Doctoral theses at NTNU, 2008:228

Frank Alexander Kraemer

Engineering Reactive Systems

A Compositional and Model-Driven Method
Based on Collaborative Building Blocks

Frank Alexander Kraemer

Engineering Reactive Systems

A Compositional and Model-Driven Method Based on Collaborative Building Blocks

Thesis for the degree of philosophiae doctor

Trondheim, July 2008

Norwegian University of Science and Technology

Faculty of Information Technology, Mathematics and Electrical Engineering

Department of Telematics

NTNU
Norwegian University of Science and Technology

Thesis for the degree of philosophiae doctor

Faculty of Information Technology, Mathematics and
Electrical Engineering
Department of Telematics

ISBN 978-82-471-1146-8 (printed version)
ISBN 978-82-471-1147-5 (electronic version)
ISSN 1503-8181

Doctoral theses at NTNU, 2008:228

Printed by NTNU-trykk

TO MY FOLKS
AND FAMILY.

ABSTRACT

This thesis introduces SPACE, an engineering method for reactive systems which enables the rapid composition of services from reusable building blocks. In contrast to traditional approaches which often focus on reuse of separate system components, we use collaborations as the main specification units and building blocks. Collaborations are distributed, stateful functions related to a certain task. They span across several components and describe both the local behavior of all participating components as well as the necessary interactions in an explicit form. We express collaborations by a combination of UML 2.0 collaborations and activities. To encapsulate their internals, we use a special form of UML state machines to describe their externally visible behavior, so that building blocks can be composed without understanding inner details. This opens up new possibilities for the reuse of specifications, since solutions to specific tasks that incorporate several components can be encapsulated within self-contained building blocks. We describe how collaborative building blocks can be composed by means of UML activities. There may be arbitrary composition levels, and development may follow a top-down or bottom-up approach. In the end, a complete system specification is obtained. To implement this specification, we have developed an automated model transformation that synthesizes executable state machines from the collaborations, which are implemented by code generation.

As a semantic foundation to reason about the correctness of our approach, we use temporal logic, and in particular the compositional Temporal Logic of Actions, cTLA. Since we formalize collaborations as cTLA processes which are composed by joint actions, the property of superposition holds for them. This means that properties of a collaborative building block are also maintained by a system composed from it. This makes the incremental verification of systems possible: during the analysis of a collaboration composed from others, we only use the abstract external description of the sub-collaborations. This reduces the state space during model checking, and we only need to check one composition level at a time, that is, each collaboration separately. In addition, since we also formalized the executable state machines for the execution, the model transformation from collaborations to components corresponds to a formal refinement step, so that the correctness of the transformation is ensured as well.

The method is supported by a set of tools. Besides the quick composition of systems in a drag-and-drop like manner, the tools create the state machines and provide the automated analysis of collaborations and their composition on the basis of incremental model checking.

"I'm writing a book on magic,"

I explain, and
I'm asked, "Real magic?"
By real magic people mean miracles,
thaumaturgical acts, and supernatural powers.
"No," I answer, "Conjuring tricks, not real magic."
Real magic, in other words, refers to the magic that is not real,
while the magic that is real, that can actually be done, is not real magic.

Lee Siegel, Net of Magic: Wonders and Deceptions in India. Univ. of Chicago Press, 1991.

PREFACE

About the Magic in this Book*

*And Why You Should Read It

If you hold a book of some hundred pages in your hand, it is your good right to ask: "Why should I read this one?" After all, to say it with the words of the citation on the previous page, this book contains nothing but a handful of tricks to build reactive systems with a little less headache.

But is it really just tricks?

For everybody who spends a substantial amount of time with learning, it is very rewarding to get the time and money to learn some more and eventually apply the learned to solve some real problems. With this background, my major motivation behind this thesis was to find a method that is useful and practically applicable. This is hard to achieve, because it is difficult to estimate how a method turns out in the end, how it will be adopted, and, above all, how it meets the changing demands of engineering and technology.

To reach this goal, I thought it would be most fruitful to combine existing approaches, techniques, logics and languages in a new constellation. My primary concern was to enable an effective form of reuse, because, as D'Souza and Wills describe in their book on Catalysis, *reuse does not mean that you can copy-and-paste code.* Effective reuse means to reuse knowledge, working designs and analytic results in such a manner that one does not always have to understand all details of the work already done. If one cares about reuse, one also has to carefully handle composition, so that reusable elements can be combined effectively to get something valuable. Indeed, it should just work like constructing with LEGO bricks. However, while this analogy is used frequently, it is rarely implemented right. Often, one still has to know too much about the details of the building blocks, or the task of connecting them is very difficult in itself.

Apart from this strong focus on reuse and composition, I tried to keep balance between different aspects of a method in general:

- A balance between views and perspectives, and different levels of abstractions, so that design decisions are made at the right places. This also means that details should not call for so much attention that the big picture is lost, but exactly so much that they are handled correctly.
- A balance between theoretical and practical considerations. Obviously, system specifications as well as the method itself must be sound and correct, and logic is the proper means to ensure that. But we should not use logic for the sake of formalism, and not forget that there are numerous other, quite subtle, practical considerations that need to be made as well.
- A balance between old and new, so that we carefully keep what has worked in the past and do not invent the wheel again, but at the same time not get paralyzed or intimidated by existing paradigms.
- A balance between what computers and what humans can do, so that these two entities are addressed as a team. A method should allow us humans to use our ingenuity, creativity and intelligence, and be assisted by computers in tasks that we are not so good at.

I hope you appreciate these guidelines as well, and see most of them satisfied by the proposed method. You may disagree here and there, and maybe come with some new ideas. I am interested to learn more. And therefore I hope that also some others may take this work as a starting point to solve some more problems, just like I got inspired by so many other works as described in Part I.

So, if you ask me, why you should read this book, then I would probably tell you that, besides the fact that you can build great systems with it, it's the constellation of the method's elements and the balances between them that is the real magic. The rest is just tricks.

Praises and Thanks

I was very lucky having two marvelous supervisors, Rolv Bræk and Peter Herrmann. The combination of their experience and knowledge and the resulting discussions turned out to be very beneficial for this work. That I made the way to Trondheim and stayed there goes considerably onto Rolv's account. He accepted me as student for my Master's thesis in 2002 and was to a large degree the reason why I decided to continue my work with a doctoral degree at the department. Discussing systems engineering with Rolv has always been fun. He has the ability of combining his considerable experience with the will to develop new ideas and provides in that way a very fruitful ground for discussions, which I enjoyed since the first meeting in his office. Peter joined the department right after I finished the lecture of Leslie Lamport's book *Specifying Systems* and concluded some experiments on service specifications. I was more than surprised as

I found out that Peter not only was an expert in temporal logic, but also developed the compositional Temporal Logic of Actions in his own doctoral thesis. That, obviously, helped to get things started very quickly. Peter was immediately interested in the already existing ideas and was willing to get familiar with them. For a researcher who already has enough areas of expertise maybe something that should not be taken for granted in the first place. With Peter I enjoyed especially the process of publishing. He had the patience to teach me how to get ideas written down on an empty piece of paper. So, Rolv and Peter, thank you very much for the work so far. The results in this thesis are also your results. Without any of you, your expertise, continuous help, motivation and patience, they would not have been possible. I hope we'll continue with a lot more stuff. I would also like to thank your wives and Turid and Gaby for their hospitality and some good advices every now and then.

There are of course my other colleagues, a whole bunch of great people. Among them I would like to mention Cyril, Fritjof, Finn Arve, Jacqueline, Judith, Haldor, Hien, Humberto, Leif Arne, Lill, Linda, Mátè, Mazen, Richard, Surya and Thanh. It was fun to work with all of you. Your continuous feedback has influenced this work more than you might think, and your comments on the publications certainly improved both contents and presentation. Thank you all. From the other research groups I want to thank the heads of the departments Svein and Norvald for their efforts, and must not forget Andreas, Harald, Jing and Tord for their excellent company at dinner time. I also want to thank Geir Hasnes, who every now and then dropped by at the department. Geir has a clear opinion about virtually everything and is pretty articulate about what's right and wrong. Listening to him and explaining my ideas in return contributed much to the creative process behind this thesis. Thanks, Geir.

At a visit in Tampere I was able to discuss with Reino Kurki-Suonio and Tommi Mikkonen and their research group (in particular Risto Pitkänen, Perti Kellomäkki, Timo Aaltonen and Mika Katara) many interesting aspects on system engineering, and got valuable feedback on my work. Reino's book on reactive systems not only had a big influence on my way of looking at reactive systems as action systems, but is also written so brilliantly that I stayed motivated enough to learn the theory behind it. At similar occasions I could discuss my work together with Daniel Amyot, Gregor von Bochmann and Gunter Mussbacher in Ottawa, as well as Ina Schieferdecker, Birger Møller-Pedersen, Øystein Haugen, Andreas Prinz and Ketil Stølen, who provided valuable insights and motivation at the right time. A big thank you to all of you!

During the research I worked in two applied research projects, ARTS and its follow-up ISIS, which gave me the opportunity get use cases and motivations from industry colleagues, hereby to mention in particular Telenor with Reidar Martin Svendsen, Steinar Brede, Frode Flægstad and Espen Nersveen, Gintel with Frank Paaske and Ericsson (later Tellu) with Geir Melby and Knut Eilif Husa. Good to know that someone is actually using all that stuff we make.

Thanks go also to the students I supervised over time during their studies, among them Sebjørn Sæther Birkeland, Øystein Gisnås, Lars Erik Karlsen, Alf

Kristian Støyle, Ronnie Nessa and Nina Heitmann. I hope you learned something, I certainly learned from you. A special remark must be dedicated to Vidar Slåtten, who not only shares the same supervisor (Peter), but also had me as a supervisor for his project thesis and master thesis. His commitment and enthusiasm helped very much to integrate model checking into the approach by the tools that were the results of this work. Vidar, thanks a lot! Good luck with your own thesis.

I want to mention also the helpful souls of the institute and faculty, Randi, Mona and Hilde, helping me with a lot of complicated processes there are for a researcher, like completing travel balances, sending faxes and filling out all these documents needed to keep a university running. And of course our two helpful ghosts, Pål and Asbjørn, helping with equally important tasks. Thank you!

Several friends of mine also have their share in the process of writing this thesis over several years. Among them, (in order of appearance): Karim, Thomas, Markus, Thomas, Martina, Daniel, Martin, Matthias, Jeremy, Synnøve, Øyvind, Pål, Christian, Kristin, Alexander, Runar, Øyvind, Andrea, Vasile, Sverre, Dottore Marco, Johan, Jasper, Stefan and Bente. What fantastic friends you are, in any respect. Sorry that I so often had no time for you because I was writing another paper. A special thank goes to Jeremy, who patiently read the entire thesis and freed it from various offenses against the English language.

In the end, it's of course my family that I would like to thank and dedicate the remaining pages of this thesis. Claudia, Zita, Wolfgang Ludwig, Getrud (Dodo) and Änne. I am so glad I have you. Unfortunately my granfather, Wolfgang, had to leave us this spring while I was writing the last pages of my thesis. I owe him very much. He was the main reason why I became, like him and my father, an engineer, probably due to the interest in technology that he encouraged with all the funny things we invented in my early childhood.

And now, read on for some *real* magic.

Frank Alexander Kraemer
Gløshaugen, July 2008

INCLUDED PUBLICATIONS

1. **Service Specification by Composition of Collaborations — An Example**.
   Frank Alexander Kraemer and Peter Herrmann. Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology), IEEE Computer Society, 2006.

2. **Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services**.
   Frank Alexander Kraemer, Peter Herrmann and Rolv Bræk. Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), Volume 4276 of Lecture Notes in Computer Science, Springer, 2006.

3. **Transforming Collaborative Service Specifications into Efficiently Executable State Machines**.
   Frank Alexander Kraemer and Peter Herrmann. Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT), Volume 6 of the Electronic Communications of the EASST, 2007.

4. **Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications**.
   Frank Alexander Kraemer, Rolv Bræk and Peter Herrmann. Proceedings of the 13th Int. SDL Forum, Volume 4745 of Lecture Notes in Computer Science, Springer, 2007.

5. **Design of Trusted Systems with Reusable Collaboration Models**.
   Peter Herrmann and Frank Alexander Kraemer. Proceedings of the Joint iTrust and PST Conferences on Trust, Privacy, Trust Management and Security (IFIPTM), Springer, 2007.

6. **Formalizing Collaboration-Oriented Service Specifications using Temporal Logic**.
   Frank Alexander Kraemer and Peter Herrmann. Proceedings of the Networking and Electronic Commerce Research Conference (NAEC) ATSMA, 2007.

7. **Engineering Support for UML Activities by Automated Model-Checking — An Example**.
   Frank Alexander Kraemer, Vidar Slåtten and Peter Herrmann. Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE), 2007.

8. **Arctis and Ramses: Tool Suites for Rapid Service Engineering**.
   Frank Alexander Kraemer. Proceedings of NIK-2007 (Norsk informatikkonferanse), 2007.

CONTENTS

Part I

Introduction and Overview

CHAPTER
ONE

INTRODUCTION

SPECIFYING REACTIVE SYSTEMS is surprisingly difficult. Even if the technologies for implementation and execution are mature and well-known, it's hard to specify precisely what the system should do in the first place. This is due to the nature of the problems these systems should deal with; in reality, many things are going on at the same time, and we want our systems to support this appropriately. If you have ever tried to organize a visit to the cinema with a bunch of friends via telephone, where all of them had to agree on a certain movie and time, you know that coordinating many participants in reality can be hard. As we will later see, reactive systems decompose into physically distributed components, running on different devices. On a very detailed level, it's difficult to get the interactions between the system components right, so that no communication errors arise. But the devil is not only in the details: Even if we treat all interactions with accuracy, we also have to ensure that the system as a whole works properly and the services offered to the environment do what we expect.

What surprises is that most of the problems, once identified and considered in isolation, are well-known and actually not so hard to solve. It is rather the sheer complexity of functionality that is difficult, when everything comes together. Most likely, the complexity of systems will even increase as the number of devices in our environment grows. Much of their value will depend on how we manage to connect them. Nevertheless, specifying such systems must get much cheaper and more flexible than it is today. It is therefore important that we find a way to specify quickly and flexibly how these devices may be coordinated.

Obviously, a specification technique should allow us to factor out problems and reuse solutions. This seems to be quite difficult, however. One inherent reason for that is that a reactive system can be decomposed in two orthogonal dimensions [Mik99]: We can divide a system into its physical components that are spatially distributed (sometimes called *vertical* decomposition), or we can separate its logical functions from each other and describe the distinct services a system offers to its environment (focusing on a *horizontal* axis). Traditionally there was a strong focus on the vertical axis, i.e., the component behavior, expressed for instance by SDL processes, since such a description is ultimately needed to implement a system. However, as it has been argued in the literature (see, for example [DA85, VL85, Mik99, RGG01, KM03]), the horizontal axis has its benefits when it comes to the documentation of a system's functionality, which often involves several components: Typically, the most difficult problems involve several components and the necessary interaction between them. It is therefore desirable to understand, specify and analyze collaborative behavior among several components in a more explicit way than it is possible in the component-oriented view. This had influence on numerous approaches and development techniques, for example DisCo [JKSSS90], OORAM [RWL95], Use Case Maps [BC96], Catalysis [DW99], Micro Protocols [GKS02], CoSDL [RGG01], STAIRS [HHRS05] and [DKMR05], which allow us to specify collaborative behavior among several components using different techniques.

Despite these efforts, we think that there is much to gain by further exploiting the collaborative perspective. We envision an approach in which *collaborations*, that is, the local behavior of participating components as well as the necessary interactions that are related to a certain distributed function or task, are the major specification units. In particular, we want to model collaborations in the form of encapsulated building blocks which can easily be composed with each other. However, while such an idea may seem quite straightforward, it turns out that it is hard to get right in practice. There are several difficulties that need to be solved for such an approach to work:

1. The nature of the building blocks and how they can be combined with each other is a challenge in the first place. On the one hand, they need to be self-contained and encapsulated in some way, so that we do not need to understand them completely just to apply them. On the other hand, it should be possible to combine them flexibly with other specification units so that new systems may be constructed from them.

2. When we want to use existing building blocks as solutions within composed service specifications, it is important that these specification units preserve their properties during composition and implementation, that means, that their properties are also present in the resulting system.

3. Even if models of the collaborative view are beneficial for the specification of services and the reuse of building blocks, the traditional, component-oriented perspective is still needed for the implementation. That means,

although we like to think of services in terms of collaborations, we eventually need components. In many approaches used in practice, models belonging to these different perspectives need to be aligned manually, which is costly and a source of inconsistencies.

To solve these problems, we employ reasoning in temporal logic. In particular, we use three principles from the Temporal Logic of Actions (TLA, [Lam02]), the compositional Temporal Logic of Actions (cTLA, [HK00]), as well as DisCo [JKSSS90]:

1. To address the first problem, we formalize collaborative building blocks as cTLA processes consisting of a set of actions and variables. As such, they are operational and can be analyzed for arbitrary properties. Composition with other building blocks is done using the principle of joint actions found in DisCo and cTLA.

2. The composition of building blocks by joint actions enables the formal principle of superposition [KS05], according to which the composition of processes preserves the safety properties of all sub-processes. This ensures that once established properties of the behavior of building blocks are in fact obeyed by the resulting system.

3. To ensure consistency between collaborations expressing services and the components of the implementation, we follow an automated approach, in which components and state machines are synthesized from the collaboration using a model transformation. Formally, these two views are connected by a refinement relation, meaning that each step executed by a component's state machine maps to a step of the collaboration. This principle ensures that properties exhibited by the service specification are present in the final implementation as well.

However, even solving these problems is not enough to enable a user-friendly approach in practice; after all, tackling the complexity of systems is an engineering challenge in which theory has to be accompanied by pragmatics, experience and tools. For that reason, this thesis explicitly addresses theoretical and practical aspects. Suitable notations for the building blocks and their effective composition need to be developed, preferably based on a standard to increase their acceptance and the availability of tool support. For this purpose, we use a combination of UML 2.0 collaborations to cover the structural aspects of collaborations, UML 2.0 activities to describe behavior and precisely compose building blocks, and a special form of UML 2.0 state machines to capture their external behavior in so-called external state machines or ESMs. We map the syntax of these notations to semantics defined by cTLA processes, so that the composition method of joint actions is mapped to the behavioral composition of activities by control flows. In this way, the underlying formalism can be completely hidden from the engineers. We provide tool support to demonstrate that the concepts proposed are effectively usable.

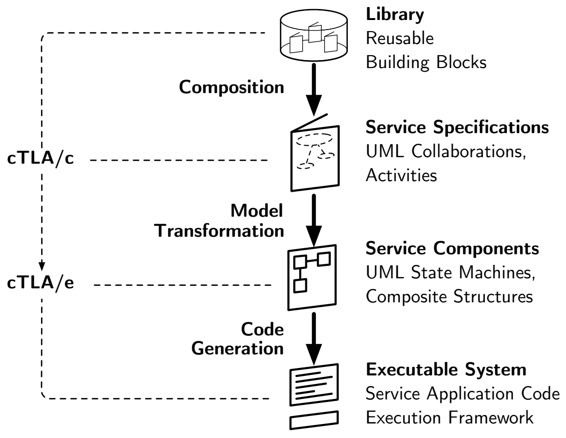


Fig. 1.1: The SPACE Engineering Method (from Paper 6)

As a result, we describe a method for the specification of reactive systems, SPACE, outlined in Fig. 1.1. For the design of a new system or service, collaborative building blocks are taken from libraries containing generally useful functionality or domain-specific building blocks. Building blocks may realize simple functions or entire sub-services. Missing functionality can be added as new building blocks. Building blocks may be composed to realize more comprehensive ones. Due to the formal setting, building blocks may be analyzed using model checking. Due to the compositional semantics, this happens incrementally, i.e., block by block, which reduces the state space that has to be analyzed. Once a service specification is complete and consistent, it is transformed into components and state machines from which code can be generated. In practice, the approach is supported by the tool set *Arctis*, which has been implemented as part of this thesis and allows us to use all the features described in the following. From a theoretical point of view, SPACE specializes cTLA with the two styles cTLA/c for collaborations and cTLA/e for executable state machines. These styles can be expressed graphically using UML, as illustrated by the dashed lines in Fig. 1.1, which connect the modeling used by engineers with their semantics and logical reasoning in cTLA.

We believe that the languages and the tools offered make the method appealing to practitioners as well as theoreticians. All models used have formal semantics and can be analyzed to ensure correctness, and the implemented tools relieve engineers from repetitive tasks so they can spend more time developing solutions for the specific problems of their domain.

1.1 Contributions

Through the eight publications presented in Part II, an engineering approach for reactive systems is developed and presented. It comprises three key features that enable a rapid engineering process without compromising quality:

1. **Services as Compositions of Collaborative Building Blocks**

   Systems and services can be composed from existing, reusable solutions encapsulated as building blocks. Building blocks may be collaborative, meaning that they describe the local behavior of several components as well as the necessary interactions between them. This increases the potential for reuse, since solutions that require the cooperation of several components may be reused within one self-contained, encapsulated building block.

   This means that collaborations encapsulate the buffered transmissions between components, including necessary synchronization mechanism to ensure sound communication. This also means that collaborations are coupled *within* components, i.e., under local control where no intercepting medium is introducing communication delays, which makes their composition much easier to handle. With the syntax of activities, this coupling is intuitively based on the wiring of building blocks.

2. **Automated Transition from Collaborations to Components**

   The thesis describes an automated transition from collaborations to components, so that engineers only need to build and maintain one model. In particular, this approach avoids a difficult, and typically time-consuming manual synthesis of state machines.

3. **Incremental Model Checking and Automated Analysis**

   The definition of collaborative building blocks and their semantics in temporal logic enable an incremental strategy for model checking, since each building block may be checked separately, which reduces the state space during analysis. Moreover, the knowledge about building blocks enables a highly automated analysis process. Model checking can be performed without intervention of the user, and results can be shown to the user in terms of the original context of activities. This means that users do not have to study formal techniques in order to use their results.

The innovations and novelties implied by these key features are discussed in detail in the respective papers and summarized and discussed in Chapt. 6. The papers, the appendices and the implemented tool support together provide a rather complete set of artifacts needed for the development of reactive systems and services. In particular, these are:

– Two specification styles in cTLA that formalize specifications in the collaboration-oriented view (cTLA/c, Paper 6) and the derived specifications of the executable component-oriented view (cTLA/e, Paper 2).

– Two corresponding UML profiles. One covers the UML models of the collaborative service specifications in terms of UML 2.0 collaborations, activities and ESMs (Papers 1, 4, 6, as well as a summary in App. C). The other one covers the models for the executable state machines (Paper 2 and App. E).

- A mapping from activities to cTLA/c (Paper 6) and an automated implementation of this transformation to TLA$^+$ (Paper 7).
- A mapping from the collaborative service specifications in form of UML 2.0 activities (formalized by cTLA/c) to the executable state machines (formalized by cTLA/e) in Paper 3. The mapping is implemented as an algorithm implemented in Java.
- A set of reusable building blocks and some design rules (App. B).
- Tool support with Arctis and Ramses that support the proposed specification style, the analysis of models as well as the implementation via transformation and code generation. The tools are introduced in Sect. 4.5 and Paper 8, and App. A provides a more comprehensive example of their application.

During our research we worked on several case studies and examples, presented in the papers of Part II and App A:

- Paper 1 and Paper 3 present a classic example of an access control system that was also used in [BH93] and [BS01].
- Paper 2 presents a small system to track the location of devices.
- Paper 4 uses the example of a taxi control system to demonstrate how multiple instances of collaboration sessions may be coordinated.
- Paper 5 presents a larger and more complex example for the management of trusted auctions.
- Paper 6 discusses a home automation system utilizing most of the features of the approach to exemplify the formalization of collaborations and their composition.
- Paper 7 uses an intricate example of a hotel wake-up alert to demonstrate how model checking can be used to support engineers. Especially, a building block to handle mixed initiatives is introduced.
- Appendix A presents the example of a mobile treasure hunt, which we use to demonstrate the composition, analysis and implementation using the tool support provided by Arctis and Ramses.

1.2 Focus of this Thesis

We will focus on the complexity of reactive systems that arises from the coordination of tasks and necessary interactions of components. Algorithmic complexity, that means the complexity of transformational operations on data, is not considered. For the UML activities, this means in particular that we concentrate on the behavior expressed by control flows and largely ignore data associated by object flows. An expansion to handle data flows is part of future work, outlined in Chapt. 7. Moreover, we consider user interfaces as a part of the environment that communicate with the system by means of explicit signal transmissions.

Although temporal logic is especially powerful to reason about liveness properties in specifications, we focus on safety properties, following Lamport's advice that *"most of your effort should be devoted to examining the safety part"* [Lam02].

1.3 Guide to this Thesis

The work presented in the following consists of three parts.

Part I: Introduction and Overview

Part I continues in Chapt. 2 with an analysis of reactive systems and services where we discuss some of the reasons that make these systems difficult to specify. In particular, we present a number of questions that have guided our research. In Chapt. 3 we present a number of approaches and paradigms that form the background of our work and that inspired our method. We discuss a number of existing engineering approaches that are used to develop reactive systems. We also introduce the temporal logics TLA, DisCo and cTLA. Since collaborations are the primary specification units of our approach, we provide an overview how this concept has been used by others for the development of systems. Thereafter, in Chapt. 4, we present our method SPACE. It is a combination of different specification styles, formalizations, graphical languages and mappings to develop systems rapidly based on reusable building blocks. Chapter 5 summarizes the publications included in this thesis. We discuss our results in Chapt. 6 by presenting our answers to the research questions of Chapt. 2 and present related approaches. We close in Chapt. 7 with suggestions for future work.

Part II: Included Publications

Part II contains eight peer-reviewed and published papers. They are presented in an order that builds up the core of the approach with the first three papers, then expands successively and demonstrates its use by means of several examples.

- Paper 1 [KH06] gives an overview of the specification style used in our approach. It is based on a combination of UML 2.0 activities and collaborations and special state machines (ESMs) to document the externally visible behavior of collaborations.
- Paper 2 [KHB06] formalizes the executable state machine models that are the target of our development process and discusses their execution.
- Paper 3 [KH07b] describes how the service specifications can be transformed automatically into the state machine models.

– Paper 4 [KBH07] expands the approach and the transformation so that also systems with multiple instances of components can be specified and transformed.

– Paper 5 [HK07] demonstrates the feasibility of the approach with an example from the domain of trust management systems, an electronic auction system.

– Paper 6 [KH07a] formalizes the collaborative specifications in temporal logic by describing a mapping of UML activities to temporal logic and describing their composition.

– Paper 7 [KSH07] shows how the concepts of the approach can be used in model checking, so that engineers can be supported during the design of services.

– Paper 8 [Kra07] provides a brief overview of the tool support for the approach through our tools Arctis and Ramses.

A comprehensive overview of the contributions of each paper is given in Chapt. 5.

Part III: Appendices

– Appendix A presents a submitted journal article that provides an overview of the entire approach using an example. It is written from an engineering perspective and presents the different steps of development, illustrated with screenshots of the tools.

– Appendix B executes a refinement proof for a system example showing that a set of state machines synthesized by the model transformation implements the behavior described by the activities.

– Appendix C summarizes the modeling constraints and semantics for the collaborative specifications based on activities by describing a UML profile.

– Appendix D provides an overview of building blocks and patterns we developed during our case studies. They cover different domains and concerns and can be useful in many applications. We also explain some design rules that should be taken as guidelines.

– Appendix E summarizes the conventions for the executable state machines models on which our runtime model and the code generation is based. Like App. C, the constraints are defined as a UML profile.

Ei pil og noen boksa
og sjå så mange paradoksa.

CHAPTER
TWO

REACTIVE SYSTEMS: WHY THEY ARE DIFFICULT

In this chapter we analyze some of the challenges that make reactive systems difficult to specify and understand. After a consideration of reactive systems as such and how they can be decomposed into physically distributed components that can be described by extended finite state machines, we will argue that the actual implementation of components once given in the form of state machines is manageable. The difficulties are to create state machines for components in the first place, and here we identify a number of challenges that will motivate our search for solutions in the remainder of this work. In the end, we will state the aims of this thesis in more detail by posing the research questions directing our work.

2.1 Reactive Systems and Reactive Components

The systems in focus of our interest are *reactive*, which means, as Pnueli explains, that they *"maintain some interaction with their environment"* [Pnu86]. The environment consists of users and other components that are not under the direct control of the system. As an example, we consider a system supporting the transactions of an electronic auction, depicted on the left hand side of Fig. 2.1. The system serves entities in the real world, namely a buyer, a seller and an auction house.[1] Since these real-world entities reside at different places, the system is necessarily distributed. Depending on some design decisions, it may have the structure as shown on the right hand side of Fig. 2.1, where each of the real-world entities corresponds to a dedicated system component. Pnueli argues further that a component of such a system should always be viewed as a reactive component, as it *"maintains a reactive interaction with the other components in the system"* [Pnu86].

[1]The example is discussed in detail in Paper 5. We currently disregard the fact that there are several potential buyers from which only one wins a bid.

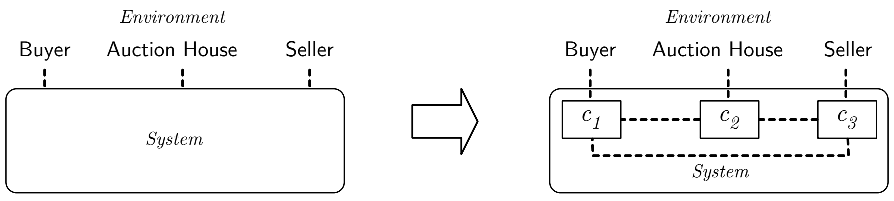


Fig. 2.1: An auction system

In the following and throughout all included publications, we mean by *components* separate units that each have their own threads of control and can be separately deployed and executed. Other terms used in the literature are, among others, *agents* [ITU02], *actors* [SGW94, BHM02] or *active objects* [Obj07b]. Objects in the sense of encapsulated data as used in object-oriented programming are in the following called *passive objects*, *objects* or simply *data*.

In order to decouple components from each other and to prevent that a component has to block or wait actively until another component answers a request, an asynchronous communication scheme is used, in which signals are transferred via a medium that can be seen as a buffer. This is a quite flexible and general communication paradigm, allowing for symmetric peer-to-peer communication, as often necessary by the systems we consider. For a discussion of this communication paradigm, see for example [BF04].

2.2 Now, What's So Difficult?

In the following, we will discuss the difficulties that can arise when specifying a reactive system. We found it convenient to present our arguments in the form of a dialog, inspired by [Lam89]. Although we want to focus on the problems, one of the participants obviously already has some vague ideas about solutions that influence the discussion.

When we build a reactive system, we eventually have to construct its reactive components. But how do I describe these components?

The reactive behavior of the components can be expressed using a notation that connects a stimulus (for example the reception of a signal) to a response, that means, local actions and sending of signals to other components. Such a notation is provided for example by extended finite state machines. A component can be described by a single state machine, but often we find several state machines within a component. State machines have been used for several decades, and are standardized for example by the processes in SDL [ITU02]. We will later discuss this in Paper 2, where we formalize this notation and align it to the current standard of UML, which offers similar notations.

Can't I write code right away? Why do I need the state machines?

You could program right away, if your system is really small or only contains very simple interactions. For systems of realistic size, however, you will likely fail. The problem is the complexity introduced by the concurrency of the components. Even though programming languages may support concurrency by synchronization mechanisms and threading, these concepts do not always scale to the needs we have (see, for example [Bræ79]). Using state machines, however, enables an effective scheduling scheme by an additional layer of process multiplexing introduced by a special runtime support system, as discussed for example in [BH93, San00] and Paper 2. This prevents numerous errors related to the synchronization of tasks and threading. We can look back on several decades of this practice, going back to early telephone switching systems and methods as for example SOM [Bræ79].

That's about 30 years ago. I guess someone has come up with modern platforms that work completely differently. Is this still relevant?

You will be surprised how little really has changed. There may be new languages, new standards and new technologies, but the underlying principles are still the same. And they are very well applicable to a range of more recent platforms, as well. For the remainder of our work, we will therefore see the state machines as the target of our development efforts. Once we have the state machines, it is a rather technical task to implement, deploy and execute them. We have provided code generators [Stø04, Kra03] that produce executable components from them, as described later in Paper 2.

Where's the problem then?

Getting the components with their state machines right in the first place is a major challenge. Most of the functionality provided by the systems we consider can only be provided by cooperation of several components. This implies that their behavior needs to be synchronized and coordinated, which, given our asynchronous communication mechanisms, means that signals have to be exchanged correctly. We can also speak of the interactions between a pair of components as *conversations*. With an increasing number of interconnected and communicating components, a component also needs to integrate more of these conversations, to serve its communication partners properly.

I remember creating a state machine on my own. It was a small example, just one state machine talking to three other components. I had to look all the time into the state machines of the other components to ensure they were compatible and pay attention not to forget anything. Luckily these state machines were easy to debug — I got notified once a signal arrived in an unexpected state. So I added transitions here and there, and in the end, it worked. But that was a tough job.

Exactly. When doing this manually, for example by designing an SDL process communicating with several other components, we have to be aware of which signal may arrive from the other components, and which signals they are waiting for. This means that we have to be aware of the state of conversation with each of the communication partners in all of the control states of the state machine. Seen from a practical point of view, that means we have to consider the state machine diagrams of all the other communication partners while designing the new one. While this may be acceptable for the communication with one partner, holding control over several ones may obviously cause some headache.

Oh, yes...

But that is not the only difficulty when creating state machines. We also must take into account that signal transfers are necessarily delayed, due to the communication medium. This may lead to some intricate overall behavior with race conditions and conflicts that makes it necessary to send additional signals to resolve these situations. So besides looking at all the other state machines of the communication partners, we also have to consider the possible state of the communication channels.

I guess that's why my system still crashes sometimes — my test cases probably did not cover all of these cases. Maybe I should have applied model checking to my system?

Model checking is quite powerful. It means to examine all reachable states of a system and check if some of them harm general assumptions like freedom of deadlocks, or more specific invariants like for example the exclusive access to some resources. However, it is again the complexity of the systems that lets this kind of examination quickly come to its limits as well. The number of states that have to be examined often grows exponentially with the size of a specification. This has as a result that the model checking either takes a long time, or exceeds the memory limit of our machines. This problem is well-known and also called state space explosion.

Ka-Boooom!

Exactly. To handle this problem, either the state space must be kept reasonably small by partitioning the system into different parts that can be checked individually, or a simplified version of the system must be checked.

Couldn't we just describe the interfaces of those components we are communicating with, so that automatic checks get manageable?

A very good idea, indeed! Such *behaviored* interfaces, as we may call them, are described for example in [CFN03, BK98, Men04, dAH01]. In [Flo03] such an idea is realized for SDL processes, where the observable behavior of a component towards another one is abstracted by a projection. Using this projection, an algorithm can check if your state machine under construction actually communicates correctly with the other components. And that works even without examining the state space of the entire components, so that the algorithm is very quick.

So then the algorithm is taking care of my troubles?

With the algorithm you may find errors related to a conversation between pairs of state machines, but you still have to write the state machines manually. We can call that a corrective approach, that tells you when you did something wrong.

But I guess the projections also help me to construct the state machines?

Yes, they do. Instead of looking at an entire state machine, you just have to look at the projection, which of course is easier. To a certain degree, this can also be used by a tool. In fact, we have experimented with tool support for the synthesis of state machines based on given behavioral interfaces in [Gis06]. While such a tool can reduce the effort to edit state machines and can prevent the introduction of errors in the first place, we still have to deal with difficult synchronization situations manually.

That still sounds like many problems are solved?

There are some more challenges. In our eagerness we focused on how to initially write down correct state machines. But writing them down, even if correctly, does not mean we are finished. As you know, specifications (just like programs) are only written once, but usually read many times more often. So we should not only consider how costly it is to create a state machine, but also how costly it is to understand it again after its creation, and how costly it is to change it. If all the conversations are tightly interwoven with each other, this is difficult. Instead, we should try to to retain the original conversation towards one interface, maybe in some kind of building block. To synchronize the different conversations, we connect the building blocks somehow. In this way we could see at a glance how the different conversations are synchronized. Changing some conversation would probably only affect the internals of the building block. Adding a new conversation would change maybe at some points how the conversations are connected, but not their internals.

Okay. I'm starting to understand. Given the situation that we design the entire system, you would like to have an approach where we can select building blocks that solve our problems and compose them together. The composition has two benefits. First, we can quickly get a solution as we do not have to write specifications completely from scratch. Second, once we have a finished specification, *the fact that it is composed of building blocks helps us to understand it, as we may already know some of the building blocks. Right?*

Right! You already went into reuse of specifications, a very important issue. But let us stay for a moment at the correctness of specifications. Beyond the correctness of the separate conversations between the components we just discussed, also the overall behavior of the system has to be performed correctly by the set of components. In the auction system of Sect. 2.1, for example, it is important that only one of the buyers gets the bid accepted, and that the others loose the auction — a typical case of mutual exclusion. Such a property goes beyond pure protocol correctness and we need to consider the overall behavior, not only interfaces or single conversations.

Let me guess — model checking is still in the race? What about the ka-boooom?

We have to find a clever way to reduce the state space during model checking. Maybe it is possible to use model checking incrementally, that means each building block of a system separately. And then we could use composition principles that tell us what we get when combining these building blocks. We even could abstract building blocks by simpler models just showing their external events, so that also compositions of building blocks may be model-checked.

That would mean that we decompose the system so that the state space gets manageable. As an engineer, I like the idea with the building blocks. But won't it be very hard to express the conversations with them, I mean, separately, as you emphasized, and then precisely compose these conversations together with the same power as if I compose my state machines?

I have some ideas, especially if I look at the way temporal logic allows us to compose specifications. There you can have both; processes that are complete, self-contained and operational. Yet you can compose each detail of them together with other processes by joining their actions in a kind of instant rendezvous, also called joint action.

Sounds interesting. But I'm still thinking about the state machines. One drawback of them is that one state machine only describes one side of some functionality. As mentioned before, often I have to examine several state machines just to find out what the system does. Look for example at Fig. 2.2. I have drawn the components of the system and considered which responsibilities they have in the system, similar to CRC cards.[2] *A buyer, for example, is involved in bidding for a product, retrieving trust from the reputation server, reporting back experiences about the seller, and in the end participating in a trusted sale. The other components have similar tasks. If I want to understand the trusted sale, for example, I need to look at the specification of the auction house, the buyer and the seller. That is cumbersome. Am I the only one noticing that?*

[2]CRC cards originate from the technique *Class, Responsibilities, and Collaborators* by Beck and Cunningham, described in Sect. 3.4.8.

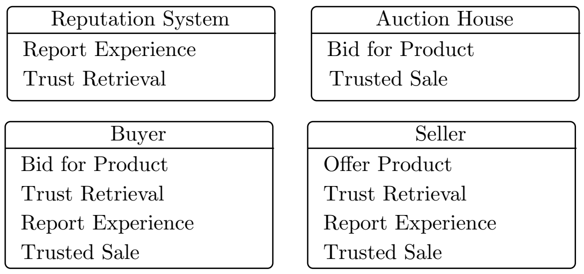


Fig. 2.2: Components of the auction system

What you are referring to is a problem also discussed in literature. The role models of OOram [RWL95], for example, emphasize the collaboration of objects. Mikkonen and Kurki-Suonio describe this as the two perspectives on an architecture [Mik99, KKSM04]. Many others have acknowledged this *cross-cutting* nature of services as well (see, for example [KM03, RGG01, FK01]). The service specifications described in [vBG86], for example, cover behavior executed at several places.

You used the new term "service." What exactly do you mean by that?

The term *service* is used with different meanings. A characterization of them can be found in [QSPvS07]. For us, a service is simply some identified functionality that a system offers to its environment. This often requires that several components of the system work together, hence the cross-cutting, horizontal nature of services. A system can offer several services. How exactly you divide the systems functionality into services, is up to you. In telecommunications, we would often like to develop, deploy and maintain services separately, so that a service can also have aspects that go beyond pure functionality, which, however, we will not discuss in the following.

When we talk about a *service specification* we mean a complete functional description of a behavior necessary to execute a service, that means the necessary interactions as well as the local behavior of the participants. With the term *collaborative service specifications* we emphasize that services span over several components and that they may be composed from other services, also referred to as *sub-services.*

Indirectly you already have identified the services in Fig. 2.2. If we rearrange them and introduce two decomposition axes, the point with the services gets clearer, as in Fig 2.3. It is inspired by a similar figure presented in the SISU report [BMP95] and resembles the *hat stand synthesis model* of roles presented in OOram [RWL95]. The horizontal axis shows the components of the system, while the vertical axis shows the sub-functionalities between the sub-sets of participants. Obviously, the notions of reactive components and reactive services are orthogonal to each other. While components are a structuring mechanism for the physical composition of a system, services are a structuring mechanism

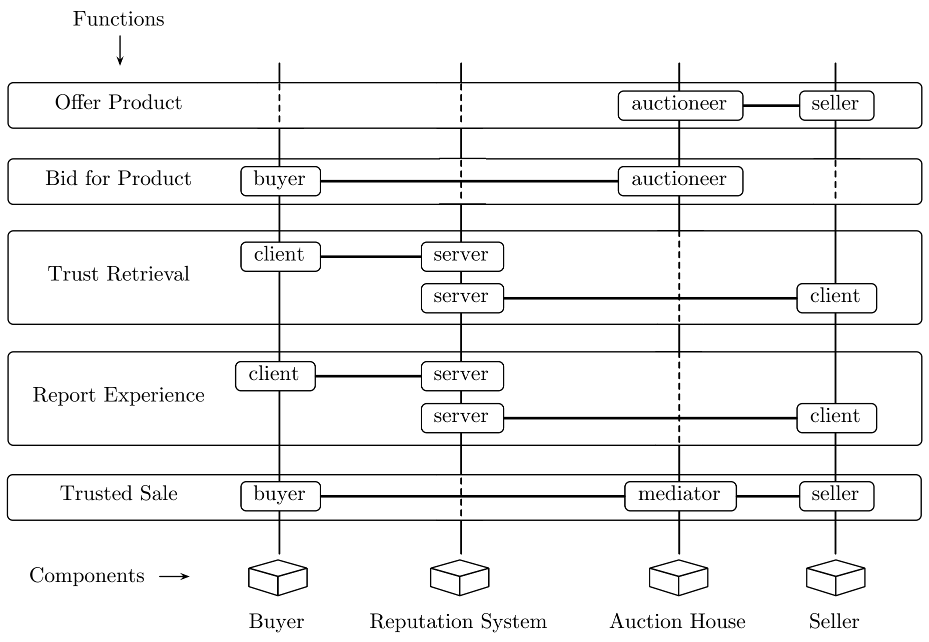


Fig. 2.3: The auction system decomposed into its components and functions

for the system's functionality. On the horizontal axis, the conversations can be seen between the components that are necessary to coordinate their execution.

In the auction system, this means that a complete auction can be decomposed into sub-functionalities, such as offering a product, bidding for a product, interacting with the reputation server and eventually dispatching the trusted sale. Note that functionality can be decomposed over several levels and not just one as shown here. In Paper 5 we will see, for example, that the *trusted sale* sub-functionality can in turn be decomposed into more elementary functionalities. In Sect. 3.4.10 we will come back to this figure and present it in a more convenient, more recent and standardized form.

I see. This figure instantly provides an overview of the functions. I like it. And I guess these horizontal tasks can be described for example by message sequence charts?

That's one possibility, yes. And indeed, they have been used for a long time as well, to complement the aspects not covered by state machines. But remember there are also other types of diagrams to choose from when showing cross-cutting behavior, as for example Petri nets [Mer79], activities [Obj07b], use case maps [BC96], or non-graphical descriptions like joint actions, for example from the DisCo language [JKSSS90].

So the state machines aren't that great, after all?

Be careful, we are speaking about the *horizontal* axis now, covering *collaborative* behavior. For such collaborative behavior, state machines may not be the ideal notation for a specification, as we discussed. But they are still the best way to specify the reactive behavior of components. Remember the platforms and experience we have for their execution, that's nothing you simply want to throw away. And there really is no need for it, either.

Then you advocate two description forms?

Yes. One description form to capture the collaborative behavior of the system which tells us what each service does and how they are composed from sub-services. Another one to describe how the components execute their behavior, and here, as you may imagine, we use state machines. Most of the approaches characterized in [AE03] use two description forms, often with state machines as the one close to an implementation.

That sounds idealistic to me. It already takes lots of persuasion to get programmers to model state machines. And now they should use some other description in addition? Even if you actually get them to write these two models initially, they will never maintain both of them when they change the system's implementation. The result of that idea is that you have some sketches of the services in a collaborative model, which, however, are completely out of sync with the actual components.

We already mentioned code generation from the state machines. If code generation is completely automated, having the code synchronized with the state machine models can be done by re-generating it once the state machines changed. A similar strategy should also work from collaborative service specifications towards the state machines. The state machines could be derived from the collaborative models automatically, which then are the only specification of the system that is edited manually. The surveys in [PS91, AE03, LDD06] give an overview of some existing approaches. Often, state machines are derived from scenario descriptions given for example by MSCs, but also other languages are used. For such an approach to be effective in our setting, we have to develop a completely automated transformation from the service specifications that also takes into account the idea of composing reusable and encapsulated building blocks and the notation that we are using for them.

But now we are already starting to talk about possible solutions. Let's stick to the problems. That was just to make clear that we can keep state machines and also have collaborative specifications.

I guess one challenge is how to specify the services in such a form that the behavior of the components can be completely derived from them?

Yes. And this does not only mean that we have to describe the precise behavior of the trusted sale, but also how this functionality (or sub-service) is connected to the other functionalities, for example the trust retrieval. Only if we manage to address that as well, there is a chance that we can derive the components completely automatically.

That also puts the idea of the reusable building blocks in a new light. I guess we are not reusing elements of the component-oriented perspective, the state machines, but elements from the collaboration-oriented one? I just wonder, weren't components with state machines designed to be reusable?

Our experience with the reuse of components in our kind of systems is rather discouraging. Especially for application-specific functionality, reuse of entire components is quite limited. Granted, there are some components, usually those providing very basic functionality, that can be directly reused. However, we have seen how nicely the example in Fig. 2.3 could be decomposed into small, collaborative functions like the trust retrieval. It seems more likely to find another application that may reuse the trusted sale than the entire component of the buyer, for example. So there is a potential of reuse far beyond what components give us as soon as we look at the services and sub-services of the horizontal perspective. Speaking of that — do you remember the problem of the mixed initiative[3] you once had?

Yes! Identifying the mixed initiative in the first place was hard, but then I used the solution proposed in [Flo03] and it worked. I had to add several transitions to both state machines. That was a bit tricky, since it was not possible to just copy the solution and paste it into my state machine. I had to understand the solution first and apply it manually, as the state machine was also handling another conversation, so that I had to interweave the states of the solution with the ones I already had. But after a while I got it right.

Good job. And the right way to do it if you have state machines. But now think of the idea with the building blocks. Wouldn't it be great if you just could reuse a building block solving exactly that problem? As the problem involves several parties that have to be coordinated, the building block should be collaborative, in other words address the behavior of more than one component. This means we would define this building block on the level of collaborations and services.

That sounds quite good. Maybe these building blocks could also solve another issue with the state machines: I know my state machine is correct now, but please don't ask me to change anything. Then I would have to understand each transition and which problem it solves again.

That happens if applying a building block means you have to look into its details. In the state machine, for example, you have to apply the solution to the

[3]Mixed initiatives are discussed in Paper 7.

transitions that already handle other functionalities or conversations, and that is complicated. Instead, we should refer to a building block that stays structurally intact, so that we can recognize it when we read our specification again. Maybe that is possible with some diagrams of the collaboration-oriented perspective.

If I remember right, model-checking was part of a solution. However, I think it is sometimes quite difficult to apply.

It's not that difficult. But I agree, some engineers may want to focus on their problem domain and not spend the effort and learn model checking or formal methods in general. Maybe we can help here with some tool support. In some cases we could automatically find design flaws.

Then the users would not even notice that they actually use formal techniques to check their design. Smart. I think I have read about this idea in [Rus00]. However, I have bad experience with modeling tools so far. Some are more like drawing tools for UML or SDL. They often tell me nothing really interesting about my system.

The tools you talk about have the sheer impossible task when they want to implement for example the UML standard. That is only a language, not a method in itself. But to create good tools, you certainly have to know in which way the users want to work, which means you have to know about the method. Capturing all features of UML may be too much for a single tool, which then has to be quite generic — a drawing tool, as you say. You, for example, use state machines as executable specifications for reactive components. Another customer could use them for an abstract business process. A tool really helping you in validation and verification has to know more about your domain of application. Therefore, we should always be aware of the execution semantics underlying our models. And for that we have a solid basis with the executable state machines, no matter what ideas we develop for the collaborative perspective. ■

2.3 Research Questions

The introduction as well as the fictional conversation in the last section raised some questions that have established the objectives of this thesis. As an overall question, we ask:

> 1. *What should be the basis for an expressive approach that lets us rapidly compose services from reusable elements?*

We argued in the previous section, that the potential for reuse is higher if the reusable elements are part of the collaboration-oriented perspective, so that functionality involving several components may be reused directly. However, for the implementation, we still need the description of separate components that describes how they coordinate the different services in which they participate. This poses another question:

> *2.1 How can we bridge the gap between collaboration-oriented, horizontal models focusing on functional decomposition of a system into services, and vertical models decomposed into components?*

If the components with their state machines could be synthesized completely automatically from a collaboration-oriented specification, consistency between both perspectives could be ensured by construction. We narrow and rephrase therefore the previous question by asking:

> *2.2 Can state machines and components be synthesized completely from collaboration-oriented specifications?*

A prerequisite for a positive response is that collaboration-oriented models are more than just incomplete scenarios that guide a manual design step for components, but specify the complete system behavior. This implies two detailed follow-up questions:

> *3.1 How can collaborative elements be expressed separately and in a self-contained, complete way?*

> *3.2 How can collaborative elements be composed, so that detailed dependencies between sub-functionalities can be expressed precisely?*

We pointed out in the previous section that reuse only unfolds its full potential if reused elements stay structurally as well as behaviorally intact after their composition, so that the acquired knowledge about a building block can be reused as well. As we want developers to work on graphical models as provided by UML, we ask:

> *4. Which (graphical) notation supports a flexible, precise composition that preserves the integrity of building blocks?*

We anticipate that the application of formal methods and model checking will play a major role in ensuring value and consistency of specifications. We already acknowledged that model checking faces the challenge of complexity, and therefore ask:

> *5. Can the individual reusable elements be analyzed separately?*

Analyzing reusable elements separately is only valuable if we can ensure, that the composed ensemble maintains the properties of the individual building blocks. Furthermore, related to questions 3.1 and 3.2, we must ensure that also the executable system that is derived from the composed building blocks is correct, and we ask:

> *6. How can the correctness of a composed and eventually implemented system be ensured?*

Many developers avoid using formal methods as they fear the initial costs of learning these techniques or do not see an immediate benefit for their problem domain. We therefore pose our last question:

> *7. How can the threshold to applying formal analysis be kept low for a practitioner?*

We will elaborate our answers to these questions in the papers of Part II, and recapitulate them in our discussion in Chapt. 6.

CHAPTER
THREE

BACKGROUND

The research questions posed in the previous chapter are not of a pure scientific character. To answer them, we also need to consider the overall engineering process for system development. There exists a considerable number of methods that guide the development of software and systems. Not all of them are targeted specifically at the design of reactive systems; their focus varies from object-oriented design over the design of embedded real-time systems to the specification of telecommunication systems. We limit ourselves to methods targeting reactive systems using the standards of SDL and UML in Sect. 3.1, since the components that we eventually generate are based on models originating from these approaches. In particular, we will discuss SOM in Sect. 3.1.1 as it establishes the execution mechanisms based on state machines that we will use. In Sect. 3.2, we consider model-driven development and in particular the Model-Driven Architecture and how it relates to the discussed engineering approaches, since we want to make a model transformation a part of our approach, as we already outlined in the previous chapter. Because our reasoning is based on temporal logic, we introduce in Sect. 3.3 briefly the Temporal Logic of Actions, TLA, as well as the two related logics and approaches, DisCo and cTLA. In the previous chapter we motivated our interest for the notion of collaborations and anticipated that this concept will be used as major specification unit in our approach. We will therefore consider in Sect. 3.4 how the notion of collaborations has been used in other approaches, and how this concept evolved over time.

3.1 Approaches Based on SDL and UML

For the development of reactive communication systems based on the Specification and Description Language (SDL, [ITU02]), a number of methods have been developed, among them SOM [BHS81], TIMe [BGH^{+}97], SPECS [OFMP^{+}94], and SOMT [Tel02]. While these approaches vary in their details and emphasis of certain aspects, partly due to the evolution of the SDL standard, they have many similarities. In fact, an appendix [ITU03] of the SDL-92 standard contains guidelines for using SDL that are based on most of these existing approaches. The current standard contains a revised version of these guidelines called SDL+ [ITU97b] (see also [Ree96]). Two methods that contributed in particular to the design and execution models for our approach presented were SOM which is presented in Sect. 3.1.1 and TIMe which is presented in Sect. 3.1.2.

The Unified Modeling Language (UML, [RJB05]) has its origins in the methods Booch [Boo91], OMT [LSR87, RBL^{+}91], as well as OOSE [JCJÖ92] and focuses on object-oriented software engineering. A number of other approaches, notations and methods are built upon UML and have contributed to its evolution, for example by extending and elaborating its notations or building semantic foundations. Examples are Catalysis [DW99], Executable UML [MB02] based on the Shlaer-Mellor method [SM92], and ROOM [SGW94]. A survey of other object-oriented methods can be found in [Wie98].

Although UML is merely a meta-model and collection of languages and not a method, it shapes to a large degree how systems can be specified. Methods specifically addressing UML are for example the Unified Process [JBR99] or its commercial version from Rational, RUP [KK03]. UML also has a central position within the model driven architecture, as we will present in Sect. 3.1.3. We will have a closer look at ROOM in Sect. 3.1.3 and present Catalysis with a focus on collaborations in Sect. 3.4.7. With its version 2.0 [Obj05], UML was considerably revised and contained elements from Catalysis, ROOM and especially SDL and MSC. We will present the aspects of UML 2.0 relevant for this work in Sect. 3.4.10 and the papers of Part II.

3.1.1 SOM, the SDL-Oriented Method

The work with SOM[1] [BHS81] started around 1976 to offer method guidelines for the design of telecommunication systems. Complementary to the description of behavior based on state machines, SOM provided structural diagrams, which were not supported by early versions of SDL. These structural diagrams show processes connected by signal channels and identify processes with special categories of functionality, as for example resource allocation, translation and routing. In addition, SOM made use of graphical situation descriptions embedded into state symbols to characterize states for ease of understanding and validation. The early versions of SOM used simple sequence diagrams and story boards (so-called *cartoons*) to specify scenarios. This was later replaced by MSC, and the EFSM notation was replaced by SDL processes when these languages matured.

SOM not only dealt with the definition of system designs, but also with implementation and execution issues. To implement a system, SOM by default used a runtime support system (RTS) for extended finite state machines (EFSMs) that was shared among many state machine instances. This provided a lightweight

[1]The acronym SOM was initially spelled *Structure-Oriented Method*. This name was later adjusted to emphasize its suitability for SDL.

concurrency support that was necessary since the concurrency support offered by programming languages such as CHILL [Rek82, ITU99] is *"too general to give an efficient implementation"* [BHS81]. Instead, the EFSM support introduces an additional multiplexing level for processes. Since the execution times for each transition is negligible, no preemption among the state machines is necessary. This enables efficient scheduling schemes and freedom in the realization of the implementation mechanisms. The software architecture suggested by SOM is divided into three parts:

– the EFSM support as described above, responsible for scheduling and execution of state machines.
– interface software, for signal transmissions between components, using input and output processes of the underlying hardware
– application-specific procedures called by the transitions of the state machines and the corresponding data.

As we have argued in the previous chapter, this architecture is still useful in combination with current technologies. For that reason, we will take SOM's execution model for state machines also as basis for our approach, as we will present in Chapt. 4 and detail in Paper 2 of Part II.

3.1.2 TIMe, The Integrated Method

TIMe was developed in the SISU projects [SIS96], that were carried out between 1988 and 1996. The method is covered in a textbook [BH93] and further elaborated in an electronic handbook [SIN99]. While the execution and modeling paradigms based on the state machines were largely carried over from SOM, TIMe integrated the now matured version of SDL with the use of Message Sequence Charts (MSC, [ITU04]) and some parts of UML: UML to capture objects and their relations, MSCs to specify interactions between components, and SDL processes for the local behavior of components.

TIMe supports the two perspectives on systems outlined in Sect. 2.2 by distinguishing between objects and properties. Objects correspond to what we call components, i.e., the description of structural parts in the system, how they are composed and connected, and what their behavior is. Property models describe properties of objects or groups of objects from the outside. Property models were mainly expressed by MSCs. For object models, UML classes and SDL processes were used.

Similarly to other methods, TIMe distinguishes between problem analysis, specification, design, implementation and instantiation. During the specification, properties are described. TIMe gives advice on how to build designs in the form of SDL processes based on these properties, and how they are kept consistent with the property models. While tools may help for the alignment of some elements (for example to ensure that signal names in MSCs correspond to those in SDL processes) this alignment is mostly carried out manually. TIMe further describes the implementation design, i.e., how to implement an SDL description, and discusses the mechanisms for execution.

TIMe proposes the usage of frameworks as a means to organize reuse across system families. Following this approach, a system is partitioned into an application-specific part and an infrastructure part, where the latter can be reused in other systems. As an example of this type of reuse we refer to ServiceFrame [BHM02]. In this framework, system components (called *actors*) are amended with some default behavior that can be used to create sessions between them, using a dedicated protocol. The default behavior controls the life cycle of actors, which consists of some management states (like *idle* and *paused*) and an active state *playing*. Application-specific logic is added by extending the state *playing* with transitions and logic.

Like most of the SDL-based approaches, TIMe has a strong focus on components as the design units. Although collaborative behavior among several components can be described by means of MSCs, direct reuse of such behavior is rather difficult. Although some reuse is possible using the framework approach explained above, in the end, application-specific logic must be provided in the form of state machines for each participating component. We will therefore, as already indicated in the last chapter, try to reuse collaborative behavior in a more direct way, as described later.

3.1.3 Real-Time Object-Oriented Modeling, ROOM

Selic et al. describe ROOM (Real-Time Object-Oriented Method, [SGW94, SGME92]), which uses many concepts known from SDL (see [Wie98] for a comparison). Similar to SDL, ROOM expresses system designs in form of interconnected objects, called *actors*, which have their own thread of control. The behavior of these actors is described by state machines similar to Harel state charts [Har87], called ROOM charts. They describe local behavior triggered by signal receptions. Similar to gates and channels in SDL, actors send their messages via ports. In ROOM, however, protocol specifications may be attached to ports, specifying the legal message sequences. The method is supported by tools from ObjecTime (later Rational and IBM), and offers the animation of specification as well as automatic code generation [SGME92]. The code is executed with help of a runtime system, based on similar principles as those described in 3.1.1 (see [SFR97]). ROOM was later aligned with UML [Sel98] from which UML-RT developed, and elements from ROOM found they way into UML 2.0.

The problems with reuse of collaborative behavior discussed already in Sect. 3.1.2 also apply for ROOM.

3.2 Model-Driven Development and MDA

With the Model-Driven Architecture (MDA, [Obj03]) the Object Management Group (OMG) described a framework for software development approaches and argues for the extensive use of models within a development process and the (possibly automated) transformation of these models to executable code. The aim is to provide more independence from specific platforms, so that application or business logic (as a valuable assets) does not have to be changed when implementations evolve. As the MDA guide [Obj03] acknowledges and Sherratt [She05] points out, these ideas are rather well-known. For example, Bræk uses the term *translation-based* approach in [Bræ00] for an SDL based approach that focused on the definition of models, the separation of application logic and platforms and the use of automated code generation. Another term for a similar paradigm is *specification-driven* development, as used for example in [Pit06]. For that reason, it may be more accurate to understand MDA as a consolidation or standardization effort to align languages, notations, viewpoints, techniques and tools to enable such model-driven approaches in a range of domains. These standards include

– a language called *Meta Object Facility* (MOF, [Obj06]) to define metamodels for other languages, like UML,
– model-based transformation languages like QVT [Obj07a] to define automated mappings between models,
– profiling mechanisms, that means the extension of UML models by stereotypes, for example to annotate models in order to guide transformations and add semantics.

MDA declares three viewpoints on a system: a computation-independent viewpoint, a platform-independent viewpoint and a platform-specific viewpoint [Obj03]. The corresponding models are also referred to as *CIM*, *PIM* and *PSM*. In SDL-based methods like TIMe, a computation independent view may be provided by a property model, for example in form of MSC, focusing on the interactions of the system with the environment. A SDL design consisting of blocks and processes corresponds to a platform independent viewpoint (as SDL specifications may be implemented on different middleware platforms and programming languages). A concrete implementation derived from SDL, for example in CHILL, would then constitute a platform specific model (see [BM05]).

Apart from the causal relationships of CIM, PIM, PSM and the resulting implication on which models have to be elaborated or transformed in which sequence, MDA does not explicitly specify a detailed development process and gives only generic method guidelines. Therefore, MDA must be adapted in order to provide working solutions. For instance, executable UML is related to MDA in [RFW04]. In [BM05] and [KGW06], existing and established SDL-based approaches are expressed within the framework of MDA. Using the profile in [ITU07], SDL specifications can be expressed as UML 2.0 models, so that the SDL-based approaches described above can be used with the MDA framework as well.

The focus on modeling implied by MDA provides a good working ground for formal techniques. Similar to work around the SDL language, formal approaches can be used to define the semantics of models, analyze them or reason about refinement relations between models. For instance, STAIRS [HHRS05] defines semantics and refinement operations of UML sequence diagrams. In [GH04], Graw and Herrmann use cTLA to formalize UML models on different abstraction levels and show their correspondence by cTLA refinement proofs. Pitkänen [Pit06] defines a specification-driven approach based on UML and the DisCo language and formalism which also fit into the MDA framework.

For our work in the following, MDA is a technical framework in which we define our UML-based models, their semantics by means of profiles, and bridges between them by means of model transformations.

3.3 Temporal-Logic-Based Approaches

The model-based approaches as outlined above may be complemented by logic reasoning. SDL, for example, has its semantics defined by means of the abstract state machine logic and counts therefore as a formal description technique (FDT). This is relevant when we want to describe the semantics of a language, reason about properties of specifications and verify that an implementation actually implies the desired behavior expressed by a specification. For our work, we have chosen temporal logic as basis for logical reasoning, since, as already mentioned in the introduction, we consider the properties of superposition, the joint action composition and the refinement relations as a suitable means to ensure consistency in our approach. Therefore, we introduce in the next sections the Temporal Logic of Actions (TLA, [Lam94]), DisCo [JKSSS90] and compositional TLA (cTLA, [HK00]).

3.3.1 Temporal Logic of Actions: TLA

The Temporal Logic of Actions (TLA, [Lam94]) is a variant of linear-time temporal logic. It describes behavior as an infinite sequence of states[2], where a state is an assignment of values to variables. Actions are defined as boolean expressions on primed and unprimed variables. Primed variables refer to the next state, so that an action can be true or false for a pair of states. If an action is true, it describes a step between two states. Specifications are given as a conjunction of an initial state and disjunctions of all actions. Liveness properties can be added in the form of weak and strong fairness formulas that are conjoined to the specification, as well.

In [AL95], Abadi and Lamport describe the composition of specifications based on shared states. In this composition style, two components communicate via a shared variable. Alternatively, composition can also be done by joint actions [BKS88], as used for example in DisCo and cTLA. In a joint action, several actions that are attributed to different components execute within the same step (as conjunction). A taxonomy of these different composition styles is given in [Lam02].

One important question during system development, as mentioned above, is if a certain specification implements another one. In TLA, this implementation

[2]Finite behaviors are expressed by infinite ones which stutter from a certain state on.

relationship corresponds to logical implication. To prove a refinement relationship, one can give a refinement mapping [AL91] that maps the variables of the different specifications to each other, possibly introducing new ones. We give an example for a refinement proof in App. A.

Specifications are written in the language TLA$^+$. This language can (with some restrictions) serve as input for the model checker TLC [YML99]. With this tool, theorems on specifications can be verified mechanically. As usual in model checking, once TLC finds a theorem violated, it presents an error trace, showing the state sequence that leads up to the violation of the theorem. Details about TLC are presented in Paper 7.

3.3.2 Distributed Cooperation: DisCo

The specification technique and language DisCo [JKSSS90] describes reactive behavior of distributed objects based on the principles of action systems [BKS83] introduced by Back and Kurki-Suonio. Although the principles used in DisCo have been developed before the occurrence the Temporal Logic of Actions, the semantics of DisCo are defined in TLA, as described in [Jär92, KS05]. Operations involving several objects are expressed in form of joint actions [BKS88], which describe a synchronization between the participating objects. Objects not part of a joint action remain unchanged. Groups of related actions and objects are combined in the specification units of DisCo, so-called layers.

Several layers can describe a system on various levels of abstraction, and joint actions modeling a synchonization among several objects within one atomic step may be refined to use certain communication mechanisms [KSM98, KS05]. The refinement of layers is based on the principle of superposition [BKS83]. A superposition step extends an existing specification layer and yields a new one. In a superposition step, additional variables and actions may be introduced. However, each action must either be a new action (that means the original specification stutters) or a refinement of an existing action. This is a constructive way of refinement; it ensures that original safety properties are maintained, so that the new specification implies the original one. With its syntax, the DisCo language supports the definition and superposition of layers. Several layers may also be composed with each other. Such a composition step corresponds to a simultaneous superposition of the composed layers.

The superposition-based design allows different development strategies, described in [KS05]. In a *top-down* manner, an initial abstract specification layer may be refined until the desired degree of details is reached. Following a *bottom-up* strategy, layers modeling system parts are composed until the total system specification is obtained. In addition, following an *aspect-oriented* style, layers may have a common predecessor but address different areas of concerns. These different layers are then composed to a complete system specification.

With the DisCo tools [AKP01], specifications may be animated, and formally verified using a mapping [Kel97] to the input of the PVS theorem prover [ORS92].

3.3.3 Compositional TLA: cTLA

With compositional TLA (cTLA, [HK00]) Herrmann and Krumm describe an extension to TLA that facilitates the composition of system specifications from processes. There are two forms of processes, simple ones and compositional ones. A simple cTLA process consists of variables and actions, similar to a TLA$^+$ specification. Actions may only access variables declared within the same process. Compositional processes declare instances of the processes. These instances are composed using the principles of joint actions: Each joint action of the compositional process is a conjunction of actions of the instantiated sub-processes. Each instantiated process contributes with exactly one action to the joint action: either a regular action, or a stuttering step. Action parameters can be used to pass values between process instances. Through these constraints, process composition in cTLA has the properties of superposition as described above for the composition of layers in DisCo. To support the construction of valid process compositions, cTLA provides a programming-language-like syntax.

The compositional style is especially useful when verifying systems, as we will explain in Sect. 3.4.3, where we present the application of cTLA to the domain of protocol engineering. Further details on cTLA are introduced in papers 2, 3, 6 and 7.

3.4 Focus on Collaborative Behavior

The engineering approaches based on SDL and UML presented in Sect. 3.1 express collaborative behavior mainly by means of MSCs or UML interactions, often in the form of incomplete scenarios, without focus on the necessary local behavior. In the following, we will discuss various other ways to specify collaborative behavior, and pay special attention to approaches that express complete behaviors.

3.4.1 Collaborations in Protocol Engineering

Communication between physical components is based on communication protocols, which are essentially collaborations: Not only must the exchanged data units have specific formats, but also the sequence in which messages are exchanged between the protocol entities must follow certain ordering rules. In the OSI reference model [ITU97a], collaborations are going on among the protocol entities within a protocol layer, utilizing the communication primitives (or services, as they are called in that context) of the lower layers.

Different specification styles and notations have been used to specify protocols, among them state machines, Petri nets, programming languages or combinations of these languages. A survey can be found in [BD02]. The different languages put more or less emphasis on an explicit representation of the collaborations. In this context, Diaz and Azema [DA85] emphasize the suitability of Petri nets as they *"allow an explicit specification of the synchronization* *constraints between different state machines"*, while the *"state machine representation does not explicit the interactions that exist between machines."* Merlin [Mer76, Mer79] describes a protocol engineering approach based on petri nets and presents a specification of the alternating bit protocol where one Petri net covers both protocol entities, the sender and the receiver. Similarly, Yamaguchi et al. [YEFvBH03] use extended Petri net models covering several protocol entities within one net, similar to the service specifications by von Bochmann and Gotzhein in [vBG86].

Another explicit representation of collaborative behavior is used in the description of the transmission control protocol (TCP, [Pos81]). A state machine is used to show the global states of a connection. However, the local behavior necessary to execute the protocol is not included in this diagram. More traditional state/transition based instructions for each protocol side are given for that purpose.

Despite these examples where collaborations are used in a quite explicit manner, it seems that protocols are most often described with the local behavior of the separate protocol entities. The protocol for basic call control in ISDN [ITU98] is for example specified by SDL processes for the user and the network, alongside some MSCs that illustrate some scenarios.

3.4.2 Constraint-Oriented Specification Style

Bolognesi and Brinksma [BB87] introduced the constraint-oriented specification style as one of four architectural specification styles [VSvS88, VSvSB91]. In contrast to a component-oriented style (called *resource-oriented* in [VSvS88]), where a system specification is decomposed into its physical components, in the constraint-oriented style, a system is decomposed in to a set of constraints, each one possibly spanning over several components. Therefore, constraints can be applied to collaborative behavior, focusing on a certain concern or task. This is especially useful in early design phases to capture requirements, as pointed out in [VSvSB91].

In [BB87], this style was applied to LOTOS, and [Bol00] describes an alignment with the object-oriented principles of Java. Herrmann and Krumm [HK98] used this style in combination with cTLA for the specification and verification of communication protocols, as described in the following.

3.4.3 Constraint-Oriented Protocol Specification in cTLA

Processes in cTLA may not only represent physical resources of a system, but also constraints spanning over several components, enabling a constraint-oriented specification style as introduced above.

In [Her97, HK98], these properties are utilized to define a framework for the specification of transport protocols in cTLA. This framework contains various elements that can be combined to compose more complex protocol functionalities. Fig. 3.1 illustrates the construction of a communication service. On the left hand side, the service specification, i.e., the externally visible behavior of

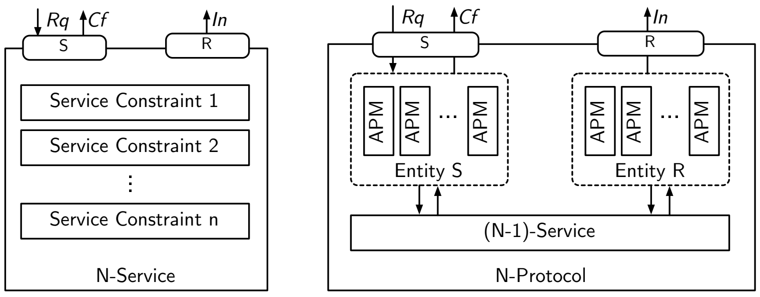


Fig. 3.1: Protocol specifications with cTLA (adapted from [Her97])

the service layer at the locations of the service access points, is given as a composition of basic service constraints, which the service must fulfill. These service constraints span over both participants. Examples for such service constraints are flow control or the absence of errors like re-orderings.

On the right hand side, the realization of this service specification is described. Each protocol entity is composed of a number of abstract protocol mechanisms (APM) that model basic functionality, as for example the management of sequence numbers. For the communication between the protocol entities, a lower layer service specification is used, in turn described as a service specification composed of individual service constraints.

The approach enabled by cTLA has properties that help to verify behavior: To prove that a certain set of protocol mechanisms fulfill a certain service constraint only a subgroup of service mechanisms (together with the underlying communication medium) have to be composed and considered. Due to the principle of superposition that holds for the process compositions, constraints can be composed together and the resulting system specifications obeys all of them. For this reason, we will use cTLA as semantic foundation for our method, as we will discuss later.

3.4.4 Collaborative Behavior in DisCo

As described in Sect. 3.3.2, the joint action specifications of DisCo describe behavior among its objects. Often, those joint actions and objects are grouped together within one specification layer that serve some common task, similar to the way we have decomposed the example in Sect. 2.2. In earlier work [KSK88], Kurki-Suonio and Kankaapää model a telephone exchange system, in which collaborations among participants are described explicitly using *virtual* objects. These objects describe the abstract states of the system as they occur in the initial (textual) description of the system requirements. As they do not directly interfere with the resources in the system, they can be seen as observers for behavior going on between the participants of the system and therefore also called *wiretappers.* Their behavior can be described by state charts, similar to the description used in the TCP protocol documentation described above. In this way, the collaborative description style from requirements is maintained and terms of

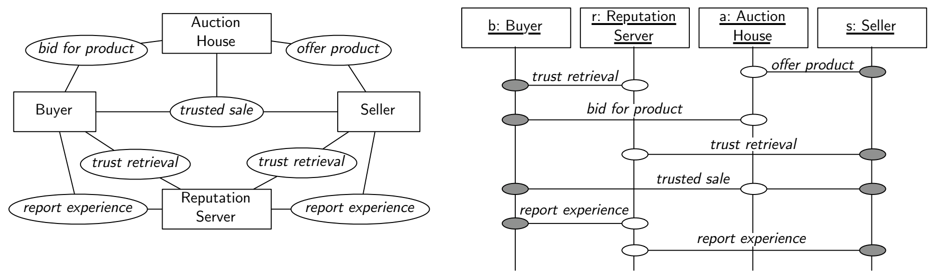


Fig. 3.2: Catalysis collaboration and action sequence diagram

the requirements find their direct correspondence in the system specification.

3.4.5 Roles and Collaborations in OOram

The Object-Oriented Role Analysis Method (OOram, [RWL95], formerly OORASS [RAB$^+$92]) is the first object-oriented method that made collaborations a *"central way of thinking"* [DW99]. OOram uses models that focus on behavior among components, as we have already outlined in Sect. 2.2. OOram explicitly uses the notion of collaborations by providing so-called collaboration views. These diagrams show how a set of roles, connected by ports and connectors, interact with each other. OOram focuses mainly on the features of roles that an object is composed of and less on the detailed behavior. Scenarios are described with an adapted form of message sequence charts. The collaborations employed in OOram had influence on the roles used in TIMe and influenced the concept of collaborations in Catalysis and the UML standard.

3.4.6 Collaborations and UML 1.x

In UML 1.3 [Obj00] and 1.4 [Obj01], collaborations are introduced as modeling elements. They describe how a set of participants (objects) are related to each other and communicate to handle a certain task. Collaboration diagrams show the interactions between the participants using messages, depicted as arrows. To define the order of interactions, sequence numbers are attached to the messages.

3.4.7 Catalysis

D'Souza and Wills describe the Catalysis approach [DW99], which has its roots in early UML-based techniques. Based on the ideas of joint actions in DisCo and the collaborations in OOram, Catalysis uses collaborations as specification units that complement component-oriented models.

A collaboration diagram in Catalysis shows how a number of objects may execute a set of joint actions together to accomplish a certain task. Figure 3.2 shows the trusted auction system as a Catalysis collaboration diagram. These joint actions are assumed to take some time, and it is possible to attach pre- and post-conditions to them, expressing how the state of the participating objects

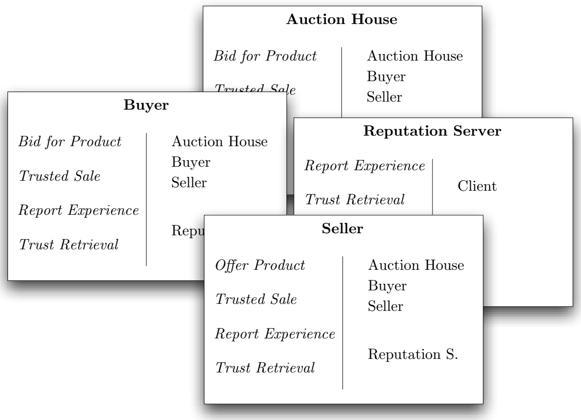


Fig. 3.3: CRC Cards for the trusted auction system

change by its execution. A joint action may be refined by a collaboration in a kind of refinement relation, which is called *"zooming"* in Catalysis. This refinement is, however, not treated formally. *Action sequence diagrams* document possible sequences of joint actions. Similar to MSCs, objects are represented by a lifeline. Joint actions are shown as lines orthogonal to the lifelines, connecting those objects involved in a joint action. At the crossing points, the local occurrences of actions are depicted by ellipses, with the possibility to identify the initiator of a joint action by a filled ellipse. In addition to these diagrams, Catalysis also uses diagrams similar to interaction diagrams of UML 1.4. Moreover, to illustrate the state changes of a joint action, it is possible to show *snapshots* of the involved objects, their connections and some of their variable values. A pair of snapshots can be used to illustrate the before and after states of a joined action, for instance.

While the concept of Catalysis collaborations allows us to focus on collaborative behavior, the final system design is given in the form of objects and operations, so that catalysis, in the end, is rather component-centered. The collaborations are merely used as guides through the design and implementation process, and are not utilized in a rigorous validation or verification.

3.4.8 CRC: Class, Responsibility, Collaboration

Beck and Cunningham developed a practical technique for learning about the global interactions within object-oriented programs called CRC (Class, Responsibility, Collaboration, [BC89]). Each class that is part of a system is represented by a paper index card. Fig. 3.3 shows an example for some classes of our trusted auction system from Sect. 2.2. The left column of a card identifies all responsibilities of a class, and the right column lists the collaborators. The cards are used as a basis for a process to identify the collaborations among objects and find out their responsibilities. Developers are encouraged to move the cards around

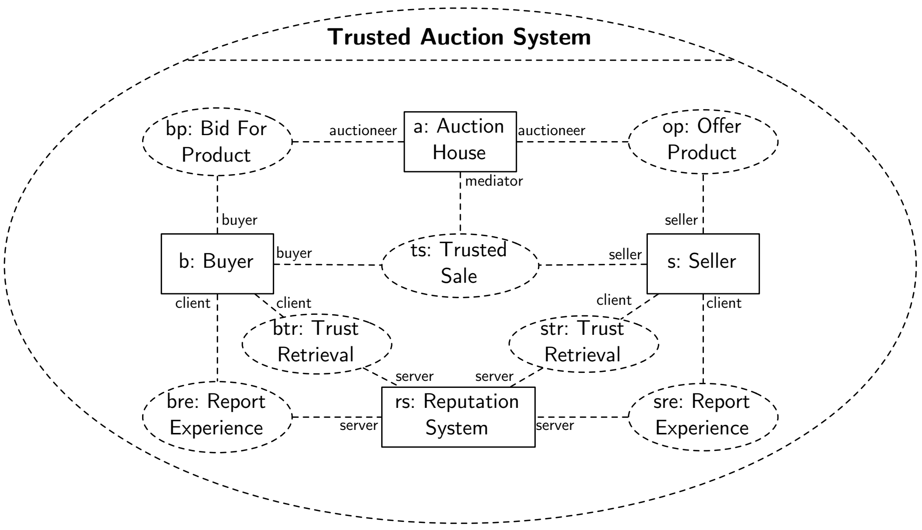


Fig. 3.4: UML 2.0 collaboration for the Trusted Auction System

and execute some scenarios. During this process, classes may be modified, new classes may be introduced and responsibilities may be re-assigned when a scenario reveals better solutions for a problem. This technique is therefore most useful in very early design phases, where detailed collaborations have to be elaborated in the first place.

3.4.9 Design Patterns

Design patterns [BC87, Coa92] are used to document solutions to reoccurring problems in object-oriented programs. As these solutions usually involve several objects, patterns essentially describe collaborative behavior. The description style used for design patterns is often quite informal and illustrative; in [GHJV95], for instance, patterns are specified by UML class diagrams for the method signatures of the participants and the connections among them, while the behavior is covered implicitly by code fragments of some illustrative sequence diagrams. This implies that patterns have to be understood and integrated manually by a programmer.

3.4.10 UML 2.0 Collaborations

Influenced by methods such as Catalysis and OOram described above, the concept of collaborations was reworked in version 2.0 of UML [Obj07b]. (The original collaboration diagrams of UML 1.x are now called *communication diagrams.*) Fig. 3.4 shows a UML 2.0 collaboration[3] of the trusted auction system known

[3]To distinguish between the general concept of collaborations and the specific UML elements, we will in the following refer to the latter explicitly as *UML collaborations.*

from the introduction. Participants of the system are depicted by collaboration roles, such as the buyer *b* or the seller *s*. The system is decomposed into sub-functionalities (or sub-services, as we may call them) by so-called collaboration uses. Although the employed notation is different from the one presented in 2.3, its information content is effectively equivalent. The collaborations are showing the horizontal decomposition of a system, such as the collaboration views in OOram. The trusted auction example will be discussed in detail in Paper 5.

The UML standard focuses in particular on the structural aspects of UML collaborations. UML does not, however, elaborate detailed semantics of the behavioral implications of the structural composition. Collaborations are intended as a context in which behaviors may be defined. Compared to the other uses of collaborations, and what we need, this is an obvious shortcoming. We will later see how a combination of collaborations with activities may solve this problem.

"Ah, remember Ted, pieces of the puzzle make funny shapes,
but they still fit together in the end."
– Nicky Flippers, Hoodwinked

CHAPTER
FOUR

THE ENGINEERING METHOD *SPACE*

The papers of Part II introduce *SPACE*[1] as one concrete constellation of notations, semantics and algorithms to form an engineering method. In this chapter, we present the approach in its entirety. We start in Sect. 4.1 with a compilation of the key decisions that shaped our work and we outline the development process implied by SPACE. We continue in Sect. 4.2 with an overview of the UML specifications and their features. The semantic foundation of the method based on temporal logic is explained in Sect. 4.3. In this section we also provide an overview of how logical reasoning can be done, especially with respect to the verification of implementations and the automatic model checking. In Sect. 4.4, we provide an overview of the model transformation from collaborations to executable components. We close in Sect. 4.5 with a presentation of the provided tool support.

4.1 Overview of the Approach

Based on the challenges observed in Chapt. 2, the experience from the methods presented in Sect. 3.1, the properties of temporal logic approaches from Sect. 3.3, as well as the discussion on the role of collaborations in specifications in Sect. 3.4, we present here the five key decisions that formed our approach and guided our efforts to answer the research questions of Sect. 2.3.

1. **Collaborations as Major Specification Units**

   We use collaborations as main specification units. However, in contrast to approaches like Catalysis (Sect. 3.4.7), we let collaborations be the *only* manually edited specification units, from which everything else is derived. Therefore, collaborations describe complete behavior between a set of participants like processes in cTLA or layers in DisCo. This includes

[1]The name came up in a discussion with Geir Hasnes. Since then, SPACE stands for specification by activities, collaborations and external state machines.

both their local behavior and the necessary interactions. The specifications for collaborations are given as coherent, self-contained building blocks. To abstract from internal details, they have in addition a description only referring to the externally visible events used for composition.

2. **cTLA as Foundation for Logical Reasoning**

   As a formal basis we chose temporal logic and in particular cTLA. We formalize collaborations as cTLA processes with the style cTLA/c. This enables us to reason about refinement relations between models, and to use the composition mechanism of joint actions to support superposition. However, we will hide this formalism completely from the engineers working with the approach. Instead, we use UML activities which are capable of expressing collaborative behavior and the joint action composition in a visual form.

3. **Analysis via Model Checking**

   To ensure the correctness of collaborations and their composition, we use model checking. However, instead of checking the entire system, we utilize the compositional properties guaranteed by our collaboration semantics in cTLA. According to them, we can check collaborations separately. Compositions of collaborations are checked using the abstract external descriptions, effectively reducing the state space. Due to the superposition principle, we know that the composed system will behave as expected.

4. **Execution Model Based on State Machines**

   Like SOM and ROOM, we define a runtime support system for the event-based execution of state machines that will be the basis for the operational semantics based on run-to-completion steps. We will formalize this runtime system in temporal logic with the style cTLA/e, to enable reasoning as described in the next point.

5. **Component Synthesis by a Model Transformation**

   Similar to SDL-methods such as TIMe, we utilize a multi-model approach, where one model focuses on the collaboration-oriented decomposition, while the other model decomposes a system into its components. However, we want to synthesize the component-oriented model completely from the collaboration-oriented model via a model transformation with the techniques of the Model Driven Architecture, MDA. To ensure the correctness of this transformation, we will establish and prove a refinement relations between the activities as source and the state machines as target models, that means a refinement between cTLA/c and cTLA/e specifications.

Figure 4.1 provides an overview of the development process implied by the approach. To ensure the consistency the building blocks and specifications as well as the correctness of the transformation, we use temporal reasoning across the entire approach, illustrated by the dashed lines. In the following, we will go through each element of Fig. 4.1.

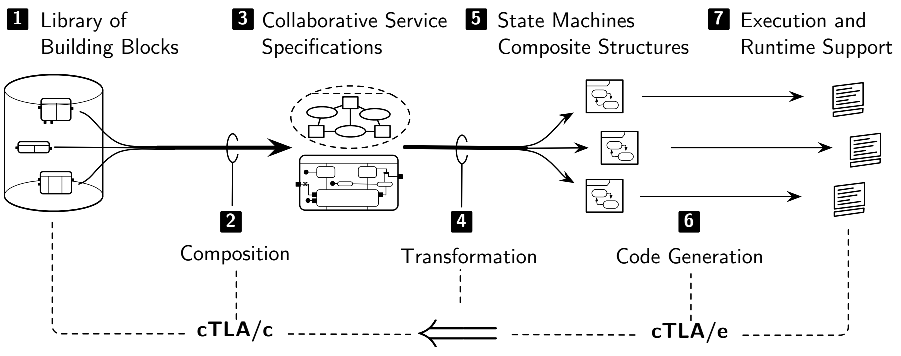

1 Library of Building Blocks
3 Collaborative Service Specifications
5 State Machines Composite Structures
7 Execution and Runtime Support
2 Composition
4 Transformation
6 Code Generation
cTLA/c
cTLA/e

Fig. 4.1: The SPACE Engineering Approach

1 Collaborations as the major specification can be designed, analyzed and reused as building blocks in the form we introduce in Paper 1 and summarize in Sect. 4.2.1. We will formalize collaborations in temporal logic to define their semantics in Paper 6. Engineers design the behavior of collaborations by writing UML activities. To hide details of the collaborative behavior, once a collaboration is reused, it has only to be considered by an external description. This external description is also exploited under model checking to reduce the state space. To reason about the correctness of the specifications, the correctness of their composition as well as the transformation, we introduce formal reasoning on the level of collaborative service specifications using temporal logic specification style cTLA/c (c for *collaborative*). One may build libraries of building blocks dedicated to certain domains, such as trust management, as we describe in Paper 5.

2 Since collaborations are temporal logic processes, we use the composition by joint actions of cTLA to compose collaborations with each other, as explained in Paper 6. This composition mechanism is general and can express all kinds of dependencies. Graphically, this composition mechanism can be mapped to activities as introduced in Paper 1, so that an engineer composing collaborative building blocks just has to instantiate them as sub-activities and connect them via activity flows using arbitrary glue logic. In this way, a system can be built successively, following either a bottom-up or top-down approach. In the end, a system is specified with a collaboration on the highest decomposition level.

3 Due to the semantics of the behavior as well as the composition in temporal logic, service specifications may be analyzed formally. While many checks are purely syntactical, the most interesting properties are assured via automated model checking, as exemplified in Paper 7 and presented in more detail in Sect. 4.3.3.

4 From such a complete, collaborative system specification we synthesize the components using an automated model transformation described in Paper 3, Paper 4, and summarized in Sect. 4.4. This is possible because the collaborations and the compositions specify the complete behavior. As mentioned, this transformation corresponds to a refinement step from cTLA/c to cTLA/e, discussed in Sect. 4.4.

❺ For the component models that are the result of this transformation, we use state machines as provided by SDL. We will, however, express them using stereotyped UML 2.0 state machines, as this is the input for our code generators, and to keep all employed models in the realm of UML. To enable the aforementioned refinement proof, we formalize the execution of state machines in temporal logic as well. We do this by prescribing a special specification style, called cTLA/e. It is introduced in Paper 2 and 4.3.2.

❻ From these executable state machines, we can generate executable code via code generation, as described in [Kra03, Stø04].

❼ For the execution we have chosen the principles going back to SOM (see Sect. 3.1.1) based on a runtime-support system. As one concrete framework we use ServiceFrame as briefly introduced in 3.1.2 and Paper 2.

4.2 The Nature of Specifications

In the following, we will present the specification style used in SPACE. Most of the papers of Part II present detailed examples. Therefore, we only outline the main features the specifications here with the aim to provide and illustrate the general specification structure and main points; for the details, we refer to the individual papers and the appendix.

Figure 4.2 illustrates the decomposition of the *Trusted Auction System* presented in Paper 5. At the system level, a UML collaboration, stereotyped as ≪system≫, identifies by means of collaboration roles the components of the system, for example the seller *s* and the buyer *b*. Between them, occurrences of sub-services are denoted by collaboration uses, which refer in turn to collaborations, as for instance the trusted sale *ts*.

As mentioned in Chapt. 3.4.10, UML collaborations alone do not specify any behavior, but only show how functionality may be decomposed. Therefore, as Paper 1 introduces, we attach to each UML collaboration a UML activity which focuses on the behavior of collaborations as well as how behaviors of subordinate collaborations are composed. Figure 4.3 illustrates the corresponding decomposition of the system using activities. In comparison to the UML collaborations, the activities can also refer to building blocks that involve only local behavior of a single component, such as *timeliness observer*. The detailed specification of the trusted auction example is explained in Paper 5.

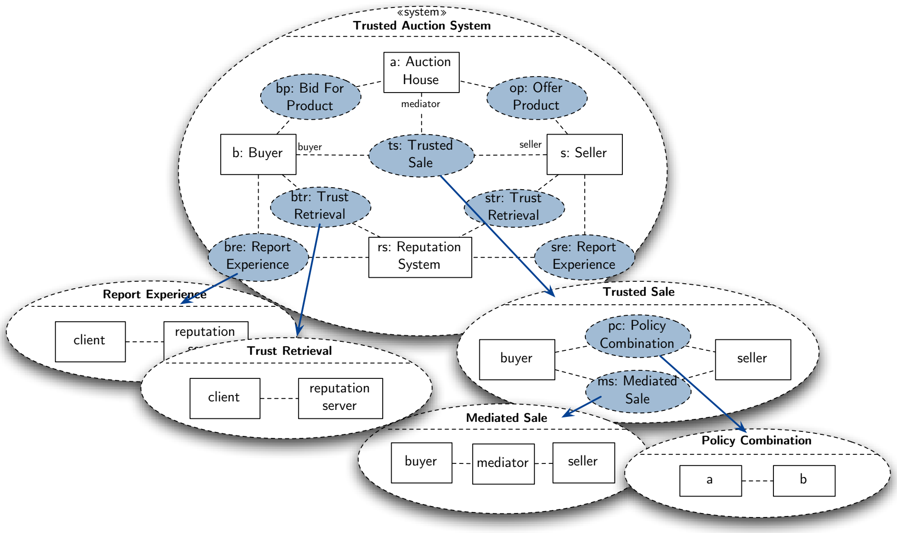


Fig. 4.2: Illustration of a system's decomposition by UML collaborations

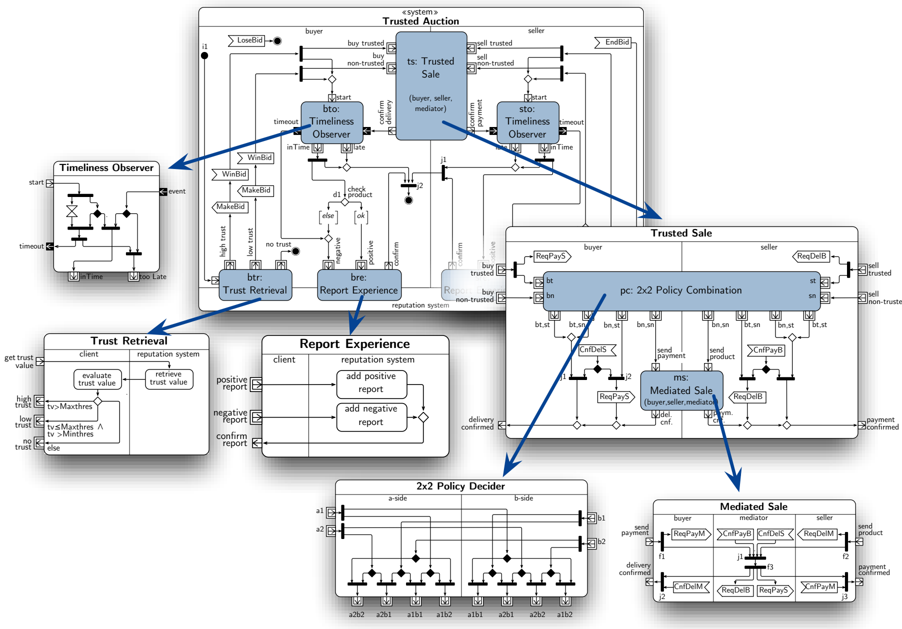


Fig. 4.3: Illustration of a system's decomposition by UML activities

4.2.1 Building Blocks

Paper 1 introduces the building blocks used for specifications along some examples. Building blocks are the units of specifications; they may be designed, analyzed and reused separately from each other. The behavior and structure of building blocks are described by the following diagrams:

- The structure is described by a UML 2.0 collaboration. If the building block is elementary, like *Trust Retrieval* in Fig. 4.2, it only declares the participants (as collaboration roles) and connections between them. If it is composite, like *Trusted Sale*, it may additionally refer to other collaborations between the collaboration roles by means of collaboration uses.
- The internal behavior is described by a UML activity. It is declared as the classifier behavior of the collaboration and has one activity partition for each collaboration role in the structural description. For each collaboration use, the activity declares a corresponding call behavior action referring to the activities of the employed building blocks. To compose the behavior with other building blocks, activity parameter nodes are used that can be connected to other elements when the activity is referred to by a call behavior action in a composite collaboration.
- The external behavior is represented by a special UML state machine, a so-called *external state machine* or ESM. Its transitions refer to the activity parameter nodes of the activity. In this way, it specifies the allowed sequence in which tokens may pass through the parameter nodes of the activity. This can be used as a contract when the building blocks is instantiated and composed. In model checking, the ESM is used to reduce the state space, as it abstracts away some inner states.

Depending on the number of participants, connectivity to other blocks and level of decomposition, we distinguish three different kinds of building blocks, which use different combinations of the diagrams itemized above.

- The most general building block is a collaboration with two or more participants providing functionality that is intended to be composed with other functionality, like *Trusted Sale* in Fig. 4.3. We refer to such a building block as a *service collaboration*.
- A special building block is a *system collaboration*, which is a collaboration on the highest composition level, like for intance *Trusted Auction System*. In contrast to a service, a system is closed and cannot be composed with other building blocks. Consequently, it has no ESM as its activity has no activity parameter nodes.
- Building blocks that involve only local behavior of one participant are referred to as *activity blocks*. They are represented by activities and ESMs and have no UML collaborations. An example is *Timeliness Observer* in Fig. 4.3.

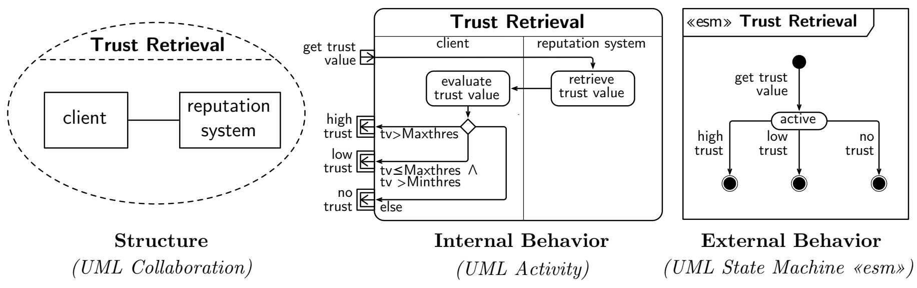


Fig. 4.4: Diagrams for a building block

4.2.2 Composition of Building Blocks

For the composition, UML collaborations and UML activities are used complementary to each other; UML collaborations focus on the role binding and structural aspect, while UML activities complement this by covering also the behavioral aspects for composition. For this purpose, call behavior actions are used, as explained in Paper 1. Each call behavior action represents an instance of a building block and refers to an activity. For each activity parameter node of the referred activity, a call behavior action declares a corresponding pin. Pins have the same symbol as activity parameter nodes and represent them on the frame of a call behavior action. Arbitrary logic between pins may be used to synchronize the events of the building block events and transfer data between them. There are different kind of pins (resp. activity parameter nodes), illustrated on the building block in Fig. 4.5:

- Starting pins (like *start*) activate the building block, which is the precondition of any internal behavior.
- Streaming pins (like *event* and *timeout*) may pass tokens throughout the active phase of the building block.
- Terminating pins (*inTime* or *late*) mark the end of the block's behavior.

If collaborations may be started or terminated via several alternative pins, they must belong to different parameter sets. This is visualized in UML by an additional box around the corresponding node. The sequence of allowed token passes is further constraint by the description given by the ESM of the building block.

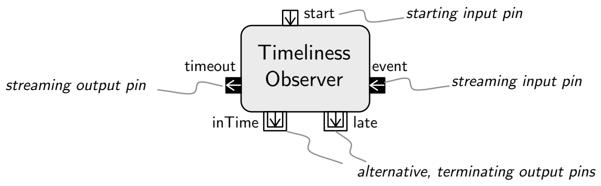


Fig. 4.5: Timeliness observer with pins

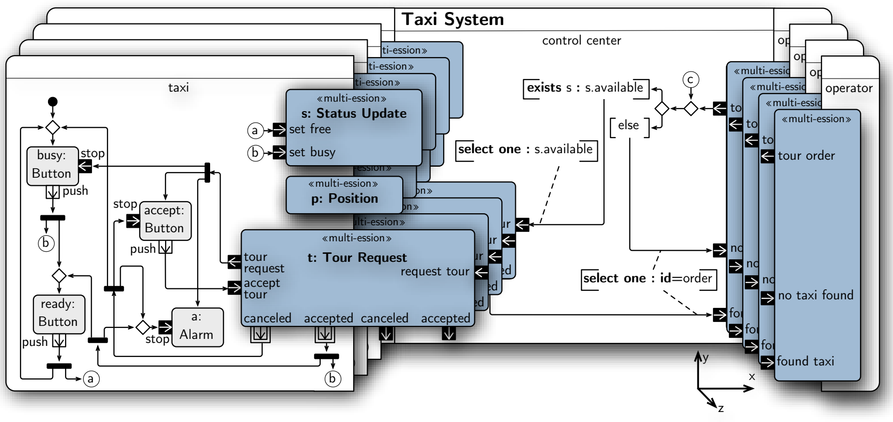


Fig. 4.6: Illustration of session multiplicity

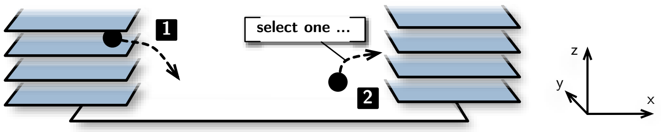


Fig. 4.7: Illustration of session selection

4.2.3 Multiplicity

The taxi control system presented in Paper 4 contains many taxis to be coordinated and many operators receiving tour orders from customers. Fig. 4.6 shows the system specification, with an activity partition for the taxi, the control center and the operators. To reflect the multiplicity of taxis and operators, their partitions are illustrated by several layers. This multiplicity of partitions implies a certain multiplicity of collaborations that have to be coordinated by the participants: Each taxi only has to handle one instance of each of the collaborations with the control center. The control center, however, has to handle several instances of the collaborations with the taxis and operators.

Figure 4.7 illustrates a situation with multiple collaboration instances (at the sides) and a partition (in the middle) together with token flows between the collaborations. A token arriving from any of the collaboration instances (❶) simply enters the partition. Vice-versa, when a token should enter a specific collaboration instance from the partition (❷), we need to determine which instance should receive the token. UML does not give any means to select such sessions. Therefore, we described in Paper 4 a selection operator and explain in Paper 6 how such a selection is expressed in cTLA/c.

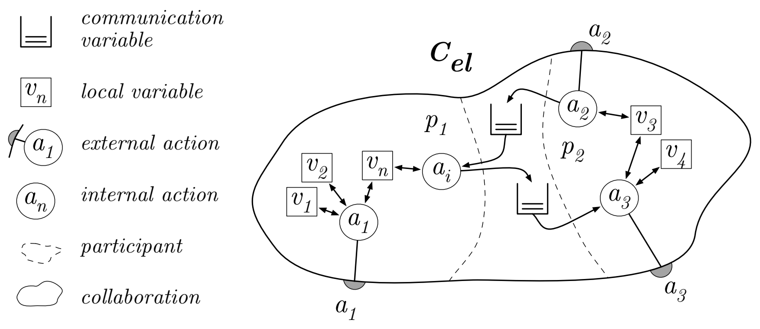


Fig. 4.8: Illustration of an elementary collaboration (from Paper 6)

4.2.4 Extensions to UML

In order to utilize the diagrams of UML, we made some extensions using stereotypes. System building blocks are marked by stereotype ≪system≫. With ≪environment≫, collaboration roles may be marked as part of the environment, whereafter no components will be synthesized for them. To express that collaborations between the same participants can be executed more than once, collaboration uses, which in standard UML have no multiplicity attached, can be marks as ≪multi-session≫, as introduced in Paper 4. In addition, we define some constraints on activity graphs to ensure our semantics and introduce a specialized form of a decision nodes on which nodes may wait, to model stateful behavior. All extensions are documented in detail in App. C. In the component-oriented designs, the behavior of components is modeled by ≪executable≫ state machines, which we discuss in detail in Paper 2 and App. E.

4.3 Semantics and Logical Reasoning

The semantics of SPACE is defined as two special specification styles in cTLA, illustrated by the dashed lines in Fig. 4.1. Collaborative service specifications expressed as UML activities are interpreted as cTLA/c formulas, while the executable state machines find their correspondence in cTLA/e formulas. Therefore, reasoning about refinement and the proof of properties in general is available, so that the approach is a formal description and development technique (FDDT) based on cTLA.

4.3.1 Collaborative Service Specifications: cTLA/c

In Paper 6, we describe the specification style cTLA/c, which formalizes the collaborative service specifications introduced in Paper 1. Collaboratins are mapped to cTLA processes. In contrast to general cTLA processes, cTLA/c contains the notions of participants, illustrated by the compartments of the elementary collaboration in Fig. 4.8. Actions are assigned to a participant and may

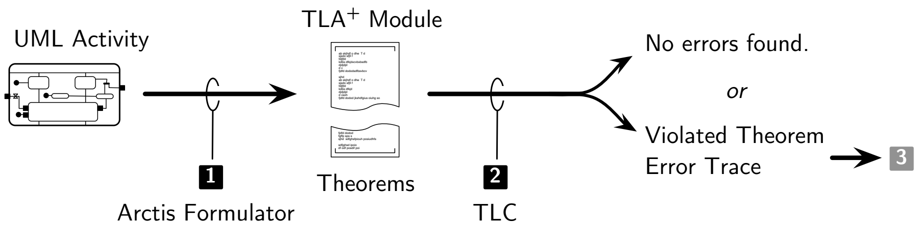


Fig. 4.9: Automated model checking in Arctis (Steps 1 and 2)

only access variables assigned to the same participant. To communicate, actions can access special, queue-like communication variables between the participants.

While cTLA/c is independent of any specific graphical representation, we chose to use UML activities in our approach. Therefore, Paper 6 also provides a set of production rules that generate cTLA/c from activities. These rules are formulated such that several activity elements of a flow are handled within the same cTLA action, representing a run-to-completion step during execution. This is important for the refinement proofs, in which we can show that the behavior of a system generated in form of state machines implies the original activities, as shown later.

4.3.2 Executable State Machines: cTLA/e

The state machines we use for the implementation of components via code generation have some constraints that we capture by the specification style of cTLA/e. In this style, each cTLA action corresponds to a state machine transitions. The communication of the participants via the communication variables is mapped to separate send and receive actions, and each transition must be triggered by either a reception of a signal or the expiration of a local timer, which is modeled by special local variables. The details are elaborated in Paper 2, where we also show the correspondence of this logical representation with the execution model and implementation.

4.3.3 Automated Model Checking and Analysis

Due to the semantics in temporal logic, we may use the model checker TLC to analyze specifications. Following the suggestion of Rushby [Rus00], we would like to hide the details of this process from the users, so that they do not need to work on formulas. Instead, users should be informed about problems in their specifications by explanations that refer to certain elements of the specification and tell what's wrong. For that purpose, we combine model checking with a syntactical analysis and interpretation and start the necessary tools from within the editor. An example showing how engineers can be supported in that way is presented in Paper 7. Figure 4.9 outlines the first part of the analysis prcess enabled by our tools.

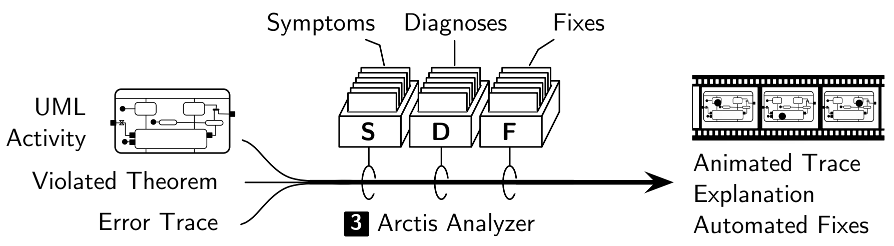


Fig. 4.10: Automated model checking in Arctis (Step 3)

1 In the first step, an activity is transformed into its TLA$^+$ formula using the Arctis Formulator developed in [Slå07]. In addition to the TLA actions needed to model the token flows of the activity, a number of theorems are added that must hold for the specification. These theorems are of two types:

- General theorems that must hold for any activity with our semantics, ensuring properties such as 1-boundedness of inner places (see Paper 3).
- Theorems derived from additional assertions added by the developer, such as the mutual exclusion of certain actions (see App. B).

2 In the second step, the TLC model checker is used to verify the specification against the theorems. If no theorem is violated, the analysis ends successfully. Otherwise, TLC reports which theorem is violated and presents a (textual) trace showing behavior up to the error, and the analysis continues with step 3, depicted in Fig. 4.10.

3 In a third step, the Arctis Analyzer takes the original activity, the theorem that is violated and the error trace as inputs. The violated theorem constitutes a *symptom*. Based on it, a number of diagnoses are considered. These diagnosis may trigger syntactic inspections on the original activity and the error trace to check if they apply. Depending on these additional inspections, explanations about the possible cause of the error may be provided to the user. In some cases, automated fixes can be provided based on a diagnosis. In addition, the error trace leading to the violation is visualized as animated tokens in the activity editor. For more details, especially for which errors can be found and fixed, we refer to the work of Slåtten [Slå08]. A comprehensive example with screenshots of an analysis is given in App. A.

4.3.4 Manual Logical Reasoning

If a user desires, an analysis based on temporal logic can also be done manually, based on the TLA modules of a building block. This corresponds to the approach shown for example in [Her06]. Such reasoning may also be applied when tools and algorithms are developed, as we will see in Sect. 4.4.4, where the correctness of a transformation is ensured by a refinement proof.

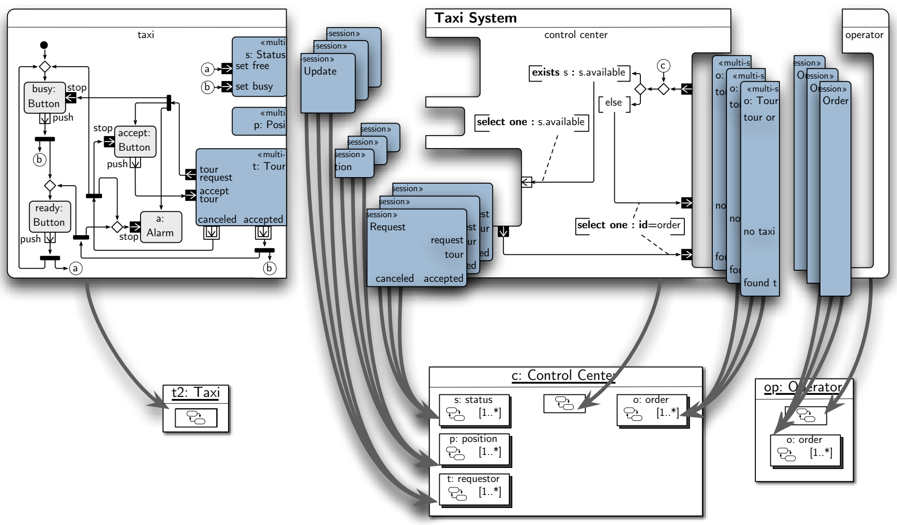


Fig. 4.11: Illustration of the component synthesis

4.4 From Collaborations to Components

To keep the models from the collaboration-oriented and the component oriented perspectives consistent with each other, we use a synthetic approach in which the components are entirely generated from the collaborations. This keeps the components consistent *by construction*. In this transformation, basically three things happen:

– The behavior expressed for each collaboration role is mapped to UML composite structures and session state machines, corresponding to the execution model.

– The collaboration behaviors expressed by the activities are split accordingly, and the flows crossing partitions are replaced by signal transmissions.

– The implicit state presentation of activities in the form of all possible token markings is transformed to a presentation using explicit control states in state machines.

The transformation of the collaborative service models to executable components is treated in Papers 3 and 4. Paper 3 discusses the detailed synthesis of state machines from activities. Paper 4 is concerned with the generation of composite structures containing those state machines so that also multi-sessions may be handled.

4.4.1 Generation of Component Structures

Figure 4.11 illustrates how the example of the taxi control system presented in Paper 4 is transformed into components. Each collaboration role (resp. activity partition) of the taxi system is implemented as a separate component. System components are modeled as UML classes. UML classes can define their behavior by means of a *classifier behavior*, which in our case is described by a state machine. This is a kind of default behavior; in addition, UML classes may declare further inner parts which can be state machines as well.

Since a taxi is only involved in collaborations with a single session (relative to partition *taxi*), all behavior can be integrated into the classifier behavior state machine of each taxi component. For the *control center* partition, the transformation results in a more complex component. The behavior executed by the control center partition is implemented by the classifier state machine of the component, while the behavior of the multiple sessions which the control center maintains with each taxi and operator is implemented by state machine arrays providing one state machine instance per collaboration instance. The details of this mapping of multiplicity and especially how the sessions are coordinated is described in Paper 4.

4.4.2 State Machine Synthesis

Paper 3 explains the mapping between cTLA/c described in form of activities to the state machines of cTLA/e. Some concepts of activities have their direct correspondence in state machines, such as decisions, guards and operations, which can merely be copied. The challenge, however, is to map the states of activities representing the implicit token markings to the explicit control states of state machines. For this purpose, the different run-to-completion steps of an activity have to be found and then transformed into state machine transitions. One activity flow will often have to be represented by several state machine transitions to handle different token marking states. For example, a join node with two incoming edges needs at least four transitions: A transition for each incoming edge for the case that the join is not yet complete, and one for each incoming edge for the case that all other incoming edges can offer a token, so that the join may fire. The number of transitions increases further if the join node is combined with other stateful nodes. To construct the transitions, the algorithm considers all possible token markings within a partition.

4.4.3 Extension of the Synthesis Algorithm

As argued in Paper 3, the separate and purely *syntactical* consideration of the partitions may result in transitions in the synthesized state machines that are never executed. While this is not a real problem, we have also written a variant of the algorithm, which considers all partitions and does not produce any superfluous transitions. This comes at the cost of a larger state space during synthesis, as not only the token markings of the partition under construction

$$\begin{tikzcd}
\textbf{UML Activity} \arrow[r, "\textit{Paper 3 / Arctis Transformer}"] \arrow[d, "\textit{Paper 6 / Arctis Formulator}"'] & \textbf{UML State Machine} \arrow[d, "\textit{Paper 2}"] \\
\textbf{cTLA/c} & \textbf{cTLA/e} \arrow[l, Rightarrow, dashed, "\textit{Refinement Proof (App. A)}"']
\end{tikzcd}$$

Fig. 4.12: Connections between formulas and diagrams

need to be considered, but also those in the other partitions as well as the queue places between partitions. However, the algorithm does not have to deal with the entire state space of the complete system either, since those collaborations that are not directly involved in the state machines under construction are replaced by their ESMs, in similar ways as in the strategy employed in model checking. So far, all examples tested were transformed within fractions of a second even for the extended version. Therefore, we plan to use the first variant only in those cases, where the state space is too large for the second variant. Both algorithms have been implemented as part of Arctis, which is presented in Sect. 4.5 and Paper 8.

4.4.4 Refinement Proof from Activities to State Machines

The transformation algorithm maps the activities to state machines so that there is a refinement relation between the corresponding cTLA styles. This means that the behavior of the state machines implements (or implies) the behavior of the activities. Figure 4.12 illustrates these relations. The correctness of the transformation of a system can be proven by a refinement proof using the corresponding formalizations. For the state machines, cTLA/e formulas can be derived according to the definitions of Paper 2. For the activities, the cTLA/c formula can be derived by the rules in Paper 6 or automatically by the algorithm implemented in [Slå07] as the Arctis Formulator. Detailed guidelines are provided in Paper 3, and we execute such a proof in App. A.

4.5 Tool Support for SPACE

Tools are an integral part of our strategy for rapid service development. Obviously, tools accelerate the development by providing editing support, automated model transformation and code generation, and reveal errors by consistency checks and model checking. Moreover, tools are an effective way to document a method and guide its users, which in the end also facilitates the application of formal methods by practitioners. We also used tools as one criteria in the evaluation of our approach, as discussed in Chapt. 6.

SPACE is compliant with UML 2.0. The proposed extensions to UML collaborations, activities and state machines as outlined in Sect. 4.2.4 and documented in App. C and App. E can be covered by UML profiles. Therefore, any modeling

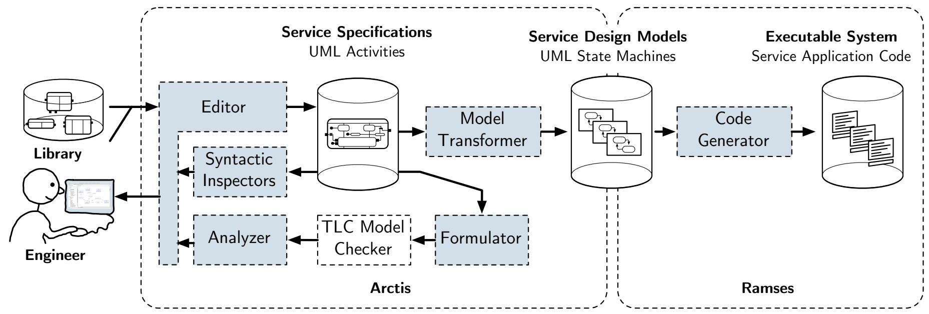


Fig. 4.13: Tool support for SPACE by Arctis and Ramses

tool compliant with UML 2.0 and the possibility to define profiles should be able to create the collaborative service specifications as proposed in this work. The implemented transformations of models to state machines respective TLA$^+$ modules can be integrated into existing tools by accessing a common UML repository, such as the one provided by the Eclipse Modeling Project [Ecl].

Nevertheless, we decided to support SPACE by our own tailored set of tools. This has the advantage that editing tools can directly support the claimed specification style, so that it is easier for the engineers to construct valid specifications. For instance, building blocks may simply be dragged into the editor to instantiate them, and error traces may be animated directly in the editor, as described in Sect. 4.3.3. The tools are partitioned into two sets of plug-ins:

– *Arctis* supports the construction of collaborative service specifications based on the building blocks expressed by UML 2.0 collaborations and activities, as well as the analysis of them and the transformation to the state machine-based models of Ramses.

– *Ramses* covers the more traditional, component-oriented part of the development, facilitating the implementations of state machine-based models as known from SOM or TIMe, similar to SDL.

The structure of the tools is sketched in Fig. 4.13. Arctis provides an editor for the service specifications which allows a user to create collaborations from scratch or compose existing ones taken from a library to create composite collaborations. In addition, Arctis contains the Formulator for TLA$^+$, the Analyzer and the model transformation as introduced above. The external model checker TLC is invoked by Arctis from the command line, invisible to the user.

A brief overview of the tools is provided in Paper 8. In Paper 7, we demonstrate the analysis via model checking of an example. A closer view of the tools with a more comprehensive example and screenshots is provided in App. A.

CHAPTER

FIVE

SURVEY OF THE PUBLICATIONS

Part II contains eight papers published between July 2006 and November 2007. To illustrate how the results of the individual papers contribute to the objectives of the thesis, we use the two-dimensional map in Fig. 5.1. Its horizontal axis sketches the development process presented in Fig. 4.1 on page 41. It starts with the library of reusable building blocks and their composition into service specifications, which are transformed into the component-oriented descriptions based on state machines, that are used to create an executable implementation via code generation. On the vertical axis, we distinguish between three conceptual levels:

– The logic level of formal reasoning. It is used to define the semantics and ensure the consistency of the specifications and their composition, as well as the correctness of the model transformation. In addition, it provides the basis for model checking.

– The modeling level, that means the level visible to the engineer developing a new system. It consists of the UML models of building blocks and composed systems. We also count the Java programs produced by the code generators for the implementation and execution as a part of this level.

– The level of tool support. This level assists the creation of systems using the modeling and programming concepts of the model level, indirectly ensuring the application of the concepts from the logic level. In addition, transformations synthesize the executable state machines.

The logical level is covered by the formalisms of cTLA/c and cTLA/e. On the modeling level, the formal constraints of cTLA/c and cTLA/e are captured by the two profiles we apply to UML and the techniques described in the papers as well as the generated Java programs. At the tool level, we distinguish between Arctis for service specifications and Ramses for state machines.

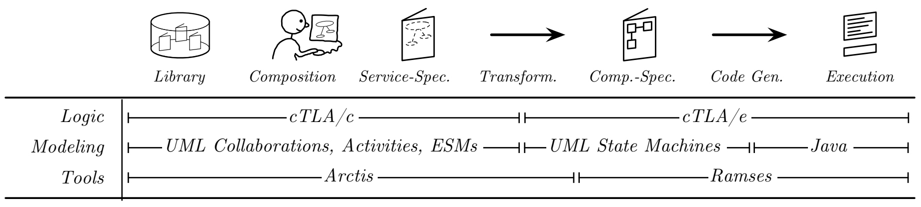


Fig. 5.1: Conceptual map of the papers' content

5.1 Individual Contributions

Most of the papers were published together with my supervisors Peter Herrmann and Rolv Bræk. I was the first author of all papers with exception of Paper 5. In Paper 1, I developed the main ideas of the approach, and was responsible for about 90% of the paper. In Paper 2, around 70% of the contents go to my account. The transformation algorithm presented in Paper 3 was developed together with Peter Herrmann, and I wrote 70% of the paper. As mentioned in Sect. 4.4.3, I implemented in addition an extended algorithm for the case that full model checking is possible. I developed the mechanisms for selection of multi-sessions described in Paper 4 and was responsible for 80% of the paper. In Paper 5, I was mainly concerned with the application of the approach to the example and its implications on the development method; trust-related aspects are mainly contributed by Peter Herrmann, so that my contribution of the paper is about 30%. The formalization of our approach in cTLA in Paper 6 was done in cooperation with Peter Herrmann, and my portion of the article is about 70%. Paper 7 presents the application of model checking based on the tool developed by Vidar Slåtten in [Slå07] under my guidance. I developed the initial example and wrote about 90% of the paper. For the submitted paper presented in App. A I have a similar share with the other authors as in Paper 7 so that my part of writing accounts for around 85%. I was the sole author of Paper 8 as well as of all chapters of Part I and appendices B to E.

5.2 Published Papers

In the following, we give a brief summary of each paper and how it contributes to the overall objectives of this thesis. We mark the primary areas addressed by a paper on the two-dimensional map introduced above.

Paper 1, together with Peter Herrmann:

Service Specification by Composition of Collaborations – An Example

Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology), IEEE Computer Society, 2006.

	Library	*Composition*	*Service-Spec.*	*Transform.*	*Comp.-Spec.*	*Code Gen.*	*Execution*
Logic							
Modeling	★	★	★				
Tools							

This paper is the first one presenting the paradigms used in SPACE. It provides an overview of the specification technique with collaborations and activities exemplified by an access control system. The concept of collaborations as building blocks is described. We present how UML activities can express the behavior of collaborations as well as describe how collaborations are composed precisely, utilizing activity parameter nodes and pins. We introduce the external state machines (ESMs) and explain how they define the externally visible behavior of building blocks. We demonstrate that collaborations can be used as one form of interface description, in which also local behavior of the partner using the interface is included. In the end, we proposed plans for the tool support and the model transformation into executable state machines, presented in Paper 3.

Paper 2, together with Peter Herrmann and Rolv Bræk:

Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services

Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), Volume 4276 of Lecture Notes in Computer Science, Springer, 2006.

	Library	*Composition*	*Service-Spec.*	*Transform.*	*Comp.-Spec.*	*Code Gen.*	*Execution*
Logic					★	★	★
Modeling					★	★	★
Tools					★	★	★

To enable a correctness-preserving, automated development approach, we establish the target modeling and specification style for our approach based on executable state machines. For that purpose, we describe the event-based execution of extended finite state machines based on a runtime-support system, similar to those used in SOM (see Sect. 3.1.1). To enable an easy and efficient scheduling mechanism, we describe constraints on UML 2.0 state machines which are summarized by a profile for executable state machines (see App. D). Moreover, we introduce the cTLA specification style cTLA/e to describe state machines corresponding to this profile. We continue by pointing out how the separate processes for the state machines may be composed to form a global system. In the end, we show how a correct implementation actually is a refinement of a cTLA/e process, and which properties cTLA/e processes (resp. executable state machines) have.

Paper 3, together with Peter Herrmann:

Transforming Collaborative Service Specifications into Efficiently Executable State Machines

Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT), Volume 6 of the Electronic Communications of the EASST, 2007.

	Library	*Composition*	*Service-Spec.*	*Transform.*	*Comp.-Spec.*	*Code Gen.*	*Execution*
Logic				★			
Modeling			★	★	★		
Tools				★			

We develop the transformation from activities to the executable state machines that is pivotal for the entire approach. After a recapitulation of the component-oriented and collaboration-oriented models, we start by considering the principles for mapping the concepts of activities to those of state machines. The most important difference is that state machines have an explicit control state, while activities represent states implicitly by token markings. In addition, activity flows crossing partition borders need to be replaced by signal transmissions. We continue by explaining an algorithm in detail which transforms activities given in an UML repository into state machines. The algorithm considers which elements of an activity imply events that trigger transitions in the corresponding state machine. For each of these events, and for each possible state, the algorithm creates state machine transitions by following the elements of the activity graph. The algorithm is implemented in the Arctis tool set presented in Paper 8. In the end, proof guidelines are given which explain in detail how the actions of the activities (cTLA/c) are mapped to the actions of the state machines (cTLA/e), so that the correctness of the transformation can be proven by a refinement proof in temporal logic, as we demonstrate in App. B.

Paper 4, together with Rolv Bræk and Peter Herrmann:

Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications

Proceedings of the 13th Int. SDL Forum, Volume 4745 of Lecture Notes in Computer Science, Springer, 2007.

	Library	*Composition*	*Service-Spec.*	*Transform.*	*Comp.-Spec.*	*Code Gen.*	*Execution*
Logic							
Modeling	★	★	★	★	★		
Tools							

In real systems, there are typically multiple instances of certain components, for example to provide access for several users. This usually implies that also multiple instances of collaborative behavior (also called *sessions*) are executed at the same time. When we compose such systems, we often need to specify precisely which exact session instance we want to address. However, although UML allows call behavior actions in activity to execute with several parallel sessions, it does not offer suitable means to coordinate them with the precision we need.

Therefore, we introduce stereotypes to model multi-session collaborations and operators for the management of session instances. While the **select** operator can be used to select specific instances from a set of sessions, the **exists** operator models decisions depending on current state of the sessions of a collaboration. We demonstrate this extension to specifications by means of the taxi control center, already mentioned in Sect.4.2.3. In addition, we describe how the existing transformation algorithm and code generators are extended to handle these extensions as well.

Paper 5, together with Peter Herrmann:

Design of Trusted Systems with Reusable Collaboration Models

Proceedings of the Joint iTrust and PST Conferences on Trust, Privacy, Trust Management and Security (IFIPTM), International Federation for Information Processing, Springer, 2007.

	Library	*Composition*	*Service-Spec.*	*Transform.*	*Comp.-Spec.*	*Code Gen.*	*Execution*
Logic							
Modeling	★	★	★				
Tools							

We present the application of SPACE to the domain of trust management by developing a trusted auction system. The example, already introduced as illustrations in Sect. 2.2 and 4.2, shows how a rather complex system gets manageable and understandable using the decomposition into collaborative building blocks. It is especially demonstrated how each of the building blocks can be understood separately, as they document collaborative behavior related to a certain task in a self-contained manner. We also stress another aspect supported by the development approach: The collaborative building blocks provide a basis for precise communication among experts of different domains. In the example given, an expert of trust can design generally useful collaborations for trust-based systems, while an expert in the domain of electronic sale systems may use these building blocks without a detailed knowledge of their contents.

Paper 6, together with Peter Herrmann:

Formalizing Collaborative Service Specifications using Temporal Logic

Proceedings of the Networking and Electronic Commerce Research Conference (NAEC) ATSMA, 2007.

	Library	*Composition*	*Service-Spec.*	*Transform.*	*Comp.-Spec.*	*Code Gen.*	*Execution*
Logic	★	★	★				
Modeling	★	★	★				
Tools							

This paper establishes the semantic foundation of the collaborative building blocks and how they are composed. We start the discussion by a recapitulation of the specification style used in the approach based on UML 2.0 collaborations and activities by an example form the home automation domain, including multi-session collaborations. We then describe the specification style cTLA/c for collaborations, already briefly outlined in Sect. 4.3.1. Elementary collaborations are mapped to simple cTLA processes, and composite collaborations are expressed by corresponding compositional cTLA processes. We formulate a set of production rules that describe how the graph of an activity consisting of activity nodes and edges is transformed to cTLA actions, and provide an example. To create entire systems, collaborations are composed, which, given the semantics in cTLA/c, can be formalized as cTLA process compositions. When the action of one collaboration is joint with the action of another one, there is a synchronous composition. Asynchronous composition comes in two variants: one where no multiplicity is involved, and one where only specific session instances are selected for composition. Both kinds are explained. In the end, we point out how the composition leads to a global specification that describes the behavior of the entire system.

Paper 7, together with Vidar Slåtten and Peter Herrmann:

Engineering Support for UML Activities by Automated Model-Checking

Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE), 2007.

	Library	*Composition*	*Service-Spec.*	*Transform.*	*Comp.-Spec.*	*Code Gen.*	*Execution*
Logic	★	★	★				
Modeling	★	★	★				
Tools	★		★				

We exemplify how the mapping from UML activities to temporal logic provided by Paper 6 can be used for the model checking of service specifications. For that, we use the Arctis Formulator (see Sect. 4.5 and [Slå07]) to generate a TLA^{+} module for each building block of the system, which are then checked incrementally (block by block) using TLC. The automatically generated theorems lead to the detection of errors that can be presented to the users which do not need to be experts in the field of formal techniques: Once an error is found, the corresponding trace is visualized within the activities, illustrating the problem to the engineer. In particular, the example highlights how situations in which several communication partners can take simultaneous initiatives, can be solved by introducing a reusable building block. Once the need for this building block is identified, its application is much easier and faster compared to a manual introduction of the solution on the level of state machines.

Paper 8:

Arctis and Ramses: Tool Suites for Rapid Service

Proceedings of NIK-2007 (Norsk informatikkonferanse), 2007.

	Library	*Composition*	*Service-Spec.*	*Transform.*	*Comp.-Spec.*	*Code Gen.*	*Execution*
Logic							
Modeling							
Tools	★	★	★	★	★	★	

In this paper we briefly outline the SPACE approach and describe the tools that Arctis provides to support the development of collaborative service specifications. In particular, these are tools for editing and syntactically inspecting specifications, tools for analyzing specifications based on model checking as demonstrated in Paper 7, and the model transformation presented in Paper 3. Ramses is presented with a focus on its code generation capabilities.

5.3 Appendices

Appendix A, together with Vidar Slåtten and Peter Herrmann:

Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services

Submitted to a journal for review.

We present the Arctis and Ramses tools using the example of a mobile treasure hunt, which is composed from a number of reusable building blocks. After a general introduction to our approach, we use Arctis to analyze a collaboration by means of the model checker TLC. The feedback to users by means of animated error traces and diagnoses and proposals for improvements is demonstrated. We close by transforming the system into executable components and discuss a number of related approaches.

Appendix B:

Refinement Proof for a System
We present a refinement proof for an example system (following the guidelines provided in Paper 3) that the behavior of the automatically synthesized state machines implies the behavior described by the activities of the collaborative service specifications. As both the formula for the activity and the state machines are written in TLA$^+$, we also use the model checker TLC to demonstrate that the given refinement mapping is correct.

Appendix C:

UML Profile and Semantics for Service Specifications
This report summarizes the syntax of service specification and the semantics of activities based on token flows. This form of description complements the

formalization in terms of cTLA/c given in Paper 6. All stereotypes are explained and summarized as UML profile, which is the base for the tools presented in Paper 8 and [Slå07].

Appendix D:

Building Blocks, Patterns and Design Rules for Collaborations and Activities
This report is a collection of useful building blocks, patterns and design rules that we gathered during our case studies. In comparison to App. B, it focuses on the strategies for solving certain re-occurring problems effectively.

Appendix E:

UML Profile for Executable State Machines
This report summarizes the syntax of the composite structures and executable state machines in form of a UML profile. This profile serves as a starting point for code generators, since models synthesized by Arctis conform to its constraints.

CHAPTER

SIX

DISCUSSION

The contributions of this work are presented in detail in the individual papers of Part II. In this chapter, we focus on the approach in its entirety. To do so, we start in Sect. 6.1 by presenting the main contributions of our work, as already briefly outlined in Sect. 1.1, and review our research question from Sect. 2.3 in Sect. 6.2. We close with some comments on the practical applicability of our results in 6.3 and discuss related work in 6.4.

6.1 Innovations and Results

We want to point out three characteristics of our method that support the overall goal of accelerating the development of reactive services and which, in our opinion, contribute to innovation in the area of service engineering:

1. Services as Compositions of Collaborative Building Blocks

Due to the external description by ESMs, the internals of the building blocks do not have to be considered when they are instantiated and composed. The formal semantics of the building blocks in TLA and cTLA, the chosen composition principle of joint actions and the property of superposition ensure, that the behavior of a building block is preserved in the composed system. Despite this formal setting of the semantics, the user only works on UML models in a rather intuitive way, since we managed to map the collaborations as well as the joint action composition to the visual representation in UML activities. Because the building blocks stay structurally intact during composition, the resulting specification can also later be understood as a combination of existing and already known elements.

2. Automated Transition from Collaborations to Components

Developers specify systems entirely using collaborative building blocks. This enables a focus on the functional decomposition of the system's services and present collaborative behavior in a self-contained, compact form crossing component borders. The state machines necessary for the efficient implementations are consistent with the activities by construction, due to the automated transformation. Because of the formal refinement mapping underlying the transformation, the behavior implied by the activities of the collaboration-oriented perspective is maintained by the behavior implemented by means of the state machines.

This means in particular that a manual and time-consuming synthesis of state machines is not necessary. Obviously, for this to be possible, the activities have to be constructed first. We claim, however, that this is easier than the construction of consistent state machines since activities represent their states only indirectly by token markings and because of the joint action composition we apply. It is therefore more convenient to compose behavior that is executed in parallel and synchronized at some points, as it is often necessary in service compositions.

3. Incremental Model Checking and Automated Analysis

We demonstrated how model checking can be applied incrementally to collaboration-oriented specifications. Due to the semantics we described in temporal logic, the composition mechanism of joint actions and the principle of superposition, the resulting system is correct as well.

The input for the model checking is produced automatically from the UML activities, alongside theorems that describe generally desirable behavior. We discussed how the results of model checking can be presented to the users in a meaningful way, and how error traces can be visualized by animations within the UML activities. These principles in combination with the proposed tools should make formal reasoning effectively usable to practitioners.

6.2 Answers to the Research Questions

To review our results, we recapitulate the research question posed in Sect. 2.3.

1. *What should be the basis for an expressive approach that lets us rapidly compose services from reusable elements?*

   The main idea was to use collaborations as operational, complete and self-contained building blocks, as presented in Paper 1 and demonstrated on a larger example in Paper 5. Thinking in terms of collaborations facilitates the understanding of specifications, since they can be decomposed according to their functions, which enables a horizontal view as illustrated earlier in Fig. 2.3 on page 18. This means that services can be shown as compositions of sub-services, and that their behavior can be presented in a rather self-contained form. This is in contrast to component-oriented descriptions, where service behavior is presented implicitly by the combination of several component descriptions. In our approach, service behavior may be understood in isolation first, then composed with other service behaviors to realize more complex services.

To realize this idea, we had to find a suitable formalism for the building blocks that we could use to define their semantics and how they may be composed. For this purpose, we chose temporal logic, as it enables the composition principle of joint actions as well as the property of superposition. The users of the approach, however, just work on a combination of UML collaborations and activities which are interpreted by tools according to their formal semantics.

2.1 *How can we bridge the gap between collaboration-oriented, horizontal models focusing on functional decomposition of a system into services, and vertical models decomposed into components?* and

2.2 *Can state machines and components be synthesized completely from collaboration-oriented specifications?*

With the presentation of the synthesis algorithm in Paper 3 and the extension in Paper 4 to handle multiple sessions we showed that an automated transformation from the activities to executable state machines is possible. The implementation of the algorithm demonstrated the practicality of this approach. As an effect, developers only need to consider the collaborations with their behavior expressed by activities. The component-oriented models containing the state machines for the execution are synthesized automatically, which makes them correct by construction.

3.1 *How can collaborative elements be expressed separately and in a self-contained, complete way?* and

3.2 *How can collaborative elements be composed, so that detailed dependencies between sub-functionalities can be expressed precisely?*

We express collaborations as special cTLA processes (see Paper 6) that describe their behavior in a complete way, covering both the local behavior of the participants as well as the necessary interactions. By specifying the externally visible behavior by extra state machines (ESMs), building blocks can be understood and analyzed in isolation.

For the composition we use the principle of joint action composition supported by cTLA processes as the semantic foundation of collaborations. This is an elementary composition mechanism that may describe all necessary kind of dependencies and synchronizations between collaborations. It enables synchronous coupling, which is needed if actions of two collaborations should be joint and implemented by the same state machine transition. This composition principle enables also buffered couplings by the introduction of dedicated coupling processes or variables.

4. *Which (graphical) notation supports a flexible, precise composition that preserves the integrity of building blocks?*

   As a graphical representation of the formal description of collaborations we have chosen UML activities as presented in Paper 1. For that purpose, we described their formal semantics in a rule-based manner in Paper 6. As illustrated by various examples, they enable expressive compositions of behaviors. We also pointed out that multiple instances of collaboration may be coordinated and composed using an extension developed in Paper 4.

   Some activities may seem complex at first glance. We must emphasize, however, that they specify a complete solution of a problem and not only a scenario, so that a system's behavior implementation may be derived from them. For example, the internals of the mixed initiative building block (as introduced in Paper 3 and detailed in App. D) may look overwhelming at first. A solution in state machines, however, is also difficult to understand, as more than one side has to be considered in order to understand the complete behavior. Using activities has the advantage that the entire solution can be encapsulated in one collaborative building block that may be abstracted by a much less complex ESM. This building block can from then on be reused without ever looking at its internals again. In a state-machine-based solution, one always has to look at the details when state machines are created.

5. *Can the individual reusable elements be analyzed separately?*

   Our understanding of collaborations as special cTLA processes defined in Paper 6 makes it possible to analyze them for behavioral properties using logical reasoning and model checking as shown in Paper 7. Concerning the analysis by model checking, we only have to check one collaboration at a time and verify that its internal behavior (described by the activity) is a refinement of its external behavior (described by its ESM). When model checking composite collaborations, only the ESMs of the directly included collaborations have to be taken into account, not the entire system or the entire hierarchy of collaborations. This reduces the state space during the analysis considerably.

6. *How can the correctness of a composed and eventually implemented system be ensured?*

   In Paper 6 we demonstrate how the collaborations are composed to systems, using the process composition mechanisms of cTLA. The superposition principle ensures that properties of a collaboration are present in the composed system. The correctness of the collaborations can be ensured by model checking as exemplified in Paper 7.

   To ensure that the implementation based on components and state machines obey these properties as well, we formalized their execution semantics in Paper 2. In Paper 3, we present how to go from collaborations to components via an automated transformation from activities to state machines. In addition, we showed that this transformation is correctness-preserving, as the state machines (formalized in cTLA/e) imply the behavior of the activities (described by cTLA/c) so that cTLA/e $\Rightarrow$ cTLA/c holds. In App. B we present such a refinement proof. Moreover, the implementation of the model transformation was tested by transforming many example specifications into state machine and ensuring by a manual inspection that their behavior is correct. For that we used our experience in the design of state machines and ensured that the state machines synthesized by the transformation were equivalent to state machines we would have written manually, for example by following the guidelines in [BH93].

7. *How can the threshold to applying formal analysis be kept low for a practitioner?*

   As described, an engineer applying the approach to build reactive systems does not have to think in terms of formulas; the main specification language is UML activities which can be interpreted and understood by token flow semantics. The main benefits of formal reasoning, namely the compositional features, the possibility of ensuring certain properties and the refinement to an executable implementation, are facilitated by the tools described in Sect. 4.5, Paper 8 and [Slå07, Slå08]. Due to the semantics provided in Paper 2 and 6, each specification can be presented as UML diagrams and a temporal formula. While the diagram may be easier accessible for the practitioner, the temporal formula allows for a thorough analysis, as shown in Paper 7. Here, we could show how model checking can be used in a rather automated way, so that the formal specification as well as the theorems to be proved are generated automatically, and where results of the analysis can be presented to the user in the context of the original activity.

6.3 Practical Applicability of the Results

Our experiences from the application of the approach are indeed very positive. The chosen specification style allows us to focus on the properties of systems we are most interested in, namely the coordination of collaborative behavior consisting of the synchronization of tasks among components. The token flow semantics of activities seem to be easily understandable when the token flows are animated, as shown in Sect. 4.5 and Paper 7. We used this technique in the presentation of our work at conferences, and got very positive feedback from the audience, as people were impressed how easily they could understand the specifications. This is especially encouraging as we presented the complete specifications (from which executable state machines can be generated) and had, as usual at conferences, only little time.

With the tool support we were able to realize, we proved that the proposed concepts are implementable and that the proposed techniques work also in a practical setting. Beyond our own tools, the approach can also be used by any

UML compliant tool supporting UML profiles to edit service specifications. As the analysis and transformation tools fit within the framework of MDA and are compatible with UML 2.0, it should be possible to integrate them into other tools as well.

The examples we did as case studies and used in the papers were chosen so that they cover most of the situations needed in real systems, yet are compact enough to be presented completely within the space constraints of an article. We claim, however, that the approach scales well and also can handle specimen of real system specifications. In fact, as we argue in Paper 7, we do not expect collaborations to get considerably more complex than the ones we presented. We rather expect that there will be additional levels of decomposition. This does not obstruct the formal analysis, as it has to consider only one decomposition level at a time. For the transformation, we have described strategies (see Paper 3 and Sect. 4.4) to handle complexity as well.

We are currently working on further examples within the applied research project Infrastructure for Integrated Services (ISIS, [ISI07]) for the domain of home automation. In addition, a master's thesis is currently under development that evaluates the approach based on a larger example of a reactive system.

6.4 Related Approaches

Work related to the individual contributions are discussed in the respective publications. In the following, we focus on related approaches with a comparable overall intent.

Collaborations as reusable specification units in DisCo are treated by Kellomäkki and Mikkonen [KM00]. They describe templates to capture patterns involving several objects in so-called *archived steps*. These are refinement steps consisting of a context specification and a solution specification, where the latter is a refinement of the former one. Due to the superposition principle of DisCo specifications as explained in Sect. 3.3.2, properties once verified for the solution layer of an archived step are obeyed also by the application using an instance of the pattern. This means that the verification and design effort are reusable. In comparison with our building blocks, patterns are integrated into the specification by composing DisCo layers, which means to extend certain actions. While this opens for some more flexibility, users work directly on DisCo actions instead of the graphical representation of activities as in our case.

Another approach utilizing the notion of patterns is that of the SDL pattern approach by Geppert et al. [GR01]. In this approach, a pattern is described by a set of separate SDL process fragments which are added to the SDL processes under construction. This task is supported by the SDL pattern tool (SPT, [DEG04]) which is integrated into the Telelogic Tau SDL tool. After the application of a pattern, the system specification may have to be further refined to be executable [GR01]. Validation is done on the resulting SDL model using standard tools for simulation or state space analysis. This means that validation does not directly draw benefits from the definition of patterns. Moreover, once patterns are integrated, they are often tightly interwoven with each other, and not separated in form of building blocks. When the design is maintained or enhanced later, this implies the challenges we problematized in Sect. 2.2.

Rößler et al. developed with CoSDL [RGG01] an experimental language to address collaborative behavior in a more explicit form. In [RGG02], a three-step process is proposed, in which individual collaborations are first developed in isolation. In a second step, collaborations are analyzed manually for their dependencies concerning certain control states and data, and assigned to system components that should realize them. Finally, the collaborations are implemented manually following some guidelines. Similar to the pattern approach, validation is done once the system is completely specified by SDL processes, using standard tools. To build systems from collaborations, [RGG01] describes operations for the sequential, parallel and exclusive composition of collaborations. In our experience, however, collaborations often execute in parallel, with some synchronization every now and then, for example to exchange some data. If such a composition is not supported, collaborations have to be partitioned into parts that can be composed with the available composition operators, which may lead to collaborations that are too small to be effectively reused.

To facilitate reuse in the design of communication systems, Gotzhein et al. propose an approach based on *micro protocols* [GKS02], which are characterized as *"ready-to-use, self-contained, distributed components"* [FG07]. A micro-protocol is specified by an SDL package containing a state machine description for each protocol entity, expressed as separate composite states in SDL 2000 or processes in SDL-96 [GKS02]. The tool described in [FG07] documents a micro protocol by its incoming and outgoing signals as a set of scenarios. An example is presented in [FGG$^+$05], in which an existing system is extended with a safety mechanism that monitors the activity of a control unit and triggers a fail-safe behavior if the unit does not respond. To do so, a watchdog micro protocol is added on the observer side, and a heartbeat protocol is added to the controller side. To connect the protocol with the existing system, a virtual transition within the watchdog has to be refined, which means that the engineer needs to consider some of the internals of the micro protocol. The significant difference from our work is the fact that micro protocols are reusable units on the level of distributed state machines, while our reuse happens on the level of activities, which are transformed in a subsequent step into state machines. Concerning the example in [FGG$^+$05], we would therefore add only one collaboration (expressed as activity) that encapsulates the watchdog and heartbeat functionality within one building block.

In [CB06a], Castejón and Bræk use *goal sequence diagrams* [CB06b] to describe the intended order of execution of a set of collaboration uses. The behavior of collaborations is described by UML sequence diagrams, focusing on the interactions between participants in terms of signal transmissions. In contrast to our representation of collaboration uses by single call behavior actions in activities, goal sequence diagrams display different phases or states of a collaboration as distinct elements, and connections between these elements denote orderings between collaborations. These orderings define desired scenarios, from a global point of view, which may be convenient in early design phases. Since this may imply undesired scenarios in an implementation, goal sequence diagrams are analyzed for such situations in [CvBB07].

The SIMS project [SIM07] also uses UML collaborations for the development of services, based on the work on behavioral projections and interface compatibility by Floch [Flo03] and on progress labels by Sanders [San07]. In SIMS, elementary collaborations, so-called *semantic interfaces*, consist of two communication peers. Each peer has a description of its observable behavior attached. Components bound via semantic interfaces interact correctly if both obey the interface descriptions. In SIMS, composite collaborations describe how a group of components may interact using elementary collaborations to achieve some desirable outcome, named *goal*. So-called goal sequence diagrams identify possible strategies of the composite collaboration to achieve a goal [CFS08]. While the definition of compatible interfaces as well as the implied goal dependencies are helpful when constructing the service components, the SIMS approach currently implies a manual synthesis of state machines to integrate all semantic interfaces. As we argued in Sect. 2.2, this task can be quite difficult and time-consuming. In particular, when semantic interfaces are combined in a new manner, a new state machine has to be constructed. With respect to verification, SIMS is focusing on interface behavior rather than overall correctness of a service. The current SIMS method is somewhat complementary to our approach, as SIMS focuses on the case that parts of a system are only known by their interface. There may be several reasons for such a situation, for example that different parts belong to different operators that do not want to reveal implementation details, or that legacy components are used from which we only know the external behavior. In our work, collaborations typically also cover the local behavior of all participants. In future work, these paradigms could be aligned, as we will outline in Chapt. 7.

CHAPTER
SEVEN

FUTURE WORK

Full Support for Data We are currently working on a full support for data flows. While we already treated data formally in Paper 6, it has to be supported by the tools as well, in particular the transformation algorithm. For this, we will allow UML object nodes and object flows in activities according to [Obj07b]. As UML lacks a concrete action language, we employ Java methods that are linked to the UML model to express the contents of call operation actions. Changes to the transformation algorithm will not be fundamental since the state machine transitions are constructed in the same manner as before. The necessary additions affect only the way effects of transitions and Java code are written.

Related to the handling of data is also the possibility to parametrize building blocks. The mixed initiative building block, for instance, currently only handles control flows. In some cases, we may want to transport data within the initiatives, which typically is specific for a certain application. Instead of having extra building blocks for each possible type, we may employ generic UML templates [Obj07b], that allow the specification of concrete data types once a template is applied. In Arctis, such a parameterization should be implemented as part of the instantiation process of building blocks.

Further Analysis of Building Blocks and Semantics We have described a framework to analyze the behavior of building blocks using model checking (see Sect. 4.3.3 and the work of Slåtten [Slå07, Slå08]). This analysis will be expanded by additional theorems and interpretations of error situations. This includes further studies about semantics for activities, depending on mechanisms from the underlying execution mechanisms. We assume for instance, that different activity flows can overtake each other, as presented in Paper 7. This is useful if different flows should be implemented using different channels, due to security aspects, for example. In some cases, however, it is reasonable to assume that all signals between a pair of components (resp. activity partitions) are sent by the same communication medium and that their order is preserved. Since this can simplify some designs, such options should be made available to engineers.

Similarly, an underlying platform could offer some functionality to terminate sessions between components or to identify obsolete signals, which could be utilized for the termination of collaborations.

Reasoning about Real-time Properties For some domains, reasoning about real-time properties may be desirable. As this has been solved already for TLA (see [Lam05]) and cTLA (see [GHK00]), it is very likely that the introduced solutions can also be used within SPACE, with appropriate additions to the modeling elements.

Advanced Coupling Operators To further facilitate the constraint-oriented specification style [VSvS88] as described in Sect.3.4.2, additional coupling operators may be introduced. For example, decisions in one collaboration may be directly coupled to decisions in other collaborations. This is especially useful in the security domain, as exemplified for pure cTLA in [Her06].

Replacement of Collaborations In the current version of the approach, one building block has exactly one ESM for the external behavior and one activity specifying the internal implementation of the collaboration. In some cases, it may be desirable to realize the same ESM behavior by different activities, possibly with a different number of participants. For example, an authorization could be performed locally, but also using a third party. While such a replacement simply implies different refinement mappings between an ESM and an activity, such flexibility must be integrated also technically into the approach. Moreover, it should be studied how only the local, internal behavior of some of the participants in a collaboration may be exchanged.

Using Existing Components via Behavioral Interfaces In some cases, it may be desirable to only specify and implement certain parts of a system, while other parts are only described by interfaces, for example the ones in [Flo03, San07]. In our method, interfaces between components are internal to collaborative building blocks. To incorporate such interfaces into our method, a two-step approach seems to be feasible, as we briefly hinted at in Paper 1: In a first step, a building block with two participants is constructed. One participant is part of the environment, the other one is part of the system under construction. The behavior of the building block is constructed so that the behavior observed between system and environment is compatible with the original interface description. In a second step, the resulting building block can be composed in the well-known form with the activities so that the behavior towards this component may be integrated into the overall service specification.

Run-Time Configurations Related to the previous point is the dynamic configuration of a system at run-time. Services (or collaborations in general) can be seen as units not only during design-time, but also at run-time. In some systems, it may be desirable to deploy new functionality at run-time, by exchanging existing ones or making new ones available via some discovery mechanism. This form of dynamism is currently not directly addressed by our approach. Given a replacement functionality, a service specification could simply refer to an abstract external description, while the internal behavior is chosen at run-time. This would require some semantics for management and possibilities to reason about the current configuration. Criteria for the selection of collaborations may be taken for example from the work of Csorba et al. [CHH08], which describes a cost analysis based on the collaborative specifications employed in SPACE using the Cross Entropy Ant System (CEAS, [HW01]).

Additional Specifications on Higher Levels of Idealization The service specifications based on activities are complete and formal descriptions. In early development phases it may be desirable to sketch some behavior of the system in form of scenarios, meaning possibly partial or informal behavior. Although such incomplete descriptions are not sufficient to create a complete implementation, there may be a systematic way to refine them until a valid SPACE specification is obtained which can be analyzed and implemented as we described.

Bibliography

[AE03] Daniel Amyot and Armin Eberlein. An Evaluation of Scenario Notations and Construction Approaches for Telecommunication Systems Development. *Telecommunication Systems*, 24(1):61–94, 2003.

[AKP01] Timo Aaltonen, Mika Katara, and Risto Pitkänen. DisCo Toolset – The New Generation. *Journal of Universal Computer Science*, 7(1):3–18, January 2001.

[AL91] Martín Abadi and Leslie Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[AL95] Martín Abadi and Leslie Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–535, May 1995.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[BC87] Kent Beck and Ward Cunningham. Using Pattern Languages for Object Oriented Programs. Technical Report CR-87-43, Tektronix Inc., Presented at the OOPSLA'87 Workshop on Specification and Design for Object-Oriented Programming, September 1987.

[BC89] Kent Beck and Ward Cunningham. A Laboratory for Teaching Object-Oriented Thinking. *ACM SIGPLAN Notices*, 24(10):1–6, 1989.

[BC96] Ray J. A. Buhr and Ron S. Casselman. *Use Case Maps for Object-Oriented Systems*. Prentice-Hall, Inc., 1996.

[BD02] Fulvio Babich and Lia Deotto. Formal Methods for Specification and Analysis of Communication Protocols. *IEEE Communications Surveys and Tutorials*, 4(1), 2002.

[BF04] Rolv Bræk and Jacqueline Floch. ICT Convergence: Modeling Issues. In Daniel Amyot and Alan W. Williams, editors, *SAM'04 - Fourth SDL and MSC Workshop*, volume 3319 of *Lecture Notes in Computer Science*, pages 237–256. Springer, 2004.

[BGH$^+$97] Rolv Bræk, Joe Gorman, Øystein Haugen, Geir Melby, Birger Møller-Pedersen, and Richard Sanders. Quality by Construction Exemplified by TIMe — The Integrated Methodology. *Telektronikk*, 95(1):73–82, 1997.

[BH93] Rolv Bræk and Øystein Haugen. *Engineering Real Time Systems: An Object-Oriented Methodology Using SDL*. The BCS Practitioner Series. Prentice Hall, 1993.

[BHM02] Rolv Bræk, Knut Eilif Husa, and Geir Melby. *ServiceFrame Whitepaper*. Ericsson NorARC, Asker, Norway, April 2002.

[BHS81] R. Bræk, O. Helle, and F. Sandvik. SOM — A SDL Compatible Specification and Design Methodology. In *4th International Conference on Software Engineering for Telecommunication Switching Systems, Conventry*, volume 198, pages 111–117, July 1981.

[BK98] Manfred Broy and Ingolf Krüger. Interaction Interfaces - Towards a Scientific Foundation of a Methodological usage of Message Sequence Charts. In *ICFEM '98: Proceedings of the Second IEEE International Conference on Formal Engineering Methods*, page 2, Washington, DC, USA, 1998. IEEE Computer Society.

[BKS83] R. J. R. Back and R. Kurki-Suonio. Decentralization of Process Nets with Centralized Control. In *PODC '83: Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 131–142, New York, NY, USA, 1983. ACM.

[BKS88] R. J. R. Back and Reino Kurki-Suonio. Distributed Cooperation With Action Systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, 1988.

[BM05] Rolv Bræk and Geir Melby. Model-Driven Service Engineering. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, *Model-Driven Software Development*, pages 385–401. Springer, 2005.

[BMP95] Rolv Bræk and Birger Møller-Pedersen. Common Methodology. Technical Report 1112-6, SISU II Project, 1995.

[Bol00] Tommaso Bolognesi. Toward Constraint-Object-Oriented Development. *IEEE Trans. Softw. Eng.*, 26(7):594–616, 2000.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[Bræ79] Rolv Bræk. Unified System Modelling and Implementation. In *International Switching Symposium*, pages 1180–1187, Paris, France, May 1979.

[Bræ00] Rolv Bræk. On Methodology Using the ITU-T Languages and UML. *Telektronikk*, 4:96—106, 2000.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Springer, 2001.

[CB06a] Humberto N. Castejón and Rolv Bræk. A Collaboration-based Approach to Service Specification and Detection of Implied Scenarios. In *SCESM '06: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pages 37–43, New York, NY, USA, 2006. ACM Press.

[CB06b] Humberto N. Castejón and Rolv Bræk. Formalizing Collaboration Goal Sequences for Service Choreography. In Elie Najm and Jean-François Pradat-Peyre, editors, *26th IFIP WG 6.1 Intl. Conf. on Formal Methods for Networked and Distributed Systems (FORTE'06)*, volume 4229 of *Lecture Notes in Computer Science*. Springer, September 2006.

[CFN03] Cyril Carrez, Alessandro Fantechi, and Elie Najm. Behavioural Contracts for a Sound Assembly of Components. In *23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2003)*, pages 111–126, 2003.

[CFS08] Cyril Carrez, Jacqueline Floch, and Richard Sanders. Describing Component Collaboration using Goal Sequences. In René Meier and Sotirios Terzis, editors, *Distributed Applications and Interoperable Systems - Proceedings of DAIS 2008, Oslo, Norway*, volume 5053 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2008.

[CHH08] Máté J. Csorba, Poul E. Heegaard, and Peter Herrmann. Cost-Efficient Deployment of Collaborating Components. In René Meier and Sotirios Terzis, editors, *Distributed Applications and Interoperable Systems - Proceedings of DAIS 2008, Oslo, Norway*, volume 5053 of *Lecture Notes in Computer Science*, pages 253–268. Springer, 2008.

[Coa92] Peter Coad. Object-Oriented Patterns. *Communications of the ACM*, 35(9):152–156, September 1992.

[CvBB07] Humberto N. Castejón, Gregor von Bochmann, and Rolv Bræk. Realizability of Collaboration-based Service Specifications. In *Proc. of 14th Asia-Pacific Soft. Eng. Conf. (APSEC'07)*. IEEE Computer Society, December 2007.

[DA85] M. Diaz and P. Azema. Petri Net Based Models for the Specification and Validation of Protocols. In *Advances in Petri Nets – Proceedings of the European Workshop on Applications and Theory in Petri Nets*, volume 188 of *Lecture Notes in Computer Science*, pages 101–121, London, UK, 1985. Springer-Verlag.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface Automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, 2001.

[DEG04] Jörg Dorsch, Anders Ek, and Reinhard Gotzhein. SPT - The SDL Pattern Tool. In Daniel Amyot and Alan W. Williams, editors, *System Analysis and Modeling, 4th International SDL and MSC Workshop, SAM 2004, Ottawa, Canada, June 1-4, 2004, Revised Selected Papers*, volume 3319 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2004.

[DKMR05] Martin Deubler, Ingolf Krüger, Michael Meisinger, and Sabine Rittmann. Modeling Crosscutting Services with UML Sequence Diagrams. In Lionel C. Briand and Clay Williams, editors, *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *Lecture Notes in Computer Science*, pages 522–536. Springer, 2005.

[DW99] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML: the Catalysis Approach.* Addison-Wesley, 1999.

[Ecl] Eclipse Modeling Project. http://www.eclipse.org/modeling.

[FG07] Ingmar Fliege and Reinhard Gotzhein. Automated Generation of Micro Protocol Descriptions from SDL Design Specifications. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL Forum*, volume 4745 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2007.

[FGG$^+$05] Ingmar Fliege, Alexander Geraldy, Reinhard Gotzhein, Thomas Kuhn, and Christian Webel. Developing Safety-Critical Real-Time Systems with SDL Design Patterns and Components. *Computer Networks*, 49(5):689–706, 2005.

[FK01] Kathi Fisler and Shriram Krishnamurthi. Modular Verification of Collaboration-Based Software Designs. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 152–163, New York, NY, USA, 2001. ACM Press.

[Flo03] Jacqueline Floch. *Towards Plug-and-Play Services: Design and Validation using Roles.* PhD thesis, Norwegian University of Science and Technology, 2003.

[GH04] Günter Graw and Peter Herrmann. Transformation and Verification of Executable UML Models. *Electronic Notes on Theoretical Computer Science*, 101:3–24, 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GHK00] Günter Graw, Peter Herrmann, and Heiko Krumm. Verification of UML-Based Real-Time System Designs by means of cTLA. In *Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC2K)*, pages 86–95, Newport Beach, 2000. IEEE Computer Society Press.

[Gis06] Øystein Gisnås. A Constructive Approach to Support the Design of State Machines. Master's thesis, Norwegian University of Science and Technology, June 2006.

[GKS02] Reinhard Gotzhein, Ferhat Khendek, and Philipp Schaible. Micro Protocol Design: The SNMP Case Study. In Edel Sherratt, editor, *Telecommunications and beyond: The Broader Applicability of SDL and MSC. Third International Workshop on SDL and MSC (SAM 2002), Revised Papers*, volume 2599 of *Lecture Notes in Computer Science*, pages 61–73. Springer, 2002.

[GR01] Birgit Geppert and Frank Rößler. The SDL Pattern Approach — A Reuse-Driven SDL Design Methodology. *Computer Networks*, 35(6):627–645, 2001.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[Her97] Peter Herrmann. *Problemnaher korrektheitssichernder Entwurf von Hochleistungsprotokollen.* PhD thesis, Universität Dortmund, 1997.

[Her06] Peter Herrmann. Temporal Logic-Based Specification and Verification of Trust Models. In Ketil Stølen, William H. Winsborough, Fabio Martinelli, and Fabio Massacci, editors, *iTrust 2006*, volume 3986 of *Lecture Notes in Computer Science*, pages 105–119, Heidelberg, 2006. Springer–Verlag.

[HHRS05] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. STAIRS Towards Formal Design with Sequence Diagrams. *Journal of Software and Systems Modeling*, 4:355–367, 2005.

[HK98] Peter Herrmann and Heiko Krumm. Modular Specification and Verification of XTP. *Telecommunication Systems*, 9(2):207–221, 1998.

[HK00] Peter Herrmann and Heiko Krumm. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337, 2000.

[HK07] Peter Herrmann and Frank Alexander Kraemer. Design of Trusted Systems with Reusable Collaboration Models. In Sandro Etalle and Stephen Marsh, editors, *Trust Management*, volume 238, pages 317–332. IFIP International Federation for Information Processing, Springer, 2007.

[HW01] Bjarne E. Helvik and Otto Wittner. Using the Cross-Entropy Method to Guide/Govern Mobile Agent's Path Finding in Networks. In *MATA '01: Proceedings of the Third International Workshop on Mobile Agents for Telecommunication Applications*, pages 255–268, London, UK, 2001. Springer-Verlag.

[ISI07] ISIS Project Website. http://www.isisproject.org/, 2007.

[ITU97a] ITU-T. *Recommondation X.901: Open Distributed Processing — Reference Model: Overview*, August 1997.

[ITU97b] ITU-T. *SDL+ Methodology: Manual for the use of MSC and SDL (with ASN.1). Supplement 1 to Z.100*, 1997.

[ITU98] ITU-T. *Recommondation Q.931: ISDN User-Network Interface Layer 3 Specification for Basic Call Control*, May 1998.

[ITU99] ITU-T. *Recommendation Z.200: CHILL - The ITU-T Programming Language*, 1999.

[ITU02] ITU-T. *Recommendation Z.100: Specification and Description Language (SDL)*, August 2002.

[ITU03] ITU-T. *Z.100: Appendices I and II*, March 2003.

[ITU04] ITU-T. *Recommendation Z.120: Message Sequence Charts (MSC)*, 2004.

[ITU07] ITU-T. *Recommondation Z.109: SDL-2000 combined with UML*, June 2007.

[Jär92] Hannu-Matti Järvinen. *The Design of a Specification Language for Reactive Systems.* PhD thesis, Tampere University of Technology, 1992.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[JCJÖ92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[JKSSS90] H.-M. Järvinen, Reino Kurki-Suonio, M. Sakkinen, and K. Systä. Object-Oriented Specification of Reactive Systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71. IEEE Computer Society Press, 1990.

[KBH07] Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007*, volume 4745 of *Lecture Notes in Computer Science*, pages 166–185. Springer–Verlag Berlin Heidelberg, September 2007.

[Kel97] Pertti Kellomäki. *Mechanical Verification of Invariant Properties of DisCo Specifications.* PhD thesis, Tampere University of Technology, 1997.

[KGW06] Thomas Kuhn, Reinhard Gotzhein, and Christian Webel. Model-Driven Development with SDL - Process, Tools, and Experiences. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 83–97. Springer, 2006.

[KH06] Frank Alexander Kraemer and Peter Herrmann. Service Specification by Composition of Collaborations — An Example. In *Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology)*, pages 129–133. IEEE Computer Society, 2006. 2nd International Workshop on Service Composition (Sercomp), Hong Kong.

[KH07a] Frank Alexander Kraemer and Peter Herrmann. Formalizing Collaboration-Oriented Service Specifications using Temporal Logic. In *Networking and Electronic Commerce Research Conference 2007 (NAEC 2007)*, pages 194–220, USA, October 2007. ATSMA Inc.

[KH07b] Frank Alexander Kraemer and Peter Herrmann. Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In Karsten Ehring and Holger Giese, editors, *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, volume 7 of *Electronic Communications of the EASST*. EASST, 2007.

[KHB06] Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk. Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. In R. Meersmann and Z. Tari, editors, *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), 2006, Montpellier, France*, volume 4276 of *Lecture Notes in Computer Science*, pages 1613–1632. Springer–Verlag Heidelberg, 2006.

[KK03] Per Kroll and Philippe Kruchten. *The Rational Unified Process Made Easy.* Addison-Wesley, 2003.

[KKSM04] Mika Katara, Reino Kurki-Suonio, and Tommi Mikkonen. On the Horizontal Dimension of Software Architecture in Formal Specifications of Reactive Systems. In Iowa State University Department of Computer Science, editor, *FOAL 2004 Proceedings, Foundations of Aspect-Oriented Languages Workshop at AOSD 2004*, pages 37–43, 2004.

[KM00] Pertti Kellomäki and Tommi Mikkonen. Design Templates for Collective Behavior. In Elisa Bertino, editor, *Proceedings of ECOOP 2000, 14th European Conference on Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 277–295. Springer–Verlag, 2000.

[KM03] Ingolf Krüger and Reena Mathew. Component Synthesis from Service Specifications. In *Scenarios: Models, Transformations and Tools*, volume 3466 of *Lecture Notes in Computer Science*, pages 255–277. Springer, 2003.

[Kra03] Frank Alexander Kraemer. Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart, 2003.

[Kra07] Frank Alexander Kraemer. Arctis and Ramses: Tool Suites for Rapid Service Engineering. In *Proceedings of NIK 2007 (Norsk informatikkonferanse), Oslo, Norway.* Tapir Akademisk Forlag, November 2007.

[KS05] Reino Kurki-Suonio. *A Practical Theory of Reactive Systems.* Springer, 2005.

[KSH07] Frank Alexander Kraemer, Vidar Slåtten, and Peter Herrmann. Engineering Support for UML Activities by Automated Model-Checking — An Example. In *Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE)*, November 2007.

[KSK88] Reino Kurki-Suonio and T. Kankaanpää. On the Design of Reactive Systems. *BIT*, (28):581–604, 1988.

[KSM98] Reino Kurki-Suonio and Tommi Mikkonen. Abstractions of Distributed Cooperation, their Refinement and Implementation. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 94–102. IEEE Computer Society, April 1998.

[Lam89] Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam02] Leslie Lamport. *Specifying Systems*. Addison-Wesley, 2002.

[Lam05] Leslie Lamport. Real-Time Model Checking Is Really Simple. In Dominique Borrione and Wolfgang J. Paul, editors, *Correct Hardware Design and Verification Methods, (CHARME 2005)*, volume 3725 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2005.

[LDD06] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A Comparative Survey of Scenario-Based to State-Based Model Synthesis Approaches. In *SCESM '06: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pages 5–12, New York, NY, USA, 2006. ACM Press.

[LSR87] M. E. S. Loomis, A. V. Shah, and James Rumbaugh. An Object Modeling Technique for Conceptual Design. In G. Goos and J. Hartmanis, editors, *ECOOP '87 – European Conference on Object-Oriented Programming: Paris, France, June 1987. Proceedings*, volume 276 of *Lecture Notes in Computer Science*, pages 192–202. Springer, 1987.

[MB02] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Men04] Vladimir Mencl. Specifying Component Behavior with Port State Machines. *Electronic Notes in Theoretical Computer Science*, 101:129–153, 2004.

[Mer76] Philip M. Merlin. A Methodology for the Design and Implementation of Communication Protocols. *IEEE Transactions on Communications*, 25(6):614–621, June 1976.

[Mer79] Philip M. Merlin. Specification and Validation of Protocols. *IEEE Transactions on Communications*, 27(11):1671–1680, November 1979.

[Mik99] Tommi Mikkonen. The two Dimensions of an Architecture. In *WICSA1, First Working IFIP Conference on Software Architecture*, 1999.

[Obj00] Object Management Group. OMG Unified Modeling Language Specification Version 1.3, March 2000.

[Obj01] Object Management Group. Unified Modeling Language: Superstructure, version 1.4, September 2001. ptc/2006-09-67.

[Obj03] Object Management Group. *MDA Guide Version 1.0.1*, omg/2003-06-01 edition, June 2003.

[Obj05] Object Management Group. *Unified Modeling Language: Superstructure, version 2.0*, July 2005. formal/2005-07-05.

[Obj06] Object Management Group. Meta Object Facility (MOF) Core Specification Version 2.0, January 2006.

[Obj07a] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, July 2007.

[Obj07b] Object Management Group. Unified Modeling Language: Superstructure, version 2.1.2, November 2007. formal/2007-11-01.

[OFMP$^+$94] Anders Olsen, Ove Færgemand, Birger Møller-Pedersen, Rick Reed, and J.R.W. Smith. *Systems Engineering Using SDL-92.* Elsevier North-Holland, Inc., Amsterdam, The Netherlands, 1994.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 748–752, London, UK, 1992. Springer-Verlag.

[Pit06] Risto Pitkänen. *Tools and Techniques for Specification-Driven Software Development.* PhD thesis, Tampere University of Technology, August 2006.

[Pnu86] Amir Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. *Current Trends in Concurrency. Overviews and Tutorials*, pages 510–584, 1986.

[Pos81] J. Postel. RFC 793: Transmission Control Protocol, 1981.

[PS91] Robert L. Probert and Kassem Saleh. Synthesis of Communication Protocols: Survey and Assessment. *IEEE Transactions on Computers*, 40(4):468–476, 1991.

[QSPvS07] Dick A. C. Quartel, Maarten W. A. Steen, Stanislav Pokraev, and Marten van Sinderen. COSMO: A Conceptual Framework for Service Modelling and Refinement. *Information Systems Frontiers*, 9(2-3):225–244, 2007.

[RAB$^+$92] Trygve Reenskaug, Egil P. Andersen, Arne Jorgen Berre, Anne Hurlen, Anton Landmark, Odd Arild Lehne, Else Nordhagen, Eirik Ness-Ulseth, Gro Oftedal, Anne Lise Skaar, and Pål Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *Journal of Object-oriented Programming*, 5(6):27–41, October 1992.

[RBL+91] James Rumbaugh, Michael Blaha, William Lorensen, Frederick Eddy, and William Premerlani. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[Ree96] Rick Reed. Methodology for Real Time Systems. *Comput. Netw. ISDN Syst.*, 28(12):1685–1701, 1996.

[Rek82] Kristen Rekdal. CHILL—The Standard Language for Programming SPC Systems. *IEEE Transactions on Communications*, 30(6):1318–1328, June 1982.

[RFW04] Chris Raistrick, Paul Francis, and John Wright. *Model Driven Architecture with Executable UML.* Cambridge University Press, New York, NY, USA, 2004.

[RGG01] Frank Rößler, Birgit Geppert, and Reinhard Gotzhein. Collaboration-Based Design of SDL Systems. In *Proceedings of the 10th International SDL Forum Copenhagen on Meeting UML*, pages 72–89. Springer-Verlag, 2001.

[RGG02] Frank Rößler, Birgit Geppert, and Reinhard Gotzhein. CoSDL: An Experimental Language for Collaboration Specification. In Edel Sherratt, editor, *Proceedings of the 3rd SAM Workshop*, volume 2599 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2002.

[RJB05] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual, Second Edition.* Addison-Wesley, 2005.

[Rus00] John Rushby. Disappearing Formal Methods. In *High-Assurance Systems Engineering Symposium*, pages 95–96, Albuquerque, NM, November 2000. ACM.

[RWL95] Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working with Objects, The OOram Software Engineering Method.* Prentice Hall, 1995.

[San00] Richard Sanders. Implementing from SDL. *Telektronikk*, 96(4):120–129, 2000.

[San07] Richard Sanders. *Collaborations, Semantic Interfaces and Service Goals: A Way Forward for Service Engineering.* PhD thesis, Norwegian University of Science and Technology, 2007.

[Sel98] Bran Selic. Using UML for Modeling Complex Real-Time Systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, volume 1474 of *Lecture Notes In Computer Science*, pages 250–260. Springer-Verlag, 1998.

[SFR97] M. Saksena, P. Freedman, and P. Rodziewicz. Guidelines for Automated Implementation of Executable Object Oriented Models for Real-Time Embedded Control Systems. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*, pages 240–251, Washington, DC, USA, 1997. IEEE Computer Society.

[SGME92] Bran Selic, Garth Gullekson, Jim McGee, and Ian Engelberg. ROOM: An Object-Oriented Methodology for Developing Real-Time Systems. In *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering (Case'92)*, pages 230–240. IEEE Computer Society, July 1992.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley & Sons, Inc., New York, NY, USA, 1994.

[She05] Edel Sherratt. Model-Driven Development of Reactive Systems with SDL. In Andreas Prinz, Rick Reed, and Jeanne Reed, editors, *Proceedings of the 12th SDL Forum*, volume 3530 of *Lecture Notes in Computer Science*, pages 224–233. Springer, 2005.

[SIM07] SIMS Project Website. http://www.ist-sims.org/, 2007.

[SIN99] SINTEF Telecom and Informatics. TIMe: The Integrated Method Electronic Handbook. Available at http://www.sintef.no/time/, 1999.

[SIS96] SISU II Project. http://www.sintef.no/units/informatics/projects/sisu/, 1996.

[Slå07] Vidar Slåtten. Model Checking Collaborative Service Specifications in TLA with TLC. Project Thesis, August 2007. Norwegian University of Science and Technology, Trondheim, Norway.

[Slå08] Vidar Slåtten. Automatic Detection and Correction of Flaws in Service Specifications. Master's thesis, Norwegian University of Science and Technology, June 2008.

[SM92] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles: Modeling the World in States.* Prentice Hall, 1992.

[Stø04] Alf Kristian Støyle. Service Engineering Environment for AMIGOS. Master's thesis, Norwegian University of Science and Technology, 2004.

[Tel02] Telelogic. *Tau 4.4 User's Manual.* Malmö, 2002.

[vBG86] Gregor von Bochmann and Reinhard Gotzhein. Deriving Protocol Specifications from Service Specifications. In *SIGCOMM '86:*

Proceedings of the ACM SIGCOMM conference on Communications architectures & protocols, pages 148–156, New York, NY, USA, 1986. ACM Press.

[VL85] Chris A. Vissers and Luigi Logrippo. The Importance of the Service Concept in the Design of Data Communications Protocols. In *Proceedings of the IFIP WG6.1 Fifth International Conference on Protocol Specification, Testing and Verification V*, pages 3–17, Amsterdam, The Netherlands, The Netherlands, 1985. North-Holland Publishing Co.

[VSvS88] Chris A. Vissers, Guiseppe Scollo, and Marten van Sinderen. Architecture and Specification Style in Formal Descriptions of Distributed Systems. In Sudhir Aggarwal and Krishan K. Sabnani, editors, *Protocol Specification, Testing and Verification*, volume VIII, pages 189–204, Amsterdam, The Netherlands, 1988. North-Holland.

[VSvSB91] Chris A. Vissers, Guiseppe Scollo, Marten van Sinderen, and Hendrik Brinksma. Specification Styles in Distributed System Design and Verification. *Theoretical Computer Science*, 89:179–206, 1991.

[Wie98] Roel Wieringa. A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. *ACM Computing Surveys*, 30(4):459–527, 1998.

[YEFvBH03] Hirozumi Yamaguchi, Khaled El-Fakih, Gregor von Bochmann, and Teruo Higashino. Protocol Synthesis and Re-Synthesis with Optimal Allocation of Resources based on Extended Petri Nets. *Distrib. Comput.*, 16(1):21–35, 2003.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA$^+$ Specifications. In L. Pierre and T. Kropf, editors, *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer-Verlag, 1999.

Part II

Included Publications

The papers are presented with their originally published content. The layout of some figures has been adjusted to fit the paper format of this print. In some cases, additional notes not part of the original publication are given after the bibliographies.

PAPER

ONE

SERVICE SPECIFICATION BY COMPOSITION OF COLLABORATIONS — AN EXAMPLE

Frank Alexander Kraemer and Peter Herrmann.

Presented at the 2^{nd} International Workshop on Service Composition (SERCOMP), Oct. 31th — Nov. 3rd, 2006 Hong Kong, China.

Published in *Proceedings of the 2006 WI-IAT Workshops* (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology), IEEE Computer Society, 2006.

The original publication is available at www.ieeexplore.ieee.org via
`http://dx.doi.org/10.1109/WI-IATW.2006.121`.

PAPER

TWO

ALIGNING UML 2.0 STATE MACHINES AND TEMPORAL LOGIC FOR THE EFFICIENT EXECUTION OF SERVICES

Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk.

Presented at the 11th International Conference on Distributed Objects and Applications (DOA), October 31 — November 3, Montpellier, France.

Published in *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, Volume 4276 of Lecture Notes of Computer Science, p. 1613-1632, Springer, 2006.

The original publication is available at www.springerlink.com via `http://dx.doi.org/10.1007/11914952_41`.

Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services

Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk

Abstract. In our service engineering approach, services are specified by UML 2.0 collaborations and activities, focusing on the interactions between cooperating entities. To execute services, however, we need precise behavioral descriptions of physical system components modeling how a component contributes to a service. For these descriptions we use the concept of state machines which form a suitable input for our existing code generators that produce efficiently executable programs. From the engineering viewpoint, the gap between the collaborations and the components will be covered by UML model transformations. To ensure the correctness of these transformations, we use the compositional Temporal Logic of Actions (cTLA) which enables us to reason about service specifications and their refinement formally. In this paper, we focus on the execution of services. By outlining an UML profile, we describe which form the descriptions of the components should have to be efficiently executable. To guarantee the correctness of the design process, we further introduce the cTLA specification style cTLA/e which is behaviorally equivalent with the UML 2.0 state machines used as code generator input. In this way, we bridge the gap between UML for modeling and design, cTLA specifications used for reasoning, and the efficient execution of services, so that we can prove important properties formally.

2.1 Introduction

The ongoing convergence of the communication and the computing domain enables a wide range of advanced services, involving a complex mixture of technologies, devices and networks. The development has reached a degree of complexity in which formal reasoning about specifications and corresponding tool support are increasingly important to design services of high quality within acceptable time and cost limits. In consequence, *service engineering* has become a discipline in its own right. In earlier publications we demonstrated the close conceptual relationship between services and collaborations, and the suitability of collaborations as a framework for service specifications [SCKB05, RB06, CB06]. Collaborations model cross-cutting, partial behavior involving several participants. Service specifications consisting of several sub-functionalities may be constructed from collaborations, which can be reused in several services.

While we use the concept of UML 2.0 collaborations [Obj05] to model the structural aspects of collaborations and services, we use UML 2.0 activities to specify their behavior. Figures 2.1 and 2.2 describe a service that retrieves the locations of small devices as part of a group communication service. In Fig. 2.1, we use icons for the collaboration roles and omit the frame of the system collaboration for clarity. Each device is connected to one local node, and all local nodes are connected to one central node. Sensors capture the

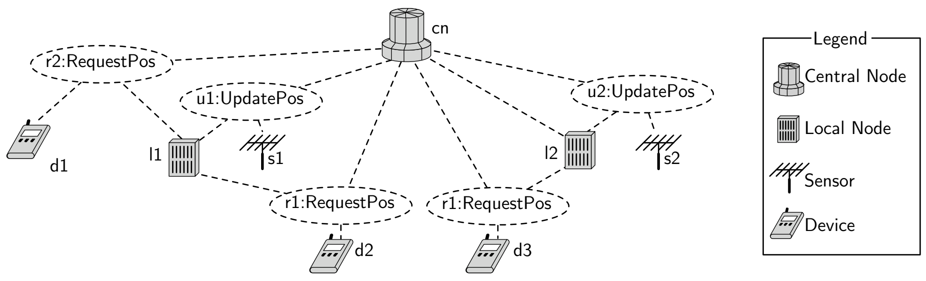


Fig. 2.1: Collaboration for the entire system

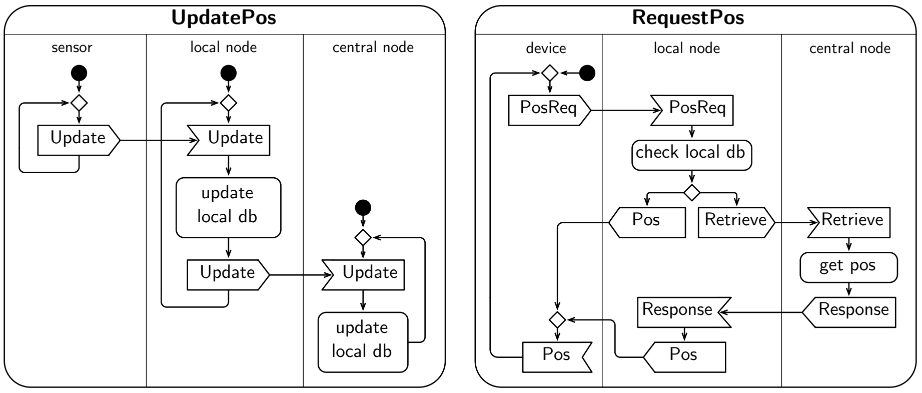


Fig. 2.2: Activities describing collaborations *GetPosition* and *UpdatePosition*

movement of the devices and update the position information in the local nodes. This is specified by the collaboration uses *u1* and *u2* of the collaboration type *UpdatePos*. The behavior of this collaboration type is expressed in detail by the activity on the left side of Fig. 2.2. The sensors send updates of the device positions to their connected local node, which updates the entry in its local table. Thereafter, the local node forwards the update to the central node, which refreshes the central table containing the location of all devices. Furthermore, devices can ask their local node for the position of other devices. This behavior is specified by collaboration *RequestPos* outlined on the right side of Fig. 2.2. Device *d1* can, for instance, ask for the position of *d2* by sending *PosReq* to its local node. As *d2* is registered on the same local node, *l1* has the position of it in its local table and can therefore answer right away. If *d1* asks for *d3*, then *l1* sends a request to the central node and returns the reply to *d1*.

Our experiences using collaborations and activities to specify services are very encouraging, as collaborations allow intuitive but yet precise specifications of services. In order to execute a service, however, a behavioral description for each participating component is needed that can be efficiently executed in form of a program running on available platforms. To accomplish that, we follow the

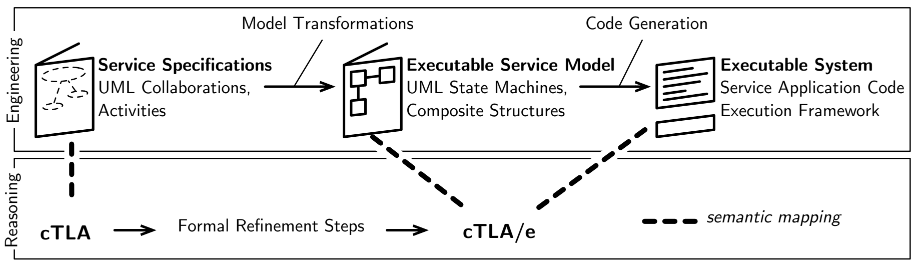


Fig. 2.3: Development approach using UML and cTLA

approach of stepwise refinement, adding more and more details until we get models that can be directly transformed into executable code. We intend to achieve these design steps by a set of model transformations in the spirit of MDA, as shown in the engineering part of Fig. 2.3. The result of the model transformations is an executable service model that is based on UML 2.0 state machines and composite structures. It is the input for our existing code generators that can generate programs executing the services on various Java platforms, appropriate for both the telecommunication as well as for the computing domain (cf. [BF04]).

To ensure the correctness of these transformations, we need a formal reasoning technique. The temporal logic cTLA [HK00a] offers operators and techniques suitable for refinement [MK95], and it can capture such transformation steps in a formal way quite well, as shown in [HK00a, Her03, Her06]. Moreover, the composition of services from collaborations can be directly expressed with the well-understood concept of process composition in cTLA.

Of course, a correctness-preserving development approach is only meaningful if we can guarantee that the generated code corresponds to the executable service model. Thus, we have to clarify formally that the executable code is a correct refinement of the executable service model in spite of the practical limitations of execution frameworks such as finite message buffers. For this sake, we introduce a cTLA specification style cTLA/e. It corresponds directly to the executable service model and describes which form a cTLA specification must have to be efficiently executable on existing service platforms. In this way, we establish a relationship between an intuitive service execution model in UML 2.0, the efficient execution of services on real platforms, and a formal model allowing reasoning and analysis.

In the following, we describe the execution platform in Sect. 2.2 and outline a profile for the executable service model based on UML 2.0 state machines in Sect. 2.3. This execution model is based on the experience of about three decades of system engineering and originates from the SDL-based design methodology SOM [Bræ79], which described the basic modeling and execution mechanisms also used in the projects SISU and SISU II [SIS96]. These projects resulted in the system engineering method TIMe [BGH^{+}97] and had a *"major impact on the SDL methodology guidelines as well as on the SDL and MSC standards"* [MT01, p. 171]. As UML 2.0 adopted most of the language elements for the SDL mechanisms used in TIMe, we use it as a base for our model description. In addition, we sketch the specification technique cTLA in Sect. 6.3 and the specification style cTLA/e in Sect. 2.5. In Sect. 2.6 we outline the conformance of the executable service model with the executable system based on cTLA/e and discuss the properties a cTLA/e specification should have in order to properly address practical software and hardware limits. We close with a reflection about related approaches and some concluding remarks.

2.2 Service Execution Based on State Machines

Systems that execute services fall into the category of reactive systems as characterized by Pnueli [Pnu86]. A service typically requires the coordinated effort of several physically distributed devices [FB00], so that a system delivering a service needs to be decomposed into a number of reactive components running on different execution nodes. To define the behavior of the service components, we use communicating extended finite state machines in the form of UML 2.0 state machines. Similar descriptions are applied in ROOM [SGW94] as well as in the formal description techniques Estelle [Est97] and SDL [ITU02]. We assume that state machines communicate asynchronously using buffered message passing. This enables both asymmetrical client-server interactions typical for the computing domain as well as symmetrical peer-to-peer interactions common in the telecom domain (cf. [BF04]). Buffered communication also helps to decouple the different state machine instances and simplifies distribution, as this mechanism can be implemented for local as well as for remote communication without making changes to the model.

State machines define an executable abstract machine that can be implemented as a virtual machine layer providing runtime support. This gives the benefits of virtual machines in terms of adaptability and portability of applications. Contrary to other virtual machine approaches, the communicating state machines enable a highly efficient solution due to the following reasons:

- Asynchronous message passing avoids blocking on message sending.
- Transition-based execution models enable a very simple scheduling.
- Several state machines can be efficiently integrated in one native process.
- Generic mechanisms for input protection, error handling, and testing can be provided easily as part of the runtime support.

In this section, we outline how state machines can be executed using runtime support systems or execution frameworks. As an example, we present the runtime support system JavaFrame that facilitates execution of state machines on Java. Thereafter, we sketch other execution frameworks built on JavaFrame.

2.2.1 Runtime Support Systems and Execution Frameworks

To achieve a good performance of the executable code, the integration of state machines to native processes (e.g., operating system processes, Java threads) plays a significant role. A naive approach is to execute each state machine instance in a separate native process. This, however, would result in a significant space and time penalty caused by excessive context switching of the operating system. Therefore the common practice is to integrate several state machine instances into a single native process, which is called *light integration* in [MT01]. Scheduling of such state machines is extremely space and time efficient, providing the state machines have equal priority and can be allowed to run each transition to completion. Support for state machines can be integrated into a general virtual machine layer supporting the execution of state machines, the so-called *runtime support system* (RTS). Alongside process management and scheduling, an RTS can offer a range of services to the application layer, such as communication, timer routines, instance creation, logging, debugging and monitoring, as well as mobility management and load control. This approach has been used on numerous performance-critical products by many different companies in the telecommunication and automotive industry. Layered approaches can also be found in the computing domain. Instead of an RTS, one uses execution platforms like J2EE or newer platforms as for example JAIN SLEE [LF04], which try to more directly target the needs of traditional telecommunication services.

2.2.2 The Runtime Support System *JavaFrame*

To illustrate runtime support, we introduce JavaFrame [HMP00], which is an RTS and Java execution framework facilitating the execution of UML 2.0 state machines. It is based on an RTS in C++ presented by Bræk and Haugen in [BH93], which implements an abstract SDL machine. JavaFrame provides a scheduler and base classes for state machines that can be extended with application-specific logic. Mediator objects encapsulate various communication protocols and routing functionality to send signals between state machines. Mediators can also be used to connect the state machines to environments not modeled by state machines.

With the behavior in form of single transitions, state machines naturally offer scheduling units that can be executed individually. Transitions are programmed in JavaFrame using transition methods containing nested if-statements. These if-statements differentiate the available trigger and the current control state and thereby realize the transition table of the state machine. The bodies of these nested statements contain the code that is executed as the effect of the transition. In particular, signals are sent to other state machines, operations are performed on local data or timers, and the next control state is determined. If an incoming signal should not be handled in the current state, it can be deferred by putting it into a dedicated defer queue, which is moved back into the input queue when another transition is executed. This corresponds to the save-concept of SDL.

A scheduler controls a set of state machine instances and dispatches the events in their input queues by executing their transitions with the transition method described above, using a FIFO ordering of queues. Following the *action-oriented* approach [MT01, BH93], the state of each state machine instance is stored explicitly in a data structure. This facilitates an efficient implementation, where the scheduler can manage the states of a large number of state machine instances. To execute a transition, the scheduler retrieves the current state and makes it available to the transition method as a parameter once a transition should be executed. Consequently, the transition method is reentrant, and needs to be provided only once per state machine type.

With this transition method, JavaFrame handles the selection of transitions and their execution in one single method call (as opposed to a more general solution, where transition selection and execution are implemented in separate methods [MT01]). This simplifies the scheduler further and considerably reduces the computation time to find an enabled transition, as the scheduler simply calls the transition method each time an event is available in the input queue. In consequence, transitions are enabled depending on their source state and their trigger only, and must not contain additional enabling conditions. In practice, this is not a real constraint, as transitions still may include decisions.

The scheduler of JavaFrame runs in one Java thread and executes only one transition at a time. In this way, JavaFrame complies to the run-to-completion semantics assumed in modeling languages like SDL, ROOM, and UML. The fact, that the Java thread of the scheduler may be interrupted by another thread, is not problematic, as these do not access any data of the interrupted state machine. The simple structure of the transition method (i.e., with the nested conditional statements) also implies that if several transitions are enabled in the same source state by the same input trigger, only the first one written in the transition method will be executed, while the code of the other transitions is never reachable. Such a situation can easily be avoided by combining these competing transitions to a single one, which contains a choice leading to the different effects of the original transitions.

2.2.3 State Machine Execution on other Platforms

JavaFrame can be used directly to implement state machine-based specifications on the standard Java platform . Due to its simplicity, it may also be seen as a prototype that can guide the implementation of execution frameworks on other platforms. To facilitate the execution of telecommunication services further, a prototypical service execution framework [BHM02] was devised by Ericsson extending the basic execution mechanisms with concepts targeted to service engineering and deployment. Specifically for the domain of telecommunication services, ServiceFrame contains a part defining service components like *user agents* and *terminal agents*. In addition, a number of resource adapters were defined to connect the system to existing technologies, interfaces and transport protocols like Parlay-X, SIP, as well as Bluetooth connections and location tracking via GSM or WLAN for mobile devices. The part for the execution of services, called

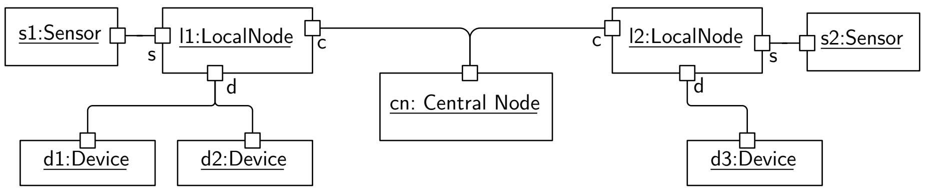


Fig. 2.4: Object diagram of the system

ActorFrame [MH05], is an extension of JavaFrame, adding routing mechanisms, an addressing scheme and protocols for the management of the system structure. The initial version of ActorFrame was implemented on the standard Java platform, J2SE, followed by versions running on the J2EE [Mel03] and later also the platform, which makes it possible to run parts on the system also on mobile devices.

2.2.4 Code Generation

As parts of our integrated service engineering tool suite Ramses [KS06], we developed code generators [Kra03, Stø04] for the ActorFrame platform, both the J2EE as well as the J2SE version. Based on them, a number of prototypical service applications were realized and deployed, including services running in the operational network of the Norwegian telecom operator Telenor. As input, Ramses uses UML 2.0 models based on state machines as presented in the next section.

2.3 Executable Service Models in UML 2.0

In the following, we outline a profile in UML 2.0 that can be directly mapped to an execution in JavaFrame-based systems. The complete profile is presented in [Kra06]. In particular, we introduce constraints on transitions to facilitate scheduling and refine some semantic variation points of UML 2.0.

Figure 2.4 shows an object diagram of our example system with a number of devices, two local nodes, and the central node in the middle. Each state machine owns some ports that are used to transmit the signals to other state machines via the links connecting them. A state machine can send a signal by putting it into the output queue of a port. Thereafter, the signal is transmitted via a link. At the receiver side, the signal is added to the common input queue of the state machine. If one port is connected to several others, the signal contains some routing information that can be used by the sending port to choose the correct receiver. We assume hereby that signals are transferred in an order-preserving and reliable way, and that the queues are unbounded.[1]

[1]In Sect. 6, we describe in the context of the formal method cTLA under which circumstances these practical limitations can be handled.

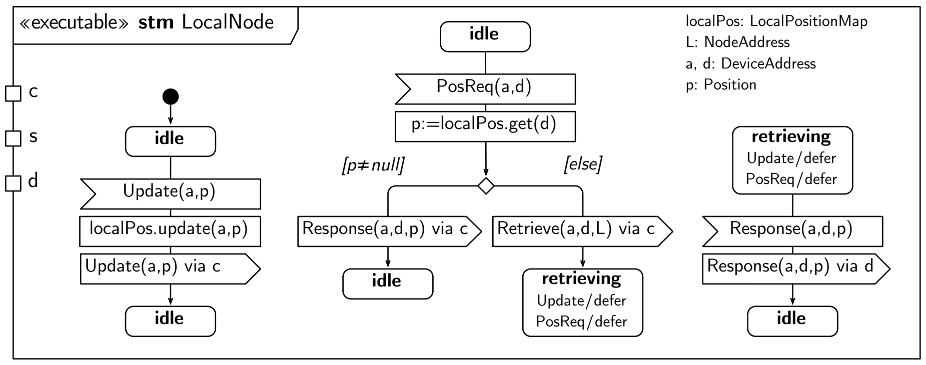


Fig. 2.5: State machine for a local node

Figure 2.5 shows the state machine for a local node. For the execution model, we use only a subset of UML 2.0 state machines. In particular, we assume that a state machine has exactly one region and that transitions have a certain structure, which is described later. To distinguish such state machines from other state machines in UML, we mark them with the stereotype ≪executable≫.

As in JavaFrame, events are either the reception of signals or the expiration of local timers. UML assumes the events arriving at a state machine to be stored in an event pool, and gives no further rules for the order of dispatching them, intentionally allowing different strategies. We assume the input pool to be a FIFO queue like in SDL processes, so that events are dispatched in the order of their arrival. This matches the scheduling procedure in JavaFrame. Deferred events are specified in UML by writing them into the state symbol with the keyword */defer*. For example, in the state *retrieving* in Fig. 2.5, incoming *Update* or *PosReq* signals are deferred, until *Response* arrives and the state machine changes into state *idle*.

Actions can be executed as the effect of transitions. A state machine operates on its auxiliary variables, controls local timers, and sends signals to other state machines. Actions may also call operations defined for the auxiliary data. Such actions must execute within the same run-to-completion step and therefore be local and not waiting on external events. In our example, after receiving an update, the state machine updates the local data structure *localPos* by using its operation *update()*. Send signal actions can be used to transmit signals to other state machines. We assume that these actions are assigned to a port (using the keyword *via*) and that the signals contain information so that the port may decide about the destination. The local node, for instance, includes its own address *L* in signal *Retrieve* that is sent via port *c* towards the central node. This address may be used by the output port of the central node to route signal *Response* for the answer.

As we only use simple states and pseudo states of the kinds *choice* and *initial*, we may distinguish the following forms of transitions:

– A *simple transition* connects two control states without any decisions or pseudo states.

- A *compound transition* is similar to a simple transition but can contain choices, so that its effects and the target state can depend on a decision. The decision is made by the guards that are declared on each branch originating from a choice. UML requires that at least one of these guards is true, so that a compound transition can always be completed once it is started.
- An *initial transition* originates from an initial pseudo state and is executed when the state machine is started. Each state machine has exactly one initial transition. Like a compound transition, an initial transition may also use choices that result in different branches.

Due to the scheduling mechanism in the execution platform, we assume that all transitions, that do not originate in an initial pseudo state, have exactly one trigger that matches either a signal reception or a timer expiration event. The scheduler assumes a transition to be enabled if the state machine is in the declared source state and the next event to be dispatched matches the one declared as trigger by the transition. In consequence, a transition may not declare any additional guards that would prevent its execution. We further assume that an event is not deferred in a state if a transition originates from that state with the same event as trigger. Thus, an incoming event is either consumed by a transition or deferred in a given state.

2.4 Compositional Temporal Logic of Actions (cTLA)

Lamport's Temporal Logic of Actions (TLA, [Lam02]) is a linear-time temporal logic modeling the behavior of a system as a set of infinitely long state sequences

$$\langle s_0, s_1, s_2, \ldots \rangle.$$

Thus, the TLA semantics fits excellently with that of the state machines introduced above which, in the end, also model infinite sequences of states s_i starting with an initial state s_0. Compositional TLA (cTLA, [HK00a]) was derived from TLA to provide more easily comprehensible specifications and offer a more flexible composition of specifications. cTLA is oriented at programming languages and introduces the notion of processes. A cTLA process can be in a simple form which directly describes system behavior by means of state transition systems. A process can also be compositional and describe systems as a combination of other process instances each specifying a sub-functionality of the system.

An example of a simple process type is sketched in Fig. 2.6. The header *LocalNode* declares the name of the process type while generic module parameters like *DeviceAddr* enable to specify a spectrum of similar process instances by a single process type. *Signals* is a constant record-typed expression. The body of

```
 PROCESS LocalNode (DeviceAddr: ANY; MyDevices: SUBSET(DeviceAddr);
                  NodeAddr: ANY; MyAddress: NodeAddr;
                  Pos: ANY; unknownPos: Pos)
CONSTANTS
  Signals ≜  [[t: {Start, Update, PosReq, Response, Retrieve};
               a: DeviceAddr; d: DeviceAddr; l: MyAddr; p: Pos]] ;
VARIABLES
  state: {initState, idle, retrieving}; localPos: [MyDevices → Pos];
  inQueue: QUEUE OF Signals;    deferQueue: QUEUE OF Signals;
  outQueueC: QUEUE OF Signals;   outQueueD: QUEUE OF Signals;
INIT ≜
  state = initState  ∧  inQueue = EMPTY  ∧  deferQueue = EMPTY ∧
  outQueueC = EMPTY  ∧  outQueueD = EMPTY ∧ localPos ∈ [MyDevices  →  Pos];
ACTIONS
  enqueue (inSignal : Signals) ≜
    inQueue′ = inQueue ∘ ⟨inSignal⟩ ∧ state ≠ initState ∧
    UNCHANGED ⟨deferQueue, state, outQueueC, outQueueD, localPos⟩;
  dequeueC (outSignal : Signals) ≜
    outQueueC ≠ EMPTY  ∧  outSignal = FIRST(outQueueC) ∧
    outQueueC′ = TAIL(outQueueC) ∧
    UNCHANGED ⟨inQueue, deferQueue, state, outQueueD, localPos⟩;
  dequeueD (outSignal : Signals) ≜ ... ;
  initial ≜ state = initState ∧ state′ = idle ∧
    localPos′ = [d ∈ MyDevices | d ↦ unknownPos] ∧
    UNCHANGED ⟨inQueue, deferQueue, outQueueC, outQueueD⟩;

INTERNAL ACTIONS
  update ≜ state = idle  ∧  FIRST(inQueue).t = Update ∧
    state′ = idle ∧
    inQueue′ = deferQueue ∘ TAIL(inQueue)  ∧  deferQueue′ = EMPTY ∧
    localPos′ = [localPos EXCEPT FIRST(inQueue).a ↦ FIRST(inQueue).p] ∧
    outQueueC′ = outQueueC ∘ ⟨ [[t ↦ Update; a ↦ FIRST(inQueue).a;
                             d ↦ FIRST(inQueue).d; l ↦ MyAddress;
                             p ↦ FIRST(inQueue).p ]] ⟩ ∧
    UNCHANGED ⟨outQueueD⟩;
  requestPos ≜ state = idle  ∧  FIRST(inQueue).t = PosReq ∧
    state′ = IF FIRST(inQueue).a ∈ MyDevices THEN idle ELSE retrieving ∧
    inQueue′ = deferQueue ∘ TAIL(inQueue)  ∧  deferQueue′ = EMPTY ∧
    localPos′ = localPos ∧
    outQueueC′ = outQueueC ∘ IF FIRST(inQueue).d ∈ MyDevices THEN EMPTY
                          ELSE ⟨ [[t ↦ Retrieve; a ↦ FIRST(inQueue).a;
                                  d ↦ FIRST(inQueue).d;
                                  l ↦ MyAddress ]] ⟩ ∧
    outQueueD′ = outQueueD ∘ IF d = FIRST(inQueue).d ∈ MyDevices
                          THEN ⟨ [[t ↦ Response; a ↦ First(inQueue).a;
                                  d ↦ FIRST(inQueue).d; l ↦ MyAddress;
                                  p ↦ localPos[First(inQueue).d]] ]⟩
                          ELSE EMPTY;
  retrievePos ≜ ... ;
  deferInRetrieving ≜ state = retrieving ∧ FIRST(inQueue).t ∈ {Update, PosReq}
    ∧ inQueue′ = TAIL(inQueue) ∧ deferQueue′ = deferQueue ∘ ⟨FIRST(inQueue)⟩ ∧
    UNCHANGED ⟨state, localPos, outQueueC, outQueueD⟩;
WF: dequeueC, dequeueD, initial, update, requestPos,
    retrievePos, saveInRetrieving;
END
```

Fig. 2.6: cTLA/e process modeling the local node

a simple cTLA process type describes a state transition system. It contains a set of variables like *state* or *inQueue* modeling the state space. The subset of initial states is specified by the predicate *INIT*. The transitions are expressed by actions (e.g., *enqueue*, *dequeueC*) which are predicates on pairs of a current and a next state describing a set of transitions each. Variables in simple form (e.g., *inQueue*) refer to the current state while the next state is described by the so-called primed form (e.g., *inQueue'*). The statement *UNCHANGED* lists variables not changed by an action. Action parameters like *inSignal* allow to model different actions by a single representation. Actions can be distinguished into two classes. External actions can be coupled with actions of the process environment while internal actions cannot.

We can provide actions with weak and strong fairness properties guaranteeing that they are carried out in a lively manner. In particular, weak fairness forces the execution of an activity if it would be enabled continuously otherwise. Strong fairness forces the execution even if the action is sometimes disabled. Unlike TLA, cTLA provides for conditional fairness assumptions to ensure the consistency of the process compositions introduced below. A fairness statement refers to periods of time in which an action is both enabled and the environment of the process is ready to tolerate the action. The statement *WF: dequeueC, dequeueD,...* indicates that the listed actions have to be carried out weak fairly.

A process type describes a set of TLA state sequences. The first state s_0 of each modeled state sequence has to fulfill the initial condition *INIT*. The state changes $\langle s_i, s_{i+1} \rangle$ either correspond with a process action or with a so-called stuttering step in which the current and the next states are equal (i.e., $s_i = s_{i+1}$). The fairness assumptions have to be fulfilled as well. cTLA also allows to define additional real time properties [GHK00] and the description of continuous behavior [HK00b] which we omit here for the sake of brevity.

Compositional cTLA processes model systems as compositions of concurrent process instances. Since the process variables are encapsulated and can only be referenced by the actions of the process defining them, the system state space is basically the vector of the variables of all process instances belonging to the system. We compose processes with each other by coupling their external actions to joint system actions. Formally, a system action is a conjunction of the corresponding process actions which therefore are executed simultaneously. A process may contribute to a system action with either exactly one process action or with a stuttering step. An internal process action, however, must only be coupled with stuttering steps of the other processes.

Figure 2.7 describes a compositional process type. It consists of the process instances *c*, *l1*, *l2*, etc. which are listed in the section *PROCESSES*. At that place, we also specify the module parameter instantiations (e.g., the parameter *myDevices* of the instance *l1* of process type *LocalNode* in Fig. 2.7 is instantiated with the set $\{d1, d2\}$). The system actions are depicted in the lower part of the specification[2] as conjunctions of process actions. For instance, the system

[2]To keep the specification short, we omitted processes performing stuttering steps in each system action description.

```
PROCESS System
CONSTANTS
  DevAddr ≜ {d1, d2, d3}; NodeAddr ≜ {l1, l2};
  Ps ≜ [[x : REAL; y : REAL; z : REAL]] ;
  uPs ≜ [[x ↦ 0, y ↦ 0, z ↦ 0 ]] ;
  Sig ≜ [[t : {Start, Update, PosReq, Response, Retrieve};
          a : DevAddr; d : DevAddr; p : Ps]] ;
PROCESSES
  cn: CentralNode (DeviceAddr ← DevAddr, Pos ← Ps, unknownPos ← uPs);
  l1: LocalNode (DeviceAddr ← DevAddr, myDevices ← {d1, d2},
                 NodeAddr ← NodeAddr, MyAddress ← l1,
                 Pos ← Ps, unknownPos ← uPs);
  s1: Sensor (DeviceAddr ← DevAddr, myDevices ← {d1, d2}, Pos ← Ps);
  d1: Device (DeviceAddr ← DevAddr, myDeviceAddr ← d1, Pos ← Ps);

...initializations of local node l2, sensor s2 and devices d2 and d3...
INTERNAL ACTIONS
Initial
  cnInitial ≜ cn.initial; l1Initial ≜ l1.initial; l2Initial ≜ l2.initial;
  d1Initial ≜ d1.initial; d2Initial ≜ d2.initial; d3Initial ≜ d3.initial;
  s1Initial ≜ s1.initial; s2Initial ≜ s2.initial;

local nodes ↔ central node (portC)
  l1toc(sig: Sig) ≜ l1.dequeueC(sig) ∧ c.enqueue(sig);
  l2toc(sig: Sig) ≜ l2.dequeueC(sig) ∧ c.enqueue(sig);
  ctol1(sig: Sig) ≜ c.dequeueC(sig) ∧ l1.enqueue(sig) ∧ sig.l = l1;
  ctol2(sig: Sig) ≜ c.dequeueC(sig) ∧ l2.enqueue(sig) ∧ sig.l = l2;

... actions for other connections ...
END
```

Fig. 2.7: cTLA/e process modeling the global system

action *ctol1* corresponds to the joint execution of the process actions *dequeueC* of process *C* and *enqueue* of *l1* while the other processes perform stuttering steps. The data transfer between *c* and *l1* is modeled by the action parameter *sig*. Moreover, we added an additional conjunct *sig.l* = *l1* enabling the execution of the action for certain action parameter settings only. In [HK00a] we proved that compositional cTLA processes can be transformed into equivalent simple processes which enables nested system specifications.

2.5 cTLA/e: An Executable Form of cTLA

cTLA is a powerful means to describe various forms of behavior. The cTLA specifications, however, may have a form that is difficult to implement efficiently. Therefore, we describe a special cTLA specification style (cTLA/e), which directly models the mechanisms of the execution platforms exemplified by JavaFrame in Sect. 2.2. cTLA/e determines a form for simple processes corresponding to state machines explained in Sect. 2.5.1 and a form for compositional processes to couple the state machines in Sect. 2.5.2.

2.5.1 cTLA/e Process for State Machines

In the following, we will sketch the cTLA/e models of state machines by the specification of the local node from the example listed in Fig. 2.6. This process corresponds to the state machine of the local node depicted in Fig. 2.5. One state machine is represented by one cTLA/e process. The control state is described by a cTLA variable *state* expressing the enumeration of the control state identifiers. Incoming signals are placed in the data structure *inQueue*, which is a sequence of signals with the operations *FIRST()* to obtain the first element and *TAIL()* to get the queue after removing the first element. The operator $\circ$ denotes the concatenation of queues. Similarly, the defer queue for signals is modeled by the cTLA variable *deferQueue*. Signals are appended to the input queue by the action *enqueue*, which has the received signal as action parameter.

For each port used to send signals to other state machines, the process contains an output queue. [3] Signals are records, where the field t denotes the type of the signal, a the device address calling for a position or part of an update, d the device address for which a position is requested, l the address of a local node retrieving a position, and p the position information. To send a signal via a port, a transition adds it to the corresponding output queue. A dequeue action defined for each port (e.g., *dequeueC* and *dequeueD*) is used to transmit the signals from the output queues to their respective receivers. Additional variables represent the auxiliary variables of the state machine. For instance, a local node stores the positions of its local devices in the map *localPos*.

Every transition of the state machine is represented by a cTLA action formulated as a conjunction of several sub-actions $t_{trans} = t_{en} \wedge t_{next} \wedge t_{qm} \wedge t_{send} \wedge t_{aux}$, each having a distinct purpose:

– The enabling sub-action $t_{en} = t_{\text{trigger}} \wedge t_{prev}$ determines whether a transition is ready to execute. This depends on the first event in the input queue (t_{trigger}) and the current control state (t_{prev}). For example, the action *update* defines a transition enabled in control state *idle* and for the signal *Update* with *state = idle* $\wedge$ *FIRST(inQueue).t = Update*. As an initial transition has no trigger, sub-action t_{trigger} is omitted in action *initial.*

– The target state sub-action t_{next} specifies the change of the control state. It simply is an assignment to the control state variable. For compound transitions including several branches, the assignment can include an if-statement. The target of the *requestPos* transition, for instance, is either state *idle* or *retrieving.*

– The queue maintenance sub-action t_{qm} describes the move of the content of the defer queue to the front of the input queue, so that they are again available for consumption in the next state. The sub-action $t_{qm} \triangleq$ *inQueue'= deferQueue* $\circ$ *tail(inqueue)* $\wedge$ *deferQueue'= EMPTY* is identical for every

[3]In the example, no signals are sent from the local node to the sensor via port s. Therefore, this port is not represented with an output queue.

transition. As the defer queue is empty when the initial transition is executed, it is not necessary to include this sub-action in the action *initial*.

– Sub-actions t_{send} model the transmission of signals, simply by appending them to the corresponding output queue. The signals sent may depend on conditions, which can be expressed by an if-statement. For example, transition *requestPos* either sends *Response* via port D or *Retrieve* via port C.

– Sub-actions t_{aux} specify the new settings of the local auxiliary variables. The local position map *localPos*, for instance, is updated with the new position in transition *update*.

Like on our execution platforms based on JavaFrame, deferred signals are moved into a the dedicated defer queue by an explicit action. This action is a conjunct $t_{trigger} \wedge t_{prev} \wedge t_{defer}$, with t_{defer} performing the actual move into the defer queue. For instance, in Fig. 2.6, *deferInRetrieving* removes the signals *Update* or *PosReq* from the input queue and appends them to the defer queue.

Similarly to SDL, we model timers by means of signals. The starting, stopping and triggering of a timer is specified by auxiliary cTLA actions. Once a timer expires, the runtime support system places a signal representing the timer expiration in the input queue. In our example system, we use timers in the sensors which, however, are not listed for the sake of brevity.

2.5.2 cTLA/e Process for the Global System

The system is specified by a compositional cTLA process combining the processes for the individual state machines, as shown in Fig. 2.7. After declaring constants for the used types such as device addresses, signal formats and positions, it defines a process instance for each state machine instance and passes parameters to them. The configuration reflects the system structure given in Fig. 2.4 by initializing local nodes and sensors with the device addresses attached to them.

According to the system structure, the corresponding dequeue and enqueue actions are coupled together, so that signals can be transfered from an output queue to the input queue of the receiver. In our example, we represent each link between two state machines by an individual cTLA action. For example, the links from the central nodes to each of the local nodes are represented by cTLA actions *ctol1* and *ctol2* in which the additional conjuncts *sig.l* = *l1* and *sig.l* = *l2* model the routing decision. To enable scalable system models, we can also use coupling descriptions specifying various links and, in particular, dynamic connections by a single cTLA system action (cf. [Her03]).

2.6 Executing cTLA/e Specifications

To provide the complete formal proof, that our code generators produce software code implementing a cTLA/e specification correctly, we need to create a fully-fledged cTLA model of the code, which is beyond the scope of this paper. Therefore, we only provide a sketch of the proof. As mentioned previously, the specification style cTLA/e was laid out in a way that its actions correspond with the program steps of the generated code based on JavaFrame. Moreover, the variables used in cTLA/e reflect directly the variables in the executable code. For instance, the sub-action t_{qm} is similar to the step of the implementation where in a transition the first signal is removed from the input queue and previously deferred signals are moved to the front of the input queue in the order of their deferral. Thus, we can describe the execution of a transition as an order of steps:

$$S_i \xrightarrow{t_{\text{trigger}}} \widehat{S}_{i,1} \xrightarrow{t_{\text{prev}}} \widehat{S}_{i,2} \xrightarrow{t_{\text{aux}}} \widehat{S}_{i,3} \xrightarrow{t_{send}} \widehat{S}_{i,4} \xrightarrow{t_{next}} \widehat{S}_{i,5} \xrightarrow{t_{qm}} S_{i+1}$$

As previously mentioned, the implemented state machines follow the run-to-completion semantics, so that the sequence of steps is carried out without interruptions by other events. Therefore, it is easy to prove formally that this sequence implies the sequence

$$\overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{stutter} \overline{S_i} \xrightarrow{t_{\text{trans}}} \overline{S_{i+1}}$$

That means the first five steps of the executed transition are mapped to stuttering steps in cTLA/e, while the last step is mapped to the cTLA/e action modeling the entire transition in one (atomic) step. This is a well-known example of a formally correct refinement step as described for example in [Lam96]. Likewise, we can verify that a signal deferral consisting of the steps

$$S_i \xrightarrow{t_{\text{trigger}}} \widehat{S}_{i,1} \xrightarrow{t_{\text{prev}}} \widehat{S}_{i,2} \xrightarrow{t_{\text{defer}}} S_{i+1}$$

implements the cTLA/e defer action.

In cTLA/e, a signal transmission is modeled by three distinct actions: (1) the transition putting the signal into an output queue, (2) the action transferring the signal from the output queue to the input queue of the receiver (as a conjunction of two process actions) and (3) the transition triggered by the signal that consumes it. Thus, action (2) is an abstraction of the transmission mechanism of a middleware layer in an implementation and the signals currently in the cTLA/e output queues are assumed to be under transmission.

Of course, we have to consider that resources in the real world are limited and computation steps take time. In particular, the size of signal queues is bounded and buffer overflows may occur. In our example, a sensor may send position updates so frequently, that the local node cannot process all of them. To avoid this, we can introduce mechanisms already on the specification level. We may, for instance, require the sensor to wait for an acknowledgment from the local node before sending another update. Alternatively, updates may only be sent when requested by the local node. In this case, one can verify by cTLA-based invariant proofs that the queues do not exceed an upper bound. Furthermore, we may use real-time reasoning to guarantee the boundedness of queues. For instance, we may enforce a minimum waiting time for the sensor and maximum response time properties for other system actions using the real-time extension of cTLA [GHK00]. Then we can prove that the local node can handle an update signal even in peak situations before the next update is triggered. This is complementary to the technique used in [BH93] which estimates the execution time of transitions.

A deadlock can occur if there is a signal at the first position of the queue that a state machine cannot handle in its current state (i.e., neither consume in a transition nor defer). To prevent this kind of design flaw, we should verify by means of cTLA invariant proofs that every incoming signal can be handled. In our example, the local nodes have two states[4] *idle* and *retrieving.* The signals *Update* and *PosReq* can always be consumed in state *idle* (by the actions *update* or *requestPos* in Fig. 2.6) and deferred in state *retrieving* (by action *saveInRetrieving*). In contrast, signal *Response* is only consumed in the state *retrieving.* Therefore, we must verify the cTLA invariant, that this signal is only sent by the central node if the local node is in the state *retrieving.* Based on the activity diagram in Fig. 2.2 it is evident that this invariant is straightforward.

So far, we considered safety properties guaranteeing that "nothing wrong" happens. Beyond that, the layout of cTLA/e, and the scheduling mechanisms based on JavaFrame also allow assertions about liveness properties, describing that *"...something good eventually happens..."* [AS85]. In cTLA, liveness is expressed by the fairness assumptions introduced in Sect. 2.4. The layouts of cTLA/e and the JavaFrame-based scheduler guarantee that every transition once enabled will eventually be executed, since the following properties hold:

– Due to the isolation of state machines and the fact that transitions are enabled based on the source state and trigger event only, a transition once enabled will remain enabled until it is executed.

– As explained in Sect. 2.2, there is at most one transition enabled for each combination of a source state and a trigger event.

– Due to the cTLA invariant proof, all received signals can be handled.

– The scheduler serves all of its state machines in a round-robin fashion.

One can verify that these properties imply the strong fairness properties (and, in consequence, the weak fairness properties) of the corresponding cTLA/e actions. This is a valuable property of our execution platform, as it is the prerequisite to include fairness reasoning on the more abstract specifications of our system as well. If we can prove that fairness assumptions of more abstract collaborations are fulfilled by the cTLA/e refinement, it is evident, that these assumptions are also realized by the executable code.

[4]We can disregard the initial pseudo state *initState* here, as the originating initial transition is enabled independently of the input queue and will be eventually executed due to its fairness property.

2.7 Related Work

Closest related to our work is probably that of the specification approach and language DisCo [KS05], which, like cTLA, is based on the Temporal Logic of Actions. Similar to collaborations, DisCo is focusing on the cooperation of objects. Instead of processes as in cTLA, DisCo uses layers that may be composed or refined. To facilitate a specification-driven approach, Pitkänen [Pit04, Pit06] introduces an additional level of refinement called TransCo. This is a subset of the DisCo language and oriented towards business components and transactions. TransCo can be derived from DisCo by refinement and then further be translated into J2EE applications by an experimental code generator. The concept of an intermediate formal language like cTLA/e or TransCo is also present in the B-Method [Abr96], where a subset of B — called B0 — is closer to imperative languages that are easier to implement. The intermediate languages TransCo, B0, and cTLA/e focus on different domains or platforms. While TransCo targets at transaction processing, and B0 is close to sequential code like ADA, cTLA/e is an abstraction of the executable state machines described above.

In this paper, we focused on the formal treatment of an execution model to ensure correctness of the resulting programs. If we extend our scope towards the development of reactive systems in general, we naturally find other methods with slightly different aims, specialized towards other domains. One approach that seems to cover the step from specifications to executable code in a rather complete way, is that of Burmester et al. in [BGHS04] which is integrated into the FUJABA toolset. They focus on specifications of systems including real-time properties. Similar to collaborations, they specify patterns that can be verified independently and composed together. For the description of these patterns, UML state machines extended with real-time properties are used. To transform a specification into executable systems, an intermediate model is described in [BGS05] that takes platform-specific aspects into consideration, such as the assignment of state machine instances to execution threads. For the implementation they propose a direct mapping of one state machine instance to one real-time execution thread, instead of using a scheduler that takes advantage of the state machines, as described in Sect. 2.2.

2.8 Concluding Remarks

We described how distributed services can be efficiently executed based on communicating state machines. Moreover, we outlined the mechanisms of JavaFrame to exemplify how execution platforms and support systems can be constructed. It was further discussed which form UML 2.0 state machines should have in order to be easily transformable to programs using the presented execution mechanisms. We defined a cTLA specification style (cTLA/e) to combine the correctness-preserving service design with the efficient execution mechanisms. cTLA/e is dedicated to an easy and correct mapping of the state machines forming the input of JavaFrame-based implementations. We made plausible that the implementations fulfill interesting properties concerning the fairness of execution and we outlined how boundedness of signal queues can be ensured.

We described a triangle relationship between the efficient execution of services, the intuitive modeling based on UML, and the formal analysis based on temporal logic with cTLA/e. This relationship also aligns the scopes of three different kinds of engineers that perform activities in service engineering:

– Execution platform designers create mechanisms for the execution and deployment of services that need computational models allowing an efficient execution, such as the state machines presented in Sect. 2.2.

– Service engineers focused on specific applications want to have suitable modeling concepts and generally accepted notations, such as the UML 2.0 state machines of Sect. 2.3.

– Providers of tools for modeling, analysis, and transformations need a formal logic like cTLA of Sect. 2.4 to to reason about the correctness of tools and methods.

With cTLA/e we provided such an alignment for the execution of services. It is the final stage in our strategy to generate executable components from formal collaborations describing the services. In addition to cTLA/e, we developed another cTLA specification style modeling collaborations which uses UML 2.0 collaborations for a structural description and UML activities for the behavioral part, like the ones briefly presented in the introduction. In the next step, we will specify how this cTLA style can be refined to cTLA/e. In particular, we want to provide service engineers with the suitable means for the correctness-preserving top-down construction of distributed services. Here, cTLA already proved its capability for various application domains [HK00a, Her03, Her06, HK00b]. As part of this work we have to reduce collaborations to component models as needed for the execution. This means to re-arrange the process structure described by the collaborations and to split it into the behavior that each service component contributes to a collaboration. An integral part of such a refinement is the adaption of the process couplings and the cTLA actions into the form we described by cTLA/e.

The combination of UML 2.0 modeling with cTLA-based reasoning offers a number of practical advantages for service engineering in general. Most prominent is the realization of the correctness-preserving refinement as UML 2.0 model transformations. Here, the cTLA refinement steps are a fundament for creating MDA tools performing suitable model transformations. While these tools use cTLA formalizations of the UML models and the refinement steps, for the service engineer cTLA will in fact be invisible. ■

Bibliography

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA, 1996.

[AS85] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.

[BF04] Rolv Bræk and Jacqueline Floch. ICT Convergence: Modeling Issues. In Daniel Amyot and Alan W. Williams, editors, *SAM'04 - Fourth SDL and MSC Workshop*, volume 3319 of *Lecture Notes in Computer Science*, pages 237–256. Springer, 2004.

[BGH$^+$97] Rolv Bræk, Joe Gorman, Øystein Haugen, Geir Melby, Birger Møller-Pedersen, and Richard Sanders. Quality by Construction Exemplified by TIMe — The Integrated Methodology. *Telektronikk*, 95(1):73–82, 1997.

[BGHS04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In *Proc. of the International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004*, pages 1–20, October 2004.

[BGS05] Sven Burmester, Holger Giese, and Wilhelm Schäfer. Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code. In *Proceedings of the European Conference on Model Driven Architecture — Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany*, volume 3748 of *Lecture Notes in Computer Science*, pages 25–40. Springer, 2005.

[BH93] Rolv Bræk and Øystein Haugen. *Engineering Real Time Systems: An Object-Oriented Methodology Using SDL.* The BCS Practitioner Series. Prentice Hall, 1993.

[BHM02] Rolv Bræk, Knut Eilif Husa, and Geir Melby. *ServiceFrame Whitepaper.* Ericsson NorARC, Asker, Norway, April 2002.

[Bræ79] Rolv Bræk. Unified System Modelling and Implementation. In *International Switching Symposium*, pages 1180–1187, Paris, France, May 1979.

[CB06] Humberto N. Castejón and Rolv Bræk. A Collaboration-based Approach to Service Specification and Detection of Implied Scenarios. In *SCESM '06: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pages 37–43, New York, NY, USA, 2006. ACM Press.

[Est97] ISO. *ESTELLE: A Formal Description Technique Based on an Extended State Transition Model*, International Standard ISO/IEC 9074 edition, 1997.

[FB00] Jacqueline Floch and Rolv Bræk. Towards Dynamic Composition of Hybrid Communication Services. In *SMARTNET '00: Proceedings of the IFIP TC6 WG6.7 Sixth International Conference on Intelligence in Networks*, pages 73–92, Deventer, The Netherlands, 2000. Kluwer, B.V.

[GHK00] Günter Graw, Peter Herrmann, and Heiko Krumm. Verification of UML-Based Real-Time System Designs by means of cTLA. In *Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC2K)*, pages 86–95, Newport Beach, 2000. IEEE Computer Society Press.

[Her03] Peter Herrmann. Formal Security Policy Verification of Distributed Component-Structured Software. In Hartmut König, Monika Heiner, and Adam Wolisz, editors, *Proceedings of the 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'2003), Berlin, Germany*, volume 2767 of *Lecture Notes in Computer Science*, pages 257–272. Springer-Verlag, September 2003.

[Her06] Peter Herrmann. Temporal Logic-Based Specification and Verification of Trust Models. In Ketil Stølen, William H. Winsborough, Fabio Martinelli, and Fabio Massacci, editors, *iTrust 2006*, volume 3986 of *Lecture Notes in Computer Science*, pages 105–119, Heidelberg, 2006. Springer–Verlag.

[HK00a] Peter Herrmann and Heiko Krumm. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337, 2000.

[HK00b] Peter Herrmann and Heiko Krumm. A Framework for the Hazard Analysis of Chemical Plants. In *Proceedings of the 11th IEEE International Symposium on Computer-Aided Control System Design (CACSD'2000)*, pages 35–41, Anchorage, 2000. IEEE CSS, Omnipress.

[HMP00] Øystein Haugen and Birger Møller-Pedersen. JavaFrame — Framework for Java Enabled Modelling. In Proceedings of Ericsson Conference on Software Engineering, September 2000.

[ITU02] ITU-T. *Recommendation Z.100: Specification and Description Language (SDL)*, August 2002.

[Kra03] Frank Alexander Kraemer. Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart, 2003.

[Kra06] Frank Alexander Kraemer. Profile for Service Engineering: Executable State Machines. Avantel Technical Report 2/2006 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway, March 2006.

[KS05] Reino Kurki-Suonio. *A Practical Theory of Reactive Systems.* Springer, 2005.

[KS06] Frank Alexander Kraemer and Haldor Samset. Ramses User Guide. Avantel Technical Report 1/2006, Department of Telematics, NTNU, Trondheim, Norway, 2006.

[Lam96] Leslie Lamport. Refinement in State-Based Formalisms. Technical Report 1996-001, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, 1996.

[Lam02] Leslie Lamport. *Specifying Systems.* Addison-Wesley, 2002.

[LF04] Swee Boon Lim and David Ferry. *JAIN SLEE 1.0 Specification, Final Release.* Sun Microsytems, Inc. and Open Cloud Ltd., 2004.

[Mel03] Geir Melby. Using J2EE Technologies for Implementation of Actor-Frame Based UML 2.0 Models. Master's thesis, Agder University College, Grimstad, Norway, May 2003.

[MH05] Geir Melby and Knut Eilif Husa. *ActorFrame Developers Guide.* Ericsson NorARC, Asker, Norway, September 2005.

[MK95] Arnulf Mester and Heiko Krumm. Composition and Refinement Mapping based Construction of Distributed Applications. In *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Aarhus, Denmark, 1995. BRICS.

[MT01] Andreas Mitschele-Thiel. *Systems Engineering with SDL: Developing Performance-Critical Communication System.* John Wiley & Sons, Inc., New York, NY, USA, 2001.

[Obj05] Object Management Group. *Unified Modeling Language: Superstructure, version 2.0*, July 2005. formal/2005-07-05.

[Pit04] Risto Pitkänen. A Specification-Driven Approach for Development of Enterprise Systems. In *Proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques (NWPER'04)*, Turku, Finland, August 2004.

[Pit06] Risto Pitkänen. *Tools and Techniques for Specification-Driven Software Development.* PhD thesis, Tampere University of Technology, August 2006.

[Pnu86] Amir Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. *Current Trends in Concurrency. Overviews and Tutorials*, pages 510–584, 1986.

[RB06] Judith E. Y. Rossebø and Rolv Bræk. Towards a Framework of Authentication and Authorization Patterns for Ensuring Availability in Service Composition. In *Proceedings of the 1st International Conference on Availability, Reliability and Security (ARES'06)*, pages 206–215. IEEE Computer Society Press, 2006.

[SCKB05] Richard Sanders, Humberto N. Castejón, Frank Alexander Kraemer, and Rolv Bræk. Using UML 2.0 Collaborations for Compositional Service Specification. In *ACM / IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, 2005.

[SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley & Sons, Inc., New York, NY, USA, 1994.

[SIS96] SISU II Project. http://www.sintef.no/units/informatics/projects/sisu/, 1996.

[Stø04] Alf Kristian Støyle. Service Engineering Environment for AMIGOS. Master's thesis, Norwegian University of Science and Technology, 2004.

Comment

In Fig. 2.2, we used pairs of send and accept signal actions to model communication between activity partitions, for example *Update* in activity *UpdatePos*. This article was the first one published. In all later publications and the current version of our specification style, we use send and accept signal actions only to communicate with the environment. Comunication between different partitions is done by simple activiy flows, represented as single lines.

PAPER

THREE

TRANSFORMING COLLABORATIVE SERVICE SPECIFICATIONS INTO EFFICIENTLY EXECUTABLE STATE MACHINES

Frank Alexander Kraemer and Peter Herrmann.

Presented at the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT) March 31 – April 1, 2007 Braga, Protugal.

Published in *Graph Transformation and Visual Modeling Techniques*, Volume 6 of Electronic Communications of the EASST, 2007.

Transforming Collaborative Service Specifications into Efficiently Executable State Machines

Frank Alexander Kraemer and Peter Herrmann

Abstract. We describe an algorithm to transform UML 2.0 activities into state machines. The implementation of this algorithm is an integral part of our tool-supported engineering approach for the design of interactive services, in which we compose services from reusable building blocks. In contrast to traditional approaches, these building blocks are not only components, but also collaborations involving several participants. For the description of their behavior, we use UML 2.0 activities, which are convenient for composition. To generate code running on existing service execution platforms, however, we need a behavioral description for each individual component, for which we use a special form of UML 2.0 state machines. The algorithm presented here transforms the activities directly into state machines, so that the step from collaborative service specifications to efficiently executable code is completely automated. Each activity partition is transformed into a separate state machine that communicates with other state machines by means of signals, so that the system can easily be distributed. The algorithm creates a state machine by reachability analysis on the states modeled by a single activity partition. It is implemented in Java and works directly on an Eclipse UML2 repository.

3.1 Introduction

In a highly competitive market for modern networked services, it is important to deliver new services with short development times, in order to react on new customer demands quickly and to keep development costs low. These efforts are hampered by the typically high complexity of such services, which arises mainly from the fact that a service needs the coordinated effort of several participating components (cf. [FB00]). Hence, if we want to understand what a service does, we have to look at the behavior of all its participating components. Moreover, when services need to be adjusted or composed from other services, we must consider the descriptions of all participating components again and make sure that they interact correctly. Literature (e.g., [RGG01]) as well as experience from our own work [SCKB05, KH06, KHB06] stated that there are two dominant perspectives on a system delivering services:

– In the component-oriented perspective, systems are decomposed into physically distributed components, which are modeled separately. Services are specified indirectly by the composed behavior of the components. This perspective is well supported by traditional standards like SDL and its descriptions are easily transformable into executable code.

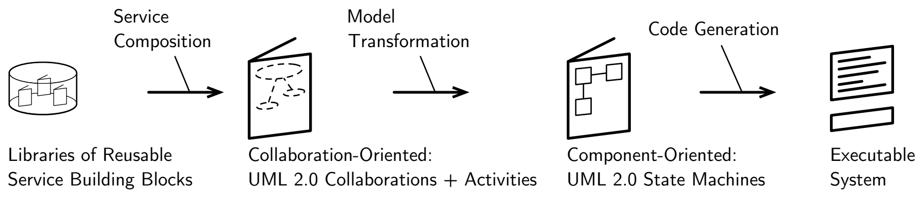

Service Composition
Model Transformation
Code Generation
Libraries of Reusable Service Building Blocks
Collaboration-Oriented: UML 2.0 Collaborations + Activities
Component-Oriented: UML 2.0 State Machines
Executable System

Fig. 3.1: Engineering approach for interactive services

- In the collaboration-oriented perspective, services are modeled by a number of collaborations as the main structuring elements. A collaboration specifies the interactions between the components involved in it, as well as the corresponding local behavior of the components to accomplish the service. Collaborations describe services in a self-contained form and may be composed from other ones. Within an application domain, collaborations contributing to a service are often similar which makes them ideal elements of reuse.

These two perspectives are the shaping forces behind our approach for the rapid engineering of interactive services, outlined in Fig. 3.1. Services are composed from collaborations that identify the interactions as well as the local behavior of a set of components that are necessary to fulfill a certain task. To express the structural aspects of collaborations as well as their composition (e.g., the participants and which roles they play in a service), we use the conforming concept of UML 2.0 collaborations. For the behavioral aspect (e.g., what a collaboration does as well as how collaborations are coupled together), we use UML 2.0 activities.

As an example, we consider an access control system (ACS) [BH93, BS01]. It controls the opening mechanism of a door and lets pass only authorized people that can prove their identity by presenting a security card and a secret number at an input panel. The opening mechanism and input panel are connected to a local station installed close to the door. Once a user draws the card and enters the pin, the resulting data (called pid) is transferred to a central station that authenticates the user and checks authorization right by querying two servers. If both, the authentication and authorization are successful, *ok* is sent back to the local station that opens the door.

In [KH06] we introduced how the ACS can be easily composed from reusable collaboration elements expressed by a combination of UML 2.0 collaborations for the structure (Fig. 3.2) and activities for the behavior (Fig. 3.3). These diagrams describe the system from a collaboration oriented perspective. To execute the system, however, we need a description of the behavior of the individual components, i.e., a description from a component-oriented perspective, as outlined above. In our approach, we use for this purpose so-called *executable* UML 2.0 state machines and composite structures, that are a suitable input for our code generators. To automate the step from the collaboration-oriented specifications in form of activities to the component-oriented design in form of state machines,

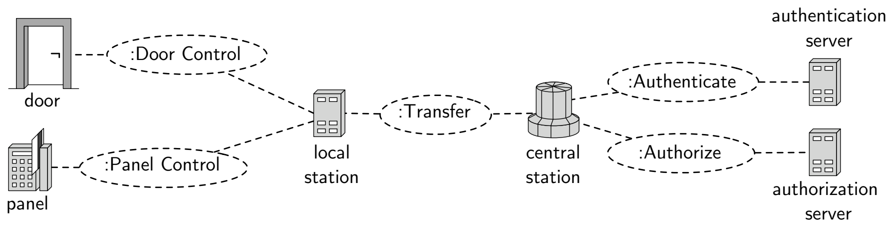


Fig. 3.2: Collaboration to compose the sub-services of the access control system

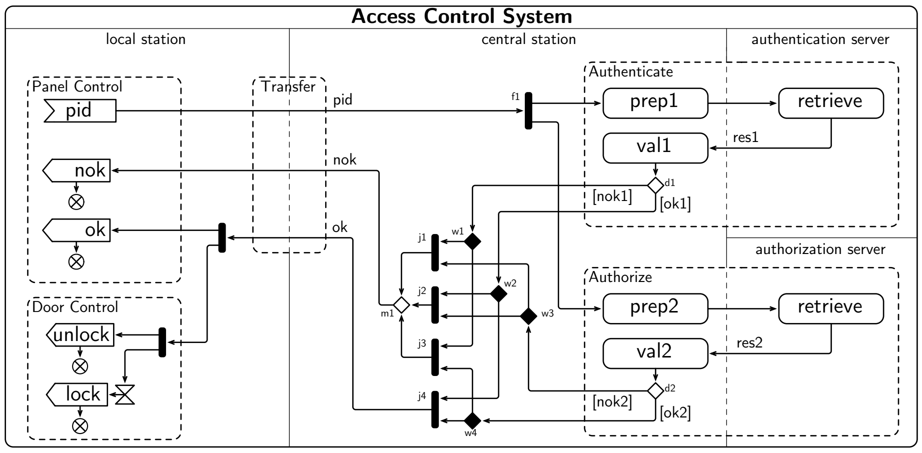


Fig. 3.3: Activity diagram modeling the detailed behavior of the system

we use a model transformation performed by the algorithm described in this article. Evidently, the introduction of such an automated transformation step accelerates the development of services drastically. In addition to the omission of manual labor for constructing the state machines, no new errors are introduced. Whenever a service specification needs to be updated, the state machines are simply generated again to ensure consistency. The algorithm creates the state machines without any intermediate representation and is therefore quite efficient concerning memory usage. Before we describe the principles of the transformation in Sect. 3.4 and the detailed algorithm in Sect. 3.5, we outline in the next two sections the two development perspectives outlined above. Sect. 3.6 sketches then a proof of the correctness of the transformation in temporal logic. We close with a discussion of related approaches and some concluding remarks.

3.2 Collaborations and Activities for Service Composition

While the collaboration in Fig. 3.2 shows how the system is composed structurally from elementary collaborations that were taken from a library, the activity diagram in Fig. 3.3 states how their behavior is coordinated. For each collaboration use of Fig. 3.2 (e.g., *Authenticate*), we find a structured node (in dashed lines) in the activity that specifies the behavior contributed by the collaboration use. Each collaboration role of Fig. 3.2 (e.g., *central station*) is a location of computation and represented by an activity partition in Fig. 3.3. The door and the panel are part of the environment, and, hence, do not have their own activity partition. Instead, they communicate with the local station by explicit signal send and receive actions. The local station receives a pid from the panel control and forwards it via the transfer collaboration to the central station. Depending on the result received from the central station (*ok* or *nok*), the local station will either cause the panel to display *nok* and leave the door locked, or it will cause the panel display to show *ok* and unlock the shutter of the door. In this case, a timer will be started which locks the door shutter again after a while. As activities have a Petri net like semantics [Obj06], we can use tokens and places to understand the behavior of the diagram in Fig. 3.3. Once a token representing a pid arrives at the central station, it is prepared (described by the operations *prep1* and *prep2*) and sent to the authentication respective the authorization server. For that, the token is duplicated at the fork node *f1*, so that the subsequent behaviors may happen in parallel. Both servers evaluate the pid and send their results back to the central station.

The results may arrive in any order. For example, if the result of the authorization server arrives first, it is evaluated by the central station (operation *val2*) which branches in decision *d2* depending on the validity of the authorization. If the result was valid, a token is placed in *w4*. This node is an extension of a decision node (cf. [KH06]) as tokens can rest in it. It is represented by a filled diamond. The central station waits now for the arrival of the authentication result, which is evaluated in *val1*, and a token is placed either on *w1* or *w2*. When the other result arrives, two waiting decisions hold one token, so that exactly one of the join nodes *j1..j4* can fire. Obviously, *j4* fires in the case that both results were ok and causes the central station to send an ok to the local station. In the other three cases (when at least one result is *nok*) one of the other join nodes *j1..j3* fires. These cases are combined by merge node *m1* and a *nok* is sent to the local station. We assume that the panel control only sends a new *pid* after it received an *ok* or a *nok*.

3.3 State Machines for Service Execution

Fig. 3.4 presents the executable UML state machines generated by our algorithm from the activity in Fig. 3.3. The state machines interact with each other

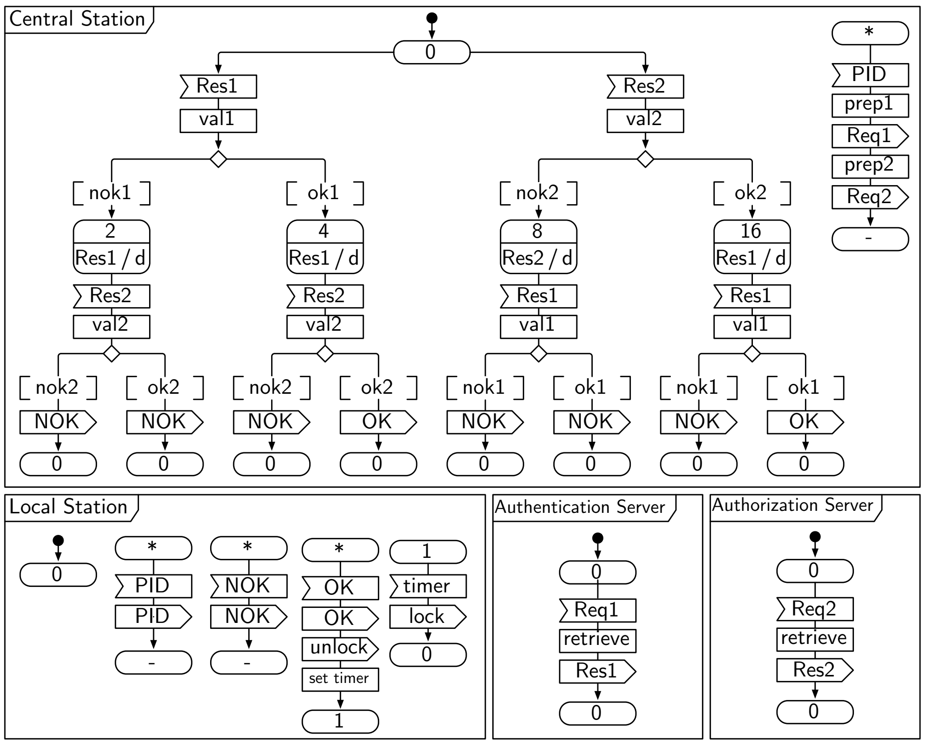


Fig. 3.4: Executable state machines for the system components

by transmitting signals which are buffered in event queues. Similar to SDL, UML allows for the use of send signal actions and signal triggers to describe the transmission and reception of signals. Each state machine has an initial state and a number of transitions that are triggered by either signal receptions or by timeouts. Transitions may include choices guarded by constraints (like *ok1*). As an effect, a transition may execute actions such as the sending of signals, the call of an operation (like *val1*) or the control of a timer. States can declare an event to be deferred by listing it in their body followed by the keyword "/defer" (abridged here to "/d"). This event is left in the queue until a state is entered that does not defer it anymore but declares a transition for its consumption. For compactness, we presented a transition that can be executed from any control state by referring to a state called "*". After it executes, the state machine returns into its originating state, denoted by "-".

As these executable state machines are the input for our code generators [Kra03], they must fulfill some constraints to achieve efficient code. In particular, they are event-driven, which means that each transition is only executed as the reaction to either the creation of the state machine itself, the reception of a signal, or the expiration of an internal timer. In consequence, transitions are enabled based purely on their source state and trigger, so that guards may only be declared on branches following choices. Moreover, for each pair of control state and trigger, merely one transition may be declared to prevent fairness conflicts between competing transitions.

These executable state machines have a long tradition in the telecommunication area (see for example [Bræ79]) and facilitate the efficient implementation on a range of different platforms and architectures, including J2EE. We defined in [KHB06] their execution semantics in terms of temporal logic and described, how they can be efficiently implemented using a scheduler as virtual machine layer. Of course, the constraints on the executable state machines needed to generate efficient code highly influence the layout of our algorithm which we discuss in the following.

3.4 Transformation from Activities to State Machines

In our approach, an activity partition corresponds to one physical point of execution. We therefore generate one state machine for each activity partition. This also makes it possible to consider the activity partitions separately and not the entire activity, as discussed later. To separate the partitions from each other, we have to cut those edges which cross partition borders. These edges model the control flows between different system components. As communication between the state machines is done entirely by means of signals, a flow crossing the boundaries of activity partitions must be implemented as a signal transmission. In activities, flows between actions occur instantaneously, i.e., a token leaves an action and enters a subsequent one without resting in the flow. The transmission between state machines, however, is buffered. Introducing signal transmissions in flows between partitions therefore implies virtual places that may hold tokens. We add these places where flows enter a partition, as illustrated in Fig. 3.5 by the circles with the queue symbols inside. These so-called *queue places*, which are also attached to receive actions, simulate the input queue of the state machines implementing an activity partition. In the model, these input queues are of unlimited capacity.[1] Thus, the virtual places are unbounded (i.e., can hold any number of tokens).

As described above, the state machines execute a transition as a reaction to the arrival of a signal. This event corresponds to the emission of a token from the virtual queue places. Hence, when we construct a transition, we simulate the emission of one token from a queue place. The token passes along the flows and nodes of the activity diagram until it reaches a control node where it has to wait for further events to happen. These are three kinds of nodes: (1) *Join nodes* synchronize different flows, that may arrive in any order. An incoming token may have to wait for the other incoming flows to arrive. (2) *Waiting decisions* synchronize competing join nodes (see Sect. 3.2). A token has to rest

[1] Of course, in an implementation, buffer capacity is limited, which can be addressed the means described in [KHB06].

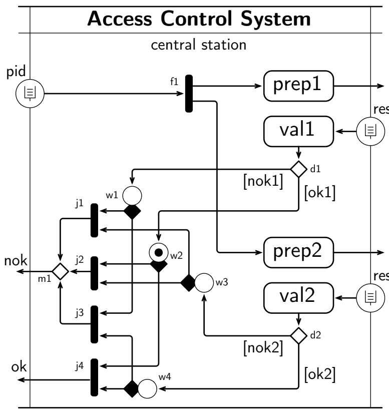


Fig. 3.5: Places for the nodes

inside a waiting decision if none of the succeeding joins can fire. (3) *Timer nodes* may contain tokens describing that the timers are active. At these nodes and also at initial nodes, we append *inner places.* For instance, Fig. 3.5 illustrates the inner places of the central station in which tokens rest to wait for further input events. In contrast to the queue places, these inner places will constitute the control states of the state machine. For instance, the token in the waiting decision *w2* of Fig. 3.5 means that a valid authentication result arrived and that the central node waits for the result of the authorization. (The join nodes do not have own places, as all their incoming flow originate from waiting decisions, which hold the token instead.) For the number of control states to be finite, the number of tokens in an inner place must be bounded. Moreover, to keep the state space small, we allow only one token in each inner place. This is not a limitation since tokens that would fill an inner place are stored in the unlimited queue places as discussed later. The set of control states for one state machine is then the powerset of the inner places.

We can construct a state machine transition by following the passing of the tokens between two stable token markings. The marking of the inner places before passing the tokens define the source state of the transition and the next stable marking refer to the target state. The token taken from a queue place models the input signal consumed by the transition. The activity nodes passed by the token are transformed in the following way: Call operation actions and send signal actions are simply copied into the effect of a transition. Decision nodes with guards are added to the transition and lead to different branches. A flow leaving the current activity partition is translated to a send signal action. Fork nodes duplicate tokens, to that the subsequent flows are executed in parallel. In the transition, this is mapped by executing their actions interleaved. For instance, the transition triggered by *PID* in *Central Station* in Fig. 3.4 simply executes first the action *prep1* and then *prep2.* Initial nodes emit tokens once the activity is started and are treated by the initial transition of a state machine.

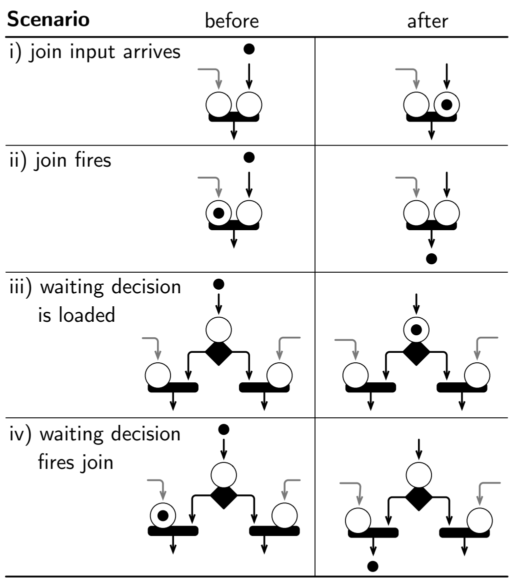

Scenario
before
after
i) join input arrives
ii) join fires
iii) waiting decision is loaded
iv) waiting decision fires join

Fig. 3.6: Rules for token transitions

A problem is the handling of joins. The passing of a token after all incoming flows arrived would result in a transition without a trigger event, violating constraints of our event-driven state machines. Therefore we use the token passing rules illustrated in Fig. 3.6. When a token arrives at a join and there are other incoming edges that do not yet offer a token (since their flow did not arrive yet), the token is stored and a new stable control state is reached *(i)*, awaiting the next event. If, however, the arriving token completes the join *(ii)*, the transition continues with its outgoing edge, and all tokens of the incoming edges are removed. Waiting decisions work similarly, but consider a set of subsequent join nodes. If none of these joins is ready, the decision is filled with a token *(iii)*, and a new stable state is reached. If one of the joins is ready, the transition continues at its outgoing edge, consuming the token from the decision node *(iv)*.

In addition to the events of signal reception resulting from the split control flows, explicit signal receptions contribute to the set of unbounded queue places. Furthermore, timers are sources of events. When a timeout occurs, a token is emitted on the timer's outgoing edge. The transition is then constructed in the same way as for signal receptions. Some events may lead to states describing that an inner place contains more than one token. For example, if the central station is connected to several local stations, a pid could arrive while another pid is under evaluation. In this case it might happen, that, after the central station received a valid *res1* and waits for *res2*, another valid *res1* is coming, which requires node *w2* to hold two tokens. To prevent this, we do not create a transition for flows leading to a marking with several tokens in an inner place, but defer the incoming event, which may proceed after the inner place is emptied.

3.5 The Transformation Algorithm

To realize the transformation from activities to state machines introduced above, we can proceed in quite different ways. For instance, one could perform a complete reachability analysis over all allowed token allocations in the previously introduced inner and queue places of an activity and create a transition for every step. The disadvantage is that, especially in highly concurrent systems, the number of reachable states is very large, rendering the approach not scalable. Another possibility would be to perform a purely syntactical analysis of an activity. Here, for each edge between places, a set of transitions is generated. Thereby, a separate transition is created for all states in which tokens are contained in the corresponding places. This algorithm is quite efficient since every inner place of the activity is checked only once, but will lead to a large number of transitions leaving unreachable states. To prevent these disadvantages, we follow an intermediate approach. Reflecting that for every activity partition a separate state machine is created, we perform a reachability analysis over the states of an activity partition only, which are constituted by its inner places. Thus, the number of reachable states is kept small. Starting from the initial state, for each reached state and every possible input signal a separate transition is created. As we handle each incoming signal in all control states, some of these transitions may be never fired (if their input signal cannot occur in the state). However, an unnecessary transition would simply result in a code fragment that is never executed. While this is not a real problem, nevertheless, we plan to eliminate these transitions using interface descriptions of the other partitions. These interface descriptions may be offered as part of the collaboration building blocks of a library.

In the following, we explain our algorithm in detail. Fig. 3.7 depicts the main loop (lines *7* to *27*). As in most reachability analysis algorithms, this loop guarantees that all reachable markings of an activity partition are analyzed. The markings yet to be checked are listed in the variable *reachable* while *visited* contains all markings which were already analyzed. In the initial part of the algorithm *(1..6)*, a new and empty state machine is created. Thereafter, the first marking to be checked, the initial transition of the state machine and the set of events to be received by the state machine are computed. As our algorithm creates a state machine transition for each pair of reachable marking and event (see Sect. 3.4), the loop contains a nested for-loop *(10..26)* cycling through all events. The for-loop contains two nested if-statements. The first one *(11..25)* is used to ignore events triggered by a timer which is not active in the current state. The second if-statement enables us to handle violations of the desired 1-boundedness property correctly. If the traversal of an edge in the checked activity would lead to two or more tokens in any inner place, the algorithm does not create a transition but defers the event in the current state *(13)*. Otherwise, a new transition is built in the else-statement *(15..23)*.

The transitions of a state machine are created by means of the recursive method *buildTransition (20)*, listed in Fig. 3.8 and 3.9. It considers the traver-

```
transform(ActivityPartition a) : StateMachine

1  var stm: StateMachine = new StateMachine()
2  var firstState: State = computeFirstState(a)
3  createInitialTransition(firstState, stm)
4  var events: Set of Event = computeEvents(a)
5  var visited: Set of State = $\emptyset$
6  var reachable: Set of State = {firstState}
7  while reachable $\neq \emptyset$ do
8      var current: State = reachable.removeFirst();
9      visited = visited $\cup$ {current}
10     for all e $\in$ events do
11         if $\neg$(e is timeout $\wedge$ timerActive(current)) then
12             if harmsBoundedness(current, e) then
13                 current.deferEvent(e)
14             else
15                 var t: Transition = new Transition(stm);
16                 t.setSource(current)
17                 t.setTrigger(e)
18                 var marking: long = getMarking(current)
19                 if e is timeout then marking = unsetTimer(e,marking)
20                 var Edge edge = retrieveEdge(e)
21                 var targets: Set of State
22                     = buildTransition(edge,marking,t,a,stm)
23                 reachable = reachable $\cup$ (targets / visited)
24             end if
25         end if
26     end for
27 end while
28 return stm.                                                   □
```

Fig. 3.7: Main control

sal of a token from one stable marking to another. For each edge part of the flow triggered by the event it is called recursively and builds the corresponding transition along the way. The method returns the set of stable states reached by the transition. It is a set, as a flow may lead to several distinct reachable states after a decision node. The returned states are used by the main loop to determine the reachable markings of the partition yet to be checked. The method contains an order of nested if-statements describing the behavior for each possible node in the analyzed activity edge. It returns if the edge leaves the partition *(2)*, reaches a join which cannot be fired in the current activity marking *(12)*, starts a timer *(17)*, arrives at a waiting decision in which none of the corresponding joins can be fired in the current marking *(50)*, or reaches a flow final resp. activity final node *(58, 62)*. In all these cases a new stable state is reached and the created transition can be completed. When another edge is

```
buildTransition(Edge edge, long c, Transition t, ActivityPartition a, StateMachine
stm) : Set of State

1  var node: Node = edge.getTarget()

2  if leavesPartition(edge,a) then
3      addSendSignalAction(t,edge)
4      var target: State = getState(c)
5      t.setTarget(target)
6      return {target}
7  else if node is join then
8      if canFire(join,c) then
9          var n: long = markingAfterJoinFired(c)
10         return buildTransition(outgoing(edge),n,t,a,stm)
11     else
12         var n: long = markingAfterJoinInputArrived(c)
13         var target: State = getState(n)
14         t.setTarget(target)
15         return {target}
16     end if
17 else if node is timer then
18     addSetTimerAction(t,node)
19     var n : long = markingAfterTimerSet(c)
20     var target: State = getState(n)
21     t.setTarget(target)
22     return {target}
23 else if node is send action then
24     addSendSignalAction(t,node)
25     var target: State = getState(c)
26     t.setTarget(target)
27     return buildTransition(outgoing(node),c,t,a,stm)
28 else if node is call operation action then
29     t.addEffect(node)
30     return buildTransition(outgoing(node),c,t,a,stm)
31 else if node is merge then
32     return buildTransition(outgoing(node),c,t,a,stm)f
```

Fig. 3.8: Method to build a transition (beginning)

reached, the transition is not yet complete and its building process has to be continued by a recursive call of *buildTransition*. These cases are a join which can be executed after being reached by a token on the analyzed edge *(10)*, a send action *(27)*, an operation call *(30)*, a merge *(32)*, a decision *(40)*, a waiting decision

```
33 else if node is decision then
34     var p: Pseudostate = new Pseudostate(stm,CHOICE)
35     var reachable: Set of State
36     for all $o \in$ node.outgoings() do
37         var t = new Transition(stm)
38         t.setSource(p)
39         t.setGuard(o.getGuard())
40         var r: Set of State = buildTransition(o,c,t,a,stm)
41         reachable = reachable $\cup$ r
42     end for
43     return reachable
   ───────────────
44 else if node is waiting decision then
45     for all $o \in$ node.outgoings() do
46         var join: Node = o.target;
47         if canFire(join, marking) then
48             return buildTransition(o,c,t,a,stm)
49         end if
50     end for    //no join could fire
51     var n : long = markingAfterDecisionSet(c)
52     var target: State = getState(n)
53     t.setTarget(target)
54     return {target}
   ───────────────
55 else if node is fork then
56     collectEffects(outgoings(node),t)
57     return computeForkedState(outgoings(node))
   ───────────────
58 else if node is flow final then
59     var target: State = getState(c)
60     t.setTarget(target)
61     return {target}
   ───────────────
62 else if node is activity final then
63     t.setTarget(new FinalState(stm))
64     return {}
65 end if                                                  □
```

Fig. 3.9: Method to build a transition (end)

from which a corresponding join can be fired after being reached by a token *(48)*, and a fork *(57)*. While most steps in creating a transition follow directly the ideas presented in Sect. 3.4, we will look now on the decisions and forks which are a little subtle. A decision leads to the addition of a choice pseudo state to the transition behind which more than one continuing transition fragments are added. This is done by the for-loop *(36..42)* which calls *buildTransition* for each of the choice's branches. The parallel emission of tokens at forks is addressed at *(55)*. As one state machine executes only one action at a time, we map parallel executing flows inside one activity partition to an interleaved execution, which is a correct refinement. This execution is computed by the method *collectEffects* which is not listed here for the sake of brevity.

3.6 Correctness of the Transformation

To verify that the algorithm carries out transformations in a correctness-preserving manner, we use the linear-time temporal logic cTLA [HK00] as a formalism which is based on Leslie Lamport's TLA [Lam02]. cTLA enables the description of resources and constraints in a process-like notion and provides a coupling structure based on conjoining actions (i.e., predicates on pairs of states describing sets of transitions). Refinement verifications are carried out as temporal logic implication proofs (cf. [Lam02]). As the semantics of activities is based on Petri-nets [Obj06], UML 2.0 activities can easily be expressed by cTLA processes as pointed out in [GH04]. An activity, basically, is a cTLA system description consisting of processes each describing a single activity partition. The variables of a process model its inner places while each queue place of a partition is described by a separate input queue.

For the state machines forming the input of our code generators, we defined a special dialect cTLA/e [KHB06] which describes the coupling between components by assigning a single input queue to each component. A state machine transition is specified by a cTLA action which reflects that the transition depends only on the current state and the first signal in the input queue. Moreover, each component contains an extra queue to handle deferred events. The refinement of specifications modeling activities to cTLA/e-based descriptions is carried out by a sequence of correctness-preserving refinement steps accompanied by cTLA/TLA implication proofs (cf. [Lam02]). For the sake of brevity, we do not give a thorough introduction to cTLA here and sketch the proof steps only briefly.

To verify formally that a state machine S derived from an activity partition A keeps all the functional properties state by A, we must perform by temporal logic deductions that the implication $S \Rightarrow A$ holds. According to Abadi and Lamport [AL91], this can be achieved by finding a so-called refinement mapping from the states of S to those of A. A refinement mapping takes into account that cTLA models applications as state transition systems. A system formula consists of an initial condition describing the set of initial states, cTLA actions which are predicates on a pair of a current state and a next state and model a set of state transitions each, and liveness properties expressed by fairness assumptions on actions which enforce that actions are eventually executed when they are consistently enabled. A refinement mapping fulfills the following properties:

– An initial state of S is mapped to an initial state of A.

- Each cTLA action of S is either mapped to an action of A or to a so-called stuttering step in which the mapped current and next states of A are identical.
- Each fairness assumption of A is provided by the fairness assumptions of S (i.e., if an action ψ of A is consistently enabled, the fairness assumptions of S enforce a state sequence in which eventually an action is carried out which is mapped to ψ).

In Sect. 3.4 we stated that the state space of an activity partition A is partly defined by its inner places which are placed before joins, at decision nodes, at initial nodes, and at timers. Moreover, it contains queue places which are situated at points where an incoming flow passes the partition border and on receive actions. The state space of a state machine is defined in [KHB06] and consists of the literal states of the state machine, an input queue, a defer queue, output queues for all connected state machines, and flags for each timer. Furthermore, activities may contain auxiliary variables which our algorithm directly maps to auxiliary variables of the corresponding state machines. Every auxiliary variable must be read and modified in one activity partition only. To outline the correctness of the algorithm, we introduce a mapping of the state space from S to that of A and sketch thereafter that it fulfills the refinement mapping properties:

- To find a mapping from S to the queue places of A, we have to consider the state machines S_n linked with S since the queue places describe the interaction between different system elements. At an activity partition, we have a separate queue place for every signal type st while in the corresponding state machine, we have central queues for all signals. Moreover, in the activity we do not distinguish if a signal is still at the side of the outgoing partition, already in the incoming partition, or deferred. Reflecting these properties, we map all signals s of type st, which are either in the output queue of a neighboring state machine S_n, in the input queue of S, or in its defer queue, to the queue place qp_{st} for st in A:

$$\begin{aligned}\forall st : qp_{st} = \{s| & \wedge s.type = st \\ & \wedge s \in inputQ_S \cup deferQ_S \cup \bigcup_{S_n \in Neighbors_S} S_n.outputQ_S\}\end{aligned} \tag{3.1}$$

- A mapping of S to the inner places of A located at joins, decision nodes, and initial nodes has to consider that we use 1-boundedness in the inner places ip and that the algorithm creates the states of S as a string of flags fl_{ip} each being set to 0 if the corresponding inner place ip is empty and to 1 if ip contains a token to:

$$\forall ip : ip = \text{IF } fl_{ip} = 1 \text{ THEN } \{to\} \text{ ELSE } \{\}$$

- To find a mapping from S to the inner places of A describing a timer is a little more complex. Indeed, the algorithm adds also a flag fl_t for each timer t in A to the state representation in S. Nevertheless, to find a decent mapping, one has to consider the handling of timers in state machines. When a timer expires, it creates a signal which is attached to the local input queue. Thus, we must map both the states of S in which the flag fl_t of timer t is enabled and in which a signal s_t caused by t is in the input or defer queue to a setting in A where a token *to* is on the inner place ip_t of t. That is expressed by the mapping listed below:

$$\forall ip_t : ip_t = \text{IF } fl_t = 1 \vee s_t \in inputQ_S \cup deferQ_S \text{ THEN } \{to\} \text{ ELSE } \{\}$$

– The mapping from the auxiliary variables from S to those of A is the identity function.

In the first step of the proof that the function listed above fulfills the refinement mapping properties, we have to verify that the initial state of S is mapped to that of A. Initially, the queue places in A are empty while the input, output, and defer queues of S do not contain elements as well. Thus, the mapping of the queue places fulfills the property. The inner places of A are empty except those located at an initial node and the algorithm maps this token placement to the initial state of S in which just the flags representing the inner places of the initial nodes are set to 1. Since the auxiliary variables of S and A contain the same initial settings, therefore, the initial state of S is mapped to the initial state of A.

Next, we prove that every cTLA action in the model of the state machine S is mapped either to a cTLA action of the activity partition A or to a stuttering step. As introduced in [KHB06], the model of S contains different kinds of actions. One type describes the transitions of S and for each transition tr_f, a cTLA action ϕ_{tr_f} is defined. The algorithm creates tr_f only if a flow f exists modifying the token setting of A. In the following, we state a number of properties preserved by the algorithm in the creation of the corresponding transition tr_f which are used for the refinement proof:

1. A transition tr_f is only created if in its source state all flags fl_{ip} representing the inner places ip of f, which must contain tokens to enable the execution of f, are set to 1.
2. The algorithm creates tr_f only for a flow f if the execution of f does not violate the 1-boundedness property of the inner places in A.
3. If the queue place in f, from which a token is removed, has the type *st*, tr_f is only triggered if *st* is at the front of the input queue.
4. By executing a transition tr_f which does not leave an initial state, the signal at the front of the input queue is consumed.
5. A transition tr_f consuming a signal from the input queue, which was created by a timer, is generated if the corresponding flow f starts at an inner place describing a timer node.
6. The target states of tr_f are generated by starting with the source state and resetting the flags representing inner places, from which tokens were removed, to 0 while those with a new token are set to 1.

7. If in the flow f a token crosses the border to a partition A_n or heads to a send action with destination A_n, tr_f puts a send signal into the output queue devoted to the state machine S_n realizing A_n.
8. A call operation action passed in f is reflected by adding its code to tr_f. Here, we demand that an auxiliary variable may be modified only once in f and, in consequence, in tr_f.

Assuming that ϕ_{tr_f} is the cTLA action modeling tr_f and ψ_f those of the flow f, these properties are sufficient to prove the implication $\phi_{tr_f} \Rightarrow \psi_f$. By the first three properties, we can assure that the enabling condition of ϕ_{tr_f} implies that of ψ_f since according to the mapping all necessary tokens are set (1), the 1-boundedness after carrying out f is preserved (2), and the queue place, from which f leaves, contains an element (3).

The other properties are used to verify that the effects of ϕ_{tr_f} are correctly mapped to those of ψ_f. The elimination of a signal of type st from the input queue is mapped to the removal of a token from the queue place st (4). In addition, if tr_f consumes a signal s_t created by a timer from the input queue, s_t is mapped to a flow f removing a token from the corresponding timer node (5). We can further verify that tr_f is a correct realization of the token flow between the inner places in f (6). The delivery of a signal s to an adjacent state machine S_n does not spoil the corresponding mapping of S_n to a neighboring activity partition A_n since s is added to an incoming queue place of A_n if S puts it to its output queue devoted to S_n (7). Finally, it is guaranteed that the auxiliary variables are correctly mapped (8). It is not difficult to verify that these properties imply that the mapping listed above maps ϕ_{tr_f} to ψ_f which is omitted, however, for brevity.

Other cTLA actions in S specify the execution of timers and the addition of timer signals to the input queue, model the deferral of a signal by transferring it from the input to the defer queue, and describe the transfer of signals from the neighbor's output queue to the own input queue. It can be easily shown that these actions lead to stuttering steps in A.

In the third step, we have to verify that the fairness assumptions of the actions ψ_f describing the flows in A are kept. The algorithm guarantees that for every token placement in the inner nodes of A enabling a flow f, a transition tr_f is generated implementing f. Thus, with respect to the first two properties listed above, an action tr_f is enabled whenever f can fire. The only impeding condition is the third property since tr_f may only be executed if the signal s consumed by it is at the first place of the input queue. According to the mapping, however, the cTLA action ψ_f specifying f can be enabled if s is either in the output queue of the neighboring state machine S_n or in any place on the input or defer queues of S. Thus, we must verify that s is eventually being moved to the front of the input queue where it will remain consistently until an action ϕ_{tr_f} is executed. If s is still in the output buffer of S_n, it will be moved to the end of the input buffer of S by the fair[2] action modeling the transmission from S_n to

[2]In [HK00] we established that liveness can only be guaranteed in a distributed system if transmitted messages are eventually being delivered. This is expressed by the fairness

S. Since signals before s in the input resp. defer queue are either continuously being deferred or eventually being consumed, s will eventually be consistently at the front of the input queue. If f is not enabled, s may be deferred itself but is brought back to the front of the input queue by other transitions. As there is only a finite number of transitions tr_f modeling f, in consequence, one of those will be consistently being enabled if f can be triggered infinitely often as well. Due to the fairness assumption of the corresponding cTLA action ϕ_{tr_f} it will be eventually fired which, because of the mapping, causes also the triggering of ψ_f.

Thus, we could verify that the mapping listed above is a refine mapping. According to [AL91], we thereby proved that the state machine S together with its neighboring state machines S_n produced by the algorithm is a correct implementation of the activity partition A. Since this proof can be carried out for all partitions of the activity, we established that the algorithm transforms activities to state machines in a correct way.

3.7 Related Work

To our best knowledge, the algorithm presented here is the first one that directly transforms UML 2.0 activity diagrams into the executable state machines described above. Our work is related to that of Eshuis on model checking of activity diagrams [Esh06], in which activity diagrams are transformed into the input language of NuSMV, a symbolic model verifier [CCG$^+$02]. We could not adapt this algorithm for our work, since, as discussed in Sect. 3.5, syntactical algorithms cause in our field of application a high number of considered unreachable states. To execute activity graphs, Eshuis and Wieringa [EW01] describe an algorithm for an event router to coordinate the behavior of components. Aiming at workflow systems, their execution differs from ours as it assumes a centralized architecture and the activity is considered as a whole, rather than splitting up the activity into its partitions and creating distributed state machines as we do.

There is a number of approaches that take scenario descriptions based on sequence diagrams (like MSCs or UML sequence diagrams) to synthesize state machines [MZ99, WS00, KGSB99, UKM03]. While the resulting state machines are similarly executable as the ones we described, the input of these synthesizers in form of sequence diagrams differs from activity diagrams. Sequence diagrams often specify only a set of scenarios rather than a complete behavior, which may lead to behaviors that are not expressed explicitly. They focus on the interactions and identify signals. In contrast, activities focus on the operations and decisions that have to be performed by its participants, and our algorithm generates the necessary interactions in form of signal transmissions automatically.

Use case maps (UCM, [BC96]) offer a notation that is close to that of UML activities, as they also allow the specification of behavior in terms of causal paths that may involve several components. Yong He et al. conducted an experiment [HAW03] in which a specification expressed by use case maps was

assumption on the action specifying the transmission.

transformed into message sequence charts. These, in turn, were transformed into executable SDL specifications using the tool MSC2SDL [MZ99]. Similar to that, Castejón [Cas05] outlines an algorithm that takes specifications in UCM and UML 2.0 collaborations to generate state machines from sequence diagram fragments contained in the collaborations.

3.8 Concluding Remarks

We described an algorithm that transforms UML 2.0 activities into a UML 2.0 state machines, from which we can easily generate efficiently executable code. The algorithm is implemented in Java and integrated into our Eclipse-based tool suite, so that we now have a complete automated development process from collaborative specifications based on activities to implementations on various platforms. As input and output we use models stored in the Java UML 2.0 repository from the Eclipse UML2 project. The algorithm does not construct an intermediate graph, but only UML model elements that are part of the desired output state machines, so that it is efficient with respect to the memory needed. The time for the transformation of the presented example is negligible; the state machines appear practically instantly. Moreover, we expect the algorithm to scale well also for more complex systems, as the increased complexity of a system leads more to a higher number of partitions than to more complex ones causing only a linear increase.

This work describes a step of a more comprehensive engineering approach for the creation of interactive services by correctness-preserving design steps. Initially, a service specification is composed from various abstract collaborations that, to a large extent, can be obtained from domain-specific libraries. Such abstract collaborations are often quite simple and can also be understood by customers, who are not experts in software technology but want to focus on their actual business. In succeeding steps, such abstract specifications are incrementally refined until the specification has a degree of detail that enables direct translation to software. Due to the algorithm, we are now able to perform these refining design steps entirely in the collaboration-oriented perspective. As pointed out in [KH06], for this purpose we can use the activities with their convenient properties as reusable building blocks. ■

Bibliography

[AL91] Martín Abadi and Leslie Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[BC96] Ray J. A. Buhr and Ron S. Casselman. *Use Case Maps for Object-Oriented Systems.* Prentice-Hall, Inc., 1996.

[BH93] Rolv Bræk and Øystein Haugen. *Engineering Real Time Systems: An Object-Oriented Methodology Using SDL*. The BCS Practitioner Series. Prentice Hall, 1993.

[Bræ79] Rolv Bræk. Unified System Modelling and Implementation. In *International Switching Symposium*, pages 1180–1187, Paris, France, May 1979.

[BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement.* Springer, 2001.

[Cas05] Humberto N. Castejón. Synthesizing State-machine Behaviour from UML Collaborations and Use Case Maps. In Andreas Prinz, Rick Reed, and Jeanne Reed, editors, *12th International SDL Forum, Grimstad, Norway*, volume 3530 of *Lecture Notes in Computer Science*. Springer, 2005.

[CCG+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002), Copenhagen, Denmark*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[Esh06] Rik Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Transactions on Software Engineering and Methodology*, 15(1):1–38, 2006.

[EW01] Rik Eshuis and Roel Wieringa. An Execution Algorithm for UML Activity Graphs. In *UML'01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 47–61, London, UK, 2001. Springer-Verlag.

[FB00] Jacqueline Floch and Rolv Bræk. Towards Dynamic Composition of Hybrid Communication Services. In *SMARTNET '00: Proceedings of the IFIP TC6 WG6.7 Sixth International Conference on Intelligence in Networks*, pages 73–92, Deventer, The Netherlands, 2000. Kluwer, B.V.

[GH04] Günter Graw and Peter Herrmann. Transformation and Verification of Executable UML Models. *Electronic Notes on Theoretical Computer Science*, 101:3–24, 2004.

[HAW03] Yong He, Daniel Amyot, and Alan W. Williams. Synthesizing SDL from Use Case Maps: An Experiment. In Rick Reed and Jeanne

Reed, editors, *Proceedings of the 11th SDL Forum, Stuttgart, Germany, 2003*, volume 2708 of *Lecture Notes in Computer Science*, pages 117–136. Springer, 2003.

[HK00] Peter Herrmann and Heiko Krumm. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337, 2000.

[KGSB99] Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to Statecharts, 1999.

[KH06] Frank Alexander Kraemer and Peter Herrmann. Service Specification by Composition of Collaborations — An Example. In *Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology)*, pages 129–133. IEEE Computer Society, 2006. 2nd International Workshop on Service Composition (Sercomp), Hong Kong.

[KHB06] Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk. Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. In R. Meersmann and Z. Tari, editors, *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), 2006, Montpellier, France*, volume 4276 of *Lecture Notes in Computer Science*, pages 1613–1632. Springer–Verlag Heidelberg, 2006.

[Kra03] Frank Alexander Kraemer. Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart, 2003.

[Lam02] Leslie Lamport. *Specifying Systems*. Addison-Wesley, 2002.

[MZ99] Nikolai Mansurov and D. Zhukov. Automatic Synthesis of SDL Models in Use Case Methodology. In Rachida Dssouli, Gregor von Bochmann, and Yair Lahav, editors, *SDL Forum*, pages 225–240. Elsevier, 1999.

[Obj06] Object Management Group. Unified Modeling Language: Superstructure, version 2.1, April 2006. ptc/2006-04-02.

[RGG01] Frank Rößler, Birgit Geppert, and Reinhard Gotzhein. Collaboration-Based Design of SDL Systems. In *Proceedings of the 10th International SDL Forum Copenhagen on Meeting UML*, pages 72–89. Springer-Verlag, 2001.

[SCKB05] Richard Sanders, Humberto N. Castejón, Frank Alexander Kraemer, and Rolv Bræk. Using UML 2.0 Collaborations for Compositional Service Specification. In *ACM / IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, 2005.

[UKM03] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of Behavioral Models from Scenarios. *IEEE Trans. Softw. Eng.*, 29(2):99–115, 2003.

[WS00] Jon Whittle and Johann Schumann. Generating Statechart Designs from Scenarios. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 314–323, New York, 2000. ACM Press.

PAPER
FOUR

SYNTHESIZING COMPONENTS WITH SESSIONS FROM COLLABORATION-ORIENTED SERVICE SPECIFICATIONS

Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann

Presented at the 13th SDL Forum, September 18 – 21, 2007, Paris, France.

Published in *SDL 2007*, Volume 4745 of Lecture Notes of Computer Science, p. 166–185, Springer, 2007.

The original publication is available at www.springerlink.com via `http://dx.doi.org/10.1007/978-3-540-74984-4_11`.

Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications

Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann

Abstract. A fundamental problem in the area of service engineering is the so-called cross-cutting nature of services, i.e., that service behavior results from a collaboration of partial component behaviors. We present an approach for model-based service engineering, in which system component models are derived automatically from collaboration models. These are specifications of sub-services incorporating both the local behavior of the components and the necessary inter-component communication. The collaborations are expressed in a compact and self-contained way by UML collaborations and activities. The UML activities can express service compositions precisely, so that components may be derived automatically by means of a model transformation. In this paper, we focus on the important issue of how to coordinate and compose collaborations that are executed with several sessions at the same time. We introduce an extension to activities for session selection. Moreover, we explain how this composition is mapped onto the components and how it can be translated into executable code.

4.1 Introduction

In its early days, reactive software was mainly structured into activities that could be scheduled in order to satisfy real-time requirements. As a result, the rather complex and stateful behavior associated with each individual service session and resource usage was fragmented and the overall behavior was often difficult to grasp, resulting in quality errors and costly maintenance.

The situation was considerably improved by the introduction of state machines modeling stateful behavior combined with *object-based* and later *object-oriented* structuring. By representing individual resources and sessions as state machines, their behavior could be explicitly and completely defined. This principle helped to substantially improve quality and modularity, and therefore became a widespread approach. It also facilitates the separation between abstract behavior specifications and implementation, and enabled model-driven development in which executable code is generated automatically from state machines. SDL [ITU02] was developed as a language to support this approach and, considering its adoption and support, we must say that it has been successful at it.

However, there is a fundamental problem. Service behavior is normally distributed among several collaborating objects, while objects take part in several different services. By structuring according to objects, the behavior of each individual object can be defined precisely and completely, while the behavior of a service is distributed across the objects. This is often referred to as the "cross-cutting" nature of services [KM03, RGG01, FK01], and is one of the underlying

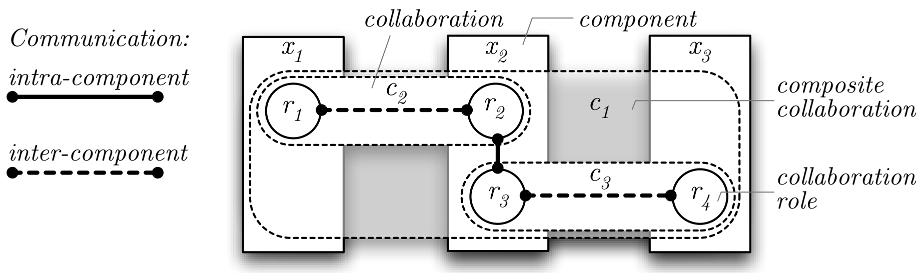


Fig. 4.1: Relationships between components and collaborations

reasons why compositional service engineering is such a challenge. Fundamentally, the behavior of services is composed from partial object behaviors, while object behaviors are composed from partial service behaviors.

A promising step forward to solve this problem is to adopt a *collaboration-oriented* approach, where the main structuring units are formal specifications of services containing both the partial object behavior and the interactions between the objects needed to fulfill the service. These specifications are called collaborations. Albeit many of the underlying ideas have been around for a long time [RWL95, RAB^{+}92], the new concept of UML 2.0 collaborations [Obj07] provides a modeling framework that opens many interesting opportunities not fully utilized yet. First of all, collaborations model the concept of a service very nicely. They define a structure of partial object behaviors, the collaboration roles, and enable a precise definition of the service behavior. They also provide a way to compose services by means of collaboration uses and role bindings.

Figure 4.1 shows a coarse system architecture illustrating the relations between collaborations and objects (referred to as components in the following). A service is delivered by the joint behavior of the components x_1 to x_3, which may be physically distributed. The service described by collaboration c_1 can be composed from the two sub-services modeled by collaborations c_2 and c_3. The necessary partial object behavior used to realize the collaborations is represented by so-called collaboration roles r_1 to r_4. Note how the collaborations cut across the components and define inter-component behavior. Orthogonal to this, component behavior is defined by composition of collaboration roles. Communication between components is assumed to be based on asynchronous message passing only (cf. [BF04]), while communication within one component may also use shared variables and synchronously executed actions (i.e., an event in one collaboration can cause actions in another collaboration).

We have found that collaboration-oriented decomposition tends to result in sub-collaborations corresponding to interfaces and service features [SBvBA05] with behavior of limited complexity that may be defined completely and be reused in many different services. This simplifies the task of defining inter-component behavior and separates it from the intra-component composition. It has been shown in [CB06b, CB06a] that collaborations also provide a basis for analysis and removal of errors at a higher level of abstraction than detailed interactions.

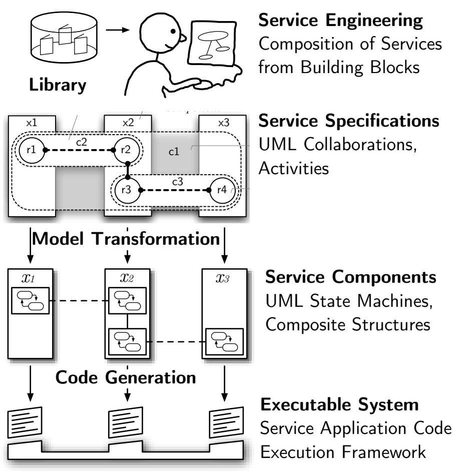


Fig. 4.2: Service Engineering Approach

A well established approach is to model "horizontal" collaborative behavior using MSCs or UML sequence diagrams. They provide the desired overview, but will normally not be used to define the complete behavior. In this paper, we present our approach (see also [KH07, KH06]) in which the complete behavior of collaborations is defined using UML activity diagrams. We offer an extension to UML that enables to compose also behavior that is executed simultaneously in several sessions. This enables a complete and precise definition of the inter-component behavior of each collaboration as well as the intra-component behavior composition of collaborations, without the need to specify interaction details. The approach enables an automatic synthesis of component behaviors in the form of state machines from which executable code is automatically generated, as illustrated in Figure 4.2. By defining the semantics of activities and state machines using the temporal logic cTLA [HK00], we are able to verify by formal implication proofs that the transformations of the collaboration-oriented models to the state machines are correct (see [KH07]). This formal aspect, however, is not the focus of this paper. In the following we first introduce the collaboration-oriented specification approach by means of an example, and show how multiple session instances can be coordinated. Afterwards, we describe the transformation from collaboration to component behavior.

4.2 Collaborations

In Fig. 4.3 we introduce a taxi control system. Several taxis are connected to a control center, and update their status (*busy* or *free*) and their current position. Operators accept tour orders from customers via telephone. These orders are processed by the control center which sends out tour requests to the taxis. Taxis may also accept customers directly from the street, which is reported to the

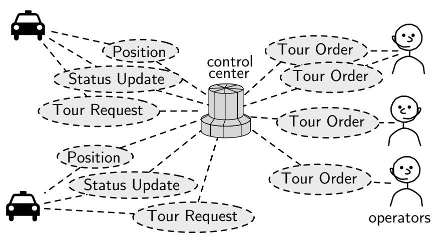


Fig. 4.3: Illustration of the system

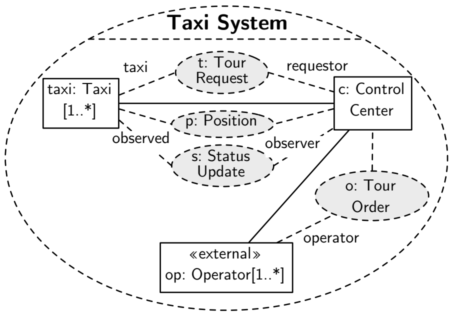


Fig. 4.4: UML Collaboration

control center by a status update to *busy*. Fig. 4.4 defines this as a UML 2.0 collaboration. Participants in the service are represented by collaboration roles *taxi*, *c*, and *op*. For the taxis and the control center we will later generate components. The operators are part of the environment and therefore labeled as ≪external≫. The control center *c* has a default multiplicity of one, while there can be many taxis and operators in the system, denoted by multiplicity [1..*]. Between the roles, collaboration uses denote the occurrence of behavior: taxis and control center are interacting with collaborations *Status Update*, *Position* and *Tour Request*, while the operators are cooperating with the control center by means of collaboration *Tour Order*. In this way, the entire service, represented as collaboration *Taxi System*, is composed from sub-services.

4.2.1 Describing Behavior of Collaborations

Besides being a so-called *UML structured classifier* with parts and connectors as shown in Fig. 4.4, a collaboration is also a *behaviored classifier* and may as such have behavior attached, for example state machines, sequence diagrams or activities. As mentioned in the introduction, we use activity diagrams. They present complete behavior in a quite compact form and may define connections to other behaviors via input and output pins. In [KH06, HK07] we showed how

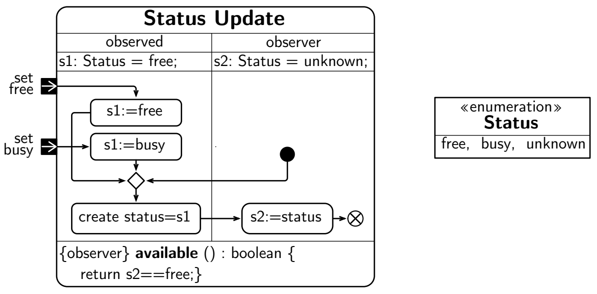


Fig. 4.5: Activity for Status Update

service models can be easily composed of reusable building blocks expressed as activities.

The activity *Status Update* (Fig. 4.5) describes the behavior of the corresponding collaboration. It has one partition for each collaboration role: *observer* and *observed*. As depicted in Fig. 4.4, these roles are bound to *c* and *taxi*, so that the observer is the control center that observes a taxi. A pleasant feature of our approach is that we can first study and specify the behavior of the control center towards *one* taxi and we later compose this behavior, so that the control center may handle several taxis.

Activities base their semantics on token flow [Obj07, p.319]. Hence, a token is placed into the initial node of the observer in Fig. 4.5 when the system starts. The token moves through the merge node, upon which the observed party sends its current status to the observer. The observer then updates its local variable *s2*. From then on, the taxi pushes any status change to the control center. As these changes depend on events external to this collaboration, they are expressed by the parameter nodes *set free* and *set busy*. These are *streaming* nodes through which tokens may pass while the activity is ongoing. Later, the parameter nodes (represented by corresponding pins on call behavior actions) will be used to couple the *status update* collaboration with the other collaborations. In addition, we defined an operation *available* for the activity that we will later use to access the status of a taxi from the control center. As this operation accesses variable *s2* localized in the observer, we use the constraint {*observer*} to mark that it may only be accessed from the side of the observer. The collaboration *Position* (not shown) works similarly by notifying the observer about the current geographical position.

The collaboration *Tour Request* depicted in Fig. 4.6 models the process of notifying a taxi about a tour. It is started via parameter node *request tour*, which starts timer *t* and places a token in waiting decision node *w*. A waiting decision node is the extension of a decision node with the difference that it may hold a token similar to an initial node, as defined in [KH07]. *w* is used in combination with join nodes *j1* and *j2* to explicitly model the race between the acceptance of the tour by the driver and the timeout mechanism. Another flow is forwarded to the taxi which first checks its status. This is necessary as the taxi can in fact be busy even if it was available when the requestor started. This is due to the

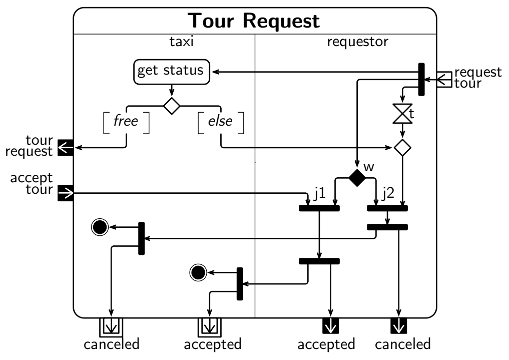


Fig. 4.6: Activity for Tour Request

inevitable delay of signals between the distributed components, so that the taxi may have accepted a customer from the street while a request is on its way. In general, the flows between the control center and the taxi (as well as all other flows crossing partitions) are buffered. We describe this in a so-called execution profile (see [Obj07, p. 321]) for our service specifications [Kra08] and model it by implicit queue places, as described in [KH07]. If the taxi is still free, the control flow is handed over to some external control not part of this collaboration. If the taxi driver accepts the tour, the control flow returns, and a token is offered to join node *j1*. If *w* still has its token, *j1* can fire, emit a token on *accepted* on the requestor side, and then terminate the collaboration on the taxi side with an activity final node and output node *accepted*.[1] In case the taxi turned busy or a timeout occurs, a token is offered to *j2*. It fires if *w* still has its token, so that the collaboration first notifies the requestor upon the cancelation and then terminates the collaboration on the taxi side.

Note that the events *accept tour* and the timeout may both happen, as they are initiated by different parties. This is a so-called mixed initiative [Flo03] that must be resolved to prevent erroneous behavior in which one side accepts the request while the other one considers the request as canceled. The taxi therefore sends the acceptance of a tour first to the requestor and waits for a confirmation; if the timer expired in the meantime, the acceptance is intercepted in *j1* and the collaboration terminates consistently with *canceled* on both sides.

4.2.2 Composing Collaborations with Activities

To generate state machines, components and finally the executable code for the system components, the structural information about how the collaborations are composed (as shown in Fig. 4.4) is not sufficient. In fact, we need to specify in detail how the different events of collaborations are coupled so that the desired overall behavior is obtained. For this purpose we use UML activities as well,

[1]As this ending is alternative to the cancelation of a tour request, it must be expressed by its own UML parameter set, denoted by the additional box around the node.

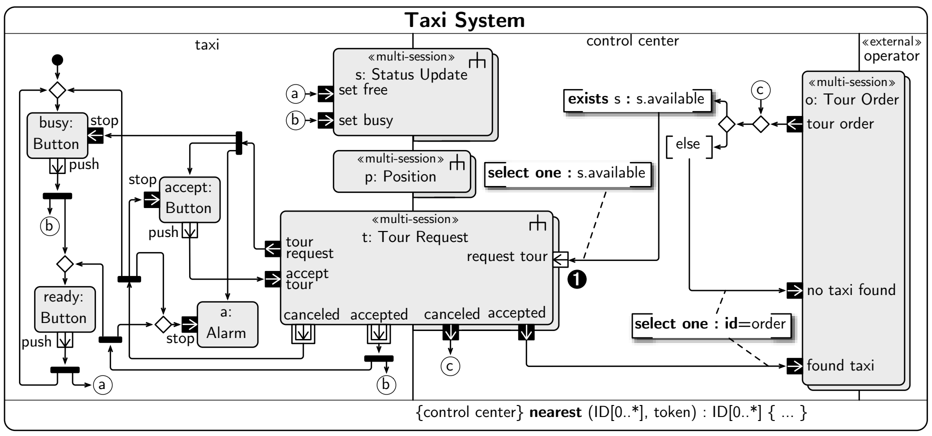


Fig. 4.7: The Taxi System Activity

as they allow us to specify *the coordination of executions of subordinate behaviors* [Obj07, p. 318]. Using call behavior actions, an activity can refer to other activities. Like this, the activity of a composite collaboration may refer to the activities of its sub-collaborations and specify how they are coordinated.

Fig. 4.7 shows the activity for the composed taxi system. Again, each collaboration role is presented by its own activity partition. As the taxi system collaboration is composed from several other collaborations, the activity refers to them via the call behavior actions *s*, *p*, *t* and *o*. Let us first focus on the partition for the taxi on the left hand side. It describes the local coupling between the collaborations a taxi participates in, including some additional logic for the user interface of the taxi, modeled as activities for three buttons and an alarm device that have been fetched from our library of reusable building blocks [Kra07]. When the taxi partition starts, button *busy* is activated. The driver presses it once a customer from the street orders a tour whereupon the button emits a token at exit *push*. This updates the status of the taxi to *busy* by coupling *push* of the busy button with *set busy* of the status collaboration.[2] In addition, button *ready* is activated to signal the termination of the tour by the driver. As the taxi participates in the collaboration *Tour Request* (represented by the call behavior action *t*), it must also handle the event when a tour request arrives from the control center, which is accessible through the output pin *tour request* of *t*. This event triggers the deactivation of the busy button, and activates the accept button as well as an alarm to notify the driver. The accept button, which is pushed if the driver accepts, notifies the collaboration *t*. Depending on the final outcome of the tour request collaboration (it may still be aborted by a timeout), either the ready button is activated and the status is changed to *busy*, or the taxi remains available and the busy button is activated again. The position collaboration needs no coupling, as it constantly sends the position independently of the other behaviors.

[2]For presentation reasons, this flow is segmented graphically by connector *b*.

4.3 Multiple Behavior Instances and Sessions

From the viewpoint of one taxi, there is exactly one collaboration session for each of the three collaboration uses s, p, t towards the control center. This can be handled easily with the UML activities in their standard form. The control center, on the other hand, has to maintain these sessions with each of the taxi cars. From its viewpoint, several instances of each of the collaboration uses s, p and t are executed at the same time; one instance for each taxi. Moreover, the tour order collaboration not only has to be executed concurrently towards several operators, but each operator may also request new tours while others are being processed. From the viewpoint of the control center, the collaborations it participates in, are what we call *multi-session* collaborations. We express this by applying a stereotype ≪multi-session≫ to the call behavior actions and represent it graphically by a shadow-like border in those partitions where sessions are multiple.[3] Consequently, the call behavior actions (resp. sub-collaborations) s, t, and p in Fig. 4.7 have a shadow within the *control center* partition, while o is multiple both in the control center and the operators.[4]

This raises the question about how the different instances of collaborations may be distinguished and coordinated, so that the desired overall system behavior is obtained. A selection of sessions must take place whenever a token enters a multi-session sub-collaboration (as for example via the pin at ❶). While in some cases we may want to address all of the sessions, in other ones we like to select only a subset or one particular session. The UML standard, however, does not elaborate this matter but instead forbids streaming nodes on reentrant behaviors completely, as it *is ambiguous which execution should receive streaming tokens* [Obj07, p. 398]. This is too restrictive, as most systems exhibit patterns with several executions going on at a time, that possibly need coordination. We therefore added the new operators **select** and **exists** to our execution profile.

4.3.1 Identification of Session Instances

First of all, the different sessions must be distinguished at runtime. This resembles the well-known session pattern (see for example [Ris01, p. 191]) that is found in client/server communication, where the server has some kind of identifier to distinguish different sessions. Accordingly, each collaboration session has an ID. For collaborations having one session instance for a specific participant, the session ID can be chosen to be identical to that of the participant. For example, we can use the ID of the taxi to identify the session instances of the *Tour Request*, *Status Update* and *Position* collaboration. This is similar to SDL, in which a process identifier *pid* of a communication partner is often used to refer to a session. If there can be more than one session per communication partner (the control center can for instance have several ongoing tour orders from the

[3]Technically, the corresponding partitions are stored as a property of the stereotype.

[4]In this paper we focus on the partitions *taxi* and *control center* and do not further look into the *operator* partition.

```
 select  :=  'select' mod ':' [{filter}] ['/' {filter} ].
 exists  :=  'exists' name ':' filter ['/' {filter} ].
    mod  :=  'one' | 'all'.
 filter  :=  name | 'self' | 'active' | 'id=' variable.
```

Fig. 4.8: EBNF for **select** and **exists**

same operator) any other unique identifier can be used; for collaboration *Tour Order* we can use a unique order number.

4.3.2 Choosing Session Instances with select

When an operator accepts an order from a customer, a token leaves the output pin *tour order* of *o* in Fig. 4.7. Let us ignore for the moment the decision and assume it takes the upper branch, towards input pin *request tour* of *t* at ❶. At this point we have to specify into which session instance of *t* the token should enter. We do this by attaching an expression as guard to the edge entering the input pin. If we would like to select all instances (by duplicating the token), we could write **select all**, resulting in an alarm in each taxi, whether busy or free, which is not desired. Instead, we would like to select only one of the free taxis. This means, we want to access properties of the *s: Status Update* sessions. As collaboration uses *s* and *t* have the same set of IDs, we would like to obtain an ID of *s* for which the status is free. To enable the control center to check the status of its taxis, we defined in the activity *Status Update* (Fig. 4.5) a boolean operation *available* which is executable from the observer side. This operation is used in the select statement. As there may be more than one free taxi, we further specify by adding the keyword **one** that only one of them should eventually be selected. The entire statement is then

select one : s.available.

If none of the taxis is free, no session is selected and the token flow simply stops. We describe later how this situation is ruled out by an alternative behavior using the decision node. If a tour request is canceled, another taxi can be contacted (via connector *c*) by iterating a new tour request.

Once the selected taxi accepts the tour, a token leaves output pin *accepted* and enters *o: Tour Order*. Here we have to select again which of the instances should be chosen. As they are distinguished by the order number, we leave this number as attribute *order* inside the token,[5] and extract it by writing

select one : id=order.

[5]This implies an UML object flow instead of a simple control flow, which we do not show here to keep the diagrams easier to comprehend.

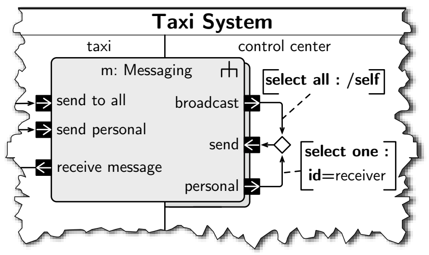


Fig. 4.9: Messaging service extension

The complete EBNF definition for session selection and existence is given in Fig. 4.8. It allows specifying several filters (e.g., *available*) that are applied in the order of their listing. In this way, we may flexibly use a sequence of filters, for example to call the taxi that is closest to the street address. In this case we would introduce a filter *nearest* which considers the location of the taxis provided by collaboration p and computes the taxi which is closest to the customers position. As we still want to select only free taxis, we can apply the available filter before, and write **select one :** s.available nearest, so that an ID has to pass both filters.

To study another form of session selection, we extend the system with a messaging service, where taxi drivers may send messages to each other; either to a specific taxi or to all taxis. Parts of this addition are shown in Fig. 4.9. Messages are sent via the control center, which maintains one instance of a collaboration *Messaging* with each taxi. As we attach the select statement to the incoming edges and not the nodes directly, a node may be entered with different selection strategies, combined by a merge node. Personal messages arrive from a taxi at pin *personal* and are forwarded by the ID stored as *receiver*, with the known selection statement. Broadcast messages are sent to all other sessions, except the session sending the message, expressed by **select all : /self**. The slash allows to specify negative filters for exclusion. (If for any reason drivers should send broadcasts just to free taxis, we would write **select all :** s.available **/self**.)

4.3.3 Reflecting on Sessions with **exists**

In some cases we have to reason about the status of certain sessions. For example, before we process a request from the tour order collaboration, we check if there are any free taxis available at all. We do this with the operator **exists** that returns a boolean value that can be the guard in a decision. In Fig. 4.7, we include therefore **exists** s **:** s.available, where s.available denotes the filter introduced above. Thus, in the example, the selection at ❶ is only reached if at least one taxi is free. If we want to make a decision depending on the fact whether there are any currently ongoing collaboration sessions (which have an active token flow) we may use the standard filter **active**.

4.3.4 Modeling of Filters

A filter is modeled as an UML operation. Boolean filters only considering one session can be defined as part of the activity describing the collaboration (like *available* in Fig. 4.5). Filters that need to consider an entire set of sessions or combine data from different collaborations are defined as part of the surrounding activity, such as the filter *nearest*. In contrast to the boolean filter *available*, *nearest* receives and returns an entire set of IDs, from which it can determine the one with the minimal distance to the address given by the token. The address is contained in the token, which is handed over to the filter by the parameter *token*. In principle, the body of operations may be expressed as any kind of UML behavior; in our current tool we use Java, embedded in a language-specific UML *OpaqueBehavior* [Obj07, p. 446], since our code generators create Java code.

4.4 Mapping to the Component Model and Implementation

In the following, we will discuss how the collaboration models are transformed into the executable component model of our approach. After introducing the component model, we explain the translation of single-session behavior and thereafter the mapping of multi-session behavior to state machines.

4.4.1 Component Model

Our component model is based on UML 2.0 state machines and composite structures. In [KHB06] we presented an UML profile with constraints ensuring that state machines can be implemented efficiently on different platforms. The internals of such *executable* state machines are similar to SDL processes. They communicate by sending signals, and transitions are triggered by either the reception of a signal or the expiration of a timer. Transitions do not block, so that they can be executed in one run-to-completion step without waiting.

We extend this model with *components* that may contain a number of state machines. Such system components are described by UML classes, and contain one dedicated state machine describing the so-called *classifier behavior*. This state machine typically manages the lifecycle of the component as well as stateless requests arriving from other components, as we shall see later. In addition, a system component can contain further state machines. These are modeled as UML *parts* owned by the structured classifier and have a type referring to a state machine. In contrast to the static state machine expressed by the *classifier behavior*, these parts may have a multiplicity greater than one, so that a system component can hold any number of session instances of different state machine types. A component structure generated by our transformation algorithm is illustrated in Fig. 4.10 with two taxis and three operators. The taxis have only

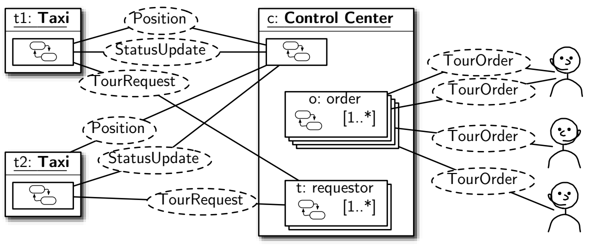


Fig. 4.10: Component structure and their internal session state machines

their default classifier state machines, while the control center component needs additional session state machines, as we will explain in Sect. 4.3.

A system component keeps track of its state machine instances in a data structure for reflection. Each state machine instance has an ID, so that each of them may be addressed within the component by its part name and ID. State machines may access data of other state machines within the same system component. This is used when behavior in one state machine depends on variables in another ongoing collaboration that is executed by another state machine.

4.4.2 Mapping of Single-Session Collaborations

In [KH07] we described an algorithm that transforms activities into executable state machines. One activity partition is translated into at least one state machine. The algorithm scales well since only one partition needs to be considered at a time, not the entire activity. The core idea of this transformation is to map a flow crossing partition borders to a signal transmission between two state machines. Token movements within one partition are translated into state machine transitions. A token starts hereby always at the reception of a signal (where a flow enters a partition) or at a timer node, so that the resulting transitions are triggered by signal receptions or timeouts. A token flow continues traversing the activity graph until the next stable marking is reached, either in form of a join node that cannot yet fire, a waiting decision, a timer node or by leaving the current partition. This stable marking is encoded as control state of the state machine. In this way, the algorithm constructs the entire state machine by a state space exploration of the activity partition corresponding to the state machine.

These basic transformation rules enable a direct mapping of activity flows to state machine transitions as explained and verified in [KH07]. Moreover, several single-session collaborations composed within the same partition may be integrated within the same state machine by combining their state spaces. Therefore, when we synthesize the component for a taxi, both the behavior for the status update and the tour request collaborations may be implemented by the default state machine, as shown in Fig. 4.10.

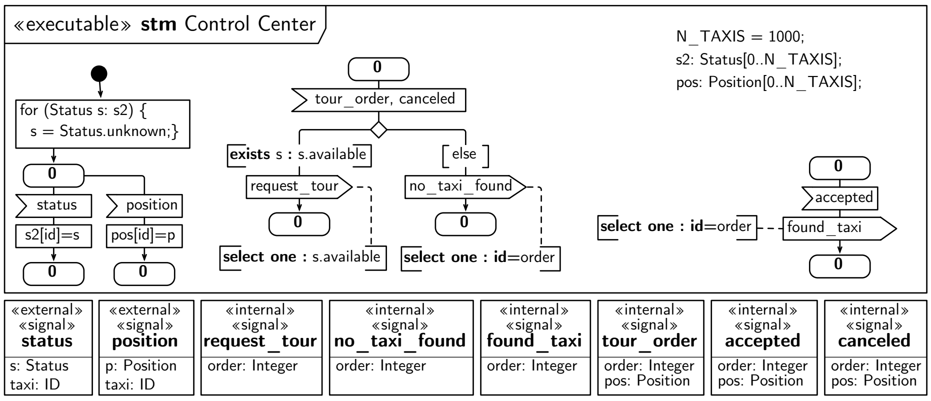


Fig. 4.11: The classifier behavior state machine for the control center

4.4.3 Mapping of Stateless Multi-Session Collaborations

When we analyze the collaboration for the status updates, we find that taxis can send updates at any time, and that the central control has to be prepared at any time to receive them. The behavior on the side of the central control (partition *observer* in Fig. 4.5) is *stateless*, i.e., an update does not cause a change of behavior, but only modifies data. Our algorithm detects this by looking for partitions to be executed by the central control that do not contain any activity nodes that imply waiting (joins, timers or waiting nodes). The algorithm transforms status updates into one state machine transition that has identical source and target control states. This means for the central control that it does not have to distinguish separate control states for each taxi. Instead, the logic to handle status updates of all taxis may be integrated into one single state machine. The same holds for the behavior of the position collaboration, so that both the status update and the position collaborations may be synthesized into the default classifier behavior of the control center. Fig. 4.11 depicts the classifier state machine of the control center. The just mentioned behavior for status and position updates are carried out by the two transitions on the left side which are triggered by the external signals *status* and *position* arriving from taxis. The data about position and status has to be stored for each taxi individually, which is done via the the arrays *s2* and *pos* with the taxi IDs as keys.

4.4.4 Mapping of Stateful Multi-Session Collaborations

For stateful behavior towards multiple partners, the state must be kept for each individual session. There are two principal solutions. One solution is to integrate several sessions into one state machine and to distinguish the conversational states by data structures. This, however, leads to state machines with many decisions. The other solution is to use a dedicated state machine instance for each session, such that the state of each session is represented by an individual control state. If state machines are edited manually (for example in TIMe [BGH^{+}97]), the second solution is preferred, as the state of conversation towards communication partners can be expressed explicitly by the control state of the state machine, which makes them easier to understand [BH93]. This may be of minor significance in an approach generating state machines automatically, but it is nonetheless beneficial if results of the transformation shall be read by humans or be validated with existing techniques [Flo03]. We therefore decided to use one state machine for each session. The fact that this solution may lead to many state machine instances is not problematic, as even large numbers of state machines may be implemented efficiently within the same native operating system process by means of a scheduler (see, e.g., [KHB06, BH93]). A context switch between such state machines just requires to retrieve the current state from a data structure. In a solution integrating all sessions into one state machine, a similar operation would be needed, as we also have to retrieve data belonging to the current state of conversation with a communication partner.

4.4.5 Mapping and Implementation of **select**

The instances of stateful multi-session collaborations are represented locally by session state machines, as we discussed above. Directing control flow to a single or a set of collaboration instances means therefore to transfer control flow to the individual session state machines within a component. This is done by notifying the corresponding state machines via internal signals. In order to reduce the possible interleaving of internal and external signals, we apply the design rule given in [BH93] recommending that internal signals are assigned a higher priority than signals coming from other components. In general, this leads to components that complete internal jobs before accepting external input. In our case, it solves the problem as any **select** signal sent to a session state machine will be handled before an external signal can change its state.

Which state machine(s) should receive the signal(s) is determined by the selection statements from the activities. The transformation therefore copies each selection statement from the edge of the activity and attaches it to the corresponding send signal action. The UML signal is created from the flow. It includes parameters for the data contained in the activity token it represents. The session selection at point ❶ in Fig. 4.7 is, for example, done by the send signal action *request_tour* in the center of Fig. 4.11, with the attached selection statement to determine the receiver address.

It is the task of the code generator to create Java statements from the selection expression that compute the actual addresses of the targeted state machine instances. As discussed above, **select** uses a set of positive and negative filters, with an additional flag indicating whether only one matching state machine instance should be returned or all of them. The generated Java method simply sends the set of state machine IDs through all of the filters specified in **select** by using the Java code already expressed in the activity models. The standard filters **self** and **active** are added accordingly. If a collaboration is started (such as at point ❶) the code for the selection includes mechanisms to create new state machine sessions or retrieves instances from a pool, which is not further discussed here.

4.4.6 Mapping and Implementation of **exists**

In contrast to the select statement, **exists** does not cause a handover of control flow. It is used to get information about properties of the state machines of the system component. As such, it is used in guards of decisions. Decision nodes in activities are mapped directly to choice pseudostates in state machines that have an outgoing transition for each edge leaving the decision node (see [KH07]). The model transformation simply has to copy the exist guards of the activity edges to the corresponding UML state transitions. The implementation of **exists** for execution in Java is similar to that of **select**, with the difference that a boolean value is returned if one session ID passed all filters.

4.5 Concluding Remarks

Much research effort has been spent on the problem of deriving component behaviors from service specifications [vBG86, YEFvBH03]. In many approaches, the service behavior is specified in terms of sequence diagrams or similar notations, which are translated into component behaviors defined as state machines (see [LDD06] for a survey). It is also possible to derive message sequence scenarios from higher-level specifications in the form of activity diagrams or Use Case Maps [AHHC03], and then derive component behaviors in a second step. A direct derivation from Use Case Maps was demonstrated in [Cas05]. In this paper, however, we consider the direct and fully automated derivation of component behavior from the specification of collaboration behavior expressed as activities. While we presented the transformation from single-session collaborations to state machines in [KH07], we have extended the notation of activities and our transformation algorithm to handle also collaborations executed in several sessions at the same time, as presented in this paper. The advantage of our notation with **select** and **exists** is that they can express the relations between sessions explicitly on an abstract level and are still straight-forward to map to state machines that can be implemented by our code generators [Kra03]. The transformation algorithm is implemented as an Eclipse plug-in and works directly on the UML 2.0 repository of the Eclipse UML2 project.

We consider the specification of services in a collaboration-oriented way as a major step towards a highly automatic model-based software design approach. As depicted in Fig. 4.1, we hide the *inter-component* communication in the collaborations and activities while the *intra-component* communication is carried out by linking activities with each other in partitions of surrounding activities. This makes it possible to express sub-services in separation, which facilitates the general understanding of their behavior. Moreover, each collaboration models a clear, separate task such that interaction-related problems like mixed initiatives can be detected and solved more easily since only the problem-relevant behavior is specified. The composition of collaborations profits from the input and output nodes of activities which form the behavioral interfaces of the collaboration roles. Different collaborations can be suitably composed by connecting their nodes using arbitrary activity graphs.

Another advantage of collaboration-oriented specifications is the higher potential for reuse. Usually, the sub-services modeled by collaborations can be used in very different applications (such as for example the distributed status update expressed by the collaboration *Status Update*). These sub-services can be modeled once by a collaborations which can be stored in a library. Whenever such a sub-service is needed, its activity is simply taken from the library, instantiated and integrated into an enclosing collaboration. In our example, *Status Update*, *Button*, *Alarm* and *Position* are good candidates for reuse.

An ongoing research activity is the development of suitable tools for editing, refining, analyzing, proving and animating collaboration-based models. This will be performed within the research and development project ISIS (Infrastructure of Integrated Services) funded by the Research Council of Norway. The concept of our approach will be proven by means of real-life services from the home automation domain. We consider collaboration-oriented service engineering as a very promising alternative to traditional component-centered design and understand the extensions for modeling and transforming sessions, presented in this paper, as an important enabler. ■

Bibliography

[AHHC03] Daniel Amyot, Xiangyang He, Yong He, and Dae Yong Cho. Generating Scenarios from Use Case Map Specifications. *qsic*, 00:108, 2003.

[BF04] Rolv Bræk and Jacqueline Floch. ICT Convergence: Modeling Issues. In Daniel Amyot and Alan W. Williams, editors, *SAM'04 - Fourth SDL and MSC Workshop*, volume 3319 of *Lecture Notes in Computer Science*, pages 237–256. Springer, 2004.

[BGH$^+$97] Rolv Bræk, Joe Gorman, Øystein Haugen, Geir Melby, Birger Møller-Pedersen, and Richard Sanders. Quality by Construction Exemplified by TIMe — The Integrated Methodology. *Telektronikk*, 95(1):73–82, 1997.

[BH93] Rolv Bræk and Øystein Haugen. *Engineering Real Time Systems: An Object-Oriented Methodology Using SDL.* The BCS Practitioner Series. Prentice Hall, 1993.

[Cas05] Humberto N. Castejón. Synthesizing State-machine Behaviour from UML Collaborations and Use Case Maps. In Andreas Prinz, Rick Reed, and Jeanne Reed, editors, *12th International SDL Forum, Grimstad, Norway*, volume 3530 of *Lecture Notes in Computer Science.* Springer, 2005.

[CB06a] Humberto N. Castejón and Rolv Bræk. A Collaboration-based Approach to Service Specification and Detection of Implied Scenarios. In *SCESM '06: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pages 37–43, New York, NY, USA, 2006. ACM Press.

[CB06b] Humberto N. Castejón and Rolv Bræk. Formalizing Collaboration Goal Sequences for Service Choreography. In Elie Najm and Jean-François Pradat-Peyre, editors, *26th IFIP WG 6.1 Intl. Conf. on Formal Methods for Networked and Distributed Systems (FORTE'06)*, volume 4229 of *Lecture Notes in Computer Science*. Springer, September 2006.

[FK01] Kathi Fisler and Shriram Krishnamurthi. Modular Verification of Collaboration-Based Software Designs. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 152–163, New York, NY, USA, 2001. ACM Press.

[Flo03] Jacqueline Floch. *Towards Plug-and-Play Services: Design and Validation using Roles.* PhD thesis, Norwegian University of Science and Technology, 2003.

[HK00] Peter Herrmann and Heiko Krumm. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337, 2000.

[HK07] Peter Herrmann and Frank Alexander Kraemer. Design of Trusted Systems with Reusable Collaboration Models. In Sandro Etalle and Stephen Marsh, editors, *Trust Management*, volume 238, pages 317–332. IFIP International Federation for Information Processing, Springer, 2007.

[ITU02] ITU-T. *Recommendation Z.100: Specification and Description Language (SDL)*, August 2002.

[KH06] Frank Alexander Kraemer and Peter Herrmann. Service Specification by Composition of Collaborations — An Example. In *Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology)*, pages 129–133. IEEE Computer Society, 2006. 2nd International Workshop on Service Composition (Sercomp), Hong Kong.

[KH07] Frank Alexander Kraemer and Peter Herrmann. Transforming Collaborative Service Specifications into Efficiently Executable

State Machines. In Karsten Ehring and Holger Giese, editors, *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, volume 7 of *Electronic Communications of the EASST*. EASST, 2007.

[KHB06] Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk. Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. In R. Meersmann and Z. Tari, editors, *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), 2006, Montpellier, France*, volume 4276 of *Lecture Notes in Computer Science*, pages 1613–1632. Springer–Verlag Heidelberg, 2006.

[KM03] Ingolf Krüger and Reena Mathew. Component Synthesis from Service Specifications. In *Scenarios: Models, Transformations and Tools*, volume 3466 of *Lecture Notes in Computer Science*, pages 255–277. Springer, 2003.

[Kra03] Frank Alexander Kraemer. Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart, 2003.

[Kra07] Frank Alexander Kraemer. Building Blocks, Patterns and Design Rules for Collaborations and Activities. Avantel Technical Report 2/2007 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway, March 2007.

[Kra08] Frank Alexander Kraemer. UML Profile and Semantics for Service Specifications. Avantel Technical Report 1/2007 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway, June 2008.

[LDD06] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A Comparative Survey of Scenario-Based to State-Based Model Synthesis Approaches. In *SCESM '06: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pages 5–12, New York, NY, USA, 2006. ACM Press.

[Obj07] Object Management Group. Unified Modeling Language: Superstructure, version 2.1.1, February 2007. formal/2007-02-03.

[RAB$^+$92] Trygve Reenskaug, Egil P. Andersen, Arne Jorgen Berre, Anne Hurlen, Anton Landmark, Odd Arild Lehne, Else Nordhagen, Eirik Ness-Ulseth, Gro Oftedal, Anne Lise Skaar, and Pål Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *Journal of Object-oriented Programming*, 5(6):27–41, October 1992.

[RGG01] Frank Rößler, Birgit Geppert, and Reinhard Gotzhein. Collaboration-Based Design of SDL Systems. In *Proceedings of* *the 10th International SDL Forum Copenhagen on Meeting UML*, pages 72–89. Springer-Verlag, 2001.

[Ris01] Linda Rising, editor. *Design Patterns in Communications Software.* Cambridge University Press, New York, NY, USA, 2001.

[RWL95] Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working with Objects, The OOram Software Engineering Method.* Prentice Hall, 1995.

[SBvBA05] Richard Sanders, Rolv Bræk, Gregor von Bochmann, and Daniel Amyot. Service Discovery and Component Reuse with Semantic Interfaces. In *Proceedings of the 12th SDL Forum*, 2005.

[vBG86] Gregor von Bochmann and Reinhard Gotzhein. Deriving Protocol Specifications from Service Specifications. In *SIGCOMM '86: Proceedings of the ACM SIGCOMM conference on Communications architectures & protocols*, pages 148–156, New York, NY, USA, 1986. ACM Press.

[YEFvBH03] Hirozumi Yamaguchi, Khaled El-Fakih, Gregor von Bochmann, and Teruo Higashino. Protocol Synthesis and Re-Synthesis with Optimal Allocation of Resources based on Extended Petri Nets. *Distrib. Comput.*, 16(1):21–35, 2003.

Comment

Tour Request in Fig. 4.6 uses activity final nodes to terminate the collaboration. We changed this kind of termination with the profile in App. A, so that only terminating parameter nodes are needed. Activity final nodes are then only used on the system level.

PAPER

FIVE

DESIGN OF TRUSTED SYSTEMS WITH REUSABLE COLLABORATION MODELS

Peter Herrmann and Frank Alexander Kraemer.

Presented at the Joint iTrust and PST Conferences on Privacy, Trust Management and Security (IFIPTM), July 30 - August 2, 2007 Moncton, Canada.

Published in *IFIP International Federation for Information Processing*, Volume 238, p. 317-332, Springer, 2007.

The original publication is available at www.springerlink.com via `http://dx.doi.org/10.1007/978-0-387-73655-6_21`.

Design of Trusted Systems with Reusable Collaboration Models

Peter Herrmann and Frank Alexander Kraemer

Abstract. We describe the application of our collaboration-oriented software engineering approach to the design of trust-aware systems. In this model-based technique, a specification does not describe a physical system component but the collaboration between various components which achieve system functions by cooperation. A system model is composed from these collaboration specifications. By a set of transformations, executable code can be automatically generated. As a modeling language, we use UML 2.0 collaborations and activities, for which we defined a semantics based on temporal logic. Thus, formal refinement and property proofs can be provided by applying model checkers as well. We consider our approach to be well-suited for the development of trust-based systems since the trust relations between different parties can be nicely modeled by the collaborations. This ability facilitates also a tight cooperation between trust management and software engineering experts which are both needed to create scalable trust-aware applications. The engineering approach is introduced by means of an electronic auction system executing different policies which are guided by the mutual trust of its principals. While the approach can be used for various trust models, we apply Jøsang's Subjective Logic in the example.

5.1 Introduction

Since the turn of the millenium, the management of trust has gained more and more momentum. While this field is inherently multi-disciplinary and researchers from psychology, sociology, philosophy, law and economics work on trust issues for many years, computer science seems to be the driving force behind the current advances. An important reason for that is the maturing of the internet-based consumer commerce [Che99]. The acceptance of e-commerce services depends directly on the trust the different parties involved in it can build up in each other. In the internet, however, commerce partners are often unknown, live in another country with a different legal system, and are selected on an ad hoc basis guided by the best offer. Therefore, traditional trust building mechanisms like personal experience, recommendations by friends, or the general reputation "in town" cannot be used in the same way as in traditional commerce. The trust management community started to overcome this deficiency by developing trust models consisting of both representations for trust in computers and related mechanisms specifying the building of trust. Some of these models describe trust in a more general way from either a mathematical-philosophical perspective (e.g., [Jøs01, JF01]) or from a sociological-cognitive view (e.g., [FC01, Mez04]). Other approaches are devoted to realize trust building mechanisms which take the practical limits of computer systems and networks into account [BFL96, GS02, ARH00, AD01, AM04].

The invention of computer-readable trust mechanisms facilitates the design of applications incorporating trust. Most approaches enhance or replace traditional security mechanisms at points where they are not suitable for modern ad hoc-networks. In particular, a number of solutions were developed for access control of both peer-to-peer networks [XL02, KSGM03, Ing05] and business processes for web services [BS02, YWS03, KM04] while other tools approach authorization [LWBW06], authentication and identity management [PM06] as well as privacy [Pea05]. A second field of application design is devoted to federate systems combined of separate partners and, in particular, to determine the kind of mutual protection of the partners. Here, a wide field starting at security-protecting routing algorithms [JO06] via the formation of virtual organizations [KHKR06] to the trust-based protection of component-structured software [Her03, LTM06] and the protection of collaborations of pervasive devices [QHC06] is covered. It does not require prophetic skills to expect that there will be a lot more trust-encompassing systems to come in various application domains.

As the design of trust-based systems can be quite complex, it has to incorporate typical software engineering techniques. The application of these techniques is usually so difficult that experienced software engineers are required. Thus, to develop a trust-aware system, we need experts both for the trust management and for software engineering who have to cooperate very closely since the trust management functions of a system are tightly interwoven with the rest of the system logic. Ideally, the trust management developer should be able to integrate trust models into a system design process without necessarily understanding the full application logic, while the software designer should be capable to make the general software engineering decisions without comprehending the complete functionality of the underlying trust management model.

We consider our software engineering approach based on collaboration-oriented formal system models [KH06] as a solution to this problem. Most modeling techniques combine system specifications from models specifying a separate physical software component each. In contrast, in our technique a specification building block describes a partial system functionality which is provided by the joint effort of several components cooperating with each other. Every component taking part in a collaboration is represented in the form of a so-called collaboration role. The behavior models of collaborations specify both the interactions between the collaboration roles as well as local behavior of collaboration roles needed to provide the modeled functionality. Collaborations may be composed with each other to more comprehensive collaborations by means of collaboration uses. Thus, hierarchical system models are possible.

As an example, we depict in Fig. 5.1 the collaboration uses of the highest hierarchical level to model a trusted electronic auction system which will be introduced in detail in sections 5.3 and 5.4. The system specifies an automatic internet-based auction system which could, for instance, be built upon the web services offered by eBay. From a trust management perspective, the major problem of such a system is the sale between the winning buyer and the seller

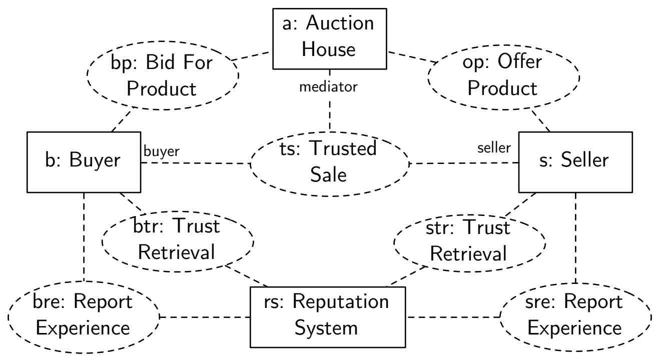


Fig. 5.1: Collaboration of the *Trusted Auction System*

after the auction since the reluctance of one party to pay resp. to deliver the product may cause damage to the other side. As a solution, we provide a trust-encompassing application based on a reputation system (e.g., the eBay feedback forum). According to their mutual trust, both parties can decide how to carry out the sale. As a consequence, the example system incorporates four major components: the winning buyer, the seller, the reputation system and the auction house. Its functionality is expressed by means of seven collaboration uses depicted in Fig. 5.1. The collaboration use *btr* models the access to the reputation system by the buyer in order to retrieve the current trust of the community in the seller. We will see in Sect. 5.4 that this retrieval is done before bidding for the product. Likewise, the collaboration use *str* describes the retrieval of the buyer's trust value by the seller which takes place after the auction. According to the mutual trust, the buyer and seller perform the sale which is modeled by *ts*. Indeed, this collaboration is a composition from more basic collaborations specifying four different modes which depend on the trust of the participants in each other. After finishing the sale, both parties report their mutual experiences to the reputation system which is expressed by the collaboration uses *bre* and *sre*. The remaining collaboration uses *op* and *bp* describe the offering of goods by the seller and the bidding of the buyer. As these collaboration uses are not relevant from a trust management perspective, they are not discussed further.

Fig. 5.1 is a collaboration in the popular graphical modeling language UML 2.0 (Unified Modeling Language [BRJ99, Obj06]). These diagrams are used to describe the basic structure of a collaboration (i.e., the collaboration uses forming it and the relation between the roles of the collaboration uses and those of the comprehensive collaboration). To specify the behavior of the collaborations and the logic combining collaboration uses is described by UML activities which are introduced in Sect. 5.3.

As trust relations are inherently collaborative and always comprise at least a trustor and a trustee, we consider the collaboration-oriented specification style very helpful to develop trust-based systems. The reduction of systems to sub-functionalities supports their understanding to a high degree (cf. [KH06, SCKB05, RB06, CB06]). As discussed in Sect. 5.2, we consider this property useful to provide trust management experts and software developers with a fundament for tightly interwoven cooperation. In addition, the model structure enables a higher reuse of collaborations. In many distributed application domains, the system components cooperate with each other by means of a relatively small number of recurrent sub-functionalities which can be specified once and thereafter stored in a library. System developers can create their specifications in a relatively simple way by selecting collaborations from the library, instantiating them, and composing them to a system description. In our example, *btr*, *str*, *bre*, and *sre* are instantiations of the collaborations *Trust Retrieval* resp. *Report Experience* which are suitable building blocks to create applications using reputation systems.

By means of an algorithm [KH07], we can automatically transform the collaboration-oriented models into executable state machines from which in a second step executable code can be generated [KHB06]. Moreover, we currently develop a transformation to TLA$^+$ [Lam02], the input syntax of the model checker TLC [YML99] which facilitates formal proofs of system properties. This will be further discussed in Sect. 5.5. Before that, we discuss in Sect. 5.2 the benefit of our approach for the generation of trust management-based systems. Thereafter, the specification of collaborations by UML collaboration diagrams and activities is introduced by means of the trusted auction example in Sect. 5.3. The coupling of collaboration uses to more comprehensive collaborations is outlined in Sect. 5.4.

5.2 Trust Management Aspects

In recent years, numerous definitions for trust have been published. A significant one was introduced by Jøsang [Jøs96] who distinguishes between trust in humans and trust in computers. He calls humans as well as organizations formed by humans with a free will *passionate entities*. In contrast, computers and other entities without a free will are named *rational entities*. Trust in a passionate entity is defined as *"the belief that it will behave without malicious intent"* while trust in a rational entity is *"the belief that it will resist attacks from malicious agents"* [Jøs96]. Both definitions have in common that a trustor can only be a passionate entity since trust needs a free will. Nevertheless, in specific application domains both the building of trust and its deployment selecting different policies to deal with the trustee is so rational that it can be handed over to a computer. A good example is the decision making process of banks whether to provide loans or not. A bank's behavior is basically guided by its trust in a debtor that he will be able to pay back a loan. To build this trust, typical mechanisms as the debtor's behavior in previous cases (i.e., the debtor's reputation) are taken into account and the decision is made according to fixed policies. These policies can be implemented on a computer as already applied in some banks.

For the representation of trust one can apply trust values. For instance, Jøsang introduces so-called opinion triangles [Jøs01, Jøs99]. These are effectively triples of probability values, the sum of which is always 1. Two of these values describe the belief resp. disbelief in the trustee while the third one states the uncertainty based on missing knowledge on the trustee. The building of trust is, in consequence, described by traces of changing trust values. In between, a lot of trust models were developed which are suited for computers (cf. [Jøs01, Mez04, BFL96, GS02, ARH00, AD01, AM04]). The utilization of trust in dealing with a trustee can also be realized on a computer by defining trust-related policies. The actual policy can then be selected based on the current trust value.

Our collaboration-oriented software development approach is well-suited to model the mechanisms used to describe the building of trust. A collaboration is appropriate to describe the various functions of a trust model since every function affects more than one partner. Moreover, the collaborations can be used as building blocks for trust-encompassing applications. For instance, the collaborations *Trust Retrieval* and *Report Experience* used in the trusted auction model (see Fig. 5.1) describe the two aspects typically used in dealing with a reputation system, i.e., the decision about how to deal with the trustee depending on its current trust value as well as improving the trustee's assessment by sending the reputation system a positive or negative experience report. Similar collaborations can be defined to model other trust gaining mechanisms such as considering one's own experience or the recommendation by third parties. In addition, to support the design of more complex trust building mechanisms, one can add building blocks enabling the combination of different trust values.

The method is also useful to simplify the cooperation between the trust management experts and the software engineers. A trust expert can specify the trust building functions of the system on its own by utilizing collaborations from a library. The outcome will be a set of collaboration uses that the software engineers can integrate into the overall system model without fully understanding their internal behavior. The engineers only need to recognize that different trust-based policies are possible but not the steps to decide which actual policy should be used.

Somehow more difficult is the support of the cooperation between the two expert groups in modeling the enforcement of the different trust policies. Here, aspects of the general application functionality and special trust-related properties have to be combined. This can be achieved by a twofold proceeding. First, characteristic trust-based functions may be used to enforce policies. These functions can also be modeled by collaborations and used in several system models. For instance, a sale between two parties with a low degree of trust in each other can be performed by including a trusted third party which mediates the sale by guaranteeing that a buyer cannot receive the product before sending the money, while the seller must send the product before receiving the payment. It is easy to model this as a collaboration which can be used by the software engineer without understanding the exact functionality (see also Sect. 5.4).

Second, the trust expert can inform the software engineer about trust-related functionalities the application has to follow. For instance, a requirement of the trusted sale should be that the buyer only issues the money transfer to the seller without having evidence of receiving the product in time if her trust in the seller is high. The software engineer considers these properties in the system development. Afterwards, the trust expert can check that the system complies with the properties by, for instance, proving them with the model checker TLC [YML99]. In the following, we will clarify how trust-based systems like the trusted auction example can be developed using the collaboration-oriented specification style.

5.3 Activity-Based Collaboration Models

As depicted in Fig. 5.1, we use UML collaborations to specify the overall structure of system models composed from collaboration uses. In particular, a collaboration describes the different components forming a system and the assignment of the roles of the collaboration uses to the components. To model the behavior of a collaboration, UML offers various diagram types like state machines, sequence diagrams, and activities [Obj06]. We decided to use activities mainly for two reasons: First, activities are based on Petri Nets and specify behavior as flows of tokens passing nodes and edges of a graph. This proved to represent flows of behavior quite naturally and is therefore easy to understand (cf. [KH06]). Second, activities are self-contained. Sequence diagrams, for instance, typically describe in one diagram only a set of system scenarios rather than the complete behavior. In contrast, activities facilitate the specification of the full behavior of a collaboration within one diagram.

A typical example for an activity is *Trust Retrieval* which models the behavior of the collaborations *btr* and *str* in the trusted auction example[1] (see Fig. 5.1). It is listed in Fig. 5.2 and describes the access of a caller to a reputation system in order to retrieve a trustee's reputation. Moreover, it models the decision about a certain level of trust which may lead to different trust policies. Since the collaboration comprises two different roles, the client of the reputation system and the reputation system itself, we use two activity partitions in the diagram which are named by the role identifiers. The interface of the collaboration to its environment is located at the activity partition of the client and consists of three output pins each describing a certain level of trust.[2]

The behavior of the activity is described by a token flow which is started at the input node in the partition of the client. It passes a token from the client via the partition border to the reputation system. The token contains an identifier of the trustee which is computed in the call operation action *retrieve trust value*. This call operation action contains the logic to access the number of good and bad experiences with the trustee and to generate the current trust value. The trust value is thereafter forwarded back to the caller and evaluated in the call operation action *evaluate trust value* (i.e., the trust value is copied to the auxiliary collaboration variable *tv*). Thereafter, the token proceeds to a

[1]We use Jøsang's approach [Jøs01, JK98] to specify trust and trust building in the example but could adopt the specifications easily to other trust models.

[2]As these output pins are mutual exclusive, they belong to different parameter sets shown by the additional box around them.

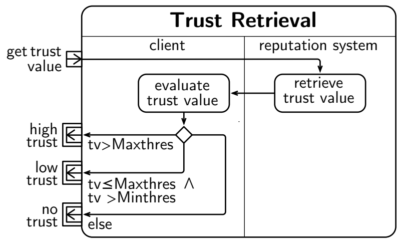


Fig. 5.2: Activity *Trust Retrieval*

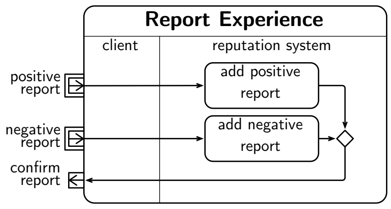


Fig. 5.3: Activity *Report Experience*

decision node (◇) from which it branches to one of three edges. The branching is guided by the conditions of the decision node, which depend on two thresholds. Finally, the token is forwarded to the activity environment via one of the output pins *high trust*, *low trust*, or *no trust*. By passing one of the output pins, the overall activity is terminated. A trust management expert can instantiate *Trust Retrieval* simply by defining suitable thresholds.

Activity *Report Experience* (Fig. 5.3) models the report of positive or negative experiences with a trustee to the reputation system adjusting the trustee's reputation. It is started with a token passing one of the input pins *positive report* or *negative report*. The tokens are forwarded to the reputation system which adapts the trustee's data base entry in the call operation actions. The edges leaving the two call operation actions lead to a merge node (◇) that merges its incoming flows by forwarding all incoming tokens to the only outgoing edge. In this way, after registering either a positive or negative report, the token is passed back to the client's output pin *confirm report* describing the confirmation of the experience report.

The activity *Mediated Sale* introduced in Fig. 5.4 expresses a functionality with several parallel flows. As discussed before, a mediator acts here as a trusted third party which assures a fair sale by collecting the payment and the product which are delivered to their recipients not before both are received by the mediator. The activity consists of three partitions for the buyer, the seller and the mediator. It is started by two separate tokens arriving from the buyer through the input pin *send payment* and from the seller via *send product*. The token from the buyer heads to the fork node f_1. In a fork node every incoming token is reproduced and one copy is sent via every outgoing edge. One of the tokens leaving f_1 reaches the send action *ReqPayM*. We use send actions to model the

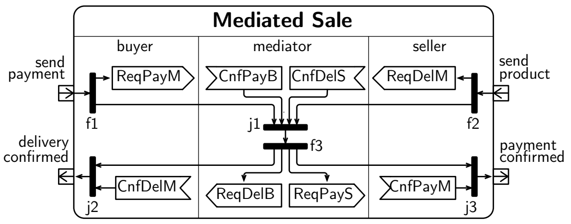


Fig. 5.4: Activity *Mediated Sale*

transfer of signals to external applications which are not an inherent part of the modeled application. For instance, the accounting unit of the buyer is an example of an external system which is notified by *ReqPayM* to issue the payment to the mediator. The other token leaving f_1 is forwarded to the mediator which is notified thereby about the start of the payment. Likewise, the seller calls its delivery unit to send the product to the mediator which is expressed by the send action *RegDelM* and notifies the mediator as well. When the payment arrives at the mediator, it is notified by its accounting unit using the receive action *CnfPayM* while *CnfDelS* reports the reception of the product. Similar to send actions, we use receive actions to model incoming signals from the environment. All tokens coming from the two receive actions and from the buyer resp. seller lead to the join node[3] j_1. A flow may only leave a join if tokens have arrived on all of its incoming edges. During the execution of the join, all but one token are removed and the remaining token leaves it via its outgoing edge. The token leaving j_1 continues to the fork f_3 from which both deliveries to the final recipients and the notifications are issued. Thus, by the combination of j_1 and f_3 we guarantee that deliveries are only carried out if both the payment and the product have arrived at the mediator.

The notification for the buyer heads to the join node j_2 and can only be forwarded if the buyer's delivery unit reports the product's reception which is specified by the receive action *CnfDelM.* The token passing j_2 leaves the activity via the output pin *delivery confirmed.* Likewise, the seller sends a confirmation of the payment via *payment confirmed* after receiving the money. As the two activities introduced above, *Mediated Sale* can be provided by the trust management expert. The only necessary cooperation with the software engineer is to agree about the formats of the transmissions with the various accounting and delivery units.

5.4 Coupling Activities

Activities are especially powerful for the composition of behaviors from existing ones. This is done by means of call behavior actions that refer to other

[3]UML uses identical symbols for join and fork nodes. They can be distinguished by the number of incoming and outgoing edges. Fork nodes have exactly one incoming edge while join nodes have exactly one outgoing edge.

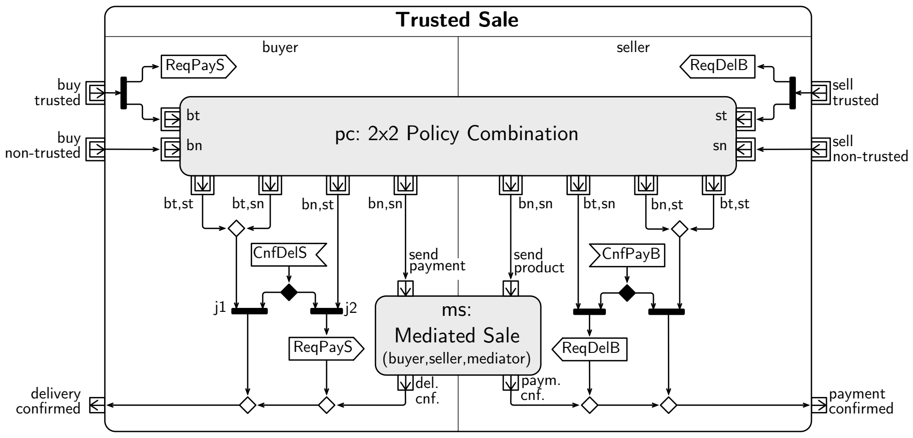


Fig. 5.5: Activity *Trusted Sale*

activities. The events of the activities may be coupled using all kinds of control nodes and edges, so that arbitrary dependencies between the sub-activities may be described. As activities are used in our approach to describe the behavior of collaborations, this technique is applied to compose the collaborations behaviorally (while the UML collaboration in Fig. 5.1 shows the structural aspect of this composition.) An example of a composed activity is *Trusted Sale* in Fig. 5.5 which is composed from the call behavior actions *ms* and *pc* referring to the behavior of subordinate activities (resp. collaborations).

Trusted Sale describes the functionality of selling a product between a buyer and a seller after finishing an auction. The two parties in the sale may either have a high or a low degree of trust in the other one, which is modeled by the two input pins in both the buyer and the seller partition. If the buyer has a high degree of trust in the seller, she is willing to send the payment immediately without waiting for the partner. That is described by the send action *ReqPayS* to which a token is forwarded directly after entering the activity via *buy trusted.* By this send action, the accounting unit of the buyer is notified to start the payment to the seller. Likewise, the seller is ready to send the product to the buyer immediately if he has a high level of trust which is expressed by the flow to the send action *ReqDelB.*

Since both parties may either have high or low trust in each other, four different trust relations between the two parties are possible and for each one a separate sale policy is defined. Nevertheless, to decide about a sale policy, both parties have to know the partner's trust in themselves. As a mutual distributed combination of policies is a quite common function in many networked systems, we have a collaboration and a corresponding activity *2×2 Policy Combination* available from our general pattern library which can be applied here in the form of the call behavior action *pc.* This activity has two input pins and four output pins on each side. The two parties define the selected input policy by transferring a token via the corresponding input pin which causes the delivery of tokens through those output pins describing the combination of the two policies (e.g., if the buyer sends a token via input pin *bt* (for *buy trusted*) and the seller via *sn* (for *sell non-trusted*), the tokens will eventually arrive at the output pins *bt,sn*). The input nodes of *Trusted Sale* are connected with the corresponding ones of *pc* and its output pins can be used as the starting points to model the four sale policies (*bt,st; bt,sn; bn,st; bn,sn*):

- If both partners have a high degree of mutual trust (*bt,st*), they simply send the payment resp. the product without waiting for the other. Each partner completes the sale after the delivery has arrived. As the payment has already been started, the buyer has to wait for a token arriving via output pin *bt,st* in join j_1 for the delivery of the product. The reception of the product is described by the accept signal action *ConfDelS* forwarding a token to j_1 as well.[4] Thus, j_1 can be triggered and a token leaves the activity *Trusted Sale* via the output pin *delivery confirmed* which specifies the completion of the sale on the buyer's side. The behavior in the partition of the seller is similar.

- If the buyer has only a low trust in the seller but the seller a high one in the buyer (*bn,st*), we use a policy in which the seller transfers the product first and the buyer initiates the payment not before receiving the product. Thus, the buyer does not send the payment initially, but waits for the delivery of the product which is expressed by the token in join j_2. After the delivery is notified as modeled by a token heading from *ConfDelS* to j_2, the buyer initiates the payment, which is described by the send action *ReqPayS*, and finishes the sale. The handling of this policy on the seller's side is identical to the first one since it behaves similarly in both policies.

- If the buyer has a high degree of trust in the seller which, however, trusts the buyer only lowly (*bt,sn*), we use the reciprocal policy to that listed above. Here, the seller does not send the product before receiving the payment. As the effective behavior for the buyer is the same as for the policy (*bt,st*), the flow from *bt,sn* is simply merged into the behavior for *bt,st*.

- If both partners have a low degree of trust in each other (*bn,sn*), they decide to rely on a mediator. This can be modeled by applying the activity *Mediated Sale* introduced in Sect. 5.3. The pins *bn,sn* are simply connected with the input pins of *Mediated Sale* and its output pins with the output pins of *Trusted Sale*.

When one of the partners cheats by not sending anything, the activity is not finished correctly but stops somewhere. We will see below that this case leads to a negative rating of the partner.

[4]The token leaving *ConfDelS* is stored in a so-called waiting node (◆, cf. [KH07]) which forwards it to join j_1 or j_2 depending on which join can be executed first.

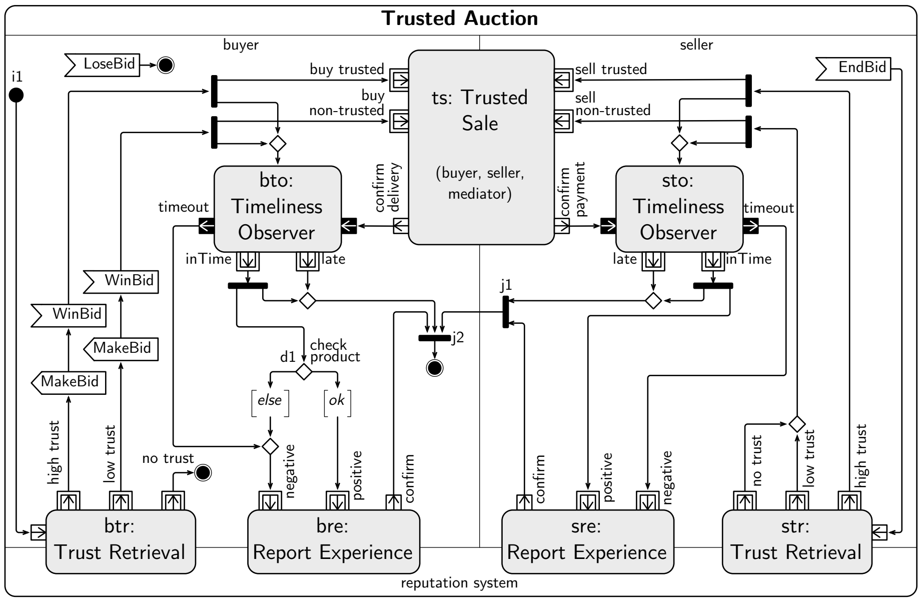


Fig. 5.6: Activity *Trusted Auction*

The activity *Trusted Sale* exemplifies the interplay between both expert groups. The trust management expert provides the software engineer with the activity *Mediated Sale* and describes the four sale policies. Based on this information, the software engineer accomplishes the overall model of the trusted sale which can be added to the library of building blocks for trusted systems facilitating a later usage in other applications.

The last activity introduced here is *Trusted Auction* depicted in Fig. 5.6 which describes the behavior of the overall system. The collaboration uses it is composed of (see Fig. 5.1) are represented by the call behavior actions *btr*, *str*, *bre*, *sre*, and *ts*. While an electronic auction encompasses an arbitrary number of buyers and sellers, we laid out the activity in a way that only the relation between exactly one buyer and one seller is modeled by the activity. In consequence, the whole application is described by multiple instances of *Trusted Auction*. For the sake of brevity, we omitted the part in which the seller registers the product since that is not relevant for trust management. Thus, the activity is started by the buyer, who becomes active if she finds an interesting product. This is expressed by the initial node i_1 from which, at first, the trust level of the seller is retrieved by accessing *btr*. If the reputation of the seller is so bad that there is almost no trust, the buyer decides not to bid and the activity is terminated by a final node (◉). If the buyer trusts the seller to at least some degree, she makes a bid[5] which is modeled by the send action *MakeBid* and waits in the receive node *WinBid* for the end of the bidding. If the bid is not sufficient, a token

[5]For brevity, we assume that a buyer makes only one bid in an auction.

is received via the accept signal action *LoseBid* and the activity is terminated since no further action is necessary. If the bid won, a token leaves *WinBid* and the trusted sale is started by forwarding a token to *ts*. Moreover, the instance *bto* of activity *Timeliness Observer* is started. It specifies a timeout process to detect late deliveries of the product which will be discussed below.

On the seller's side, a flow is started after the auction is finished which is expressed by *EndBid*. Thereafter, the reputation of the buyer is retrieved in *str* and the trusted sale is started as well. Due to the nature of an electronic auction system, the seller has to start the sale process even if he does not trust the buyer at all. Furthermore, *sto* is initiated starting a timer as well. In the case of a timeout, a token leaves the output pin *timeout* immediately, meaning that the payment did not arrive in due time, and via *sre* a negative report on the buyer is sent to the reputation system. The confirmation is forwarded to the join node j_1 used to synchronize the activity termination in the seller partition. If the payment is confirmed, a token proceeds from *ts* to *sto*. If this confirmation arrives at *sto* after the timeout, a token is issued at the output pin *late* which is forwarded to j_1. If the negative report was already confirmed, j_1 can fire which notifies the buyer's side that the seller can accept to terminate the activity. If the payment confirmation arrives in time, a token leaves the output pin *inTime* of *sto*, issuing a positive report about the buyer. In addition, a token is forwarded to j_1 such that the buyer can be notified about the readiness for termination after the experience report was confirmed.

The behavior after finishing the sale on the buyer's side is similar except for the decision d_1. We assume that the delivery unit of the buyer attaches information to the token sent to the activity *Trusted Sale* describing if the quality of the product is sufficient. In that case, a positive report is triggered while a bad condition of the product leads to a negative report. The join j_2 can only be executed if the delivery of the product was confirmed, the report about the seller was attested and the seller reported that it is ready to terminate. The execution of j_2 causes the termination of the activity.

As in the activity *Trusted Sale*, this activity can be developed combining the competence of the two expert groups. The trust management expert delivers the activities describing the access to the reputation system as well as some policies defining, for instance, which reports have to be issued to the reputation system under which circumstances. This provides the software engineer with the sufficient knowledge to develop the behavioral model specified by the activity.

5.5 Implementation and Verification

The fact that activities render a complete system behavior facilitates automatic generation of code from the collaboration-oriented model which is performed in a series of steps: At first, we apply the algorithm introduced in [KH07] which transforms the activities into a set of UML state machines each describing a system component. As we defined both the semantics of the activities and the state machines based on the compositional Temporal Logic of Actions (cTLA) [HK00], the correctness of the transformation could be verified by a cTLA refinement proof sketch (cf. [KH07]). For our example, the algorithm in its current version creates separate state machines modeling the behavior of the buyer, the seller, the reputation system and the auction house acting as mediator. Due to the varying complexity of the four components, the state machines have a quite different size. Since the behavior of the reputation system is stateless, its state machine consists only of one control state and three transitions modeling the retrieval of trust values as well as the addition of positive and negative experience report. In contrast, the state machine of the mediator consists of 15 control states, while that of the buyer models the most complex functionality using 64 control states.

The state machines have a special "executable" form in which, except for the initialization, all transitions are triggered by incoming signals from the environment or from local timers. Since, in addition, the enabling condition of a transition depends only on the control state of the state machine but not on its auxiliary variables, very efficient executable code can be generated. This kind of code generator has been built for nearly 30 years now (see, for instance, [Bræ79, BGH$^+$97]). To implement our example, we used a generator creating Java code which is executed on the middleware platform JavaFrame [HMP00]. During testing the application, we could not detect any significant overhead. The application of the code generators, the related middleware platforms, and a cTLA-based correctness proof are described in [KHB06].

The trust expert can check if the produced collaboration-oriented model fulfills the trust-related properties passed to the software engineer by applying an animation tool. Moreover, due to defining the semantics of the activities by cTLA, formal refinement and invariant proofs are also facilitated. For instance, the property that the buyer may only start a payment to the seller immediately if she has high trust in him can be expressed by an invariant. This excludes a state in which (1) the trust level is low, (2) the payment was already sent to the seller and (3) the product is not yet delivered. By a cTLA proof, one can verify that the cTLA formula specifying the activity *Trusted Sale* always fulfills the invariant. In the context of trusted systems, this kind of proof was introduced in [Her06]. We currently develop a tool transforming activities directly into the input syntax TLA$^+$ [Lam02] of the model checker TLC [YML99] carrying out the proofs automatically. Of course, model checkers are subject to the state space explosion problem. Thus, the number of states to be inspected in a scalable system can be too large to be handled by the checker. cTLA, however, supports a coupling style reflecting the activity combinations in a quite natural way. For each activity, a separate cTLA model is created and, in a proof, only those models realizing the verified property need to be considered. For instance, to prove the invariant listed above, only the states of the cTLA model representing the activity *Trusted Sale* must be checked. This quality of cTLA makes our approach not only well-suited for the design and implementation of realistic trust-based systems but also enables formal property proofs in a relatively user-friendly way.

5.6 Concluding Remarks

In this paper we introduced our collaboration-oriented software development approach which facilitates system modeling by specifying the various cooperations between the system components separately. We consider the approach well-suited for the design of trust-aware systems since trust relations between principals can be directly modeled as collaborations. This property enables the tight cooperation of trust management experts and software engineers without affording a too close insight in the competence of the other expert group. The collaboration-oriented development approach is supported by the Research Council of Norway (RCN) that approved the research and development project ISIS (Infrastructure for Integrated Services). ISIS is mainly devoted to the creation of a tool set supporting the suitable design of collaboration-oriented systems. Moreover, we want to combine the methodologies of collaboration-oriented software design and security protocol composition. As a result of this project, we expect methods facilitating the engineering and deployment of secure and trust-aware distributed systems. The work presented above is considered as a major cornerstone for these research goals. ■

Bibliography

[AD01] Karl Aberer and Zoran Despotovic. Managing Trust in a Peer-2-Peer Information System. In Henrique Paques, Ling Liu, and David Grossman, editors, *Proceedings of the 10th International Conference on Information and Knowledge Management (CIKM'01)*, pages 310–317, New York, November 2001. ACM Press.

[AM04] Farag Azzedin and Muthucumaru Maheswaran. A TrustBrokering System and Its Application to Resource Management in Public-Resource Grids. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, April 2004. IEEE Computer Society Press.

[ARH00] Alfrarez Abdul-Rahman and Stephen Hailes. Supporting Trust in Virtual Communities. In *Proceedings of the 33rd Hawaii International Conference*, volume 6, Maui, Hawaii, January 2000. IEEE Computer Society Press.

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *Proceedings of the 17th Symposium on Security and Privacy*, pages 164–173, Oakland, 1996. IEEE.

[BGH$^+$97] Rolv Bræk, Joe Gorman, Øystein Haugen, Geir Melby, Birger Møller-Pedersen, and Richard Sanders. Quality by Construction Exemplified by TIMe — The Integrated Methodology. *Telektronikk*, 95(1):73–82, 1997.

[Bræ79] Rolv Bræk. Unified System Modelling and Implementation. In *International Switching Symposium*, pages 1180–1187, Paris, France, May 1979.

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley Longman, 1999.

[BS02] Pierro Bonatti and Pierangela Samarati. A Unified Framework for Regulating Access and Information Release on the Web. *Journal of Computer Security*, 10:241–272, 2002.

[CB06] Humberto N. Castejón and Rolv Bræk. A Collaboration-based Approach to Service Specification and Detection of Implied Scenarios. In *SCESM '06: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pages 37–43, New York, NY, USA, 2006. ACM Press.

[Che99] Cheskin Research and Studio Archetype/Sapient. *eCommerce Trust Study*, January 1999.

[FC01] Rino Falcone and Cristiano Castelfranchi. Social Trust: A Cognitive Approach. In Cristiano Castelfranchi and Yao-Hua Tan, editors, *Trust and Deception in Virtual Societies*, pages 55–90. Kluwer Academic Publishers, 2001.

[GS02] Tyrone Grandison and Morris Sloman. Specifying and Analysing Trust for Internet Applications. In *Proceedings of the 2nd IFIP Conference on E-Commerce, E-Business & E-Government (I3E)*, pages 145–157, Lisbon, 2002. Kluwer Academic Publisher.

[Her03] Peter Herrmann. Trust-Based Protection of Software Component Users and Designers. In Paddy Nixon and Sotirios Terzis, editors, *Proceedings of the 1st International Conference on Trust Management*, LNCS 2692, pages 75–90, Heraklion, May 2003. Springer-Verlag.

[Her06] Peter Herrmann. Temporal Logic-Based Specification and Verification of Trust Models. In Ketil Stølen, William H. Winsborough, Fabio Martinelli, and Fabio Massacci, editors, *iTrust 2006*, volume 3986 of *Lecture Notes in Computer Science*, pages 105–119, Heidelberg, 2006. Springer–Verlag.

[HK00] Peter Herrmann and Heiko Krumm. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337, 2000.

[HMP00] Øystein Haugen and Birger Møller-Pedersen. JavaFrame — Framework for Java Enabled Modelling. In Proceedings of Ericsson Conference on Software Engineering, September 2000.

[Ing05] David Ingram. An Evidence Based Architecture for Efficient, Attack-Resistant Computational Trust Dissemination in Peer-to-Peer Networks. In Peter Herrmann, Valérie Issarny, and Simon Shiu, editors, *Proceedings of the 3rd International Conference on Trust Management*, LNCS 3477, pages 273–288, Paris, May 2005. Springer-Verlag.

[JF01] Andrew J I Jones and Babak Sadighi Firozabadi. On the Characterisation of a Trusting Agent — Aspects of a Formal Approach. In Cristiano Castelfranchi and Yao-Hua Tan, editors, *Trust and Deception in Virtual Societies*, pages 157–168. Kluwer Academic Publishers, 2001.

[JK98] Audun Jøsang and S. J. Knapskog. A metric for trusted systems. In *Proceedings of the 21st National Security Conference*. NSA, 1998.

[JO06] Christian D. Jensen and Paul O'Connell. Trust-Based Route Selection in Dynamic Source Routing. In Ketil Stølen, William H. Winsborough, Fabio Martinelli, and Fabio Massacci, editors, *Proceedings of the 4th International Conference on Trust Management*, LNCS 3986, pages 150–163, Pisa, May 2006. Springer-Verlag.

[Jøs96] Audun Jøsang. The right type of trust for distributed systems. In *Proceedings of the UCLA conference on New security paradigms workshops*, pages 119–131, Lake Arrowhead, September 1996. ACM.

[Jøs99] Audun Jøsang. An Algebra for Assessing Trust in Certification Chains. In J. Kochmar, editor, *Proceedings of the Network and Distributed Systems Security Symposium (NDSS'99)*. The Internet Society, 1999.

[Jøs01] Audun Jøsang. A Logic for Uncertain Probabilities. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 9(3):279–311, June 2001.

[KH06] Frank Alexander Kraemer and Peter Herrmann. Service Specification by Composition of Collaborations — An Example. In *Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology)*, pages 129–133. IEEE Computer Society, 2006. 2nd International Workshop on Service Composition (Sercomp), Hong Kong.

[KH07] Frank Alexander Kraemer and Peter Herrmann. Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In Karsten Ehring and Holger Giese, editors, *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, volume 7 of *Electronic Communications of the EASST*. EASST, 2007.

[KHB06] Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk. Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. In R. Meersmann and Z. Tari, editors, *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), 2006, Montpellier, France*, volume 4276 of *Lecture Notes in Computer Science*, pages 1613–1632. Springer–Verlag Heidelberg, 2006.

[KHKR06] Florian Kerschbaum, Jochen Haller, Yücel Karabulut, and Philip Robinson. PathTrust: A Trust-Based Reputation Service for Virtual Organization Formation. In Ketil Stølen, William H. Winsborough, Fabio Martinelli, and Fabio Massacci, editors, *Proceedings of the 4th International Conference on Trust Management*, LNCS 3986, pages 193–205, Pisa, May 2006. Springer-Verlag.

[KM04] Hristo Koshutanski and Fabio Massacci. Interactive Access Control for Web Services. In *Proceedings of the 19th IFIP Information Security Conference (SEC 2004)*, pages 151–166, Toulouse, 2004. Kluwer Academic Publisher.

[KSGM03] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The EigenTrust Algorithm for Reputation Management in P2P Networks. In *Proceedings of the 12th International World Wide Web Conference*, Budapest, May 2003. ACM.

[Lam02] Leslie Lamport. *Specifying Systems.* Addison-Wesley, 2002.

[LTM06] Gabriele Lenzini, Andrew Tokmakoff, and Johan Muskens. Managing Trustworthiness in Component-Based Embedded Systems. In *Proceedings of the 2nd International Workshop on Security and Trust Management*, Hamburg, September 2006.

[LWBW06] Adam J. Lee, Marianne Winslett, Jim Basney, and Von Welch. Traust: A Trust Negotiation Based Authorization Service. In Ketil Stølen, William H. Winsborough, Fabio Martinelli, and Fabio Massacci, editors, *Proceedings of the 4th International Conference on Trust Management*, LNCS 3986, pages 458–462, Pisa, May 2006. Springer-Verlag.

[Mez04] Nicola Mezzetti. A Socially Inspired Reputation Model. In Sokratis K. Katsikas, Stefanos Gritzalis, and Javier Lopez, editors, *1st European Workshop on Public Key Infrastructure (EuroPKI 2004)*, LNCS 3093, pages 191–204, Samos Island, June 2004. Springer-Verlag.

[Obj06] Object Management Group. Unified Modeling Language: Superstructure, version 2.1, April 2006. ptc/2006-04-02.

[Pea05] Siani Pearson. Trusted Computing: Strengths, Weaknesses and Further Opportunities for Enhancing Privacy. In Peter Herrmann, Valérie Issarny, and Simon Shiu, editors, *Proceedings of the 3rd International Conference on Trust Management*, LNCS 3477, pages 305–320, Paris, May 2005. Springer-Verlag.

[PM06] Siani Pearson and Marco Casassa Mont. Provision of Trusted Identity Management Using Trust Credentials. In Ketil Stølen, William H. Winsborough, Fabio Martinelli, and Fabio Massacci, editors, *Proceedings of the 4th International Conference on Trust Management*, LNCS 3986, pages 267–282, Pisa, May 2006. Springer-Verlag.

[QHC06] Daniele Quercia, Stephen Hailes, and Licia Capra. B- Trust: Bayesian Trust Framework for Pervasive Computing. In Ketil Stølen, William H. Winsborough, Fabio Martinelli, and Fabio Massacci, editors, *Proceedings of the 4th International Conference on Trust Management*, LNCS 3986, pages 298–312, Pisa, May 2006. Springer-Verlag.

[RB06] Judith E. Y. Rossebø and Rolv Bræk. Towards a Framework of Authentication and Authorization Patterns for Ensuring Availability in Service Composition. In *Proceedings of the 1st International Conference on Availability, Reliability and Security (ARES'06)*, pages 206–215. IEEE Computer Society Press, 2006.

[SCKB05] Richard Sanders, Humberto N. Castejón, Frank Alexander Kraemer, and Rolv Bræk. Using UML 2.0 Collaborations for Compositional Service Specification. In *ACM / IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, 2005.

[XL02] Li Xiong and Ling Liu. Building Trust in Decentralized Peer-to-Peer Electronic Communities. In *Proceedings of the 5th International Conference on Electronic Commerce Research (ICECR-5)*, Dallas, November 2002. ATSMA, IFIP.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA$^+$ Specifications. In L. Pierre and T. Kropf, editors, *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer-Verlag, 1999.

[YWS03] Ting Yu, Marianne Winslett, and Kent E. Seamons. Supporting Structured Credentials and Sensitive Policies through Interoperable Strategies for Automated Trust Negotiation. *ACM Transactions on Information and System Security*, 6(1):1–42, February 2003.

Comment

In the example, we have specified activities in which several participants have starting and terminating pins. In *Trusted Sale* in Fig. 5.5, for instance, the start and termination is symmetric for the buyer and the seller. Since we did not need this feature in other specifications, we did not further elaborate on the implications of such a symmetric start and termination.

PAPER
SIX

FORMALIZING COLLABORATION-ORIENTED SERVICE SPECIFICATIONS USING TEMPORAL LOGIC

Frank Alexander Kraemer and Peter Herrmann.

Presented at the 6th International Conference on Networking and Electronic Commerce Research Conference 2007 (NAEC 2007) October 18 – 21, 2007 Lake Garda, Italy.

Published in: Bezalel Gavish, ed., Proceedings of the International Conference on Networking and Electronic Commerce Research Conference, p. 194-220, ATSMA Inc. USA, 2007.

Formalizing Collaboration-Oriented Service Specifications using Temporal Logic

Frank Alexander Kraemer and Peter Herrmann

Abstract. In our highly automated engineering approach, reactive services are specified using UML 2.0 collaborations and activities. This enables to focus on complete behaviors between of a set of participants in isolation, and to decompose systems according to the functionalities it should offer. Of course, precise semantics for the specifications are necessary, as we use them as input for model checking and automatic synthesis of components for implementation. For this reason we formalize the concept of collaborations in the temporal logic cTLA by defining the specification style cTLA/c. Collaborations are hereby represented as cTLA processes, and the composition of collaboration can be reduced to process couplings. While cTLA/c is general to capture the semantics of different languages, we show in detail how UML 2.0 activities are mapped to cTLA/c by a set of cTLA processes and production rules.

6.1 Introduction

A networked service is a system offering certain functionalities that are used by concurrently acting entities in its environment. The service functions often render a reactive behavior in the sense that they *"maintain some interaction with their environment"* [Pnu86]. From a physical point of view, such a system naturally decomposes into its components, that means the distributed entities providing the system functionality. In the setting of a model-driven development approach, the components may be expressed for example by SDL processes and blocks [ITU02], or UML state machines and composite structures [Obj07]. These component descriptions form the input for automatic code generation tools (see, e.g., [Bræ79, Flo95, Vef85]).

For a system offering a whole bunch of functionalities to its environment, such a component-based view leads to complex specifications as each component model describes partial aspects of various functionalities. Instead, we desire a specification style in which a specification block models all aspects of a single functionality facilitating the individual development, deployment, invocation and maintenance of separate functionalities. As a functionality is basically a service spanning over several components, we therefore need specifications describing the collaboration of various components. Modeling languages like UML interactions [Obj07], MSC [ITU04] and Use Case Maps [BC96] offer a solution by enabling the description of both partial and collaborative behaviors. We apply UML 2.0 collaborations to express static properties and UML 2.0 activities to model collaborative behavior [HK07, KBH07, KH06].

Of course, the need for component models remains as, in the end, the components are the entities which have to be created and deployed on different devices

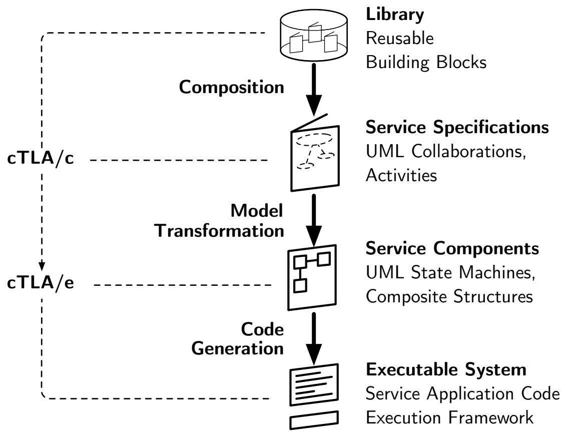


Fig. 6.1: The SPACE Engineering Approach

to realize the system. Consequently, we often find system models utilizing diagrams of several types which describe a system from different viewpoints. This, however, imposes the challenge of keeping the diagrams consistent. Given the need to build services and to adjust their system functions rapidly, approaches that rely on the discipline of its developers to maintain the diagrams manually are rather naive. A consequent way to go is therefore to let developers create only one group of diagrams and infer the others completely automatically, for example by means of model transformations.

Several examples [CB06a, HK07, KBH07, KH06, RB06, SCKB05] illustrate how the notion of collaborations can be used to specify services. All these specifications have in common that they decompose a system according to its tasks, delay the construction of components to a later stage, and identify only participants relevant for the modeled functionality. Such tasks (or sub-functionalities resp. sub-services) often show up in more than one application. They typically have a concise objective or function, resulting in building blocks which can be used for various service descriptions in a particular application domain.

With our approach for the specification by activities, collaborations and executable state machines (SPACE), we develop a tool-supported process that analyzes and transforms collaborative service specifications in a highly automated way [KH06]. The approach is outlined in Fig. 6.1. Often, a new service can be composed from already existing collaboration patterns that may be adapted to the operations and data types specific for the application under construction. An engineer therefore may consider a library of reusable building blocks which are subsequently composed obtaining a composite service specification. To come to an executable system, the service specifications in terms of UML 2.0 activities and collaborations are then transformed into UML 2.0 components and state machines, as detailed in [KH07b]. From this representation, code generation is quite straightforward, as for example explained in [KHB06]. In this way, the engineer just works on the creation of the service specifications, while the rest of the approach is automated, so that consistency is ensured by construction. Tool support for the SPACE approach exists in form of two integrated Eclipse-based tool sets [Kra]. *Arctis* offers support for collaborative service specifications, their analysis and transformation into executable state machines. These can then be further analyzed by *Ramses*, which also offers code generators to create implementations based on Java.

Of course, to guarantee the precise understanding of the models and the correctness of the transformations, the approach requires formal reasoning on the semantics of the languages used, the transformation tools, and the consistency of building blocks and their composition. Temporal logic is a suitable instrument for that. In particular, the principle of superposition supported by cTLA [Her97, HK00] makes it possible to describe systems from different viewpoints by individual processes that are superimposed (see Sect. 6.3.2). Therefore, the development approach in Fig. 6.1 is complemented by formal reasoning (shown on the left side). In a first step, we formalized the behavior of the state machines with cTLA/e [KHB06]. This cTLA style defines a set of constraints for cTLA specifications that directly reflect the special properties of the state machines needed to enable the generation of efficiently executable code. Due to this foundation of the state machines, we can ensure that the generated program code is compliant with the behavior described in the state machines (see [KHB06]).

In this paper, we focus on the definition of cTLA/c, a style of cTLA that allows us to formalize the collaborative service specifications given by UML 2.0 activities. By expressing collaborations as cTLA processes, we can ensure that a composed service maintains the properties of the individual collaborations it is composed from. The semantic definition presented here enables us to prove formally that the transformation from activities to state machines is correctness-preserving. As sketched in [KH07b], this corresponds to substitute that an activity modeled by the cTLA/c specification A is always transformed to a state machine described by cTLA/e model S which is a correct refinement of A (i.e., $S \Rightarrow A$ holds).

The semantic definition of collaborations and activities in form of temporal logic is implemented as a transformation tool [Slå07] which produces TLA$^+$ modules from activities. These modules may then be used as input for the model checker TLC [YML99]. The tool also generates a number of theorems, so that collaborations may be analyzed for more advanced properties than simple syntactic checks allow.

In Sect. 6.2, we introduce service specifications based on collaborations and activities by means of an example. An introduction to general cTLA in Sect. 6.3 is followed by the presentation of the specification style cTLA/c in Sect. 6.4. Thereafter, Sect. 6.5 is devoted to the formal definition of the UML 2.0 activities used in our approach. For that, special cTLA specification blocks to model the behavioral features of the activity nodes are used. In addition, we define a set of production rules guiding the creation of the final cTLA/c specifications according to the nodes and edges of an activity. In Sect. 6.6, we discuss how these specifications modeling elementary service collaborations can be composed

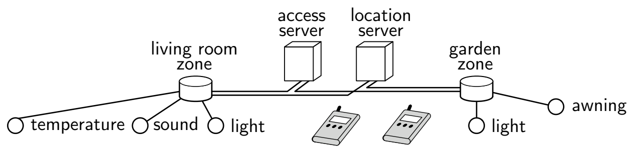


Fig. 6.2: Illustration for a system configuration

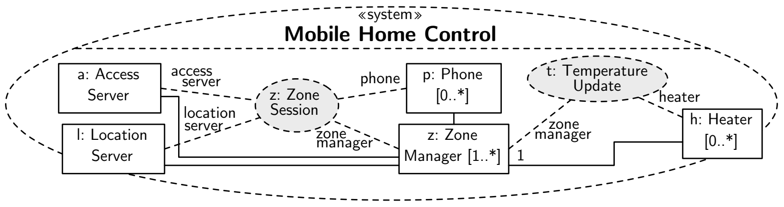


Fig. 6.3: System collaboration

to specify also composite specifications and service descriptions that may handle several users and multiple sessions at a time.

6.2 Specifications in SPACE

As an example, we consider a system for home automation, which allows the residents of a house to control various devices with their mobile phones. Devices may be heaters and volume controls, lights, motors of awnings, or the intercom with the door bell. The house is organized in zones covering different possibly overlapping areas. Each device is assigned to a zone manager, as illustrated in Fig. 6.2 with a zone for the living room and one for the garden. To make the user interface easier, the control options offered by a mobile phone should depend on its current location, so that one may adjust the room temperature of only the room one currently stays in, or roll out the awnings when one is on the terrace. For this reason, we assume that the position of the phones can be determined by a location server with sufficient accuracy (e.g., by equipping them with WLAN capabilities). Via this channel, the mobile phones may also communicate to the zone managers. In addition to the location server, an access server keeps record of the user authorizations and access rights, for example, to grant guests and children only limited control.

6.2.1 UML 2.0 Collaborations

Figure 6.3 shows a UML 2.0 collaboration specifying the structure of the system. For each participant it declares a collaboration role, i.e., a and l for the access and location server, p representing the phones, z for the zone managers and h for a set of heaters.[1] With the stereotype ≪system≫ we express that Fig. 6.3 documents

[1]To keep the example manageable, we look here only at heaters as devices.

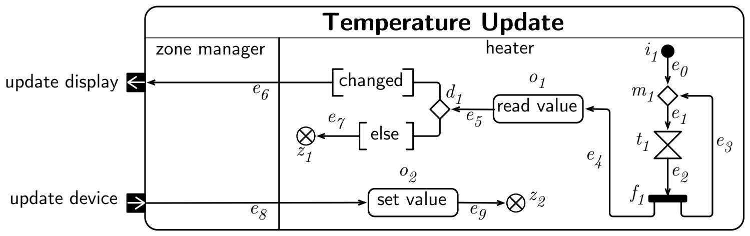


Fig. 6.4: Activity describing the control of the temperature

the highest system level, and that its collaboration roles should be realized as separate components.[2] In addition to the type information, the collaboration roles specify the multiplicity of the components. While access server and location server have default multiplicity “1”, there may be any number of phones in the system and an arbitrary number of zones, which in turn may be connected to several heaters. The multiplicity of the connector end right to the zone manager is “1”. This tells us that a heater is only connected to one zone manager while one zone manager is connected to many devices.

Collaborations may refer to other collaborations by means of collaboration uses, so that a specification may reuse existing building blocks or be decomposed to reduce complexity. In this way, the specification in Fig. 6.3 expresses that a zone communicates with the heaters by collaboration *Temperature Update*, which is bound to the system by collaboration use t. Whenever a phone enters a zone, the location server starts a zone session collaboration between the phone and the entered zone, represented by collaboration use z. The labels at the lines which connect the ellipse of the collaboration use describe to which collaboration roles of the referring collaboration the collaboration roles of the referred collaboration are bound.

6.2.2 Activities for Elementary Collaborations

As mentioned in the introduction, SPACE uses activities to describe the behavior of collaborations. Activities can be understood as token flows, similar to Petri nets. Let us first consider the activity for the elementary collaboration given in Fig. 6.4 for the control of the room temperature with a heater. The activity has two partitions, *zone manager* and *heater*, one for each collaboration role. At startup, a token is emitted from the initial node i_1, which then finds its way through merge node m_1 towards timer t_1, where it starts the timer and rests. When the timer expires, the token is emitted and duplicated within the fork node f_1. One token is then redirected via merge node m_1 back to the timer, which starts again. The other token flows via operation *read value* to decision node d_1. If the temperature changed, the token is sent to the zone

[2]For this reason, we also include the type names (like *ZoneManager*) which will be taken as type names for the components to be generated.

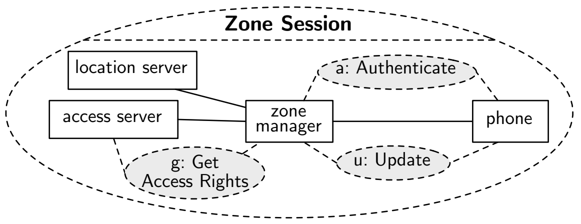


Fig. 6.5: Collaboration for a phone within a zone

manager. For the transmission, we assume a queue place on edges that cross partitions (in the following called *transfer edges*), so that the token first rests within the queue before it is read out by the zone manager. If the change of temperature is insignificant and does not need to be reported to the zone manager, the token flow is ended in the flow final node z_1. A second flow starts at the input parameter node *update device*, forwards to the heater, causes a value change as expressed by the operation o_2, and finally terminates at node z_2.

6.2.3 Activities for Compositional Collaborations

The zone session referred from the collaboration in Fig. 6.3 is further detailed in Fig. 6.5. It is in turn a composite collaboration with the collaboration roles for the location and access server, a zone manager and a phone as its participants. All collaboration roles have their default multiplicity “1”, so that this collaboration describes the cooperation of exactly one phone and one zone manager, covering the behavior whenever a phone is within a certain zone. For that, collaboration uses a, u and g refer to collaboration describing the retrieval of access rights from a server, the authentication of the phone, and the bidirectional update mechanisms between zone and phone. The detailed coupling of these collaboration uses is described by Fig. 6.6. It has again one activity partition for each of the collaboration roles. In addition, the collaboration uses of Fig. 6.5 are represented as call behavior actions g, a and u. They refer to other activities that describe their detailed behavior. Their pins are used to couple them together, using some additional logic in the zone manager and the phone. The collaboration starts when the location server detects that a phone enters a zone, via input parameter node *enter*.[3] Upon that, the zone manager simultaneously invokes via the fork node the sub-collaborations g to retrieve the access rights as well as a to request an authentication from the phone. Once the access right information arrives, a token is placed into the waiting decision node. This is an extension of a normal decision node with the difference that a token rests in it until one of the downstream joins may fire [KH07b]. This is the case when the authentication finishes. Depending on the outcome, either the left join node may eventually fire, which causes the flow to stop, as the user is not authenticated. Upon a successful authentication, the right join fires and the zone

[3]Activity parameter nodes are represented by pins owned by a call behavior actions when their activity is referred to by another activity.

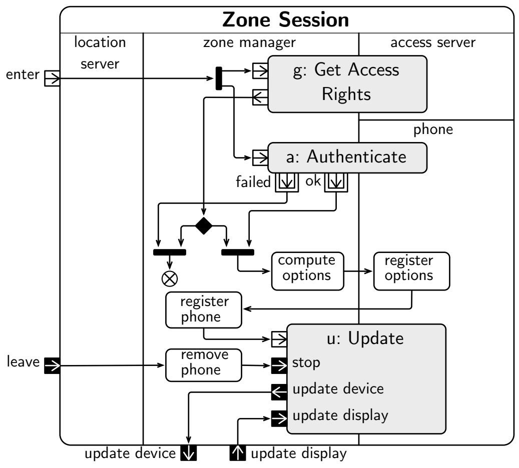


Fig. 6.6: Collaboration for entering and leaving a zone

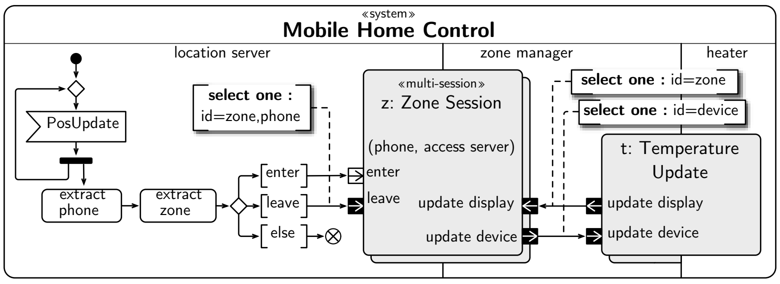


Fig. 6.7: Activity composing the entire system

manager computes the options that are offered to the phone. After the phone has registered the options, the zone manager registers the phone enabling it to handle updates. Collaboration u is started which manages the update handling, so that the phone may send updates to the devices and vice versa.

6.2.4 Multi-Session Collaborations

Within one occurrence of a zone session, each collaboration role was considered only once, and each collaboration use was executed only once at a time. The collaboration for the entire system, *Mobile Home Control* in Fig. 6.3, however, has several zones, heaters and phones. As a consequence, the collaborations z for the zone sessions and t for the temperature updates are executed with several executions (also called sessions) at the same time. The one location server is invoked in several sessions of a zone session (with different zone managers and phones). One zone manager is connected to many heaters, and maintains therefore several temperature update collaboration instances with them at the same time. We emphasize this by adding a shadow-like border to the call behavior actions within those activity partitions that can choose from different collaboration instances. In Fig. 6.7 this is the entire zone session, as both zone manager and location server have to handle several instances. For the temperature update, the collaboration is multiple only within the zone manager (which may be connected to several heaters) but is single within the heater partition, as one heater participates in only one temperature update session as it is connected to only one zone manager. The activity of Fig. 6.7 starts within the location manager that waits for the reception of position updates. Once a position update arrives, it extracts the phone and zone information from the update and decides, if the change of location should be interpreted as the entering or the leaving of a zone. If the phone enters a zone, a zone session is started, if it leaves a zone, the corresponding ongoing zone session is informed (and stopped). In both cases, the token has to enter a specific zone session instance. UML activities may handle several execution at the same time, that means, a call behavior action may refer to several executions that go on at the same time. However, UML does not provide means to distinguish the different session from each other, so that we may compose them in a more advanced manner. We therefore introduced an selection operator, that distinguishes different session instances by a set of filters (see [KBH07]). To enter the zone session, we take the zone and the phone as ID identifying the corresponding session. More filters that also take data within the session instances into consideration, are described in [KBH07].

6.3 Temporal Logic and cTLA

Temporal logic enables to specify behavior which, according to Kurki-Suonio, is the *"abstraction of reactive executions"* [KS05]. Since networked services are reactive by nature, temporal logics are therefore suited to model the service behavior formally. One can distinguish temporal logics in linear time logics (LTL), which express behaviors by sets of infinite sequences of states, and in branching time logics (BTL) modeling state orders by tree structures. While the latter concept offers a higher degree of expressiveness, LTLs often lead to easier understandable specifications.

A well-known LTL is Leslie Lamport's Temporal Logic of Actions (TLA, [Lam02]) in which behavior is described by special state transition systems as well as fairness properties. The TLA coupling method by means of states common to several element specifications [AL95], however, makes it difficult to create constraint-oriented models in which not single physical components but properties reflecting partial system behavior spanning several components are specified [VSvSB91]. As our collaboration-oriented models demand exactly this specification style, we use the compositional Temporal Logic of Actions (cTLA, [Her97, HK00]). This is a variant of TLA which provides couplings based on jointly executed transitions enabling to glue interacting constraints nicely. Moreover, cTLA makes the description of state transition systems in a process-like style possible. A cTLA process can either be in a simple form, mod-

```
PROCESS Timer(TT: Any)
VARIABLES
  i: {"idle", "active"};
  tv:  TT;
INIT ≜ i = "idle" ∧ tv ∈ TT;
ACTIONS
  start(it: TT) ≜
    i = "idle" ∧ i′ = "active" ∧ tv′ = it;
  expire(ot: TT) ≜
    i = "active" ∧ ot = tv ∧ i′ = "idle" ∧ unchanged(tv);
  expireAndRestart(it, ot: TT) ≜
    i = "active" ∧ ot = tv ∧ tv′ = it ∧ unchanged(i);
END
```

Fig. 6.8: cTLA process for Modeling Activity Timers

eling the state transition systems directly, or in a compositional form combining several process instances which interact by the jointly fired transitions. In the following, we introduce both process types in detail.

6.3.1 Simple cTLA Processes

Simple cTLA processes are used to model single resources or constraints of a system. Figure 6.8 depicts a simple process which specifies a timer node of an UML 2.0 activity. In the process header, the process name and a list of process parameters are listed. The parameters enable to model several shapes of process instances by a single process type. For example, the process parameter *TT* describes the signature[4] of the tokens modeled by a particular UML activity so that we can use the process *Timer* for various token formats. As said, a cTLA process models a state transition system, the state of which is described by variables. In the example process, we use the variables *i* distinguishing if the timer is idle or active and *tv* storing the data of an activity token passing it. The set of initial states which hold in the beginning of executing a process are defined by the predicate `INIT`. Here, the variable *i* is initially idle while *tv* contains any data set from *TT*.

The transitions are specified by actions (e.g., *start*) which are predicates on a pair of a current and a next state. Variable identifiers in simple form (e.g., *i*) refer to the current state while variables describing the successor state occur in the primed form (e.g., i'). The conjuncts of an action referring only to variables in the current state specify the enabling condition while those with primed variable identifiers express the state change. Thus, the action *start* is enabled if variable *i* is *"idle"* while its execution leads to a new process state change in which *i* carries the new value *"active"*. Actions may have parameters modeling transfer between processes. For instance, *start* has the parameter *it* of type *TT* describing the data set of a token arriving at the timer which is stored in

[4]Like in colored Petri nets (see [Jen91]), we assume activity tokens to contain special data sets to specify the forwarding of data.

the variable *tv*. Actions can be distinguished into two classes. External actions denoted by the keyword `ACTIONS` may be coupled with actions of other processes. In contrast, internal actions defined in a compartment headed with `INTERNAL ACTIONS` must not be joined with actions of the process environment so that they express purely local process behavior. In the process *Timer* we use only externally visible actions. Moreover, we may provide the actions with fairness assumptions guaranteeing a lively behavior. Since we concentrate in this paper on safety aspects only, we do not discuss that in detail.

Formally, a cTLA process can be expressed as a TLA-formula, the so-called canonical formula C:

$$C \triangleq INIT \ \wedge \ \Box[\, \exists it, ot \in TT : start(it) \ \vee expire(ot) \vee \\ expireAndRestart(it, ot)]_{\langle i, tv \rangle}$$

The conjunct at the left side of the formula states that the predicate `INIT` holds in the first state of every state sequence modeled by C. The conjunct on the right side starts with the temporal operator $\Box$ ("always") specifying that the expression right to it has to hold in all states of all state sequences. The TLA expression $[pp]_{\langle i, tv \rangle}$ defines that either the pair predicate *pp* holds or that a stuttering step[5] takes place in which the annexed variable identifiers do not change their state (i.e., $i' = i \wedge tv' = tv$ holds). As pair predicate *pp* we listed the disjunction of the process actions in which the process parameters are existentially quantified. This models that a state change in the process always corresponds to the execution of one of its actions using any action parameters of the set TT. Thus, the cTLA process specifies that the first process state fulfills `INIT` and that all state changes follow the process actions or are stuttering steps.

As outline above, cTLA processes are special TLA formulas which, however, follow certain constraints facilitating the cTLA-based action couplings. Mainly, a process action may access only variables of the process, it is defined in, and, like in DisCo [KS05], the actions can be uniquely identified which enables a reference of process actions in compositional descriptions. Some other conventions are necessary for guaranteeing liveness properties and are introduced in [Her97].

6.3.2 Compositional cTLA Processes

Compositional cTLA processes specify systems and subsystems as compositions of simple cTLA process instances which cooperate by means of synchronously executed process actions. Data transfer between the simple processes is modeled by aligning the parameters of the coupled process actions. Since the variables of the simple processes are encapsulated and cannot be read or modified by other processes, a system state is defined as the vector of the process variables. The system transitions are modeled by the synchronously executed process actions. Each stateful simple process (i.e., each process in which variables are defined) contributes to a system action by either exactly one process action or by a

[5]Stuttering steps are necessary for carrying out refinement proofs.

```
PROCESS TemperatureUpdate
  (TT: [[temp: NATURALS, ANY]];
   VT: [[tsTemp, tsOldTemp: NATURALS; tsChg: BOOLEAN; ANY]])
CONSTANTS
  ET = {"e1", "e2", "e3", "e4", "e5", "e6", "e7", "e8", "e9"};
  nv1 = [n ∈ VT × TT → VT
          ↦ [[n.1 EXCEPT !.tsOldTemp = n.1.tsTemp
                    EXCEPT !.tsChg = n.1.tsTemp ≠ n.1.tsOldTemp];
  nt1 = [n ∈ TT × TT → TT ↦ [[n.2 EXCEPT !.temp = n.1.tsTemp]] ];
  nv2 = [n ∈ VT × TT → VT ↦ [[n.1 EXCEPT !.tsTemp = n.2.temp]] ];
  nt2 = [n ∈ TT × TT → TT ↦ n.2];
  gu1 = [n ∈ 1..2 × VT × TT → BOOLEAN
          ↦ IF n.1 = 1 THEN n.2.tsChg ELSE NOT n.2.tsChg ];
PROCESSES
  i1: Initial(TT);
  t1: Timer(TT);
  o1: Operation(nv1,nt1);
  d1: Decision(2,gu1);
  e6: Transfer(TT);
  e8: Transfer(TT);
  o2: Operation(nv2,nt2);
ACTIONS
  act3(it: TT; ot: [ET → TT], iv: VT, ov: [ET → VT],
               is: TT, os: SUBSET TT, last: SUBSET ET) ≜
     i1.start(it)
   ∧ t1.start(it)
   ∧ os = {}
   ∧ ov = ["e0" ↦ iv, "e1" ↦ iv]
   ∧ ot = ["e0" ↦ it, "e1" ↦ it]
   ∧ last = {"e1"}
   ∧ e6.Stutter ∧ e8.Stutter;
  ...
END
```

Fig. 6.9: cTLA Process describing the activity *Temperature Update*

stuttering step modeling concurrency as interleaving. In consequence, a system action is a conjunction of process actions and process stuttering steps.

Figure 6.9 models the cTLA specification of the UML activity *Temperature Update* (see Fig. 6.4) as a compositional cTLA process with the same name. The cTLA process is composed from the process instances listed in the section *PROCESSES*. For example, *t1* is an instance of the process type *Timer* introduced above. The process parameter *TT* of *t1* is instantiated with the process parameter *TT* defined as a process parameter in the compositional process. The external and internal system actions are specified in the blocks `ACTIONS` and `INTERNAL ACTIONS` as conjunctions of process actions and process stuttering steps. We depicted the system action act_3 modeling the flow from the initial UML activity node i_1 to the timer t_1. The action is a coupling of the actions *start* in both composed processes *i1* and *t1* while the processes *e6* and *e8* perform stuttering

```
PROCESS TemperatureUpdate
  (TT : [[temp : NATURALS, ANY]] ;
   VT : [[tsTemp, tsOldTemp : NATURALS; tsChg : BOOLEAN; ANY]] )
CONSTANTS
  ET = {"e1", "e2", "e3", "e4", "e5", "e6", "e7", "e8", "e9"};
VARIABLES
  i1xi : {"init","active"};
  t1xi : {idle, active};
  t1xtv: TT;
  e6xq : QUEUE(TT);
  e8xq : QUEUE(TT);
ACTIONS
  act3(it: TT; ot: [ET → TT], iv: VT, ov: [ET → VT],
              is: TT, os: SUBSET TT, last: SUBSET ET) ≜
     i1xi = "init" ∧ it ∈ TT ∧ i1xi′ = "active"
  ∧ t1xi = "idle" ∧ t1xi′ = "active" ∧ t1xtv′ = it
  ∧ ov = ["e0" ↦ iav, "e1" ↦ iv]
  ∧ ot = ["e0" ↦ it, "e1" ↦ it]
  ∧ last = {"e1"}
  ∧ unchanged(e6xq,e8xq);
  ...
END
```

Fig. 6.10: Simple form of cTLA Process *Temperature Update*

steps.[6]

Formally, a compositional cTLA process corresponds to the conjunction of the canonical formulas of the composed simple processes and an additional coupling constraint CC:

$$C \triangleq i1.C \wedge t1.C \wedge \ldots \wedge o2.C \wedge CC$$

The coupling constraint defines the conjunction of the process actions to system actions. Moreover, it defines some constraints on the process action fairness properties which, together with the encapsulation of the process variables, guarantee that the cTLA composition principle fulfills the superposition property (see [Her97]). Superposition [BKS89] ensures that each property of a simple cTLA process holds also for each compositional one including it. As mentioned in the introduction, this is an important ingredient to describe systems from different viewpoints since we can define elementary service functions as separate simple cTLA processes and compose these easily to both collaborative and component-oriented system models. In addition, this property facilitates the formal proofs vastly that component models realize collaboration-based ones.

A compositional cTLA process can be transformed to an equivalent process in simple form as proven in [HK00]. Basically, the simple process comprises the local variables of the included process instances as its variable set while the transitions are modeled by the expanded system actions. As an example, Fig. 6.10 depicts the process *Temperature Update* in simple form. This transformation

[6]The processes *d1*, *o1* and *o2* are not referred to since they are stateless.

from compositional to simple cTLA processes is essential for our UML activity modeling approach as we discuss in Sect. 6.5.

6.4 Formalizing Collaborations

The concept of UML 2.0 collaborations as introduced in Sect. 6.2 is rather structural and as such *"describes a structure of collaborating elements"* [Obj07]. Although UML enables collaborations, being so-called *behaviored classifiers*, refer to behaviors in form of interactions, state machines or activities, the coordination of these behaviors is not elaborated in detail. For our approach, however, in which we want to specify systems completely by composing collaborations, the behavioral part and the coordination of behavioral descriptions are essential. Therefore, we understand collaborations first and foremost as processes jointly executed by a set of participants. Composing systems from collaborations corresponds then to the task of synchronizing these processes. This demands for a precise formal semantics describing both the behavior of collaborations as well as their composition. This does by far not exclude UML; on the contrary, such a well-defined formal basis enables us to use different UML diagrams, utilizing their specific advantages where appropriate. For this reason, we defined the cTLA style cTLA/c used to model collaborations in a way that several diagram types can be formalized.

To illustrate cTLA/c, Fig. 6.11 depicts a collaboration from an abstract, external viewpoint. It is a process between its participants p_1 and p_2. While most of the behavior may be executed internally to the collaboration, we need some mechanism to couple the collaborations with others during composition. For example, the end of one collaboration could trigger the start of another one, or collaborations may exchange data. For this, two principle solutions exist: communication by variables and synchronously executed actions. Only relying on the first one (i.e., allowing only producer/consumer synchronization) implies buffering, so that it would always take two execution steps for a collaboration to influence another one. In some cases, however, following the idea of constraint-oriented modeling (see Sect. 6.3), we may want to describe that events happen at the same time in several collaborations. Thus, both interaction principles can be useful, and cTLA/c is laid out to support both of them. For synchronous couplings, we simply conjoin the cTLA actions of different collaborations, while for buffered communication, we assume that the collaborations are linked to a special collaboration modeling the buffered communication. Both approaches use the cTLA coupling principle of joint actions [Her97, HK00] (see also [KS05]). In Fig. 6.11, the externally visible cTLA actions a_1, a_2 and a_3 can be used to couple C_1 with other collaborations.

6.4.1 Elementary Collaborations

A collaboration that is not composed of other collaborations but describes its behavior directly by its actions and variables is called an *elementary collabo-*

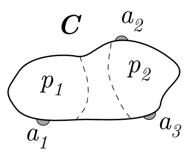


Fig. 6.11: External view of a collaboration C

ration. For this, we use a simple cTLA process, as introduced in Sect. 6.3.1. Ignoring fairness assumptions and process parameters, a simple cTLA process can bee seen as a tuple $P_{simple} = \langle Act_{int}, Act_{ext}, Var, Init\rangle$, which declares the set of its variables, internal and external actions and an initial statement. To describe collaborations, we impose additional invariants on how cTLA processes are written. This basically defines the style of cTLA/c. For an elementary collaboration, we use the tuple

$$C_{el} = \langle Act_{int}, Act_{ext}, Init, Var_{loc}, Var_{com}, Part, p_{var}, p_{act}, NT\rangle.$$

Act_{int}, Act_{ext} and $Init$ have the same meaning as in P_{simple}. In addition, we define the participants of a collaboration by the set $Part = \{p_1 \dots p_n\}$. As these participants describe behavior to be executed by separate, physically distributed components, we assume only buffered communication between the different participants. This communication is expressed by special communication variables defined by the set Var_{com} while the state variables of the participants by Var_{loc} (in which $Var_{loc} \cap Var_{com} = \{\}$ holds and $Var_{loc} \cup Var_{loc}$ forms the set of all variables Var). Function $p_{var} \triangleq [Var \rightarrow Part]$ maps each variable to exactly one participant.

– A *local* variable v_l can be read and written only by the participant assigned to it via the function p_{var}. These variables are used to model local data, status of timers or the history of what has happened so far, to synchronize interactions with other participants.

– A *communication* variable v_c is a ~~bag~~ queue.[7] It can be read by one participant only while the other participants may add elements. We attach v_c to the partition which can read it and constrain the function p_{var} accordingly.

The actions modeling interaction with the environment of the collaboration are described by the set act_{ext}, while act_{int} models behavior that is not visible from the outside. Similarly to the variables, each action is attached to a participant via function $p_{act} \triangleq [Act \rightarrow Part]$. An action a_i attached to participant p_i may access local variables that are assigned to p_i as well. In addition, it may add elements to all communication variables of the neighboring collaborations, and receive from communication variables assigned to p_i. These definitions implied

[7]This was a mistake in the original publication. Communication variables are FIFO queues, as indicated by the symbol in Fig. 6.12 and the cTLA process `Transfer` in Sect. 6.5.1.

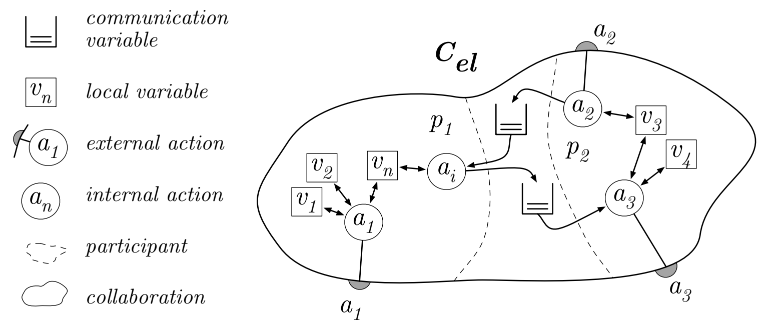


Fig. 6.12: A cTLA/c process for an elementary collaboration C_{el}

by cTLA/c on elementary collaborations are illustrated in Fig. 6.12. It is basically a bipartite graph showing the relationships between actions, participants and local and communication variables.

For the execution of components, we use state machines as expressed by cTLA/e, where actions correspond to state machine transitions that are triggered either by the expiration of a timer or the arrival of a signal. To enable an easier mapping from the actions of cTLA/c to the actions of cTLA/e in form of an implication $S_{cTLA/e} \implies S_{cTLA/c}$, we require also cTLA/c actions to be triggered. As the exact mode of triggering depends highly on the particular concepts of the diagrams formalized, we simply take the tuple element NT to mark all actions that are not triggered. Internal actions have always to be triggered such that NT may only include external actions.

6.4.2 Compositional Collaborations

A *compositional* collaboration refers to other collaborations and composes their behavior to describe its own behavior. For the description of a compositional collaboration, we use the tuple

$$C_{comp} = \langle Act_{int}, Act_{ext}, Part, p_{act}, Cu, bind, NT \rangle$$

As for the elementary collaboration, C_{comp} declares a set of internal and external actions, a set of participants, and a function p_{act} that assigns each action to one participant. In addition, there is the set Cu of collaboration uses. The participants of the instantiated collaborations are mapped to the participants of the collaboration under construction by function $bind \triangleq [Part \rightarrow Part]$. This enables us to use different partition names for the elementary and for the compositional collaborations which can be mapped into each other. Fig. 6.13 gives an illustration. For formal simplicity, there are no variables defined directly within a compositional collaboration. Coupling logic that needs variables and cannot be expressed by action couplings only, can be included by using dedicated collaborations.

As mentioned before, we use joined actions to couple collaborations. In cTLA/c there are some restrictions concerning the topology of the collaborations,

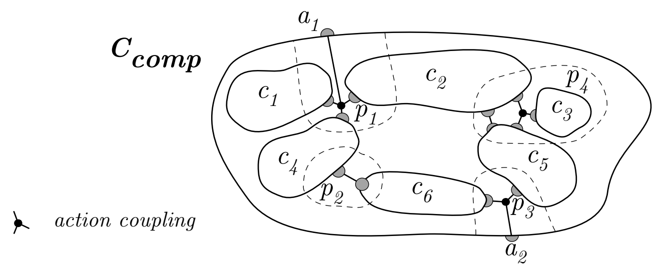


Fig. 6.13: A cTLA/c process for a compositional collaboration C_{comp}

and taking into account that internal actions must be triggered. Each action within a compositional collaboration is a conjunction of one action of each the n collaborations listed in *Cu*:

$$act = \bigwedge_{i=1\ldots n} a_i$$

with a_i being an external action or a stuttering step of collaboration use i. For the action coupling the following constraints must hold:

- Only actions mapped to the same participant may be coupled with each other. Collaborations not bound to the same collaboration role must stutter.
- Within each set of coupled actions, at most one action may be triggering.
- If none of a set of coupled actions has a link to the environment of the composite collaboration C_{comp}, the joint action *act* is internal. In this case, exactly one of the bound actions must be triggered.
- If one of a number of bound actions is linked to the environment of C_{comp}, the joint action is external. Here, either one or none of the joined actions must be triggered.

Following composition concept of cTLA, the compositional cTLA/c tuple C_{comp} can be expanded to an elementary collaboration such that hierarchical collaboration structures are possible.

We abstain from describing a special syntax for the constraints introduced by cTLA/c as we consider it only as a semantic concept behind other languages, such as UML activities, as described in the following. Once a specification semantically has the form of cTLA/c, it can be taken by our transformators and code generators to create an executable system.

6.5 Activities for Elementary Collaborations

In SPACE, we use UML 2.0 activities to express the behavior of collaborations as introduced in Sect. 6.2. A UML 2.0 collaboration is complemented by an activity which uses one separate activity partition for each collaboration role. In the terms of cTLA/c, an activity partition corresponds to a collaboration participant.

As already pointed out in Sect. 6.2, the semantics of UML 2.0 activities is based on Petri nets [Obj07]. Thus, an activity basically describes a state transition system, with the token movements as the transitions and the placement of tokens within the graph as the states. In consequence, the variables of a cTLA/c specification model the actual token placement on the activity while its actions specify the flow of tokens between places. Activity edges may cross partition borders. According to the cTLA/c definition and due to the fact that the partitions are implemented by distributed components, flows moving between partitions are modeled by means of communication buffers while places assigned to activity nodes are represented in cTLA/c by local variables.

We discussed above that certain variables of a cTLA/c collaboration may be triggers. For activities, triggers are represented by initial nodes starting a flow in the beginning, timer nodes which trigger a flow upon expiration, and edges crossing a partition border in which a token in the corresponding communication buffer is triggered to forward in the receiving partition.[8] Moreover, a token may be triggered from the activity environment which is expressed by flows passing pins at the border of call behavior actions. This leads to flows which — from a local view of a single activity — are non-triggered. Of course, in order to achieve lively flows, non-triggered partial flows have to be connected with other flows in a way that in the system description all flows start at a triggering node.

The properties on the triggering of flows lead to constraints on the alignment of places to activity nodes since not every flow could be triggered if a token can rest at any node. In general, we allow places only on nodes modeling either entities that can trigger or locations in which a token has to wait for another flow to synchronize. The first group comprises initial nodes, timer nodes, and crossing points of edges through partition borders while the second one covers the following cases:

– a join node in which a token must wait if the other incoming edges are not yet filled,

– a waiting decision node (see Sect. 6.2) enabling a token to leave via different joins (see [KH07b]), where are token has to wait if none of the succeeding joins can fire.

In contrast, tokens do not rest within call operation actions. This is not useful as no trigger is available which may lead a flow to leave the call operation action after the invoked operation is finished (see [KHB06]). Instead, we consider the execution of the operation “on-the-fly” by a token passing its call operation action.

In the following, an activity is given by the tuple

$$A \triangleq \langle nodes, edges, type, part, location, guard \rangle$$

[8]This definition results from the need to transform activities to the special UML state machines forming the input of our code generators (see [KH07b]).

with *nodes* as the set of activity nodes. $edges \subseteq \text{SUBSET } nodes \times nodes$[9] describes the set of activity edges, while the function $type \in [nodes \rightarrow Type]$ assigns to each activity node the node type which is an element of the set $Type = \{$*initial*, *fork*, *join*, *merge*, *decision*, *waiting*, *timer*, *receive*, *send*, *input*, *output*, *operation*, *callBehavior*, *inputPin*, *outputPin*$\}$. *part* models the set of partitions, while $location \triangleq [nodes \ \cup \ edges \rightarrow \text{SUBSET } part \backslash \{\}]$ assigns each node and edge a non-empty set of partitions. Here, all nodes except for call behavior actions must only be mapped to exactly one partition while edges may belong to several partitions as they can arbitrarily cross partition borders. Guards are assigned to all edges following a decision node by function $guard \triangleq [edges \rightarrow Guards \cup \{\}]$. In addition to the tuple A, we define functions *outgoing* and *incoming* as $\triangleq [node \rightarrow \text{SUBSET } edges]$ that give us the set of incoming and outgoing edges of a node. In particular, only decision and fork nodes have more than one outgoing edge, and only merge and join nodes have more than one incoming edge.

To define the semantics of activities using cTLA/c, we opted for an approach that makes directly use of the composition mechanisms of cTLA.

1. We describe for some node types[10] of an activity a separate cTLA process which are introduced in Sect. 6.5.1. This already helps to understand the semantics of the nodes.

2. To obtain a cTLA/c representation C_A for an activity A, we define C_A as a compositional cTLA process and include for every activity node an instance of the corresponding cTLA process modeling the node.

3. Thereafter, we create the actions of C_A specifying A's flows of tokens. In particular, we traverse the edges of the activity. At a starting point of a flow, a new cTLA action is created which is amended successively when the traversal passes an activity node. The creation and amendment of the actions is guided by a set of production rules introduced in Sect. 6.5.2. In Sect. 6.5.3 we clarify the action creation with an example.

4. As C_A is yet a compositional cTLA specification, we finally expand it to an equivalent simple process as discussed in Sect. 6.3.2. The result of this transformation is the formal model of the activity.

6.5.1 cTLA Processes for Activity Elements

As mentioned above, we model flows passing partition borders by buffered queues. In consequence, the communication variables used for the communication between the participants in the cTLA/c definition are described by buffers storing tokens. Thus, in the tuple C_{el}, introduced in Sect. 6.4, we define the element Var_{com} as the set of queues of tokens containing a separate element for each crossing of an edge through a partition border. Each queue is located at the

[9] SUBSET S is also called power set of S

[10] For merges, forks, and final nodes special cTLA processes are not necessary as we will discuss later.

partition entered by the edge and the tuple element p_{var} is accordingly defined. The places on the activity nodes on which tokens may rest can store at most one token each. For input nodes, timers and waiting decisions, we assign one place for the overall node while joins are provided with a separate place for each incoming edge not leaving a waiting decision. In our cTLA/c processes, every place is described by a boolean flag each modeling if the place is filled by a token or not. Moreover, UML activities enable to use auxiliary variables to express data. As tokens can store data in local signatures,[11] we further need variables storing their current signature if they rest on a place. In consequence, we define the set Var_{loc} of local states as the union of the flags describing the places and the cTLA representations of the auxiliary variables resp. token signature stores. Each place and the assigned signature store are directly linked with an activity node local to a particular partition. In addition, we assume that each auxiliary variable is also local to a partition, so that we can define the mapping p_{var} in a straightforward manner.

The cTLA process types modeling the different activity node types have to fulfill these constraints. Thus, initial nodes, joins (including waiting decisions), timers and receive nodes are represented by cTLA processes defining their places and token signature stores. The cTLA process modeling edges crossing partition borders defines meanwhile the communication queue specifying the partition change. In addition, for each partition, we describe a cTLA process storing the local auxiliary variables. Decision, sending and operation nodes are represented as stateless cTLA processes since that helps to encapsulate their specific properties. For brevity, we introduce only the cTLA processes for initial nodes, decisions, timers, operations as well as for transfer edges which are necessary to understand the example sketched in Sect. 6.5.3. A complete documentation comprising the cTLA processes for all activity nodes is provided in [KH07a].

Before discussing the cTLA processes in detail, we introduce some generic data types used as process parameters. *VT* describes the types of all auxiliary variables in a partition. Here, we assume that a list of variables is expressed by a single record element. The signature set of the tokens is represented by the type *TT*, while *ET* is an enumeration providing each edge in an activity a unique identifier.

Initial Nodes The variable i is the flag describing the place at an initial node. The place is only filled in the initial system state (value *"idle"*) while it will remain empty when the activity is running (value *"active"*). The leaving of the token from the initial node is modeled by the action *start* which must only be executed if the token is in its place (i.e., i = *"idle"*) and removes the token (i' = *"active"*). The action parameter t specifies the signature of the token. Since that is not defined explicitly, it may contain any correct value of set *TT*. *start* is a trigger action modeling the start of a new token flow.

```
PROCESS Initial(TT: Any)
VARIABLES i: {"init","active"};
```

[11]This implies UML object flows [Obj07], which we do not consider here in detail.

```
INIT ≜ i = "init";
ACTIONS
  start(t: TT) ≜ i = "init" ∧ t ∈ TT ∧ i′ = "active";
END
```

Timer Nodes A timer node[12] contains also a place on which a token may rest. In the corresponding cTLA process that was already introduced in Fig. 6.8, we use a boolean flag i and a store *tv* for the token signature. An idle timer is activated by an arriving token, represented by the cTLA action *start*. This action uses the parameter *it* to model the parameters of the arriving token. It is enabled if the place is empty (i.e., i = *"idle"*) which will, consequently, being filled with the token (i.e., *i' = "active"* ∧ *tv' = it*). As we do not model time explicitly yet, the timer can expire at any time, described by the action *expire* which can only be executed if the place is filled. Here, i is set to *"idle"* and the parameter *ot* specifying the signature of the leaving token is assigned with *tv*. The third action, *expireAndRestart* models that the timer expires but is restarted within the same step. This extra action is needed, as a conjunction of action *start* and *expire* would assign contradicting values to the state which would block it forever. *expire* and *expireAndRestart* are trigger actions.

Transfer Edges The queue modeling the transfer of a token from one partition to another one is modeled by the variable q. It stores for every received token the corresponding signature and delivers this information in FIFO order. The arrival of a token with the signature *it* at the partition border is specified by the action *send* while *receive* models the consumption of a token with signature *ot*. According to this definition, the action *start* is assigned to the partition from which the edge leads to the partition border while *receive* is part of the one consuming the token. *receive* is a trigger action.

```
PROCESS Transfer(TT : Any)
VARIABLES
  q: Queue(TT);
INIT ≜ q = EMPTY;
ACTIONS
  send(it: TT) ≜ q′ = append(q,it);
  receive(ot: TT) ≜ q ≠ EMPTY ∧ ot = first(q) ∧ q′ = tail(q);
END
```

Call Operation Actions An operation may change the values of local auxiliary variables of the partition, in which it is defined, as well as the signature of the token flowing through it. We describe operations by the stateless cTLA process *Operation*, which takes two functions as parameters *nv* and *nt*. These parameters reflect that a call operation action may change both the signature of the tokens and the auxiliary variables. Consequently, *nv* is a function that describes the operation's effect on the values of the auxiliary variables. Similarly, *nt* describes the deriving of new token values. The method *execute* models the

[12]Technically, a timer node is a UML accept event action with a time event as its trigger.

computation of new values according to these functions. As action parameters it uses *iv* expressing the auxiliary variable setting and *it* specifying the token signature before executing the operation. The new value of the auxiliary variables and the new token signature are described by the action parameters *ov* resp. *ot*.

```
PROCESS Operation(nv: [VT × TT → VT]; nt: [VT × TT → TT])
ACTIONS
  execute (iv: VT; it: TT; ov: VT; ot: TT) ≜
      ov = nv[iv,it] ∧ ot = no[iv,it];
END
```

Decision Nodes A decision is specified by a stateless cTLA process, too. It may have n outgoing edges modeled by the process parameter of the same name. The other parameter is a function characterizing the guards of the outgoing edges. Its domain set is a tuple defining the identifier of the guard as a number between 1 and n as well as the current value of the auxiliary variables and the token signature. The tuples are mapped to boolean values. The action *decide* reflects a semantics according to which exactly one guard of a decision node has to be true. The parameter e refers to the number of the checked guard and the action may only be executed for this guard if all guards with smaller numbers *ed* are not executable and either the guard of e holds or e is the highest number. The latter condition reflects that one guard should always contain the value *else*.

```
PROCESS Decision (n: NATURALS; gu: [1..n × VT × TT] → BOOLEAN])
ACTIONS
  decide(e: 1..n; av: VT; t: TT) ≜
    ∀ ed ∈ {1..n}:
      ed < e ⇒ ¬ gu[ed,av,t] ∧ e = n ∨ gu[e,av,t];
END
```

6.5.2 Production Rules for cTLA/c Actions

The processes for the activity nodes explained in the last section are instantiated as part of the cTLA process for the activity C_A and constitute the state space for this process. They also declare actions for their respective nodes, which, in the following, have to be coupled in accordance with the activity edges building the system edges of C_A.

We decided to present the way producing the system actions from the local process actions as a set of rules, so that each activity element can be discussed separately. There are two types of rules:

- Rules that *create* a new action. These rules treat triggering nodes like timers or incoming transfer edges. As well as edges starting at an input or output pin of a call behavior action. They simply start the construction of a new action in C_A.

- Rules that *replace* an existing action. These rules model the continuation of a flow. They start at an edge that is not triggering, take the already produced action *act* for the upstream graph and add a conjunct c to the existing action, so that a new action $act^\star = act \wedge c$ is created. This new action replaces the existing one. Except for the special case in which a flow reaches the activity node that triggered it, this replacement corresponds to a cTLA process composition. The existing action is encapsulated as external action within a process and then composed in a compositional process together with another process encapsulating the additional subaction c. The result can then be expanded to a simple cTLA process, which is equivalent to a (maybe more intuitive) replacement of the action. If a flow reaches the node from which it started, we have to replace the action specifying the triggering by another one modeling both the triggering and the consumption of a token. E.g., for a timer, the action *expire* defined in process *Timer* (see Fig. 6.8) has to be exchanged by *expireAndRestart*. In this case, we have a genuine replacement.

Each production rule is presented in two parts. The first compartment collects the preconditions of the rule. It refers to the structure of an activity and defines the activity edges resp. nodes for which the rule can be used. Moreover, the cTLA action to be replaced is listed. As an additional precondition, we need to remember when traversing an activity which of its edges have still to be visited. In a production rule, we therefore use the function $toVisit \in [Act \rightarrow \textsc{subset}\ ET]$ storing for a particular cTLA action the edges still to be passed.

The second compartment shows the effect of the rule. It gives instructions whether a new action should be created or an existing one should be replaced, and how the emerging action is constructed. It also declares any changes to the function $toVisit$ by updating the set of edges still to be visited for an action.

The construction of an action begins with one of the starting points of an activity, that means at initial nodes, at the exit of timers (which means expiration), when an edge enters a partition, or when an external signal arrives. The rules INITIAL, TIMEREXPIRE, TRANSFERENTER and RECEIVE introduced below describe hereby how the action is written. Afterwards, other rules are applied to it guided by the nodes and edges that follow in the activity graph. A new action is created by adding conjuncts to the original one. In case we reach a decision or join node, the action created from the incoming graph is replaced by an entire set of actions. The construction of an action is finished when a new stable state is reached in the activity partition, that means that we either leave the partition, rest in a join or waiting decision, set a timer or reach a final or receiving node. Moreover, the leaving of a flow through the pin of a call behavior action is also a stopping point.

The actions under construction have the signature

```
        act(it: TT; ot: [ET → TT]; iv: VT; ov: [ET → VT];
            is, os: SUBSET TT; last: SUBSET ET)
```

The parameter *it* specifies the value of the token when the flow starts. While the function *ot* describes the token signature after leaving a particular edge. Similarly, parameters *iv* and *ov* describe the values of the local variables after the flow starts and after traversing a particular edge. Signals sent within an action are described with parameter *os*, and signals received by *is*. Parameter *last* keeps track of the edges in an action after those the flow stops. This is needed to support the storage of the auxiliary variables discussed in Sect. 6.5.1.

In the following we will show the rules for initial nodes, merges, forks, timers, operations, transfer edges as well as decisions and flow final nodes. The remaining rules are listed in [KH07a].

Initial Nodes As an initial node is a trigger, it is the startpoint for the production of an action. The rule is enabled for an initial node i with an outgoing edge e. It creates action *act*, which is coupled with action *start* to the process instance corresponding to the initial node, p_i. As the flow is not yet finished, *last* is empty. The node neither produces any output signals ($os = \{\}$) and does not change the value of the variables or token, so that *ov* and *ot* remember their respective initial values for this edge. As we continue the production of the action with whatever comes after edge e, we store it as still to be visited.

INITIAL

$\exists\, i, e : \; type(i) = \text{“}initial\text{”}$
$\qquad\quad outgoing(i) = \{e\}$

$\rightarrow$ *Create* `act` *with*

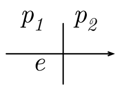
```
act(it: TT; ot: [ET → TT]; iv: VT; ov: [ET → VT];
                 is, os: SUBSET TT; last: SUBSET ET) ≜
     pi.start(it)
   ∧ ov = [ e ↦ iv ] ∧ ot = [ e ↦ it ]
   ∧ os = {} ∧ last = {}
```

$toVisit' = [toVisit \text{ EXCEPT } ![act^\star] = \{e\}]$

Transfer Edges Edges crossing partition borders are handled by two rules, TRANSFERLEAVE modeling the leaving of the current partition, and TRANSFER-ENTER for edges entering a partition. TRANSFERLEAVE is a rule that adds a conjunct invoking *send* on process p_t modeling the buffered communication. As the flow ends where it leaves the partition, the edge is removed from the edges that must be visited but entered to the set *last* describing final edges. The rule for receiving (not shown) is similar to the expiration of a timer or an initial node. It creates a new action referring to the triggered action *receive* of the transfer process.

TransferLeave

$\exists\, e, p_1, p_2, act: \; e \in edges$
$\quad location(e) = \langle p_1, p_2 \rangle$
$\quad e \in toVisit[act]$



$\rightarrow$ *Replace* `act` *by* `act`* *with*

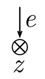
```
act*(...) ≜ ∃ last_o: act(it, ot, iv, ov, is, os, last_o)
  ∧ p_t.send(ot[e]) ∧ last = last_o ∪ {e}
```

$toVisit' = [toVisit \text{ EXCEPT } ![act^\star] = toVisit[act] \setminus \{e\}]$

Flow Final Nodes Flow final nodes simply terminate a token flow. The original action is finished by noting its last edge.

FlowFinal

$\exists\, z, e, act: \; type(z) = \text{“flowFinal”}$
$\quad incoming(z) = \{e\}$
$\quad e \in toVisit[act]$



$\rightarrow$ *Replace* `act` *by* `act`* *with*

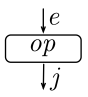
```
act*(...) ≜ ∃ last_o: act(it, ot, iv, ov, is, os, last_o)
  ∧ last = last ∪ {e}
```

$toVisit' = [toVisit \text{ EXCEPT } ![act^\star] = toVisit[act] \setminus \{e\}]$

Call Operation Actions As described above, operations are modeled as functions that assign new values to the token passing through it as well as the variables, modeled by the cTLA process *Operation* with its action *execute*. This action is coupled with the original one, and the production continues with the outgoing edge of the operation. The values *ot* and *ov* for the outgoing edge *j* reflect the changes carried out by the operation which are described by the action parameters new_v and new_t.

OPERATION

$\exists\, op, e, j, act:\ type(op) = \text{“operation”}$
$\quad incoming(op) = \{e\}$
$\quad outgoing(op) = \{j\}$
$\quad e \in toVisit[act]$



$\rightarrow$ *Replace* `act` *by* `act*` *with*

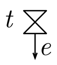
```
act*(...) ≜ ∃ ot_o, ov_o, new_v, new_t:
      act(it, ot_o, iv, ov, is, os, last)
    ∧ p_o.execute(iv_o[e],ot_o[e],new_v,new_t)
    ∧ ov = [ov_o EXCEPT !j ↦ new_v]
    ∧ ot = [ot_o EXCEPT !j ↦ new_t]
```

$toVisit' = [toVisit \text{ EXCEPT } ![act^\star] = toVisit[act] \cup \{j\} \setminus \{e\}]$

Timer For a timer, three rules determine the creation and coupling of actions. The expiration of a timer triggers an action. Rule TIMEREXPIRE defines therefore the creation of a new action that starts with the outgoing edge of the timer, similarly to an initial node. It couples the *expire* action of the timer process p_t, so that this action is only enabled if the timer is active.

TIMEREXPIRE

$\exists\, t, e:\ type(t) = \text{“timer”}$
$\quad outgoing(f) = \{e\}$



$\rightarrow$ *Create* `act` *with*

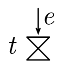
```
act(...) ≜
     p_t.expire(it)
   ∧ ov = [e ↦ iv] ∧ ot = [e ↦ it]
   ∧ os = {} ∧ last = {}
```

$toVisit' = [toVisit \text{ EXCEPT } ![act^\star] = \{e\}]$

To start a timer, action $p_t.start()$ is conjoined with the action modeling the rest of the subgraph. It must only be used if the flow started from another element than the timer itself. In the precondition, this is stated by adding the condition $t \notin visited[act]$ describing that the timer t is not in the list of visited nodes.

TIMERSTART

$\exists\, t, e, act:\ type(t) = \text{“}timer\text{”}$
$\quad incoming(f) = \{e\}$
$\quad e \in toVisited[act]$
$\quad t \notin visited[act]$



$\rightarrow$ *Replace* `act` *by* `act`* *with*

```
act*(...) ≜
  ∃ last_o: act(it, ot, iv, ov, is, os, last_o)
    ∧ p_t.start(ot[e])
    ∧ last′ = last ∪ {e}
```

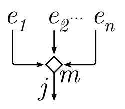
$toVisit' = [toVisit \text{ EXCEPT } ![act^\star] = toVisit[act] \setminus \{e\}]$

Rule TIMEREXPIRERESTART (not shown) is used instead of TIMERSTART if a timer expires and is restarted within the same action. This is stated in the precondition by $t \in visited[act]$. For the result, the only difference is that action *expireAndRestart* instead of just *expire* or *start* is called.

Merge Nodes Merge nodes copy the behavior following the node to the behavior started before the node. The rule is applied to all actions already produced for each of the incoming edges. As an merge does neither change the token signature nor the values of the auxiliary variables, *ot* and *ov* are set to the same values for j as for the incoming edge e_p.

MERGE

$\exists\, m, e_p, j, act:\ type(m) = \text{“}merge\text{”}$
$\quad e_p \in incoming(m)$
$\quad outgoing(m) = \{j\}$
$\quad e_p \in toVisit[act]$



$\rightarrow$ *Replace* `act` *by* `act`* *with*

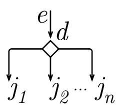
```
act*(...) ≜
  ∃ ov_o, ot_o: act(it, ot_o, iv, ov_o, is, os, last)
  ∧ ov = [ov_o EXCEPT !j ↦ ov_o[e_p] ]
  ∧ ot = [ot_o EXCEPT !j ↦ ot_o[e_p] ]
```

$toVisit' = [toVisit \text{ EXCEPT } ![act^\star] = toVisit[act] \cup \{j\} \setminus \{e_p\}]$

Decision Nodes Decision nodes multiply the incoming actions by the number of its outgoing edges (the alternatives). Therefore, the incoming action is replaced by a set of actions. The original content of the incoming action is maintained, it is just expanded with the additional call of the decision action of the decision process.

DECISION

$\exists\, d, e, act:\ type(d) = \text{“decision”}$
$incoming(d) = \{e\}$
$outgoing(d) = \{i_1 \ldots j_n\}$
$e \in toVisit[act]$



$\rightarrow$ *Replace* `act` *by* $\mathtt{act}_p$ *for all* $p \in \{1 \ldots n\}$ *with*

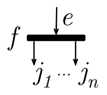
```
act_p(...) ≜ ∃ ov_o, ot_o: act(it, ot_o, iv, ov_o, is, os, last)
    ∧ p_d.decide(p, ov[e], ot[e])
    ∧ ov = [ov_o EXCEPT !j_p↦ ov_o[e] ]
    ∧ ot = [ot_o EXCEPT !j_p↦ ot_o[e] ]
```

$toVisit' = [toVisit \text{ EXCEPT } ![act_p] = toVisit[act] \cup \{j_p\} \setminus \{e\}]$

Fork Nodes Forks multiply a token and emit one token on each outgoing edge. All the behaviors implied until all tokens rest, are executed within one step, so that the whole behavior has to be modeled within one action. Therefore, all outgoing edges are added to the edges still to be visited for the action under construction. Since the fork does not change the value of the tokens or variables, the functions for token and variable values are updated to match the incoming values for each outgoing edge.

FORK

$\exists\, f, e, \{j_1 \ldots j_n\}, act:\ type(f) = \text{“fork”}$
$incoming(f) = \{e\}$
$outgoing(f) = \{j_1 \ldots j_n\}$
$e \in toVisit[act]$



$\rightarrow$ *Replace* `act` *by* $\mathtt{act}^\star$ *with*

```
act*(...) ≜ ∃ ov_o, ot_o: act(it, ot, iv, ov, is, os,last)
   ∧ ov = [ov_o EXCEPT !j_1 ↦ ov_o[e] EXCEPT ! ···
                EXCEPT !j_n ↦ ov_o[e] ]
   ∧ ot = [ot_o EXCEPT !j_1 ↦ ot_o[e] EXCEPT ! ···
                EXCEPT !j_n ↦ ot_o[e] ]
```

$toVisit' = [toVisit \text{ EXCEPT } ![act^\star] = toVisit[act] \cup \{j_1, \ldots, j_n\} \setminus \{e\}]$

The produced action couplings conform to the constraints in cTLA/c.

– The production of an action always stays within the partition where the production started. Edges leaving a partition terminate the production of an action by a corresponding send action of a transfer process. Consequently, all produced actions can be assigned to exactly one participant of the cTLA/c process under construction.

– Actions are by default internal (i.e., $\in Act_{int}$). Only if they pass an input or output node (such as *update display* or *update device* in Fig. 6.4), they are declared external.

– Just for flows starting at an input or output node, actions are created that do not contain a trigger. According to the definition above, however, these actions are external and the cTLA/c claims for non-triggered actions are met. These actions are added to the set *NT* listing the non-triggered actions. Due to the structure of activities and the layout of the rules, a sub-graph corresponding to an action can never contain more than one trigger.[13]

6.5.3 Example

We will now use the rules to produce parts of the cTLA/c process for the activity *Temperature Update* given in Fig. 6.4. First, we instantiate processes *i1*, *t1*, *o1*, *d1*, *e6*, *e8* and *o2* for the corresponding activity nodes, as shown in the corresponding cTLA process in Fig. 6.9. In the following, we will stepwise create some of the actions for the partition of the heater.

Step 1: We choose to start with the initial node i_1 and apply rule INITIAL to edge e_0, which leads to the construction of action act_1. As no signal has yet been sent, *os* is empty. There is no final edge, as the flow continues. The variables did not change with the initial node, such that *ov* notes the original value *iv* for edge e_0. The same applied for the value of the token managed by *ot*.

```
act₁(it: TT; ot: [ET → TT]; iv: VT; ov: [ET → VT];
     is, os: SUBSET TT; last: SUBSET ET) ≜
     i1.start(it)
  ∧  ov = ["e0"↦ iv] ∧ ot = ["e0"↦ it]
  ∧  os = {} ∧ last = {}
```

Step 2: We continue with this flow by applying rule MERGE to edge e_1 and the already created action act_1. It is replaced by act_2, which is an extension of act_1. A merge does not change tokens or variables, so *ov* and *ot* are complemented with an entry for edge e_1.

```
  act₂(...) ≜
  ∃ ov₀, ot₀: act₁(it, ov₀, iv, ot₀, os, is, last)
    ∧ ov = [ov₀ EXCEPT !"e1"↦ ov₀("e0")]
```

[13]In faulty activities, sub-graphs may exist that have neither a triggering element nor a connection via a parameter node. These constructs would not cause an action to be produced, as none of the actions could be applied in the first place. These situations may be detected by syntactic inspections. As such sub-graphs do not express any useful behavior, they are forbidden.

```
    ∧ ot = [ot_o EXCEPT !"e1"↦ ot_o("e0")]
```

We expand act_2 by replacing act_1 with its actual definition:

```
  act_2(...) ≜
  ∃ ov_o, ot_o:
       i1.start(it)
    ∧ ov_o = ["e0"↦ iv] ∧ ot_o = ["e0"↦ it]
    ∧ os = {} ∧ last = {}
   ∧ ov = [["e0"↦ iv] EXCEPT ! "e1"↦ ["e0"↦ iv]("e0")]
   ∧ ot = [["e0"↦ it] EXCEPT ! "e1"↦ ["e0"↦ it]("e0")]
```

We can replace the existentially quantified terms ov_o and ot_o by the equal function definitions. Further, $[”e_0” \mapsto x](”e_0”)$ is of course x, so that we can simplify act_2:

```
act_2(...) ≜
     i1.start(it)
  ∧ os = {} ∧ last = {}
  ∧ ov = ["e0"↦ iv, "e1"↦ iv]
  ∧ ot = ["e0"↦ it, "e1"↦ it]
```

Step 3: Rule TIMERSTART is now applicable to act_2 and edge e_1. It extends act_2 by adding action *t1.start* from the timer process and updates last.

```
act_3(...) ≜
  ∃ last_o: act_2(it, ot, iv, ov, is, os, last_o)
  ∧ t1.start(ot["e1"])
  ∧ last = last_o ∪ {"e1"}
```

After expansion of act_2 and removal of true conjuncts, we get

```
act_3(it: TT; ot: [ET → TT]; iv: VT; ov: [ET → VT];
     is, os: SUBSET TT; last: SUBSET ET) ≜
     i1.start(it)
  ∧ os = {}
  ∧ ov = ["e0"↦ iv, "e1"↦ iv]
  ∧ ot = ["e0"↦ it, "e1"↦ it]
  ∧ t1.start(it)
  ∧ last = {"e1"}
```

No more rules can be applied to act_3, as there are no more edges to visit for this action. The action is complete now and can be added to the cTLA process describing *Temperature Update* (see Fig. 6.9).

Step 4: Rule TIMEREXPIRE can be applied to edge e_2, which results in the creation of act_4:

```
act_4(...) ≜
     t1.expire(it)
  ∧ ov = ["e2"↦ iv]
  ∧ ot = ["e2"↦ it]
  ∧ os = {} ∧ last = {}
```

Step 5: Edge e_2 flows into fork f_1, so that rule FORK may be applied to act_4. It replaces act_4 by act_5 (here with act_4 already expanded).

```
act5(...) ≜
     t1.expire(it)
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv ]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it ]
  ∧ os = {} ∧ last = {}
```

$toVisit[act_5]$ contains now both outgoing edges, e_3 and e_4.

Step 6: Following edge e_3 into m_1, rule MERGE is applicable. It simply complements *ov* and *ot* with entries for edge e_1.

```
act6(...) ≜
     t1.expire(it)
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv, "e1"↦ iv ]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it, "e1"↦ it ]
  ∧ os = {} ∧ last = {}
```

Step 7: Continuing edge e_1 we enter timer t_1. As the currently traversed activity flow started at this node, we apply rule TIMERSETEXPIRE instead of rule TIMERSET. Therefore, the action *expire* of process *t1* is replaced by *expireAndRestart* that handles a flow immediately restarting the timer from which is was triggered.

```
act7(...) ≜
     t1.expireAndRestart(it, it)
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv, "e1"↦ iv ]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it, "e1"↦ it ]
  ∧ os = {} ∧ last = {"e1"}
```

Step 8: To handle the operation by following edge e_4, we replace act_7 with act_8 that adds the conjuncts according to rule OPERATION. The operation is a function *o*1*nav* that computes new values for all the variables in the partition, and an *o*1*nto* that computes the value of a new token.

```
act8(...) ≜
     t1.expireAndRestart(it, it)
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv, "e1"↦ iv,
          "e5"↦ o1nav[iv,it]]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it, "e1"↦ it,
          "e5"↦ o1nto[iv,it] ]
  ∧ os = {} ∧ last = {"e1"}
  ∧ op.execute(iv, it, o1nav[iv,it], o1nto[iv,it])
```

Step 9: We apply rule Decision. It replaces action act_8 with one action for each outgoing branch. For the branch of edge e_6 we get act_9, for the e_7 branch we get act_{10}

```
act9(...) ≜
     t1.expireAndRestart(it, it)
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv, "e1"↦ iv,
          "e5"↦ o1nav[iv,it], "e6"↦ o1nav[iv,it]]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it, "e1"↦ it,
          "e5"↦ o1nto[iv,it], "e6"↦ o1nto[iv,it]] ]
  ∧ os = {} ∧ last = {"e1"}
  ∧ op.execute(iv, it, o1nav[iv,it], o1nto[iv,it])
  ∧ d1.decide(1,o1nav[iv,it], o1nto[iv,it])

act10(...) ≜
     t1.expireAndRestart(it, it)
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv, "e1"↦ iv,
          "e5"↦ o1nav[iv,it], "e7"↦ o1nav[iv,it]]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it, "e1"↦ it,
          "e5"↦ o1nto[iv,it], "e7"↦ o1nto[iv,it]] ]
  ∧ os = {} ∧ last = {"e1"}
  ∧ op.execute(iv, it, o1nav[iv,it], o1nto[iv,it])
  ∧ d1.decide(2,o1nav[iv,it], o1nto[iv,it])
```

Step 10: When we continue with action act_9 and edge e_6, we apply rule TransferSend and replace it by action act_{11}, that simply adds an conjunction sending the token. As the set of edges to visit is empty for this edge, this is an action present in the final process.

```
act11(...) ≜
     t1.expireAndRestart(it, it)
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv, "e1"↦ iv,
          "e5"↦ o1nav[iv,it], "e6"↦ o1nav[iv,it]]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it, "e1"↦ it,
          "e5"↦ o1nto[iv,it], "e6"↦ o1nto[iv,it] ]
  ∧ os = {} ∧ last = {"e1", "e6"}
  ∧ op.execute(iv, it, o1nav[iv,it], o1nto[iv,it])
  ∧ d1.decide(1,o1nav[iv,it], o1nto[iv,it])
  ∧ e6.send(o1nto[iv,it])
```

Step 11 Also *act*10 may be finalized by applying rule FlowFinal with edge e_7. We obtain action act_{12}, which is like act_3 and act_{11} one of the final coupling actions.

```
act12(...) ≜
     t1.expireAndRestart(it, it) this has input and output
  ∧ ov = ["e2"↦ iv, "e3"↦ iv, "e4"↦ iv, "e1"↦ iv,
          "e5"↦ o1nav[iv,it], "e7"↦ o1nav[iv,it]]
  ∧ ot = ["e2"↦ it, "e3"↦ it, "e4"↦ it, "e1"↦ it,
          "e5"↦ o1nto[iv,it], "e7"↦ o1nto[iv,it]] ]
```

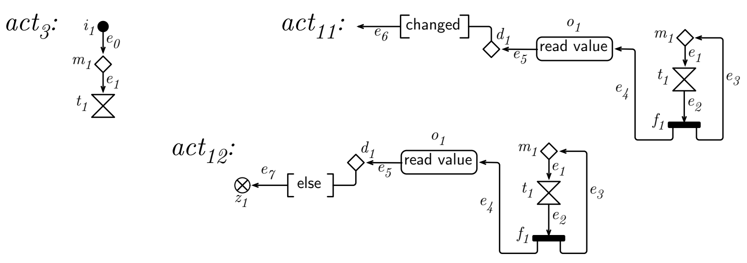


Fig. 6.14: The subgraphs covered by the produced actions

```
∧ os = {} ∧ last = {"e1", "e7"}
∧ op.execute(iv, it, o1nav[iv,it], o1nto[iv,it])
∧ d1.decide(2,o1nav[iv,it], o1nto[iv,it])
```

Above, we sketched the production of the three actions act_3, act_{11} and act_{12} modeling the flows listed in Fig. 6.14. In a similar way, we can produce the other cTLA actions modeling flows in the activities of our *Mobile Home Control* example.

All-in-all, the production rules give a powerful means to formalize UML 2.0 activities as they are used in SPACE. If necessary, the generation process can be automated as done in [Slå07] for checking activities with the model checker TLC[14] [YML99].

6.6 Composing Collaborations by Activities

Following the style of cTLA/c, a composite activity A referring to n sub-activities $A_1, \ldots, A_n$ is modeled by a compositional cTLA/c tuple C_{comp} as described in Sect. 6.4.2. For each collaboration use (and consequently call behavior action of A) that is declared in the UML specification, $C_{comp}.Cu$ contains the elementary cTLA/C tuples $c_{el_1}, \ldots, c_{el_n}$. The compositional cTLA process C realizing C_{comp} includes the cTLA process instances $c_1, \ldots, c_n$ specifying the elementary collaborations. Besides the behavior within the call behavior actions, there may be arbitrary complex logic in A, coupling the referred sub-activities. This behavior of A without its sub-activities is an activity itself represented in C as its own cTLA/c process c_0. Thus, if A contains n call behavior actions, the composite collaboration $C_{comp}.Cu$ has $n+1$ processes as elements, $c_0, \ldots, c_n$. As an example, we refer to the activity *Zone Session* depicted in Fig. 6.6. The mapping from the activity to a compositional cTLA/c process is illustrated in Fig 6.15. Activity A is cut into the activities modeled by the call behavior actions $g \equiv A_1$, $a \equiv A_2$, and $u \equiv A_3$ as well as the one surrounding the call behavior actions A_0. These are expressed by simple cTLA/c processes.

The participants $C_{comp}.Part$ of the compositional cTLA/c process correspond to the activity partitions resp. collaboration roles of the UML collab-

[14]As TLC is based on the TLA modeling language TLA$^+$, Slåtten had to modify the cTLA descriptions which, due to the foundation of cTLA (see Sect. 6.3), was straightforward.

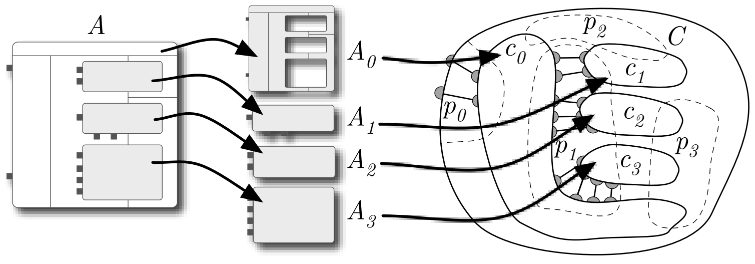


Fig. 6.15: Mapping activities to a compositional cTLA/c process C

oration. Following the collaboration role binding of the UML collaboration resp. the topology and partition mapping of the corresponding activity, the function $C_{comp}.bind$ maps each external action of $c_0, \ldots, c_n$ to a participant in $C_{comp}.Part$.

6.6.1 Synchronous Coupling

The link between an activity (like A_0) and an activity referred from it (like A_1) is described by the input and output pins of call behavior actions. An input pin models a flow of tokens from A_0 to another activity A_i while an output pin specifies an opposite flow. Every pin has exactly one incoming as well as one outgoing edge which makes the formal definition of the coupling straightforward. Due to the production rules introduced in Sect. 6.5.2, an edge heading to a pin can be modeled by an arbitrary number of cTLA actions $outp_1, \ldots, outp_k$ while an edge leaving a pin is modeled by a number of cTLA actions $inp_1, \ldots, inp_l$. These actions use the action parameter signature introduced in Sect. 6.5.2. Assuming that an input or output pin of the call behavior action hosting activity A_i is reached by an edge with the marking e_o and left by the edge e_i, we define $k \cdot l$-many system actions $act_{q,r}$ with $q \in \{1..k\}, r \in \{1..l\}$ of the corresponding cTLA process c as follows[15] ($v, w \in \{0, i\}$, $v \neq w$):

```
C.act_{q,r}(it: TT; ot: [ET → TT]; iv: VT; ov: [ET → VT];
        is, os: SUBSET TT; last: SUBSET ET) ≜
  ∃ ot_o, ov_o, is_o, os_o, last_o, ot_i, ov_i, is_i, os_i, last_i:
     C_v.outp_q(it,ot_o,iv,ov_o,is_o,os_o,last_o)
  ∧  C_w.inp_r(ot_o["e_o"],ot_i,ov_o["e_o"],ov_i,is_i,os_i,last_i)
  ∧  ot = FMERGE(ot_o,ot_i) ∧ ov = FMERGE(ov_o,ov_i)
  ∧  is = is_o ∪ is_i ∧ os = os_o ∪ os_i
  ∧  last = (last_o ∪ last_i) \ {"e_o"};
```

Given that the activities are syntactically correct and consistent with the UML collaborations they complement, these couplings produce valid cTLA/c process couplings.

[15] *FMERGE* is a function merging two functions with mutually exclusive domains to one that is defined on the union of these domains and preserves both original mappings.

- Due to the fact, that the pins are unambiguously allocated to one partition p, all output and input actions belong to p as well. Thus, we can assign the system action $c.act_{q,r}$ to p as well which will be reflected in the function p_{act} of tuple C_{comp}.
- In addition, $c.act_{q,r}$ contains a trigger if and only if $c_v.outp_q$ contains a trigger, too (i.e., $c.act_{q,r} \in C_{comp}.NT \Leftrightarrow c_v.outp_q \in c_v.NT$ holds).
- If the edge modeled by both $outp_q$ and inp_r is only attached to the pin linking them and to no other one, $act_{q,r}$ will be internal and is consequently added to $C_{comp}.Act_{int}$. Otherwise, it is an external action and added to $C_{comp}.Act_{ext}$.
- If $outp_q$ is not attached to another link, it is triggered according to the production rules. Thus, $act_{q,r}$ is triggered as well if it is an internal action. So, the synchronous coupling follows the cTLA/c constraint that internal actions must have triggers.

Besides of the process actions modeling the coupling of activities by input and output pins, the local activities A_0 to A_n may have also internal local actions and A_0 may have actions describing its links to the pins of the call behavior action in which it is defined. For each of these actions, a system action is defined in C guaranteeing that they are also executed in C. Moreover, these actions are added to $C_{comp}.Act_{int}$ or $C_{comp}.Act_{ext}$.

6.6.2 Asynchronous Coupling

The synchronous coupling of collaborations is possible whenever the actions that should be coupled are bound to the same collaboration role in the enclosing collaboration. This means that, in a component-oriented specification produced by model transformation, they can be implemented within the same state machine and hence be executed within the same state machine transition. For implementation purposes, however, we may want that also partitions bound to the same collaboration role may be realized by different state machines of the same component. A reason for that might be, for example, to let different parts of a collaboration be executed on different operating system threads to prevent long-running operations from blocking other behaviors. As passing events between different state machines is always buffered in our approach (see [KHB06]), this implies an asynchronous coupling between the processes.

In this case, we can add a stereotype to call behavior actions that should be coupled asynchronously. For the cTLA/c model we assume that activities A_0 and A_i to be linked via buffers are not coupled directly but via a collaboration B modeling the buffering of tokens. B is specified by a cTLA process c_B similar to *Transfer* from Sect. 6.5.1. In contrast to *Transfer*, however, both actions *send* and *receive* are associated to the same partition P in which the pin is located. Assuming k different output actions $outp_1, \dots, outp_k$ and l input actions $inp_1, \dots, inp_l$, the buffered coupling from activity v to activity w is specified by the $k + l$-many system actions assigned to C specified in the following (with $q \in \{1..k\}, r \in \{1..l\}$):

$$\left.\begin{array}{l} C.send_q(\ldots, ot : [ET \rightarrow TT], \ldots) \triangleq \\ \qquad c_v.outp_q(\ldots, ot, \ldots) \wedge \; c_B.send(ot(\text{“}e_o\text{”})) \\ C.receive_r(it : TT, \ldots) \triangleq \\ \qquad c_B.receive(it) \wedge c_w.inp_r(it, \ldots) \end{array}\right\} \begin{array}{l} v, w \in \{0, i\}, \\ v \neq w \end{array}$$

The l actions $C.receive_r$ have a trigger. In contrast, the actions $C.send_q$ are only trigger actions if the bound action $C_v.outp_q$ is also triggered (i.e., $C.send_q \in C_{comp}.NT \Leftrightarrow c_v.outp_q \in C_{el_v}.NT$ holds). The other settings of C_{send_q} and $C_{receive_r}$ in the cTLA/c tuple C_{comp} are similar to those of the synchronous case.

6.6.3 Asynchronous Multi-Session Coupling

To make our approach SPACE versatile for the development of real services, we must be able to deal with a number of different components providing identical functionality. For instance, the *Mobile Home Control* specification depicted in Fig. 6.7 is only useful if it specifies an arbitrary number of zone managers and telephones. To model several entities of a particular type, in the UML 2.0 collaborations the components may contain multiplicities (e.g, arbitrary many entities of the phone p and heater h as well as at least one entity of the zone manager z may occur). To achieve this multiplicity for the behaviors, cTLA/c may contain not only simple collaborations but also collaboration arrays each defining a whole number of identical simple collaborations. In cTLA/c, an collaboration array with multiplicity m corresponds to m-many simple collaboration instances each providing the same behavior. We express multiple occurrences of a collaboration by cTLA array processes as shown below for the zone session:

```
PROCESS ZoneSessionMult (m: INTEGER; ...)
ARRAY
  id: 1..m OF z: ZoneSession(...);
END
```

In general, this cTLA array operator (see also [Her97]) defines for each variable v of type $VType$ an array variable zXv of type $[1..m \rightarrow vType]$ keeping m-many values of v. Likewise, every action of z gets a new parameter $id : 1..m$ describing which occurrence of zXv is accessed. In this way, we can specify several concurrent sessions of a certain collaboration by one cTLA array process.

To determine the number of instances of a collaboration (i.e., the parameter m in the array process), we have to analyze the multiplicities of the participants bound to it. A meaningful solution is to provide a collaboration instance for each combination of participant instances. In the example, this is done for the activity *Zone Session*. If N_p is the number of phones modeled and N_z the number of zone managers, we create, as the other participating collaboration roles are not defined multiple, $N_p \cdot N_z$-many instances of the zone session collaboration.[16]

[16]The formal existence of a session instance does not necessarily imply an ongoing behavior or demand real system resources, as an execution platform may instantiate needed state machines

For the activity *Temperature Update*, however, a similar determination of the number of instances would not be useful since one heater is only connected to one zone manager. Therefore it is sufficient to provide only on activity instance for each heater in the system.

On the other side, it could be useful to create more than one instance for every combination of collaboration roles. This is not possible to express in standard collaborations, as UML does not foresee multiplicities on collaboration uses. In these cases we therefore add a stereotype as part of our profile [Kra08] to the collaboration use marking that it can be execute several times among the same participants. The multiplicity of the collaboration use is then multiplied with that of its participants, and the ID for the session takes the sequence number as an additional field.

Every activity referred by an call behavior action that represents a multi-session collaboration is formally modeled by one cTLA array process. For the activity describing the surrounding part of the call behavior actions, we face the problem that its partitions may have a different multiplicity. For instance in Fig. 6.3, we have one location server, N_z zone managers and N_h heaters. Thus, we cannot describe all partitions with a single activity process as in the singular case from Sect. 6.6.1. Instead, we define a separate activity process for every partition expressed by an cTLA/c tuple. Formally, the partitions representing a collaboration role with multiplicity larger than "1" are also specified by cTLA array processes.

The multiplicity of the collaborations has consequences for the coupling. In some cases we may still apply the couplings from the previous sections even if the originating collaboration role is multiple. This is the case if from a partition P only one activity instance A_i can be reached (e.g., it holds for the heaters, as each of them is only connected to one zone manager activity). In consequence, the decision how to traverse between P and the call behavior action hosting A_i is unambiguous and we can use the synchronous or asynchronous couplings discussed in the preceding subsections. In the activities, this situation is made visible by the lack of the shadow-like border around the call behavior action.

In contrast to that, shadow-like borders of call behavior actions highlight crossovers in which selections are necessary. This is generally only the case for flows into a call behavior action, as every outgoing flow is univocal due to the fact that an activity instance is non-ambiguously attached to an instance of all partition instances to which it is attached. For input flows, we use the special statement **select** [KBH07] already mentioned in Sect. 6.2.4. This statement is attached to each flow entering a call behavior action when there are multiple sessions to choose from. The statement simply describes a list of filters that can be applied successively to find those collaboration instances that should be notified. Filters can access data within the token or within the variables of the current partition and may also depend on the data within the individual sessions that it chooses among, which is possible as they are implemented within the same component and only requires read-access.

only when the behavior actually starts.

In cTLA/c, we model this coupling by linking the two collaborations via a special collaboration C_s implemented in cTLA as follows:

```
PROCESS SelectActivity(mc, mp: INTEGER;
                       select: [AVT × TT → SUBSET {1..mc}])
CONSTANTS PartId ≜ [{1..mc} → {1..mp}];
VARIABLES
  q : [1..mc → QUEUE(TT)];
INIT ≜ ∀ i ∈ {1..mc}: q[i] = EMPTY;
ACTIONS
  send(id: {1..mp}; iav: AVT; it: TT) ≜
    q′ = [c ∈ {1..mc} ↦
          IF (PartId[c] = id ∧ c ∈ select[iav,it])
          THEN APPEND(q[c],it) ELSE q[c]];
  receive(id: {1..mc}; ot: TT) ≜
       q[id] ≠ EMPTY ∧ ot = FIRST(q[id])
    ∧ q′ = [q EXCEPT! id ↦ TAIL(q[id])];
END
```

The parameters of the process are m_c describing the number of instances of the activity A_i bound in the call behavior action, m_p as the number of instances for the partition P, and *select* as a function describing the select statement assigned to the input pin. In particular, *select* maps settings of the auxiliary variables in P and the signature of the token traversing through the pin to the set of the instance identifiers which should get a copy of the token. *PartId* is a function mapping the identifier of A_i's instance to those of P. The process has a queue for every instance of A_i as specified by the array variable q. The action *send* models the appending of tokens to the buffers according to the select statement. Its parameter *id* refers to the identifier of the partition instance. Of course, a token may only be send to instances of the activity in the call behavior action attached to P as expressed in the condition $PartId[c] = id$. Moreover, it has to follow the select statement as specified by $c : in : select[iav, it]$. The action receive describes the consumption of an element from the queue by an instance of A_i with the identifier *id*.

In consequence, *send* is coupled with each of the k actions $outp_1, \ldots, outp_k$ of the cTLA array process c_0m generated from c_0 modeling the flow e_o towards the input buffer. Here, the parameter *id* of the action created by the cTLA array operator is mapped to *id* in *send*. In the same way, the action *receive* is joined with the l-many actions $inp_1, \ldots, inp_l$ of the process c_im specifying the downstream flow e_i of the pin (with $q \in \{1..k\}, r \in \{1..l\}$):

$$\begin{aligned}
&C.send_q(id : 1..m_p; \ldots; ot : [ET \to TT]; \ldots; ov : [ET \to VT]; \ldots) \triangleq \\
&\qquad c_om.outp_q(id, \ldots, ot, \ldots, ov, \ldots) \wedge c_s.send(id, ot[\text{“}e_o\text{”}], ov[\text{“}e_o\text{”}]) \\
&C.receive_r(id : 1..m_c; it : TT, \ldots) \triangleq \\
&\qquad c_s.receive_r(id, it) \wedge c_im.inp(id, it, \ldots)
\end{aligned}$$

For flows through an output pin, we have, as mentioned above, no freedom to select an activity. Thus, this flow is specified by a simple buffer as introduced in Sect. 6.6.1.

6.6.4 Final System Model

After composing all cTLA/c representations of the UML activities with each other, we achieve a preliminary cTLA/c system description C_{pSys}. This model consists only of internal actions act_{pSys} each having a dedicated trigger. This reflects that all call behavior actions are properly bound and each pin has exactly one upstream and downstream link. In the activities, we model the interaction of a service with its environment (e.g., the service user functionality) by means of special signals expressed by send and receive nodes [KH06]. Formally, these signals are described by the action parameters *is* and *os* which model the sets of signals coming from resp. heading towards the environment.

To achieve the final cTLA/c system model C_{Sys}, we still have to specify the handling of the local auxiliary variables defined in the activities. This is done that lately in order to enable the access of all auxiliary variables defined for a component participant P from all activity partitions, the collaboration roles of which are assigned to P. To model the auxiliary variables, we use the process *AuxVar*:

```
PROCESS AuxVar (initavt: VT;
                navt: [ SUBSET ET × [ET → VT] → VT])
VARIABLES
  store: VT;
INIT ≜ store = initavt; ACTIONS
  access (current: VT; finedge: SUBSET ET; new: [ET → VT]) ≜
     current = store ∧ store′ = navt[finedge,new];
END
```

Here, the auxiliary variables are stored in a variable *store* which initially carries the values expressed by the process parameter *initavt*. The access to the store is modeled by the action *access* which in a single step accesses the current value of the auxiliary variables (with action parameter *current*) and stores the new one reflecting the atomicity of a cTLA action. The problem of handling auxiliary variables is that flows modeled by an action may be forked and the resulting downstream flows may pass different call operation actions which can create conflicting variable assignments. To solve this problem, we defined a process parameter *navt*. It is provided by the set of final edges in a flow and the corresponding variable settings from which a unique setting is computed. This function is applied in the action *access* to calculate the new value of the variable *store*.

For every single participant, we define an instance of *AuxVar* and for every multiple partition one of the corresponding array process modeling m_p instances of the partition variables. Every action act_{pSys_k} of C_{pSys} assigned to the partition P is linked with the action $P.access$ storing the auxiliary variables for the instance id of partition P to which act_{pSys_k} is assigned:

$$
\begin{aligned}
&act_{Sys_k}(id : 1..m_p; is, os : SUBSET\ TT) \triangleq\\
&\exists : it_s, ot_s, iv_s, ov_s, last_s :\\
&\quad act_{pSys_k}(id, it_s, ot_s, iv_s, ov_s, is, os, last_s) \wedge\\
&\quad P.access(id, iv_s, last_s, ov_s)
\end{aligned}
$$

Besides the identifier *id*, the resulting system action act_{Sys_k} uses only *is* and *os* as parameters modeling that the external signals are the means to interact with the environment. The overall system specification C_{Sys} defines now the formal semantics of the full service model described by UML 2.0 collaborations and activities following the SPACE approach.

6.7 Concluding Remarks

We presented cTLA/c, a style of the compositional Temporal Logic of Actions that captures the behavior of collaborative system specifications. We think of cTLA/c foremost as a background technique to understand the formalism of collaborative specifications expressed in other languages, such as UML. In our approach, we use UML 2.0 collaborations in combination with activities, and have therefore presented how they can be transformed to cTLA/c specifications and in this way provide them with a non-ambiguous formal semantics. The provision of a formal semantics does not end in itself but, in our opinion, is a central ingredient for the automated development of high-quality software. It is the basis for meaningful semantic checks as, for instance, to be done with the model checking approach introduced in [Slå07]. With such kind of methods, one can analyze collaborative service specifications thoroughly and ensure, for instance, that different views using distinct diagrams describe one consistent execution model. Another application for the formal semantics based on cTLA/c is model transformation. It provides us with the means to verify formally that transformation tools generate target models that fulfill the behavioral constraints of the source models. As presented in [KH07b], we checked that the transformation from UML activities to state machines is correctness-preserving.

In the moment, the service specifications in form of collaborations and activities are the most abstract ones that are used in our approach. Nevertheless, there may be further layers of abstraction in specifications. These specifications could consider collaborations on higher abstraction levels, which may be useful in early specification attempts when the complete system behavior or aspects of distribution are not yet known and must successively be elaborated. For that, notations like goal sequences [CB06b, San07] or DisCo [KSM98] may be useful. As cTLA resp. cTLA/c can also be used to specify such abstract models, we can carry out formal logic proofs to guarantee the correctness of the manual or automated refinement steps from these very abstract specifications to those used in SPACE. Thus, a complete formal and highly automated development of distributed services all the way from very abstract scenario-based descriptions to executable code will be feasible. ■

Bibliography

[AL95] Martín Abadi and Leslie Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–

535, May 1995.

[BC96] Ray J. A. Buhr and Ron S. Casselman. *Use Case Maps for Object-Oriented Systems.* Prentice-Hall, Inc., 1996.

[BKS89] R. J. R. Back and Reino Kurki-Suonio. Decentralization of Process Nets with Centralized Control. *Distributed Computing*, 3:73–87, 1989.

[Bræ79] Rolv Bræk. Unified System Modelling and Implementation. In *International Switching Symposium*, pages 1180–1187, Paris, France, May 1979.

[CB06a] Humberto N. Castejón and Rolv Bræk. A Collaboration-based Approach to Service Specification and Detection of Implied Scenarios. In *SCESM '06: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pages 37–43, New York, NY, USA, 2006. ACM Press.

[CB06b] Humberto N. Castejón and Rolv Bræk. Formalizing Collaboration Goal Sequences for Service Choreography. In Elie Najm and Jean-François Pradat-Peyre, editors, *26th IFIP WG 6.1 Intl. Conf. on Formal Methods for Networked and Distributed Systems (FORTE'06)*, volume 4229 of *Lecture Notes in Computer Science.* Springer, September 2006.

[Flo95] Jacqueline Floch. Supporting Evolution and Maintenance by Using a Flexible Automatic Code Generator. In *Proceedings of ICSE-17 – 17th International Conference on Software Engineering*, Seattle, April 1995.

[Her97] Peter Herrmann. *Problemnaher korrektheitssichernder Entwurf von Hochleistungsprotokollen.* PhD thesis, Universität Dortmund, 1997.

[HK00] Peter Herrmann and Heiko Krumm. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337, 2000.

[HK07] Peter Herrmann and Frank Alexander Kraemer. Design of Trusted Systems with Reusable Collaboration Models. In Sandro Etalle and Stephen Marsh, editors, *Trust Management*, volume 238, pages 317–332. IFIP International Federation for Information Processing, Springer, 2007.

[ITU02] ITU-T. *Recommendation Z.100: Specification and Description Language (SDL)*, August 2002.

[ITU04] ITU-T. *Recommendation Z.120: Message Sequence Charts (MSC)*, 2004.

[Jen91] Kurt Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets*, pages 342–416, London, UK, 1991. Springer-Verlag.

[KBH07] Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007*, volume 4745 of *Lecture Notes in Computer Science*, pages 166–185. Springer–Verlag Berlin Heidelberg, September 2007.

[KH06] Frank Alexander Kraemer and Peter Herrmann. Service Specification by Composition of Collaborations — An Example. In *Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology)*, pages 129–133. IEEE Computer Society, 2006. 2nd International Workshop on Service Composition (Sercomp), Hong Kong.

[KH07a] Frank Alexander Kraemer and Peter Herrmann. Semantics of UML 2.0 Activities and Collaborations in cTLA. Avantel Technical Report 4/2007 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway, September 2007.

[KH07b] Frank Alexander Kraemer and Peter Herrmann. Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In Karsten Ehring and Holger Giese, editors, *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, volume 7 of *Electronic Communications of the EASST*. EASST, 2007.

[KHB06] Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk. Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. In R. Meersmann and Z. Tari, editors, *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), 2006, Montpellier, France*, volume 4276 of *Lecture Notes in Computer Science*, pages 1613–1632. Springer–Verlag Heidelberg, 2006.

[Kra] Frank Alexander Kraemer. The Ramses and Arctis Tools. http://www.item.ntnu.no/~kraemer/tools.

[Kra08] Frank Alexander Kraemer. UML Profile and Semantics for Service Specifications. Avantel Technical Report 1/2007 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway, June 2008.

[KS05] Reino Kurki-Suonio. *A Practical Theory of Reactive Systems*. Springer, 2005.

[KSM98] Reino Kurki-Suonio and Tommi Mikkonen. Abstractions of Distributed Cooperation, their Refinement and Implementation. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 94–102. IEEE Computer Society, April 1998.

[Lam02] Leslie Lamport. *Specifying Systems*. Addison-Wesley, 2002.

[Obj07] Object Management Group. Unified Modeling Language: Superstructure, version 2.1.1, February 2007. formal/2007-02-03.

[Pnu86] Amir Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. *Current Trends in Concurrency. Overviews and Tutorials*, pages 510–584, 1986.

[RB06] Judith E. Y. Rossebø and Rolv Bræk. Towards a Framework of Authentication and Authorization Patterns for Ensuring Availability in Service Composition. In *Proceedings of the 1st International Conference on Availability, Reliability and Security (ARES'06)*, pages 206–215. IEEE Computer Society Press, 2006.

[San07] Richard Sanders. *Collaborations, Semantic Interfaces and Service Goals: A Way Forward for Service Engineering.* PhD thesis, Norwegian University of Science and Technology, 2007.

[SCKB05] Richard Sanders, Humberto N. Castejón, Frank Alexander Kraemer, and Rolv Bræk. Using UML 2.0 Collaborations for Compositional Service Specification. In *ACM / IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, 2005.

[Slå07] Vidar Slåtten. Model Checking Collaborative Service Specifications in TLA with TLC. Project Thesis, August 2007. Norwegian University of Science and Technology, Trondheim, Norway.

[Vef85] Eirik A. M. Vefsnmo. DASOM — A Software Engineering Tool for Communication Applications Increasing Productivity and Software Quality. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 26–33, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[VSvSB91] Chris A. Vissers, Guiseppe Scollo, Marten van Sinderen, and Hendrik Brinksma. Specification Styles in Distributed System Design and Verification. *Theoretical Computer Science*, 89:179–206, 1991.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA$^+$ Specifications. In L. Pierre and T. Kropf, editors, *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working*

Conference on Correct Hardware Design and Verification Methods (CHARME'99), volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer-Verlag, 1999.

PAPER

SEVEN

ENGINEERING SUPPORT FOR UML ACTIVITIES BY AUTOMATED MODEL-CHECKING — AN EXAMPLE

Frank Alexander Kraemer, Vidar Slåtten and Peter Herrmann.

Presented at the 4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2007) November 26 – 27, 2007 Luxembourg.

Published in: Nicolas Guelfi, ed., Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2007), p. 51-66, University of Luxembourg, 2007.

Engineering Support for UML Activities by Automated Model-Checking — An Example

Frank Alexander Kraemer, Vidar Slåtten, and Peter Herrmann

Abstract. In our approach for the engineering of reactive services, we specify systems as collaborations by means of UML 2.0 activities. In automated and correctness-preserving steps, these models are transformed into executable code. As the semantics of the models are defined using temporal logic, we can utilize model checking to prove that the collaborations fulfill certain general well-formedness properties. This is quite important since communication delays in the interactions between the participants realizing a collaboration aggravate the design of correct collaborative behavior. The well-known state space explosion problem of model checkers is mitigated by using special external state machines which define the interface behavior of sub-activities. The generation of the formal input for the model checker TLC from the activities is completely automated, so that the engineers working on the activities do not need to be experts in temporal logic and model checking. In this paper, we describe the utilization of TLC to detect and correct design errors by means of an example.

7.1 Introduction

In our engineering approach for reactive services SPACE [KBH07, KH06, KH07b, KHB06, HK07], system specifications are composed of building blocks that model functionality related to a certain task. The building blocks are collaborations covering several components. In addition to the necessary interactions, they also define the local behavior of all the participating components. We use UML 2.0 activities to describe the behavior of collaborations. Activities can be divided into several partitions, each identifying the tasks of the individual participating components. Control flows are represented explicitly and may be synchronized by a number of control nodes. Moreover, activities can be decomposed into sub-activities, so that systems may be built from already existing building blocks.

Enabling collaborations as the structuring units of service specifications is beneficial in various respects. First, services usually involve several participating components. Describing them by collaborations gives a holistic view of the service which can be understood without combining all the component descriptions. Second, the degree of reuse is potentially higher since a collaboration solves only a certain subtask and is therefore more likely to be useful in other applications than entire components that typically combine several tasks making them very specific (see for example [HK07]).

Figure 7.1 outlines the development process along with the tools supporting it. An engineer works on collaborative service specifications, using a library of reusable building blocks providing solutions to reoccurring problems. The building blocks can be composed together with additional “glue” logic using an editor for activities. For the execution of the services, however, descriptions

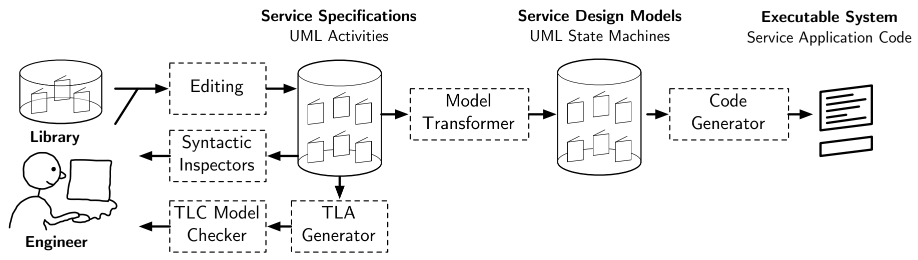


Fig. 7.1: Tool Support for the SPACE engineering approach

of the system components are needed. We hereby follow a specification-driven approach, in which the service specifications composed of the collaborations are automatically transformed to component-oriented service design models in the form of UML 2.0 state machines, as described in [KH07b]. This has the benefit that consistency between the different development stages is ensured, and engineers just have to maintain the service specifications. The state machines are then the input for our code generators that produce executable code for various platforms (see [KHB06]).

For such an approach and its tools to be correct, formal reasoning is needed to guarantee that properties described by the individual collaborative building blocks are preserved by the composed system. Furthermore, we must ensure that these properties are also maintained by the model transformation to state machines and the implementation on the various execution platforms. For this, we use the compositional Temporal Logic of Actions (cTLA, [HK00]). We formalized both the service specifications in terms of activities [KH07a] as well as the state machines [KHB06]. The coupling principle of cTLA supports the property of superposition [BKS89], in which properties of a part of the system (i.e., the individual building blocks) are also valid for the composed system. This makes it possible to map the composition of activities and state machines directly to the cTLA couplings. The model transformation and code generation correspond to refinement steps. Thus, we can use cTLA refinement proofs to verify that these steps are correct (see [KH07b, KHB06]).

This approach is already beneficial for specification quality, since the abstraction level of the models is higher which allows for a better understanding of the behavior. Coding errors are avoided due to the automatic translation. Of course, the abstract specification models need to be correct in the first place. While some properties may be ensured by a purely syntactic analysis, others require us to consider entire behaviors, for example, that interface events of building blocks have to occur in a certain order. This is usually hard to guarantee manually as behavior involving several components can get quite complex due to the unavoidable delays of the communication medium connecting them. To assure correctness of such behaviors, model checking (i.e., the examination of all reachable states a behavioral description implies) can be used. This technique, however, presupposes a certain amount of expertise in formal reasoning, which we do not want to claim from the engineers using our approach. A possibility to overcome this situation is, as Rushby suggests in "Disappearing Formal Methods" [Rus00], to wrap formal techniques within tools so that they are not perceived as difficult anymore, and to increase their user-friendliness. The idea behind this is that a user does not necessarily need to understand the details of a formal technique and model-checking, if an automated checking tool gives understandable feedback addressing the problem in the language of the engineer's domain.

In this paper, we focus on such an approach. Utilizing the formalization of the activities in cTLA, we developed in [Slå07] an automatic transformation tool from UML 2.0 activities to TLA$^+$, the input language of the model checker TLC [YML99]. TLC can check a specification for various temporal properties that are stated in form of theorems. For each activity, we automatically generate a set of theorems which claim certain properties to be kept by activities in general. Examples of these properties are the correct usage of building blocks within the activity, that the activity itself satisfies a certain externally visible behavior, and that queues used for transmissions are bounded. When TLC detects that a theorem is violated, it produces an error trace displaying the state sequence that leads to the violation. This trace can be given in terms of easily comprehensible token markings within an activity as well. So, an engineer using our tools does not have to write or understand the temporal logic formulas.

The presented approach for model checking makes use of the compositional nature of our service specifications. As described in [KH07a], a system composed of collaborations guarantees the properties of each single collaboration to be maintained. This follows directly from the semantics based on cTLA [HK00] and the principle of superposition. The activities describing the complete behavior of collaborations may be specified in a more abstract form by means of special state machines that refer to externally visible events dedicated for composition. When model checking a composite specification, only these abstract specifications have to be taken into account, which reduces the state space. Thus, we check each collaboration separately and do not consider the entire hierarchy which effectively mitigates the likelihood of state space explosions.

After discussing some related work done on formal checking of UML models, we give an introduction to temporal logic as well as the model checker TLC in Sect. 3. We proceed by introducing an example specification based on activities, and explain the semantics of activities in temporal logic in Sect. 4. Thereafter, we use our tools in Sect. 5 to develop one building block of the example, starting with a naive solution which we incrementally correct based on the feedback of the model checking. We close in Sect. 6 with some concluding remarks.

7.2 Related Work

Formal checks on UML models are done as part of OMEGA [Hoo02], FUJABA [BGHS04] and HUGO [BBK$^+$04]. However, these approaches mainly concentrate on state machines or sequence diagrams, not on activities as in our case. In [GM05], UML activities are translated into PROMELA, the input language for the SPIN model checker [Hol03]. In [Stö05], a mapping from UML 2.0 activities to Colored Petri Nets is described enabling the usage of Petri net tools for analysis. In [DS03], UML activities are transformed into the π-calculus where safety and liveness properties can be expressed using the modal mu-calculus and checked using the MWB tool [VM94]. Eshuis [Esh06] uses NuSMV, a symbolic model verifier to check the consistency of activity diagrams. The difference of these approaches to ours mainly lies in the domain that activities are used for and the chosen semantics. While they focus on activities more from a perspective of business processes assuming a central clock or synchronous communication, we need for our activities reactive semantics [KH07a] reflecting the transmission of asynchronous messages between distributed components. This semantics enables us to generate the executable state machines defined in [KHB06].

7.3 Temporal Logic

The model checker TLC is based on Leslie Lamport's Temporal Logic of Actions (TLA, [Lam94, Lam02]), which is also the basis of cTLA. TLA is a linear-time temporal logic in which behavior is expressed by infinite state sequences. The corresponding syntax is TLA$^+$ that enables describing system behavior by special state transition systems and additional fairness properties. Fig. 7.2 is an example of a TLA$^+$ specification. After a frame containing the module name (i.e., *HotelWakeUpSystem*), it uses the expression EXTENDS *Naturals* describing the import of a module including definitions, operators and axioms to model the natural numbers. The states of the state transition system are modeled by variables (here i, t, h and a). The predicate *Init* specifies the set of values the variables shall have in the initial state. The transitions are described by actions each specifying a pair of a current state and its successor state. Here, the current state is referred to by variable identifiers in a simple form while the next state is modeled by primed variable identifiers. An example is the action *initial* which may be executed if the variable i has the value 1 and h has the value "off". After its execution, i will carry the value 0 which is described by $i' = 0$. In addition, h will have the value "started" in the following state while the two other variables a and t do not change their values during the execution of the action. The set of system transitions is modeled as the disjunction of the system actions which is expressed by the definition *Next*, the so-called next-state relation. The overall system description is modeled by the canonical formula *Spec*. The first conjunct of this temporal formula defines that the predicate *Init* holds in the first state of every state sequence modeled by *Spec*. The second conjunct uses the temporal operator $\Box$ ("always") specifying that the rest of the conjunct is valid in all states of all state sequences describing the behavior of the system. The TLA expression $[Next]_{\langle i,t,h,a \rangle}$ determines that a state transition has to be either a stuttering step in which all variables listed in the subscript maintain their values or satisfies the condition *Next*. Thus, every state sequence begins with a state fulfilling *Init* and corresponds only to state transitions which either meet one of the system actions or are stuttering steps. Further conjuncts may

MODULE *HotelWakeupSystem*

EXTENDS *Naturals*

VARIABLES $i,\ t,\ h,\ a$

$Init \triangleq$
$\quad \land\ i = 1 \land t = 0$
$\quad \land\ h = \text{“off”} \land a = \text{“off”}$

$initial \triangleq$
$\quad \land\ i = 1 \land i' = 0$
$\quad \land\ h = \text{“off”} \land h' = \text{“started”}$
$\quad \land\ \text{UNCHANGED}\ \langle a,\ t\rangle$

$startAlert \triangleq$
$\quad \land\ h = \text{“started”} \land h' = \text{“alerting”}$
$\quad \land\ a = \text{“off”} \land a' = \text{“active”}$
$\quad \land\ \text{UNCHANGED}\ \langle i,\ t\rangle$

$stopAlert \triangleq$
$\quad \land\ h = \text{“alerting”} \land h' = \text{“stopped”}$
$\quad \land\ a = \text{“active”} \land a' = \text{“off”}$
$\quad \land\ \text{UNCHANGED}\ \langle i,\ t\rangle$

$aborted \triangleq$
$\quad \land\ h = \text{“stopped”} \land h' = \text{“off”}$
$\quad \land\ t = 0 \land t' = 1$
$\quad \land\ \text{UNCHANGED}\ \langle i,\ a\rangle$

$confirmed \triangleq$
$\quad \land\ h = \text{“stopped”} \land h' = \text{“off”}$
$\quad \land\ t = 0 \land t' = 1$
$\quad \land\ \text{UNCHANGED}\ \langle i,\ a\rangle$

$timeout \triangleq$
$\quad \land\ t = 1 \land t' = 0$
$\quad \land\ h = \text{“off”} \land h' = \text{“started”}$
$\quad \land\ \text{UNCHANGED}\ \langle i,\ a\rangle$

$Next \triangleq$
$\quad \lor\ initial \lor startAlert \lor stopAlert$
$\quad \lor\ aborted \lor confirmed \lor timeout$

$Spec \triangleq Init \land \Box[Next]_{\langle i,\ t,\ h,\ a\rangle}$

$t0 \triangleq \Box((i = 1) \Rightarrow (h = \text{“off”}))$
$t1 \triangleq \Box((h = \text{“stopped”}) \Rightarrow (t = 0))$
$t2 \triangleq \Box((h = \text{“started”}) \Rightarrow (a = \text{“off”}))$
$t3 \triangleq \Box((h = \text{“alerting”}) \Rightarrow (a = \text{“active”}))$
$t4 \triangleq \Box((t = 1) \Rightarrow (h = \text{“off”}))$

Fig. 7.2: TLA Module

be used to describe liveness properties by fairness assumptions on actions which, however, is not discussed in this paper.

The second paragraph of the specification contains a list of properties $t0$ to $t4$ which shall be kept by the system. As they all start with the always operator, they state invariant behavior (e.g., if variable i has value 1, h must be “off”). To verify an invariant, one has to prove that it holds in the initial condition *Init* and that it is preserved by every system action.

The compositional Temporal Logic of Actions (cTLA [HK00]) mentioned in the introduction is a derivative of TLA. It resolves a shortcoming of TLA which is limited to compositions based on joined variables [AL95]. In contrast, cTLA combines modules by defining joined system actions as simultaneously executed module actions which is a prerequisite for constraint-oriented models [VSvSB91]. There, one specifies not single physical components but properties describing partial system behavior which spans several components. As the UML 2.0 collaboration and activity-based models used in our approach demand this particular specification style, we used cTLA instead of TLA to define their semantics [KH07a]. cTLA uses a process-like specification style which encompasses both simple and compositional process descriptions. As the compositional process models can be transferred to simple ones (see [HK00]) and the simple processes are basically defined by the same canonical formulas as TLA$^+$, it is quite straightforward to transform the UML activities to TLA$^+$ modules like

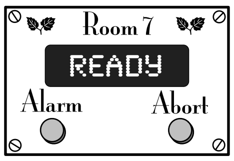


Fig. 7.3: Reception Panel

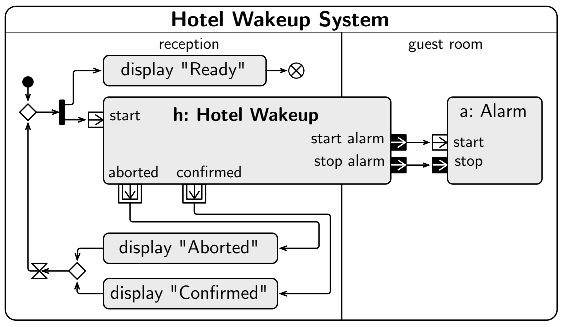


Fig. 7.4: Activity for the entire system

the one depicted in Fig. 7.2. This is done by the tool introduced in [Slå07] such that we can use the model checker TLC [YML99] to automatically prove that the activities fulfill certain properties since TLC uses TLA$^+$ specifications as input. TLC performs an exhaustive exploration of all reachable system states and verifies that invariant properties are maintained by every checked state.[1] In the case of a failure, a path of states leading to the one not fulfilling a property is shown which facilitates the search for the error and can be visualized in the UML activities.

7.4 UML 2.0 Activities in the SPACE Approach

In order to study an intricate The system is partly automated, as the requests for wake-up alarms are noted manually by the receptionist in a book. The guests prefer to be woken by an alarm instead of a direct phone call, to avoid contact with the personnel at an early morning hour. To convince the receptionist that they really are awake, they confirm the alarm by pressing a button. The reception has a control panel with two buttons and a display for each of the guest rooms, illustrated in Fig. 7.3. At wake-up time, the receptionist pushes the alert button which sounds the alarm in the guest room. If the guest confirms, the display shows *Confirmed* for some seconds so that the receptionist knows that the guest is actually awake. If the guest does not confirm, the receptionist can abort the alert after some time, upon which he or she may visit the room and rouse the guest with more drastic measures.

7.4.1 Informal Explanation of Activities

The behavior of the example system is described by the UML 2.0 activity shown in Fig. 7.4. It is divided into two activity partitions, one denoting the hotel

[1]For liveness proofs not introduced here, TLC checks sequences of states.

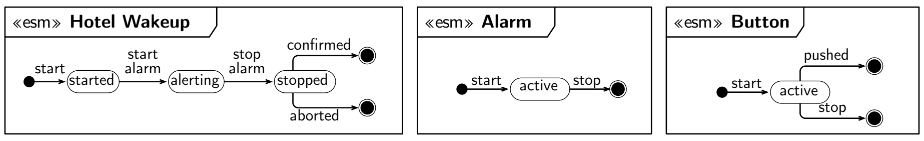


Fig. 7.5: External state machines

reception and one for a guest room.[2] On the reception side, the activity contains three operations to control the display by printing the messages *Ready*, *Aborted* and *Confirmed*. On the side of the guest room, an alarm device is represented by a so-called *call behavior action*. This is a node that may refer to other activities (in the following referred to as *sub-activities*) and be used for decomposition. In the system here, we do not know about the internals of the alarm, just that it can be started by a token entering via *start* and stopped by a token via *stop*. Similarly, *h* refers to another activity realizing the protocol between the reception and the hotel room.[3] In contrast to the building block for the alarm, *h* spans over both activity partitions and as such describes a collaboration between the reception and the guest room.

The system activity starts on the side of the reception at the initial node. A token is emitted upon system startup and moved to a fork node, where it is duplicated. One of the tokens continues to operation *display Ready*, causing the display to show that the system is ready. Afterwards, it ends at a flow final node. The other token leaves the fork and moves into the call behavior action *h* via input pin *start*. This activates the *Hotel Wakeup* sub-activity. On this level, we just need to know about its externally visible behavior, described by the state machine *Hotel Wakeup* in Fig. 7.5. The stereotype ≪esm≫ applied to it marks that the diagram denotes an external state machine (ESM, [Kra08]) for the sub-activity. Its transitions refer to the input and output pins of the corresponding sub-activity, describing in which sequence tokens may be passed. We see that after *start*, event *start alarm* will eventually happen, followed by *stop alarm*. Thereafter, the sub-activity terminates as either *aborted* or *confirmed*, depending on the behavior of the guest. On the side of the guest room, the flow leaving *start alarm* and *stop alarm* of *h* is connected to *start* resp. *stop* of the call behavior action *a* modeling the alarm. On the reception side, the display informs the receptionist about the outcome via two distinct display messages once sub-activity *h* terminates. As soon as the display messages *Confirmed* or *Aborted* appear, a timer is started waiting for a certain time, so that the message can be read. Upon a timeout, the display is reset to *Ready* and the hotel wake-up can be used again.

A first (naive) solution for the internals of the call behavior action *h: Hotel Wakeup* is shown in Fig. 7.6. Note that the dashed lines are not a concept of UML activities but are here used to illustrate the preliminary state of the

[2]To keep the discussion simple, we only consider one room. Using the mechanisms presented in [KBH07], this design can easily be expanded to multiple rooms.

[3]The decision to put the alarm and the display outside of the *Hotel Wakeup h* was here mainly to ease the presentation of the contents of *h* as shown in Sect. 5.

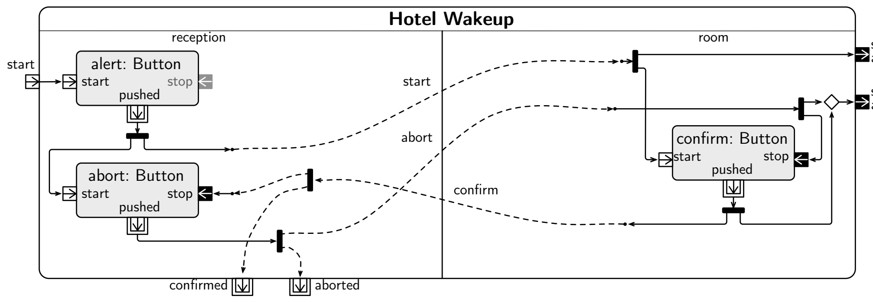


Fig. 7.6: Solution 1

model which we will replace later, based on the findings of the model checking. The flows in solid lines remain stable throughout all solutions. The activity is composed from three buttons *alert*, *confirm* and *abort* from our library of reusable building blocks [Kra07]. Their external behavior is described by ≪esm≫ *Button* in Fig. 7.5. There, a button is activated via *start.* In this state, it may be pushed by the user, which causes its termination via *pushed.* It may also be stopped by a token through *stop*, whereupon any pushes by the user are ignored.

When the *Hotel Wakeup* collaboration is started, the alert button is activated immediately. Once it is pressed, a token is emitted via *pushed*, activating the abort button. At the same time, the flow continues towards the partition for the guest room. As the partitions will be implemented by different, physically remote components, we assume a buffered communication between activity partitions. Therefore, a token waits for an arbitrary time in a virtual queue place where a flow crosses partition borders. This corresponds to the transmission through a physical medium. When the flow from the alert button is received by the guest room partition, the confirm button is activated, and a token is branched off towards the output node *start alarm* to notify the alarm device. If the confirm button is pushed, the alarm is stopped via output node *stop alarm* and a confirmation is routed back to the reception partition where the collaboration terminates via output pin *confirmed.* If the receptionist presses the abort button, the guest room is notified to switch off the alarm and the confirmation button, and the collaboration is terminated via *aborted.*

7.4.2 Semantics of UML 2.0 Activities in Temporal Logic

Formally, UML activities are based on Petri Nets and describe as such a state transition system. In [KH07a] we defined the semantics of activities in terms of cTLA, which can be easily mapped to TLA$^+$, the input language for the model checker TLC, as discussed in Sect. 7.3. The transformer from UML 2.0 activities to TLA$^+$ [Slå07] uses UML activity models stored in the UML2 repository of Eclipse as input. Roughly speaking, the tool maps each token movement of an activity to an action in a TLA$^+$ formula in which stateful nodes such as timers, sub-activities, joins and accept signal actions are represented by their own variables. The buffering of flows that cross activity partitions is formalized by queue variables which are bags of tokens. Whenever a token leaves a source partition, it is added to the corresponding queue place. In a second action, it is removed from the queue place and continues the flow in the target partition.

As an example, the specification in Fig. 7.2 displays the TLA$^+$ code generated for the system activity depicted in Fig. 7.4. It consists of six actions,[4] each modeling a token movement. The module declares a variable for each stateful node of the activity, that is, the initial node by variable i, the timer by t as well as the sub-activities for the wake-up h and the alert a. For both the timer and the initial node, we simply use an integer to store the number of tokens that are resting in them. Initially, there is one token in the initial node (which means the activity is ready to start) and no token in the timer (i.e., the timer is idle). This is expressed with the initial predicate $Init$ by $i = 1 \wedge t = 0$. The variables for the sub-activities store the current states of the ESMs that represent their externally visible behavior. Initially, both ESMs are in their initial state, so that value "off" is assigned to h and a by $Init$. The six actions model the token movements within the activity. Action *initial* specifies the start of the activity. The token resting in the initial node is removed from it ($i' = 0$) and enters h via input pin *start*. The ESM of h (according to its definition in Fig. 7.5) makes a transition to state *started*.[5] When h is in state "started", action $startAlert$ is enabled. It models the emission of a token from h via *startAlert* activating the alarm ($a' =$"active"). Eventually, the alarm will be deactivated again by the execution of *stopAlert*. After that, the two actions *aborted* and *confirmed* are enabled, modeling the termination of sub-activity h (by $h' =$"off"). Due to the merge node, both of these actions start timer t (by $t' = 1$), enabling action *timeout*, which restarts sub-activity h.

7.4.3 Theorems for Well-Formed Activities

An important property of our activity specifications is that the events of the sub-activites are invoked in the order specified by their ESMs. This means for example that whenever a token attempts to enter *start* of sub-activity h, then h must not yet be activated, i.e., $h =$"off". A token can be released from the initial node whenever it has a token, i.e., $i = 1$. So, we want to be sure that whenever there is a token in the initial node, the sub-activity is not yet active. Formally, this is an implication $(i = 1) \implies (h = \text{"off"})$. As this property must always hold, our tool writes the theorem as an invariant $t0 \triangleq \Box((i = 1) \implies (h = \text{"off"}))$. The other theorems describe the other cases in which the ESM of a sub-activity must not be violated by its environment. For example, $t4$ ensures that whenever the timer is active ($t = 1$), sub-activity h may be started again ($h =$ "off"). The violation of ESMs is only one source of errors. The current transformation tool also writes theorems to check the boundedness

[4] We adjusted the automatically chosen variable and action names for readability.

[5] The token is further forked into operation *display Ready*, which we can ignore here since no stateful node is reached.

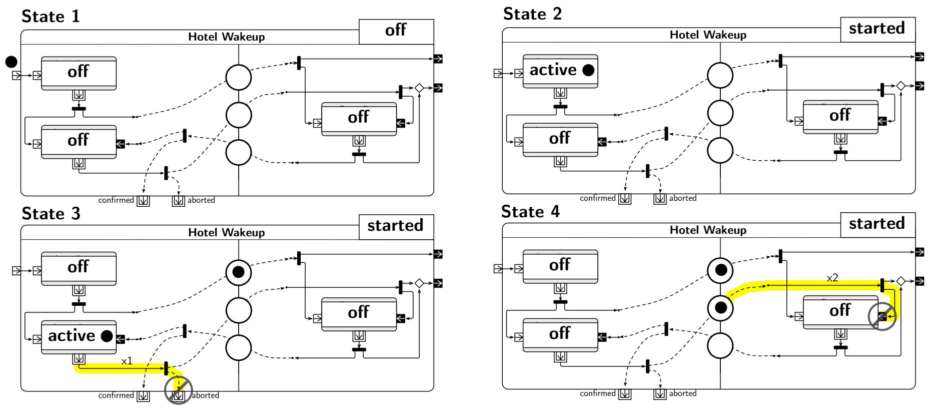


Fig. 7.7: Error trace of solution 1

of queues as well as assertions on the execution of operations that can be added with additional stereotypes [Slå07]. This is, however, not discussed here.

7.5 Developing and Model Checking the Example

The use of model checking to correct activity-based service specifications is outlined by discussing the improvements of the hotel wakeup system. We start by applying our transformation tool and create the TLA$^+$ specification of the system activity listed in Fig. 7.4. The outcome is the TLA module introduced in Fig. 7.2 which is checked by TLC. The model checker notifies that 5 distinct states were generated and that no errors were found. Given the theorems that are included in our automatically generated formal specification, this means that the contracts of the used building blocks h and a are obeyed. Thus, we can proceed by checking the design of the *Hotel Wakeup* activity.

7.5.1 Solution 1: A Naive Start

As an initial solution, we consider the activity introduced in Fig. 7.6. On a first glance, it looks quite reasonable. When the alarm button is pushed, the guest room is notified to activate the confirmation button. A push on their button by either the receptionist or the guest stops the alarm and the respective other button. However, when we model check this activity, TLC says that temporal properties are violated and prints a trace of states that describes the behavior up to the moment when the violation took place. This trace may be projected onto the activity, as illustrated in Fig. 7.7. Hereby, the transfer queues are shown as token places where the flows cross partitions, and the activity and its sub-activities are amended with boxes showing the current state of their ESM.

State 1. The activity is not yet active and its ESM is in state *off*. The queues *a*, *b* and *c* are empty, and all sub-activities are in state *off* as well.

State 2. A token was moved via the input node of the activity and activated the alarm button, which is now in state *active*.

State 3. After the alarm button was pressed, a token was forwarded into queue *a* and the abort button is now active. In this state, TLC reports that a theorem is violated. This theorem states that whenever the abort button is active (and may therefore emit a token at any time), the ESM of Hotel Wakeup is in state *stopped*, as an outgoing token from *abort* would pass through parameter node *aborted* (see Fig. 7.5). So, in the current state, the active abort button could terminate the entire activity through flow *x1* and contradict the ESM. In practice this means that the system using *Hotel Wakeup* could assume the alarm to be aborted after the abort button was pressed, although the alarm was never started. To check for further errors before a redesign, the tool allows us to ignore this error for a moment and let the abort button be pushed.

State 4. By pushing the abort button, a token was emitted via *aborted* and another one is placed in queue *b*. In this state, the guest room may decide to consume the token from queue *b*, which would then be moved via *x2* into the confirm button that is in state *off*, which is against the ESM of the button (see Fig. 7.5). Obviously, the activity in Fig. 7.6 does not regard that due to the transfer medium, an abort flow may overtake the alarm flow.

7.5.2 Solution 2: Improved Version with a Sequencer

The problem found in state 4 of solution 1, where the confirm button could be stopped before it was even started, can be solved by adding a building block of type *Sequencer* from our library [Kra07] to the new activity in Fig. 7.8. It controls two flows arriving in any order at *i1* and *i2* such that their respective outputs may only happen in the order *o1* followed by *o2*. The problem found in state 3 of the previous solution, according to which the ESM of Hotel Wakeup was violated, can be solved by an additional flow *f* that returns from the hotel room after the alarm was started. A new run of TLC on the activity in Fig. 7.8 reveals, however, that there are still flaws in the system. Figure 7.9 shows the new error trace. The two first states are omitted as they correspond to the ones of Fig. 7.7.

State 3. The alert button has been pressed and a token is waiting to cross from partition *reception* to partition *room* in queue *a*. The abort button has also received a token and is in state *active*.

State 4. The token waiting in queue *a* has passed through the sequencer and activated the confirm button. The token was also forked so that a copy left the activity via *start alarm* causing the ESM of Hotel Wakeup (Fig. 7.5) to change from *started* to *alerting*. Both buttons are now waiting to be pushed.

State 5. The confirm button has been pushed sending a token via *stop alarm* changing the state of the ESM to *stopped*. The token was also forked into the queue *c* where it is waiting to enter the reception partition. The confirm button has returned to state *off*.

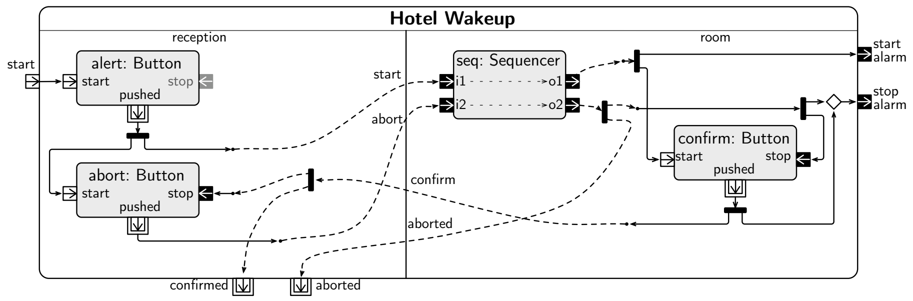


Fig. 7.8: Solution 2

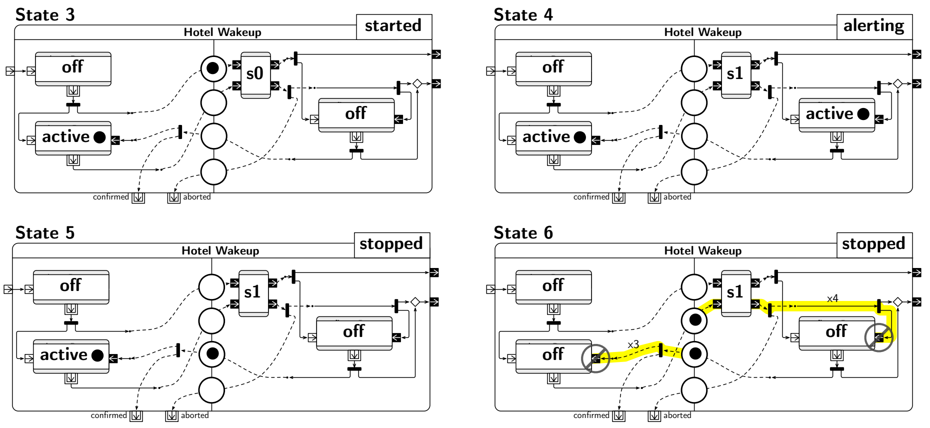


Fig. 7.9: Error trace of solution 2

State 6. The receptionist pushed the abort button, which switched to *off* and emitted a token into queue *b*, so that there is now one token in each of the queues *b* and *c*. This harms, however, two theorems that protect the contracts of the buttons. The confirm and stop button are both in state *off*, but tokens are placed in the queues that flow into the stop pin of the buttons via flows *x3* and *x4*, which would violate their ESMs.

7.5.3 Solution 3: A Building Block for Mixed Initiatives

State 6 of the trace in Fig. 7.9 reveals an intrinsic peculiarity of the system: Due to the communication delay between the reception and the hotel room, both, an abort and a confirmation, can be in progress simultaneously. This is since during the alerting phase, both the receptionist and the hotel guest may take their initiative at nearly the same time. Although not always recognized, this situation occurs frequently in reactive systems, and has several names such as *conflicting* [BH93] or *mixed initiative* [Flo03] as well as *non-local choice* [BAL97]. As the problem is quite general, our library of building blocks contains a collaboration

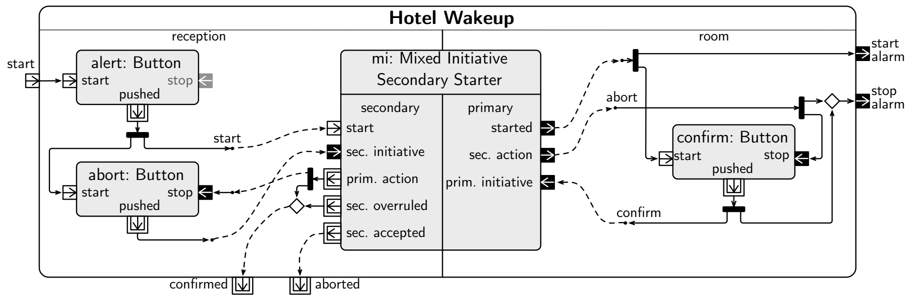


Fig. 7.10: Correct solution with a building block to handle mixed initiatives

to handle mixed initiatives [Kra07]. This collaboration has two participants, a *primary* and a *secondary* one. These names reflect which of the sides gets priority over the other if both sides take initiative. Two variants of the building block exist, one where the primary participant starts the collaboration, and one where the secondary one starts. In our system, we use the latter one and assign the primary role to the guest room, so that a confirmation from a hotel guest has priority over the abort from the reception. Fig. 7.10 shows the building block already embedded into the new solution while the ESM showing the detailed interleaving of its events is given in Fig. 7.11. For the sake of brevity, we look here just at the externals of the block, as an engineer would do when reusing it. The internals are similar to the building block *Tour Request* introduced in [KBH07].

After the start of the collaboration via *start* on the secondary side, *started* notifies the primary side that the state is reached in which it may trigger an initiative. We couple this action with the start of the alarm. Input pin *prim. initiative*, denoting an initiative taken by the primary participant, is coupled with the pushing of the confirmation button. As the primary side has priority, we know that the confirmation will succeed, and can therefore stop the alarm right away. If the secondary side takes initiative (input pin *sec. initiative*), the primary side gets notified via *sec. action*, which is used to stop both the alarm as well as the confirmation button.

On the secondary side we have to take into account that an initiative from the abort button can be overruled by the confirmation of the guest room. Besides the nodes to start the collaboration and to take initiative, the secondary side has therefore three terminating output pins, from which only one will eventually release a token.

– Pin *primary action* releases a token if the primary side took initiative, and the secondary remained passive, i.e., only the guest confirmed. This leads to stop the abort button and to terminate via *confirmed.*

– Pin *sec. overruled* models that both initiatives have been taken, from which only the primary prevails. It is sensible to distinguish this case from the first one, as the reception in this case does not have to switch off the abort button, which already terminated because of its initiative.

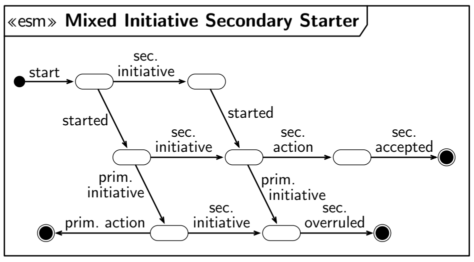


Fig. 7.11: ESM for Mixed Initiative Secondary Starter

– Pin *sec. accepted* emits a token if the secondary initiative was the only one, and the primary side did not start an initiative on its own, i.e., the alarm was aborted without the confirmation button being pressed.

When we translate this activity into TLA$^+$ and start TLC, we get the message that all properties are fulfilled now. Thus, the activity handles all the incorporated building blocks as prescribed by their respective ESMs. Moreover, it respects its own ESM and can be correctly used within the system described in Fig. 7.4. After checking the activity realizing the call behavior action a modeling alarms we know that the overall service specification is well-formed and can use it as input for the transformation steps producing executable code.

7.6 Concluding Remarks

We presented our service development approach SPACE that uses collaborations as building blocks. Their behavior is described by UML 2.0 activities which we can transform automatically into temporal formulas and a number of theorems expressing relevant properties to be fulfilled by an activity. The correctness of these theorems is model checked by TLC and its error messages lead to stepwise improvements of the models. The approach works both bottom/up and top/down. Sub-services may be arranged and their composition to larger services may be checked. Vice-versa, as done for the hotel wakeup, we may first assume a certain external behavior and then realize the internals of the service. Of course, many real systems are more extensive than the example used for the discussion here. The larger scale of these system results, however, mostly in a higher number of collaborations to be executed than in more complicated interactions. Thus, we will have a higher number of decomposition levels (see, for instance [HK07]), while the complexity of the models describing individual collaborations will remain of manageable size.

Once a collaboration between components in form of activities is model checked, it can be used in other systems without further proof efforts. This is feasible as the building blocks may be abstracted by their ESMs describing their external behavior. Thus, if we check an activity containing a sub-activity, we only have to consider the ESM of the sub-activity which hides the internal states, such that the state space of the model checked activity is reduced. In consequence, model checking is never done on the entire system with all its details, but it is enough to successively check activities on their decomposition level separately. In this way, services and their compositions from sub-services may be verified in a compositional way which effectively rules out state explosions.

With the automatic formulation of the temporal formulas and theorems we created the base for user-friendly model checking of the service specifications based on UML activities. In future versions, we may offer more advanced feedback to the user that may explain error situations further and suggest typical improvements. This work will be performed as part of the research and development project *Infrastructure for Integrated Services* ISIS, funded by the Research Council of Norway, where we develop methods, tools and building blocks for services in the domain of home automation. ■

Bibliography

[AL95] Martín Abadi and Leslie Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–535, May 1995.

[BAL97] Hanene Ben-Abdallah and Stefan Leue. Syntactic Detection of Process Divergence and Non-Local Choice in Message Sequence Charts. In *Proc. of the 2nd Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, 1997.

[BBK$^+$04] Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums. Interactive Verification of UML State Machines. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *Proceedings of the International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 434–448. Springer, 2004.

[BGHS04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In *Proc. of the International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004*, pages 1–20, October 2004.

[BH93] Rolv Bræk and Øystein Haugen. *Engineering Real Time Systems: An Object-Oriented Methodology Using SDL.* The BCS Practitioner Series. Prentice Hall, 1993.

[BKS89] R. J. R. Back and Reino Kurki-Suonio. Decentralization of Process Nets with Centralized Control. *Distributed Computing*, 3:73–87, 1989.

[DS03] Yang Dong and Zhang Shensheng. Using π-Calculus to Formalize UML Activity Diagram for Business Process Modeling. In *Proceedings 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 47 – 54, Huntsville, AL, USA, 2003.

[Esh06] Rik Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Transactions on Software Engineering and Methodology*, 15(1):1–38, 2006.

[Flo03] Jacqueline Floch. *Towards Plug-and-Play Services: Design and Validation using Roles.* PhD thesis, Norwegian University of Science and Technology, 2003.

[GM05] Nicolas Guelfi and Amel Mammar. A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 283–290, Washington, DC, USA, 2005. IEEE Computer Society.

[HK00] Peter Herrmann and Heiko Krumm. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337, 2000.

[HK07] Peter Herrmann and Frank Alexander Kraemer. Design of Trusted Systems with Reusable Collaboration Models. In Sandro Etalle and Stephen Marsh, editors, *Trust Management*, volume 238, pages 317–332. IFIP International Federation for Information Processing, Springer, 2007.

[Hol03] G.J. Holzmann. *The Spin Model Checker, Primer and Reference Manual.* Addison-Wesley, Reading, Massachusetts, 2003.

[Hoo02] Jozef Hooman. Towards Formal Support for UML-based Developement of Embedded Systems. In *Proceedings PROGRESS 2002 Workshop, STW*, 2002.

[KBH07] Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007*, volume 4745 of *Lecture Notes in Computer Science*, pages 166–185. Springer–Verlag Berlin Heidelberg, September 2007.

[KH06] Frank Alexander Kraemer and Peter Herrmann. Service Specification by Composition of Collaborations — An Example. In *Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology)*, pages 129–133. IEEE Computer Society, 2006. 2nd International Workshop on Service Composition (Sercomp), Hong Kong.

[KH07a] Frank Alexander Kraemer and Peter Herrmann. Formalizing Collaboration-Oriented Service Specifications using Temporal Logic. In *Networking and Electronic Commerce Research Conference 2007 (NAEC 2007)*, pages 194–220, USA, October 2007. ATSMA Inc.

[KH07b] Frank Alexander Kraemer and Peter Herrmann. Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In Karsten Ehring and Holger Giese, editors, *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, volume 7 of *Electronic Communications of the EASST*. EASST, 2007.

[KHB06] Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk. Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. In R. Meersmann and Z. Tari, editors, *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), 2006, Montpellier, France*, volume 4276 of *Lecture Notes in Computer Science*, pages 1613–1632. Springer–Verlag Heidelberg, 2006.

[Kra07] Frank Alexander Kraemer. Building Blocks, Patterns and Design Rules for Collaborations and Activities. Avantel Technical Report 2/2007 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway, March 2007.

[Kra08] Frank Alexander Kraemer. UML Profile and Semantics for Service Specifications. Avantel Technical Report 1/2007 ISSN 1503-4097, Department of Telematics, NTNU, Trondheim, Norway, June 2008.

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam02] Leslie Lamport. *Specifying Systems.* Addison-Wesley, 2002.

[Rus00] John Rushby. Disappearing Formal Methods. In *High-Assurance Systems Engineering Symposium*, pages 95–96, Albuquerque, NM, November 2000. ACM.

[Slå07] Vidar Slåtten. Model Checking Collaborative Service Specifications in TLA with TLC. Project Thesis, August 2007. Norwegian University of Science and Technology, Trondheim, Norway.

[Stö05] Harald Störrle. Semantics and Verification of Data Flow in UML 2.0 Activities. In *Electronic Notes in Theoretical Computer Science*, volume 127, pages 35 – 52, 2005.

[VM94] Björn Victor and Faron Moller. The Mobility Workbench — A Tool for the π-Calculus. In David Dill, editor, *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

[VSvSB91] Chris A. Vissers, Guiseppe Scollo, Marten van Sinderen, and Hendrik Brinksma. Specification Styles in Distributed System Design and Verification. *Theoretical Computer Science*, 89:179–206, 1991.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA$^+$ Specifications. In L. Pierre and T. Kropf, editors, *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer-Verlag, 1999.

PAPER

EIGHT

ARCTIS AND RAMSES: TOOL SUITES FOR RAPID SERVICE ENGINEERING

Frank Alexander Kraemer.

Presented as poster at Norsk informatikkonferanse (NIK) 2007, November 19 – 21, 2007 Oslo, Norway.

Published in the proceedings of Norsk informatikkonferanse 2007, Tapir Akademisk Forlag, Trondheim.

Arctis and Ramses: Tool Suites for Rapid Service Engineering

Frank Alexander Kraemer

Abstract. For our highly automated service engineering approach SPACE, we built the tool suites Arctis and Ramses. Arctis focuses on abstract, reusable service specifications that are composed from UML 2.0 collaborations and activities. It supports the analysis of service specifications by model checking via TLC. A consistent specification can be transformed into UML state machines and components. For their implementation, Ramses contributes code generators to create executable systems.

8.1 Introduction — The SPACE Approach

As most services involve several participating components, our engineering approach for reactive systems SPACE [KBH07, KH06, KH07a, KH07b, KHB06] uses collaborative building blocks as reusable specification units to create more comprehensive services through composition. Figure 8.1 outlines the approach. A developer first consults a library to check if an already existing collaboration block or a combination of several blocks solves a certain task. Missing blocks can also be created from scratch and stored in the library for later reuse. The building blocks are expressed as UML models. The structural aspect, for example the participants and their multiplicity, is expressed by means of UML 2.0 collaborations. For the detailed behavior, we use UML 2.0 activities. They express the local behavior of each of the participants as well as their necessary interactions in a compact and self-contained way using explicit control flows. Coupling points later used for behavioral composition can be expressed by input and output parameter nodes. To define in which sequence these nodes may be invoked by the environment of a building block, we use so-called external state machines (ESMs). They refer with their transitions to the input and output nodes and define as such the externally visible behavior of a collaborative building block.

In a second step, the building blocks are combined to form more comprehensive services by composition. For this composition, we use UML 2.0 collaborations and activities as well. While collaborations provide a good overview of the structural aspect of the composition, i.e., which sub-services are reused and how their collaboration roles are bound, activities express the detailed coupling of their respective behaviors. Each sub-service is represented by a call behavior action referring to the respective activity of the building block. By connecting the individual input and output pins of the call behavior actions, the events occurring in different collaborations can be coupled with each other. The surrounding activity may also contain additional nodes to add further behavior, so that developers have powerful means to express the behavioral composition of the sub-services. Once complete, the service specifications are analyzed using

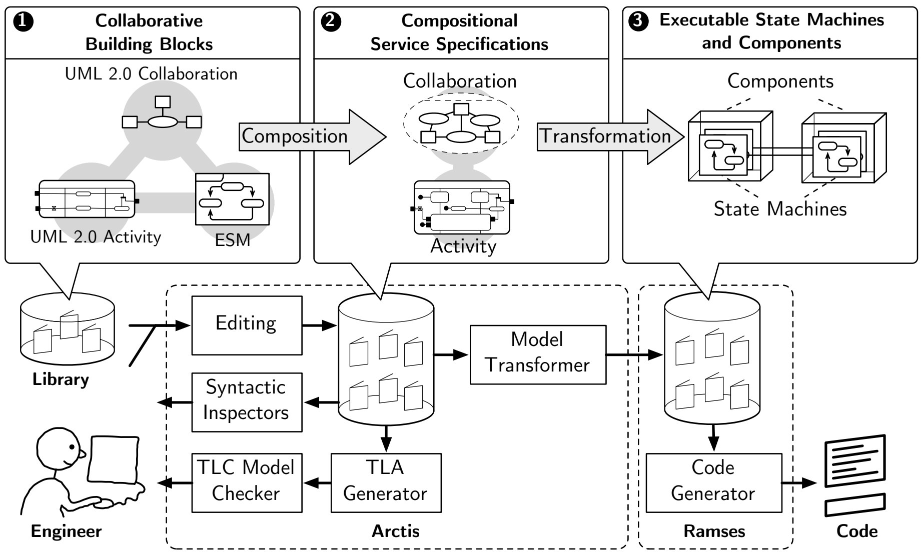


Fig. 8.1: A coarse sketch of the SPACE engineering approach and its tool support

model checking, as we will see later. Note that steps 1 and 2 can be executed iteratively, over several hierarchies, as each composed collaboration can in turn be abstracted by an ESM and used as a building block in more comprehensive specifications.

In a third step, the collaborative service specifications are transformed automatically into executable state machines and components by a model transformation. From these models, executable code can be generated easily in a successive step. So, the only manual work is done on the building blocks and their composition.

To ensure the consistency of the approach, the semantics of its languages, the composition mechanisms as well as the model transformation, we use Herrmann's compositional Temporal Logic of Actions (cTLA, [HK00]) as formal framework. We have formalized the activities [KH07a] as well as the state machines [KHB06]. The composition of service specifications can be reduced to the mechanism of process couplings in cTLA and the model transformation is a formal refinement step. Practitioners need not understand this formalism and they can focus on SPACE as an approach devoted to support the rapid creation of services. The approach relies on three principles to speed up development:

– **Collaborative Building Blocks.** By focusing on building blocks that are not only components but entire collaborations covering behavior of several components in a self-contained way, services may be composed of sub-services. This facilitates reuse, as sub-services typically serve an isolated task and are more likely to be useful in other applications than entire components.

– **Model Transformations and Code Generation.** By using automated steps from the more abstract service specifications towards the implementation, consistency is ensured between the specifications and coding errors are avoided. As state machines are generated automatically, a difficult and time-consuming manual synthesis is omitted.

– **Formal Analysis of Models.** By analyzing the abstract service specifications both in syntax and via model checking, errors can be found early during the development and on a high abstraction level, where problems can be studied independent from implementation details.

To effectively support these concepts, we have tool support [Kra] in the form of Arctis for the specification via collaborative building blocks as well as Ramses for the implementation of executable state machines. Both tools are realized as Eclipse plug-ins using the UML 2.0 repository of the UML2 project.

8.2 Support for Service Specifications in Arctis

To support the construction of building blocks consisting of activities, collaborations and ESMs, Arctis offers special actions and wizards, for example to create the skeleton of an ESM from an activity. In addition, a number of inspections ensures the syntactical consistency of building blocks.

For composite building blocks, where activities with their partitions and call behavior actions must be synchronized with the collaboration, special actions are available to update each of them. For example, Arctis automatically generates a corresponding activity for the behavioral specification of the composition. For each collaboration role, an activity partition is created and each collaboration use is represented by a call behavior action with its pins. This skeleton is then completed manually with activity flows and nodes that model the extra logic to couple the sub-collaborations. A number of inspections are executed on the model to check a consistent syntax. To analyze more advanced properties, we use model checking.

Model Checking. Because of the mapping from activities to cTLA presented in [KH07a], each activity diagram corresponds to a temporal formula. With the tool presented in [Slå07], this formula can be generated in form of TLA$^+$ [Lam02] and used as input for the model checker TLC [YML99]. If a collaboration is composed from sub-collaborations, we only use the abstract ESMs during model checking. This reduces the state space significantly. The superposition principle of cTLA as well as a formal refinement relationship between an activity and its ESM guarantee that also the complete system behaves correctly.

Model checking requires knowledge that we do not expect from the users of our tools. Therefore, not only the TLA specifications but also the theorems to ensure correct activities are generated automatically, either based on simple assertions in the form of stereotypes attached to the activity, or based on standard assumptions about what makes an activity well-formed. In addition, it is technically possible to project an error trace from TLC which documents erroneous behavior back into the activity diagram as token movements. In this way, the results of model checking can be given to the users, without them needing to know about temporal logic.

Transformation to State Machines. To create executable systems, we implemented an algorithm [KH07b] performing a model transformation from activities to state machines. The basic idea of this algorithm is to cut an activity into its partitions that correspond to different state machines. A token marking corresponds to a control state of a state machine, and each token movement is mapped to a state machine transition. The algorithm explores the state space of the activities by a partial model checking and constructs the necessary state machine transitions. For collaborations that have to be executed in several parallel sessions within one component, the transformation creates session state machines as described in [KBH07].

8.3 Support for State Machines and Components in Ramses

The models used for the design and execution of services goes back to early computer-controlled telecommunication systems and is based on communicating state machines. With its version 2.0, UML can be used to express such models. As general UML state machines allow behaviors that can be difficult to implement, we defined in [KHB06] a number of constraints on state machines, which may be ensured by syntactic inspections. In addition, state machines can be analyzed using validation algorithms based on the work of Floch [Flo03] and Sanders [San07]. These algorithms are currently developed further as part of the SIMS project [SIM07]. Currently, Ramses includes code generators [Kra03, Stø04] for various versions of the ServiceFrame platform [BHM02], a framework based on the state machine execution mechanisms of JavaFrame [HMP00].

8.4 Concluding Remarks

The functionality of Arctis will be expanded within the NFR-funded research and development project ISIS (Infrastructure for Integrated Services, [ISI07]). This project incorporates NTNU, HiA, Telenor, Tellu and Ericsson and is working on the development of service applications in the domain of mobile home automation. One aim is to introduce further abstraction levels in Arctis and develop the principle of creating services from reusable collaboration building blocks. ■

Bibliography

[BHM02] Rolv Bræk, Knut Eilif Husa, and Geir Melby. *ServiceFrame Whitepaper.* Ericsson NorARC, Asker, Norway, April 2002.

[Flo03] Jacqueline Floch. *Towards Plug-and-Play Services: Design and Validation using Roles.* PhD thesis, Norwegian University of Science and Technology, 2003.

[HK00] Peter Herrmann and Heiko Krumm. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337, 2000.

[HMP00] Øystein Haugen and Birger Møller-Pedersen. JavaFrame — Framework for Java Enabled Modelling. In Proceedings of Ericsson Conference on Software Engineering, September 2000.

[ISI07] ISIS Project Website. http://www.isisproject.org/, 2007.

[KBH07] Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007*, volume 4745 of *Lecture Notes in Computer Science*, pages 166–185. Springer–Verlag Berlin Heidelberg, September 2007.

[KH06] Frank Alexander Kraemer and Peter Herrmann. Service Specification by Composition of Collaborations — An Example. In *Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology)*, pages 129–133. IEEE Computer Society, 2006. 2nd International Workshop on Service Composition (Sercomp), Hong Kong.

[KH07a] Frank Alexander Kraemer and Peter Herrmann. Formalizing Collaboration-Oriented Service Specifications using Temporal Logic. In *Networking and Electronic Commerce Research Conference 2007 (NAEC 2007)*, pages 194–220, USA, October 2007. ATSMA Inc.

[KH07b] Frank Alexander Kraemer and Peter Herrmann. Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In Karsten Ehring and Holger Giese, editors, *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, volume 7 of *Electronic Communications of the EASST*. EASST, 2007.

[KHB06] Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk. Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. In R. Meersmann and Z. Tari, editors, *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), 2006, Montpellier, France*, volume 4276 of *Lecture Notes* *in Computer Science*, pages 1613–1632. Springer–Verlag Heidelberg, 2006.

[Kra] Frank Alexander Kraemer. The Ramses and Arctis Tools. http://www.item.ntnu.no/~kraemer/tools.

[Kra03] Frank Alexander Kraemer. Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart, 2003.

[Lam02] Leslie Lamport. *Specifying Systems.* Addison-Wesley, 2002.

[San07] Richard Sanders. *Collaborations, Semantic Interfaces and Service Goals: A Way Forward for Service Engineering.* PhD thesis, Norwegian University of Science and Technology, 2007.

[SIM07] SIMS Project Website. http://www.ist-sims.org/, 2007.

[Slå07] Vidar Slåtten. Model Checking Collaborative Service Specifications in TLA with TLC. Project Thesis, August 2007. Norwegian University of Science and Technology, Trondheim, Norway.

[Stø04] Alf Kristian Støyle. Service Engineering Environment for AMIGOS. Master's thesis, Norwegian University of Science and Technology, 2004.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA$^+$ Specifications. In L. Pierre and T. Kropf, editors, *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer-Verlag, 1999.

Part III

Appendices

APPENDIX

A

TOOL SUPPORT FOR THE RAPID COMPOSITION, ANALYSIS AND IMPLEMENTATION OF REACTIVE SERVICES

Frank Alexander Kraemer, Vidar Slåtten and Peter Herrmann.

Submitted to a journal for review.

Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services

Frank Alexander Kraemer, Vidar Slåtten, and Peter Herrmann

Abstract. We present the integrated set of tools *Arctis* for the rapid development of reactive services. In our approach, services are composed of collaborative building blocks that encapsulate behavioral patterns expressed as UML 2.0 collaborations and activities. Due to our underlying semantics in temporal logic, building blocks as well as their compositions can be transformed into formulas and model checked incrementally in order to guarantee that important system properties are kept. The process of model checking is fully automated. Error traces are presented to the users as easily understandable animations, so that no expertise in temporal logic is needed. In addition, the results of model checking are analyzed, so that in some cases automated diagnoses and fixes can be provided as well. The formal semantics also enables the correct, automatic synthesis of the activities to state machines which form the input of our code generators. Thus, the collaborative models can be fully automatically transformed into executable Java code. We present the development of a mobile treasure hunt system to exemplify the approach and the tools.

A.1 Introduction

Reactive systems consist of numerous devices like controllers, sensors and computation nodes which must be connected to provide services together that each single unit could not render separately. Unfortunately, the coordination of units often turns out to be more difficult than expected. One reason for that is the reactive nature of most systems dealing with several actuators or users; often, these systems follow a symmetric peer-to-peer structure in which several units may take initiative simultaneously. This makes the modeling of system synchronizations difficult and demands suitable modeling techniques.

Another inherent reason is the so-called *cross-cutting* nature of services. Obviously, to execute a service, we need a description of its physically deployable components. Their behavior can be expressed by means of state machines, as for example offered by SDL [ITU02] or UML [Obj07]. A service, however, is typically collaborative and spans across several components, and one component participates in several services. This collaborative dimension is orthogonal to that of components [Mik99]. If we only use component descriptions, services are specified only indirectly by the combined behavior of its participating components. In contrast, a more explicit description in the form of collaborations (see, for example [SCKB05]), not only has the benefit that service behavior can be understood and analyzed in isolation, but also opens new possibilities for

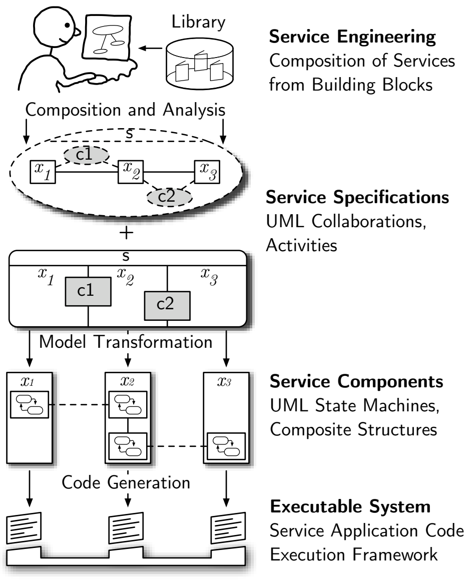


Fig. A.1: The SPACE Engineering Approach

the reuse of services as sub-functions provided by several components: Both, local functionality and solutions to problems that require coordination of several components, can be used directly in various applications.

Based on the idea to enable the reuse of collaborative, reactive behavior in the form of building blocks, we developed the engineering approach SPACE [KH06, KH07a, KH07b], depicted in Fig. A.1. To build a system, an engineer considers a library of reusable building blocks. In contrast to more traditional components, these building blocks may cover collaborative behavior among several components. They are expressed as a combination of UML 2.0 collaborations, activities and so-called *external* state machines (ESMs) to document their externally visible behavior. The building blocks are composed to more comprehensive ones, until the system specification is complete. After an analysis and potential corrections, the produced system specification is transformed automatically into state machines which can be implemented via code generation. The approach comprises three key features that speed up development:

– The design of a service is facilitated by applying reusable building blocks that are general or domain specific collaborations which can be integrated into several system descriptions. Due to the abstract description via external interfaces expressed by the ESMs, the internals of the building blocks do not have to be considered when they are applied.

– Engineers only work on collaborative, horizontal models expressed by activities. The component-oriented models expressed by state machines are

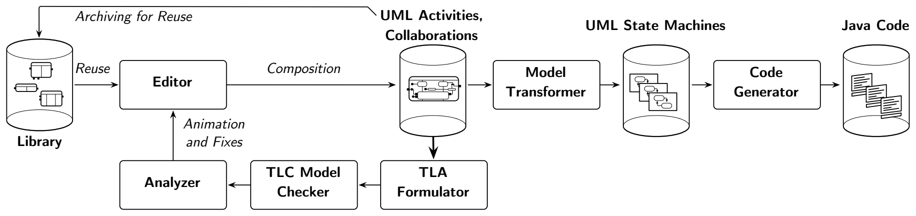


Fig. A.2: Overview of the tool support

derived fully automatically; a difficult and time-consuming manual synthesis of state machines is not necessary.

– Due to the mathematical background in temporal logic, the compositions and transformations are sound. Beyond that, model checking is possible. Due to the compositional properties of the approach, building blocks can be analyzed in separation which reduces the state space during model checking. As there exist many general criteria for a sound behavior of building blocks, the process of model checking can be automated, and engineers do not need to deal with any formal technique directly.

The theoretical foundations of the approach are detailed in [KH07a, KH07b, KHB06]. In the following, we focus on tool support for the approach, implemented by the Arctis plug-ins, as depicted in Fig. A.2. Building blocks are composed by engineers using the Arctis editor. The result can either be composite building blocks or entire systems, which are also special forms of building blocks. If desired, building blocks may be archived in the library for later reuse. To analyze a building block, its UML activity is transformed into a temporal logic formula and transferred to the TLC model checker [YML99]. It verifies the specification against theorems that we will explain later. If a theorem is violated, the analyzer tries to identify possible reasons and presents an error trace as animation in the activity to the engineer. Once a system specification is consistent and sound, it may be implemented automatically using a model transformation and code generation.

We will proceed as follows: In the next section, we present how our example of a mobile treasure hunt is composed from building blocks using the Arctis editor. In Sect. A.3, we present how Arctis supports the analysis of specifications by automating model checking and the provision of corrections in some cases. The transformation from activities to state machines is explained in Sect. A.4, and the code generation process is summarized in Sect. A.5. We close with an overview of related approaches and concluding remarks.

A.2 Composing Services from Building Blocks

As an example we develop a mobile treasure hunt, first described in [SB08]. In this game, a player receives a riddle via SMS. The answer of the riddle is associated with a certain location in the town the game takes place. To answer, the player does not reply via an SMS but tries to reach the location. Via GSM/GPS/WLAN positioning of the mobile phone, the player's position is known to the system; once at the correct goal, the next riddle is sent out until the final place is reached. To make the game more difficult, players must reach the target location within a limited time. For the discussion, we consider the realization for one player at a time. Using the mechanisms described in [KBH07], this specification can be expanded to handle multiple users as well.

The system is specified by a UML 2.0 collaboration as shown in the screenshot in Fig. A.3. On this level, the collaboration roles (depicted by rectangles) represent the components of the system. The location server is responsible for the positioning of mobile subscribers. The sms gateway provides sms-based communication from the users into the system and vice versa. We assume, that these components are realized and managed by an external operator; for our specification, they are therefore part of the environment, marked with a corresponding stereotype. In contrast, the three other components are constituents of the system we are going to implement. The game server is responsible for coordinating the game, assisted by the proximity and riddle servers. The collaboration uses (depicted as ellipses) decompose the overall functionality of the treasure hunt system into sub-services. Between the game server and the sms gateway, collaboration uses *s1: Single SMS Notification* and *s2: Send SMS* realize the necessary interaction with the player. Collaboration use *p: Proximity Alert* refers to a three-way collaboration between the location server, the game server and an additional proximity server. Within this collaboration, the proximity server constantly monitors the position of the user and alerts the game manager once the user is at a specified target.[1] A dedicated collaboration to query riddles from a data base is *r: Riddle Generation.*

A.2.1 Elementary Building Blocks

The services offered by the network operator are encapsulated within dedicated, collaborative building blocks. In addition to the interface behavior towards the operator's servers, such building blocks may also contain local behavior that simplifies the task to implement and integrate them with the rest of the system.

As UML collaborations and collaboration uses focus only on structural issues like role binding, we use a combination of UML activities and ESMs (external state machines) for the description of behavior. Figure A.4 shows the external representation for the building blocks encapsulating behavior towards the operator.[2] On the right side, they are shown in their instantiated form as call behavior actions. These are constructs of UML activities and can be composed within an enclosing activity, as we will see in Sect. A.2.2. The pins at their sides are used to control their behavior. As activities can be understood by token flow

[1]The decision to realize the proximity server as a separate component can be motivated by different reasons, for example a load analysis as explained in [BH93].

[2]The building block for the location tracking is used within collaboration *Proximity Alert*, as we will see below.

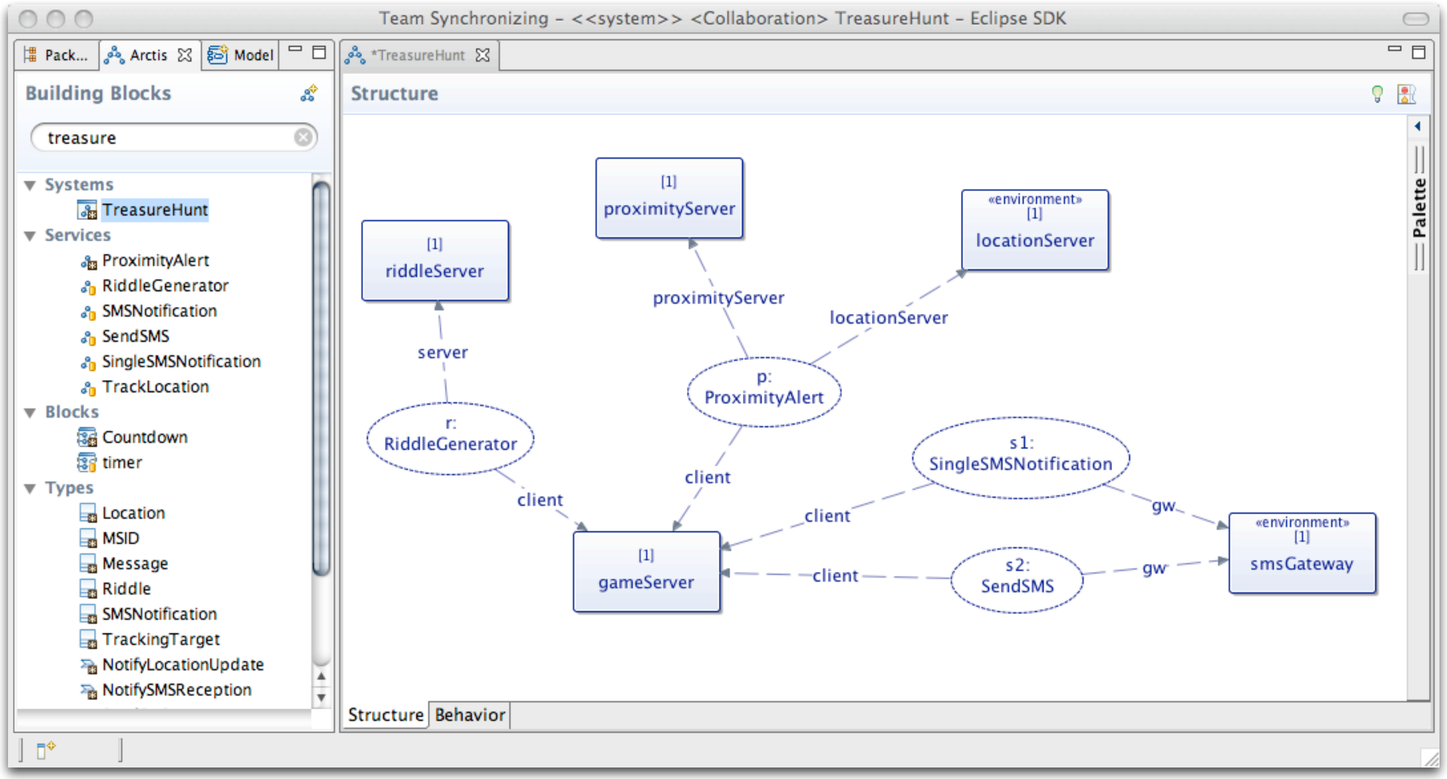


Fig. A.3: Eclipse workbench with the Arctis library browser and editor

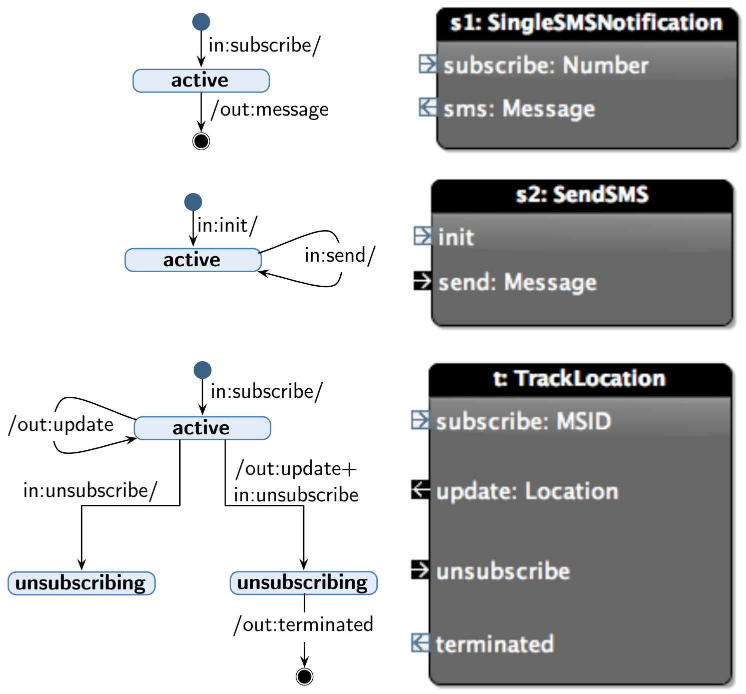


Fig. A.4: Building blocks provided by a network operator

semantics [Obj07], building blocks (instantiated as call behavior actions) are controlled by tokens passing their pins. The call behavior action *s1: Single SMS Notification* has only two pins: Input pin *subscribe* activates the block, awaiting an incoming SMS. This is issued by a token passing through the terminating output pin *sms*, which in turn deactivates the building block. As parameter, *subscribe* carries the number agreed upon with the operator that subscribers use to send in messages. Pin *sms* provides objects of type *Message* for each incoming SMS.

To document the valid sequences in which these pins may be invoked, we use the ESMs which are expressed by stereotyped UML state machines shown to the left of each building block. The labels of the transitions refer to the pins that a token passes. A slash distinguishes cause and effect, seen from the context instantiating the building block. The prefixes *in:* and *out:* are used to refer to input or output pins, respectively.

Following the description from above, the externally visible behavior of *s1: Single SMS Notification* is triggered in its starting transition from the outside via *in:subscribe/* and eventually terminates via */out:sms* which is triggered from the inside of the building block, i.e, it is spontaneous. Similarly, the building block *s2: Send SMS* is started via pin *init*. From then on, however, the client side may continuously send text messages via *sms*. As this is a so-called *streaming* node presented in black, tokens may pass while the block is active.

The block for the location tracking is a bit more complex. After a subscription that tells which mobile user (identified by a *mobile subscriber ID*, MSID) should be tracked, the client continuously receives updates while the subscriber moves via streaming pin *update*. This is expressed by the spontaneous self-transition */out:update* which has state *active* as source and target. Once the client is not interested in location data anymore, it may invoke pin *unsubscribe*, upon which the building block unsubscribes from the location server and terminates via *terminated*. The ESM also allows that an *update* is combined with an simultaneous *unsubscribe* via transition *out:update/in:subscribe*. This is useful when the reception of an update should be taken as trigger to unsubscribe.

The ESMs describe the behavior of the building blocks so that engineers may instantiate and compose them to build more comprehensive services, without looking at their internals. Furthermore, during the analysis of a building block via model checking, the behavior of the building blocks it consists of is abstracted by the ESMs as well, effectively reducing the state space. The internals of building blocks are only needed when components and their state machines are generated and are described by UML activities, as we will see in the next section. The building blocks are stored within a UML repository managed by Arctis. As each building block is a combination of a UML collaboration, an activity and an ESM, Arctis provides an editor that keeps these three views consistent. Syntactic inspections warn if any conventions are violated. Building blocks may be searched via the library of building blocks shown on the left hand side of Fig. A.3.

A.2.2 Composing Building Blocks

To create more comprehensive services from elementary building blocks, UML activities are used to describe their precise behavioral composition. As an example, we consider the collaboration for the proximity alert as described by the activity in the lower part of Fig. A.5. (The figure shows a premature design which we will analyze and improve in Sect. A.3.) The task of this sub-service is to notify the client once a mobile user reaches a certain target position. Each participant of the collaboration is represented by an activity partition. Proximity alert refers to the location tracking service, represented by call behavior action *t*. The client starts the sub-service by providing the MSID of the player to track and the target location, encapsulated by an object of type *Tracking Target*. When the tracking target arrives at the proximity server, it passes a fork node which duplicates the token. One copy follows the lower edge to the operation *extractLocation* in which the location is extracted and stored in the variable *target*. Within the same step, the other copy follows the upper flow leaving the fork so that MSID is extracted from the tracking target and the track location is started. From then on, the track location emits a token carrying the current location via *update* every time the subscriber changes position. This updated location is compared in the boolean operation *closeEnough* with the target location stored in the variable *target*. If the position is not yet close enough to

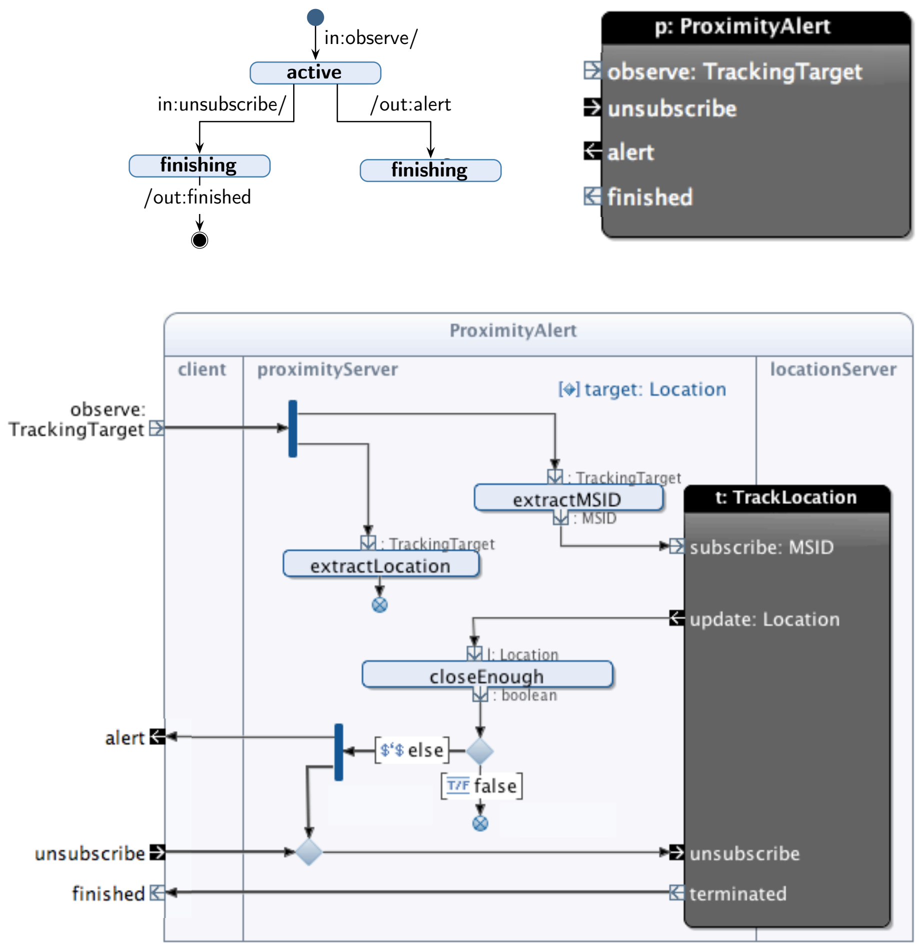


Fig. A.5: First solution of the proximity alert

the target, the *false* branch is chosen and the flow ends in the flow final node. If, however, the position is close enough to the target, the *else* branch is chosen, which notifies the client via *alert*. Within the same step, a token is pushed through *unsubscribe* of *t*, so that no more updates are received. Unsubscribe tokens coming from the enclosing context are directly forwarded to the location tracker. Likewise, the termination of *t* is forwarded to the client.

To create the specification, the location building block may simply be dragged into the editor. Arctis manages the assignment to activity partitions based on the role binding of the collaborations. As UML does not provide a language syntax to describe actions executed within the operations like *closeEnough*, the editor also maintains a Java file for each partition that contains corresponding methods which may be edited by the service engineer.

A.3 Automated Model Checking and Analysis

To analyze building blocks and complete systems, the Arctis editor constantly checks the model for a number of syntactic constraints. For a more thorough analysis of the behavior, Arctis employs the model checker TLC [YML99] based on the Temporal Logic of Actions (TLA, [Lam02]). Fig. A.2 outlines this process: When a building block is complete and syntactically correct, Arctis transforms the UML activity into TLA$^+$, the language for TLA, and starts the model checker TLC. If TLC reports an error, our tool visualizes the error trace and analyzes the results; in some cases, it provides diagnostics and proposes fixes that the user may apply. In the following, we will describe the details of this process by analyzing the proximity alert collaboration sketched in Fig. A.5.

A.3.1 Semantics in Temporal Logic

TLA specifications are structured as TLA$^+$ modules that describe behavior as sequences of steps. This mathematical interpretation fits well with the more graphical representation of activity behavior as token flows; stable states in which tokens rest in places are represented by the variables of a TLA specification, and the token movements are specified by TLA actions (see [KH07a]). Figure A.6 lists the TLA$^+$ module for the proximity alert, as generated by Arctis.[3] In its second line, the module declares the variables representing the states of the specification, followed by their initial values given by *Init*. After that, the TLA actions are declared, expressing the behavioral steps. The *Next* statement as well as *Spec* define the actual specification as the disjunction of all of these actions. The last part states some theorems which we will explain below. For details, we refer to [Lam02]. As we focus on the analysis of the coordination of concurrent behavior, we ignore in our TLA model the UML variables of the specification (like *target* in Fig. A.5) and UML operations on it. We therefore look at the version of the proximity alert in Fig. A.7, to make the discussion easier to follow.

Due to the refinement semantics employed in SPACE [KH07a], TLA actions are formulated in such a way that tokens only rest on places where they wait for other events to happen. This can be the expiration of a timer or the arrival of another token in a partition. For the proximity alert, tokens rest at the flows that cross partitions between client and proximity server, illustrated by the circles *q1..q4*, and represented in the TLA$^+$ module by the corresponding variables. We assign integers to them saving the number of tokens in the corresponding place. In addition, the ESM state of the location tracking building block is represented by variable *t*. This means, when analyzing the proximity alert, we do not consider the internal details of the location tracking, which reduces the state space. As the collaboration is open, that means, depends on the interactions from the enclosing context, we represent the variable of the enclosing ESM by variable *esm*.

[3]For readability, we adjusted the automatically derived names of variables and actions.

MODULE *ProximityAlert*

EXTENDS *Naturals*

VARIABLES $q1,\ q2,\ q3,\ q4,\ t,\ esm$

$$\begin{aligned}
&Init \triangleq \\
&\quad \land\ q1 = 0 \land q2 = 0 \land q3 = 0 \land q4 = 0 \\
&\quad \land\ t\ \ = \text{“off”} \land esm = \text{“off”}
\end{aligned}$$

$$\begin{aligned}
&observe \triangleq \\
&\quad \land\ esm = \text{“off”} \land esm' = \text{“active”} \\
&\quad \land\ q1'\ = q1 + 1 \\
&\quad \land\ \text{UNCHANGED } \langle q2,\ q3,\ q4,\ t\rangle
\end{aligned}$$

$$\begin{aligned}
&subscribe \triangleq \\
&\quad \land\ q1 > 0 \land q1' = q1 - 1 \\
&\quad \land\ t\ \ = \text{“off”} \land t' = \text{“active”} \\
&\quad \land\ \text{UNCHANGED } \langle q2,\ q3,\ q4,\ esm\rangle
\end{aligned}$$

$$\begin{aligned}
&update \triangleq \\
&\quad \land\ t = \text{“active”} \land t' = \text{“terminating”} \\
&\quad \land\ q2' = q2 + 1 \\
&\quad \land\ \text{UNCHANGED } \langle q1,\ q3,\ q4,\ esm\rangle
\end{aligned}$$

$$\begin{aligned}
&unsubscribe \triangleq \\
&\quad \land\ esm = \text{“active”} \land esm' = \text{“finishing”} \\
&\quad \land\ q3'\ = q3 + 1 \\
&\quad \land\ \text{UNCHANGED } \langle q1,\ q2,\ q4,\ t\rangle
\end{aligned}$$

$$\begin{aligned}
&unsubscribe2 \triangleq \\
&\quad \land\ q3 > 0 \land q3' = q3 - 1 \\
&\quad \land\ t\ \ = \text{“active”} \land t' = \text{“terminating”} \\
&\quad \land\ \text{UNCHANGED } \langle q1,\ q2,\ q4,\ esm\rangle
\end{aligned}$$

$$\begin{aligned}
&alert \triangleq \\
&\quad \land\ q2 > 0 \land q2' = q2 - 1 \\
&\quad \land\ esm = \text{“active”} \land esm' = \text{“alerted”} \\
&\quad \land\ \text{UNCHANGED } \langle q1,\ q3,\ q4,\ t\rangle
\end{aligned}$$

$$\begin{aligned}
&term1 \triangleq \\
&\quad \land\ t = \text{“terminating”} \land t' = \text{“off”} \\
&\quad \land\ q4' = q4 + 1 \\
&\quad \land\ \text{UNCHANGED } \langle q1,\ q2,\ q3,\ esm\rangle
\end{aligned}$$

$$\begin{aligned}
&term2 \triangleq \\
&\quad \land\ q4 > 0 \land q4' = q4 - 1 \\
&\quad \land\ esm = \text{“finishing”} \land t' = \text{“finished”} \\
&\quad \land\ \text{UNCHANGED } \langle q1,\ q2,\ q3,\ t\rangle
\end{aligned}$$

$$\begin{aligned}
&Next \triangleq \\
&\quad \lor\ observe \lor unsubscribe \lor unsubscribe2 \\
&\quad \lor\ alert \quad\ \lor subscribe \lor update \\
&\quad \lor\ term1 \quad \lor term2
\end{aligned}$$

$$Spec \triangleq Init \land \Box[Next]_{\langle q1,\ t,\ q3,\ q2,\ esm\rangle}$$

$$\begin{aligned}
&t\_deadlock \triangleq \Box \lor \text{ENABLED } (Next) \\
&\qquad\qquad\qquad\ \lor esm = \text{“finished”}
\end{aligned}$$

$$\begin{aligned}
&t\_bounds \triangleq \Box \land (q1 \quad \leq 5) \land (q2 \leq 5) \\
&\qquad\qquad\quad\ \land (q3 \leq 5) \land (q4 \leq 5)
\end{aligned}$$

$$\begin{aligned}
&t\_q1 \triangleq \Box((q1 > 0) \Rightarrow (t = \text{“off”})) \\
&t\_q2 \triangleq \Box((q2 > 0) \Rightarrow (esm = \text{“active”})) \\
&t\_q3 \triangleq \Box((q3 > 0) \Rightarrow (t = \text{“active”})) \\
&t\_q4 \triangleq \Box((q4 > 0) \Rightarrow (esm = \text{“finishing”}))
\end{aligned}$$

Fig. A.6: TLA$^+$ module for the semantics of *ProximityAlert*

In the initial state (declared by *Init*) all queues are empty ($q1..q4 = 0$) and the ESM of t as well as the enclosing ESM are in state *off*.[4] In this state, only action *observe* is enabled and can be executed. Actions refer to pairs of states, where unprimed variables (like $q1$) model the current state and primed variables (like $q1'$) refer to the next state. Consequently, action *observe* describes that the enclosing ESM changes from *off* to *active* and a token is placed into queue *q1*. In the UML activity, this corresponds to a token entering via *observe* and flowing into the transmission medium between the client and proximity server.

The other actions represent the residual steps our specification describes. Thus, *subscribe* models the arrival of a token in the proximity server upon which the location tracking is started. Action *update* represents that a new location arrived, upon which the client is notified via *q2* anf the location tracking is termi-

[4]When used as instantiated building blocks, the initial ESM state and all final states are mapped to the single state *off*, representing an inactive block. For the enclosing ESM of the main activity, we distinguish between the initial state *off* and the terminated state *finished* to reason about the life-cycle, as we will later see.

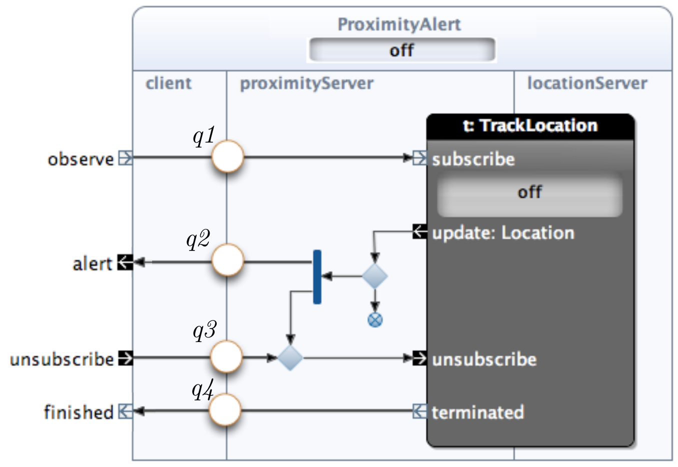


Fig. A.7: Simplified Proximity Alert without data

nated.[5] Actions *unsubscribe* and *unsubscribe2* describe how the client terminates the subscription to the proximity alert. Actions *term1* and *term2* model how a termination by the client propagates towards the server, and *alert* represents the notification of the client once the target is reached.

A.3.2 Theorems for Correct Building Blocks

A number of behavioral properties should hold for any building block. To check them, Arctis adds theorems to the TLA specification, as listed in the last compartment of Fig. A.6.

- To prevent communication errors, queue places between partition borders must be bounded. To detect violations, we state with theorem *t_bounds* that $q1$ to $q4$ must not exceed a certain number, here chosen to be 5. This has to hold in any state of the specification, which is expressed by the temporal operator $\Box$ (always).
- A building block must be free of deadlocks, in which it does not reach any of its final states. This is covered by theorem *t_deadlock*, which states that, at any time, the building block must either have reached a final state of its ESM (encoded as *finished*), or that one of its actions has to be enabled.
- The sub-activities within a building block must be used according to the ESM, so that pins are traversed only in the allowed order. In particular, this means whenever there is a token that can flow into a pin of the sub-activity, its ESM has to be in a state that accepts this token. For our example this means that whenever a token is in queue $q1$ that could enter

[5]An update not matching the target location corresponds to a step without state change which we left out for brevity.

t via *subscribe*, then the ESM of *t* (see Fig. A.4) must be in state *off*, as an entry via *subscribe* would activate it. Similarly, whenever a token from *q3* could unsubscribe, *t* has to be in state *active*. These constraints are expressed by theorems *t_q1* and *t_q3*.

– As the internal behavior of a building block must correspond to its external description, similar theorems are created for the enclosing ESM. For instance, whenever a token in *q2* could traverse via *alert*, the enclosing ESM (see Fig. A.5) has to be in state *active*, expressed by *t_q2*. Theorem *t_q4* works accordingly.

In addition to the general well-formedness properties described above, users may add application-specific constraints in form of assertions expressed by dedicated stereotypes in the UML activities. Examples for such properties are how often certain operations may or must be executed, or if certain operations are mutual exclusive.

A.3.3 Error Trace Animation in Arctis

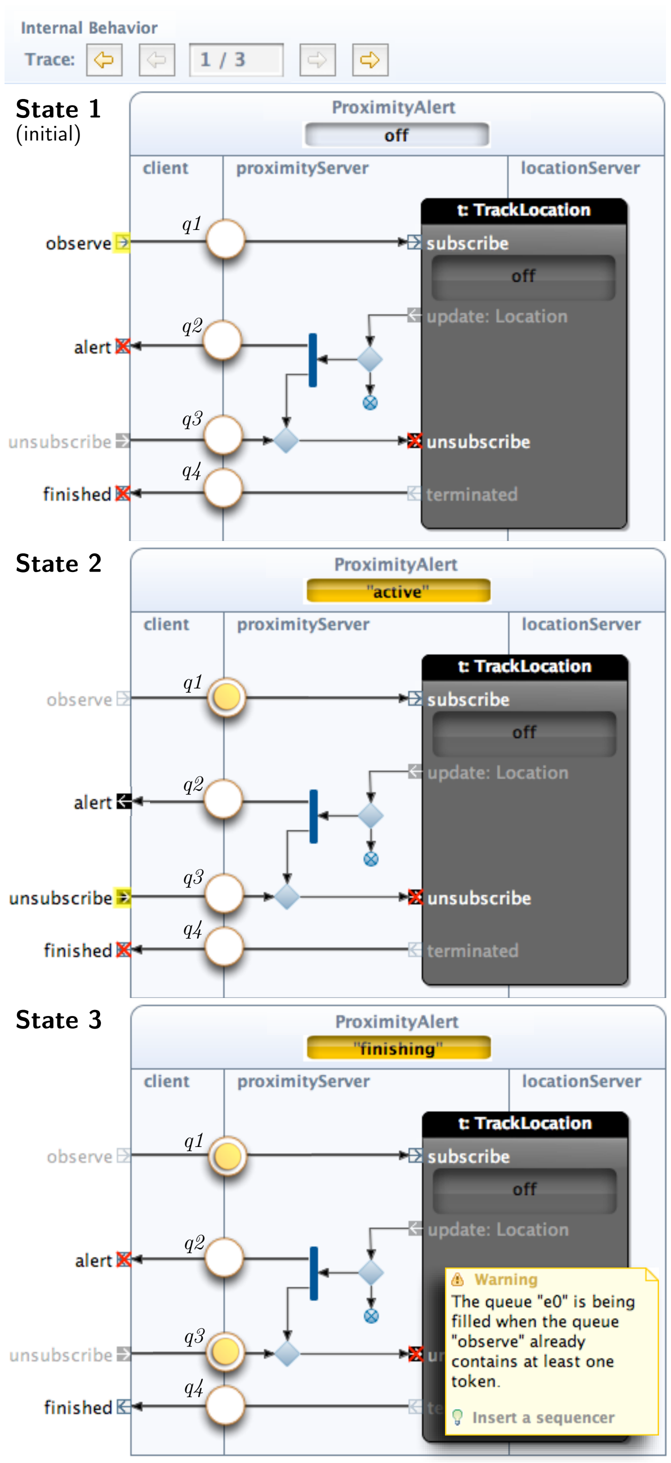
Arctis generates the TLA$^+$ module as described above and invokes the TLC model checker. During the generation of the TLA$^+$ module, a map from the variable names used in TLA to the elements of the activity is constructed. Therefore, if TLC reports that theorems are violated, our tool parses the textual error trace provided by TLC and maps each state back into the original activity diagram. Figure A.8 shows the Arctis editor after TLC reported that a theorem was violated. With a control bar, the user can jump through the error trace, animated by tokens in the editor. The state of the activity and each of its building blocks are represented by the corresponding ESM states. In addition, the pins of *t* and the parameter nodes[6] of the enclosing activity are marked based on their corresponding ESM states: A node that may release a token is marked yellow, while one that must not be passed by a token is shown with a red cross.

– In the initial state 1, a token may enter the activity via *observe* (in yellow) and place a token in queue $q1$. This changes the enclosing ESM to *active*, and state 2 is reached.

– In state 2, the client may send an unsubscribe, placing a token into $q3$.

– In state 3, TLC reports that theorem t_q3 is violated. We can see, that the token in $q3$ could enter the ESM of *t* via *unsubscribe*. The ESM, however, is in state *off*, because the token that should activate it still resides in $q1$.

[6]Conceptually, UML distinguishes between *parameter nodes* that are at the border of activities and *pins* which represent parameter nodes once the activity is instantiated as call behavior action. Both are represented by the same symbols.



The queue "e0" is being filled when the queue "observe" already contains at least one token.

Fig. A.8: Visualization of TLC's error trace in Arctis

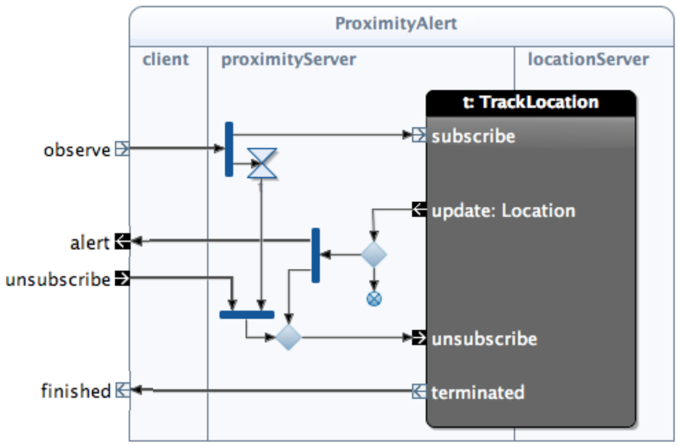


Fig. A.9: Suggested improvement by Arctis with sequencing

A.3.4 Automatic Diagnose and Fixes

The presentation of the traces within the editor is already helpful, especially as users do not have to consider any temporal logic formulas. In addition, Arctis can in many cases provide a more distinct diagnosis and suggest improvements. For that, each violated theorem triggers a number of pattern searches that take the UML activity as well as TLC's error trace as input. In the example, Arctis detected a match for the situation that a token overtakes another one during the transmission between partitions: Between state 2 and state 3, $q3$ is filled while $q1$ has not yet been emptied. This may be intended by the designer. The fact, that the token arriving in $q3$ harms a theorem, however, is a reason for Arctis to report this situation.

As a remedy, Arctis proposes to add a sequencing construct, so that a token in $q3$ can only proceed towards unsubscribe *after* $q1$ was consumed. The altered design is shown in Fig. A.9, after Arctis added an additional fork, a join node and a timer.[7] Before a token can move from $q3$ into *unsubscribe* of t, it has to wait in the join node until the other incoming flow can offer a token. This may only happen after a token was consumed from $q1$, which enforces the desired sequence. The additional timer prevents that tokens attempt to pass through *subscribe* and *unsubscribe* within the same step.

A.3.5 A Building Block to Handle Mixed Initiatives

We let Arctis analyze the improved design. After a new analysis, Arctis reports that a theorem was violated and presents the error trace. For brevity, we directly consider its last state shown in Fig. A.10. We can see that in this state, a player must have reached the target position, as one token is in queue $q2$. This token

[7] While elements are added and connected automatically, it is up to the user to adjust their layout.

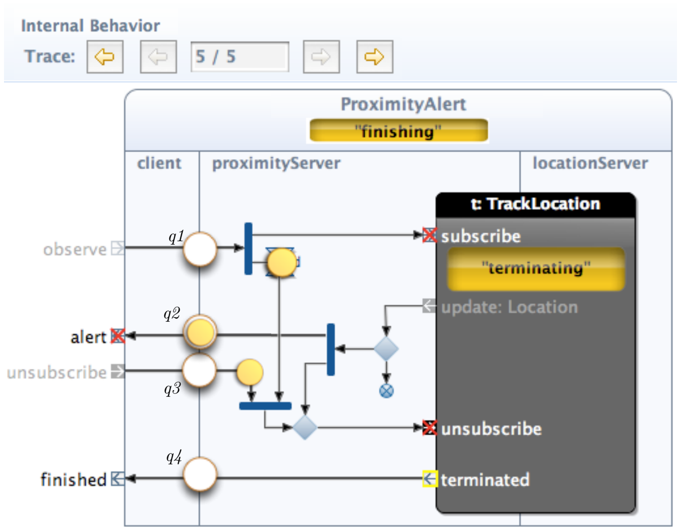


Fig. A.10: New error situation in the altered design

may have only arrived there via pin *update* of *t*. However, in the meantime, the client has chosen to unsubscribe, since the join node following $q3$ contains a token as well. This reveals a situation that is typical for systems in which several active components may take initiatives at the same time, due to the buffered communication. These two initiatives are in conflict. Once identified, such a situation can be handled by assigning primary and secondary priorities to the conflicting partners. An initiative from the primary side is accepted in all cases. For the secondary side, this means that it must be prepared to receive a primary initiative even after it issued an initiative itself, and obey the primary one; the secondary is in this case discarded. The solution may sound trivial, but such situations are intricate to get right, as the generic solution is combined with the complexity of the rest of the application. Thus, often it is not treated with the appropriate care.

As mixed initiatives are so common in this kind of system, we provide special building blocks in our library that solve such situations (see also [KSH07]). In the following, we present one in which the side that starts the interactions has secondary priority, called *Mixed Initiative Secondary Starter*, *MISS* for short.

The internal behavior is represented by a network of activity nodes that implements the desired behavior, shown in Fig. A.11. This activity appears quite complex on the first glance. However, an engineer using this building block never has to look at the inside as presented here; the external description is sufficient. It is given by two local ESMs in Fig. A.12 that describe the external behavior on each of the participants of the building block. The side with secondary priority starts the block. If the primary side takes initiative, the block eventually terminates on the secondary side via *primWins*. If the secondary side takes its initiative (in the treasure hunt this means that the timer expired), it has to wait

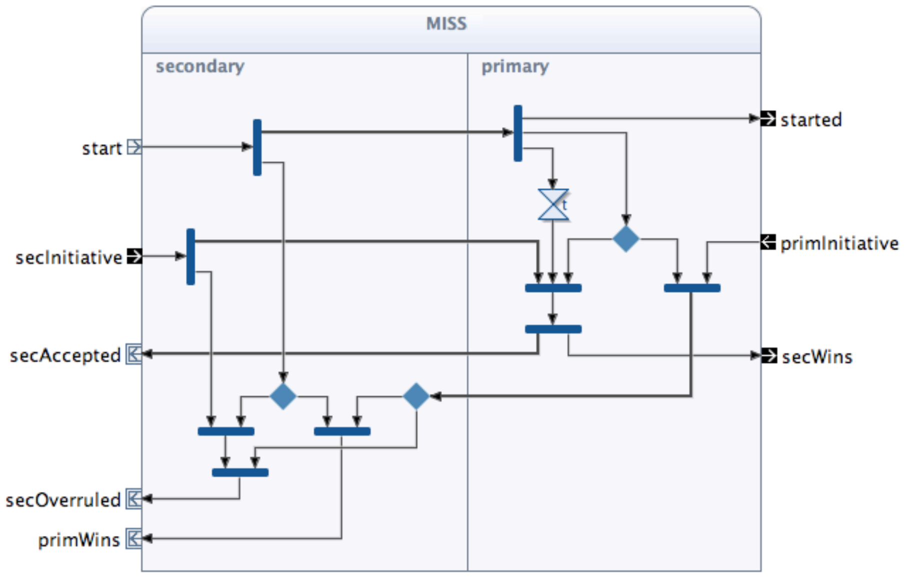


Fig. A.11: Building block to handle mixed initiatives

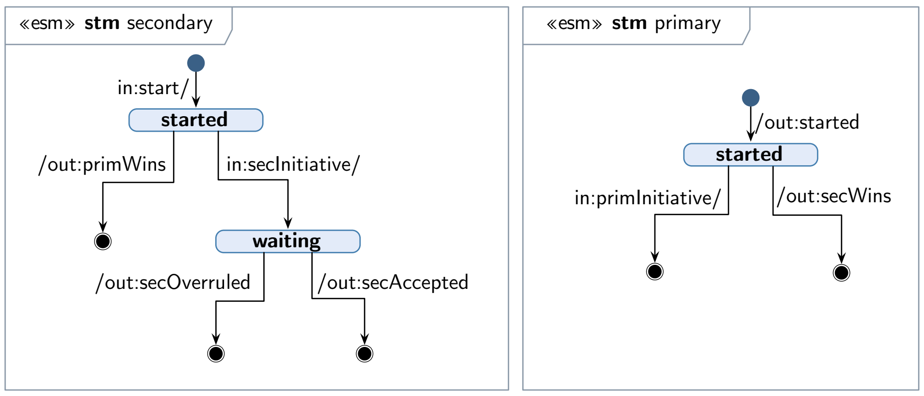


Fig. A.12: Local ESMs for both participants of *MISS*

for the primary side to either confirm via *secAccepted* or, if the primary side took initiative in the meantime, be overruled and receive a *secOverruled*. The ESM for the primary side is easier, as its initiative always succeeds, and no waiting for a confirmation in necessary.

For the proximity alert, we apply the mixed initiative block with the starting secondary side assigned to the client, as shown in Fig. A.13. In the strict sense, this means a slight advantage for the players, as an arrival is counted if the corresponding notification reaches the proximity server before the timeout. Later, during the usage of the proximity alert, we need to know if the initiative of the

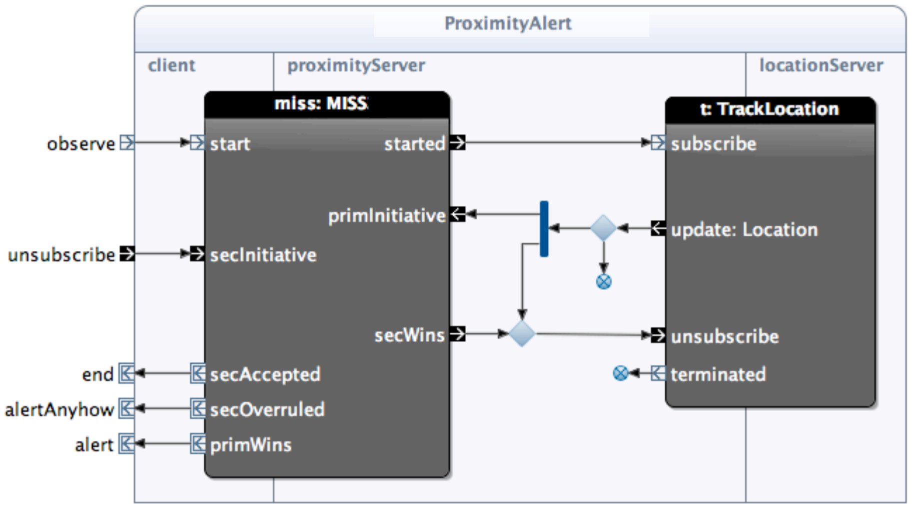


Fig. A.13: Correct Proximity Alert with the Mixed Initiative Building Block

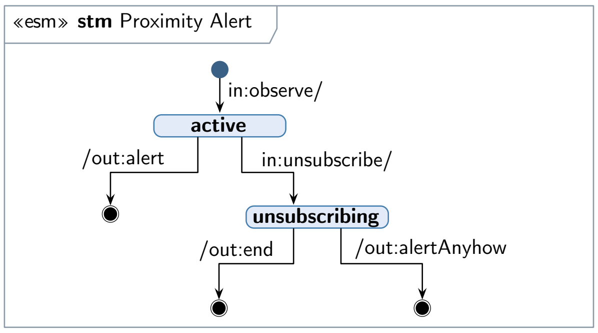


Fig. A.14: ESM for the corrected proximity alert

client was overruled. Therefore, we propagate this via the ESM of the proximity alert in Fig. A.14 using *alertAnyhow*. The introduction of a building block to handle this situation makes this design choice explicit. If we want to change this policy (so that for example the arrival of the player should get priority over the unsubscription), we would simply replace this block by one in which the primary side starts and assigns the corresponding roles to the proximity server and the client.

A.3.6 The Complete Treasure Hunt System

The complete behavioral system is described by the activity in Fig. A.15. Each collaboration use from the system collaboration from Fig. A.2 is represented by a corresponding call behavior action (*s1*, *s2*, *p*, *r*). In addition, it contains two auxiliary activity blocks *t2: Timer* to meassure time and *c: Countdown* as a decrementing counter. These blocks are local to the game server and help to describe the composition between the other collaborations. Therefore, Arctis draws them in blue.

At startup, *s1*, *s2* and counter *c* are initialized as tokens are emitted from the three initial nodes *i1..i3*. Then, the single SMS notification *s1* is waiting for an incoming SMS to start a game. Once it arrives, the player's MSID is extracted, upon which a welcome message is produced and sent out via *s2*. Within the same step, the riddle generator *r* is queried for the first riddle. It answers by issuing the next target, the granted time for the completion as well as the question in form of an SMS message. The target is used to start the proximity alert collaboration *p*. In the same step, timer *t2* is started with the granted time as input and the question is sent out to the player. It is now up to the player to move fast enough to the right target, upon which the proximity alert terminates via *alert*, which stops the timer. In addition, a token is sent through the countdown, which determines if more riddles should be sent out. In this case, after decreasing its internal counter, it directs the token to *continue*, which triggers another round. Otherwise, the game is ended successfully and a message is sent to the player. In case the player arrives too late at the target, the timeout from *t2* causes an unsubscribe from the proximity alert and the player is notified that the game is lost.

A.4 Automated Transformation

The UML activities of the building blocks together with the Java methods for the content of the call operation actions constitute a complete system description. To split this description into separate components, the activities have to be transformed into executable state machines that can be implemented via code generation to run on our execution platforms, as we will detail in Sect. A.5. Some concepts found in activities have their direct correspondence in state machines. Call operation actions are executed as operations that are part of a transition. Operations on variables stay largely unchanged, and decisions in activities map to choice pseudostates in a state machine. The remaining concepts, however, are fundamentally different (see [KH07b]):

- In contrast to the explicit control states of state machines, activities represent their states indirectly via the different token markings that occur during the execution. The transformation has to find all reachable token markings and map them to control states.
- The token movements must be mapped to transitions of state machines. There is, however, no one-to-one mapping between activity flows and state machine transitions either. Depending on the markings, one flow may have to be represented by several state machine transitions. This is the case for join nodes, where tokens have to wait until all incoming edges can fire.

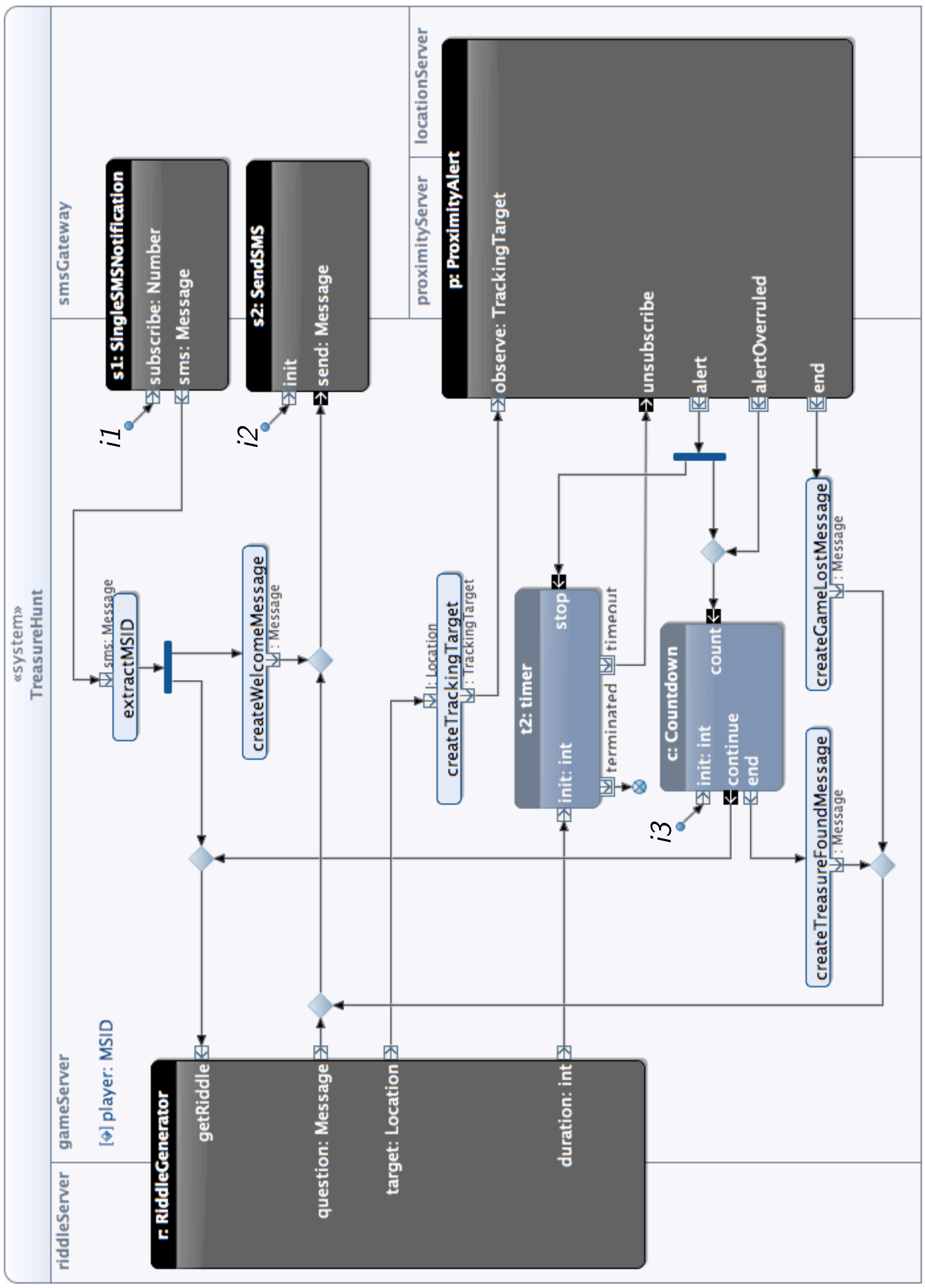


Fig. A.15: Complete system specification of the treasure hunt (screen capture)

– As components communicate via buffered message exchange, flows crossing partition borders have to be split up and translated into corresponding signal transmissions. If a flow carries objects, signal types have to take these objects as payload.

The search for reachable markings implies a state space exploration of the system's specification. To reduce the state space, we employ a strategy that, similar to our strategy in model checking, utilizes the external behavior of the building blocks as described by ESMs. Only one state machine is produced at a time. Therefore, we only need to consider those building blocks with its internals that *directly* contribute to the state machine under construction. The other building blocks are abstracted by their ESMs. When we create the state machine for the game server, for example, we may disregard the internals of *t: Track Location*, while the other building blocks need to be integrated. To generate the state machine for the proximity server, only the building blocks for the mixed initiative and the location tracking have to be seen from the inside.

The detailed algorithm to construct the state machine transition identifies the events within a partition that correspond to events in state machines. These are the expiration of timers and the arrival of signals (resp. tokens entering a partition). By traversing the activity graph and taking into account the current marking, the state machine transition is constructed successively. The details of the algorithm are explained in [KH07b]. In the following, we illustrate the transformation process for the game server component.

A.4.1 State Machine for the Game Server

Figure A.16 shows the state machine automatically produced by the Arctis transformation from the activity of Fig. A.15. As general UML state machines can be used in various ways, we describe in [KHB06] rules for transitions so that the state machines may be executed efficiently. For example, each transition with exception of the initial one must be scheduled by a signal or a timeout. The application of these rules is noted by the stereotype *executable*.

The initial transition, initializes the counter and starts the SMS notification, whereupon the state machine changes into state *state_1* and waits for an incoming SMS. Once the SMS arrives, the MSID of its sender is extracted, from which a welcome message is generated that is sent back to the player. Within the same transition, the riddle server is queried for a new riddle via signal *GetRiddle*[8]. This implements the flow in Fig. A.15 starting within *s1* via pin *sms.* In a similar way, the other flows in Fig. A.15 are transformed into transitions of the state machine of Fig. A.16.

The generated state machine handles the mixed initiative between the timer and the alert correctly. This is visible in control state *state_4*, reached after a timeout, where the state machine is prepared to receive both an acknowledgement of the timeout as well as a primary initiative modeling the arrival of the player at the target.

[8]The signal names are derived from the names of activity edges, not shown here.

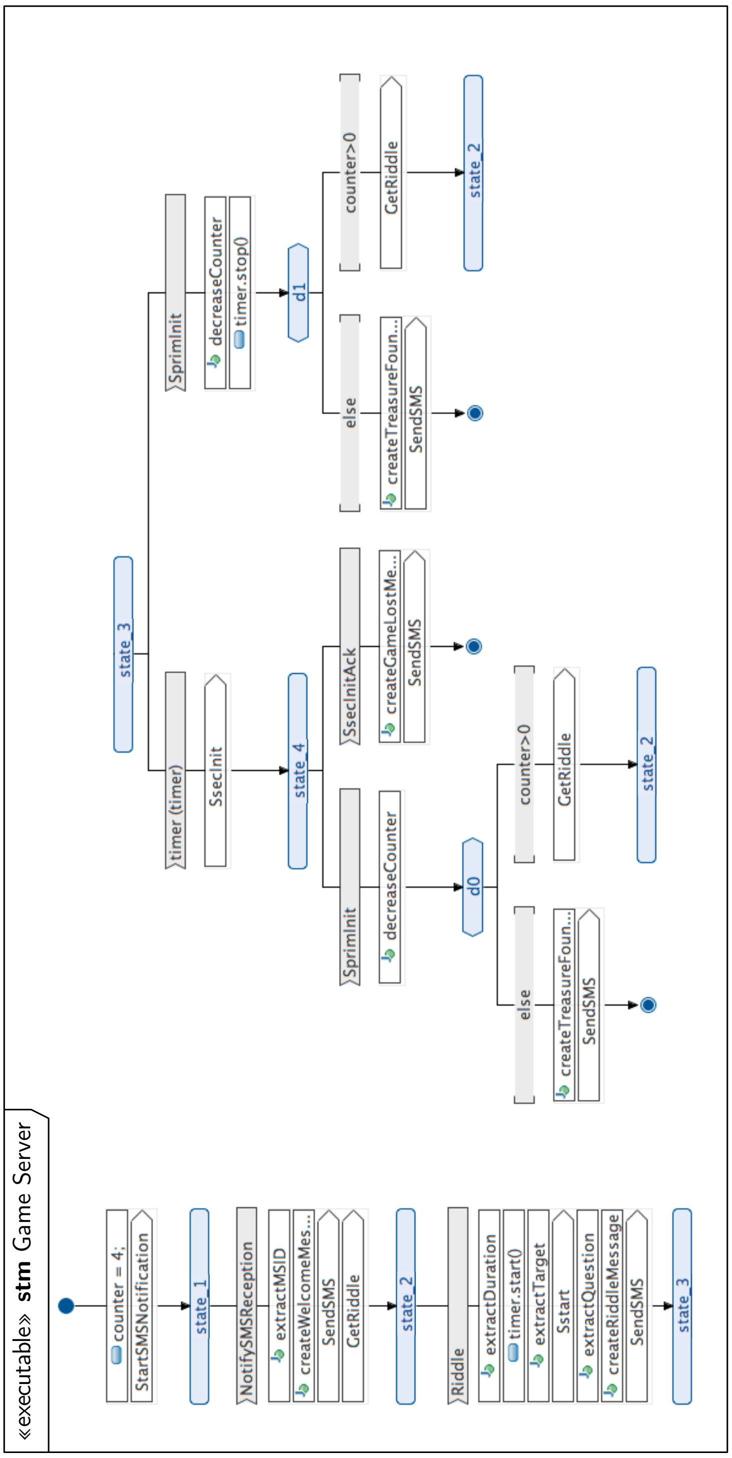


Fig. A.16: Executable state machine for the game server generated by Arctis

A.4.2 Correctness of the Transformation

Obviously, it is important that the generated state machines behave exactly as implied by the activities of the specifications. To ensure this, we use temporal logic as well. Similar to the semantics of activities presented in Sect. A.3.1, we defined formal semantics of the executable state machines in temporal logic in [KHB06]. A system of state machines can therefore be presented as a TLA specification $Spec_E$. As the implementation relation corresponds to logical implication in TLA, we have to prove that $Spec_E \Rightarrow Spec_A$ holds, where $Spec_A$ is the TLA specification of the system as expressed by activities (see [KH07a]). This relationship can be shown by a TLA refinement proof as demonstrated in [Kra08]. The necessary refinement mapping [AL91] is easy to find using the guidelines described in [KH07b]. Note, however, that such a reasoning is only necessary to ensure the soundness of the transformation once. During the implementation of systems, the service engineers can then rely on the tool to execute the transformation correctly.

A.5 Code Generation from State Machines

The mechanisms for the execution of state machines go back to principles found in telecommunication systems [BHS81] and use a run-time support system that schedule the execution of the state machine transitions, further described in [KHB06]. Through this additional level of multiplexing, many state machine instances can be executed within the same operating system thread, which is important for systems to scale. While these mechanisms can be implemented on a variety of platforms, we focus currently on Java and use the ServiceFrame/ActorFrame execution platforms [BHM02]. These frameworks take care of addressing and routing. The implementation and scheduling of state machine transitions are based on JavaFrame [HMP00]. Code for these frameworks is generated automatically with the tool described in [Kra03, Stø04]. The code generator creates OSGi bundles for the components that can be deployed on different machines.

A.6 Related Approaches

A number of other tools combine UML modeling with formal analysis techniques. The majority of these approaches directly uses state machines as the main specification units. HUGO [KM02], for example, verifies UML state machines against UML interactions using the SPIN model checker [Hol03], and UPPAAL [LPY97] to check real-time properties. Fujaba [BGHS04] uses so-called real-time state charts that represent behavioral patterns and utilizes HUppaal [ABB$^+$01] for their verification. The specifications in OMEGA [Hoo02] are based on state machines as well. Using the model checker IF [BGM02], they are verified against properties expressed by special observer state machines, as described in [OGO04].

Analysis of activities is done for example in [GM05] via SPIN. In [DS03], UML activities analyzed using π-calculus. Safety and liveness properties are expressed using the modal mu-calculus and checked using the MWB tool [VM94]. Similarly, Eshuis [Esh06] uses the model checker NuSMV to check the consistency of activity diagrams. The difference of these approaches to ours lies mainly in the semantics employed for the activities and the domain of application. While they focus on activities more from a perspective of business processes assuming a central clock or synchronous communication, we need for our activities reactive semantics [KH07a] reflecting the transmission of asynchronous messages between distributed components.

There are other tools that present the results of a model checker in terms of a graphical model. vUML [LP99] automatically creates PROMELA specifications from UML state charts and model checks them using SPIN. Like us, they mostly check general properties that the users do not specify manually, but they also allow to declare certain states as erroneous or desired goals. Any error traces are presented as sequence diagrams. Another tool is Theseus [GCKK06] which visualizes error traces from the SPIN and SMV model checkers onto UML 1.4 state chart diagrams, and also generates UML sequence diagrams from the trace. While both of the above tools visualize the trace, they do not try to find a reason for the error. Moreover, as error traces are presented as sequence diagrams, the user has manually find the relation to the original source model. In our case, errors are visible within the same editor used to create the specification.

In [FF06], a method is proposed for visualizing soundness violations of workflow Petri nets [vdA98], detected by the Woflan tool [vdA99], in the WoPeD tool [wop08]. Soundness violation is separated into five violation classes and a list of eleven error reasons is presented. In the case of a violation, the violating nodes are highlighted with the violation class and the error reason. If a violation is caused by a certain firing sequence of the net, an animation can be shown. Since this approach works on workflow Petri nets, it is quite close to the UML activities used in our case. However, similar to the works on activities mentioned earlier, focus lies on business processes, not on distributed, reactive components with asynchronous communication.

The tool support provided by the SIMS project uses collaborations as well, albeit in a form that is complementary to the current approach in Arctis. In SIMS, elementary collaborations describe a pair of behavioral interfaces [CFS08]. These can be connected within composite collaborations to describe, how an overall service goal may be achieved. Engineers are supported by validation algorithms that check compliance of state machines with behavioral interfaces. However, these state machines have to be constructed manually.

The SDL pattern tool (SPT, [DEG04]), supports the integration of patterns into SDL designs. The patterns are integrated within the component oriented perspective expressed by SDL processes. In contrast to our encapsulated building blocks, patterns are expressed as SDL fragments that have to be integrated into the state machine under construction.

A.7 Concluding Remarks

Arctis is used and advanced within the applied research project ISIS (Infrastructure for Integrated Services) funded by the Research Council of Norway. In this project we develop methods, tools and platforms for the rapid specification and deployment of services in the domain of home automation. We believe that the collaboration-oriented approach underlying Arctis is ideal in this setting: While there exists a number of rather stable sub-services that provide some basic functionality, it is the challenge to compose them quickly as demonstrated in the example. Obviously, the development time of a reuse-based approach depends heavily on which building blocks already exist. For the presented system, we could reuse the blocks for SMS communication and location tracking as well as the one to handle mixed initiatives from previous projects. Therefore, editing the entire system as presented in Fig. A.13 and A.15 took us less than one hour.

In our opinion, the specification style supported by our approach is quite intuitive. The main specification of the system as depicted in Fig. A.15 is very close to an informal functional description that can be the result of a requirements analysis. It focuses on the distribution of responsibilities and decomposes the system according to its sub-functions. In contrast, state machines (which in our approach are never read by humans) provide a less comprehensive view. To understand them, detailed signal transmission must be considered, and elements related to a single function (like counters, timers or the coordination of mixed initiatives) are mixed with each other. In activities, on the other side, related functions are encapsulated within building blocks.

For the analysis, we follow the strategy proposed by Rushby in "Disappearing Formal Methods" [Rus00], to hide formal methods in tools in such a way that users are not directly concerned with them. In our experience, this strategy not only reduces the threshold to analyze models thoroughly. This an incentive for the use of rigorous modeling in the first place and integrates well with the paradigms of the Model-Driven Architecture (MDA, [Obj03]). Based on case studies within the ISIS project, we are currently expanding the analytical capabilities of Arctis, so that more automated fixes and corrections can be offered. That gives even better assistance to the engineers which, in consequence, reduces development time further. ■

Bibliography

[ABB$^+$01] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. UPPAAL: Now, Next, and Future. In *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 99–124. Springer-Verlag, 2001.

[AL91] Martín Abadi and Leslie Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[BGHS04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In *Proc. of the International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004*, pages 1–20, October 2004.

[BGM02] Marius Bozga, Susanne Graf, and Laurent Mounier. IF-2.0: A Validation Environment for Component-Based Real-Time Systems. In *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 343–348. Springer-Verlag, 2002.

[BH93] Rolv Bræk and Øystein Haugen. *Engineering Real Time Systems: An Object-Oriented Methodology Using SDL.* The BCS Practitioner Series. Prentice Hall, 1993.

[BHM02] Rolv Bræk, Knut Eilif Husa, and Geir Melby. *ServiceFrame Whitepaper.* Ericsson NorARC, Asker, Norway, April 2002.

[BHS81] R. Bræk, O. Helle, and F. Sandvik. SOM — A SDL Compatible Specification and Design Methodology. In *4th International Conference on Software Engineering for Telecommunication Switching Systems, Conventry*, volume 198, pages 111–117, July 1981.

[CFS08] Cyril Carrez, Jacqueline Floch, and Richard Sanders. Describing Component Collaboration using Goal Sequences. In René Meier and Sotirios Terzis, editors, *Distributed Applications and Interoperable Systems - Proceedings of DAIS 2008, Oslo, Norway*, volume 5053 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2008.

[DEG04] Jörg Dorsch, Anders Ek, and Reinhard Gotzhein. SPT - The SDL Pattern Tool. In Daniel Amyot and Alan W. Williams, editors, *System Analysis and Modeling, 4th International SDL and MSC Workshop, SAM 2004, Ottawa, Canada, June 1-4, 2004, Revised Selected Papers*, volume 3319 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2004.

[DS03] Yang Dong and Zhang Shensheng. Using π-Calculus to Formalize UML Activity Diagram for Business Process Modeling. In *Proceedings 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 47 – 54, Huntsville, AL, USA, 2003.

[Esh06] Rik Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Transactions on Software Engineering and Methodology*, 15(1):1–38, 2006.

[FF06] Christian Flender and Thomas Freytag. Visualizing the Soundness of Workflow Nets. In *Proceedings 13th Workshop Algorithms and Tools for Petri Nets, AWPN*, pages 47–52, Hamburg, Germany, 2006.

[GCKK06] Heather Goldsby, Betty H. C. Cheng, Sascha Konrad, and Stephane Kamdoum. A visualization framework for the modeling and formal analysis of high assurance systems. In *MoDELS*, pages 707–721, 2006.

[GM05] Nicolas Guelfi and Amel Mammar. A Formal Semantics of Timed Activity Diagrams and its PROMELA Translation. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 283–290, Washington, DC, USA, 2005. IEEE Computer Society.

[HMP00] Øystein Haugen and Birger Møller-Pedersen. JavaFrame — Framework for Java Enabled Modelling. In Proceedings of Ericsson Conference on Software Engineering, September 2000.

[Hol03] G.J. Holzmann. *The Spin Model Checker, Primer and Reference Manual.* Addison-Wesley, Reading, Massachusetts, 2003.

[Hoo02] Jozef Hooman. Towards Formal Support for UML-based Developement of Embedded Systems. In *Proceedings PROGRESS 2002 Workshop, STW*, 2002.

[ITU02] ITU-T. *Recommendation Z.100: Specification and Description Language (SDL)*, August 2002.

[KBH07] Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007*, volume 4745 of *Lecture Notes in Computer Science*, pages 166–185. Springer–Verlag Berlin Heidelberg, September 2007.

[KH06] Frank Alexander Kraemer and Peter Herrmann. Service Specification by Composition of Collaborations — An Example. In *Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology)*, pages 129–133. IEEE Computer Society, 2006. 2nd International Workshop on Service Composition (Sercomp), Hong Kong.

[KH07a] Frank Alexander Kraemer and Peter Herrmann. Formalizing Collaboration-Oriented Service Specifications using Temporal Logic. In *Networking and Electronic Commerce Research Conference 2007 (NAEC 2007)*, pages 194–220, USA, October 2007. ATSMA Inc.

[KH07b] Frank Alexander Kraemer and Peter Herrmann. Transforming Collaborative Service Specifications into Efficiently Executable State Machines. In Karsten Ehring and Holger Giese, editors, *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, volume 7 of *Electronic Communications of the EASST*. EASST, 2007.

[KHB06] Frank Alexander Kraemer, Peter Herrmann, and Rolv Bræk. Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. In R. Meersmann and Z. Tari, editors, *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), 2006, Montpellier, France*, volume 4276 of *Lecture Notes in Computer Science*, pages 1613–1632. Springer–Verlag Heidelberg, 2006.

[KM02] Alexander Knapp and Stephan Merz. Model Checking and Code Generation for UML State Machines and Collaborations. In G. Schellhorn and W. Reif, editors, *FM-TOOLS 2002: 5th Workshop on Tools for System Design and Verification*, Report 2002-11, Reisensburg, Germany, 2002. Institut für Informatik, Universität Augsburg.

[Kra03] Frank Alexander Kraemer. Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart, 2003.

[Kra08] Frank Alexander Kraemer. From Activities to State Machines: Refinement Proof for a System. Avantel Technical Report 1/2008, Department of Telematics, Norwegian University of Science and Technology, 2008.

[KSH07] Frank Alexander Kraemer, Vidar Slåtten, and Peter Herrmann. Engineering Support for UML Activities by Automated Model-Checking — An Example. In *Proceedings of the 4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE)*, November 2007.

[Lam02] Leslie Lamport. *Specifying Systems.* Addison-Wesley, 2002.

[LP99] J. Lilius and I.P. Paltor. vUML: A Tool for Verifying UML Models. *14th IEEE International Conference on Automated Software Engineering*, pages 255–258, October 1999.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1–2):134–152, December 1997.

[Mik99] Tommi Mikkonen. The two Dimensions of an Architecture. In *WICSA1, First Working IFIP Conference on Software Architecture*, 1999.

[Obj03] Object Management Group. *MDA Guide Version 1.0.1*, omg/2003-06-01 edition, June 2003.

[Obj07] Object Management Group. Unified Modeling Language: Superstructure, version 2.1.1, February 2007. formal/2007-02-03.

[OGO04] Iulian Ober, Susanne Graf, and Ileana Ober. Validation of UML Models via a Mapping to Communicating Extended Timed Automata. In Susanne Graf and Laurent Mounier, editors, *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 127–145. Springer, 2004.

[Rus00] John Rushby. Disappearing Formal Methods. In *High-Assurance Systems Engineering Symposium*, pages 95–96, Albuquerque, NM, November 2000. ACM.

[SB08] Haldor Samset and Rolv Bræk. Describing Active Services for Publication and Discovery. In *Software Engineering Research, Management and Applications (Selected Papers)*, Studies in Computational Intelligence. Springer–Verlag, 2008. (to appear).

[SCKB05] Richard Sanders, Humberto N. Castejón, Frank Alexander Kraemer, and Rolv Bræk. Using UML 2.0 Collaborations for Compositional Service Specification. In *ACM / IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, 2005.

[Stø04] Alf Kristian Støyle. Service Engineering Environment for AMIGOS. Master's thesis, Norwegian University of Science and Technology, 2004.

[vdA98] W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[vdA99] W. M. P. van der Aalst. Woflan: a Petri-net-based workflow analyzer. *Syst. Anal. Model. Simul.*, 35(3):345–357, 1999.

[VM94] Björn Victor and Faron Moller. The Mobility Workbench — A Tool for the π-Calculus. In David Dill, editor, *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

[wop08] The WoPeD Homepage, May 2008.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA$^+$ Specifications. In L. Pierre and T. Kropf, editors, *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer-Verlag, 1999.

APPENDIX
B

REFINEMENT PROOF FOR A SYSTEM

We present the specification of an example and prove, that the behavior of the state machines synthesized by the transformation algorithm presented in Paper 3 is in fact an implementation of the specification given by its activity. Formally, this means that the behavior of the state machines implies that of the activity, which can be shown by a refinement proof. For that, we give a refinement mapping between the two behaviors which we also verify with the model checker TLC.

B.1 Introduction

Figure B.1 outlines our proceedings in the following. From an example activity we first derive its formal specification in temporal logic (in cTLA/c style) using the rules provided in Paper 6. We will then use the model transformation presented in Paper 3 (implemented by the Arctis Transformer) to produce the executable state machines for the example. We derive their semantics in temporal logic (in cTLA/e style) as defined in Paper 2. To prove, that the behavior of the synthesized state machines actually implements the behavior implied by the activity, we verify that the cTLA/e specification refines the cTLA/c specification.

As an example, we consider a collaboration which updates the clock of a

$$\begin{tikzcd}
\textbf{UML Activity} \arrow[r, "\textit{Paper 6 / Arctis Formulator}", "\textit{(Sect. A.2)}"'] \arrow[d, "\textit{Paper 3 / Arctis Transformer (Sect. A.3)}"'] & \textbf{cTLA/c} \\
\textbf{UML State Machines} \arrow[r, "\textit{Paper 2 (Sect. A.4)}"] & \textbf{cTLA/e} \arrow[u, Rightarrow, "\textit{Refinement Proof (Sect. A.5)}"']
\end{tikzcd}$$

Fig. B.1: Commuting diagram illustrating the proceedings in this appendix

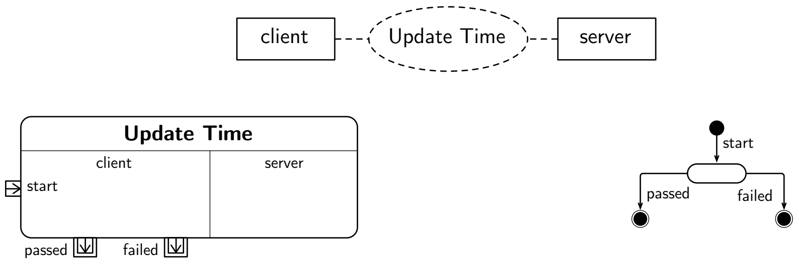


Fig. B.2: External view of the *Update Time* building block

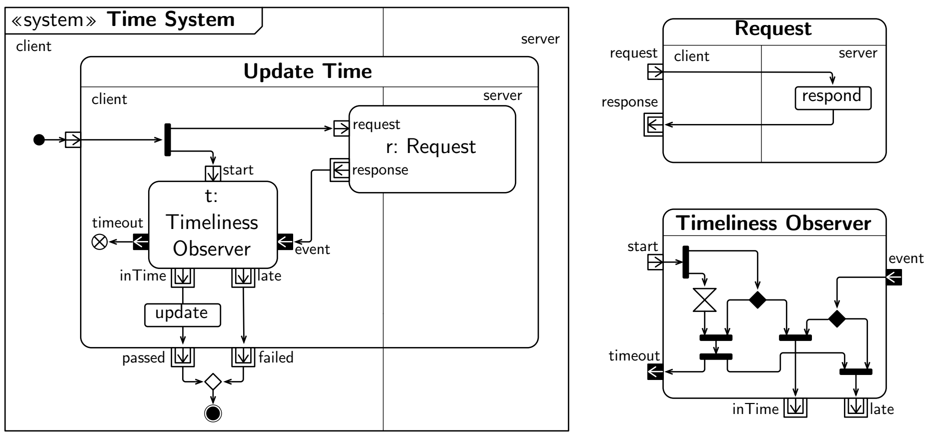


Fig. B.3: Activity for the time update in a minimal environment

client from a time server based on a method described by Cristian [Cri89]. In this method, the client simply sends a request to a time server, which responds with the current time. The idea is that roundtrip delays of the request and response between client and server are often so short that the clock of the client can be updated with sufficient accuracy. We consider here the simple case that we ask only a single time server and simply fail if the roundtrip delay exceeded a certain limit. The external view on the building block of this collaboration is presented in Fig. B.2 with its UML collaboration, external activity and ESM. We can see that the client starts the collaboration, which eventually terminates via output *passed* or *failed*. The internal view in Fig. B.3 reveals the details of the behavior, which is composed of a simple request and a timeliness observer already known from the trusted sale in Paper 5. When the time update starts, the client issues a request to the server. At the same time, the timeliness observer is started. Only if the response containing the time stamp is received before the timeliness observer expired, the time of the client is updated and the collaboration terminates via passed. Otherwise, it fails. As the system specification needs to be closed to be transformed into state machines, we have placed the building block in Fig. B.3

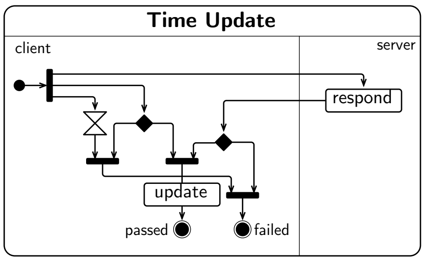


Fig. B.4: Flattened activity

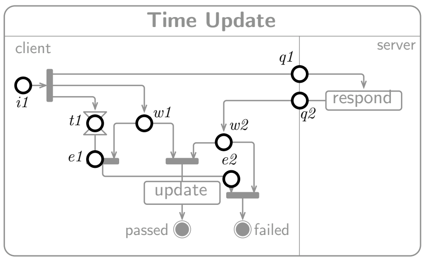


Fig. B.5: Illustration of variables

within a minimal environment.

Figure B.4 shows a flattened and simplified version of Fig. B.3. The call behavior actions r and t have been replaced by the activities they refer to, and *Update Time* of Fig. B.3 has been integrated with the enclosing *Time System*. We use this equivalent[1] activity instead, as it makes the discussion easier to understand.

B.2 Semantics of the Activities in cTLA/c

Using the rules given in Paper 6, it is straightforward to write down the formal semantics for the activity in the style of cTLA/c. As we will later use the model checker TLC to verify our refinement mapping, we write the specification directly in TLA$^+$. This is possible, as the original and compositional cTLA/c process can be transformed to a simple cTLA process that is equivalent to TLA$^+$ (see [Her97]). For this task, we can also use the Arctis Formulator (see [Slå07] and Paper 7), which automatically generates a TLA$^+$ module.[2]

Fig. B.5 illustrates the variables used in the formal specification for Fig. B.4 given in the TLA$^+$ module of Fig. B.6. The activity has the inner places *i1*, *e1*, *e2*, *w1*, *w2* for initial nodes, join nodes and waiting decision nodes, as well as *t1* for the timer. Variables *q1* and *q2* model the buffered communication medium in between. No variables are needed for the server partition, as it is stateless. Due to the focus of our work, we currently ignore the data part of the system, that means, the actual exchange of time stamps and operations on them. All variables are assumed to be integers and count the tokens resting in them. As specified by the initial condition *AInit*, all places are initially empty with exception of the initial node $i1$, which is set to 1. The behavior of the activity is described by the five actions *start*, *respond*, *timeout*, *passed* and *failed*.

start specifies the moving of the token from the initial node and its multiplication in the fork. As a result, the timer is started and the waiting decision node $w1$ as well as the queue $q1$ are filled with one token each.

[1]The equivalence between the cTLA/c formulas of composite and flattened activities was shown in Paper 6.

[2]In comparison to the specification presented here, the Arctis Formulator adds some variables to test additional theorems and uses rather generic names for variables and actions.

respond represents the reaction of the server to the request by moving the token through the call operation action *respond* to the queue place $q2$.

timeout models the expiration of the timer for the case that the response did not yet arrive, that means that waiting decision node $w1$ is still filled. As a result, the tokens are removed from $w1$ and $t1$ and a new token is placed into the inner place at $e2$.

passed represents the case that the response arrives before a timeout happens, that means that $w1$ still holds a token. This causes the activity to terminate, which also deactivates the timer. As we focus only on the reactive behavior and not on the data operations, the call operation *update* is not mentioned.[3]

failed represents the case that the response arrives after a timeout, which means that only the inner place *e2* holds a token, which causes the activity to terminate.

As the rules of Paper 6 only consider the syntax of the activity, they also produce cTLA resp. TLA actions handling situations that will never occur. Action *arrive* handles the arrival of a token from $q2$ when $w1$ and $e2$ are empty. Due to the overall behavior, such a state is not reachable. Action *timeout*2 describes the behavior of a timeout in case $w1$ is empty. This will never happen, as $w1$ could have only be emptied by an arriving response, which, however, terminates the timer. Action *timeout*3 handles the case when both waiting decisions are filled, which, due to the firing rules, is not possible as they would have caused the collaboration to terminate via *passed*. In addition, a token streaming from $i1$ into $w1$ could arrive for the case that $w2$ or $e1$ contain tokens. For this to happen, $i1$ needs to have a token. As all actions besides *start* do not change $i1$, it's easy to see that this is only the case for the initial state, where $w2 = 0 \wedge e1 = 0$ holds. (We did not include these last actions to make the specification easier to overlook.)

The behavior of the system in Fig. B.4 is hence described by formula *ASpec* in Fig. B.6. A run with TLC reveals that this specification has six distinct states.

B.3 Synthesized State Machines

When we use the activity of Fig. B.4 as input for our transformation algorithm, it produces the two state machines shown in Fig. B.7. In comparison to the original output, we have just changed the original (generic) names of the control states and signals by more meaningful ones for readability. The state machine does not include unreachable transitions as we used the enhanced algorithm described in Sect. 4.4.3 which only considers reachable states.

[3]Paper 6 demonstrates how data can be included into cTLA/c-based specifications as well.

MODULE *ATimeUpdate*

EXTENDS *Naturals*

VARIABLES $i1,\ t1,\ q1,\ q2,\ e1,\ e2,\ w1,\ w2$

$$\begin{aligned}
AInit \triangleq\ & \land\ i1 = 1 \land t1 = 0 \land q1 = 0 \land q2 = 0 \\
& \land\ e1 = 0 \land e2 = 0 \land w1 = 0 \land w2 = 0
\end{aligned}$$

$$\begin{aligned}
start \triangleq\ & \land\ i1 = 1 \land i1' = 0 \land t1 = 0 \land t1' = 1 \\
& \land\ q1' = q1 + 1 \\
& \land\ w1 = 0 \land w1' = 1 \\
& \land\ \text{UNCHANGED}\ \langle q2,\ e1,\ e2,\ w2 \rangle
\end{aligned}$$

$$\begin{aligned}
respond \triangleq\ & \land\ q1 = 1 \land q1' = q1 - 1 \\
& \land\ q2' = q2 + 1 \\
& \land\ \text{UNCHANGED}\ \langle i1,\ t1,\ e1,\ e2,\ w1,\ w2 \rangle
\end{aligned}$$

$$\begin{aligned}
timeout \triangleq\ & \land\ t1 = 1 \land t1' = 0 \\
& \land\ e2 = 0 \land e2' = 1 \\
& \land\ w1 = 1 \land w1' = 0 \\
& \land\ \text{UNCHANGED}\ \langle i1,\ q1,\ q2,\ e1,\ w2 \rangle
\end{aligned}$$

$$\begin{aligned}
passed \triangleq\ & \land\ q2 > 0 \land q2' = q2 - 1 \\
& \land\ w1 = 1 \land w1' = 0 \\
& \land\ t1' = 0 \\
& \land\ \text{UNCHANGED}\ \langle i1,\ q1,\ e1,\ e2,\ w2 \rangle
\end{aligned}$$

$$\begin{aligned}
failed \triangleq\ & \land\ q2 > 0 \land q2' = q2 - 1 \\
& \land\ e2 = 1 \land e2' = 0 \\
& \land\ t1' = 0 \\
& \land\ \text{UNCHANGED}\ \langle i1,\ q1,\ e1,\ w1,\ w2 \rangle
\end{aligned}$$

$$\begin{aligned}
arrive \triangleq\ & \land\ q2 > 0 \land q2' = q2 - 1 \qquad \text{(never executed)} \\
& \land\ w1 = 0 \land e2 = 0 \land w2 = 0 \land w2' = 1 \\
& \land\ \text{UNCHANGED}\ \langle i1,\ t1,\ q1,\ e1,\ e2,\ w1 \rangle
\end{aligned}$$

$$\begin{aligned}
timeout2 \triangleq\ & \land\ t1 = 1 \land t1' = 0 \qquad \text{(never executed)} \\
& \land\ w1 = 0 \land e1 = 0 \land e1' = 1 \\
& \land\ \text{UNCHANGED}\ \langle i1,\ q1,\ q2,\ e2,\ w1,\ w2 \rangle
\end{aligned}$$

$$\begin{aligned}
timeout3 \triangleq\ & \land\ t1 = 1 \land t1' = 0 \qquad \text{(never executed)} \\
& \land\ w1 = 1 \land w1' = 0 \land w2 = 1 \land w2' = 0 \\
& \land\ \text{UNCHANGED}\ \langle i1,\ q1,\ q2,\ e1,\ e2 \rangle
\end{aligned}$$

$$ANext \triangleq start \lor respond \lor timeout \lor passed \lor failed \lor arrive \lor timeout2 \lor timeout3$$

$$ASpec \triangleq AInit \land \Box[ANext]_{\langle i1,\ t1,\ q1,\ q2,\ e1,\ e2,\ w1,\ w2 \rangle}$$

Fig. B.6: TLA$^+$ Module in cTLA/c style for the collaboration

B.4 Semantics of the State Machines in cTLA/e

Following the guidelines in Paper 2, we can write the semantics of the state machines in the style of cTLA/e. Again, as we will use TLC to check our refine-

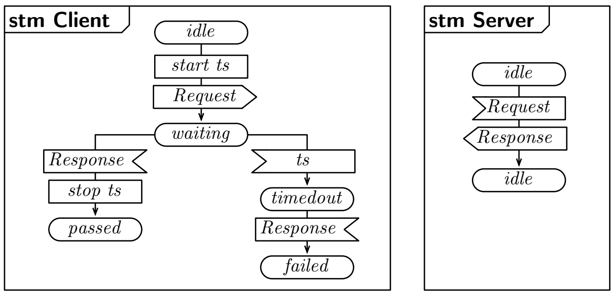


Fig. B.7: Result of the transformation

MODULE *StateMachines*

EXTENDS *Naturals*, *Sequences*

$$consume(signal, queue) \triangleq \begin{array}{l} \land Len(queue) > 0 \\ \land Head(queue) = signal \\ \land queue' = Tail(queue) \end{array}$$

$$consumeAndTerminate(signal, queue) \triangleq \begin{array}{l} \land Len(queue) > 0 \\ \land Head(queue) = signal \\ \land queue' = \langle\rangle \end{array}$$

$$transfer(source, target) \triangleq \begin{array}{l} \land Len(source) > 0 \land source' = Tail(source) \\ \land target' = Append(target, Head(source)) \end{array}$$

Fig. B.8: TLA$^+$ Module with auxiliary operators for state machines

ment mapping, we will use TLA$^+$ as language. We use module *StateMachines* in Fig. B.8 to define some useful operators. Operator *consume* checks a queue for its first signal. If it is identical to the signal provided, *comsume* removes it from the queue. Operator *consumeAndTerminate* works similar but empties the entire queue. It is used when a state machine performs a terminating transition. Operator *transfer* takes the head element of a source queue and moves it into the target queue.

Figure B.9 illustrates the variables used by the TLA$^+$ module *ETimeUpdate* in Fig. B.10 that describes their semantics. The control states are represented by variables *state1* and *state2*. Each state machine has one input queue (for signals and timeouts) and an output queue. The state of the timer is modeled by variable *ts*, which is set to 1 if the timer is active. As specified by *EInit*, all queues are initially empty and both control states are set to *idle*. Six actions model the behavior of the state machines.

start is initially enabled and represents the initial transition that is executed when the client starts. As a result, signal *Request* is sent via the output queue of the client machine and the timer is started. (A runtime support system as described in Paper 2 holds a list of active timers of all state machines it schedules. We represent this by $ts = 1$.) The client state machine changes its control state from *idle* to *waiting.*

expire models the expiration of the timer, in which the runtime support system places an expired timer into the input queue of the state machine.

timeout models the consumption of the timeout message by the state machine. Upon that, the client state machine changes its state from *waiting* to *expired.*

failing represents the arrival of the response signal after a timeout, causing a transition into state *failed.* As this is a terminating transition, the local input queues are emptied completely.

passing represents the consumption of the response signal before a timeout, causing a terminal transition into state *passed.*

respond models the only action of the server machine which simply consumes a request message from the input queue of the server and places a response in its output queue. (As with the activity, we do not model the actual retrieval of a time stamp and its inclusion in the signal.)

In addition to these actions that correspond to the transition and timer mechanisms of the state machines, two actions *transAB* and *transBA* model the transmission of signals between the output and input queues of the client and server, one for each direction.

A run with TLC shows that this specification has 22 distinct states. This is more than for the activity, due to the fact that communication between the state machines is buffered twice, in an output and input queue, and that the expiration of a timer and its consumption are modeled as two distinct steps.

B.5 Refinement Proof cTLA/e ⇒ cTLA/c

As described in Paper 3, to prove that the state machines implement the behavior described by the activities, we must prove that the specification *ESpec* refines

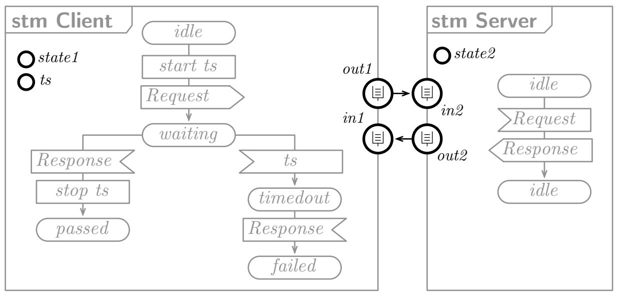


Fig. B.9: Illustration of variables for timer, control states and queues

```
┌──────────────────── MODULE ETimeUpdate ────────────────────┐
EXTENDS Naturals, Sequences, StateMachines

VARIABLES in1, out1, state1, ts, in2, out2, state2
├─────────────────────────────────────────────────────────────┤
EInit ≜ ∧ in1 = ⟨⟩ ∧ out1 = ⟨⟩ ∧ state1 = "idle" ∧ ts = 0
        ∧ in2 = ⟨⟩ ∧ out2 = ⟨⟩ ∧ state2 = "idle"

start ≜ ∧ state1 = "idle" ∧ state1′ = "waiting"
        ∧ ts′ = 1
        ∧ out1′ = Append(out1, "Request")
        ∧ UNCHANGED ⟨in1, in2, out2, state2⟩

expire ≜ ∧ ts = 1 ∧ ts′ = 0
         ∧ in1′ = Append(in1, "ts")
         ∧ UNCHANGED ⟨out1, state1, in2, out2, state2⟩

timeout ≜ ∧ state1 = "waiting" ∧ state1′ = "expired"
          ∧ consume("ts", in1)
          ∧ UNCHANGED ⟨in2, out1, out2, state2, ts⟩

failing ≜ ∧ state1 = "expired" ∧ state1′ = "failed"
          ∧ consumeAndTerminate("Response", in1)
          ∧ ts′ = 0
          ∧ UNCHANGED ⟨in2, out1, out2, state2⟩

passing ≜ ∧ state1 = "waiting" ∧ state1′ = "passed"
          ∧ consumeAndTerminate("Response", in1)
          ∧ ts′ = 0
          ∧ UNCHANGED ⟨in2, out1, out2, state2⟩

respond ≜ ∧ consume("Request", in2)
          ∧ out2′ = Append(out2, "Response")
          ∧ UNCHANGED ⟨in1, out1, state1, state2, ts⟩

transAB ≜ ∧ transfer(out1, in2)
          ∧ UNCHANGED ⟨in1, out2, state1, state2, ts⟩

transBA ≜ ∧ transfer(out2, in1)
          ∧ UNCHANGED ⟨in2, out1, state1, state2, ts⟩
├─────────────────────────────────────────────────────────────┤
ENext ≜ ∨ start ∨ expire ∨ timeout ∨ failing
        ∨ passing ∨ respond ∨ transAB ∨ transBA

ESpec ≜ EInit ∧ □[ENext]_⟨in1, out1, state1, out1, out2, state2, ts⟩
└─────────────────────────────────────────────────────────────┘
```

Fig. B.10: TLA$^+$ Module in cTLA/e style for the state machines

ASpec, or that

$$ESpec \Rightarrow ASpec \tag{B.1}$$

holds. For this proof, we search for a refinement mapping [AL91], which expresses the variables of *ASpec* in terms of those in *ESpec*. Following the guidelines given in Paper 3, a refinement mapping can be found by considering how the transformation maps the mechanisms of the activity to those of the state machines.

The initial place of the activity holds one token when the system is not yet started, that means when the client state machine is in state *idle*. Therefore, $i1$ can be expressed by the function

$$\overline{i1} \triangleq \text{IF } state1 = \text{"idle"} \text{ THEN } 1 \text{ ELSE } 0. \tag{B.2}$$

In the activity, buffered communication between two partitions is modeled by one queue place. In the state machines, a signal has to pass through two queues. Therefore, the value of the queue place of an activity is the sum of signals in the corresponding queues of that signal type of the state machines (see Paper 3). We have defined the operator $count(signal, queue)$. It takes a string denoting the signal as well as a queue (a sequence of strings) as argument and returns the number of occurrences of the string in the sequence.[4] Using this operator, the value $q1$ can be replaced by function

$$\overline{q1} \triangleq count(\text{"Request"}, out1 \circ in2), \tag{B.3}$$

where $out1 \circ in2$ is the concatenation of both queues. The value for $q2$ maps accordingly to

$$\overline{q2} \triangleq count(\text{"Response"}, out2 \circ in1). \tag{B.4}$$

As mentioned previously, the expiration of a timer is represented as one single action in the activity, but as two actions within a state machine. In the first action, the timer expires and a corresponding signal is placed in the input queue (action *expire* of Fig. B.10). In the second action, the timer signal is actually consumed (action *timeout* of Fig. B.10). While the first action is mapped to a stuttering step in the activity, the second is mapped to the action *timeout* of of Fig. B.6. For the mapping, this implies that the value of $t1$ is 1 whenever ts is set or when there is a timer signal ts in the input queue. Therefore, $t1$ can be replaced by

$$\overline{t1} \triangleq count(\text{"ts"}, in1) + ts. \tag{B.5}$$

The inner places *w2* and *e2* both map to 0, as a token never rests in them. This can be easily ensured by an invariant proof. For the other inner places, we find a mapping to the control state of the state machine, so that

$$\overline{w1} \triangleq \text{IF } state1 = \text{"waiting"} \text{ THEN } 1 \text{ ELSE } 0 \tag{B.6}$$

$$\overline{e2} \triangleq \text{IF } state1 = \text{"expired"} \text{ THEN } 1 \text{ ELSE } 0. \tag{B.7}$$

As in [Lam02], we use $\overline{F}$ to equal every formula F in *ATimeUpdate* with the original variables replaced by the refinement mappings given above. In addition,

[4]Operator *count* is defined in Fig. B.11 explained later. It uses *SelectSeq* of the sequence module, detailed in [Lam02].

subscripts $_e$ and $_a$ mark variables and actions to belong to either *ETimeUpdate* or *ATimeUpdate*. As *ESpec* and *ASpec* define their own variables, we have to prove

$$ESpec \Rightarrow \overline{ASpec}. \tag{B.8}$$

With the definitions for *ESpec* and *ASpec* this is

$$EInit \land \Box[ENext]_{var_e} \Rightarrow \overline{AInit} \land \Box[\overline{ANext}]_{\overline{var_a}}, \tag{B.9}$$

where var_e is the sequence of all variables used in *ESpec* and var_a the sequence of all variables in *ASpec*. To prove this, we have to prove two implications:

$$EInit \Rightarrow \overline{AInit} \tag{B.10}$$

$$\Box[ENext]_{vars_e} \Rightarrow \Box[\overline{ANext}]_{\overline{vars_a}} \tag{B.11}$$

To prove B.10, we replace $\overline{AInit}$ by its definitions

$$\begin{aligned} EInit \Rightarrow\ & \land \text{IF } state1 = \text{“idle”} \text{ THEN } 1 \text{ ELSE } 0 = 1 \\ & \land count(\text{“ts”}, in1) + ts = 0 \\ & \land count(\text{“Request”}, out1 \circ in2) = 0 \\ & \land count(\text{“Response”}, out2 \circ in1) = 0 \\ & \land (\text{IF } state1 = \text{“waiting”} \text{ THEN } 1 \text{ ELSE } 0) = 0 \\ & \land (\text{IF } state1 = \text{“expired”} \text{ THEN } 1 \text{ ELSE } 0) = 0 \\ & \land 0 = 0 \\ & \land 0 = 0 \end{aligned} \tag{B.12}$$

From the antecedent *EInit* we know that $state1 = \text{idle}$, $ts = 0$, and the queues $in1$, $out1$, $in2$, $out2$ are empty. Put into the right side, we get

$$\begin{aligned} EInit \Rightarrow\ & \land 1 = 1 \\ & \land count(\text{“ts”}, <>) + 0 = 0 \\ & \land count(\text{“Request”}, <>) = 0 \\ & \land count(\text{“Response”}, <>) = 0 \\ & \land 0 = 0 \\ & \land 0 = 0 \\ & \land 0 = 0 \\ & \land 0 = 0. \end{aligned} \tag{B.13}$$

With the definition of *count*, we see that the right side is true, and B.10 is proven. To show that B.11, we follow rule TLA2 from [Lam94] and prove

$$[ENext]_{vars_e} \Rightarrow [\overline{ANext}]_{vars_a}. \tag{B.14}$$

ENext is a disjunction of actions, so that we have to show that each action in *ENext* implies *ANext*. As *ANext* is a disjunction of actions as well, this means that each action in *ENext* must imply an action in *ANext* or a stuttering step. Fortunately, we do know which actions map to each other by considering what the transformation described in Paper 3 does. In particular, with $\overline{stutter_a}$ = UNCHANGED $\overline{vars_a}$

$$\begin{aligned} start_e &\Rightarrow \overline{start_a}, & expire_e &\Rightarrow \overline{stutter_a}, \\ timeout_e &\Rightarrow \overline{timeout_a}, & failing_e &\Rightarrow \overline{failed_a}, \\ passing_e &\Rightarrow \overline{passed_a}, & respond_e &\Rightarrow \overline{respond_a}, \\ transAB_e &\Rightarrow \overline{stutter_a}, & transBA_e &\Rightarrow \overline{stutter_a}. \end{aligned} \tag{B.15}$$

We begin with $start_e \Rightarrow \overline{start_a}$. With the mapping (B.2 to B.7) put into $start_a$, we get

$$\begin{aligned} start_e \Rightarrow &\wedge state1 = \text{“idle”} \ \wedge \ state1' \neq \text{“idle”} \\ &\wedge (0 = count(\text{“ts”}, in1) + ts) \ \wedge \ (1 = count(\text{“ts”}, in1') + ts') \\ &\wedge count(\text{“Request”}, out1' \circ in2') = count(\text{“Request”}, out1 \circ in2) + 1 \\ &\wedge state1 \neq \text{“waiting”} \ \wedge \ state1 = \text{“waiting”}. \end{aligned}$$

From the antecedent $start_e$ we know that $state1$ = “idle” and $state1$’=“waiting”, so that the first and last lines are true, and it remains to prove

$$\begin{aligned} start_e \Rightarrow &\wedge (0 = count(\text{“ts”}, in1) + ts) \ \wedge \ (1 = count(\text{“ts”}, in1') + ts') \\ &\wedge count(\text{“Request”}, out1' \circ in2') = count(\text{“Request”}, out1 \circ in2) + 1. \end{aligned}$$

The second line requires that the number of “Request” tokens in $out1$ and $in2$ is increased by 1, which is exactly what $start_e$ does with $out1' = append(out1, \text{“Request”})$, where a new signal is added to $out1$, while $in2$’=$in2$. This means we only have to show that

$$start_e \Rightarrow (0 = count(\text{“ts”}, in1) + ts) \ \wedge \ 1 = count(\text{“ts”}, in1') + ts' \tag{B.16}$$

From the antecedent $start_e$ we take ts’ = 1 and get

$$start_e \Rightarrow (0 = count(\text{“ts”}, in1) + ts) \ \wedge \ (1 = count(\text{“ts”}, in1') + 1) \tag{B.17}$$

As $in1$’ = $in1$, this requires that $ts = 0$. However, $start_e$ does not say anything about ts. Therefore, for $start_e \Rightarrow \overline{start_a}$ to be true, we must ensure that whenever $start_e$ is enabled, $ts = 0$ holds. We can do this by defining invariant T such that

$$T \triangleq state1 = \text{“idle”} \Rightarrow ts = 0. \tag{B.18}$$

With rule INV1 from from [Lam94], this proof can be divided into

$$EInit \Rightarrow T \tag{B.19}$$

$$T \ \wedge \ [ENext]_{vars_e} \Rightarrow T' \tag{B.20}$$

As the initial statement *EInit* specifies *ts*= 0, B.19 holds trivially. For B.20, we must consider each action in *ENext*. Actions *start*, *timeout*, *failing* and *passing* declare *state*1' $\neq$ "idle", fulfilling B.20 as the antecedent of T is false. Actions *respond*, *transAB* and *transBA* stutter for both *state*1 and *ts*, which in turn fulfills B.20. Action *expire* sets *ts* to 0, therefore fulfilling the right side of T. With that, B.19 holds for this action as well and we have proven $start_e \Rightarrow \overline{start_a}$. We continue with the proof of $expire_e \Rightarrow \overline{stutter_a}$.

$$
\begin{aligned}
expire_e \Rightarrow \land\ & (\text{IF } state1 = \text{“idle”} \text{ THEN } 1 \text{ ELSE } 0) \\
&\qquad = (\text{IF} state1' = \text{“idle”} \text{ THEN } 1 \text{ ELSE } 0) \\
\land\ & count(ts, in1) + ts = count(ts, in1') + ts' \\
\land\ & count(\text{"Request"}, out1 \circ in2) = count(\text{"Request"}, out1' \circ in2') \\
\land\ & (\text{IF } state1 = \text{“waiting”} \text{ THEN } 1 \text{ ELSE } 0) \\
&\quad = (\text{IF } state1' = \text{“waiting”} \text{ THEN } 1 \text{ ELSE } 0) \\
\land\ & (\text{IF } state1 = \text{“expired”} \text{ THEN } 1 \text{ ELSE } 0) \\
&\quad = (\text{IF } state1' = \text{“expired”} \text{ THEN } 1 \text{ ELSE } 0) \\
\land\ & 0 = 0 \\
\land\ & 0 = 0
\end{aligned}
\tag{B.21}
$$

Action $expire_e$ guarantees *state*1 = *state*1', making all lines with IF-statements true. As *out*1 = *out*1' and *in*2 = *in*2', *count*("Request", *out*1 $\circ$ *in*2) = *count*("Request", *out*1' $\circ$ *in*2') holds as well, so that remains:

$$expire_e \Rightarrow count(\text{“ts”}, in1) + ts = count(\text{“ts”}, in1') + ts' \tag{B.22}$$

With *ts* = 1 and *ts*' = 0 from $expire_e$, we get

$$expire_e \Rightarrow count(\text{“ts”}, in1) + 1 = count(\text{“ts”}, in1') \tag{B.23}$$

This requires that in queue *in*1' must be one "ts" token more than in queue *in*1. This is exactly what $expire_e$ guarantees with $in1' = append(in1, \text{“ts”})$, so that also this implication is true.

The proofs for the remaining actions of B.15 are similar and not carried out explicitly. Instead of proving them manually, we can also use TLC to check the refinement mapping. For this, we declare module *Refinement* in Fig. B.11. It extends the *ETimeUpdate* module, so that the original specification of the state machines *ESpec* and all variables of the state machines are present. *Abstract* creates an instance of the module *ATimeUpdate*. This instantiation replaces the original variables of *ATimeUpdate* with functions in terms of the state machine variables as defined previously. To check the refinement property, we declare the property *Refinement* and let TLC check it. A run reveals that no errors are found, which means that the refinement mapping is correct and the state machines imply the behavior of the activity. ■

```
──────────────── MODULE Refinement ────────────────
EXTENDS ETimeUpdate

count(signal, queue) ≜ LET match(n) ≜ n = signal
                       IN  Len(SelectSeq(queue, match))

Abstract ≜ INSTANCE ATimeUpdate WITH
                 i1 ← IF state1 = "idle" THEN 1 ELSE 0,
                 t1 ← count("ts", in1) + ts,
                 q1 ← count("Request", out1 ∘ in2),
                 q2 ← count("Response", out2 ∘ in1),
                 w1 ← IF state1 = "waiting" THEN 1 ELSE 0,
                 e2 ← IF state1 = "expired" THEN 1 ELSE 0,
                 w2 ← 0,
                 e1 ← 0

Refinement ≜ Abstract!ASpec
───────────────────────────────────────────────────
```

Fig. B.11: TLA$^+$ Module for the refinement mapping

Bibliography

[AL91] Martín Abadi and Leslie Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[Cri89] Flaviu Cristian. Probabilistic Clock Synchronization. *Distributed Computing*, 3:146–58, 1989.

[Her97] Peter Herrmann. *Problemnaher korrektheitssichernder Entwurf von Hochleistungsprotokollen.* PhD thesis, Universität Dortmund, 1997.

[Lam94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam02] Leslie Lamport. *Specifying Systems.* Addison-Wesley, 2002.

[Slå07] Vidar Slåtten. Model Checking Collaborative Service Specifications in TLA with TLC. Project Thesis, August 2007. Norwegian University of Science and Technology, Trondheim, Norway.

APPENDIX

C

UML PROFILE FOR COLLABORATIVE SERVICE SPECIFICATIONS

C.1 Introduction

In the following, we describe a UML profile for collaborative service specifications based on UML collaborations and activities. The discussion in the following is rather technical. For examples we refer to the papers of Part II. The most relevant chapters for this profile in the current UML standard [Obj07] are Chapt. 9 for Composite Structures (including UML collaborations), Chapt. 12 for Activities, Chapt. 15 for State Machines as well as Chapt. 11 and 13 for Actions and Common Behaviors.

The remainder of this section explains the semantics of token flows based on activity steps, and points out the difference between the definition of an activity and its instantiation as building block. Sect. C.2 continues with a discussion of the different kind of building blocks and how they are modeled in UML. Constraints on UML collaborations are listed in Sect. C.3. Sect. C.4 continues with the details of the state machines for the external behavior, and Sect. C.5 presents the constraints on UML activities to document the internal behavior of building blocks.

C.1.1 Token Flows in Activities

The semantics of activities are based on token flows, as explained by the UML standard [Obj07]. We specialize these semantics such that tokens move in run-to-completion steps that correspond to state machine transitions, and therefore only rest in certain places, and only if they have to wait for another event to happen. Reason for this is to ensure a proper compositional semantics based on cTLA as explained in Paper 6, and to enable a correct transformation of activities to state machines as described in Paper 3. For instance, tokens do not wait in call operation actions, as they are executed within a state machine

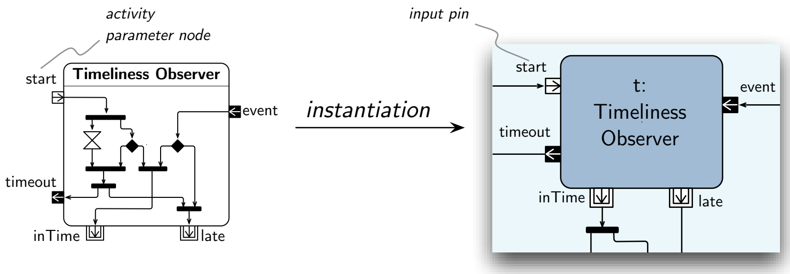


Fig. C.1: Activity and a call behavior action referring to it

transition. When a token arrives at a join node via an edge, it only waits if the other incoming edges of the join cannot offer a token yet; otherwise, the join fires and the token continues without resting. Similarly, token rest within waiting decision nodes (explained below), and accept event actions to wait for another event. Since we model the asynchronous communication between partitions, tokens wait at virtual queue places, as well.

A token movement from one stable state in which tokens wait for events to the next stable state is in the following called an *activity step*. An activity step spans over a a subgraph of an activity, consisting of edges and nodes which are traversed by tokens within one step. The same activity nodes or edges may be part of several activity steps. A decision node with two outgoing branches, for example, can be understood as two different activity steps, one for each alternative. Similarly, when a tokens enters a join node, there are different activity steps, one for each possible configuration that influence if the join can fire or not.

C.1.2 Activities and Call Behavior Actions

To clarify the difference between an activity and its instantiated form as call behavior action, we refer to Fig. C.1. On its left hand side, it shows the activity of the Timeliness Observer, known from Paper 5. It is a complete activity with internal activity nodes and edges. On its frame, there are activity parameter nodes that connect it to other elements once the activity is instantiated as building block, as shown on the right hand side. This UML element is a *call behavior action*, which only refers to the activity. The instance is named *t*. The activity parameter nodes from the activity are represented on *t* by corresponding input and output *pins*. These elements have the same symbol as the activity parameter nodes, but are owned by the call behavior action *t*.

C.2 Building Blocks

The most general building block is a *service collaboration*. It has at least two participants and can be connected to other blocks by means of activity parameter

	Structure *(UML Collaboration)*	**Internal Behavior** *(UML Activity)*	**External Behavior** *(UML State Machine «esm»)*
Service Collaboration	★	★	★
System Collaboration	★	★	
Activity Block		★	★
Shallow Activity Block		☆ *(frame only)*	★

Fig. C.2: Different kinds of building blocks

nodes. It is specified by a UML 2.0 collaboration for the structure, an activity for the internal behavior and an external state machine (ESM) to document the externally visible behavior. Based on this general service collaboration, we define three more kinds of building blocks, namely *system collaborations*, *activity blocks* and *shallow activity blocks*. The different diagrams needed to describe a building block of a certain kind are listed in Fig. C.2.

C.2.1 Block Type: Service Collaborations

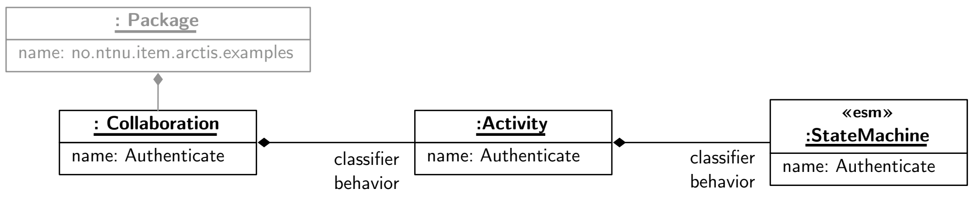


Fig. C.3: Repository model of a *service collaboration* building block

Fig. C.3 depicts a partial repository model (see [Boc03]) of a service collaboration to illustrate the connection between these elements. The container for a service collaboration is a UML collaboration, which owns an activity as classifier behavior. Similarly, the activity owns the ESM as a classifier behavior.

Constraints (Service Collaborations)

[1] A service collaboration is modeled as a UML collaboration, owned by a package.
[2] A service collaboration has an ESM and an internal behavior.
[3] A service collaboration has at least two participants (i.e., collaboration roles and activity partitions).

C.2.2 Block Type: System Collaborations

A *system collaboration* (Fig. C.4) is a service collaboration on the highest decomposition level, i.e., it models the complete system. Its collaboration roles represent the system components. Its behavior is *closed*, that means its activity does not have activity parameter nodes that could be connected to other elements. In consequence, a system collaboration does not have an ESM, and cannot be composed with other building blocks.

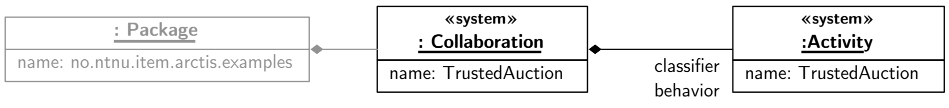


Fig. C.4: Repository model of a *system collaboration* building block

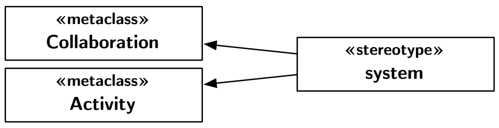


Fig. C.5: Stereotype ≪system≫

Constraints (System Collaborations)

[1] A system collaboration is modeled as a UML collaboration with stereotype ≪system≫ applied, owned by a package.
[2] A system collaboration does not have an ESM.
[3] The activity of a system collaboration is *closed*, i.e., does not have any activity parameter nodes.
[4] The activity of a system collaboration is marked as ≪system≫ as well.

C.2.3 Block Type: Activity Block

An *activity block* (Fig. C.6) is a service collaboration with exactly one participant. It has no UML collaboration, so that it is only represented by an activity with an ESM as classifier behavior.

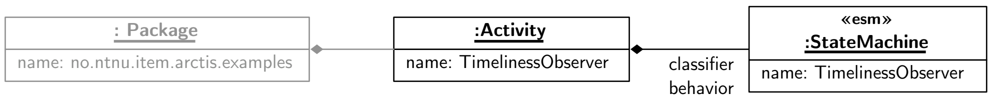


Fig. C.6: Repository model of an *activity block*

Constraints (Activity Blocks)

[1] An activity block is modeled by an activity owned by a package.
[2] An activity block has an ESM.
[3] An activity block has exactly one activity partition.

C.2.4 Block Type: Shallow Activity Block

In some cases, the external description of an activity block describes its entire behavior. This is the case for the Switch building block from App. D, for instance. Instead of giving a (redundant) internal behavior by an activity, we apply the stereotype ≪shallow≫ to the activity. A shallow activity block (Fig. C.7) only declares the activity parameter nodes, but is otherwise empty.

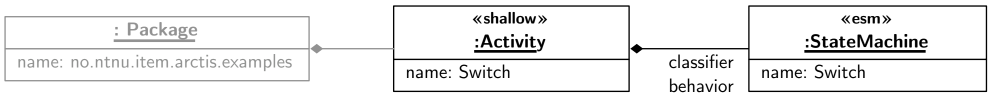


Fig. C.7: Repository model of a shallow *activity block*

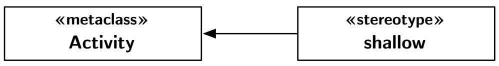


Fig. C.8: Stereotype ≪shallow≫

Constraints (Shallow Activity Blocks)

[1] A ≪shallow≫ activity is modeled by an activity with stereotype ≪shallow≫ applied and is owned by a package.
[2] A ≪shallow≫ activity only has activity parameter nodes and no edges.
[3] A ≪shallow≫ activity does not have any partitions.
[4] A ≪shallow≫ activity has an ESM.
[5] All transitions of the ESM of a shallow building block are triggered by the surrounding context.

C.3 Collaborations

As described above, the structural aspect of a building block (of kind *service* or *system collaboration*) is described by a UML collaboration. System collaborations are marked with the stereotype ≪system≫.

If a collaboration role represents an entity of the environment, it is marked by stereotype ≪environment≫. When a collaboration is instantiated as collaboration use within an enclosing collaboration, then its environment roles may only be assigned to environment roles of the enclosing collaborations.

The references of collaboration uses to other collaborations create a system specification on several levels, as illustrated in Fig. 4.2 in Part I. The graph structure of a system implied by these references must be a tree, i.e., none of the directly or indirectly contained collaboration uses may refer back to the collaboration and introduce cycles.

Collaboration uses can be bound to collaboration roles that have a multiplicity with an upper value higher than ‘1’. By default, we assume that there is executing at most one instance of a collaboration for each different set of participating collaboration roles. If, however, between an identical set of participant instances there may be several instances of a collaboration going on, the corresponding collaboration use may be tagged with ≪multi-session≫.

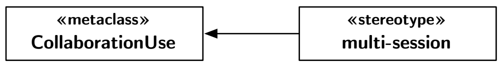


Fig. C.9: Stereotype ≪multi-session≫

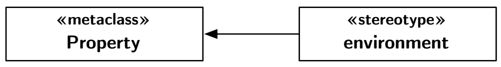


Fig. C.10: Stereotype ≪environment≫

Constraints (Collaborations)

[1] All collaboration roles of collaboration uses are bound.
[2] Collaboration roles marked as environment roles can only be bound to other environment roles.
[3] All collaboration uses of a service collaboration refer to other service collaborations, but not systems.
[4] The graph implied by the references expressed by collaboration uses is a tree.

C.4 External State Machines (ESMs)

ESMs describe the sequence in which tokens may pass the parameter nodes of an activity. Each transition of an ESM specifies the activity parameter nodes that are passed by tokens within a certain activity step.

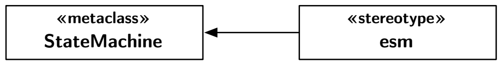


Fig. C.11: Stereotype ≪esm≫

Constraints (External State Machines)

[1] An ESM has the stereotype ≪esm≫ applied.

[2] An activity that has activity parameter nodes has an ESM as classifier behavior.
[3] An ESM has the same name as the activity it describes.
[4] An ESM contains exactly one region.
[5] An ESM region contains exactly one initial pseudostate.
[6] The only vertices in an ESM region are initial pseudostates, simple states and final states.
[7] States within an ESM do not contain entry or exit actions.

For the composition it is important to know if a certain ESM transition is triggered by the surrounding context, or if the ESM transition is spontaneous, i.e., initiated by an event within the building block. An ESM transition therefore refers to activity parameter nodes as *triggers* or *effects*. To assign activity parameter nodes accordingly, we use the stereotype ≪event≫ on an ESM transition that refers with its properties *triggers* and *effects* to sets of activity parameter nodes. (Why these are *sets* will be discussed further below.) The original UML attributes *effect* and *trigger* of standard transitions, referring to behaviors and signal or time events, are not used for this purpose and remain empty.

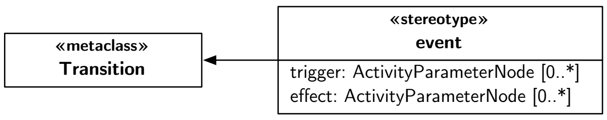
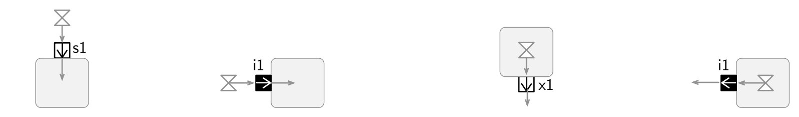

«metaclass»
Transition
«stereotype»
event
trigger: ActivityParameterNode [0..*]
effect: ActivityParameterNode [0..*]

Fig. C.12: Stereotype ≪event≫

Constraints (External State Machines, Transitions)

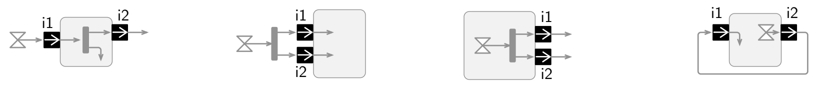
[1] Each transition of an ESM has the stereotype *event* applied.
[2] Transitions have no guard, effect or trigger.
[3] Each transition refers to one or several activity parameter nodes using the attributes *triggers* and *effects* of the stereotype *event* applied to the transition.

To visualize ESMs, transitions have labels which list the activity parameter nodes they refer to. The label on a transition consists of a trigger and an effect part, separated by a '/'. Figure C.13 shows the possible combinations when only one pin is involved in a step. The timer symbol represents any triggered flow arriving at a pin. Activity parameter nodes directly triggered by the enclosing context are listed as triggers. In case (a), the building block is started from the environment via pin *s1*. The label of the corresponding ESM transition is therefore in:s1/. The prefix 'in:' points out that *s1* is an input pin, so that the ESM is also understandable without the activity. Case (b) depicts the similar case as (a) but with a streaming pin. If the trigger is within the building block, the pin is named in the effect part. For case (c) the ESM transition is labeled



(a) in:s1/ (b) in:i1/ (c) /out:x1 (d) /out:i1

Fig. C.13: ESM transitions referring to one pin



(e) in:i1/out:i2 (f) in:i1+in:i2/ (g) /out:i1+out:i2 (h) /out:i2+in:i1

Fig. C.14: ESM transitions referring to several pins

/out:x1, meaning that the building block, due to an internal event, terminates via pin *x1*. In case (d), the environment must be prepared to accept a token via pin *i1* due to an internal event in the building block. In contrast to (c), the building block stays active as *i1* is an intermediate streaming node. Due to the selected syntax, a ESM transition that is triggered from within a block only declares effects, and therefore begins with a '/'.

In some cases, a token that enters a building block also exits the block within the same activity step. This is the case in Fig. C.14 (e), for example. The transition refers to *i1* as the trigger, while *i2* is listed as effect. In Fig. C.14 (f), two input pins are triggered within the same step. This makes sense if a building block encapsulates a number of operations, from which several may be executed in the same activity step, for example. In this case, bot *i1* and *i2* are listed as triggers. They are separated by a +. An ESM transition may also describe more than one effect, as exemplified in case Fig. C.14 (g), where tokens pass through *i1* and *i2* within the same step. This is useful in cases where a building block emits tokens of different data types within the same step. In case (h), a token first exits *i2* and then re-enters the building block via *i1*. This is useful in cases where an event triggered by a building block may lead to another effect on the block, for example to terminate a block.

Since steps are always executed within the same state machine, i.e., must be local, all parameter nodes referred by one ESM transition must be assigned to the same partition. Furthermore, to simplify the implementation of tools, we assume that a starting node is not coupled with any other nodes.

Constraints (External State Machines, Transitions)

[1] All activity parameter nodes referred by one transition (as trigger or effect nodes) must be assigned to the same activity partition.

[2] If a transition refers to a starting node, it does not refer to other nodes.

The behavior expressed by the ESM must be consistent with that of the activity it describes. Formally, an activity refines its ESM. This means, for each activity step, there is a compatible ESM transition that describes the same token movements through activity parameter nodes as the activity step, or the activity step does not cross any activity parameter nodes. Slåtten describes theorems to test the correct relation between an activity and an ESM in [Slå07, Slå08]. An example of a corresponding analysis is given in Paper 7.

In some cases it may be desirable to give ESMs for each participant to document the behavior towards each side separately. This can be achieved using *local* ESMs. Like ESMs, they are modeled as UML state machines with the stereotype ≪esm≫ applied. To assign them to a participant, they are owned by the activity they describe and have the name of the activity partition they represent.

Constraints (Local, External State Machines)

[1] A local ESM is owned by the activity it describes and has the name of the partition it focuses on.

[2] A local ESM only refers to the activity parameter nodes assigned to the partition it represents.

C.5 Activities

In the current version, we do not support UML activities of the level *complete structured activities* or *extra structured activities* [Obj07]. Supported activity nodes are initial nodes, fork nodes, join nodes, decision nodes, merge nodes, activity final nodes, flow final nodes, call operation actions, call behavior actions, send signal actions, accept event actions with signal events (accept signal actions), accept event actions with time event (timers), activity parameter nodes, input pins and output pins.

We use a strict syntax concerning incoming and outgoing flows of actions. This means, for example, that actions have exactly one incoming and one outgoing flow. If an action does not have an outgoing flow, it is interpreted that any token emitted by the action is discarded, similar to a flow final node. If an action should be started by several alternative flows, this should be specified explicitly by a merge node. Similarly, if an action should start several flows, a fork node should be used. Furthermore, we assume that each element has a unique name.

C.5.1 Activity Partitions

Activity partitions model different places of computation. At execution time, each partition is assigned to a certain component. For service and system collaborations, each activity partition corresponds to a collaboration role of the UML collaboration of the building block. Since collaboration roles may have a multiplicity other than '1', the activity partition may show this multiplicity as well.

Constraints (Activity Partitions)

[1] If an activity is part of a system or service collaboration, it contains exactly one activity partition for each collaboration role of the building block's collaboration.
[2] Activity nodes that are not call behavior actions must be assigned to exactly one partition.
[3] Call behavior actions that represent an activity block must be assigned to exactly one activity partition.
[4] Call behavior actions representing service collaborations are assigned to activity partitions according to the role binding of their corresponding collaboration use.
[5] Pins owned by call behavior actions are assigned to an activity partition in a consistent manner to the role binding of the owning call behavior action and the partition binding of the activity parameter nodes of the referred activity.
[6] If the collaboration role corresponding to an activity partition is an ≪environment≫, the partition is marked as ≪external≫.

C.5.2 Initial Nodes

An initial node initially holds one token, which is released to start the system. To model the start of collaborations explicitly, only a system collaboration may have an initial node. In contrast, ervice collaborations and activity blocks are started using activity parameter nodes. We currently assume that at most one activity partition of a system activity has initial nodes. They release their tokens within the same activity step, which means that a set of initial nodes within one partition is equivalent to one initial node with a subsequent fork node.

Constraints (Initial Nodes)

[1] Only system activities may have initial nodes.
[2] A system activity has at least one initial node.

C.5.3 Activity Final Nodes

An activity final node terminates an ongoing activity. Similar to the constraints on initial nodes, activity final nodes must only be used on system level. To terminate a service collaboration or an activity block, use the extended semantics of terminating parameter nodes as described below.

Constraints (Activity Final Nodes)

[1] Only system activities may have activity final nodes.

C.5.4 Flow Final Nodes

Flow final nodes simply end a flow and consume any arriving token. In contrast to an activity final node, a flow final node does not terminate the activity.

Constraints (Flow Final Nodes)

[1] Flow final nodes have exactly one incoming flow and no outgoing edge.

C.5.5 Decision Nodes

Decision nodes divert an incoming token to one of the outgoing edges which has a guard that evaluates to *true*. Tokens do not rest in standard activity nodes, but directly leave via one of the enabled outgoing edges.

Constraints (Decision Nodes)

[1] A decision node has at least two outgoing edges.
[2] A decision node has exactly one incoming edge.
[3] All edges originating at a decision node have a guard.
[4] A decision node has exactly one outgoing edge with an *else* guard, or, if it has a boolean input, it has a pair of *true* and *false* branches.
[5] A decision node does not have a *decisionInput* behavior.

C.5.6 Waiting Decision Nodes

Waiting decision nodes are an extension to decision nodes. In contrast to standard decision nodes, tokens may rest in waiting decision nodes. To distinguish them graphically from normal decision nodes, they are presented as filled diamonds. Waiting decision nodes are used in combination with join nodes, to model that an event can cause different effects depending on what has happened before.

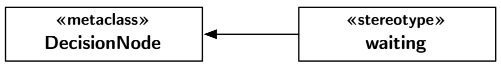


Fig. C.15: Stereotype ≪waiting≫

Constraints (Waiting Decision Nodes)

[1] A waiting decision node has at least two outgoing edges.
[2] A waiting decision node has exactly one incoming edge.
[3] All edges starting at a waiting decision have distinct join nodes as targets.
[4] Edges starting at a waiting node do not have a guard.

C.5.7 Fork Nodes

Fork nodes duplicate tokens, and the subsequent branches are executed within the same activity step. To keep the transformation of forks manageable, we currently claim some simplifications for the use of forks. See also the discussion in Paper 3.

Constraints (Fork Nodes)

[1] Fork nodes have exactly one incoming flow.
[2] Fork nodes have have at least two outgoing flows.
[3] Flows originating from the same fork may never be merged within the same activity step.
[4] Only one flow originating at a fork may have a decision within the same activity step.

C.5.8 Merge Nodes

Merge nodes simply forward any node from the incoming edge to the one outgoing edge. Tokens do not wait in merge nodes.

Constraints (Merge Nodes)

[1] Merge nodes have at least two incoming edges.
[2] Merge nodes have at exactly one outgoing edge.

C.5.9 Join Nodes

Join nodes synchronize several incoming flows and continue with the outgoing flows when all incoming flows can provide a token. Due to our semantics, tokens do only wait in a join node if not all incoming edges may provide a token. If a token runs into an edge that has token in all other incoming edges, the join directly fires within the same activity step that moved the last missing token. Paper 3 provides an illustration of these semantics.

Constraints (Join Nodes)

[1] Join nodes have at least two incoming edges.
[2] Join nodes have exactly one outgoing edge.

C.5.10 Call Operation Actions

Operations are represented in activities by call operation actions that refer to an operation, owned by the activity. The content of call operation actions is written by Java code linked to the UML model. This is, however, not discussed in this version of the profile.

Tokens do not rest in call operation actions, but leave the node within the same activity step they entered.

Constraints (Call Operation Actions)

[1] Call operation actions have exactly one incoming edge.

[2] Call operation actions have exactly one outgoing edge.

C.5.11 Variables

An activity can declare owned variables. Each variable must be assigned to an activity partition. Since variables are not activity nodes and cannot be assigned to a partition in standard UML, we use the stereotype ≪location≫ to assign an activity partition to a variable, using the stereotype attribute *partition*.

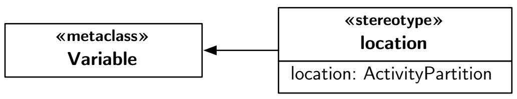


Fig. C.16: Stereotype ≪location≫

Constraints (Variables)

[1] A variable has ≪location≫ applied with the attribute *partition* set to one of the activity partitions that are owned by the activity declaring the variable.

[2] Variables may only be used by actions and guards of the same partition they are assigned to.

C.5.12 Send Signal Actions

Send signal actions are used to communicate with the environment by means of explicit signals. Send signal actions typically occur in two-way collaborations, where one participant represents the environment, so that the transformation algorithm knows where the signal should be sent. In other cases, it is assumed that the underlying platform routes the signals correctly, using addressing information within the signal.

We assume that tokens do not rest in send signal actions, but leave the node within the same activity step they entered.

Constraints (Send Signal Actions)

[1] Send signal actions have exactly one incoming edge.

[2] Send signal actions have exactly one outgoing edge.

C.5.13 Accept Signal Actions

Accept signal actions are accept event actions that refer to a signal event. In our specification style, the corresponding signal is sent by the environment. A token waits within an accept signal action until the the signal is received. UML allows accept event action to have no incoming flow, which means that they are constantly enabled whenever the surrounding activity is active. We claim, however, that accept signal actions have exactly one incoming flow, to make their activation explicit. To listen continuously to the environment or to stop waiting for a signal, we refer to the patterns and building blocks in App. D.

Constraints (Accept Signal Actions)

[1] Accept event actions have exactly one incoming edge.
[2] Accept event actions have exactly one outgoing edge.
[3] Accept event actions have exactly one trigger.
[4] The trigger of an accept event action modeling the reception of a signal is a signal event.
[5] A signal event refers to a signal.

C.5.14 Timers

Timers are modeled by accept event action that refer to a time event. We assume that time events always refer to a duration (given in milliseconds) which determines the duration that a token rests within the accept event action before it is released. Therefore, tokens wait in timers. To model periodic timers or timers that can be aborted, we refer to the patterns and building blocks given in App. D.

Constraints (Timers)

[1] Accept event actions have exactly one incoming edge.
[2] Accept event actions have exactly one outgoing edge.
[3] Accept event actions have exactly one trigger.
[4] The trigger of an accept event action representing a timer is a time event.
[5] A time event refers to a duration in milliseconds.

C.5.15 Activity Parameter Nodes

Activity parameter nodes are complemented by a parameter that they refer to. The parameter determines the direction (*in* or *out*).[1] The parameter also determines if the parameter node is a *streaming* node, that means if tokens may pass it while the activity is active.

Terminating nodes (non-streaming output parameter nodes) end an activity similarly to an activity final node, as illustrated by the equivalent graphs in

[1]Direction *inout* is currently not supported.

Fig. C.17. For compactness and clarity, the presentation on the right hand side is preferred, since activity final nodes should only be used on system level.

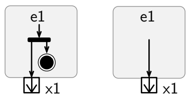


Fig. C.17: Equivalent notation for terminating pins.

If parameter nodes which start or terminate an activity are mutually exclusive (i.e., if a collaboration can be started or terminated by alternative output parameter nodes), their parameters refer to distinct parameter sets. This is represented by an additional box around the activity parameter node.

If the call behavior action is marked as ≪multi-session≫, then tokens do rest within an activity parameter node (resp. pin), if its activity partition is listed in the partitions of the multi-session stereotype on the call behavior action. This is due to the necessary buffering between state machines, since multiple instances are implemented by separate state machines. Tokens do not rest in activity parameter nodes (resp. pins) that are coupled synchronously to an activity, i.e., when there are no multiple sessions. For details of the semantics, see Paper 6.

Constraints (Activity Parameter Nodes)

[1] An activity parameter node points to a parameter that declares parameter direction and streaming.

[2] The parameter referred to by an activity parameter node has the same name and the same type as the activity parameter node.

[3] Activity parameter nodes are not assigned to an external partition.

[4] Activity parameter nodes referring to an input parameter have exactly one outgoing edge but no incoming one.

[5] Activity parameter nodes referring to an output parameter have exactly one incoming edge but no outgoing one.

C.5.16 Call Behavior Actions

Call behavior actions are used to compose behavior within activities. They represent instances of building blocks.

To mark that a collaboration is executed in several instances at the same time, the stereotype ≪multi-session≫ can be applied to a call behavior action, as described in Paper 4. Partitions that are, with respect to their own multiplicity, connected to their own multiplicity to more than one session instance of the collaboration, are included in the set attribute *partitions* of multiple activity partitions.

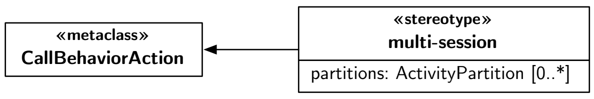


Fig. C.18: Stereotype ≪multi-session≫

Constraints (Call Behavior Actions)

[1] A call behavior action refers to an activity as behavior.
[2] A call behavior action with more than one partition corresponds to a collaboration use of the collaboration of the enclosing activity.
[3] A call behavior action with more than one partition is contained in exactly those partitions that correspond to the collaboration roles its corresponding collaboration use is bound to.
[4] A call behavior action owns pins for each activity parameter node of the activity it refers to.

C.5.17 Pins

As mentioned above and illustrated by Fig. C.1, pins owned by call behavior action correspond to activity parameter nodes of the activity the call behavior action refers to. This reference is done *by name*, i.e., for each parameter node of an activity there exists a pin owned by the call behavior action.

Constraints (Pins on Call Behavior Actions)

[1] If the pin refers to a activity parameter node with a parameter with direction *in*, it is an input pin.
[2] If the pin refers to a activity parameter node with a parameter with direction *out*, it is an output pin.
[3] The type of a pin corresponds to the type of the activity parameter node (and its parameter).

C.5.18 Activity Edges

A *transfer edge* is an edge that connects exactly two activity nodes which are assigned to different partitions. Edges crossing partitions imply communication between system components. Since communication is buffered, we assume a (virtual) queue place in the edge that may hold an unlimited number of tokens. This means that tokens do not move between partitions in one step, but rest in the queue place. In consequence, tokens sent between two partitions may overtake each other if they move along different edges.

For two partitions to communicate (i.e., have crossing edges), the collaboration roles corresponding to the activity partitions must be connected by a connector. Edges may not enter or leave external partitions, since communication with the environment is modeled by explicit signal transmissions, as described above.

Constraints (Activity Edges)

[1] Activity edges have source and target.
[2] Activity edges is contained exactly within the activity partitions of the source and target nodes.
[3] If an edge crosses partitions, the corresponding collaboration roles have to be connected.

When a flow enters a call behavior action from a partition that is marked as *multiple* for that call behavior action, the edge must have a **select** statement. It is represented by a stereotyped comment attached to an edge of the flow. The syntax for the selection statement is described in [KBH07].

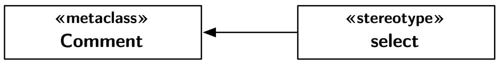


Fig. C.19: Stereotype ≪select≫

To reason about sessions, the **exists** operator can be used as a guard attached to an edge that succeeds a decision node. An **exists** guard is modeled as stereotyped string expression, with the content according to the syntax given in [KBH07].

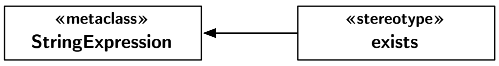


Fig. C.20: Stereotype ≪exists≫

C.6 Assertions

Developer may ammend activities with additional information to assert certain properties which will be checked during a behavioral analysis.

C.6.1 Occurrences of Action Executions

To each UML action, the developer may add the number of minimal and maximal occurrences of an action within the execution of the surrounding collaboration.

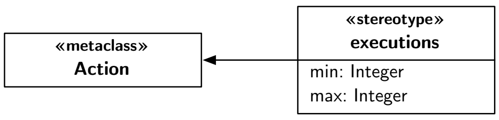


Fig. C.21: Stereotype ≪executions≫

C.6.2 Mutually Exclusive Actions

Often, within one collaboration execution, the occurrence of one action implies that certain other actions do not happen (i.e., these actions are mutually exclusive). Technically this is achieved by a stereotyped comment which can be added to an activity containing the actions. In this way, several sets of mutually exclusive actions may be specified.

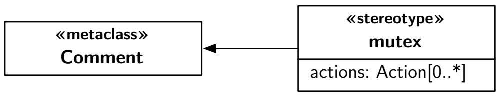


Fig. C.22: Stereotype ≪mutex≫

C.6.3 Upper Queue Bounds

Flows crossing partition borders imply queue places that can hold more than one token, representing a communication medium that buffers a number of signals. The developer can often estimate how many signals (resp. tokens) may be in the queue, and in most cases a queue must only hold one single token. So it is advised to give upper bounds, which are then checked during model checking. (If no upper bounds are given, the current version of the analysis chooses some default values as bounds, see [Slå07, Slå08].) Since only transfer edges have queues, the stereotype may only be applied to transfer edges.

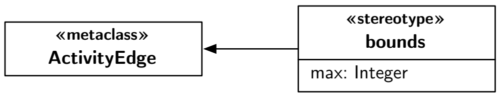


Fig. C.23: Stereotype ≪bounds≫

Bibliography

[Boc03] Conrad Bock. UML 2 activity and action models. *Journal of Object Technology*, 2(4):pp. 43–53, July-August 2003.

[KBH07] Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007*, volume 4745 of *Lecture Notes in Computer Science*, pages 166–185. Springer–Verlag Berlin Heidelberg, September 2007.

[Obj07] Object Management Group. Unified Modeling Language: Superstructure, version 2.1.2, November 2007. formal/2007-11-01.

[Slå07] Vidar Slåtten. Model Checking Collaborative Service Specifications in TLA with TLC. Project Thesis, August 2007. Norwegian University of Science and Technology, Trondheim, Norway.

[Slå08] Vidar Slåtten. Automatic Detection and Correction of Flaws in Service Specifications. Master's thesis, Norwegian University of Science and Technology, June 2008.

APPENDIX

D

BUILDING BLOCKS AND PATTERNS FOR UML COLLABORATIONS AND ACTIVITIES

This appendix provides building blocks (**B**) and patterns (**P**) which we have developed within our case studies. The syntax amd semantics follows the profile for collaborative service specifications presented in App. C. The patterns and building blocks are presented in four groups:

- **Local Flows** (Sect. D.1) helps to control tokens within one activity partition, for example to enforce a certain sequence between two flows, divert flows or simply to count iterations.

- **Time** (Sect. D.2) introduces several building blocks to measure time and schedule periodic events.

- **Environment** (Sect. D.3) lists several blocks and patterns to communicate with the environment using explicit signal transmissions.

- **Mixed Initiatives** (Sect. D.4) provides a building block to handle situations in which two sides of a collaboration may start initiatives that have to be coordinated.

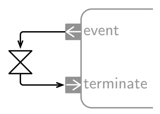
Some building blocks include a form of array, in which certain structures are repeated N times. We describe these building blocks for the case $N = 2$. The *First* building block in the following section, for instance, is described for exactly two incoming flows. It is currently left to the user to instantiate building blocks for other values of N. (See also the discussion in Chapt. 7 in Part II.)

Elements part of a pattern are presented in black, with an illustration of their context in grey. In some cases, we show building blocks in their instantiated form as call behavior actions with an additional (informal) depiction of their behavior.

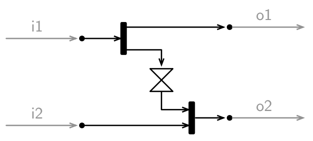
D.1 Local Flows

UML provides by default fork nodes, join nodes, decision nodes and merge nodes to coordinate several flows [Obj07]. We also provided waiting decision nodes to model more complex stateful behavior in combination with join nodes. Moreover, shallow activity blocks may be used to define more advanced nodes that coordinate several flows using ESMs as description. These extensions are described by the profile in App. C.

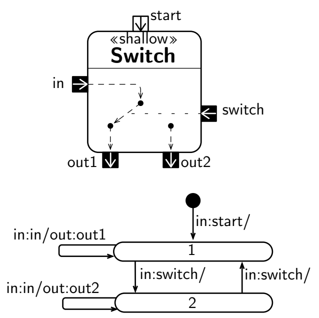
P Dividing an Activity Step In some cases, a token flow must be divided into separate activity steps. For instance, the building block shown to the right should be terminated upon the occurrence of *event*. Usually, the building block should declare an ESM transition /out:event+in:terminate so that a direct coupling of these pins is allowed. If that is not the case, and the building block cannot be changed, a timer can be added in between. This timer may be specified with a zero delay, so that the subsequent activity step is scheduled directly afterwards. (Since other behavior may interleave, it is, however, not guaranteed that the flow continues within the next execution step.)



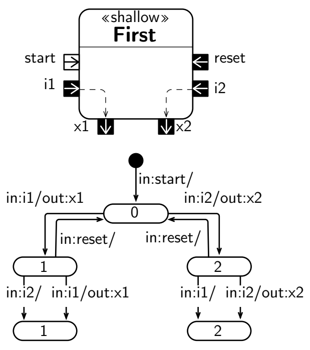
P Sequencer With the sequencer pattern one may ensure that the flow *i2* only continues after *i1*. To prevent that *o1* and *o2* continue within the same activity step, the additional timer with a zero delay separates an activity step started by the arrival of *i1*.



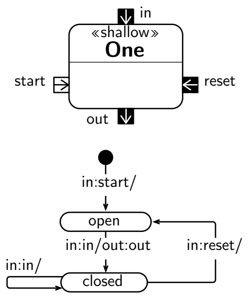
B Switch By default, a *Switch* forwards any token coming into *in* to *out1*. As soon as a token flows into pin *switch*, tokens from *in* are diverged towards *out1*, until the switch is toggled again, and so on.



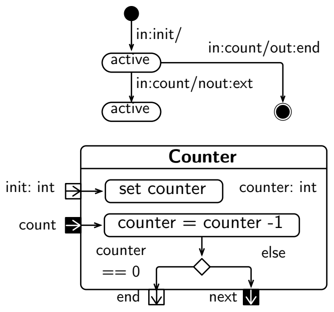
B **First** *First* lets only pass the tokens of the incoming flow that arrives first. Tokens arriving from the other flow are discarded.



B **Passing only a single token** The *One* building block only passes *one* single token through *in* towards *out*. All subsequent tokens arriving via *in* are stopped.



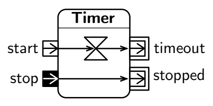
B **Counter** The counter block is initially set to a certain number and counts downwards with each token traveling through *count*. As long as the counter is not zero, the token is forwarded towards *next*. If the counter reached zero, it is diverged towards *end*. By ignoring tokens emitted from *next*, the counter can simply be used to count tokens and to trigger an event via *end* once a certain number of tokens have been received. By using pin *next*, the counter block can also be used to create a loop with a fixed number of iterations.

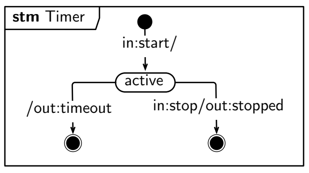


D.2 Time

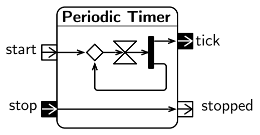
In standard UML, timers are accept event actions referring to a time event. In our profile, time events must specify a duration, not fixed points in time. Therefore, a simple timer works as an delaying element, that keeps an incoming token for a certain duration. Simple timers are only aborted if the surrounding activity is terminated. We provide therefore a timer block which can be aborted by an incoming token, as presented next.

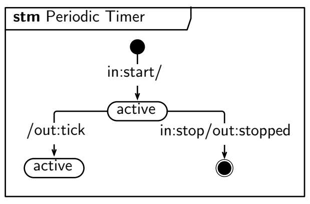
B **Timer** A timer block can be stopped to prevent a timeout event.



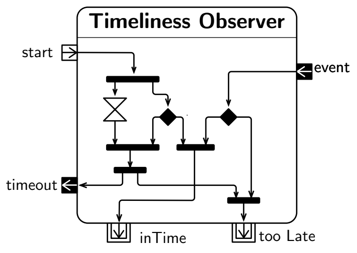


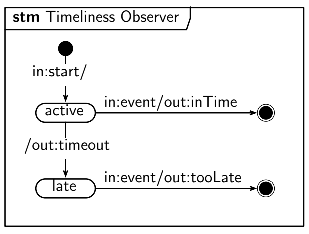
B **Periodic Timer** Since we only allow time event to specify time durations, periodic timers are provided by a special building block that activates the timer upon a timeout. Similar to the timer block, the periodic timer may be terminated at any time.





B **Timeliness Observer** The timeliness observer can be used to measure if a certain event happens within a certain time frame. After its start, the timeliness observer waits for the arrival of a token at *event*. If it arrives before the timeout, the token triggers the termination via *inTime*. Otherwise, a timeout is propagated via *timeout* and the eventually arriving event is diverged towards *tooLate*.

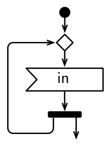




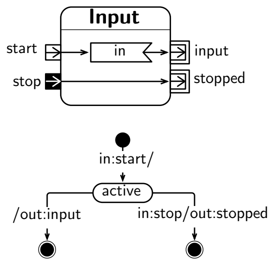
D.3 Environment

Communication with the environment is modeled using explicit signal transmissions, that means, send signal actions and accept event actions with a signal trigger (also called *accept signal actions*).

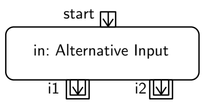
B **Continuous Input** Accept signal actions emit their token once the signal arrives and deactivate. To continuously receive signals, the accept signal action may be restarted through a fork node.

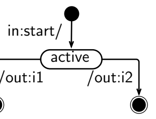


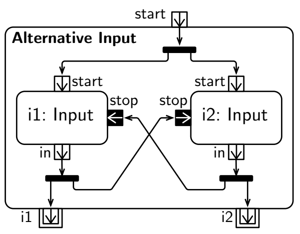
B **Controlled Input** The simple accept signal actions keep a token until the corresponding signal is received or the surrounding activity is terminated. To stop waiting for the arrival of an external signal, the Controlled Input building block can be used. It can be stopped similarly to the timer block from above. Depending on its implementation, the controlled input corresponds to a button of a user interface that can be enabled and disabled.



B **Alternative Input** In a situation where one of several inputs is acceptable but it should be ensured that only one is processed, an array of controlled inputs is used, provided by the Alternative Input building block.





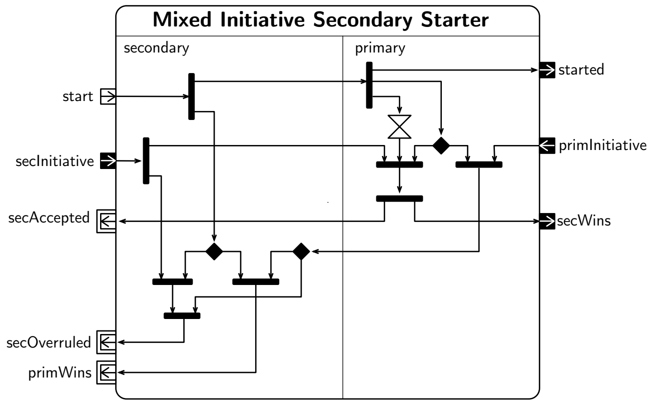


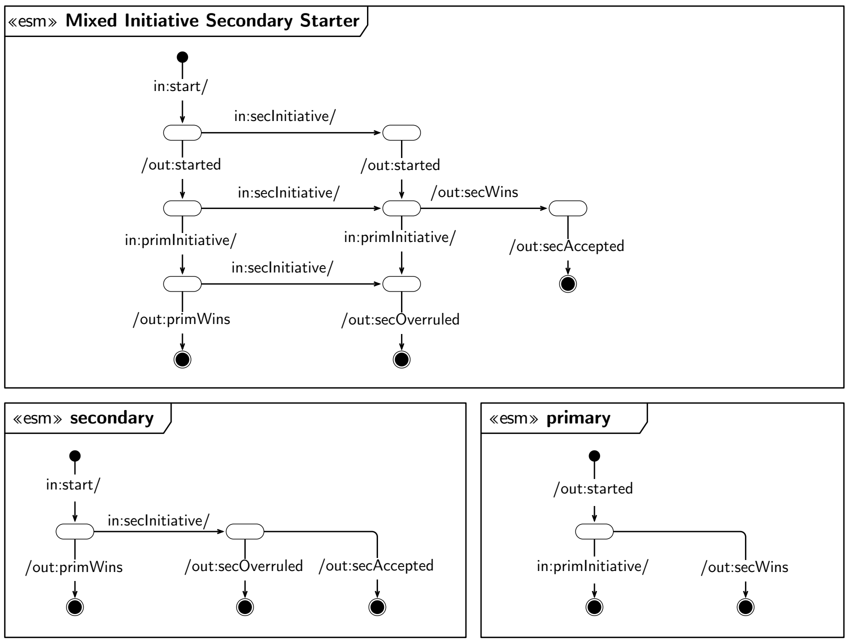
D.4 Mixed Initiatives

The *Mixed Initiative Secondary Starter* building block resolves situations in which to participants can simultaneously take initiatives that have to be coordinated. Examples are presented in Paper 7 and App. A.

B **Mixed Initiative Secondary Starter.** The two local ESMs for the primary and secondary side document the behavior of this block. The secondary side starts the block. After that, the secondary side may place its initiative or wait for the primary side to take action. If only the primary side takes initiative, the building block terminates via *primWins*. Otherwise, if only the secondary side took initiative, the block terminates via *secAccepted*. If, however, both sides take initiative, the block ends via *secOverruled*.

On the primary side, the behavior is simpler. Upon a token emitted from *started*, the primary side may decide to take initiative via *primInitiative* or wait until the secondary takes initiative via *secWins*.





Bibliography

[Obj07] Object Management Group. Unified Modeling Language: Superstructure, version 2.1.2, November 2007. formal/2007-11-01.

APPENDIX

E

UML PROFILE FOR EXECUTABLE STATE MACHINES

This chapter documents the profile for executable service models based on state machines and structured classifiers. It lists numerous constraints on UML models and complementary Java classes to facilitate code generation and ensure proper execution semantics.

E.1 Introduction

This profile uses UML state machines to specify the behavior of system components. Since UML does not provide a language to write down actions, we use additional Java classes that describe the detailed effect of state machine transitions on data. Therefore, this profile does not only address UML models, but also Java code. References between UML model and Java classes are *by name*, and we assume that Java classes are provided alongside the files for the UML model, for example within the same Eclipse project.

In addition to the connection of UML and Java, this profile has the purpose to facilitate code generation and the provision of tool support, clarify semantic variation points in the UML standard and ensure proper execution semantics. These semantics are described in detail in Paper 2 on the basis of cTLA [HK00]. The underlying model of state machines has been used in SOM [BHS81], TIMe [BGH^{+}97], and the ARTS project. The execution of state machines is based on a runtime support system as described in [BH93] which are implemented for example by JavaFrame [HMP00].

Figure E.1 illustrates the current usage of the profile within the Ramses and Arctis tools Paper 8. Models corresponding to this profile can be constructed in two ways. Using Arctis, executable state machines are synthesized automatically from the collaborative service specification expressed as activities. The models are correct by construction due to the analysis capability of Arctis and the correctness of the transformation. With Ramses, it is also possible to create the ex-

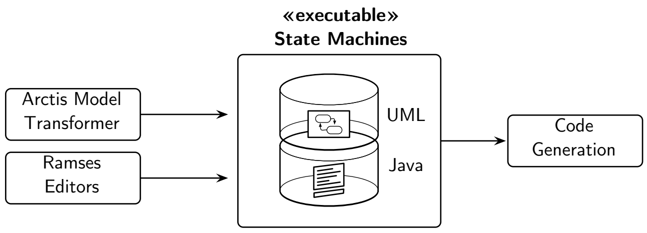


Fig. E.1: Usage of the profile for executable state machines

ecutable state machines manually. The inspectors integrated into Ramses ensure that the constraints implied by the profile apply. From models corresponding to this profile, implementation can be derived using the code generators described in [Kra03, Stø04].

The following sections list numerous constraints on UML models and Java source code. For clarity, we also rephrase some facts implied by the UML standard and explain the motivation for the constraints. The most relevant chapters for this profile in version 2.1.2 of the current UML standard [Obj07] are Chapt. 15 for State Machines, Chapt. 9 for Composite Structures as well as Chapt. 11 and 13 for Actions and Common Behaviors. The files containing the model of this profile are provided within the Ramses tool Paper 8.

E.2 Executable State Machines

The constraints on executable state machines enable a quite simple scheduling scheme and make it easy to implement them efficiently in different programming languages, as described in Paper 2. Executable state machines are marked with the stereotype ≪executable≫.

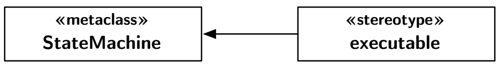


Fig. E.2: Stereotype ≪executable≫

Constraints (State Machine Elements)

[1] A state machine has exactly one region.
[2] A region contains only states, final states and pseudo states.
[3] A pseudo state is of kind *initial* or *choice*.
[4] States do not have any entry or exit activities.
[5] States are not orthogonal or composite and do not refer to submachines.

Executable state machines are *event-driven*, which means that they only execute a transition when they are triggered by an event. These events are either the reception of the signal or the expiration of a local timer. In addition, the creation of the state machine is an implicit event that triggers the initial transition. The transitions are executed by the scheduler in a run-to-completion step. This means that only one transition is executed by a scheduler at a time. Since a scheduler is mapped to one Java thread, the execution of a transition may be interrupted as the corresponding Java thread is interrupted. Therefore, state machines that share variables should always be scheduled within the same Java thread, to avoid inconsistencies due to overlapping use of data.

When the final state of a state machine is reached, the state machine will not be scheduled anymore. The run-time system may dispose all resources concerning this state machine instance.

An executable state machine has one input queue accepting the signals from any connected unit. The queue is ordered in a first-in, first-out manner, so that events are processed in the order of their arrival. Signals (including timeout signals) that are not declared in any outgoing transition of the current control state (and are hence *unexpected*) are discarded. Platform-specific mechanisms may report such an event, as it may be a sign for a design flaw.

Constraints (Triggers)

[1] A transition originating at an initial or choice pseudo state does not declare a trigger, other transitions have exactly one trigger.

[2] The trigger of a transition is either a signal trigger or a time trigger.

[3] Deferrable triggers of a state are signal triggers.

[4] The only triggers are time triggers and signal triggers.

[5] If a state has a transition triggered by an event, the state does not defer the event.

Executable state machines are *non-blocking*, that means that a transition is enabled only due to the current control state of the state machine and the declared trigger. They may not declare additional guards that may block execution. If a transition contains a decision, then at least one outgoing branch is enabled. To enable choices to be implemented by if-statements in the transition method (see Paper 2), and to prevent loops, there must be exactly one transition entering a choice pseudo state.

Constraints (Choices)

[1] Only transitions originating from a choice declare guards.

[2] A choice pseudo state has at least two outgoing transitions.

[3] Exactly one outgoing transition from a choice state has an *else* guard.

[4] Choice pseudo states must have only one incoming transition.

E.2.1 Complementary Java Classes

Each state machine is supplemented by two Java classes[1], to express detailed actions on data. For details about the implementation mechanisms, we refer to [BH93, HMP00] and Paper 2.

– The *status* class keeps the variables of the state machine that are also declared in UML. This additional representation in Java is necessary to initialize the variables with values, using the dedicated method *initialize*. Moreover, this class is passed as parameter to the methods for the actions defined by transitions.

– The action class contains one method for each action. The methods are declared as *public static* methods. As argument, each method receives the signal triggering the transition calling the action, and an instance of the status class, so that the state machine variables are accessible within the body of the method.

Constraints (Java Classes)

[1] The Java action class may contain final static variables (constants), but no instance (non-static) variables.

[2] For simplicity, the Java action class may not contain other methods than those referred by UML. Additional utility classes can be stored within the Java package of the state machine.

[3] The name of the Java action class is the name of the state machine with the suffix *'Actions'*.

[4] The name of the Java status class is the name of the state machine with the suffix *'Status'* or *'SM'*.

[5] If the state machine is owned by a package, the name of the Java package of the complementary Java classes are the name of the UML packages concatenated with the name of the state machine (separated by a period).

If the state machine is the owned behavior of a class, the name of the Java package of the complementary Java classes is the name of the UML packages concatenated with the name of the class (separated by a period).

E.2.2 Variables

A state machine may declare a number of variables, modeled by UML properties. Using some platform-specific reflection mechanism, a state machine may obtain values of variables in other state machines that are part of the same component.

[1]The fact that these are modeled by two distinct classes is based on the original JavaFrame system [HMP00].

Constraints (Variables)

[1] All variables of a state machine refer to a typed element (a UML class or a primitive type) that has the name of an existing Java class or a Java primitive type.

[2] Each variable of the state machine is represented by a corresponding Java variable in the Java status class of the state machine.

E.2.3 Actions

Actions are performed as effects of transitions. Send signal actions model the transmission of signal via ports, call operation actions may refer to methods within the Java action class of a state machine, and opaque actions may directly contain some Java code.

The methods of a Java action class may perform any operations on data and use Java APIs to connect the state machine with specific libraries. In addition, access to platform-specific mechanisms can be used to operate on timers, addresses or to reflect about the system or component structure. In addition, the Java method also contains action to send signals, as described below. Details about such operations are documented by the specific platforms and not covered here.

Several actions may be grouped in an activity. Transitions may refer to activities that include these action directly or indirectly: If a transition declares its effect directly, it *owns* the activity and all its contained actions. This means that once the transition is deleted, also these actions are gone. Moreover, if several transitions want to achieve the exact same effect, these would have to be declared for each of them individually. Therefore, we recommend that a transition refers to its effect indirectly, using a call behavior action. In this way, activities are declared as *owned behaviors* of the enclosing state machine. The effects of transitions contain a single call behavior action that refers to the activity owned by the state machine. If a transition is deleted, only the call behavior action is deleted, while the activity owned by the state machine can still be referred to by other transitions.

Executable state machines send their signals via ports. Usually, these ports are owned by the state machine directly. If the state machine is the classifier behavior of a class, the state machine may also refer to ports directly owned by the enclosing class.

Constraints (Transition Effects)

[1] If a transition has an effect, it is an activity.

[2] The effect of a transition may contain send signal actions, call behavior actions, call operation actions, and opaque actions.

[3] A call operation action refers by name to a method in the Java action class.

[4] A call behavior action refers to an activity owned by the state machine.

[5] A send signal action refers to exactly one signal.

[6] A send signal action refers to exactly one port, which is owned by the state machine or the owning class of the state machine.

Since the sending of signals are important actions of a transition, they are expressed explicitly in UML by send signal actions. This enables the presentation of the state machines in diagrams, and a formal analysis of state machines by tools. However, most signals have parameters that need to be initialized with some data when they are sent. Most often, these parameters are created as part of a transitions effect, which means that the values are present as local variables within the Java method executed as effect of the transition. Therefore, sending signals is done not only within UML, but also as as part of the Java action method. The following constraints must hold so that UML send signal actions are consistent with the Java actions.

Constraints (Java Actions for Sending Signals)

[1] Signals sent in Java code conform to the send signal actions in the model.
[2] Sending of signals in Java is unconditional, that means if a signal sending is declared, the method will send the signal in all cases, and not depend on an Java if-statement, for example.

E.2.4 Timers

UML allows to specify time events as triggers of transitions. A transition triggered by a timeout owns a trigger, which points to a time event. The name of the time event is taken as name of the timer. Time events must be owned by a package.

UML does not provide actions to start, stop, or reset timers. These are therefore put into the Java method called by a transition, using platform-specific operations to handle timers.

E.3 Signals

Since signals may be accessed within the code of the Java action methods, UML signals are represented by Java classes as well.

Constraints (Java Classes for Signals)

[1] The name of the Java class or the signal corresponds to that of the UML signal.
[2] If the signal is owned by a UML package, the Java package name of the class corresponds to the UML package name. If the signal is owned by a class, its Java package name corresponds to the Java package name for the UML class.
[3] Each parameter of the signal corresponds to a Java variable.

According to the rule in [BH93], internal signals, that means, signals sent between the state machines within a component, should be processed with higher priority than signals arriving from other components. For that reason, signals that are sent within components are owned by the component, while signals sent between components are owned by packages. Signal classes may extend framework classes so that they inherit features for addressing, routing, serialization, for instance.

E.4 Classes and Components

System components are deployable units, modeled as active classes in UML.[2] State machines within a component are assumed to execute locally concentrated, that means all state machines within one component are executed within the same scheduler.

A ≪system≫ class may be used to depict the structure of a system. Its parts are interpreted as component instances at run-time, and an execution framework will start these system components. Note that this way of declaring a system has been chosen for simplicity. UML deployment diagrams may be more suitable for this task.

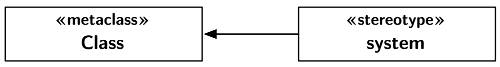


Fig. E.3: Stereotype ≪system≫

Constraints (Classes)

[1] The classifier behavior of a class is an executable state machine.
[2] Owned behaviors of a class are executable state machines.
[3] Inner parts of classes refer to other classes or to executable state machines.
[4] ≪system≫ classifiers do not have any owned behaviors.
[5] Parts of ≪system≫ classifiers refer to components.

The classifier behavior (a state machine) may access the data of the owned state machines as well, for example to select one of them.

E.5 Packages

The constraints on packages are to enable easier tool support. Nested packages are not allowed, as too many levels can make a model hard to access. Instead,

[2]Technically, it is not necessary that our components are modeled as *UML components*, since no features of this meta-class are used. Nevertheless, a tool may decide to model them as UML components to facilitate the difference to UML classes that may be used to model other aspects of data.

a hierarchical naming scheme for packages, similar to the one for Java packages (see [GJSB00]), is recommended. Since packages are also used in the creation of names for Java classes, it is recommended that UML packages are named with valid names for Java packages.

Constraints (Packages)

[1] Packages do not contain packages.
[2] Packages have valid names for Java packages (see [GJSB00]).

Bibliography

[BGH+97] Rolv Bræk, Joe Gorman, Øystein Haugen, Geir Melby, Birger Møller-Pedersen, and Richard Sanders. Quality by Construction Exemplified by TIMe — The Integrated Methodology. *Telektronikk*, 95(1):73–82, 1997.

[BH93] Rolv Bræk and Øystein Haugen. *Engineering Real Time Systems: An Object-Oriented Methodology Using SDL.* The BCS Practitioner Series. Prentice Hall, 1993.

[BHS81] R. Bræk, O. Helle, and F. Sandvik. SOM — A SDL Compatible Specification and Design Methodology. In *4th International Conference on Software Engineering for Telecommunication Switching Systems, Conventry*, volume 198, pages 111–117, July 1981.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[HK00] Peter Herrmann and Heiko Krumm. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337, 2000.

[HMP00] Øystein Haugen and Birger Møller-Pedersen. JavaFrame — Framework for Java Enabled Modelling. In Proceedings of Ericsson Conference on Software Engineering, September 2000.

[Kra03] Frank Alexander Kraemer. Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart, 2003.

[Obj07] Object Management Group. Unified Modeling Language: Superstructure, version 2.1.2, November 2007. formal/2007-11-01.

[Stø04] Alf Kristian Støyle. Service Engineering Environment for AMIGOS. Master's thesis, Norwegian University of Science and Technology, 2004.

LIST OF FIGURES

INDEX