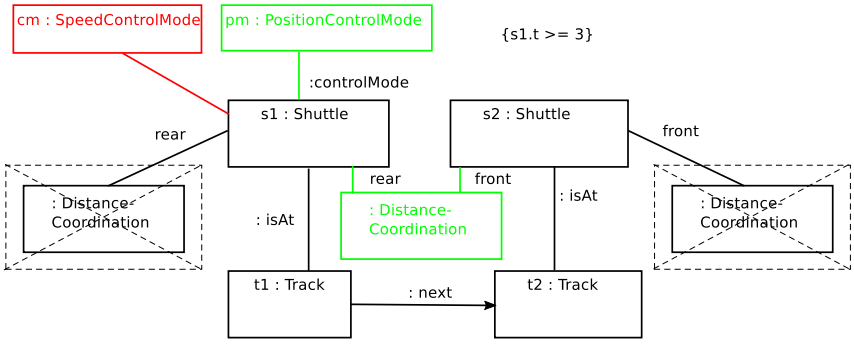


Figure A.9: The DistanceCoordination collaboration type's createDC rule Figure 8.13

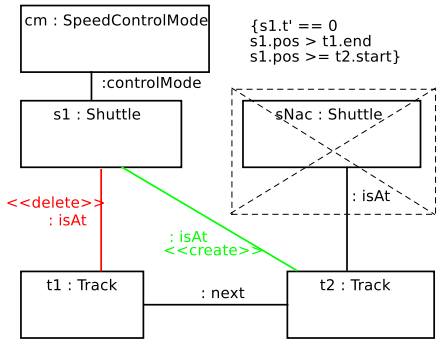


Figure A.10: `move` rule - Figure 8.14(a)

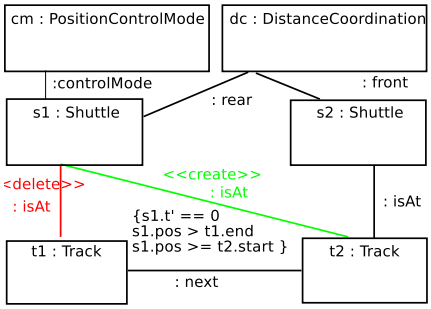


Figure A.11: `moveDC` rule - Figure 8.14(b)

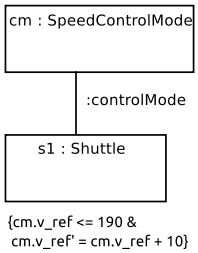


Figure A.12: The `accel` rule - Figure 8.16(a)

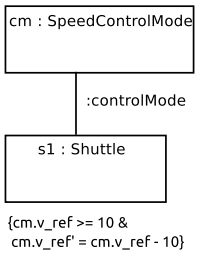


Figure A.13: The `brake` rule - Figure 8.16(b)

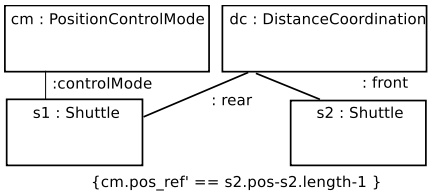


Figure A.14: The `passPos` rule

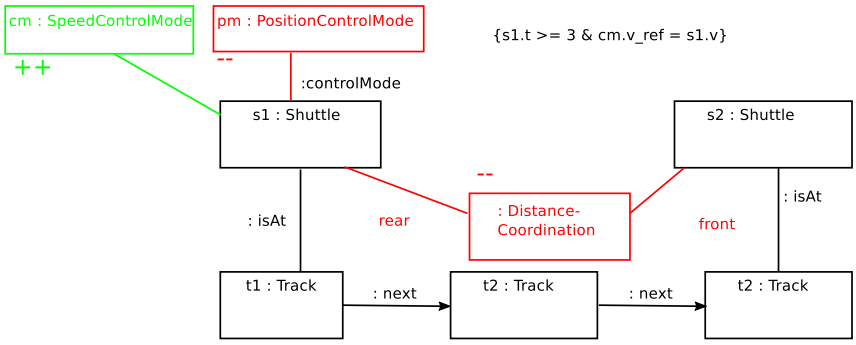


Figure A.15: The `destroyDC` rule

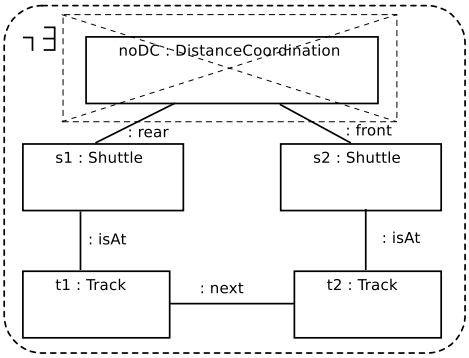


Figure A.16: The `noDC` forbidden property - Figure 8.15(a)

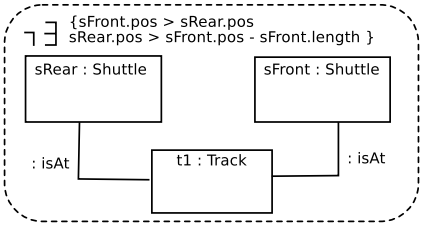


Figure A.17: The `collision` forbidden property - Figure 8.15(b)

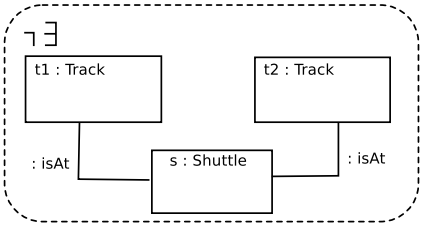


Figure A.18: The `atTwoTracks` forbidden property

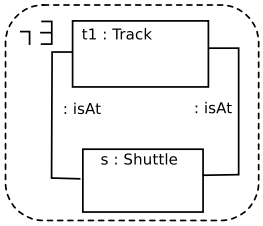


Figure A.19: The `twoIsAt` forbidden property

A.1.3 ShuttleImpl component type

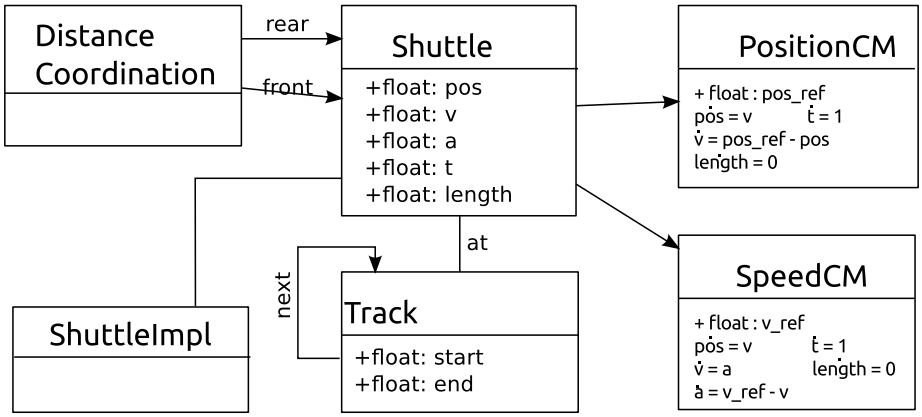


Figure A.20: Ontology used for the RailCab application example - Figure 8.12

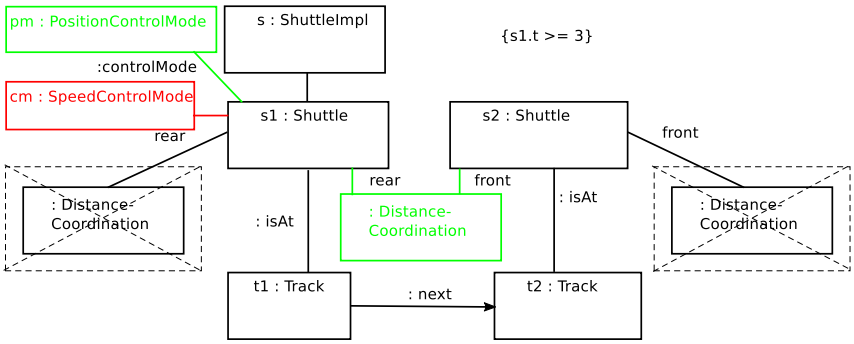


Figure A.21: The DistanceCoordination collaboration type's createDC rule Figure 8.13

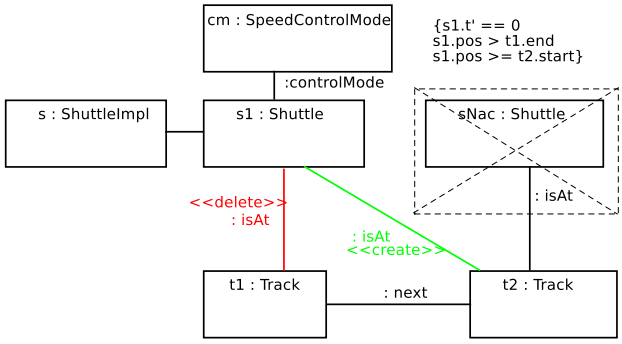


Figure A.22: `move` rule - Figure 8.14(a)

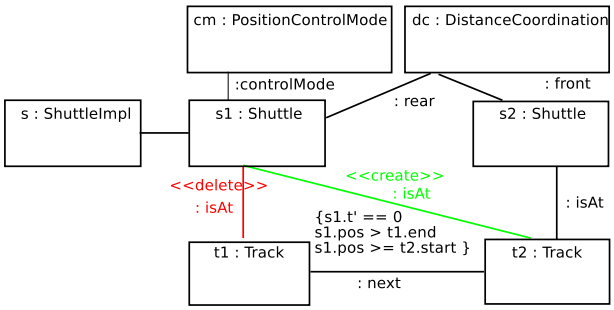


Figure A.23: `moveDC` rule - Figure 8.14(b)

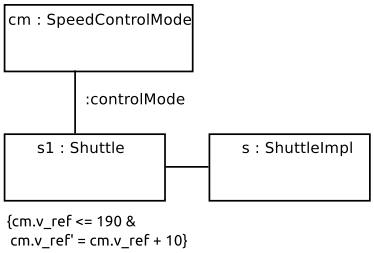


Figure A.24: The `accel` rule - Figure 8.16(a)

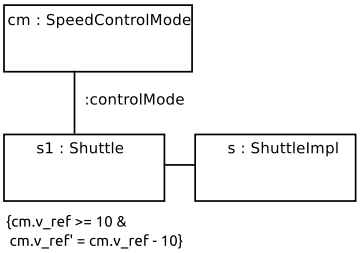


Figure A.25: The `brake` rule - Figure 8.16(b)

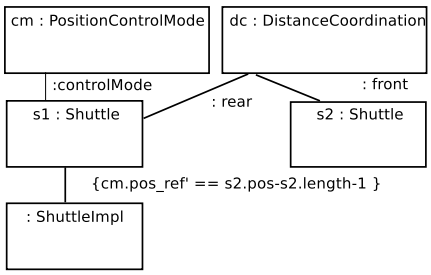


Figure A.26: The `passPos` rule

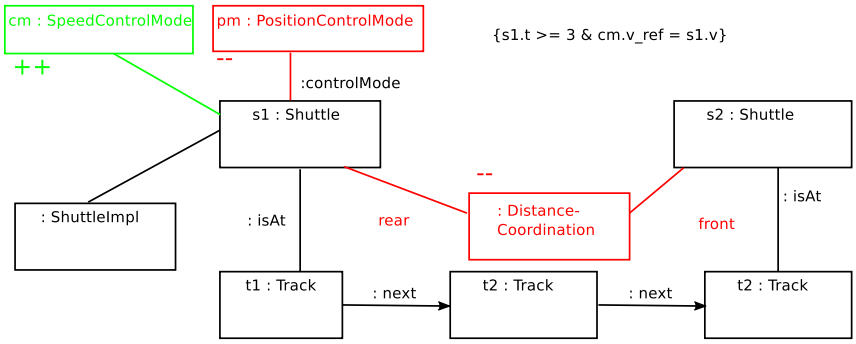


Figure A.27: The `destroyDC` rule

A.2 Supply Chain System

A.2.1 Contract Collaboration

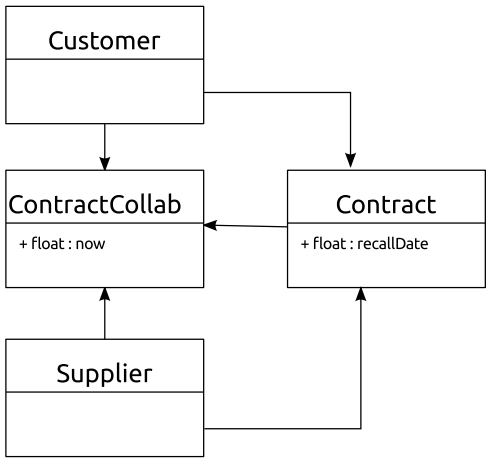


Figure A.28: The ContractCollaboration's class-diagram CD_{CC}

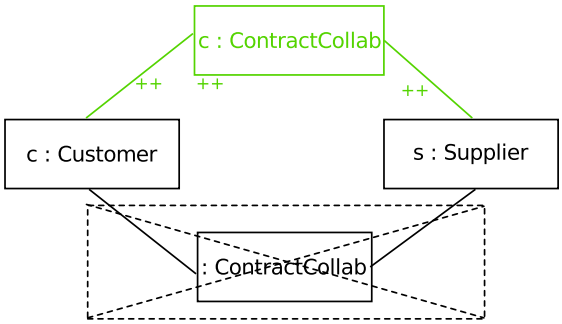


Figure A.29: `createCollab` rule

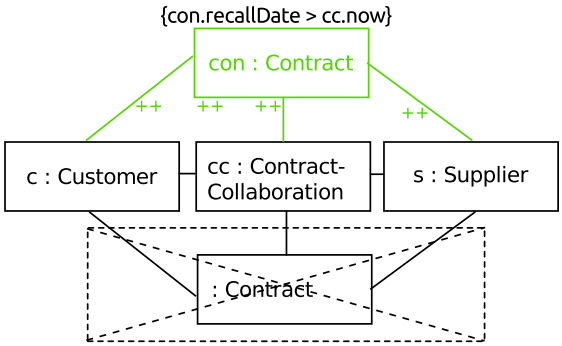


Figure A.30: `createContract` rule - also in Figure 4.4

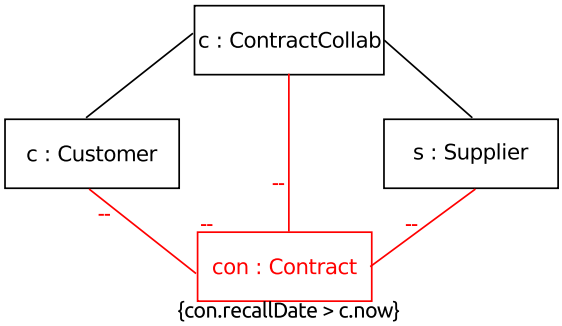


Figure A.31: `deleteContract` rule

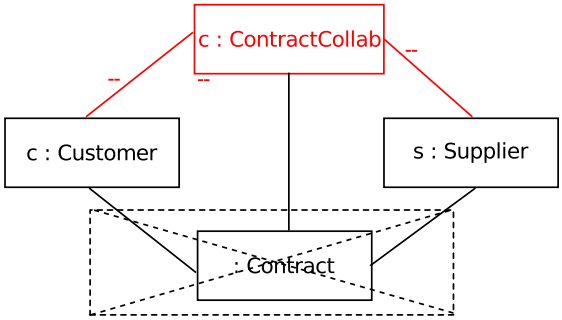


Figure A.32: `destroyCollab` rule

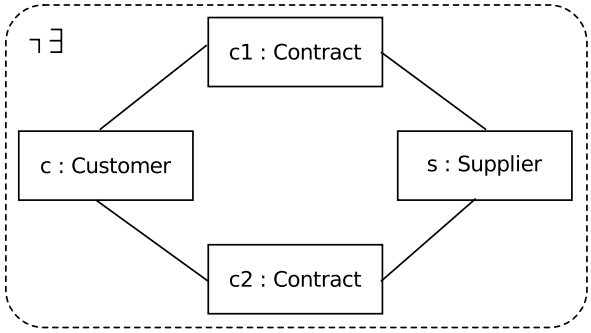


Figure A.33: `notTwoContracts` safety property - also in Figure 4.3

A.2.2 Abstract Factory

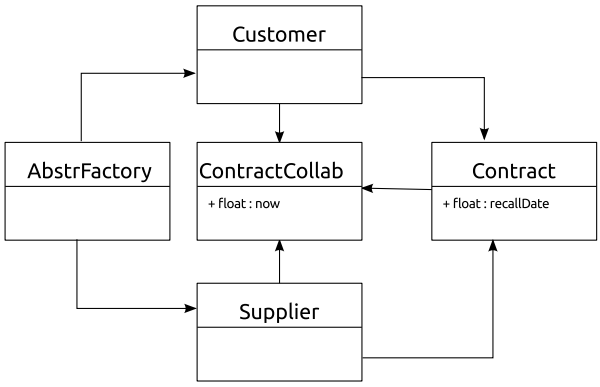


Figure A.34: The AbstrFactory's class diagram CD_{AFac}

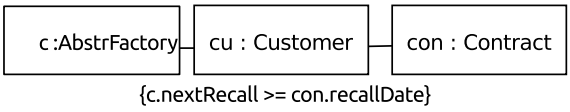


Figure A.35: `custRecall` safety property

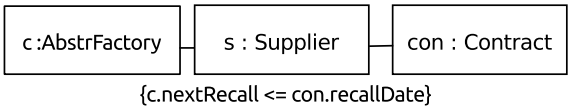


Figure A.36: `supRecall` safety property

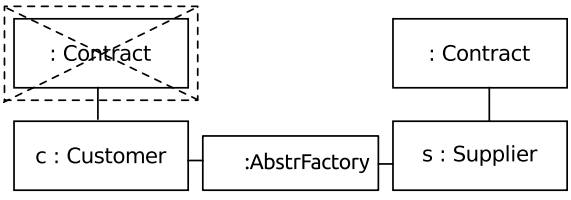


Figure A.37: `noCustContract` safety property

A.2.3 RequestOffer Collaboration

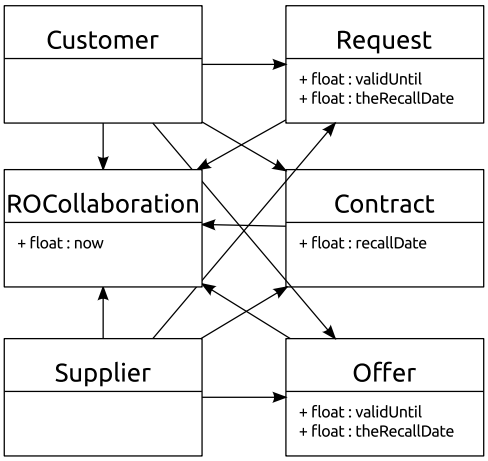


Figure A.38: The RequestOfferCollaboration's class diagram CD_{ROC}

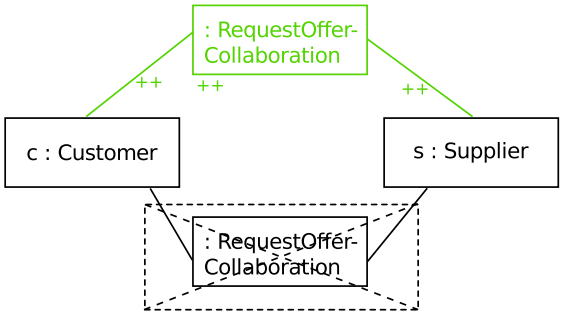


Figure A.39: `createROC` rule

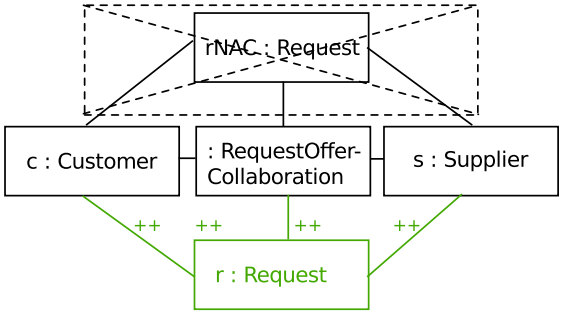


Figure A.40: `createRequest` rule

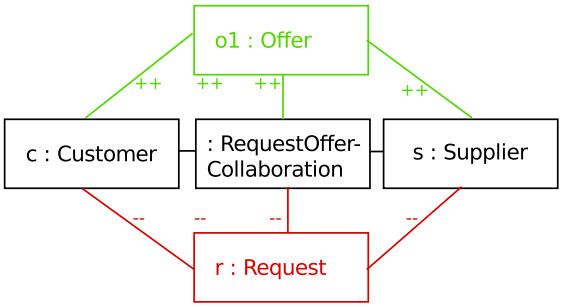


Figure A.41: `makeOffer` rule

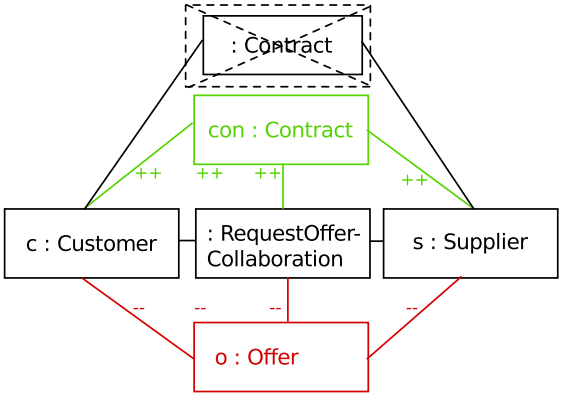


Figure A.42: `createContract` rule

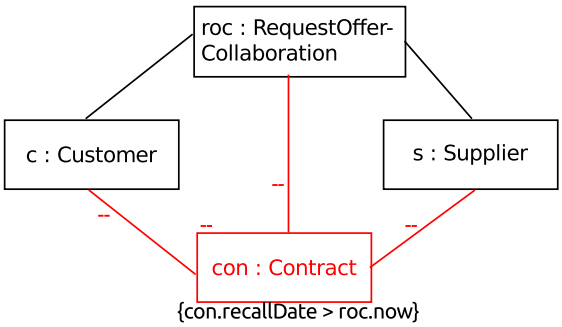


Figure A.43: `deleteContract` rule

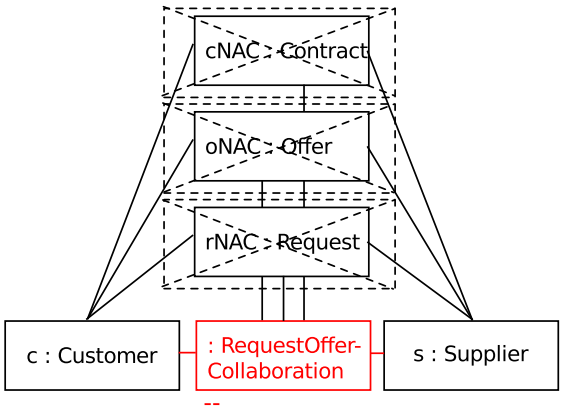


Figure A.44: `destroyROC` rule

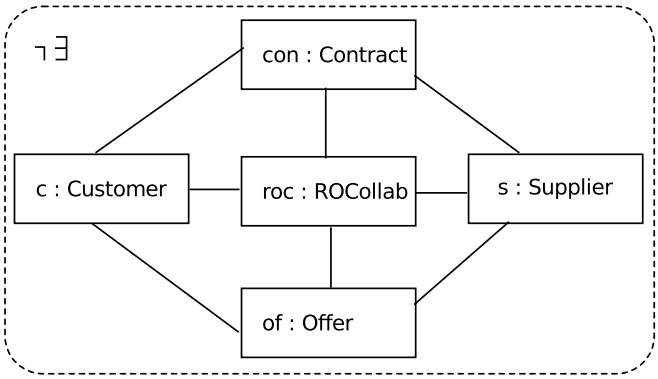


Figure A.45: `noOfandContract` safety property

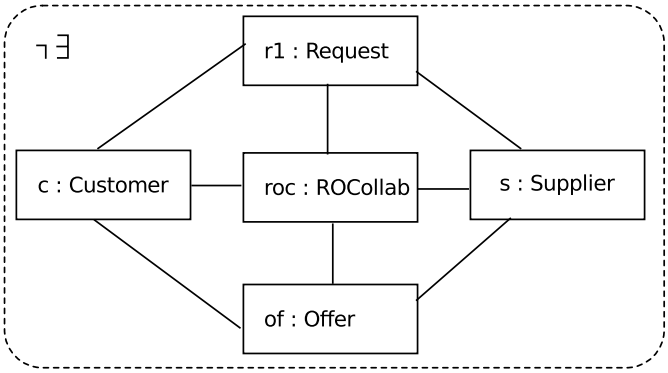


Figure A.46: `noOfandRequest` safety property

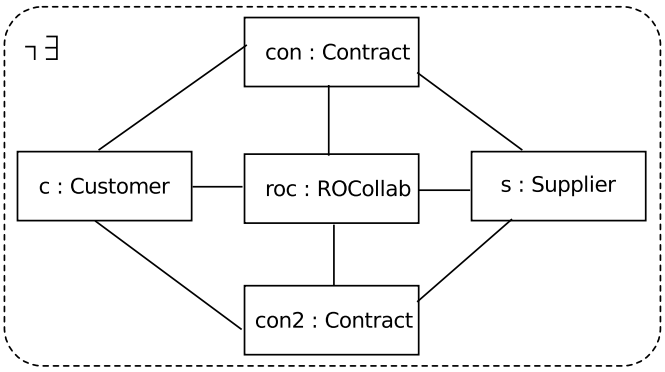


Figure A.47: `noTwoContracts` safety property

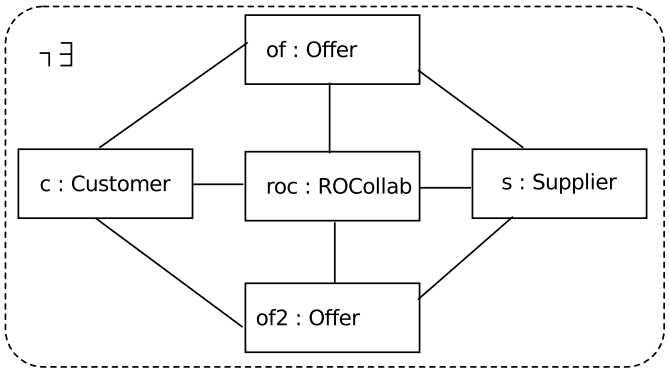


Figure A.48: `noTwoOffers` safety property

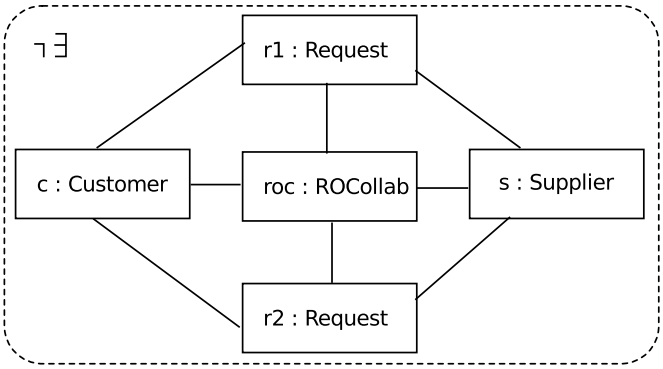


Figure A.49: `noTwoRequests` safety property

A.2.4 Factory

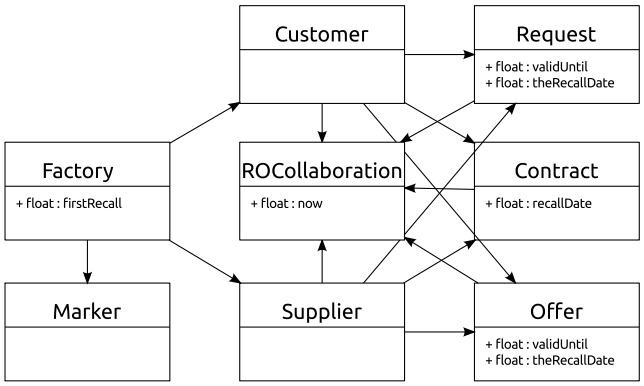


Figure A.50: The Factory's class diagram CD_{Fac}

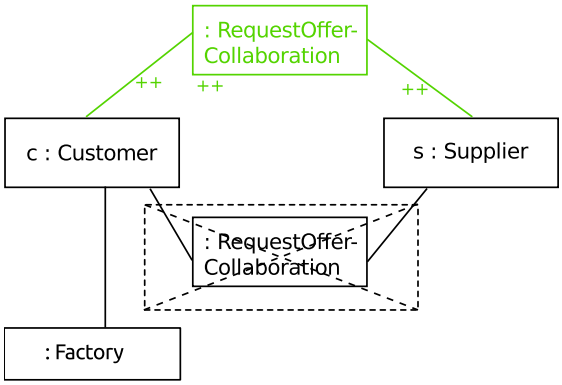


Figure A.51: `createCollab` rule

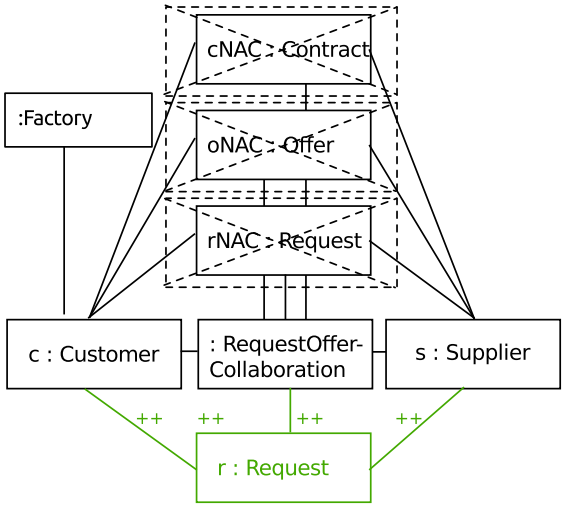


Figure A.52: `sendRequest` rule

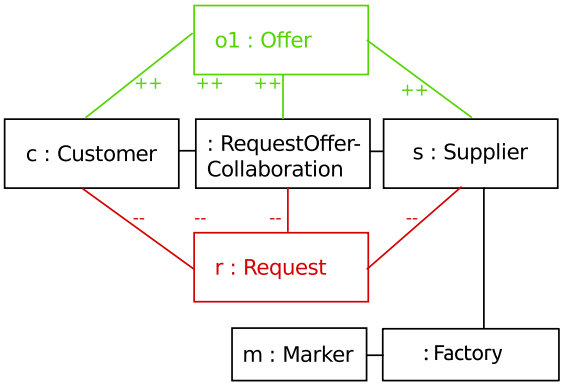


Figure A.53: `makeOffer` rule. Also in Figure 8.3

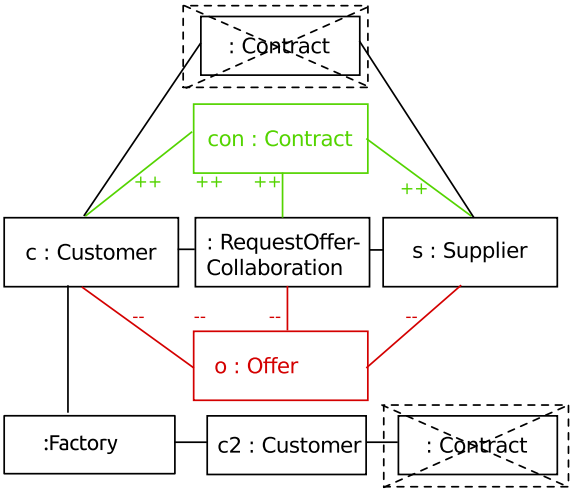


Figure A.54: `createContract` rule

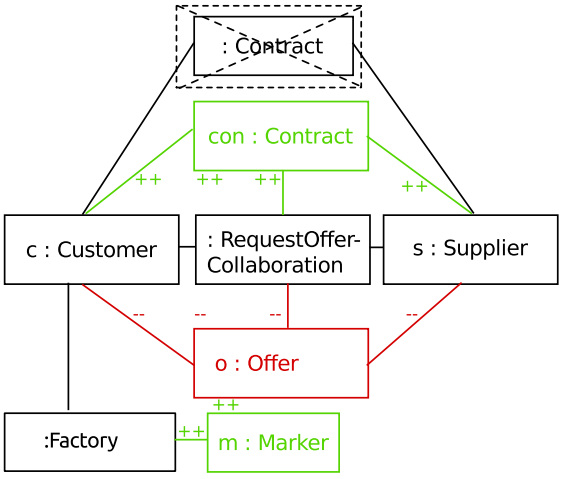


Figure A.55: `createContract2` rule

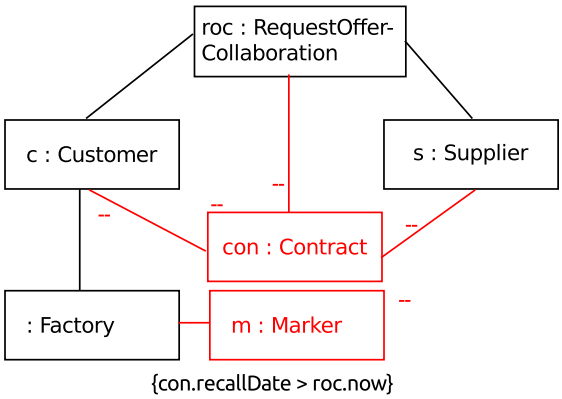


Figure A.56: `deleteContract` rule for the Customer role

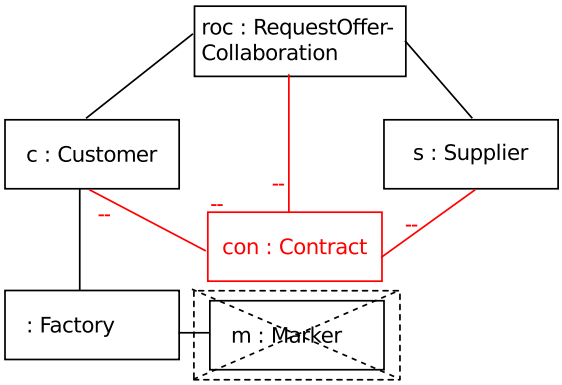


Figure A.57: `deleteOtherContract` rule for the Customer role

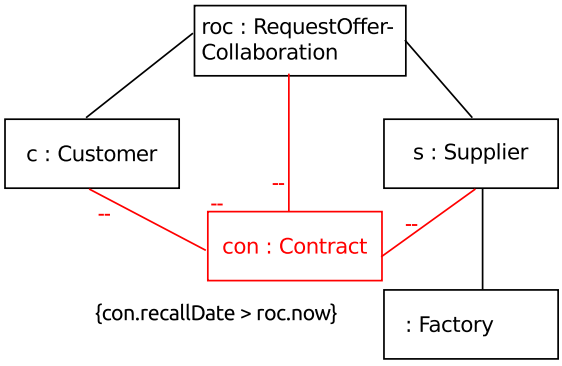


Figure A.58: `deleteContract` rule for the Supplier role

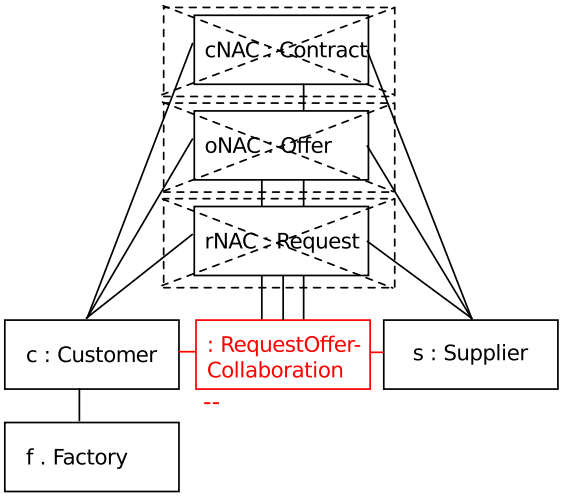


Figure A.59: `destroyCollab` rule for the Customer role

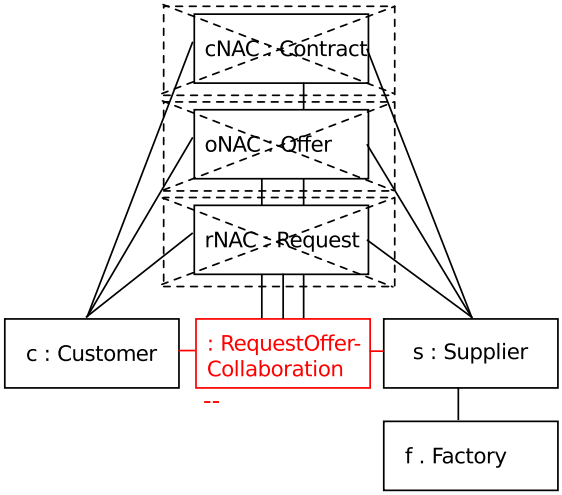


Figure A.60: `destroyCollab` rule for the Supplier role

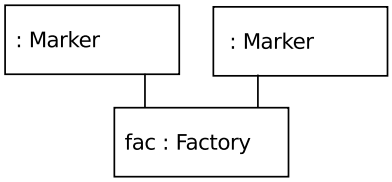


Figure A.61: `comp2Marker` safety property

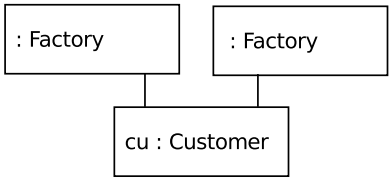


Figure A.62: `cust2Con` safety property

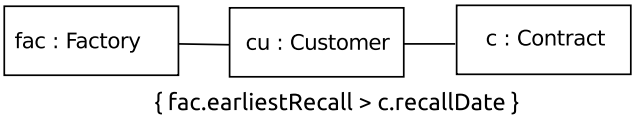


Figure A.63: `custLateRecall` safety property

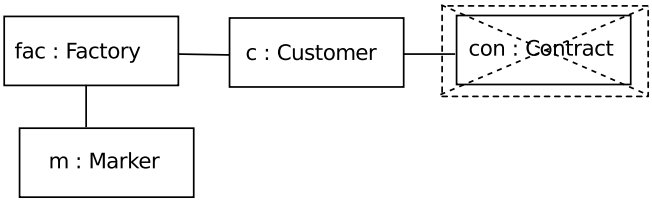


Figure A.64: `markerNoCon` safety property

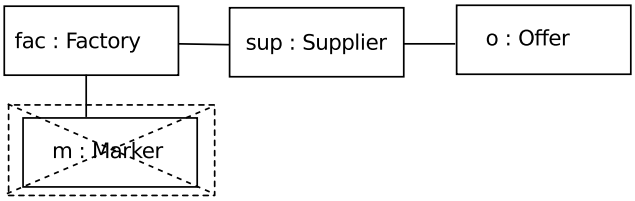


Figure A.65: `ofButNoMarker` safety property

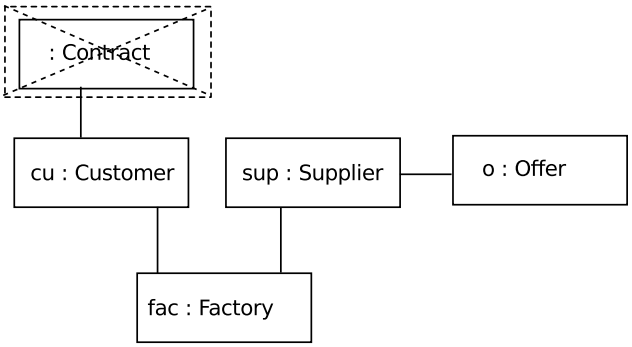


Figure A.66: `offerButNoCust` safety property

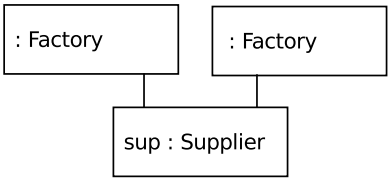


Figure A.67: `sup2Comp` safety property

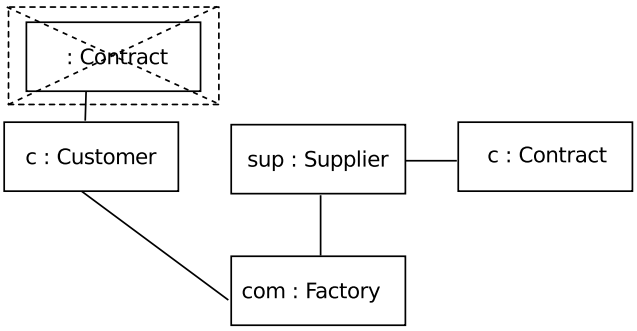


Figure A.68: `supConwithouthCustCon` safety property. See also Figure 8.2

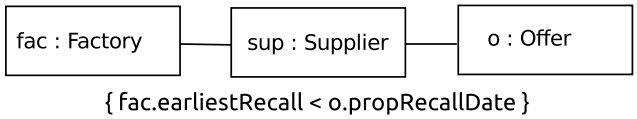


Figure A.69: `supEarlyRecall` safety property

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 $G \rightarrow H$ 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.

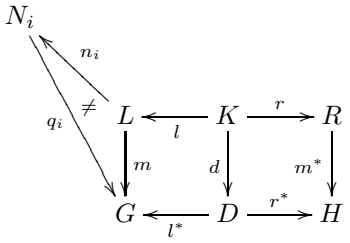
The above definition of graph constraints conforms to the widely used definition that is given in [63] for application conditions with the difference that our definition does not allow for nesting of graph constraints.

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, 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, $e \in \text{ran}(m)$. 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 sates 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

[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. [65]

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 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. [26]. The difference clearly is that the number of attributes or variables, as [26] 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. [26]).

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 $G \overset{P',m}{\Rightarrow} H$ 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.

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) $S = (G_0, R, p, T, \mathcal{R}_u)$ is given as an initial attributed graph G_0, a set of hybrid graph transformation rules R, a transitive preemption function $p : R \mapsto R$, 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 $G_T \overset{r,m}{\Rightarrow} G_Q$ and $\nexists r', m'$ with $p(r', r)$ and $G_T \overset{r',m'}{\Rightarrow} G'_Q$. $(S_P, t, S_Q) \in \delta$ 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

The Invariant Checker's Input Model

In Chapter 7 we introduced our tool — the Invariant Checker — we use to verify safety properties in GTS and HGTS. The verification algorithm have been introduced in Section 6.1 at an abstract level and in Section 7.2 and following we described the implementation aspects. In this chapter we will focus on the input model the Invariant Checker expects and how GTS including forbidden pattern are specified in that input model.

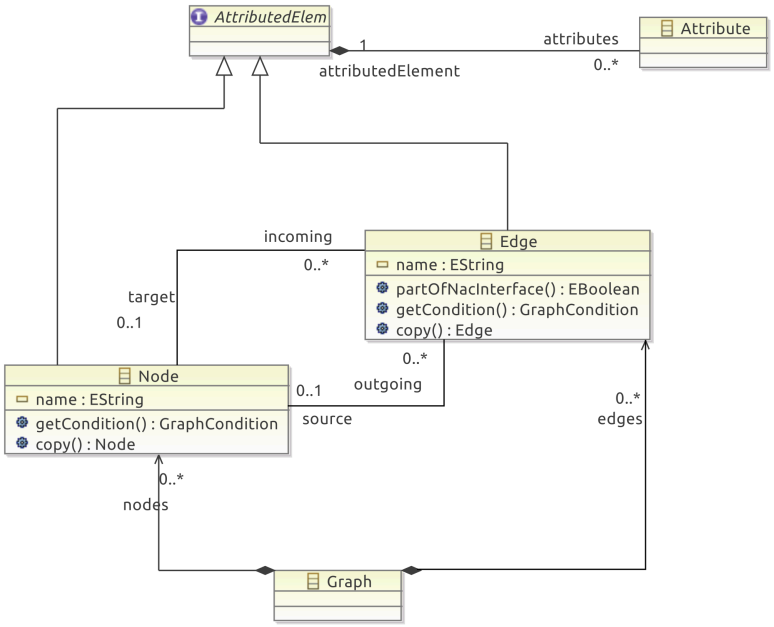


Figure C.1: SaMiGra's graph package

The Invariant Checker can verify GTS that are specified using the SaMiGra

meta-model. SaMiGra is a general purpose meta-model for the specification of graph transformation based systems. It is implemented using EMF[1]. SaMiGra consists of several packages. The most central package is the graph package that allows to specify graphs using nodes and edges. Graphs are typed over type-graphs. A type-graph mainly consists of NodeTypes and EdgeTypes. EdgeTypes can be equipped with source- and target-cardinalities. NodeTypes can have AttributeTypes for which we distinguish between discrete AttributeTypes and ContAttrTypes for continuous attributes. For each AttributeType a Sort and for each continuous attributes also a derivation has to be specified. The Sorts are not predefined and the user can model them together with a fitting algebra and signature[2]. However for the typical needs a standard-signature and -algebra is part of SaMiGra and can be used from any SaMiGra model. Although control modes can be specified in SaMiGra (simply specifying a node), the laws for the attributes' timely derivation can not be specified using SaMiGra, yet.

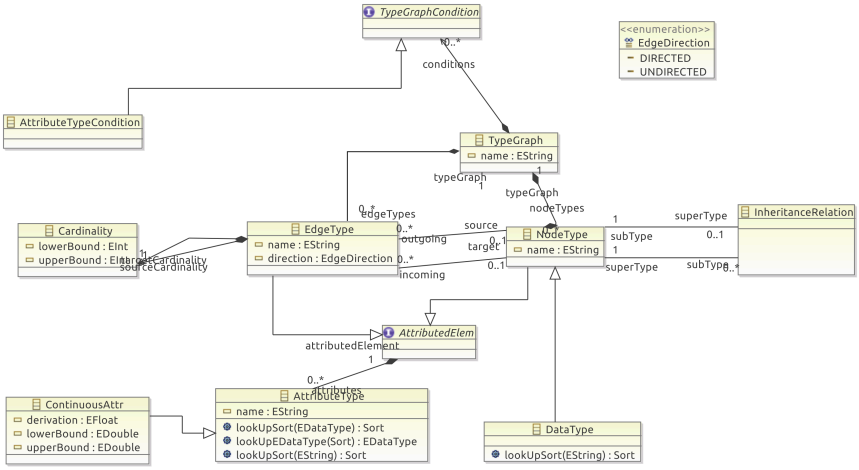


Figure C.2: SaMiGra's type-graph package

SaMiGra also allows the specification of graph transformation rules. A graph transformation rule is specified as a left- and a right hand side, whereas the LHS can contain DeletedNodes and PreservedNodes and the RHS can contain CreatedNodes and PreservedNodes (same holds for edges). For a PreservedNode or a PreservedEdge the corresponding element in the graph rule's other side has to be specified. This directly gives us the morphism between a graph rule's two sides. The type of nodes and edges is mandatory. In conformance with Definition B.5 a graph transformation rule can be augmented with NACs. In SaMiGra NACs are encoded as general nested graph conditions, but the Invariant Checker currently only supports NACs. An application condition is considered a NAC if it is a NegatedCondition, having a nested Quantification, with the quantor being set to EXISTS, a context graph no further nesting. The the nesting reached it's end is modeled through a TerminationCondition.

[1] http://www.eclipse.org/modeling/emf

[2] Although it is possible to define custom signatures and algebras, the Invariant Checker currently does only support the standard-algebra and -signature that are part of the SaMiGra plug-in.

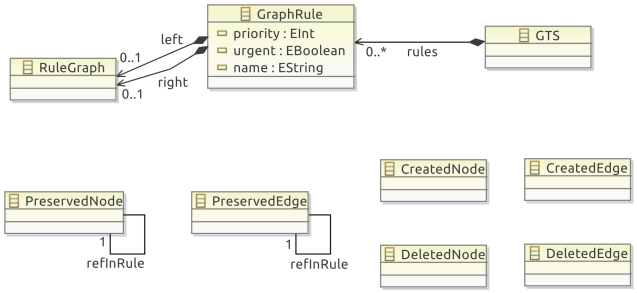


Figure C.3: SaMiGra's rules package

Forbidden properties have to be specified at the type-graph level. This is useful as their intention is to restrict the instances of the type-graph. Therefore, each type-graph can have a number of TypeGraphConditions. For TypeGraphConditions SaMiGra allows the same level of expressiveness as for graph conditions in graph transformations rules.

A meta-model can be used to express a lots of the different constraints, that a correct instance model has to fulfill. However, some constraints cannot be expressed in a meta-model. E.g. the relation between a Node and an Attribute has to reflected by the corresponding NodeType and AttributeType, or that an Edge must not connect two Nodes that are children of different Graphs. These two and a lot more subtle constraints have been specified in a validation plug-in that can be used to validate SaMiGra-models. An Invariant Checker specific plug-in checks that only these constructs of SaMiGra are used that the Invariant Checker understands.