Mälardalen University Press Dissertations
No. 122

A RESOURCE-AWARE FRAMEWORK FOR DESIGNING PREDICTABLE COMPONENT-BASED EMBEDDED SYSTEMS

Aneta Vulgarakis

2012

School of Innovation, Design and Engineering

ISBN 978-91-7485-068-0
ISSN 1651-4238
Printed by Mälardalen University, Västerås, Sweden

Mälardalen University Press Dissertations
No. 122

A RESOURCE-AWARE FRAMEWORK FOR DESIGNING PREDICTABLE COMPONENT-BASED EMBEDDED SYSTEMS

Aneta Vulgarakis

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid
Akademin för innovation, design och teknik kommer att offentligen försvaras
fredagen den 15 juni 2012, 14.00 i Kappa, Mälardalen University, Västerås.

Fakultetsopponent: Professor Elisabetta Di Nitto, Politecnico
di Milano, Dipartimento di Elettronica ed Informazione

Akademin för innovation, design och teknik

Abstract

Managing complexity is an increasing challenge in the development of embedded systems (ES). Some of the factors contributing to the increase in complexity are the growing complexity of hardware and software, and the increased pressure to deliver full-featured products with reduced time-to-market. An attractive approach to manage the software complexity, reduce time-to-market and decrease development costs lies in the adoption of component-based development that has been proven as a successful approach in other domains. Another raising challenge, due to complexity increase, in ES, is predictability, i.e., the ability to anticipate the behavior of a system at run-time. The particular predictability requirements of ES call for a development framework equipped with techniques and tools that can be applied to deal with requirements, such as timing, and resource utilization, already at early-stage of development. Modeling and formal analysis play increasingly important roles in achieving predictability, since they can help us to understand how systems function, validate the design and verify some important properties.

In this thesis, we present a resource-aware framework for designing predictable component-based ES. The proposed framework consists of (i) the formally specified ProCom component model that takes into account the characteristics of control-intensive ES, and (ii) the resource-aware timed behavioral language - REMES for modeling and reasoning about components' and systems' functional and extra-functional behavior that includes relevant resource types for ES, associated analysis techniques for various resource-wise properties, and a set of associated tools. To demonstrate the potential application of our framework, we present a number of case studies, out of which one is an industrial research prototype, where ProCom and REMES are applied.

ISBN 978-91-7485-068-0
ISSN 1651-4238

To my family

Abstract

Managing complexity is an increasing challenge in the development of embedded systems (ES). Some of the factors contributing to the increase in complexity are the growing complexity of hardware and software, and the increased pressure to deliver full-featured products with reduced time-to-market. An attractive approach to manage the software complexity, reduce time-to-market and decrease development costs lies in the adoption of component-based development that has been proven as a successful approach in other domains. Another raising challenge, due to complexity increase, in ES, is predictability, i.e., the ability to anticipate the behavior of a system at run-time. The particular predictability requirements of ES call for a development framework equipped with techniques and tools that can be applied to deal with requirements, such as timing, and resource utilization, already at early-stage of development. Modeling and formal analysis play increasingly important roles in achieving predictability, since they can help us to understand how systems function, validate the design and verify some important properties.

In this thesis, we present a resource-aware framework for designing predictable component-based ES. The proposed framework consists of (i) the formally specified ProCom component model that takes into account the characteristics of control-intensive ES, and (ii) the resource-aware timed behavioral language - Remes for modeling and reasoning about components' and systems' functional and extra-functional behavior that includes relevant resource types for ES, associated analysis techniques for various resource-wise properties, and a set of associated tools. To demonstrate the potential application of our framework, we present a number of case studies, out of which one is an industrial research prototype, where ProCom and Remes are applied.

Acknowledgements

I have always wanted to get a PhD degree and to invent something... I have never imagined that this dream will take me to Sweden, a country that I used to think should be visited only during summer. I can not say that I have invented something amazing, but I can say that I have learned a lot. However, I am not nearly as proud from the knowledge I have gained, as I am proud from the people I have met during my PhD journey. Therefore, I am delighted to put some words acknowledging all the people who helped me go through this somewhat difficult, but often amazing period of my life.

I would start with the most significant people in the act of the actual realization of this thesis. My deepest thanks goes to my main supervisor Ivica Crnković, for giving me the opportunity to be a Ph.D. student and believing in me. I am impressed by your ability to get both the details and the big picture of my research. But most of all I am impressed by your ability to work so much, and still be so positive and energetic. Second, I want to thank my assistant supervisor Paul Pettersson. I am amazed by your ability to make research topics seem less complicated. Last but not least, I want to thank my second assistant supervisor Cristina Seceleanu. I do not want to thank you only as a supervisor, but also as an invaluable friend that has always been there for me, and has supported me many times. Your friendship and guidance have made me not only a better researcher, but also a better person. Thank you so much for this!

A special thanks goes to Marin Orlić, with whom I worked the most towards the end of my PhD, and who provided me with insightful ideas and suggestions that greatly influenced my PhD. The fact that you are working in Zagreb did not influence on our great collaboration, and most importantly fun talks. It has been really great working with you!

I have authored and co-authored more than 20 different papers. I would have never done that without the help of very capable and hard working co-authors. Many thanks to my fellow authors for the pleasant collaboration: Tomáš Bureš, Jan Carlson, Aida Čaušević, Michel Chaudron, Ivica Crnković, Darko Huljenić, Dinko Ivanov, Marin Orlić, Cristina Seceleanu, Séverine Sentilles, Ivan Skuliber, Jagadish Suryadevara, Paul Pettersson and Mario Žagar.

I would like to thank the PROGRESS-ers Jan Carlson, Hans Hansson, Björn Lisper, Kristina Lundqvist, Sasikumar Punnekkat, Mikael Sjödin, Malin Rosqvist and Gunnar Widforss. Without you PROGRESS would not have been a success. I would also like to thank Hans Hansson for the guidance in the research planning course, Gordana Dodig-Crnković and Jan Gustafsson for introducing me to the research methodology, Rikard Land and Frank Lüders for the stimulating collaboration in the courses Distributed Software Development and Software Engineering, and the administrative staff at the department, in particular Hariet Ekwall, Monica Wasell, Monika Matevska Stier and Carola Ryttersson.

Next, I would like to thank my officemates, Séverine Sentilles, Hongyu Pei Breivold and Gaetana Sapienza for the talks we had, but especially for bering with my sometimes dancing behavior.

Having lunch and drinking coffee with the people from the department has been an enjoyable activity. Many ideas, mostly outside of the research were born during these breaks, such as time-machines, metaprinters and bars where people would actually pay for their beers. I want to thank Adnan Čaušević, Aida Čaušević, Nikola Petrović, Stefan (Bob) Bygde, Juraj Feljan, Cristina Seceleanu, Jan Carlson, Aleksandar Dimov, Josip Maras, Ana Petričić, Teodora Puleva, Pasqualina Potena, Antonio Cicchetti, Batu Akan, Svetlana Girs, Dag Nyström, Farhang Nemati, Hongyu Pei Breivold, Hüseyin Aysan, Séverine Sentilles, Iva Krasteva, Leo Hatvani, Luka Lednicki, Yue Lu, Raluca Marinescu, Eduard Paul Enoiu, Mehrdad Saadatmand, Saad Mubeen, Federico Ciccozzi, Jiří Kunčar, Sanja Šain, Thomas Nolte, Etienne Borde, Rikard Land, Lars Asplund, Marcelo Santos, Andreas Gustavsson, Sara Dersten, Frank Lüders, Barbara Gallina, Kathrin Dannmann, Mikael Åsberg, Andreas Johnsen, Damir Isović, Mikael Åkerholm, Jagadish Suryadevara, Johan Fredriksson, Daniel Sundmark, Andreas Hjertström, Jayakanth Srinivasan, Anton Jansen, Moris Behnam, Thomas Leveque, Radu Dobrin, Rafia Inam, Abhilash Thekkilakattil, Hang Yin, Jiale Zhou, Gaetana Sapienza, Tiberiu Seceleanu and Giacomo Spampinato. Most of

you have been more friends than colleges to me.

During my PhD studies I have spent 3 great months at the Faculty of Electrical Engineering and Computing, University of Zagreb, and additional 3 months at ABB CRC. For this I want to especially thank Prof. Mario Žagar and Magnus Larsson.

Thanks to my Macedonian friends Bojana, Marija and Suzana, and to my Bulgarian friend Velemira, who no matter the distance have still stayed very dear and close to my heart.

Thanks to my grandparents Mitka, Petranka and Angel, who unfortunately are not here anymore. I would like to thank them for their love and for everything they thought me. I know they would have been proud of me!

To my dear sister Sofija, her husband Boris and my nephew Filip. Thanks for believing in, and supporting me in different ways. You have given me positive energy when I needed it the most!

I would like to express my deepest gratitude to my parents, Mirjana and Janko, who, although thousand kilometers apart, have stood by my side, encouraged me, believed in me and loved me. Thank you for always being with me, and guiding me through life. I owe and dedicate this work to you and consider it as your success as much as it is mine!

At the end I want to thank Juraj. I have shared with you each moment of this experience and you have complemented and balanced my life in a beautiful way. I admire you in many ways! I am grateful for your unselfish love, understanding and strength that you have given to me. Thank you for being my refuge from the everyday problems and worries, and for the ability to always put a smile on my face. Without you I would have never done it!

Aneta Vulgarakis
Västerås, May, 2012

This work has been supported by the Swedish Foundation for Strategic Research (SSF), via the research centre PROGRESS.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Embedded systems, such as mobile phones, car engines, elevators, etc., are part of our daily life, and we are increasingly depending on their reliability in operation. According to IEEE Glossary [3] "an embedded system is a computer system that is part of a larger system and performs some of the requirements of that system". Embedded systems are designed to perform dedicated functions, often under real-time computing constraints. In most cases, they are made of *components* that communicate with each other and the environment via sensors and actuators.

During last decades, the amount of software in embedded systems is increasing at a breathtaking pace. For example, a modern upper-class car holds between a dozen and nearly 100 crosslinked electronic control units (ECU), each with a microprocessor software that amounts to about 1MByte compiled code [42]. This is comparable to what a typical desktop computer runs today. Reasons for this tremendous increase include the demand for new functionality on the one hand, and the availability of powerful and cheap hardware on the other hand. In contrast to the changing nature of software, the *resources* that such systems use (like computation power, memory, and channel bandwidth) are limited in capacity, expensive and usually not extensible during system's lifetime. The limited nature of the available resources, especially memory size and computation resources, complicates meeting the real-time constraints and dependability requirements.

As pointed out by Henzinger and Sifakis, designing embedded systems is not a straightforward application of either hardware or software

design methods [55]. The demanding extra-functional requirements of modern embedded systems, coupled with the increasing complexity of the underlying software, require techniques for managing complexity and for ensuring predictable system behavior. One of the ways to ensure predictable behavior of an embedded system design is to formally check it against different requirements pertaining to various kinds of constraints including functional, timing, safety, and resource usage constraints. Meeting this demanding goal resorts to a *resource-aware* embedded system modeling and analysis perspective, that is, consider from the start of the development the resource constraints imposed from the underlying hardware and/or software platforms that host the embedded system.

Designing an embedded system in a *component-based* manner, by building it from pre-existing well-specified and verified components, intends to lower its complexity, reduce time-to-market, introduce structure and abstraction. The underlying paradigm of component-based development (CBD) is that individual components are designed and developed to provide functionality that is potentially reusable for future systems. The central point of CBD has been reuse, but for embedded systems the structure and abstractions introduced by components are equally important as a basis for the construction of abstract formal models. An essential benefit of a formal model is that it enforces a precise and unambiguous way of component and system specification, which may reveal inconsistencies and gaps in the original informal description. Through abstraction formal models allow software engineers to focus on the critical issues facing them. Through logical foundations they support predictable development already at early design time, where *predictability* refers to the possibility to guarantee absence or presence of certain properties, or to predict/guarantee quantified properties. This avoids cost intensive redesigns of systems in late development phases [80]. In practice, it may often be necessary to replace a component with another one having the same functionality, yet using a more sophisticated control algorithm that requires bigger memory resources. The predictability analysis should guide the design and selection of hardware and software system components. The final implementation of the system should be arrived at, as much as possible, by using automatic transformation and synthesis from formal models describing the system behavior in order to ensure implementations that are "correct by construction" [41].

In the remainder of this chapter, we describe in detail the research

problem tackled in this thesis and list the research goals relevant to the problem (Section 1.1). Afterwards, we point out the scientific contributions of the thesis (Section 1.2), before we list the published papers that establish the contributions of the thesis (Section 1.3). Finally, we present the research methodology used for answering the research problem (Section 1.4), and provide an outline of the thesis (Section 1.5).

1.1 Problem Statement and Research Goals

In the previous section, we have argued that the development of embedded systems is a challenging task, due to their growing complexity and the pervasive nature of their most critical property: resource limitations. Resource usage should be predicted and assessed already at the early design phases, since access to such information at early stages of design might help the designer to get insights into the overall system resource usage, which in turn could help him/her prevent resource misuse at run-time. Moreover, early design prediction for embedded systems is both important and feasible, since in most cases, in particular for safety-critical systems, the embedded systems are not changed during runtime.

Based on the above discussion, we identify our general research problem coming from the embedded systems practice as:

The need to address the complexity and resource limitations of embedded systems in a structural way and ensure predictability during early stages of system development.

In order to refine this general research problem, we narrow our focus from different perspectives. Firstly, we consider that in order to achieve predictability throughout the development of embedded systems, the designer needs to employ a design framework equipped with analysis methods and tools that can be applied at various levels of abstraction. These methods and tools should provide estimations and guarantees of relevant system properties.

Secondly, we rely on the principle that CBD introduces structure in design, and provides means of abstraction, while enabling reusability of various types of analysis. Hence, we assume the CBD paradigm in our framework.

Thirdly, in our view, formal analysis of functionality, timeliness and resource usage is an important complement to testing. For instance, ensuring the resource-wise feasibility of a system/component is hard to obtain through testing. Such property can state that the composition of the worst-case resource requirements of components stays within the available resources provided by the implementation platform, or that there exists an execution path that uses no more than the available resources to behave correctly.

Taking into account these objectives, we consider that in order to be able to synthesize a predictable embedded system from components and compositions, a *resource-aware design framework* is needed. Therefore, we specify our refined research problem as an overall research goal:

Develop a resource-aware design framework encompassing modeling and formal analysis of component-based embedded systems.

Research Goals

Decomposing the overall research goal, we formulate three smaller research goals that we address in this thesis.

Research goal 1. (A component model formalization)

The potential benefits of CBD are as attractive in the domain of embedded systems as they are in other areas of the software industry. Component models are indispensable to CBD, as they define rules for constructing individual components and for assembling them into systems. Beside component models, component technologies form another central concept of CBD. They make use of component models in practice, that is, a particular component technology provides tools that enable development and deployment of systems that adhere to a corresponding component model. Although there exist several component models and technologies for the development of embedded systems (e.g., AUTOSAR [18], BlueArX [67], COMDES-II [66], Koala [101], Pecos [108], Robocop [78], Rubus [52], and SaveCCM [8]), CBD is still not broadly used in the embedded systems industry. An important reason for such limited success is the difficulty of providing solutions that meet typical embedded system requirements.

Wolf [109] discusses about which domain specific requirements a component technology targeting embedded system development should be aware of. In the embedded systems domain, designing for predictability requires architectures that meet both the corresponding functional requirements (e.g., expected services, functionality and features), as well as extra-functional ones (resource-feasibility, timing and/or reliability). In order to simplify analysis and help the intuition behind the embedded system's functioning, one could create a hierarchy of models that will alow him/her to reason about timed behavior, resource consumption, etc., without going down to the instruction level. For instance, architectural models may be used for modeling the system's structure, and high-level functionality, assuming different views, whereas behavioral models can be associated with architectures to express much richer semantic models, and describe internal functional and extra-functional behavior, as well as interface behavior [46, 89]. Also, embedded system developers must be able to verify that applications meet their functional and extra-functional specifications. All these demands should be possible to meet when employing a particular component model. However, the specifications of many component models are defined informally and component models suffer from incomplete and imprecisely defined syntax and semantics. A formalization of the component model is then needed, in order to achieve an unambiguous model that can be formally analyzed. Consequently, it is essential to associate the component model and its constructs with a formal execution semantics to which any design should conform. Such motivation justifies our first research goal:

Develop a formal description of a component model for real-time embedded systems.

(RG1)

Research goal 2. (A resource-aware behavioral language)

The diversity of approaches on resource modeling and analysis existing in the literature [15,39,45,48,75–77,85,88] indicate the complexity of handling all relevant embedded resources within the same formal model. This calls for an innovative look on resource-aware design methods, based on the experience gathered from the existing modeling approaches. In order to properly specify and analyze embedded systems, the designer

requires a modeling language that incorporates resources as primitive types, that is, built in the model. Ideally, the language should be rich enough to support modeling and analysis of functional and timing behavior too. This would allow for both separation of concerns, as well as easier model-to-model transformations, for analysis purposes. Accordingly, the second research goal can be formulated as:

> ***Develop a behavioral language and associated tool support for modeling and formal analysis of functional, timing and resource-wise behavior of components and their compositions.***
>
> ***(RG2)***

Research goal 3. (Validation)

The usefulness, applicability, and scalability of embedded systems modeling languages and analysis methods can be exercised by performing their validation against measured, quantified behavioral properties. In order to illustrate, as well as validate the applicability of our design framework, we must apply our proposed framework on a number of relevant case-studies. Thus, our third research goal is:

> ***Exercise the applicability of the proposed design framework by modeling and analyzing example embedded systems that are motivated by reality.***
>
> ***(RG3)***

1.2 Contributions

In this section, we map the contributions of the thesis to the goals formulated earlier.

Research goal 1. (A component model formalization)

> ***Develop a formal description of a component model for real-time embedded systems.***
>
> ***(RG1)***

RG1 has been addressed with the following contribution:

- **The formally specified ProCom component model for embedded systems.** To address RG1, we have contributed to the development of ProCom, the component model used in this thesis. ProCom is particularly designed to target control-intensive distributed systems, which are a special class of embedded systems that can be found in many products, such as vehicles, automation systems, or distributed wireless networks. In order to address the different concerns at different levels of granularity, ProCom is structured in two distinct, but related, layers (ProSys and ProSave). The two layers differ in terms of granularity, architectural style and communication paradigm. To facilitate analysis, we have defined the formal execution semantics of ProCom, based on an extension of finite-state machines (FSM). The proposed FSM language has notions of urgency, implicit timing and priorities. Its formal semantics is expressed in terms of timed automata with priorities [38] and urgent transitions [23]. The FSM language has graphical appeal, making it simpler than the corresponding timed automata model, by, e.g., abstracting from real-valued variables and synchronization channels. We present the ProCom component model and its formalization in Chapter 3.

Research goal 2. (A resource-aware behavioral language)

> ***Develop a behavioral language and associated tool support for modeling and formal analysis of functional, timing and resource-wise behavior of components and their compositions.***
>
> ***(RG2)***

The contributions addressing RG2 are as follows:

- **The Remes behavioral language.** Our REsource Model for Embedded Systems (Remes) is intended as a meaningful basis for modeling and analysis of resource-constrained behavior of embedded systems. Remes is a dense time state-based hierarchical behavioral language that has a notion of explicit entry- and exit points, continuous variables, flows and progress invariants, making

it fit for component-based system modeling of timed systems. We introduce the REMES language in Chapter 4.

- **ProCom and REMES integration.** In order to specify the ProCom behavior via REMES, we need to integrate the two models. The integration is done via a general attribute framework [92], that enables a developer of a ProCom component to specify the corresponding behavior by pointing to a REMES model. Both the ProCom component and the associated REMES model are seen as a reusable unit of composition. To accomplish this, in this thesis, we propose a way of connecting ProCom and REMES together. The relation between the ports of the component and the variables in the REMES model is given by a mapping between the ProCom and REMES interfaces. This contribution we present in Chapter 5.

- **Performing resource-wise analysis.** To analyze the resource-wise behavior in REMES models, we encode the total resource usage, as a weighted sum, in which the variables capture the accumulated consumption of each resource, respectively. Assuming the encoding, we perform three types of analysis: feasibility analysis, optimal and worst-case resource consumption analysis, and trade-off analysis. Feasibility analysis checks whether the accumulated values of the resources used during all possible system behaviors are within the available resource amounts provided by the implementation platform. Optimal resource usage analysis returns the cost of the of the "cheapest" trace, whereas worst-case resource consumption analysis calculates the cost of the most "expensive" trace that will eventually reach some goal. The latter analysis may help in resolving the possible non-determinism in a component implementation. Trade-off analysis is an approach to balancing trade-offs between conflicting resource requirements: memory vs. execution time, energy vs. memory, etc. The result of this analysis is the best alternative between the conflicting requirements. These analysis goals are encoded in Weighted Computation Tree Logic (WCTL) [32], which is our property specification language. In Chapter 4 we show how a number of resource analysis problems can be formalized in the framework of priced timed automata.

- **A tool-chain for the REMES language.** To be able to apply our framework, we have developed automated support, as an integrated

tool for modeling and analysis of embedded systems. The core elements of the tool-chain are as follows: (i) the REMES editor for modeling behaviors of embedded components, (ii) the REMES simulator to test timing and resource behavior prior to formal analysis, and (iii) an automated transformation from REMES into priced timed automata, needed for formal analysis. The REMES simulator is out of the scope of this thesis and therefore will only be shortly described. We present the REMES tool-chain in Chapter 6.

Research goal 3. (Validation)

Exercise the applicability of the proposed design framework by modeling and analyzing example embedded systems that are motivated by reality.

(RG3)

RG3 has been addressed with the following contribution:

- **Validating the resource-aware framework.** ProCom and REMES have been applied on simple, yet relevant "toy examples": an electronic stability control system (see Chapter 3), a temperature control system (see Chapter 4 and 5) and a turntable drilling system (see Chapter 5). In Chapter 7, we also show how to model behavior, and verify the resulted behavioral models of an industrial prototype, a component-based Ericsson Nikola Tesla prototype telecommunication system. In this last case, we validate our models by using the actual values of timing, CPU, and memory usage in our models, measured by Ericsson researchers on the prototype's source code.

Hence, all three smaller research goals have been targeted, and consequently, also the overall research goal *"develop a resource-aware design framework encompassing modeling and formal analysis of component-based embedded systems"*. Needless to say, we have provided only one solution to the overall research problem, out of a possibly large pool of valid solutions.

The resource-aware design framework that we present in this thesis includes two parts:

1. The formally specified ProCom component model that fulfills the requirements coming from a class of embedded systems that primarily perform real-time controlling tasks;

2. The REMES behavioral language for describing component's and system's functional and extra-functional behavior (such as timed behavior and resource consumption), associated analysis techniques for various resource-wise properties, and a set of tools implementing the former.

1.3 Publications

This section presents planned and published papers related to the thesis. The publications are divided into two categories: (i) papers that are fundamental for the thesis contributions; and (ii) papers that are related to the thesis.

1.3.1 Description of fundamental publications

Licentiate thesis

- *A Resource-Aware Component Model for Embedded Systems.* Aneta Vulgarakis. Licentiate Thesis, ISBN 978-91-86135-37-9, Mälardalen Univerisity Press, September 2009.

  Summary: In this thesis, we introduce the ProCom component model for building embedded systems, as well as the REMES behavioral language for describing the internal behavior of components.

  Usage in the thesis: This doctoral thesis is a continuation of the research work presented in the licentiate thesis. In the doctoral thesis we extend the REMES behavioral language, introduce a set of transformation rules that semantically translate REMES modes into priced timed automata, show a tool for modeling and analysis of REMES models, present an integration of ProCom and REMES, and validate the REMES behavioral language.

Journals

- **paper A.** *Resource-Oriented Modeling and Formal Analysis of Embedded Systems Behavior.* Marin Orlić, Aneta Vulgarakis, Cristina Seceleanu, and Paul Pettersson. To be submitted to IEEE Transactions on Software Engineering.

  Summary: This paper is based on the work presented in papers E and G. Additionally, the paper presents an extension of the REMES behavioral language, reveals a solution for the problem regarding the access to shared variables of REMES modes, and presents a set of transformation rules for translating REMES modes into priced timed automata.

  Contribution: I and Marin Orlić are the main authors of this paper. I am responsible for addressing the problem with access to shared variables of REMES modes. Together, Marin Orlić and I have formally defined the automated transformation from REMES into priced timed automata, with equal contribution. All the coauthors have contributed with writing sections of the paper, as well as with valuable suggestions and ideas.

  Usage in the thesis: This paper is a basis for Chapter 4 and Chapter 6. It describes the extended version of the REMES behavioral language, the set of transformation rules for translating REMES modes into priced timed automata, and the REMES toolchain.

- **paper B.** *A Classification Framework for Component Models.* Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel Chaudron. IEEE Transactions on Software Engineering. October, 2011.

  Summary: This paper presents a survey of a number of component models, described and classified with respect to a three dimensional classification framework, which groups different aspects of the development process of component models. As such, this classification framework identifies common characteristics as well as differences between selected component models. The results of the comparison have led to some observations which are discussed in this paper.

  Contribution: This paper was written with an equal contribution of the first three authors. All the coauthors have contributed with

ideas, discussions, and reviews. I was responsible mainly for the lifecycle dimension and shared the responsibility with Séverine Sentilles for collecting, analyzing and classifying in tables the included component models. The classification framework was developed in several iteration steps including observations and analysis. It was discussed with several CBD and empirical software engineering researchers and experts from different engineering domains.

Usage in the thesis: This paper is used in Chapter 8 for describing the state of the art of component models for embedded systems. In addition, the knowledge gained from this paper is used as a basis for designing the ProCom component model, presented in Chapter 3.

Conferences and workshops

- **paper C.** *Validation of Embedded Systems Behavioral Models on a Component-Based Ericsson Nikola Tesla Demonstrator.* Aneta Vulgarakis, Cristina Seceleanu, Paul Pettersson, Ivan Skuliber and Darko Huljenić. 11th International Conference on Quality Software, IEEE, Madrid, Spain, July, 2011.

  Summary: In this paper, we show how to model extra-functional behavior, and verify the resulted behavioral models of a component-based Ericsson Nikola Tesla prototype telecommunications system. The models are described in our REMES language, with Priced Timed Automata semantics that allows us to apply UPPAAL - based tools for model-checking the system's response time and compute optimal resource usage traces. The validation of our models is ensured by using actual values of timing, CPU, and memory usage in our models, measured by Ericsson researchers on the prototype's source code. For timing, the result of our verification is then compared to the measured value.

  Contribution: I was the main author of this paper. I contributed to this paper with modeling and analyzing the ENT system. All the coauthors have contributed with valuable discussions and reviews. The requirements and measurements of the ENT system were given by the last two coauthors of this paper, researchers at Ericsson, Croatia.

  Usage in the thesis: This paper is a basis for Chapter 7, and

describes the validation of the REMES behavioral language on the ENT system.

- **paper D.** *Integrating Behavioral Descriptions into a Component Model for Embedded Systems.* Aneta Vulgarakis, Séverine Sentilles, Jan Carlson, and Cristina Seceleanu. 36th Euromicro Conference on Software Engineering and Advanced Applications, IEEE, Lille, France, September, 2010.

  Summary: In this paper, we show how the ProCom component model can be combined with the REMES behavioral language. This permits analysis of system properties, while also supporting reuse of behavioral models when components are reused.

  Contribution: I was the main driver of this paper. I proposed a way of mapping the ProCom component interface onto the entry and exit variables of REMES modes, such that the two models become connected. Séverine Sentilles was in particular responsible for implementing this connection through a general attribute framework. I was also responsible for exemplifying the connection on a turntable system. All the coauthors have contributed with valuable discussions and reviews.

  Usage in the thesis: This paper is a basis for Chapter 5 where the integration of ProCom and REMES is presented.

- **paper E.** REMES *Tool-chain - A Set of Integrated Tools for Behavioral Modeling and Analysis of Embedded Systems.* Dinko Ivanov, Marin Orlić, Cristina Seceleanu and Aneta Vulgarakis. 25th IEEE/ ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September, 2010.

  Summary: In this paper, we present our REMES tool-chain that can be employed for construction and analysis of embedded behavioral models. The core elements of the tool-chain are as follows: (i) the REMES editor for modeling behaviors of embedded components, (ii) the REMES simulator to test timing and resource behavior prior to formal analysis, and (iii) an automated transformation from REMES into priced timed automata, needed for formal analysis.

  Contribution: I and Marin Orlić were the main authors of this paper. I was the REMES tool-chain leader and supervisor, and

contributed with suggesting a design of the REMES editor and the REMES meta-model. I and Marin Orlić developed an algorithm for transforming REMES into priced timed automata. Dinko Ivanov developed the REMES editor and Marin Orlić developed the REMES simulator. Cristina Seceleanu coordinated the work on the REMES tool-chain and reviewed the paper.

Usage in the thesis: This paper is a basis for Chapter 6 where the REMES editor and the transformation from REMES into priced timed automata are presented.

- **paper F.** *Formal Semantics of the ProCom Real-Time Component Model.* Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. 35th Euromicro Conference on Software Engineering and Advanced Applications, IEEE, Patras, Greece, August, 2009.

  Summary: In this paper, we define the formal execution semantics of the ProCom component model in a small but powerful finite state-machine based formalism, with notions of urgency, timing, and priorities. As such, the formalism provides an unambiguous description of the modeling elements of ProCom, sets the ground for formal analysis using other formalisms, and provides and intuitive and useful description for both practitioners and researchers.

  Contribution: I was the main author of this paper. I and Jagadish Suryadevara contributed with defining a formal execution semantics of the ProCom component model and exemplifying it on the modeling elements of ProCom. All the coauthors have contributed with valuable discussions and reviews. The paper proceeded from a technical report that was written together with Jagadish Suryadevara.

  Usage in the thesis: This paper is used in Chapter 3 for describing the formal execution semantics of the ProCom component model.

- **paper G.** REMES*: A Resource Model for Embedded Systems.* Cristina Seceleanu, Aneta Vulgarakis, and Paul Pettersson. 14th IEEE International Conference on Engineering of Complex Computer Systems, IEEE, Potsdam, Germany, June, 2009.

  Summary: This paper introduces the model REMES for formal

modeling and analysis of both functional and extra-functional behavior of interacting embedded components. REMES is a state-based behavioral language with support for hierarchical modeling, resource description, continuous time, and notions of explicit entry and exit points that make it suitable as a semantic basis for component-based modeling of embedded systems. The analysis of REMES-based systems is placed around a weighted sum in which the variables capture the accumulated consumption of resources, respectively.

Contribution: This paper was written with equal contribution from all the authors. I particularly worked on the classification of the resources and specified, modeled in REMES, and analyzed in UPPAAL CORA [100] the TCS system presented as a case study in the paper.

Usage in the thesis: This paper is a basis for Chapter 4 where the REMES behavioral language is introduced.

- **paper H.** *A Component Model for Control-Intensive Distributed Embedded Systems.* Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković. 11th International Symposium on Component Based Software Engineering, Karlsruhe, Germany, October 2008.

  Summary: In this paper, the two-layered ProCom component model for design and development of control-intensive distributed embedded systems is introduced. ProCom takes into account the most important characteristics of these systems and employs the concept of reusable components throughout the whole development process, from early design to deployment. The two-layered model is developed to efficiently cope with different design paradigms that exist at different abstraction levels of embedded systems (high level view of loosely coupled subsystems and a low-level view of control loops controlling a particular piece of hardware). Additionally it provides ground for analysis and predicting properties (e.g., timed behavior and resource consumptions) in such systems.

  Contribution: This paper was written with equal contribution from all the authors, and proceeded from a technical report that was written together with all the authors. I took part in the discussions and contributed with writing and improving parts of the pa-

per, particulary in the discussions about the semantics of the component model, analysis and predicting properties and the related work section. The ProCom component model that we describe in this paper was developed in several iteration steps resulting from the conducted discussions between the authors.

Usage in the thesis: This paper is a basis for Chapter 3 where the ProCom component model is introduced.

- **paper I.** *Embedded Systems Resources: Views on Modeling and Analysis.* Aneta Vulgarakis and Cristina Seceleanu. 1st IEEE International Workshop On Component-Based Design Of Resource-Constrained Systems, IEEE, Turku, Finland, July, 2008.

  Summary: In this paper, we discuss several representative frameworks that model and estimate resource usage of embedded systems, identifying their advantages and limitations. As such, we divide the variety of approaches existing in the literature into three distinctive categories: code-level resource modeling and analysis of component assemblies, UML-based description of embedded resources and higher-level formal approaches based on temporal logics and process algebras. In the end, we present the resource-aware development view that we are adopting throughout the rest of the thesis.

  Contribution: This paper was written with equal contribution from both authors. I was specifically working on the code-level and UML- based resource modeling and analysis.

  Usage in the thesis: This paper is used in Chapter 8 for describing the state of the art of embedded systems resources modeling and analysis. In addition, the knowledge gained from this paper is used as a basis for designing the Remes behavioral language, presented in Chapter 4.

1.3.2 Publications related to the thesis

Journals

- *Applying* Remes *Behavioral Modeling to PLC Systems.* Aneta Vulgarakis and Aida Čaušević. Mechatronic Systems, vol 1, nr 1, p40-49, Faculty Of Electrical Engineering, University Sarajevo, December, 2009.

Conferences and workshops

- *Classification and Survey of Component Models.* Ivica Crnković, Aneta Vulgarakis, Mario Žagar, Ana Petričić, Juraj Feljan, Luka Lednicki, and Josip Maras. DICES workshop at the International Conference on Software Telecommunications and Computer Networks, Bol, Croatia, September 2010.
- *Towards Simulative Environment for Early Development of Component-Based Embedded Systems.* Marin Orlić, Aneta Vulgarakis, and Mario Žagar. 15th International Workshop on Component-Oriented Programming , Prague, Czech Republic, June, 2010.
- *Applying* Remes *Behavioral Modeling to PLC Systems.* Aneta Vulgarakis and Aida Čaušević. 22nd International Symposium on Information, Communication and Automation Technologies, IEEE, Sarajevo, Bosnia Herzegovina, October 2009.
- *Towards a Unified Behavioral Model for Component-Based and Service-Oriented Systems.* Aida Čaušević and Aneta Vulgarakis. 2nd IEEE International Workshop On Component-Based Design Of Resource-Constrained Systems, IEEE, Seattle, Washington, July, 2009.
- *Towards a Resource-Aware Component Model for Embedded Systems.* Aneta Vulgarakis. Doctoral Symposium of 33rd Annual IEEE International Computer Software and Applications Conference, IEEE, Seattle, Washington, July, 2009.
- *A Component Model Family for Vehicular Embedded Systems.* Tomáš Bureš, Jan Carlson, Séverine Sentilles, and Aneta Vulgarakis. 3rd International Conference on Software Engineering Advances, IEEE, Sliema, Malta, October 2008.
- *A Classification Framework for Component Models.* Ivica Crnković, Michel Chaudron, Séverine Sentilles, and Aneta Vulgarakis. 7th Conference on Software Engineering and Practice in Sweden, Göteborg, Sweden, October 2007.
- *A Model-Based Framework for Designing Embedded Real-Time Systems.* Séverine Sentilles, Aneta Vulgarakis, and Ivica Crnković. Work-In-Progress track of the 19th Euromicro Conference on Real-Time Systems, Pisa, Italy, July 2007.

MRTC reports

- *Connecting ProCom and* Remes. Aneta Vulgarakis, Séverine Sentilles, Jan Carlson, and Cristina Seceleanu. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-244/2010-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, May, 2010.
- *ProCom: Formal Semantics.* Jagadish Suryadevara, Aneta Vulgarakis, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-234/2009-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, March, 2009.
- Remes*: A Resource Model for Embedded Systems* Cristina Seceleanu, Aneta Vulgarakis, and Paul Pettersson. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-232/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, October, 2008.
- *ProCom – the Progress Component Model Reference Manual, version 1.0.* Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, and Aneta Vulgarakis. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-230/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, June 2008.
- *Towards Component Modelling of Embedded Systems in the Vehicular Domain.* Tomáš Bureš, Jan Carlson, Séverine Sentilles, and Aneta Vulgarakis. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-226/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April 2008.
- *Progress Component Model Reference Manual - version 0.5.* Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, and Aneta Vulgarakis. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-225/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April 2008.

1.4 Research Methodology

Depending on the kind of problem to solve and the context of the problem, different research methodology can be used. Research methods and research methodology are two terms that are often interchangeably used.

Strictly speaking, there is a slight difference between the two. Research methods aim to find solutions to research problems and they describe the concrete ways in which one could solve a given problem. Example research methods are: conducting experiments, testing, surveys, interviews, lessons learned, critical analysis of the literature and the like. We refer the reader to [57] for a summary of computing research methods. On the other hand, the Merriam-Webster dictionary defines methodology as a "a body of methods, rules, and postulates employed by a discipline: a particular procedure or set of procedures". In other words, methodology is the general plural term for all the individual research methods one has chosen, but there are certain types of methodologies which encompass and use specific methods e.g., quantitative/qualitative methodologies.

In our view, a research process describes the stages for conducting a research; it starts with defining a problem, and ends with proposing a solution for that problem. During the research process, one may use one or combine several research methods in order to address a certain research goal. The use of one or more research methods to address a certain research goal may create several research results. The research process that is used in this thesis is presented in Figure 1.1. It consists of four main stages as follows: identification of a general research problem, identification of a refined research problem, studying and addressing the refined problem and validation. As such, the process begins with identification and formulation of a general research problem from embedded systems practice, and the ultimate goal is to provide a solution to this practical problem. The solution is obtained in a research setting by refining and narrowing down the general problem, expressing the refined problem in a form of an overall research goal, addressing the overall research goal, and finally validation. Solving the research problem is not a straightforward process but an iterative one, allowing feedbacks between stages. First the overall goal is decomposed into smaller research goals, which are clarified, formulated, studied, refined, and even sometimes left aside. When the research results are mature enough, we move to the validation stage that makes us examine the validity of our research results. In using this research process, the validation of the results is crucial in both research and industry settings. If the validation stage fails, the research goals and results need to be revisited, improved, polished, and if necessary discarded.

We have considered the general research problem, the need to ad-

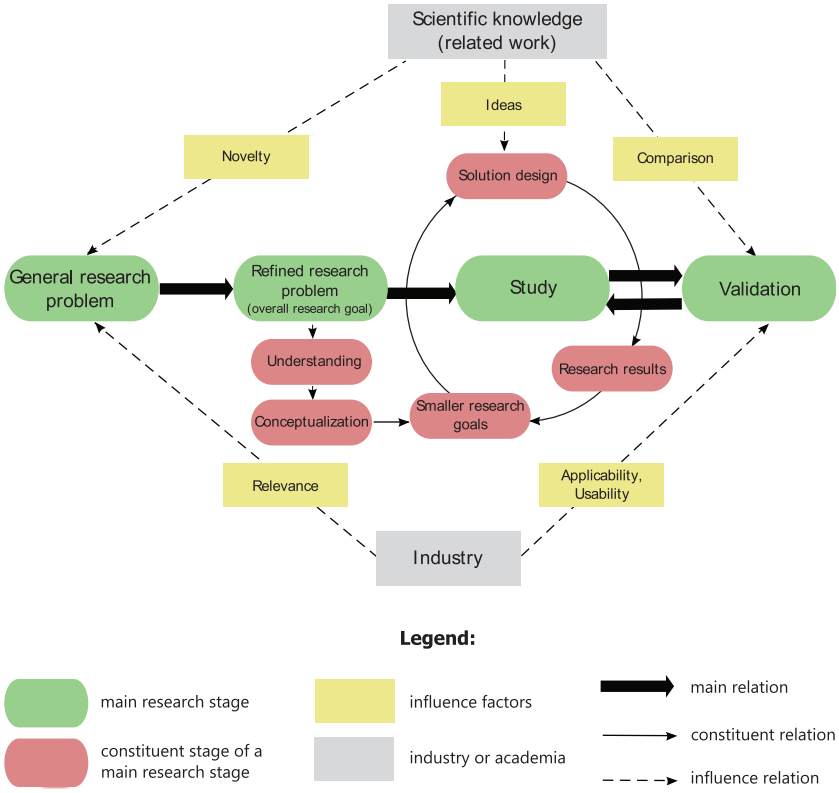


Figure 1.1: Overview of the applied research process.

dress the complexity and resource limitations of embedded systems in a structural way and ensure predictability during early stages of system development, and have transferred the problem to a research setting (see Section 1.1). In order to understand the problem both from an industrial and also scientific perspective, we have performed information gathering and studied the state of the art and state of the practise covering previous work done on the research problem. In scientific research, the role of previous work is to give a background for the research problem, and especially explicate the industrial relevance and scientific novelty of the research. During this stage we have used the research method that

is close to the so called *critical analysis of literature* [110] method. This method is a historical one that aims to provide an exhaustive summary of literature relevant to a research problem, by collecting and analyzing data from published materials. The analysis part provides the opportunity to draw conclusions from a broad range of approaches. We have performed our literature review in several iterations, and we have discussed the concluded results. In difference to the traditional critical analysis of literature, we have not identified a list of databases for searching related work, and have not classified the papers covering the related work according to their citation indexes. The investigation of the related work has resulted in two papers: paper B and paper I (see Section 1.3). As a result we have studied several (embedded systems') component models and a number of frameworks that model and estimate resource usage of embedded systems.

On this basis we have moved to the next stage of our research process - studying the refined research problem. During this stage we have used the *proof of concept* (also known as proof of principle) research method [47]. It involves creating solutions, methodologies, concepts, and techniques in an iterative manner. Note that this research method has a lot in common with software development [79], as in software development the goal is to create a working software system.

Our studying research stage has included several iterations where the research results have been improved through discussions and analysis. First we have conceptualized the refined research problem, expressed it as an overall research goal. Then we have decomposed the overall research goal into smaller research goals, presented in Section 1.1. After that, we have moved to addressing the smaller research goals by developing solutions, presenting achieved research results and comparing these research results with the research goals. In developing our solutions we have drawn ideas from the related work. In papers A, D, E, F, G and H we have presented our research results on developing a resource-aware design framework encompassing modeling and formal analysis of component-based embedded systems. We have proposed a language for component-based design of control-intensive embedded systems (ProCom), and a resource-aware behavioral language for describing component's and system's functional and extra-functional behavior (Remes).

The last stage of our research process is validation. Out of the many existing validation techniques [94], in our research, we use validation

by *persuasion*, *analysis*, and *example* validation. Firstly, we give an explanation and persuade the reader that it is reasonable to use our resource-aware framework in addressing the general research problem. Secondly, by developing examples and performing formal analysis we show how the research results work in practise and whether they can be found satisfactory. According to Shaw [94] the validation described in this thesis covers *toy-*, as well as *slice of life examples*. A toy example presents a simplified example, which might have been motivated by reality, where as a slice of life example is a system that the author has developed. As such, our research results have been illustrated on simple yet relevant "toy examples", presented in papers D, G, and H. Accordingly, in paper H we have exemplified the ProCom component model on an electronic stability control system of a car. Further, in paper A and G, we have performed a small case study demonstrating the principles of our resource modeling and analysis approach. The case study have been conducted on an abstracted version of the internal design of a temperature control system for heat producing reactor. In paper D, we have exemplified our resource-aware framework on a turntable example system, which we modeled as a collection of ProSys components that we have connected to their associated behavioral Remes models. Finally, in paper C, we have showed how to model extra-functional behavior, and verify the resulted behavioral models on a slice of life component-based Ericsson Nikola Tesla (ENT) telecommunications system. The salient point of our model, which enables its validation, is the fact that we have built it by using the timing and resource values extracted from the actual prototype implementation of the ENT system. The Remes behavioral language and the associated analysis techniques have been compared with the related work and have shown to be applicable for the development and analysis of the ENT system.

1.5 Thesis Outline

The outline of the rest of the dissertation is as follows.

- **Chapter 2 - Background** introduces basics in the areas of component-based development, and formal modeling and analysis of software systems. Section 2.1 discusses concepts of components, component-based systems and component models. Section 2.2 gives an

overview of (priced) timed automata and model-checking, as they will be used throughout this thesis.

- **Chapter 3 - ProCom: A Component Model for Embedded Systems** proposes a two-layer component model for design and development of control-intensive distributed embedded systems. The upper layer - ProSys - is presented in Section 3.3.1, and the lower layer - ProSave - is described in Section 3.3.2. Section 3.3.3 defines the relation between the two layers. The formal execution semantics of ProCom (Section 3.4) is defined by using a finite state machine (FSM) underlying formalism with notions of urgency, timing and priority, in which the semantics of each ProCom element is defined as a translation relation from ProCom to the FSM language. The ProCom model and its formal execution semantics are illustrated through a number of interesting examples.

- **Chapter 4 - Remes: A Behavioral Model for Embedded Systems** introduces the behavioral modeling language Remes for formal modeling and analysis of embedded resources, such as storage, energy, communication, and computation (see Section 4.1). Section 4.1.1 presents a classification of embedded resources, based on their rate of consumption over time, and the attribute of being referable, or not. Section 4.3 reveals the transformation rules for translating Remes modes into priced timed automata. Section 4.2 shows how a number of important resource analysis problems can be formalized in the framework of (multi-)priced timed automata. Finally, Section 4.4 demonstrates the principles of Remes on a temperature control system for a heat producing reactor.

- **Chapter 5 - Integrating ProCom and Remes** proposes a way of mapping the ProCom component interface onto the entry and exit variables of Remes modes (see Section 5.1), such that the two models become connected. We exemplify the concepts of mapping ProCom component interfaces and Remes modes on two examples: a temperature control system, where the architecture of the system is modeled in ProSave (see Section 5.3), and on a turntable drilling system, where the system architecture is modeled in ProSys (see Section 5.4). The packaging of a ProCom component and its Remes behavioral model together is done through a general attribute framework (see Section 5.2), which will only be shortly

described since it is out of the scope of this thesis.

- **Chapter 6 - The REMES Tool-chain** presents the tool-chain for the REMES language, which can be used for the construction and analysis of embedded system behavioral models.
- **Chapter 7 - Case Study: Ericsson Nikola Tesla Demonstrator** presents a case study where REMES is applied to model and analyze a telecommunication system by Ericsson Nikola Tesla (see Section 7.4). The ProCom component model is used to model the architecture of the ENT demonstrator, as shown in Section 7.3.
- **Chapter 8 - Related Work** gives a brief survey and relates the contributions presented in the thesis to relevant research, subdivided into two sections. Section 8.1 covers the state of the art of component models for embedded systems. Section 8.2 glances through several representative frameworks that model and estimate resource usage of embedded systems, pointing out advantages and limitations.
- **Chapter 9 - Conclusion** ends our dissertation with concluding remarks, enumerates the limitations of our results and lists future research directions.

Chapter 2

Background

This chapter introduces important technical concepts used throughout the remainder of this thesis. It provides an introduction to component-based development (Section 2.1) and to formal models and analysis techniques (Section 2.2). For more information on component principles and technologies, we refer to [36, 97], and for details on formal models and analysis techniques to [20, 35, 107].

2.1 Component-Based Development

The key principle of component-based development (CBD) [36, 97] is to build software systems by reusing existing software units, termed components, in much the same way as standard components are used in electronics or mechanics: integrated circuits, switches, etc. It is a promising approach for efficient software development, facilitating well defined software architectures and reuse.

With CBD it is possible to divide large and complex software systems into smaller, less complex modules. These modules can be decoupled from each other and thus be implemented in parallel by different developers, independently of each other's work. Therefore, development time is reduced. Virtually, reliability is increased because components which have been tested thoroughly and worked good for one system may be reused in another system. The extra time and effort required for selecting, evaluating, adapting, and integrating components is mitigated by avoiding the much larger effort that would be required to develop such

components from scratch. Another advantage is that software systems which consist of several modules are more flexible and maintainable than monolithic software systems.

In CBD the smallest functional building unit is a *component*. The idea behind components originates from a paper published by M.D. McIlroy [82] at a NATO conference in Garmisch in 1968 about the idea of mass-produced software components. However, since McIlroy's paper, component definitions and notions advanced in various, and in the same time contradictory directions. Up until today there is no generally accepted definition of what a component is. An early and commonly cited definition from Szyperski [97], which focuses on the key characteristics of components, states that:

> *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

This definition implies that in order a component to be deployed independently, a clear distinction between the environment and other components is required. A component must have clearly specified interfaces and the component's implementation must be encapsulated in the component and not be directly reachable from the environment. The definition inclines that components should be delivered in binary form, and that deployment and composition should be performed at run-time. Regardless of its generality, it was shown that Szyperski's definition does not fully cover a wide range of component-based technologies (e.g., those which do not support contractually specified interface or independent deployment). Further, embedded systems require optimal utilization of hardware (which in many cases has limited resources), and a predicable behavior, rather than flexibility at run time. A static compilation of components into an image is proven to be more efficient and more accurate than dynamic uploading of components. For this reason in embedded systems components are usually expressed as models or source code.

The ongoing debates about components have led to another definition by Heineman and Councill [4]:

> *"A software component is a software element that conforms to a component model and can be independently deployed and*

> *composed without modification according to a composition standard."*

Here the emphasis has been shifted from a component to a *component model* (a component model determines what is and what is not a component). Heineman and Councill [4] define a component model as follows

> *"A component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution and deployment."*

This definition points out that a component model covers multiple facets of the development process, dealing with: (i) rules for the construction of individual components, and (ii) rules for the assembly of these components into a system.

A more general definition of a component and a component model is reported by Crnkovic and Chaudron in [34]:

> *"A software component is a software building block that conforms to a component model. A component model defines standards for (i) properties that individual components must satisfy, and (ii) methods, and possibly mechanisms for composing components."*

In the above definition, the term "component properties", is meant to include functional and extra-functional specifications of individual components. The term "composing components" is meant to include mechanisms for component interaction.

Nowadays, there exist many component models. Most of them either target general purpose desktop applications or large distributed systems, and are based on certain technological platforms (such as Enterprise Java Beans, DCOM). Several component models (e.g., Koala [101], SaveCCM [8], Pecos [108], Rubus [52], Robocop [78], BlueArX [67]) target design of embedded systems.

In general, a *component-based system* is a composition of components, where the components accept inputs from the environment and produce outputs. A component interacts with its environment through its interface. The interface explicitly describes the services that the component provides and the services that it requires from other components and its execution environment. Figure 2.1 shows a component-based system made of three components A, B and C that communicate through

their ports i.e., interfaces defined for these ports. The behavior of a component-based system can be inferred from the behavior of its components and its architecture. The interface of a component is used to access internal component behavior: when the component is activated the behavior is started, using the values read from the input interface at that time.

When CBD is applied in the domain of embedded systems, where applications are often safety-critical and subject to real-time constraints, it is of significant importance that reliable predictions of a system's functional and extra-functional behavior can be derived at design-time. The employed predictability analysis should guide the design and selection of hardware and software system components. Ideally, the behavior of a component should be the same regardless of the environment in which it is deployed, i.e., the other components in the system, but this is not straightforward to achieve for properties, such as timing, resource usage or reliability. Although the behavior modeling and analysis of an embedded system is very important it is often omitted in component models targeting embedded system design. Thus, there is a need to include behavioral modeling and analysis in embedded systems.

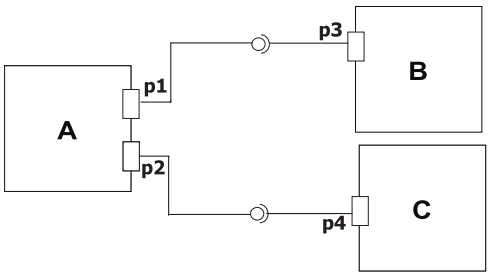


Figure 2.1: Example of a component-based system made of three components A, B and C, which are composed by connecting port p1 to p3, and p2 to p4.

2.2 Formal Models and Analysis Techniques

The act of *formal analysis* is that of rigorously exploring the correctness of system designs expressed as abstract mathematical models, most likely with the assistance of a computer. In this thesis, we consider two types of answers to formal analysis: "yes/no" answers as a result of verifying properties that can be either satisfied or not, but cannot be measured, and answers in form of numbers, in the sense that the formal analysis returns a computed number that might represent, in our case, the minimum/maximum value of the accumulated resource usage for reaching a given goal expressed as a reachability property for instance.

Today the best known formal analysis methods are *model-checking* and *theorem-proving*, both of which have sophisticated tool support and have been applied to non-trivial systems [23, 95]. Theorem-proving emphasizes highest assurance (theorems can only be created by a logical kernel, which implements the inference rules of the logic) and handling infinite-state systems, the main challenge being proof automation. Model-checking emphasizes automation, by relying on various efficient algorithms for deciding temporal logic formulas on finite state models, the main challenge being to reduce problems to a form in which they can be efficiently model checked. The advantage of model-checking of providing high level input languages that do not require expert knowledge of logics and support the modeling and checking of complex computer systems, and the highest degree of automation, justify our choice for model-checking as the verification paradigm.

To perform model-checking (see Figure 2.2), an automata *model of a system* describing the possible system behaviors is fed into a model-checking tool or a *verifier*, together with a desired property (*requirement*) typically expressed in a temporal logic. The tool then automatically traverses the system's state space in an exhaustive manner. If an invariant property is satisfied, the tool finishes the verification successfully, or if the invariant property is violated, it reports one of the traces that violates the property as a *counter-example* to the model. For reachability properties the opposite is true i.e., a trace is reported when the property is satisfied. Even if the system's desired behavior is satisfied, one can *refine* the model and reapply model-checking.

Model-checking has achieved huge success in industry for verifying hardware designs. Companies, such as IBM, Intel, Motorola, Siemens are having in-house model-checking groups. Despite these successes,

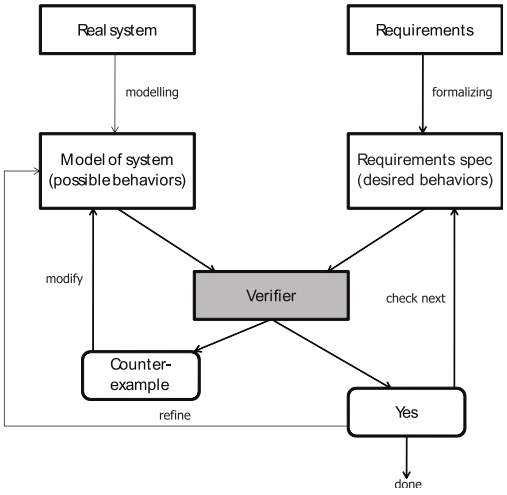


Figure 2.2: Verification methodology of model-checking [20].

formal analysis has not been widely used in the development of embedded systems. One possible reason is the lack of expertise of design engineers for constructing and understanding abstract models in an interactive environment formal specifications.

Due to the real-time requirements of embedded systems and the need to verify the models against them, the designer should be equipped with methods and tools that support modeling of real-valued variables, and the combination of discrete and continuous behaviors. The framework of timed automata is an established formal framework to support such needs. The Uppaal [2, 71] tool is one of the most popular and mature verification tools based on timed automata, and it is also used in this thesis. In the following, we recall the model of timed- and (multi) priced timed automata.

2.2.1 Timed automata

Timed automata have been proposed by Alur and Dill in the 1990s [11, 14], as a model for real-time systems. Uppaal extends the standard framework of timed automata, as it allows utilization of data variables.

In this thesis we use timed automata, as defined in UPPAAL [22–24].

A timed automaton (TA) is a timed extension of the finite-state automaton. A notion of time is introduced by a set of non-negative real numbers, called *clock* variables, which are used in clock constraints to model time-dependent behavior.

Let X be a finite set of clocks and V a finite set of all data (i.e., boolean, integer or array) variables. We use $\mathcal{B}(X)$ to stand for the set of formulas obtained as conjunctions of atomic constraints of the form $x_i \bowtie n$ or $x_i - x_j \bowtie n$, where $x_i, x_j \in X$, $n \in \mathbb{N}$, and $\bowtie \in \{<, \leq, =, \geq, >\}$. The elements of $\mathcal{B}(X)$ are called *clock constraints* over X. A clock constraint is downward closed if $\bowtie \in \{\leq, <, =\}$. Similarly, we use $\mathcal{B}(V)$ to stand for the set of non-clock constraints that are conjunctive formulas of $i \sim j$ or $i \sim k$, where $i, j \in V$, $k \in \mathbb{Z}$ and $\sim \in \{<, \leq, =, \neq, \geq, >\}$. We use $\mathcal{B}(X, V)$ to denote the set of formulas that are conjunctions of clock constraints and non-clock constraints.

TA consists of a finite set of *locations*, connected by *edges*. One of the locations is marked as initial. All clocks in TA start at zero, evolve continuously at the same rate, and can be tested and reset to zero. Each edge may have a *guard*, an *action* and an *assignment*. A guard is a finite conjunction over data constraints and clock constraints. An assignment is a comma separated list of expressions with a side-effect. It is used to reset clocks and set values of variables. We say that an edge is enabled if the guard evaluates to true and the source location is active. Locations are labeled with a downward closed clock constraints called *invariants*, which enforce that the location is left before they are violated.

Definition 1. ***(Formal Definition of a Timed Automaton).*** *A TA A is a tuple $(L, l_0, X, V, I, Act, E)$, where:*

- *L is a finite set of locations,*
- *l_0 is the initial location,*
- *X is a finite set of clocks,*
- *V is a finite set of data variables,*
- *$I : L \to \mathcal{B}(X)$ assigns invariants to locations,*
- *$Act = \Sigma \cup \{\tau\}$ is a finite set of actions, where Σ is a finite set of synchronizing actions, and $\tau \notin \Sigma$ denotes internal or empty actions without synchronization.*

- $E \subseteq L \times \mathcal{B}(X,V) \times Act \times R \times L$ *is a finite set of edges, where* $\mathcal{B}(X,V)$ *denote the set of guards and* R *denotes the reset set i.e., assignments to manipulate clock- and data variables.* ■

For an edge $e = (l, g, a, r, l') \in E$ we also write $l \xrightarrow{g,a,r} l'$. The label g is the guard of e, r is the reset set i.e., data- or clock assignment of e, and a is the action of e.

The semantics of TA is defined in terms of a state transition system. A state of TA depends on its current location and on the current values of its clocks. So, a state of a TA is a pair (l, u), where l is an active location, and $u : X \to \mathbb{R}_{\geqslant 0}$ is a clock valuation for which l evaluates to true. The initial state (l_0, u_0) puts the automation in its initial location l_0, where u_0 maps all clocks in X to zero i.e., all clocks are zero when entering the initial location initially. Intuitively, there are two kinds of transitions between states:

- *Delay transitions* are result of passage of time and do not cause a change of location. More formally, we have $(l, u) \xrightarrow{d} (l, u \oplus d)$, if $u \oplus d' \models I(l)$ for $0 \leq d' \leq d$. The assignment $u \oplus d$ is the result obtained by incrementing all clocks of the automata with the delay amount d.

- *Discrete transitions* are result of following an enabled edge in a TA. As a result, the destination location is activated and the clocks in the reset set are set to zero. More formally, a discrete transition $(l, u) \xrightarrow{a} (l', u')$ corresponds to taking an edge $l \xrightarrow{g,a,r} l'$ for which the guard g evaluates to true in the source state (l, u) and clock valuation u' of the target state is derived from u by resetting all clocks in the reset set, r, of that edge, such that $u' \models I(l')$.

Definition 2. ***(Run of a Timed Automaton).*** *A timed action is a pair* (t, a)*, where* $a \in Act$ *is an action taken by the automaton* A *after* $t \in \mathbb{R}_+$ *time units since* A *has started. Then a run of* A *with initial state* (l_0, u_0) *over a timed trace* $\xi = (t_1, a_1)(t_2, a_2) \ldots (t_n, a_n)$ *is a sequence of alternating delays and discrete transitions*

$$\xi = (l_0, u_0) \xrightarrow{d_1} \xrightarrow{a_1} (l_1, u_1) \xrightarrow{d_2} \xrightarrow{a_2} \ldots \xrightarrow{d_n} \xrightarrow{a_n} (l_n, u_n)$$

satisfying the condition $t_i = t_{i-1} + d_i$ *for all* $i \geq 1$. ■

To model a concurrent real-time system, several TA can be composed in an automata system. UPPAAL uses the CCS parallel composition operator [84] to build a system or a network of TA. CCS allows individual components to carry out internal actions (i.e., interleaving) as well as pairs of components to perform hand-shake synchronization.

Definition 3. ***(Network of Timed Automata).*** *Let $N = \{1, \ldots, n\}$ and let $A_i = (L_i, l_{0i}, X_i, V_i, I_i, Act_i, E_i)$ be a timed automaton for $i \in N$. A network of timed automata $A_1 || \ldots || A_n$ is defined as the parallel composition of n TA.* ■

For a network of timed automata, the synchronization actions play an important role. Particular automata in the network synchronize using channels and values can be passed between them using shared (global) variables. A state of a network of TA is defined by the locations of all automata in the network and the values of clocks and discrete variables. Let S be a set of channel names and out of S, there is a subset U of urgent channels on which timed automata should synchronize whenever possible. Then the set of synchronization actions of the network of timed automata is defined as $\Sigma = \{a?|a \in S\} \cup \{a!|a \in S\}$. A discrete transition with a synchronization action $a? \in \Sigma$ is only enabled if another automaton in the network simultaneously can perform a complementary action $a! \in \Sigma$. We use $name(a)$ to denote the channel name of a, defined by $name(a?) = name(a!) = a$. *Binary channels* are used to synchronize one sender with a single receiver, and *broadcast channels* are used to synchronize one sender with an arbitrary number of receivers. *Urgent channels* encode synchronization that must be taken when a transition becomes enabled, without delay. Clock guards are not allowed on edges synchronizing over urgent channels.

To restrict the behavior of a timed automaton, UPPAAL provides two special types of locations: *urgent* and *committed*. Time is not allowed to pass in both locations, but there is a difference between them as follows. When a location is urgent it must take the next transition as soon as this is possible i.e., without any delay. It does not rule out other actions happening. On the other hand, when a committed location is active, a transition from a committed location has to be taken immediately, and no other transition in other automaton can be taken in between. If such a transition does not exist or is not enabled, the system will deadlock.

The UPPAAL model checker supports verification of temporal properties, including safety and liveness properties, specified in a sub-set of

Timed Computation Tree Logic (TCTL) [12]. The simulator can be used to visualize counter examples produced by the model checker.

Illustrative example

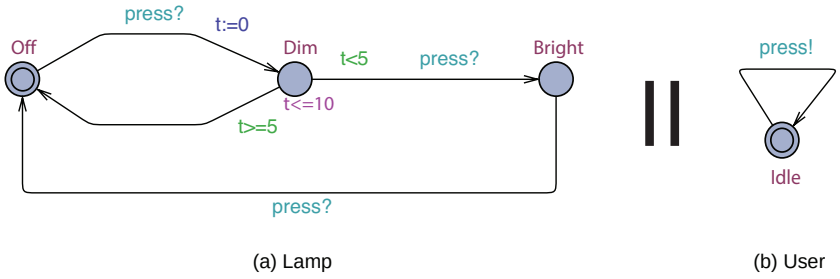


Figure 2.3: A timed automaton of a lamp and a user.

An example of a network of timed automata modeled in UPPAAL is shown in Figure 2.3. The network consists of an automaton of a lamp and an automaton of a user. The behavior of the lamp depends on when the user presses the on/off switch. The automaton of the lamp consists of three locations Off, Dim and Bright, and one clock t. The automaton starts at initial location Off. In case the user presses the switch the automaton of the lamp switches to location Dim and the clock t is reset, by the assignment t:=0. In location Dim the automaton can remain as long as the clock is smaller or equal to 10. However, if the user presses the switch of the lamp before 5 time units have elapsed then the automaton of the lamp switches to location Bright, in which it stays until the next pressing of the switch. Processes lamp and user synchronize via synchronization actions i.e., by sending and receiving events through channels. Sending and receiving via a channel press is denoted by press! and press?, respectively.

2.2.2 Priced timed automata

Priced (or weighted) timed automata [21, 73] extend timed automata with prices/costs on both locations and edges. The cost-rate for staying in a location represents the price/cost per time-unit for staying in that

location, whereas the cost labeling an edge represents the price/cost for taking the edge.

Definition 4. ***(Formal Definition of a Priced Timed Automaton).*** *A linearly Priced Timed Automaton (PTA) is a tuple* $(L, l_0, X, V, I, Act, E, P)$*, where* $(L, l_0, X, I, V, Act, E)$ *is a TA and* $P : (L \cup E) \to \mathbb{N}$ *is a cost function that assigns price-rates (or cost-rates) to locations and prices (or costs) to edges.* ■

Every run in a PTA has a global cost, which is the accumulated cost along the run of every delay and discrete transition. The cost is never tested in the automaton. In delay transitions,

$$(l, u) \xrightarrow{d,p} (l, u \oplus d)$$

the assignment $u \oplus d$ is the result obtained by incrementing all clocks of the automata with the delay amount d, and $p = P(l) * d$ is the cost of performing the delay. Discrete transitions

$$(l, u) \xrightarrow{a,p} (l', u')$$

correspond to taking an edge $l \xrightarrow{g,a,r} l'$. The cost $p = P((l, g, a, r, l'))$ is the cost associated with the edge.

Definition 5. ***(Run of a Priced Timed Automaton).*** *A timed trace* ξ *of a PTA is a sequence of transitions:*

$$\xi = (l_0, u_0) \xrightarrow{a_1,p_1} (l_1, u_1) \xrightarrow{a_2,p_2} \dots \xrightarrow{a_n,p_n} (l_n, u_n)$$

and the cost of performing ξ *is* $\sum_{i=1}^{n} p_i$*.* ■

For a given state (l, u), the *minimum cost* of reaching (l, u) is the infimum of the costs of the finite traces ending in (l, u). Dually, the *maximum cost* of reaching (l, u) is the supremum of the costs of the finite traces ending in (l, u).

In order to specify properties of PTA, the Weighted CTL (WCTL) logic has been introduced [32]. WCTL extends TCTL with resets and testing of cost variables. WCTL syntax is given by the following grammar:

$$\mathsf{WCTL} \ni \phi ::= \mathbf{true} \mid a \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{E}\,\phi\,\mathbf{U}_{\mathbf{P}\sim c}\,\phi \mid \mathbf{A}\,\phi\,\mathbf{U}_{\mathbf{P}\sim c}\,\phi$$

where a is an atomic proposition, $\mathbf{P}$ is a cost function, c ranges over $\mathbb{N}$, and $\sim \in \{<, \leq, =, \geq, >\}$. Here, $\mathbf{A}$, and $\mathbf{E}$ are the universal and existential path quantifiers of WCTL, respectively, and $\mathbf{U}_{\mathbf{P}\sim\mathsf{c}}$ is the "until" temporal modality. The temporal operators $\mathbf{F}$ (i.e., "eventually") and $\mathbf{G}$ (i.e., "always") are derived in the usual way. We interpret formulas of WCTL over labeled PTA, that is, PTA having a labeling function that associates with every location l a subset of atomic propositions. The satisfaction relation of WCTL is defined over configurations of the labeled PTA. We refer the reader to [32] for semantic details of WCTL.

Multi Priced Timed Automata

Multi priced automata (MPTA) [32, 73] are extension to priced timed automata in which a timed automation is augmented with more than one cost variable. In the case of two costs associated with a PTA, the *minimal cost reachability problem* corresponds to finding a set of minimal cost pairs (p_1, p_2) (both p_1 and p_2 are minimized) reaching a goal state. Since the costs contributed from the individual costs can be incomparable, when, e.g., for the costs of two traces, say (p_1, p_2) and (p'_1, p'_2), $p'_1 < p_1$ and $p_2 < p'_2$, the solution is a set of pairs, rather than a single pair. In this setting, the minimal cost reachability problem is to find the set of incomparable pairs with minimum cost, reaching the goal state. Dually, the *maximization cost* problem is defined as finding the set of incomparable pairs with maximal cost reaching the target location, or to conclude (∞, ∞) if the target location is avoidable in a path that is infinite, deadlocked, or has a location in which it can make an infinite delay. A specific problem is the *optimal conditional reachability problem*, in which one of the costs should be optimized, and the other bounded by an upper/lower bound. We refer the reader to [73] for a thorough description of optimization problems in MPTA.

In this thesis, the framework of (multi) priced timed automata is used for formally analyzing resource consumption in embedded systems.

Illustrative Example

Switching on a lamp and letting it burn uses energy, therefore in Figure 2.4 is depicted a priced timed automaton of the lamp elaborated earlier. The energy consumption is modeled by using costs. A special variable cost can be increased explicitly on an edge by an update, or

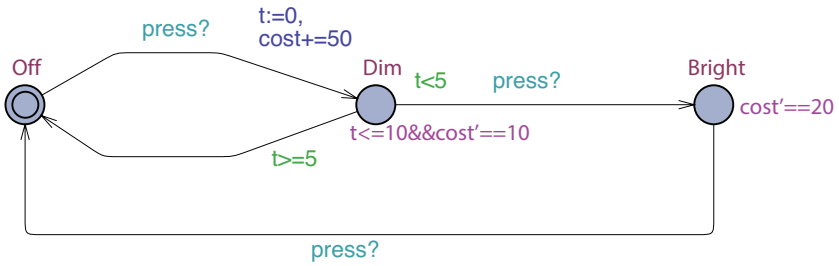


Figure 2.4: A priced timed automaton of a lamp.

implicitly by specifying a rate in a location. Guards and invariants are, however, not allowed to refer to the cost variable. The switch of the lamp from location Off to Dim is labeled with an update cost+=50, indicating that the cost is 50 for switching on the lamp. In locations Dim and Bright we have the cost rates cost'== 10 and cost'== 20, respectively, which indicate that the energy consumption is 10 and 20 units per time unit in the respective locations. When staying in these locations, cost is increasing linearly with time, with rate 10 and 20, respectively.

Chapter 3

ProCom: A Component Model for Embedded Systems

In this chapter, we describe a component model called ProCom, intended for development of embedded software for control-intensive distributed systems, which are a special class of embedded systems that primarily perform real-time controlling tasks. They can be found in many products, for example vehicles, automation systems, or distributed wireless networks.

The ProCom component model [33,93,96,106] was developed as part of the PROGRESS project [58] that distinguishes three key activities in the development: design, analysis and deployment. The *design* activity addresses the architectural description of the system, following the component model presented in this chapter, as well as the implementation of individual components. *Analysis* is carried out to ensure that the developed system meets its dependability requirements and constraints in terms of timing and resource usage. In particular, an early analysis based on models (see Chapter 4 for resource-aware behavioral analysis), budgets and estimated properties, provides means to explore and evaluate different design alternatives. The *deployment* activity concerns the allocation of functionality to the physical nodes of the system, which can have a significant impact on the overall system performance. This

activity also includes the gradual progression from design entities to executable units, such as processes and tasks.

This chapter is oriented towards design, focusing on the component model which serves as the underlying formalism for that activity. The two supplementary activities (analysis and deployment), although outside the scope of this chapter, have significantly influenced many aspects of the component model.

The remainder of the chapter is organized as follows. In Section 3.1 we identify the key concerns and requirements coming from the class of control-intensive distributed embedded systems, and in Section 3.2 we describe how the design choices of our ProCom component model were influenced by these requirements. In Section 3.3 we present each element of ProCom in detail together with its informal execution semantics. In Section 3.4 we describe how the architectural elements of the ProCom component model have been given a formal execution semantics.

3.1 Key Requirements for Development of Control-Intensive Distributed Embedded Systems

The class of control-intensive distributed embedded systems has specific requirements that need to be reflected by a component model.

Motivating example

As an example demonstrating the specific concerns of control-intensive distributed embedded systems, we consider the electronic systems of Volvo XC90 car focusing on an anti-lock braking system (ABS) in particular. The role of an ABS is to improve the braking performance by preventing the wheels from locking. Figure 3.1 shows the complex physical system architecture of Volvo XC90 that consists of approximately forty computational nodes (ECUs), connected to a number of different networks, and Figure 3.2 shows the ABS subsystem architecture. Functionally, the ABS is fairly independent from other subsystems, although it shares some information about the state of the vehicle with other subsystems. In its simple form the ABS includes rotation sensors physically placed on or close to the wheels, a brake valve actuator, and an ECU that

includes control software. Typically the ECU includes a set of software components that together provide the service.

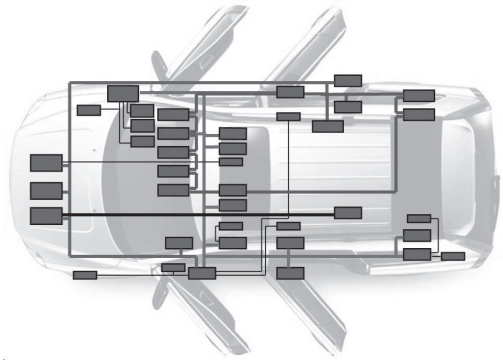


Figure 3.1: Overview of the electronic system architecture of Volvo XC90.

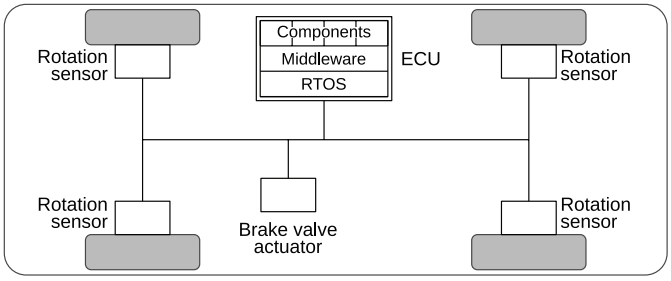


Figure 3.2: The ABS subsystem architecture.

Component granularity

In an embedded system, components constituting big parts of the system are different from those responsible for a small part of some low-level control task. Components at different granularity have different needs in terms of execution model, communication style, synchronization, etc.,

but also with respect to the kind of information that should be associated with the component and the type of analysis that is appropriate. Large components (e.g., the ABS of a car) tend to be active (i.e., with their own threads of activity), and encompassing complex functionality. Since the communication between these components often involves communication over a network (e.g., a CAN bus), it is typically realized by asynchronous messaging.

On the other side of the scale there are smaller components responsible for a part of some control functionality, such as computing the deviation of a measured value from the desired value, or for communication with a single sensor or actuator. Since they represent composable low-level functional blocks, they typically do not posses their own threads. Also, the communication between them is much more tightly synchronized since most of the communication at this level is between components located on the same physical node.

For the component model this means that it should support different types of components with respect to their size (i.e., granularity) and semantics. Having these multiple levels of components it is vital to establish the relation between them, for example allowing a big component to be built of several small components.

Coupling between the software and the target platform

The coupling between the software and the target platform is quite high in an embedded system. The hardware is typically very restricted and the software is tailored and optimized specifically for that particular hardware and real-time operating system.

In our example, it is known a priori that the ABS will be distributed over at least five physical nodes, dictated by the physical location of the wheel speed sensors and the actuators. We would also typically be able to make some assumptions about the nature of these nodes and the network between them, based on experience from other systems. However, the final choice of hardware might be made later, as well as the decision whether the main functionality of the ABS will be allocated to a dedicated node or if it will share a node with other subsystems.

This reality of system development being interwoven with target platform specification is however in contrast to the main goals of CBD – component reusability. Components can no longer be developed completely independently without some knowledge of where they are to be

deployed.

To address these issues, the component model should be able to take into account the target platform, while not sacrificing the reusability of components.

Levels of abstraction

The development of an embedded system or a subsystem typically starts with requirements specification (in a form of use-cases), domain diagrams and basic sketches of the system. These abstract models are then gradually detailed and refined to eventually end up with an implementation. With regard to CBD, the concept of a component throughout the whole development spans between vague and incomplete specification to a very concrete one. In early stages of development, a component may be seen as a functional unit with little specification. For e.g., it may have known behavior, but unknown inner structure. Later on, the component may be detailed and in the end implemented.

With respect to the component model this means that it is necessary to support coexistence of both fully implemented components having well known inner structure, and early design components with unknown inner structure.

3.2 ProCom Design Choices

In designing ProCom, we have aimed at addressing the requirements described above. To handle the differences related to the granularity scale, we distinguish two levels of granularity ProSys and ProSave, which are addressed by different concepts in the component model. ProSys covers the upper part of the granularity scale (i.e., the "big" units), and thus ProSys components are active, relatively independent, and communicate by message passing via explicit message channels. In ProSave, the lower level of granularity, components correspond to constituents of the control functionality (i.e., the "small" units), and therefore they are passive and more tightly coupled. As described later in this chapter, these two component types have different semantics and are also modeled in different ways. Also, the two layers can not be arbitrarily mixed, but they are still integrated since ProSave can be used to detail the internals of an individual ProSys component.

To address the second concern related to the coupling between the software and the target platform, we distinguish between component- and system development. Components define the functionality and specify their requirements on the target platform. We allow components to express their partial assumptions about the platform (e.g., the minimum available memory, required operating system functionality). The system is modeled as a single top-level system component. The detailed specification of the hardware and the platform, as well as the allocation of components to physical nodes, are given by separate models connected with deployment- i.e., they are not part of the component specification. Moreover, similarly to SaveCCM [8] and Robocop [78], a component is considered as "a whole", i.e., a collection gathering all the information needed and/or specified at different points of time of the development process. That means a component comprises requirements, documentation, source code, various models (e.g., behavioral and timing), predicted and experimentally measured values (e.g., performance and memory consumption), etc., thus making a component a unifying concept throughout the development process.

ProCom covers several levels of abstractness, since components at an early stage can be specified as black boxes, then gradually behaviorial models can be associated with them, and finally a concrete source code implementation can be given.

3.3 ProCom: Syntax and Informal Execution Semantics

The following Sections 3.3.1 and 3.3.2 detail the concepts and informal execution semantics of ProSys and ProSave, respectively, and Section 3.3.3 shows the link between the two layers. The complete specification of ProCom is available in [33].

3.3.1 ProSys - the upper layer

In ProSys, a system is modeled as a collection of communicating *subsystems*. Subsystems execute concurrently, and communicate by asynchronous messages sent and received at typed output and input *message ports*. The asynchronous communication style is suitable at this level of granularity, since it allows transparent communication between sub-

systems independently of whether they reside on the same or different physical nodes.

Connecting subsystems

A system consists of a collection of subsystems and connections from output to input message ports. Message ports are not connected directly, but via *message channels* – explicit design entities representing data that are of interest to more than one subsystem. Multiple message ports (output- as well as input ports) can be connected to the same message channel, allowing n-to-n communication (see Figure 3.3). A benefit of these explicit message channels is that information about a message, such as precision, format and whether it should be available to diagnostic tools, can be associated with the message channel instead of with a port where the message is produced or consumed. This way, it can remain in the design even if, for example, the producer is replaced by another subsystem. Also, since message channels can be introduced before any producer or receiver of the message has been defined, it permits early modeling of the run-time data managed by the system.

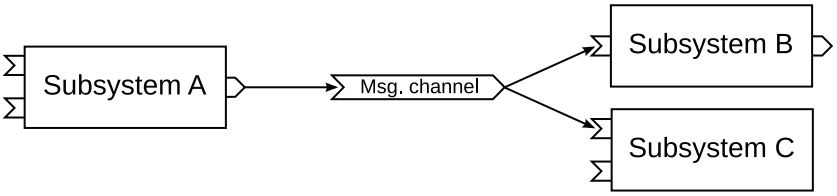


Figure 3.3: Three subsystems communicating via a message channel.

Primitive and composite subsystems

A subsystem can be built out of smaller subsystems, thus making ProSys a hierarchical component model. Contrasting such *composite subsystems*, a *primitive subsystem* is realized either directly by non-decomposable units of implementation (such as COTS or legacy subsystems), or by further decomposition in ProSave as described in Section 3.3.2.

3.3.2 ProSave - the lower layer

The ProSave layer targets the detailed design of subsystems allocated to a single physical node and interacting with the system environment through sensors and actuators. On this level, components provide an abstraction of tasks and control loops found in control systems.

A subsystem is constructed by hierarchically structured and interconnected ProSave *components*. These components are encapsulated and reusable design-time units of functionality, with clearly defined interfaces to the environment. As they are designed mainly to model simple control loops and are usually not distributed, this component model is based on the pipes-and-filters architectural style with an explicit separation between data and control flow. The former is captured by *data ports* where data of a given type can be written or read, and the latter by *trigger ports* that control the activation of components. ProSave follows the push-model for data transfers and an input data port always contain the latest value written to it.

Services, groups and ports

A ProSave component is of a collection of *services*, each providing a particular functionality. The services of a component are triggered individually and can execute concurrently, while sharing only data. A service consists of an *input port group* containing the activation trigger and the data required to perform the service, and a set of *output port groups* where the data produced by the service will be available. Figure 3.4 illustrates these concepts.

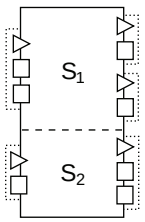


Figure 3.4: External view of a ProSave component with two services; S_1 has two output groups and S_2 has a single output group. Triangles and boxes denote trigger- and data ports, respectively.

ProSave components are *passive*, i.e., they do not contain their own execution threads and cannot initiate activities on their own. So each service remains in a passive (idle) state until its input trigger port has been activated. Once activated, the data input ports are read in one atomic operation and the service switches into an active state where it performs internal computations and produces data on its output ports. Note that a service may produce parts of the output at different points in time, which is useful when the delivery of some data is more time-critical then the rest. When each output group has been activated, the service returns to the idle state again. The data and triggering of an output group of a service are always produced at the same time. Multiple activations of an output group during a single activation of the service is not permitted.

Input data ports can receive data while the service is active, but it would only be available the next time the service is activated. The strict "read-execute-write" semantics of a service simplifies analysis by ensuring that once a service has been activated it is functionally (although not temporally) independent from other components executing concurrently.

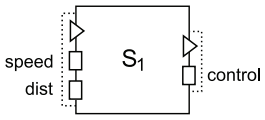
In ProCom, the components, their services, ports, subcomponents, etc., can be annotated with various functional and extra-functional characteristics, represented as *attributes*. The attributes may be as simple as numbers (e.g., static memory usage of a component), but also as complex as intricate models. ProCom uses the Attribute Framework [92] that provides a systematic way of managing and integrating extra-functional properties, during the development of a component, or a system.

Primitive and composite components

The functionality of a component can either be realized by code (*primitive component*), or by interconnected subcomponents (*composite component*). For primitive components, in addition to a function called at system startup to initialize the internal state, each service is implemented as a single non-suspending C function. Figure 3.5 shows an example of the header file of a primitive component.

Connecting and composing components

Turning back to the external and internal view of a ProSave component, the internal view distinguishes between *primitive* and *composite* components. Composite components internally consist of *subcomponents*,




```
typedef struct {
  int *speed;
  float *dist;
} in_S1;
typedef struct {
  int *control;
} out_S1;
void init();
void entry_S1(in_S1 *in, out_S1 *out);
```

Figure 3.5: A primitive component and the corresponding header file.

connections and *connectors*. A *connection* is a directed edge which connects two ports (output data port to input data port of compatible types and output trigger port to input trigger port) whereas *connectors* are constructs that provide detailed control over the data- and control flow. The existence of different types of connectors and the simple structure of components makes it possible to explicitly specify and then analyze the control flow, timing properties and system performance.

The set of connectors in ProSave, selected to support typical collaboration patterns, is extensible and will grow over time as additional data- and control-flow constructs prove to be needed. The initial set includes connectors for *forking* and *joining* data or trigger connections, or *selecting* dynamically a path of the control flow depending on a condition. Figure 3.6 shows a typical usage of the selection connector together with *or* connectors.

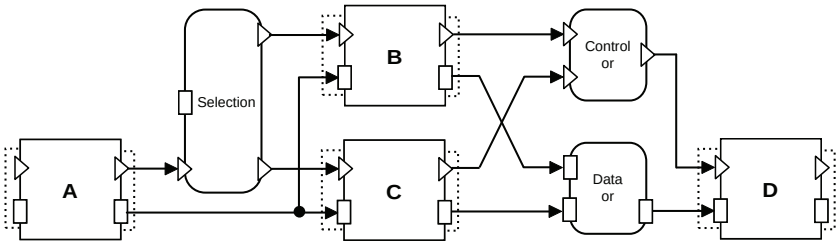


Figure 3.6: A typical usage of selection and or connectors. When component A has completed executing, either B or C is executed, depending on the value at the selection data port. In either case, component D is executed afterwards, with the data produced by B or C as input.

ProSave follows the push-model for data transfers and the triggered service always uses the latest value written to each input data port. Since ProSave components can not be distributed, the migration of data or trigger over a connection is loss-less and atomic. However, the trigger signals are not allowed to arrive to any port before all data have arrived to all end destinations. This also holds in case when the data passes through a connector. The formalization of data- and trigger connections we describe in Section 3.4.3.

3.3.3 Integration of layers – combining ProSave and ProSys

The integration of the two ProCom layers allows a primitive ProSys subsystem to be further specified using ProSave. This is done similarly to how composite ProSave components are defined internally – as a collection of interconnected components and connectors – but with some additional connector types to specify periodic activation of ProSave components and to map between the architectural styles (message passing in ProSys and pipes-and-filters in ProSave). Note that these additional connectors can only appear on the topmost level inside a primitive subsystem, i.e., they are not allowed inside composite ProSave components.

Periodic activation is provided by the *clock* connector, with a single output trigger port which is repeatedly activated at a given rate. All clocks are assumed to follow a common conceptual time, but it is not assumed that all clocks produce their first activation simultaneously. To achieve the mapping from message passing to trigger and data, and vice versa, the message ports of the enclosing primitive subsystem are treated as connectors with one trigger port and one data port, when seen from inside the subsystem. An input message port corresponds to a connector with output ports, and whenever a message is received by the message port, the message data is written to the data port and the trigger port is activated. Oppositely, an output message port corresponds to a connector with an input trigger and input data ports. When triggered, the current value of the data port is sent as a message.

In addition to strictly periodic activation, ProCom supports aperiodic activation initiated by external devices. To enable interaction of component-based applications with hardware devices (such as sensors and actuators) ProCom includes a new type of component named *device component* that has the same interface and semantics as all software

components. In difference to typical ProCom components, device components do not provide the ability for the developer to explicitly specify their realization. Since this contribution is outside the scope of the thesis, for more information about managing hardware devices in ProCom, we refer the reader to [74].

3.3.4 Example: An Electronic Stability Control System

To illustrate the ProCom component model we use as an example an electronic stability control (ESC) system of a car. The ECS handles sliding caused by under- or oversteering, and contains an ABS subsystem (described earlier) and a traction control (TCS) subsystem, which prevents the wheels from spinning when braking or accelerating. If a car would be modeled in ProCom, the ESC would be one of many subsystems at the top level, together with subsystems for engine control, airbags, climate control, etc. The ESC can be modeled as a composite ProSys subsys-

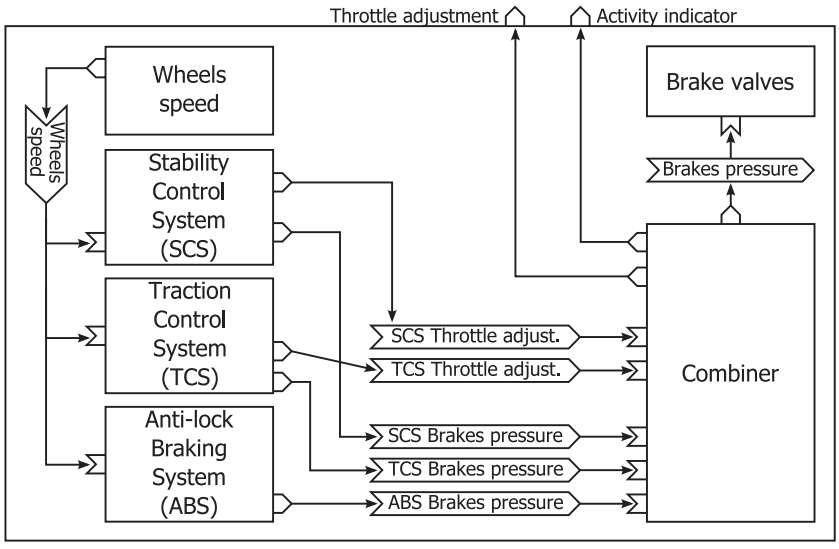


Figure 3.7: The ESC is a composite subsystem, internally modeled in ProSys.

tem, as shown in Figure 3.7. Inside we find subsystems corresponding to specific parts of the ESC functionality (SCS, TCS and ABS). We suppose that the TCS and ABS subsystems are reused from previous versions of the car, while SCS has been added to cope with under- and oversteering. These three subsystems compute responses based on their internal sensors and the speed of individual wheels, which is provided by a dedicated subsystem. Finally, the "Combiner" subsystem is responsible for combining the output of the ABS, TCS and SCS subsystems. The overall brakeage and throttle responses are forwarded to the "Brake valves" subsystem to regulate the braking pressure, and delegated to subsystems outside of the ESC, respectively. The SCS acts as a primitive subsystem on the ProSys level, meaning that it can be realized either directly by code or modeled in ProSave. We have chosen the latter – see Figure 3.8. The SCS contains one periodic activity performed at a frequency of 50 Hz, expressed by a clock connector. Once started, it reads the data from yaw-, lateral- and steering wheel angle sensor. Based on their outputs and the speed of individual wheels (obtained from the latest "Wheels speed" message) it computes the actual direction of the vehicle and the desired direction indicated by the steering wheel. After both computation components have finished, the "Slide detection" component compares their results (i.e., the actual and desired direction) and decides whether or not stability control is required. The last component in the chain computes the actual response of the SCS, which consists of adjustments of brakeage and acceleration.

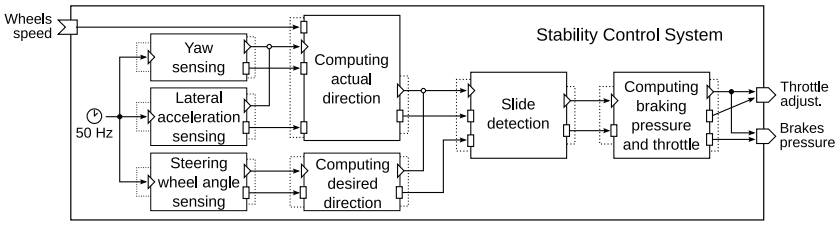


Figure 3.8: The SCS subsystem, modeled in ProSave.

3.4 Formal Execution Semantics of the ProCom Component Model

In this section we describe the formal execution semantics of the ProCom layers using a higher-level formal language. The semantics description language (see Section 3.4.1) is FSM-like i.e., presents an extension of finite state machine (FSM) notation with necessary constructs as required for formal execution semantics of the elements of ProCom. In Section 3.4.2 we formally define the language. Using this language in Section 3.4.3 we present the actual formalization of selected ProCom architectural elements.

3.4.1 Formalism and graphical notation

Definition 6. ***(Formal Definition of the FSM Language).*** *Let V be a set of variables, G a set of boolean conditions (or* guards*) over V, B the set of booleans, A a set of variable updates, and I a set of intervals of the form $[n_1, n_2]$, where $n_1 \leq n_2$ and n_1, n_2 are natural numbers. The FSM language is a tuple $\langle S, s_0, T, D \rangle$, where S is a set of states, $s_0 \in S$ is the initial state, $T \subseteq S \times G \times B \times B \times A \times S$ is the set of transitions between states, in which $B \times B$ represent priority and urgency (described below), and $D : S \rightarrow I$ is a partial function associating delay intervals with states.* ■

The FSM language relies on a graphical representation that consists of the usual graphical elements, that is, states and transitions labeled with guards, priority, urgency, and updates, see first two columns of Figure 3.9. A transition can be either *urgent* or *non-urgent*, and it can have *priority* or no priority. As shown in Figure 3.9, a transition may be decorated with the non-urgency symbol *, and/or the priority symbol ↑. Note that, a transition that is not annotated with * is urgent. A state can be associated with a delay interval, which is graphically located within the state circle.

Intuitively, the execution of an FSM starts in the initial state. At a given state, an outgoing transition may be taken only if it is *enabled*, i.e., its associated guard evaluates to `true` for the current variable values. If from the current state, more than one outgoing transition is enabled, one of them is taken non-deterministically, and prioritized transitions are preferred over non-prioritized transitions. In case all enabled outgoing

transitions of a state are non-urgent, it is possible to delay in the state. On the other hand, if there are any outgoing urgent enabled transitions, one of them must be taken immediately. Thus, the notions of priority and urgency avoid unnecessary non-determinism among enabled transitions, clarifying the modeling aspects and possibly improving the performance of formal analysis. A state that is associated with a delay interval [n_1, n_2] may be left anytime between n_1 and n_2 time units after it is entered.

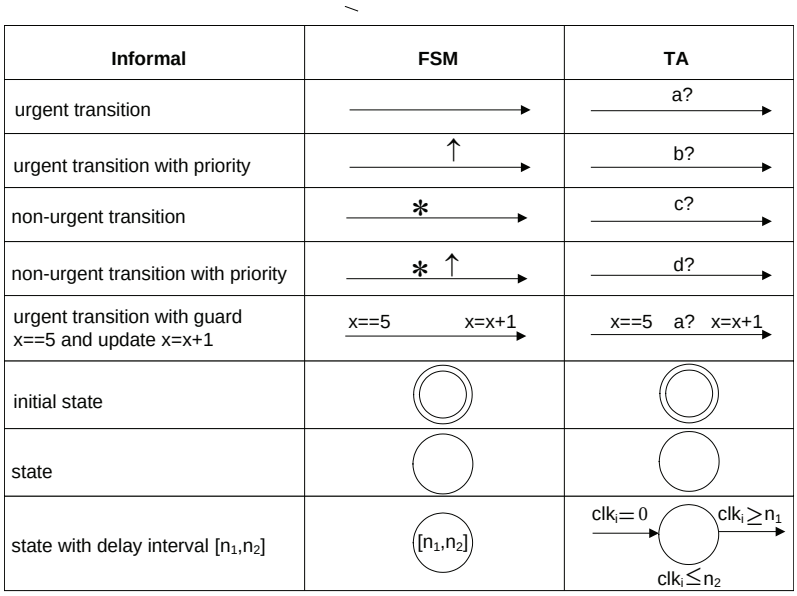


Figure 3.9: The graphical notation of the FSM elements and their translation into TA.

In order to form a system, FSMs may be composed in parallel. The semantic state of the composed system is the combined states and variable values of the FSMs. The notions of urgency and priority are applied globally, and time is assumed to progress with the same rate in all FSMs.

3.4.2 Formal semantics of the FSM language

We formally define the semantics of our FSM language using timed automata (TA) with priorities [38] and urgent transitions [23] as a seman-

tic domain. The translation of each FSM element to TA is depicted in Figure 3.9. The FSM language has four kinds of transitions: urgent transition, urgent transition with priority, non-urgent transition, and non-urgent transition with priority. In TA we introduce four channels: a, b, c, and d. Channels a and b are urgent, and channels b and d have higher priority than channels a and c. Accordingly we map the transitions of FSMs into TA edges labeled with the appropriate channels, as defined in Figure 3.9. The translated TA edges need a timed automaton offering synchronization on the complementary channels (e.g., $a!$ complementary to $a?$), depicted in Figure 3.10.

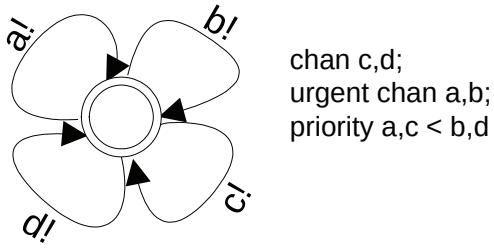


Figure 3.10: The automaton used for synchronization.

Each FSM state results into a TA location. For every FSM with delay states, a clock clk_i is introduced. Accordingly, an FSM state with delay interval $[n_1,\ n_2]$ is translated into a corresponding TA location with invariant $clk_i \leq n_2$. The clock is reset on all ingoing edges and the guards of all outgoing edges are conjuncted with $clk_i \geq n_1$.

The system represented by a composition of FSMs can be translated into a network of TA in two steps. First, each FSM is translated into a timed automaton and then all TA are composed into a network together with the automaton of Figure 3.10.

3.4.3 Formal execution semantics of selected ProCom architectural elements

In the formalization, each data and message port is represented by a variable with the same type as the port. The variables are storing the latest value written to the ports, respectively. Likewise, a trigger port is represented by a boolean variable determining the activation of that

port. Ports of composite components are represented by two variables, corresponding to the port viewed from outside and from inside. Accordingly, in the ProCom formalization we assume the following set of shared variables through which the FSMs communicate:

- v_{d_i}: variable associated with a data port d_i of corresponding type.
- v_{t_i}: boolean variable associated with a trigger port t_i indicating whether the port is triggered, default false.
- v_{m_i}: variable associated with a message port m_i of corresponding type.
- v'_{d_i} and v'_{t_i}: internal variables for ports of composite components, corresponding to port variables v_{d_i} and v_{t_i}, respectively.

We let ε be the null value of any type indicating that no data is present on a data or message port.

The complete formalization of ProCom is available in [96]. The semantics of all ProCom elements is defined as a translation to the FSM language, and the semantics of an entire ProCom system is defined by the parallel composition of FSMs for the individual constructs.

In the following, we chose the most representative, and semantically challenging, architectural elements of ProCom, and present their formalization. The elements are: services, connections, components, clocks, message ports and message channels.

Services

Assume a ProSave component with one service, say S_1 and let S_1 consist of one input port group and two output port groups (Figure 3.11 (a)). The informal execution semantics of a service in ProSave we have described in Section 3.3.2. The formal execution semantics of a service, in this case, S_1, we describe below and show in Figure 3.11 (b).

Let $w1$ and $w2$ be boolean variables corresponding to the output port groups, respectively; the variables indicate whether the respective group has been activated or not. By associating boolean variables with the output port groups, we ensure that the groups are written only once during an execution instance of a service. While being in an Execute state a service may yield into the following error scenario:

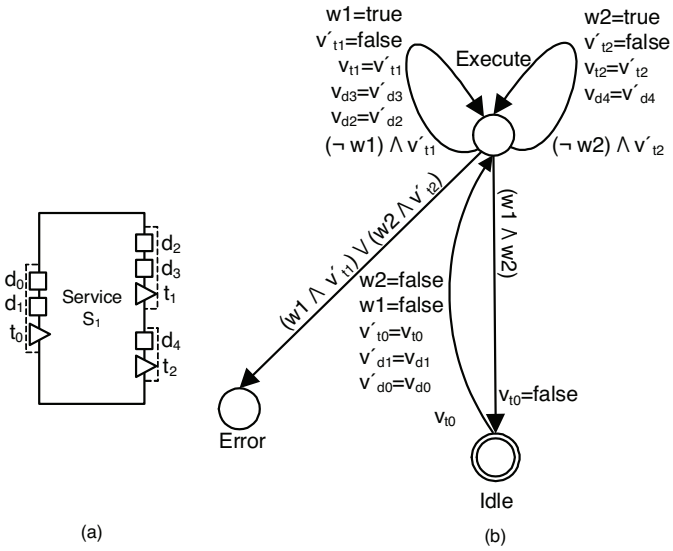


Figure 3.11: (a) A ProSave service S_1 and (b) its formal execution semantics.

- During execution, a service might try to activate an already activated output port group. This problem is captured by the state Error.

As such, the formal execution semantics, ensures the informal execution semantics described in Section 3.3.2 i.e., the triggering and data of a service is always produced atomically and each of the service output groups can be activated only once before the service returns to the Idle state.

Data and trigger connections

Let us assume the following modeling scenario: three components A, B and C, are interconnected via a Data-Fork connector (see Figure 3.12). The Data-Fork connector is used to split data connections, so data written to the input data port is forwarded to the output ports. When the component A has finished executing, the component B should start executing. However, since the input trigger port of the component B is

directly connected to the output trigger port of the component A, while the data is not transferred directly, but via a connector, there is a risk that the trigger signal may reach the component B before the data has arrived. A scenario in which trigger might arrive before data is prohibited by the formalization that we present bellow.

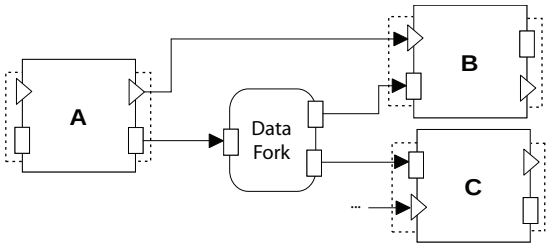


Figure 3.12: Example of a critical modeling of data and trigger transfer in ProCom.

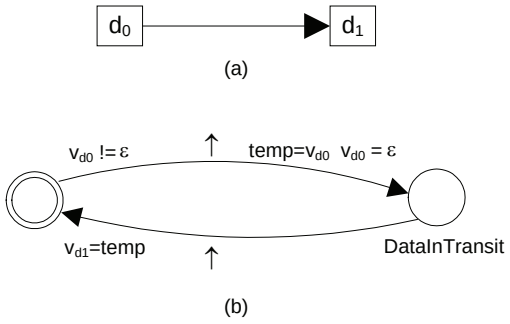


Figure 3.13: (a) A ProSave data connection and (b) its formal execution semantics.

The formal execution semantics of ProSave connections is presented in Figure 3.13, for data connection between two data ports d_0 and d_1, and in Figure 3.14, for trigger connection between two trigger ports t_0 and t_1.

To ensure that data is transferred prior to trigger, and to avoid undesirable consequences otherwise, the transitions in the FSM formalism

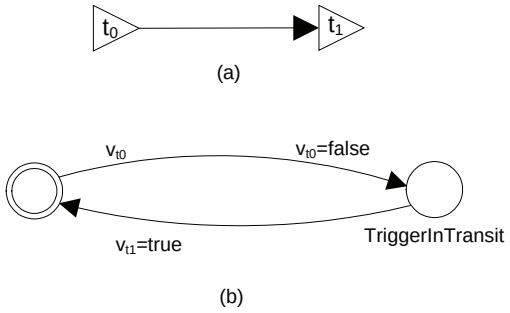


Figure 3.14: (a) A ProSave trigger connection and (b) its formal execution semantics.

(Figure 3.13) are associated with priority in the case of data connections. This is also the case in the semantics of all connectors that forward data (detailed in [96]).

Component hierarchy

As we have described in Section 3.3.2, the functionality of a ProSave component can be implemented by a single C function (primitive component) or hierarchically by inter-connected internal components (composite component).

In early stages of development, a component may still be a black box with known behavior, but unknown inner structure. Later on, the component may be detailed and in the end implemented. However, all components follow the same execution semantics. In an early stage of development, when only the behavior of the component is assumed to be known, it is the responsibility of the behavior model to signal the end of execution, and to take care of the internal variables (data and trigger) of a component accordingly. In a later stage of development, when the inner structure of a composite component is known, its formalization is handled by the inter-connected subcomponents. In this case, we assume that there is a virtual controller in charge of signaling when the internal trigger of a component has become false i.e., all subcomponents have returned to the idle state. Consequently, in both cases, the internal variables are left to be modified by the behavior, code or inner realization, but the external variables of a component are always handled by the

semantics of a service (defined in Section 3.4.3). This emphasizes the fact that, from an external observer's point of view, there is no difference between early design black box components and fully implemented components.

Linking passive and active components

By definition, ProSave components are passive and they communicate via data exchange and triggering. As mentioned in Section 3.3.3, ProSave components can be used to define the internals of an active ProSys subsystem with some additional connector types: clocks (see Figure 3.15 (a)) and input- and output message ports (see Figure 3.16 (a) and Figure 3.17 (a), respectively).

A ProSave component can be activated by receiving a periodic trigger with appropriate period. The formal execution semantics of a ProSave clock with period P we show in Figure 3.15 (b). Thus, the formal execution semantics complies to the informal execution semantics of a clock, described in Section 3.3.3.

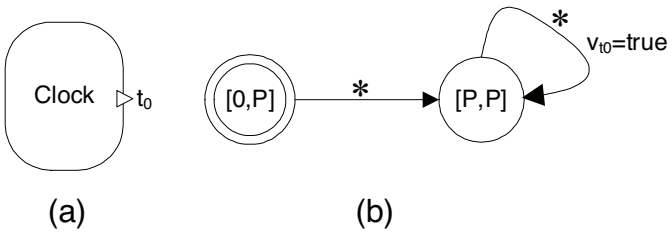


Figure 3.15: (a) A ProSave clock with period P and (b) its formal execution semantics.

Message ports bridge the gap between the two communication paradigms: pipes-and-filters in ProSave and message passing in ProSys. Each message port acts as a connector with a trigger and data port that may be connected to other ProSave elements. Whenever a message is received, the input message port writes this message data to the output data port, and activates the output trigger. Similarly, whenever the trigger from an output message port is activated, the output message port sends a message with the data currently present on its input data port.

We assume the following:

- todata(): is a function that translates messages into data.
- tomessage(): is a function that translates data into messages.

Given the above, the formal execution semantics of an input message port and an output message port can be described as in Figure 3.16 (b) and Figure 3.17 (b), respectively.

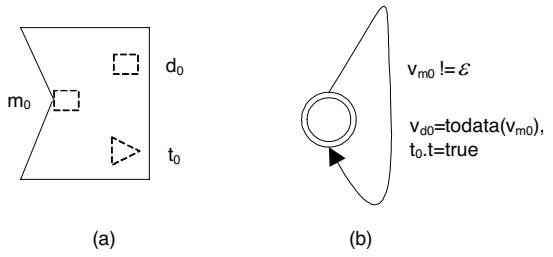


Figure 3.16: (a) A ProSave input message port and (b) its formal execution semantics.

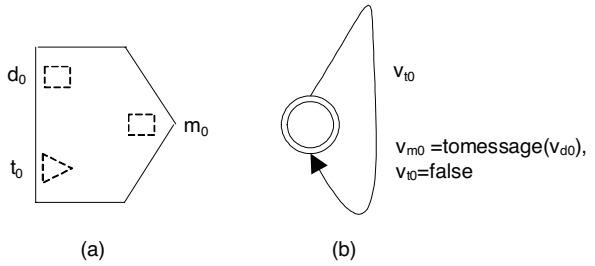


Figure 3.17: (a) A ProSave output message port and (b) its formal execution semantics.

Message channels

Let *ch* be a message channel that connects one output message port m_0 and two input message ports m_1 and m_2 (presented in Figure 3.18 (a)), and assume the following.

- $buff_i$: is an unbounded buffer of messages, with operations 'insert()' and 'remove()'.

- A message can not be removed more than once, and only a message that was previously inserted into the buffer can be removed from it.
- The operation 'remove()' removes a message from the buffer and writes it to corresponding port variable v_{mi}.

Then, the formal execution semantics of connecting input/output message port(s) to a message channel we present in Figure 3.18 (b). Note that the defined formal execution semantics represents the weakest possible behavior of a message channel such that any further refinement e.g., buffer as FIFO etc., must satisfy the above defined behavior. The detailed behavior is undefined regarding individual message values (e.g., ordering, duration of stay in buffer etc).

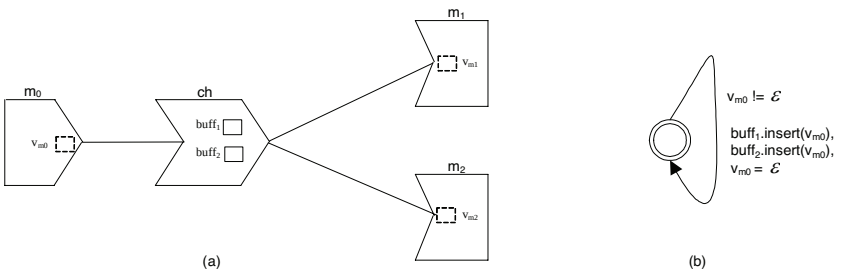


Figure 3.18: (a) Graphical representation of connecting message ports to a message channel and (b) formal execution semantics.

3.5 Summary

In this chapter, we have presented ProCom, a component model for control-intensive distributed embedded systems. The model takes into account the most important characteristics of these systems and consistently uses the concept of reusable components throughout the development process, from early design to deployment. A characteristic feature of the domain we consider is that the model of a system must be able to provide both a high-level view of loosely coupled subsystems and a low-level view of control loops controlling a particular piece of hardware. To address this, ProCom is structured in two layers (ProSys

and ProSave). At the upper layer, ProSys, components correspond to complex active subsystems communicating via asynchronous message passing. The lower layer, ProSave, serves for modelling of primitive ProSys components. It is based on primitive components implemented by C functions, and explicitly captures the data transfer and control flow between components using a rich set of connectors. To illustrate the ProCom component model we have used as an example an electronic stability control system of a car.

In order to rigorously describe the semantics of the ProCom elements, and to provide support for formal analysis, we have introduced a small, but powerful FSM language. The FSM language builds on standard FSM, enriched with finite domain integer variables, guards and assignments on transitions, notions of urgency and priority, as well as time delays in locations. Its formal semantics is expressed in terms of TA with priorities and urgent transitions, as shown in Section 3.4.2. The FSM language has graphical appeal and it is simpler than the corresponding TA model, as it abstracts from real-valued variables and synchronization channels. Moreover, thanks to the TA formal semantics, the FSM models of ProCom systems can be analyzed in a dense-time underlying framework, as well as in a discrete-time one, since TA has been recently given a sampled semantics [5]. Hence, tools, such as UPPAAL, can be employed for early-stage verification of ProCom models, whereas discrete-time model-checkers, such as DTSpin [29], could be used for later-stage analysis, as a sampled time semantics is closer to the actual software or hardware system with a fixed granularity of time, and can become appealing at later stages of design.

Using the FSM language, we have described in detail how the design constructs for services, data and trigger connections, component hierarchies, passive and active components, and message channels of ProCom have been formalized in this manner. These elements we have deliberately chosen, since they represent the different types of design elements in the language, and expose the encoding techniques used in the ProCom – FSM translation.

Chapter 4

REMES: A Behavioral Model for Embedded Systems

We pointed out in Chapter 1 that embedded systems are not only coupled by the physical world, but that they are also constrained by the physical capacities of their underlying hardware and/or software platforms. Hence, these systems are typically resource constrained, and analysis of the embedded system's resource usage at an early design stage is extremely desirable. First, it allows for carrying out a potentially large number of design experiments, without increasing cost. Second, it may guide designers in making correct decisions, such as selecting the right components from a repository, or choosing among various admissible design models. Both of these point to a need for a modeling language and analysis techniques that will treat resources as first class entities.

In this chapter, we introduce the model REMES [91] (see Section 4.1) for formal modeling and analysis of embedded resources, such as storage, power, communication, and computation. The model is annotated with both discrete and continuous resources. It is in fact a state-machine based behavioral language with support for hierarchal modeling, continuous time, and a notion of explicit entry and exit points, making it suitable for component-based modeling. The analysis of REMES-based systems (see Section 4.2) is centered around a weighted sum in which

the variables represent the amounts of consumed resources. We describe a number of important resource related analysis problems, including feasibility, trade-off, and optimal resource-utilization analysis. In order to be able to formally analyze REMES compositions, in Section 4.3, we semantically translate REMES models in the framework of priced timed automata. To illustrate the approach, in Section 4.4, we describe an example in which REMES has been applied to model resource usage of a temperature control system.

4.1 REMES: Syntax and Execution Semantics

In order to model resources at a high-level of abstraction in an effective way, we need to understand their behavior. Hence, in this section, we first define the resources of interest and then introduce the REMES model intended for functional, timing, as well as resource-wise description of embedded systems.

4.1.1 Classes of resources

We consider resources as global quantities of finite size. We refer to the *consumption* of a resource c as being the accumulated resource usage up to some point in time, whereas the derivative of c, denoted $\dot{c}$, is the rate of consumption over time. Resource consumption can be of *discrete* or *continuous* nature, depending of how the respective resource is used over time. Since we are aiming for a general, yet comprehensive resource characterization, we also consider resources as being either *referable* or *non-referable*, depending on the way they are assigned. A classical example of referable resource is memory. Memory can be dynamically allocated, deallocated, addressed, and manipulated during run-time. However, modeling and checking properties of referable memory is outside the scope of this thesis.

Taking all these into consideration, Table 4.1 shows three identified resource classes and their characteristics of interest. Resource consumption for resources that belong to class C is continuous, contrary to the discrete resource consumption nature for the resources from class A and B. The consumption of the CPU can be modeled by a discrete variable, denoting the number of accumulated clock ticks, or by a continuous vari-

Resource Class	Characteristics
A (e.g., memory)	discrete: $\dot{c} = 0$ referable
B (e.g., CPU, bandwidth)	discrete: $\dot{c} = 0$ non-referable
C (e.g., CPU, energy, bandwidth)	continuous: $\dot{c} = n$, $n \in \mathbb{Z}$ non-referable

Table 4.1: Resource classes/characteristics

able, which encodes the processor load (that is, the derivative describes, e.g., how many tasks are starting execution, every time unit). Accordingly, CPU may be in class B or C (same applies to bandwidth). In our approach, any continuous resource can be assigned either discretely, or consumed in time at some rate, whereas a discrete one can only be allocated/deallocated by an instantaneous assignment statement. Only the resources from class A are referable and can be dynamically manipulated. Please note that we do not claim that such a resource classification is unique, but that it rather serves our modeling purposes.

4.1.2 Introducing REMES

Our **RE**source **M**odel for **E**mbedded **S**ystems (REMES), that we have introduced in [91], describes the resource-wise behavior of interacting embedded components that communicate both with one another, as well as with the environment (e.g., through a trigerring signal). REMES is inspired by the modeling language CHARON [13], used for specifying embedded systems as communicating agents. The salient point of REMES is introducing resources as primitive types in the language, as well as additional constructs that facilitate modeling (real-time) component-based system behavior, and are deemed popular to system designers. There are also other syntactic and semantic differences from CHARON, which will be apparent in the following.

In REMES, the internal behavior of a component is described by a *mode*. We call a mode *atomic* if it does not contain any submode, and *composite* if it contains a number of submodes (see Figure 4.1). In addition, there is also a special type of mode called *non-lazy* whose semantics will be described in the following. The hierarchy can be of arbitrary

depth, although in this chapter we show the implementation of a two-level hierarchy only. Like in CHARON, the data is transferred between modes via a well-defined *data interface*, that is, typed global variables, whereas the (discrete) control is passed through a well-defined *control interface* consisting of *entry* and *exit* points. Observe, in Figure 4.1, that the entry and exit points are intuitively labeled, respectively. A mode, be it atomic or composite, has a "run-to-completion" semantics, so it cannot be interrupted from execution, and internal loops are not allowed as such; however, the possible interleavings between computations of one composite mode, and another's can be controlled via *history variables*, whereas iteration is possible by introducing two kinds of exit points: Write, and Exit. Assuming an architectural model supporting the system's description, the Write point serves for modeling both internal mode computations that are resumed until the Exit is reached, which signals termination, but also continuously active behaviors (corresponding to active components), which might never terminate. A mode describing active behavior can also be exited through the Exit point. Assuming a composite mode that has finished one execution round by visiting the Write point, then the history variable, can specify the submode that the execution of that composite mode should resume from. As such, the history variable h of a composite mode M may contain the names of the submodes of M as values.

The variables of mode M are partitioned into *local* variables, ($\mathsf{L_M}$), and *global* variables ($\mathsf{G_M}$). *Interface* variables, $\mathsf{I_M} \subseteq \mathsf{G_M}$, are a subset of global variables, and model variables that are mapped to an interface port and come from the assumed architectural system model. Variables can be of types boolean, natural, integer, float, array, string, list, clock that specifies continuous variables evolving at rate 1, and of special type resource, which are nonnegative real-valued variables that model continuous resource behavior. Resource variables are of type: processor load (CPU), energy (eng), bandwidth (bdw), memory (mem), and communication ports. The global variables are shared among all system modes and are in turn partitioned into *read* variables, $\mathsf{Rd_{GM}}$, and *write* variables, $\mathsf{Wr_{GM}}$, such that $\mathsf{G_M} = \mathsf{Rd_{GM}} \cup \mathsf{Wr_{GM}}$, and $\mathsf{Rd_{IM}} \subseteq \mathsf{Rd_{GM}}$, $\mathsf{Wr_{IM}} \subseteq \mathsf{Wr_{GM}}$.

Read/Write Variable Access. The local variables of M, $\mathsf{L_M}$, can not be read or written by other modes, the set $\mathsf{Wr_{GM}}$, written by M can be read by other modes, whereas the set $\mathsf{Rd_{GM}}$ may be written by other modes. The sets $\mathsf{Wr_{GM}}$, and $\mathsf{Rd_{GM}}$, respectively, need not be disjoint;

concurrent access to common write variables of modes can be regulated by specifying certain synchronization protocols in the REMES model, as described in Section 4.1.3.

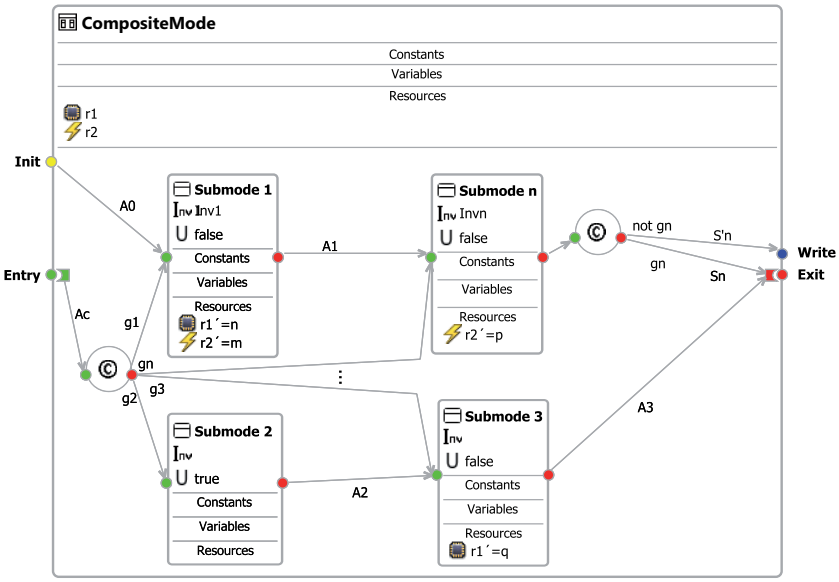


Figure 4.1: A REMES Composite Mode.

The atomic modes Submode 1, Submode 3, and Submode n in Figure 4.1 are annotated with their respective resource-wise continuous behavior, assuming that the corresponding component is consuming resources $(\mathsf{r1} : \mathsf{CPU}, \mathsf{r2} : \mathsf{eng})$ belonging to class C. Such consumption is expressed by the first derivatives of the typed resource variables, respectively, that is, $\mathsf{r1}', \mathsf{r2}'$, which give the rates at which the composite mode consumes the resources in time, depending on the executing submode.

For a composite mode, the control flow is given by a set of directed lines, called *edges*, which connect the control points of the submodes, or of the composite mode and its submodes. For example, in Figure 4.1, the composite mode takes the edge labeled A_0, in order to enter Submode 1, after initialization, and similarly, edge labeled A_1 to further enter Submode n.

REMES supports two types of actions, *delay/timed* actions and *discrete* actions. A delay action describes the continuous behavior of the mode, and its execution does not change the mode. The delay/timed actions are not visible in a REMES model, but they are usually constrained by the differential equations that annotate the modes, and they represent the solutions of such equations. Observe that Submode 2 has an urgent flag (decorated with letter U) equal to true, meaning that such a mode exits right-away after its activation, without any delay. Such modes are called *urgent*. On the other hand, discrete actions are instantaneous actions, and they are represented as annotations of the edges. Executing a discrete action results in a mode change, by taking the outgoing edge starting from the mode's exit point.

A discrete action $\mathsf{A} = (\mathsf{g}, \mathsf{S})$ is a statement list prefixed by a boolean expression, with g called the *action guard*, and S the *action body*, that is, the statement (assignment, conditional statement etc.) or sequence of statements that must be executed once the corresponding edge has been taken. We say that a discrete action A is *enabled*, hence it could be executed, if its corresponding guard g evaluates to true at some point in time. A discrete action is called *always enabled* if its guard always holds, and *empty* if its body does not change any of the mode variables (in such cases, the action body can be omitted).

In addition, one needs to specify for how long a (sub)mode is executed, so an *invariant*, e.g., $\mathsf{Inv\,1}, \ldots, \mathsf{Inv\,n}$, that is, a predicate over continuous variables, captures such a timing constraint. Once the invariant stops to hold, the mode is exited by taking one of the outgoing edges. Observe, in Figure 4.1, that Submode 3 does not contain any invariant (and has an urgent flag false) to specify how long it is allowed to delay in that mode. Such modes, we call *non-lazy* modes. Time is allowed to pass in a non-lazy mode until at least one of the guards of the outgoing discrete actions evaluates to true, in which case that action is executed right away. As such, in order to ensure the exit of a non-lazy mode, the disjunction of the action guards associated to the outgoing edges of that mode should always eventually become true.

Similar to Statecharts [53], REMES provides a *conditional connector* (depicted by C in Figure 4.1), which allows the selection of an outgoing edge, out of two or more possible ones, via the guarding boolean conditions (guards $\mathsf{g}_1, \mathsf{g}_2, \ldots, \mathsf{g}_n$) of the discrete actions that correspond to the edges exiting the conditional connector. For a discrete action to be possibly executed, the component must be in the right mode and the

corresponding guard must evaluate to true. If none of the guards evaluates to true, then no discrete action is executed and the component remains in its current mode, performing delay actions. If more than one action guards are true, then one of the enabled discrete actions could be executed non-deterministically.

Besides ordinary edges, for clarity, we distinguish for the composite mode the *init edges* that connect the Init point of the composite mode with the entry point of a submode (e.g., annotated with action A_0 in Figure 4.1), and *write edges* that end up in the Write point (e.g., decorated with $(\mathsf{not}\,\mathsf{g}_\mathsf{n}, \mathsf{S}'_\mathsf{n})$).

The control points of submodes are connected by edges, such that the termination of the internal behavior of the composite mode is ensured. Internal cycles/loops are forbidden as such, yet iterative internal computation of a mode is modeled by execution rounds that end up in the Write exit, until termination occurs and the computation finishes by visiting the Exit point, provided that the respective action guard holds. Whenever an execution of a REMES composite mode would return to an already visited submode, the composite mode is exited through its Write point, and the control state of that mode is recorded into its local history variable. Then the composite mode is automatically reentered, and the control state of that mode is restored according to the value of the history variable.

Definition 7. ***(Formal Definition of a Composite Mode).*** *A mode* M *is defined as a tuple:*

$$(\mathsf{SM}, \mathsf{V}, \mathsf{In}, \mathsf{Out}, \mathsf{E}, \mathsf{RC}, \mathsf{Inv}, \mathsf{CC}),$$

where: SM *is the set of submodes,* V *is the set of variables,* In *is the set of entry control points,* Out *is the set of exit control points,* E*, the set of edges,* RC*, the set of resource constraints that define the admissible values for the consumption rates of the involved resources in class C,* Inv *is the set of invariants, and, finally,* CC *is the set of conditional connectors. For the submodes of* M*, the following condition should hold:*

$$\mathsf{G}_\mathsf{SM} \subseteq \mathsf{L}_\mathsf{M} \cup \mathsf{G}_\mathsf{M},$$

for a local variable of a mode to be accessible only in its submodes, and not anywhere else. ■

In the particular case of the sets SM and CC being the empty set i.e., $\mathsf{SM} = \emptyset$ and $\mathsf{CC} = \emptyset$, we get the definition of an atomic mode.

Informal Execution Semantics of a Mode. The top-level mode (of a composite mode), which is activated when a corresponding event is received, enters execution for the first time through the special Init entry point, after initializing the global variables, accordingly. After that, the mode is re-entered through control point Entry.

A mode can execute either a *discrete step*, by a discrete action, or a *continuous step*, via a delay action, with such steps alternating as dictated by the urgency of the mode. When executing a continuous step, the mode follows a continuous path that satisfies the resource constraints (RC). When the mode invariant is violated, the mode must execute an outgoing discrete step. A discrete step of a mode is a finite sequence of discrete steps of the submodes, that is, a sequence of executing discrete actions. A discrete step begins in the current mode and ends either at the entry point of a submode, or when it reaches the current mode's exit point, meaning that the current mode has passed control to some other mode.

The fact that a mode can pass control is ensured by the *closure* construction: each exit point of a mode is either connected to the exit point of the composite mode, or deterministically connected to an entry point of another mode that eventually leads to the composite mode's exit.

For example, in Figure 4.1, the execution of CompositeMode proceeds as follows: after initialization, the discrete step corresponding to A_0 is executed, after which a sequence of continuous steps is executed, until the invariant Inv 1 fails to hold; alternatively, in case A_1's guard evaluates to true, the mode could take a discrete step and entry Submode n. Next, a similar sequence follows, while the mode executes Submode n. When Inv n does not hold anymore, the mode takes a new discrete step corresponding to either discrete action $(\mathsf{not}\, g_n, S'_n)$ if $(\mathsf{not}\, g_n)$ holds, or to discrete action (g_n, S_n) if g_n holds. In case $(\mathsf{not}\, g_n)$ holds, CompositeMode is exited through its Write point, meaning it will be automatically reentered without waiting for an activation from the outside. In case g_n holds, CompositeMode will be exited through its Exit point, and will wait for reactivation from the environment. The next time when the control is passed to CompositeMode, a discrete step corresponding to A_C is taken and the selection of a possible path is made through the conditional connector, etc.

4.1.3 Composition of REMES models

Sequential composition

The sequential composition of modes basically reduces to passing control from the source mode to the destination mode by connecting the exit point of the former with the entry point of the latter, as well as transferring data and triggering information via the global, and interface variables, respectively.

Parallel composition

REMES atomic modes and composite modes can be composed in parallel with each other. The parallel modes can execute concurrently, by interleaving actions, whereas the submodes can never execute in parallel; they simply obey the strict execution order imposed by the control flow.

Like in CHARON, if M is a composite mode, and $\mathsf{sm} \in \mathsf{M}$ is the variable ranging over the constituent submodes, we have: $\mathsf{L_M} \subseteq \bigcup_{\mathsf{sm}\in\mathsf{M}} \mathsf{G_{sm}}$, and $\mathsf{G_M} = \bigcup_{\mathsf{sm}\in\mathsf{M}} \mathsf{G_{sm}} - \mathsf{L_M}$. The mode composition is defined as follows.

Definition 8. ***(Parallel composition).*** *Assume* $\mathsf{Mode_A}$ *and* $\mathsf{Mode_B}$ *are two* REMES *(atomic or composite) modes. Then, the composition* $\mathsf{Mode_D} = \mathsf{Mode_A} \,||\, \mathsf{Mode_B}$ *is the mode with the set of local variables* $\mathsf{L_{Mode_D}} = \mathsf{L_{Mode_A}} \cup \mathsf{L_{Mode_B}}$*, the set of write variables* $\mathsf{Wr_{Mode_D}} = \mathsf{Wr_{Mode_A}} \cup \mathsf{Wr_{Mode_B}}$*, the set of read variables* $\mathsf{Rd_{Mode_D}} = \mathsf{Rd_{Mode_A}} \cup \mathsf{Rd_{Mode_B}}$*, and the top-level mode given by* $(\mathsf{Mode_A} \cup \mathsf{Mode_B})$*. Assuming the abstract resources CPU and bandwidth are used by both* $\mathsf{Mode_A}$ *and* $\mathsf{Mode_B}$*, the following holds for* $\mathsf{Mode_D}$*:* $\mathsf{cpu_{Mode_D}} = \mathsf{max}(\mathsf{cpu_{Mode_A}}, \mathsf{cpu_{Mode_B}})$ *and* $\mathsf{bdw_{Mode_D}} = \mathsf{max}(\mathsf{bdw_{Mode_A}}, \mathsf{bdw_{Mode_B}})$*. On the other hand, for memory and energy resources, the following is true:* $\mathsf{mem_{Mode_D}} = \mathsf{mem_{Mode_A}} + \mathsf{mem_{Mode_B}}$ *and* $\mathsf{eng_{Mode_D}} = \mathsf{eng_{Mode_A}} + \mathsf{eng_{Mode_B}}$. ■

In Definition 8, the parallel composition of composite modes subsumes the reunion of all the constituent submodes, corresponding edges and associated actions.

Parallel composition with a synchronization protocol

Let $\mathsf{Mode_A}$ and $\mathsf{Mode_B}$ be two composite REMES modes that want to update the global variables $\mathsf{gv_i}$ and $\mathsf{gv_j}$, and $\mathsf{gv_i}$ and $\mathsf{gv_k}$, respectively.

When parallel composing REMES modes, one has to deal with the scenario, illustrated in Figure 4.2, in which $\mathsf{Mode_A}$ and $\mathsf{Mode_B}$ contain edges that might concurrently require access to the global variable $\mathsf{gv_i}$.

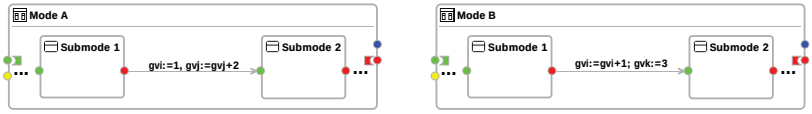


Figure 4.2: $\mathsf{Mode_A}$ and $\mathsf{Mode_B}$ might concurrently require access to global variable $\mathsf{gv_i}$.

To solve the problem presented in Figure 4.2, we need to implement a mutual exclusion protocol on the modeling level in REMES. Therefore, we:

1. extend the set of global variables of REMES modes, defined in Definition 7, with a set of lock variables, and
2. introduce a new parallel composition operator (sharp, $\sharp$) that ensures correct access to global variables, without employing additional communication between modes.

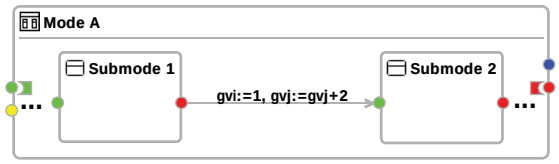
Additionally, we replace every mode that requires access to a global variable with its more detailed version, and we introduce controllers for regulating access to global variables. The composition

$$\mathsf{Mode_E} = \mathsf{Mode_A} \sharp \mathsf{Mode_B}$$

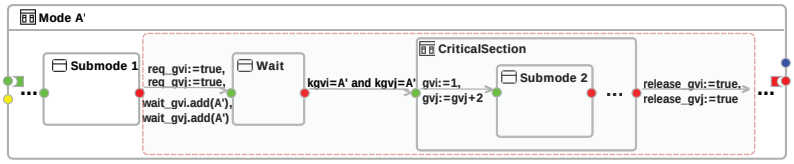
we call *parallel composition with a synchronization protocol* of $\mathsf{Mode_A}$ and $\mathsf{Mode_B}$.

Definition 9. ***(Parallel composition with a synchronization protocol).*** *Given two composite* REMES *modes* $\mathsf{Mode_A}$ *and* $\mathsf{Mode_B}$*, sets of global variables* $\mathsf{G_{Mode_A}} = \bigcup_{i=1..n} \mathsf{gv_i}$ *and* $\mathsf{G_{Mode_B}} = \bigcup_{j=1..m} \mathsf{gv_j}$*. Then, the composition* $\mathsf{Mode_E} = \mathsf{Mode_A} \sharp \mathsf{Mode_B}$ *is the parallel composition of* $\mathsf{Mode_{A'}}$*,* $\mathsf{Mode_{B'}}$*, the set of controllers* $\mathsf{Controller_{gv_i}}$ *and the set of controllers* $\mathsf{Controller_{gv_j}}$*, with the set of global variables* $\mathsf{G_{Mode_E}} = \mathsf{G_{Mode_{A'}}} \cup \mathsf{G_{Mode_{B'}}}$*, the set of lock variables* $\mathsf{K_{gv_{Mode_E}}} = \mathsf{K_{gv_{Mode_{A'}}}} \cup \mathsf{K_{gv_{Mode_{B'}}}}$ *and the set of shared variables* $\mathsf{S_{Mode_E}} = \mathsf{G_{Mode_E}} \cup \mathsf{K_{gv_{Mode_E}}}$*, such that* $\mathsf{f} : \mathsf{G_{Mode_E}} \rightarrow \mathsf{K_{gv_{Mode_E}}}$ *is a bijection that assigns locks to global variables.* ■

Observe, in Figure 4.3, that $\mathsf{Mode}_{A'}$ is a more detailed version of Mode_A. Let us assume that after visiting $\mathsf{Submode}_1$, Mode_A executes a discrete step in which it updates the global variables gv_i and gv_j, and that Mode_A accesses these variables also in its $\mathsf{Submode}_2$. This part of the behavior from Mode_A we replace in $\mathsf{Mode}_{A'}$ with a set of modes and edges (see the shaded parts in Figure 4.3 (b)). $\mathsf{Mode}_{A'}$ requires access to the global variables gv_i and gv_j by setting the boolean variables req_{gv_i} and req_{gv_j} to true, and adding the name of the mode in the waiting list wait_{gv_i} and wait_{gv_j}, respectively. Then, $\mathsf{Mode}_{A'}$ waits in the non-lazy mode Wait until the locks for both gv_i and gv_j are assigned to $\mathsf{Mode}_{A'}$, and enters the mode CriticalSection where the values of gv_i and gv_j are updated. CriticalSection is a composite mode that can consist of one or more submodes, depending on the behavior of Mode_A. It contains all the submodes of Mode_A that access gv_i and gv_j until their locks are released. For example, if Mode_A finishes accessing gv_i and gv_j after visiting $\mathsf{Submode}_2$ then the mode CriticalSection of $\mathsf{Mode}_{A'}$ will contain only $\mathsf{Submode}_2$. $\mathsf{Mode}_{A'}$ releases the locks of gv_i and gv_j after exiting CriticalSection.



(a) Mode_A before modification.



(b) $\mathsf{Mode}_{A'}$ presents a more detailed version of Mode_A.

Figure 4.3: Mode_A and $\mathsf{Mode}_{A'}$.

Figure 4.4 depicts the REMES mode modeling the behavior of a controller that regulates the access to the global variable gv_i. $\mathsf{Controller}_{gv_i}$ can be executed when their is a mode that requires access to the variable gv_i (i.e., when req_{gv_i} evaluates to true). When the list of modes requiring

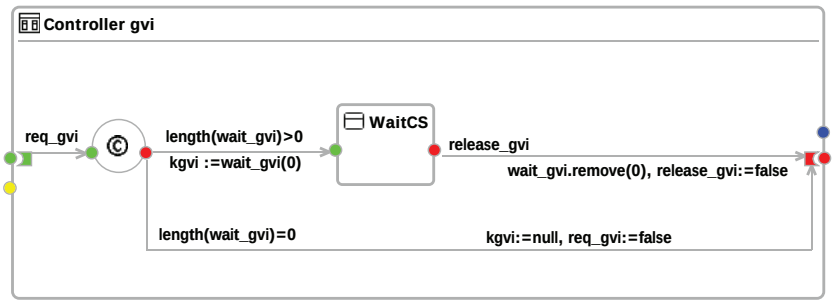


Figure 4.4: A controller mode for the global variable $\mathsf{gv_i}$ that regulates synchronous access to $\mathsf{gv_i}$.

access to $\mathsf{gv_i}$ is not empty, $\mathsf{Controller_{gv_i}}$ assigns $\mathsf{k_{gv_i}}$ to the first mode in the list waiting to gain access to the variable $\mathsf{gv_i}$. Note that the way the locks are assigned in $\mathsf{Controller_{gv_i}}$ is first-come-first-served. However, other lock assignment protocols can be modeled, if needed (such as priority locking). After assigning the lock to a certain mode $\mathsf{Controller_{gv_i}}$ waits in the non-lazy mode WaitCS until the lock is released. When the lock gets released (i.e., $\mathsf{release_{gv_i}}$ becomes true) the mode that used the lock is removed from the waiting list $\mathsf{wait_{gv_i}}$ and $\mathsf{Controller_{gv_i}}$ is exited. $\mathsf{Controller_{gv_i}}$ sets $\mathsf{req_{gv_i}}$ to false when the list $\mathsf{wait_{gv_i}}$ is empty i.e., when there is no longer mode requiring access to $\mathsf{gv_i}$.

4.2 Formal Analysis of REMES Models

4.2.1 Analysis model for REMES

Assume a set of resources $R_1, \ldots, R_n$ that a set of REMES modes have access to. Our main goal is to analyze various scenarios of the system's resource usage, and be able to compute, e.g., the maximum or minimum amounts of needed resources for guaranteeing correct resource-wise system behavior. Intuitively, this problem reduces to a scalar problem if one constructs a weighted sum of all resource consumptions, which should then be minimized, maximized, or manipulated in order to compute trade-offs. Consequently, we propose the following function as the analysis model for REMES:

$$r_{tot} \stackrel{\text{def}}{=} w_1 \times r_1 + w_2 \times r_2 + \ldots + w_n \times r_n,$$

where variable r_{tot} represents the total consumption of resources $R_1, \ldots,$ R_n, and variables $r_1, \ldots, r_n$ denote the accumulated consumption of $R_1, \ldots, R_n$, respectively. The constants, $w_1, \ldots, w_n$ (weights), represent the relative importance of $r_1, \ldots, r_n$. The values of the weights[1] are a subjective matter; the way they are chosen depends both on the application and on the analysis goals. For example, in designing a heavily resource-constrained soft real-time embedded system that might tolerate lateness at the expense of quality of service, in order to determine trade-offs between memory consumption and (execution) time, one can assign higher weight to memory than to time.

In order to be able to analyze REMES compositions, formally, we need a semantic translation of the model. If we consider resource consumptions $r_1, \ldots, r_n$ as cost variables $c_1, \ldots, c_n$, we can use the framework of Priced Timed Automata (PTA) as the underlying sem antic representation.

Informally, in its simplest form, transforming parallel composition of REMES modes into a network of PTA is quite straightforward: the syntactic REMES element of an edge corresponds to an edge in PTA, whereas the REMES semantic discrete step is a discrete transition in PTA's semantics. An atomic submode represents a PTA location, conditional connectors are removed in the transformation, and global variables of top-level modes are added to the set of global variables of the network of priced timed automata. The formal translation of two-level hierarchical REMES modes into a network of PTA, we introduce in Section 4.3. In the rest of this section, we formalize some of the main analysis goals that we are interested in.

4.2.2 Feasibility analysis

Component-based feasibility analysis reduces to checking whether the accumulated values of the resources consumed/used during all possible system behaviors are within the available resource amounts provided by the implementation platform. For resources like non-referable memory and energy, the composition of individual resource consumptions of REMES components is additive.

If one considers the PTA model of Definition 4 as the semantic translation of a REMES model, feasibility goals can then be formalized as the

[1] Actual values for resource weight constants $w_1, \ldots, w_n$ are defined in a model separate from REMES, explained in Appendix C.

following WCTL properties that the PTA model can be checked against:

$$A\,F_{cost\leq n}\,v \tag{4.1}$$
$$A\,G\,(q \Rightarrow A\,F_{cost\leq n}\,v) \tag{4.2}$$
$$E\,F_{cost\leq n}\,v \tag{4.3}$$
$$A\,G\,(q \Rightarrow E\,F_{cost\leq n}\,v) \tag{4.4}$$

where G and F are the WCTL temporal operators "always" and "eventually", respectively [32].

The above properties are in fact liveness properties (4.1), (4.2), (4.4), and a reachability property (4.3), indexed by cost constraints. The first two properties specify *strong feasibility*: property (4.1) requires that for all execution paths, the target location v is eventually reached within a total cost of n that can model the available resources provided by the platform; property (4.2) states that, for all paths, it is always the case that, once q, the cost of eventually reaching v will be no more than n, regardless of how v is reached. We say that property (4.3) models *weak feasibility*: the target location v may be reached within a total cost of n. Finally, property (4.4) states that for all paths, it is always the case that once a location q is reached, there exists a way by which v will be eventually reached within cost n. We call this last property *live feasibility*. However, model-checking WCTL formulae is decidable just for one-clock priced automata [31]. For other PTA, one can only verify reachability properties of the form given by (4.3).

Assuming that the cost function equates to $cost = w_1 \times c_1 + \ldots + w_n \times c_n$, and $c_1, \ldots, c_n$ are constants, the feasibility checks of the above properties involve a single cost variable that represents the accumulated resource consumption of all resources of interest, regardless of the class they belong to. Hence, semantically, the various resources become undistinguishable in these cases.

4.2.3 Optimal and worst-case resource consumption

Optimal and worst-case resource consumption analysis require (symbolic) algorithms on PTA, which compute the cost of the "cheapest", and/or most "expensive" trace that will eventually reach some goal. The optimal/worst-case resource consumption problem reduces to minimizing/maximizing the one-cost function $cost = w_1 \times c_1 + \ldots + w_n \times c_n$, such that a given reachability, or liveness property is satisfied.

Finding the optimal/worst-case resource consumption values to attain such goals calls for synthesis algorithms of minimal/maximal reachability costs for PTA, which have been proposed by Larsen and Rasmussen [73]. Similar to the feasibility case, only optimal/worst-case reachability costs can be synthesized by a model-checker. Later, we show how such a cost-optimal trace can be actually computed in the examples of Sections 5.3 and 7.4.

A considerable verification challenge arises in case some of the edge prices are negative, so that *cost* becomes a non-monotonically increasing cost function. In such situations, the usual branch-and-bound symbolic reachability algorithms, for PTA, cannot be applied as such anymore, since minimal/maximal reachability analysis requires a monotonically increasing cost function. The optimal- and worst-case-cost reachability problems have been theoretically solved even when negative costs are involved [30].

The tool used for verifying optimal resource consumption properties is UPPAAL CORA, where one could check, e.g., the relevant reachability property, $E\,F\,v$, while the tool calculates the minimum cost, in terms of resource exemption, "paid" to satisfy the property.

4.2.4 Trade-off analysis

Minimization of memory usage plays a major role in the design of embedded systems. Limited memory is one of the dominating constraints for many advanced embedded systems. However, while trying to minimize memory consumption, one might be forced to increase the execution time of real-time components beyond acceptable limits, that is, limits that, if exceeded, would make the set unschedulable.

As such, for a given REMES model, we may have more than one property to satisfy simultaneously, and we want to know whether it is possible to satisfy all of them, although they might be subjected to apparently conflicting constraints. In such cases, there should be possible to compute a *trade-off* between the considered resource consumptions.

Computing a trade-off between memory and execution time, or between any resource belonging to classes A and B, or A and C, or B and C of Table 4.1, could be done in PTA, by employing a single-cost function. The trade-off could then be achieved by varying the weights $w_1, \ldots, w_n$, accordingly.

In some other cases, e.g., when one needs to compute trade-offs

between consumption of resources belonging to class C, the function $cost = w_1 \times c_1 + \ldots + w_n \times c_n$ becomes a multi-cost function that lets one distinguish between various types of resources (e.g., between energy and CPU). This forces one to carry out the analysis on MPTA, rather than on PTA.

Assuming energy and CPU as the resources of interest, we want to determine which are the simultaneously achievable pairs of costs $(w_{eng} \times c_{eng}, w_{cpu} \times c_{cpu})$ such that energy consumption is minimized, while CPU consumption remains bounded from above. Such synthesis of cost pairs, which can be seen as a variant of trade-off analysis, can be achieved by applying *optimal conditional reachability* algorithms on MPTA [72], while considering c_{eng} as the primary cost and c_{cpu} as the secondary cost. Larsen and Rasmussen have proved that such problems are decidable for MPTA [72].

Alternatively, one could perform a feasibility-like check, by requiring that the following WCTL property is satisfied:

$$E\, F_{(w_{eng} \times c_{eng}) \leq n}\, (v \wedge (w_{cpu} \times c_{cpu}) \leq m)$$

The formula states that the accumulated weighted CPU usage will not be more than m ticks at location v, while v may be reached by consuming no more than n weighted energy units.

4.3 Transforming REMES Modes into a Network of (Priced) Timed Automata

In this section we introduce all the rules that describe the way in which we transform REMES modes into priced timed automata. Note that the current transformation rules cover two-level hierarchy only.

The terminology used in this section is as follows. We call a mode *atomic mode* if it is a stand-alone mode, and *atomic submode* if it is part of a composite mode. The transformation differs when transforming an atomic REMES mode, and when transforming an atomic REMES submode, which will be clarified in the following.

We divide this section into three parts. Firstly, we define how to transform an atomic mode into a priced timed automaton. Secondly, we define how to transform a composite REMES mode that contains a number of atomic submodes into a priced timed automaton. Thirdly,

we introduce the definition for transforming a parallel composition of REMES modes, be they atomic or composite, into a network of priced timed automata.

Transforming an atomic mode into a priced timed automaton. To demonstrate the difference between transforming an atomic mode into a priced timed automaton, and transforming an atomic submode, we present the result of the transformation in Figure 4.5. When transforming an atomic mode (see Figure 4.5(a)), the transformation will create an automaton augmented with an additional initial location Start and two edges – one representing the system startup activated on trigger a_1 (Start $\xrightarrow{a_1?}$ AtomicMode), and one to reactivate the mode once it has been exited and to synchronize with another mode on its exit, i.e., via trigger a_2 (AtomicMode $\xrightarrow[x:=0]{a_2!}$ AtomicMode). This synchronization information comes from the system architecture, and will be discussed later.

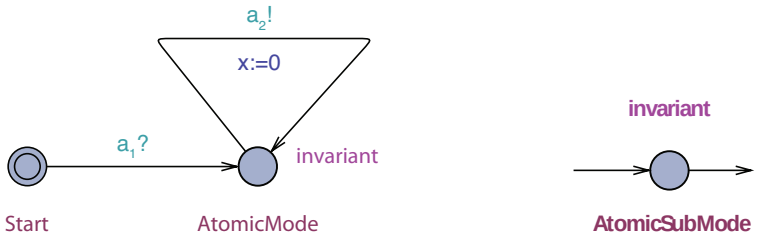


Figure 4.5: Transforming a REMES atomic mode and a REMES atomic submode into a priced timed automaton: (a) for an atomic mode, (b) for an atomic submode.

Figure 4.5(b) shows the transformation result of an atomic submode – the atomic submode is mapped to a location in a priced timed automaton corresponding to the composite mode, and the exit edges of the atomic submode are mapped to edges originating from this location in the automaton. The invariant of this location is in fact the invariant of the submode. The assigned cost of the delay in that location represents the resource consumption of the submode (expressed as a differential equation annotating the atomic submode). For an illustra-

tion, observe Figures 4.6(a) and 4.6(b). The submode SubMode2 has an invariant $x \leq C\_X$ and resource consumption rate $res2' = 10$, and the corresponding location SubMode2 in Figure 4.6(b) is annotated with an invariant $x \leq C\_X$ and with a cost-rate for delaying in that location $cost' = wres2 * 10$.

Definition 10. ***(Transforming an atomic mode into a priced timed automaton).*** *Let* $M = (V_m, In, Out, E_m, RC, Inv)$ *be a* REMES *atomic mode, and* $T = (L, l_0, X, V_t, I, Act, E_t, P)$ *be a priced timed automaton corresponding to* M*. We assume that the mode* M *contains only one clock* x*, and accordingly the corresponding set of clocks in the automaton* T *is* $X = \{x\}$*. We assume as well that the set of data variables of type integer, boolean or array of the mode*[2] M *is* V_{dm}*, where* $V_{dm} \subseteq V_m$*.*

The result of the transformation is a priced timed automaton T *with a set of clocks* $X = \{x\}$*, a set of data variables* $V_t = V_{dm}$*, an initial location* $l_0 =$ Start*, and a location* AtomicMode *representing the mode* M*. Hence* $L = \{Start, AtomicMode\}$*. The set of automaton's edges is* $E_t = \{e_{init}, e_{loop}\}$*, where* e_{init} *is an initialization edge connecting locations* Start *and* AtomicMode *(see Figure 4.5(a)), and* e_{loop} *is a loop edge, as follows:*

$$e_{init} = (Start, true, a_1?, \emptyset, AtomicMode),$$
$$e_{loop} = (AtomicTopMode, true, a_2!, x := 0, AtomicMode).$$

The action $a_1?$ *of the edge* e_{init} *ensures synchronization with the initialization automaton that we explain in Definition 15. The action* $a_2!$ *is defined by the architecture (might correspond to, e.g., a triggering signal) and is responsible for the activation of the modes triggered by* M*. Accordingly, the set of actions of* T *is* $Act = \{a_1?, a_2!\}$*.*

The invariant of the mode M *is transformed into the invariant of the location* AtomicMode*, such that* $I(AtomicMode) = Inv$*.*

The resource consumption rate RC *of the mode* M *is transformed into the cost-rate of the location* AtomicMode*, given by the cost function* P *i.e.,*

$$P(AtomicMode) = RC.$$

■

[2]Since UPPAAL does not support dynamically growing data structures, when transforming REMES into priced timed automata, we transform lists into arrays with a fixed upper bound. We map arrays of strings to integer arrays, and the REMES mode names of type string are mapped to integer constants in UPPAAL.

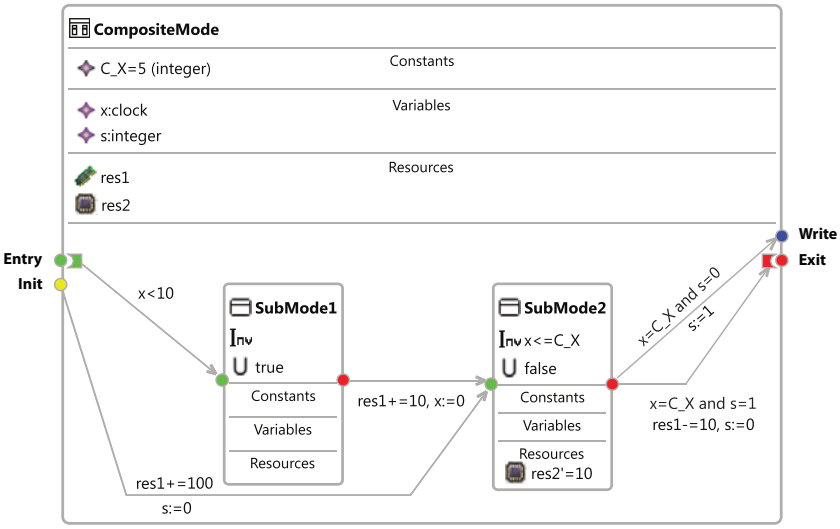
Note that in Definition 10 we assume an atomic mode that models an active component, which has no input triggers but only generates output triggers. Such a component activates other components when started on system startup. The Clock connector from ProCom is an example of a such component, and we show how to model its behavior in Sections 4.4 and 5.3.2. In the Clock automaton (see Section 5.3.3), the guard of the loop edge has been added after the transformation. If the component has input triggers, it should be modeled as a composite mode.

Steps for transforming a composite mode into a priced timed automaton. The transformation of a composite mode into a priced timed automaton is more complex than the transformation of an atomic mode. It consists of two steps as follows:

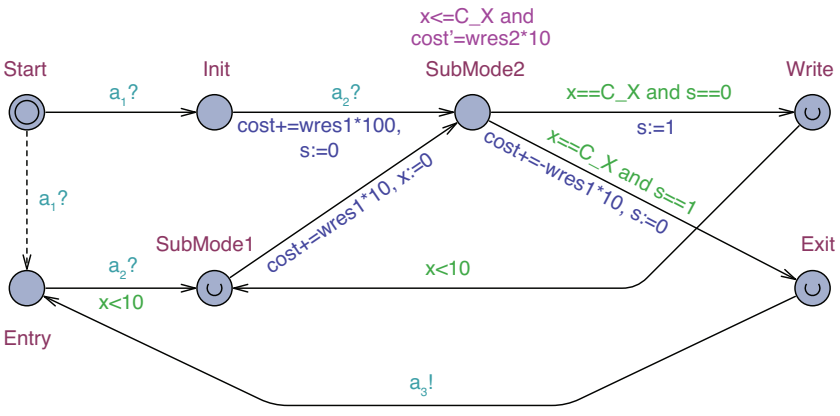
- removing conditional connectors in a composite mode (see Definition 11), and
- transforming atomic submodes into locations of a priced timed automaton (see Definitions 12 and 13).

Figure 4.6(a) shows a Remes composite mode with two atomic submodes, while Figure 4.6(b) depicts the corresponding priced timed automaton generated by the transformation. Similar to the transformation of an atomic mode, when transforming a composite mode additional locations Start, Init, Entry, Write and Exit are added to the automaton. The priced timed automaton waits in the initial location Start for the system startup, triggered by a_1. The locations Init, Entry, Write and Exit are created from the Init, Entry, Write and Exit points of a given composite mode. Note that the locations Write and Exit are marked as urgent to prohibit the automaton to stay in these locations, as mandated by the Remes "run-to-completion" semantics.

The edge connecting the location Init with the location SubMode2 originates from the composite mode's init edge. If a composite mode does not contain init edges, then the transformed automaton will not have a location Init and corresponding edges, and instead there will be an edge directly from Start to Entry (marked in Figure 4.6(b) with a dashed line). The edge from Entry to SubMode1 corresponds to the entry edge connecting the composite mode's Entry point and the submode Submode1. The return edge from Write to Submode1 ensures the internal execution rounds of the mode until the guard of the write edge holds.



(a)



(b)

Figure 4.6: A REMES composite mode CompositeMode consisting of two atomic submodes SubMode1 and SubMode2 (see Figure (a)), and a priced timed automaton resulted from the transformation of CompositeMode (see Figure (b)).

This cyclic behavior can not be interrupted from the outside. When the guard of the write edge does not hold anymore the mode can be exited only through the Exit point. In case a composite mode does not contain write edges, the transformed automaton will not have a location Write. The return edge from Exit to Entry depicts the fact that a composite mode has finished execution, and is ready to be reactivated from the outside.

Observe that in Figure 4.6(b) the $a_2?$ synchronization on the edge between the locations Entry and Submode1 comes from the system architecture, and represents, e.g., the synchronization on component entrance when modeling component interactions. This is also the case with the return edge from Exit to Entry, where the $a_3!$ marking represents, e.g., synchronization on component exit. A special case occurs when a component does not synchronize with any other component on its exit, i.e., does not trigger other components. In that case the return edge from Exit to Entry does not contain any synchronization channel.

Conditional connectors are removed during the transformation. Edges entering a conditional connector are combined with the edges exiting the conditional connector in all permutations, to retain the functionality in the resulting automaton. Definition 11 describes the removal of conditional connectors.

Definition 11. ***(Removing conditional connectors in a composite mode).*** *Let* $\mathsf{CM} = (\mathsf{SM}, \mathsf{V_m}, \mathsf{In}, \mathsf{Out}, \mathsf{E_m}, \mathsf{RC}, \mathsf{Inv}, \mathsf{CC})$ *be a composite mode, where* CC *is the set of conditional connectors and* $\mathsf{E_m}$ *the set of edges of the composite mode, and let* $\mathsf{T} = (\mathsf{L}, \mathsf{l_0}, \mathsf{X}, \mathsf{V_t}, \mathsf{I}, \mathsf{Act}, \mathsf{E_t}, \mathsf{P})$ *be a priced timed automaton corresponding to* CM*. Let* $\mathsf{cc} = (\mathsf{E_{in}}, \mathsf{E_{out}})$, $\mathsf{cc} \in \mathsf{CC}$ *be a conditional connector contained within mode* CM*, with a set of* n *input edges* $\mathsf{E_{in}} \subset \mathsf{E_m}$*, and a set of* m *output edges* $\mathsf{E_{out}} \subset \mathsf{E_m}$*. Let* $\mathsf{e} = (\mathsf{Mf}, \mathsf{A}, \mathsf{Mt})$*,* $\mathsf{e} \in \mathsf{E_m}$ *denote an edge from submode* $\mathsf{Mf} \in \mathsf{SM}$ *to submode* $\mathsf{Mt} \in \mathsf{SM}$ *with a corresponding action* $\mathsf{A} = (\mathsf{g}, \mathsf{S})$ *consisting of a guard* g *and a set of statements* S*.*

The conditional connector cc *is replaced with a set of edges* $\mathsf{E_{cc}} = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} \mathsf{e_{ij}}$*. Each edge* $\mathsf{e_{ij}}$ *is constructed from the guards and statements of* $\mathsf{e_i} = (\mathsf{Mf_i}, (\mathsf{g_i}, \mathsf{S_i}), \mathsf{Mt_i})$*,* $\mathsf{e_i} \in \mathsf{E_{in}}$ *and* $\mathsf{e_j} = (\mathsf{Mf_j}, (\mathsf{g_j}, \mathsf{S_j}), \mathsf{Mt_j})$*,* $\mathsf{e_j} \in \mathsf{E_{out}}$*, such that* $\mathsf{e_{ij}} = (\mathsf{Mf_i}, (\mathsf{g_i} \wedge \mathsf{g_j}, (\mathsf{S_i}, \mathsf{S_j})), \mathsf{Mt_j})$*.* ■

This simple method to remove conditional connectors does not guarantee correctness if a model contains chain connections of two or more

conditional connectors. To illustrate, take two conditional connectors cc_1 and cc_2. Let us assume that an entry edge to cc_1 contains statements that affect variables referenced in guards of an exit edge from cc_2. According to Definition 11, the guards g_1 and g_2 of both edges will be combined to form $\mathsf{g} = \mathsf{g}_1 \wedge \mathsf{g}_2$, and the set of statements S_1 and S_2 will be combined to form $\mathsf{S} = (\mathsf{S}_1, \mathsf{S}_2)$. The resulting guard g will refer to the the variables' values at the time of evaluation of g_1. However, this is incorrect for g_2 since the variables' values have changed as a result of the statements in S_1. A similar situation may occur with guards and statements of entry- and exit edges of a single conditional connector, but in this case the error is easier to find. We consider chaining conditional connectors to be a poor modeling practice, because of the possible side-effects of statements to the guards. In this case it is a better choice to use urgent submodes that guarantee atomicity of actions and have no side-effects.

The transformation of an atomic REMES submode is described in Definition 12, and depicted on Figure 4.5(b).

Definition 12. ***(Transforming an atomic submode into a location of a priced timed automaton).*** *Let* $\mathsf{CM} = (\mathsf{SM}, \mathsf{V_m}, \mathsf{In}, \mathsf{Out}, \mathsf{E_m}, \mathsf{RC}, \mathsf{Inv}, \mathsf{CC})$ *be a* REMES *composite mode, and let* $\mathsf{T} = (\mathsf{L}, \mathsf{l_0}, \mathsf{X}, \mathsf{V_t}, \mathsf{I}, \mathsf{Act}, \mathsf{E_t}, \mathsf{P})$ *be a priced timed automaton corresponding to* CM*. Let* $\mathsf{M} = (\mathsf{V_s}, \mathsf{In_s}, \mathsf{Out_s}, \mathsf{E_s}, \mathsf{RC_s}, \mathsf{Inv_s}) \in \mathsf{SM}$ *be an atomic submode, and* $\mathsf{e} = (\mathsf{M}, \mathsf{A}, \mathsf{Mt})$*,* $\mathsf{e} \in \mathsf{E_m}$ *be an exit edge from the atomic submode* M *to the atomic submode* $\mathsf{Mt} \in \mathsf{SM}$ *with a corresponding action* A*.*

The result of the transformation of M *extends the priced timed automaton* T *as follows. The set of locations* L *is extended with a location representing the submode* M*, e.g.,* $\mathsf{L} = \mathsf{L} \cup \{\mathsf{AtomicSubMode}\}$*. The set of edges* $\mathsf{E_t}$ *is extended with all exit edges of* M*, such that* $\mathsf{E_t} = \mathsf{E_t} \cup \{\mathsf{e} = (\mathsf{Mf}, \mathsf{A}, \mathsf{Mt}), \mathsf{e} \in \mathsf{E_m} \mid \mathsf{Mf} = \mathsf{M}\}$*. The set of actions* Act *is extended with an empty action, i.e.,* $\mathsf{Act} = \mathsf{Act} \cup \tau$*.*

The invariant of the submode M *is transformed into the invariant of the location* $\mathsf{AtomicSubMode}$*, such that* $\mathsf{I}(\mathsf{AtomicSubMode}) = \mathsf{Inv_s}$*.*

The resource consumption rate $\mathsf{RC_s}$ *of the submode* M *is transformed into the cost-rate of the location* $\mathsf{AtomicSubMode}$*, given by the cost function* P *i.e.,* $\mathsf{P}(\mathsf{AtomicSubMode}) = \mathsf{RC_s}$*. In case* M *is a non-lazy submode then it is transformed according to Definition 13.* ■

Urgent REMES atomic submodes are transformed into urgent locations of the priced timed automaton. A non-lazy atomic REMES submode

is a special case of the previous definition, as additional synchronization is needed to assure non-laziness of the mode. Definition 13 describes this special case.

Definition 13. ***(Transforming a non-lazy atomic submode into two locations of a priced timed automaton).*** *Let* $\mathsf{CM} = (\mathsf{SM}, \mathsf{V_m}, \mathsf{In}, \mathsf{Out}, \mathsf{E_m}, \mathsf{RC}, \mathsf{Inv}, \mathsf{CC})$ *be a* REMES *composite mode, and let* $\mathsf{T} = (\mathsf{L}, \mathsf{l_0}, \mathsf{X}, \mathsf{V_t}, \mathsf{I}, \mathsf{Act}, \mathsf{E_t}, \mathsf{P})$ *be a priced timed automaton corresponding to* CM*. The priced timed automaton* T *is a member of the network of priced timed automata* $\mathsf{N_T}$ *created as a result of the transformation of all* REMES *modes,* $\mathsf{T} \in \mathsf{N_T}$*, which we explain in Definition 15.*

Let $\mathsf{M} = (\mathsf{V_m}, \mathsf{In_s}, \mathsf{Out_s}, \mathsf{E_s}, \mathsf{RC_s}, \mathsf{Inv_s})$ *be an atomic non-lazy submode,* $\mathsf{M} \in \mathsf{SM}$*, such that* $\mathsf{Inv_s} \stackrel{def}{=} \mathsf{true}$*. Let* $\mathsf{e_i} = (\mathsf{M}, (\mathsf{g_i}, \mathsf{S_i}), \mathsf{Mt})$*,* $\mathsf{e_i} \in \mathsf{E_m}$ *be one of* n *exit edges from the atomic non-lazy submode* M *to the atomic submode* $\mathsf{Mt} \in \mathsf{SM}$ *with a corresponding action comprised of a guard* $\mathsf{g_i}$ *and a set of statements* $\mathsf{S_i}$*.*

The result of the transformation of M *extends the priced timed automaton* T *and the network* $\mathsf{N_T}$ *as follows. The set of locations* L *is extended with a location representing the submode* M*, e.g.,* NonLazySubMode*, and an additional committed location* $\mathsf{NonLazySubModeSync_i}$ *for each edge* $\mathsf{e_i}$*, so that* $\mathsf{L} = \mathsf{L} \cup \{\mathsf{NonLazySubMode}, \mathsf{NonLazySubModeSync_1}, \ldots, \mathsf{NonLazySubModeSync_n}\}$*.*

The set of edges $\mathsf{E_t}$ *is extended with all exit edges of* M*, where each exit edge* $\mathsf{e_i}$ *is split in two parts –* $\mathsf{e_{i1}}$ *from* NonLazySubMode *to* $\mathsf{NonLazySubModeSync_i}$*, and* $\mathsf{e_{i2}}$ *from* $\mathsf{NonLazySubModeSync_i}$ *to* Mt *i.e.,* $\mathsf{E_t} = \mathsf{E_t} \cup \bigcup_{i=1}^{n} \{\mathsf{e_{i1}}, \mathsf{e_{i2}}\}$*. Assuming that* τ *denotes an empty action, each exit edge* $\mathsf{e_i}$ *is then split to:*

$$\mathsf{e_{i1}} = (\mathsf{NonLazySubMode}, \mathsf{g_i}, \mathsf{nlSync?}, \emptyset, \mathsf{NonLazySubModeSync_i}),$$
$$\mathsf{e_{i2}} = (\mathsf{NonLazySubModeSync_i}, \mathsf{true}, \tau, \mathsf{S_i}, \mathsf{Mt}),$$
$$\mathsf{Act} = \mathsf{Act} \cup \{\mathsf{nlSync?}, \tau\}$$

An additional priced timed automaton $\mathsf{T_{nl}} = (\mathsf{L'}, \mathsf{l'_0}, \mathsf{X'}, \mathsf{V'_t}, \mathsf{I'}, \mathsf{Act'}, \mathsf{E'_t}, \mathsf{P'})$ *is added to the network* $\mathsf{N_T}$*, with one* NonLazySubModeSyncEnd *initial location, and a loop edge used for synchronization:*

$$
\begin{aligned}
&\mathsf{L}' = \{\mathsf{NonLazySubModeSyncEnd}\},\\
&\mathsf{l}'_0 = \mathsf{NonLazySubModeSyncEnd},\\
&\mathsf{X}' = \emptyset,\\
&\mathsf{V}'_t = \emptyset,\\
&\mathsf{I}' = \mathsf{true},\\
&\mathsf{Act}' = \mathsf{Act}' \cup \{\mathsf{nlSync!}\},\\
&\mathsf{E}'_t = \{(\mathsf{NonLazySubModeSyncEnd}, \mathsf{true}, \mathsf{nlSync!},\\
&\qquad \emptyset, \mathsf{NonLazySubModeSyncEnd})\},\\
&\mathsf{P}' = 0.
\end{aligned}
$$

Actions nlSync! *and* nlSync? *are synchronization actions over urgent channel* nlSync. *The resource consumption rate* $\mathsf{RC_s}$ *of the submode* M *is transformed into the cost-rate of the location* NonLazySubMode, *given by the cost function* P *i.e.,* $\mathsf{P}(\mathsf{NonLazySubMode}) = \mathsf{RC_s}$. ■

Definition 13 introduces one non-lazy synchronization automaton $\mathsf{T_{nl}}$ per each non-lazy REMES mode, which can be further optimized to have a single automaton for the entire system. In this case, there will be one synchronization edge triggering the urgent synchronization channel nlSync per each composite mode CM having non-lazy submodes, as only one submode can be active at a time. Figure 4.7 gives an example of this, where the automaton in Figure 4.7(c) synchronizes with non-lazy modes from composite modes C1, C2, and C3, using urgent synchronization channels c1nlSync, c2nlSync, and c3nlSync, respectively. We exemplify the transformation of non-lazy modes in Section 5.4. Finally, we define the transformation of a composite mode consisting of atomic submodes into a priced timed automaton.

Definition 14. ***(Transforming a composite mode into a priced timed automaton).*** *Let* $\mathsf{CM} = (\mathsf{SM}, \mathsf{V_m}, \mathsf{In}, \mathsf{Out}, \mathsf{E_m}, \mathsf{RC}, \mathsf{Inv}, \mathsf{CC})$ *be a* REMES *composite mode. Let* $\mathsf{T} = (\mathsf{L}, \mathsf{l_0}, \mathsf{X}, \mathsf{V_t}, \mathsf{I}, \mathsf{Act}, \mathsf{E_t}, \mathsf{P})$ *be a priced timed automaton corresponding to* CM. *The set of input control points of the mode contains the* Init *and* Entry *control points,* $\mathsf{In} = \{\mathsf{Init}, \mathsf{Entry}\}$, *and the set of output control points of the mode contains the* Write *and* Exit *control points,* $\mathsf{Out} = \{\mathsf{Exit}, \mathsf{Write}\}$. *The set of data variables of type integer, boolean or array of the mode* CM *is* $\mathsf{V_{dm}}$, *where* $\mathsf{V_{dm}} \subseteq \mathsf{V_m}$.

Conditional connectors, $\mathsf{cc} \in \mathsf{CC}$ *are removed and the resulting edges are added to set* $\mathsf{E_m}$, *as per Definition 11.*

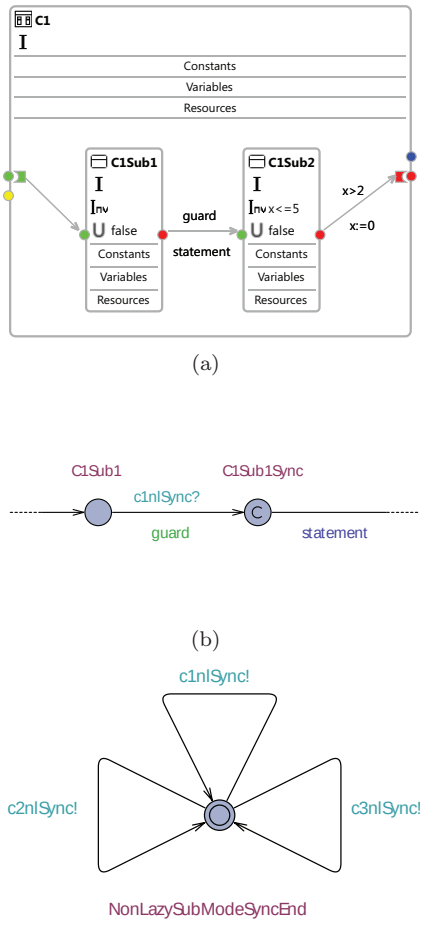


Figure 4.7: Transforming a non-lazy REMES atomic submode into two locations of a priced timed automaton. Figure (a) shows a REMES model of a composite mode C1 containing a non-lazy submode C1Sub1. Figure (b) shows locations of the priced timed automaton corresponding to the composite mode C1 created to model the non-lazy submode C1Sub1. Figure (c) shows an additional automaton used to trigger the edge between C1Sub1 and C1Sub2 once guard becomes true.

The result of the transformation is a priced timed automaton T *defined as follows. The set of locations* L *consists of an initial start location* $\mathsf{l}_0 = \mathsf{Start}$*, locations for all input and output control points of the composite mode, and locations resulting from the transformations of all submodes:* $\mathsf{SM} \rightarrow \mathsf{SM}'$*. Hence* $\mathsf{L} = \{\mathsf{Start}, \mathsf{In}, \mathsf{Out}\} \bigcup \mathsf{SM}'$*. The locations* Exit *and* Write*, representing the exit- and write point of the composite mode, respectively, are marked as urgent. The set of submodes* SM *is transformed into corresponding locations as per Definition 12 and Definition 13.*

The set of automaton's edges is $\mathsf{E_t} = \{\mathsf{e_{init}}, \mathsf{e_{loop}}\} \cup \mathsf{E'_m} \cup \mathsf{E_{write}}$*, where* $\mathsf{e_{init}}$ *and* $\mathsf{e_{loop}}$ *are initialization and loop edges:*

$$\mathsf{e_{init}} = (\mathsf{Start}, \mathsf{true}, \mathsf{a_1?}, \emptyset, \mathsf{Init}),$$
$$\mathsf{e_{loop}} = (\mathsf{Exit}, \mathsf{true}, \mathsf{a_3!}, \emptyset, \mathsf{Entry}).$$

The action $\mathsf{a_1?}$ *of the edge* $\mathsf{e_{init}}$ *ensures synchronization with the initialization automaton. The action* $\mathsf{a_3!}$ *is defined by the architecture and is responsible for the activation of the modes activated by* CM*. Synchronization on edges* $\mathsf{e_{init}}$ *and* $\mathsf{e_{loop}}$ *can be omitted, depending on the system architecture.*

We denote by $\mathsf{E'_m}$ *the set of internal edges obtained during the transformation of the submodes. The set of write edges* $\mathsf{E_{write}}$ *is obtained by copying all edges of the mode originating in the mode's entry point and modifying them to originate from the mode's write point. For an edge* $\mathsf{e} = (\mathsf{Mf}, \mathsf{A}, \mathsf{Mt}) \in \mathsf{E_m}$*, we define the function that modifies the source location of an edge to* Write *as* $\mathsf{copyToWrite}((\mathsf{Mf}, \mathsf{A}, \mathsf{Mt})) = (\mathsf{Write}, \mathsf{A}, \mathsf{Mt})$*. The resulting set of write edges is then* $\mathsf{E_{write}} = \bigcup \mathsf{copyToWrite}(\mathsf{e}), \mathsf{e} \in \mathsf{E_m} : \mathsf{Mf} = \mathsf{Entry}$*. The set of actions of* T *is* $\mathsf{Act} = \mathsf{Act} \cup \{\mathsf{a_1?}, \mathsf{a_3!}\}$*. Variables from the mode are transformed into the set of automaton's variables* $\mathsf{V_t}$ *:* $\mathsf{V_t} = \mathsf{V_{dm}}$*.* ■

Transforming a parallel composition of REMES modes into a network of priced timed automata. The transformation of the behavior of a system modeled in REMES into a network of priced timed automata starts from the parallel composition of all REMES modes, representing the system's behavior.

If we assume that $\mathsf{N_M}$ is the set of all modes that are part of the composition, we can then formally define the transformation into a network

of priced timed automata in Definition 15.

Definition 15. ***(Transforming a parallel composition of* REMES *modes into a network of priced timed automata).*** *Let* $\mathsf{N_M}$ *be the set of all atomic and composite modes of the parallel composition. If we take* M *to be a mode from the set* $\mathsf{N_M}$*,* $\mathsf{M} \in \mathsf{N_M}$*, then the model-to-model transformation* $\mathsf{r2t} \stackrel{def}{=} \mathsf{N_M} \rightarrow \mathsf{N_T}$ *transforms the set of* REMES *modes into a network of priced timed automata. The transformation r2t of a set* $\mathsf{N_M}$ *is performed by transforming all individual modes* $\mathsf{M} \in \mathsf{N_M}$ *into corresponding automata* $\mathsf{a_m} \in \mathsf{A_M}$*, with the addition of a set of automata* $\mathsf{A_T}$*, where* $\mathsf{N_T} = \mathsf{A_M} \cup \mathsf{A_T}$*.* ■

The set of predefined automata $\mathsf{A_T}$ comes from the system architecture. The temperature control system that we introduce in Section 4.4, and the transformation of its REMES model into priced timed automata, presented in Section 5.3, will exemplify the use of an initialization automaton that has been added as a result of the transformation. Additional automata may be added for synchronization, or to implement various connectors defined in the architectural model.

We expect that the transformation results are reviewed by a verification expert to check whether the transformation needs adjusting. For example, in case the guard on an exit edge of a non-lazy atomic submode is a predicate with more than one term (conjunction of predicates) the predicate might not be directly translated to a guard, and there might be a need of adding additional locations. Therefore, the locations of the resulting automaton corresponding to non-lazy atomic submodes are marked so that the verification expert pays special attention to them. Additionally, transforming an urgent REMES atomic submode into an urgent location may not be correct in all cases, and deadlocks may occur when outgoing edges of urgent locations contain synchronization. In such cases, urgent atomic submodes may be instead transformed into committed locations.

4.4 Example: A Temperature Control System

We demonstrate the modeling and analysis concepts of REMES on a temperature control system (TCS) for a heat producing reactor. The example system is taken from a case study analyzed by Alur et al. [10].

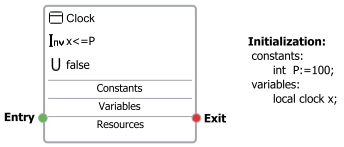
The TCS uses two independent rods that can be inserted into the core of the reactor, to control the coolant temperature in the core. If inserted into the core, the control roads absorb neutrons and consequently the reaction is slowed down, so the temperature inside the core starts decreasing. If they are pulled out, the reaction speeds up again, which in turn increases the core temperature. The goal is to maintain the temperature in the reactor core between its minimum value θ_{min} and maximum value θ_{max}. Whenever the temperature core reaches θ_{max}, a rod must be selected and inserted into the core. For safety reasons, a rod can be inserted again in the core only if T time units have elapsed since its last usage.

We model the resource usage behavior of the TCS with three REMES modes Clock, HCController, and RodSelector depicted in Figures 4.8(a), 4.8(b), and 4.8(c), respectively. The modes communicate data through the global variables temp and tempROD. The way in which the modes are activated is determined by the architecture of the system that we present in Chapter 5. Here, we assume that Clock activates HCController every P time units, and that when HCController has finished executing it activates RodSelector.

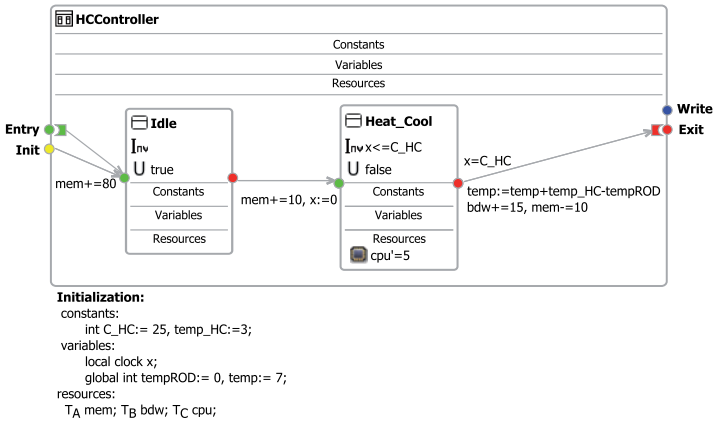
In the TCS model, we make use of three resources: processor load (CPU), bandwidth (bdw) and memory (mem). We assume mem as a discrete resource belonging to class A, bdw as a discrete resource belonging to class B, and we treat CPU as a continuous resource belonging to class C of Table 4.1. We assume that bandwidth presents the width of the pipe through which the data is being sent. We treat static memory and simple dynamic memory that is allocated when a mode is entered and released as soon as the same mode is exited, without memory management.

The Clock mode is an atomic REMES mode with an invariant $\mathsf{x} \leq \mathsf{P}$. The HCController mode contains two atomic REMES modes: Idle and Heat_Cool. The execution of HCController consumes 80 units of static memory. HCController starts executing by entering the Idle mode, which is an urgent mode that is exited instantaneously. HCController stays in mode Heat_Cool for C_HC time units. The difference (temp_HC − tempROD), where temp_HC is the heating produced by the reactor, and tempROD is the cooling rate of a selected rod, is used to update the reactor temperature.

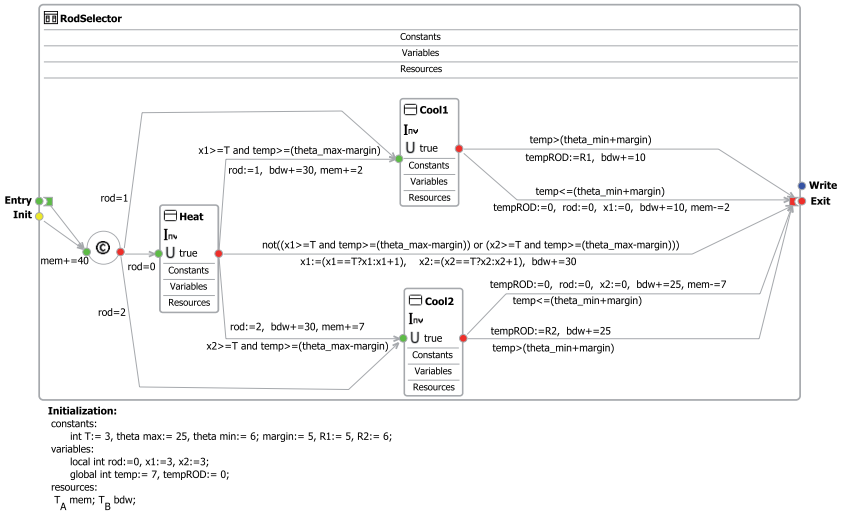
The RodSelector mode is made of a conditional connector, three urgent atomic modes Heat, Cool1 and Cool2, and edges. The selected rod



(a)



(b)



(c)

Figure 4.8: The Clock, HCController, and RodSelector modes of the TCS depicted in Figures (a), (b), and (c), respectively.

id is saved in the rod variable, where the value 0 denotes that no rod has been selected. From Heat, based on the temperature of the core, temp, and the time since a rod has been last used for cooling the core (i.e., x1 and x2 for rod1 and rod2, respectively), an available rod is selected, and, consequently, RodSelector enters modes Cool1 or Cool2, or alternatively finishes executing and exits, provided that no rod needs to be used. Note that in RodSelector we make selection of rods when there is an available mode for cooling and temp $\geq$ (theta_max $-$ margin) evaluates to true, where margin is a safety margin that assures the reactor core will not overheat. R1 and R2 present the rates of cooling of the first and second rod, respectively.

We can analyze the REMES–based TCS by transforming it into a network of PTA models, in UPPAAL CORA. In order to easily follow the transformation rules presented in Section 4.3, for modeling the architecture of the system we can use any architectural language that has a clear separation between data- and control flow (such as ProCom, AADL [44], SaveCCM [8], or Rubus [52]). In the next chapter we present the ProSave architectural model of the TCS system, the integration between ProCom and REMES, and the transformation into priced timed automata.

4.5 Summary

In this chapter, we have introduced REMES – a language for resource modeling and analysis of embedded systems. The essence of REMES is that it provides support for reasoning about discrete and continuous abstract resources characterized further by the way in which they are consumed and released, and by whether they can be referred to, or not. The abstract resources, in our case, *memory, CPU, energy, bandwidth,* have a dedicated type in the language, that is, mem, CPU, eng, bdw, respectively. In order to express resource usage in a system, REMES has a graphical behavioral language influenced by CHARON [13], timed and hybrid automata, and Statecharts. The language supports hierarchical modeling and has notions of explicit entry- and exit points that make it suitable as a semantic basis in component-based development frameworks. REMES has notions of continuous variables, flows, and progress constraints (invariants), which fit modeling timed behaviors in embedded systems.

In this setting, we have defined three important resource analysis

problems: feasibility analysis, trade-off analysis, and optimal/worst-case resource analysis. All these problems rely on weighted sums of consumed amounts of resources and their given weights. In this way, the analysis can result in optimizing the overall resource usage of a system, with respect to parameters, such as criticality or costs of the available resources. To be able to formally analyze REMES models we have provided a set of transformation rules that semantically translate REMES modes into priced timed automata. The concepts of REMES we have illustrated on a study example of a temperature control system that consumes CPU, bandwidth, and memory resources.

Chapter 5

Integrating ProCom and Remes

ProCom has been developed to facilitate the modeling and analysis of functional and extra-functional properties, but does not, per se, provide any means to actually model them. It needs to be complemented with formalisms, complying with the component-based approach, which enable early formal analysis of relevant concerns. One step towards this support for formal analysis is the integration of Remes, by which functional behavior, resource consumption and timing can be addressed within a single modeling language. To accomplish this, in this chapter, and in paper [105], we propose a way of mapping a ProCom component interface to a Remes interface (see Section 5.1). Further, in Section 5.2, we show how to pack a ProCom component, annotated with attributes, such as required resources, with its associated Remes behavioral model. Then, both the interface and internal models of component behavior are seen as the actual reusable unit of composition, which can be employed as such, without modification, in adequate design contexts. We exemplify the concepts of connecting and packaging a ProCom component with its Remes behavioral model on two examples: a temperature control system, where the architecture of the system is modeled in ProSave (see Section 5.3), and on a turntable drilling system, where the system architecture is modeled in ProSys (see Section 5.4).

5.1 Connecting Component Interfaces and Remes Modes

The connection between ProCom and Remes is done differently for ProSave and ProSys. Hence, in this section, we first define the connection between ProSave and Remes, and then we define the connection between ProSys and Remes.

5.1.1 Connecting ProSave and Remes

The connection between ProSave and Remes is done by mapping a ProSave- to a Remes interface as follows. The input- and output trigger ports from a ProSave component are mapped to interface read- and interface write variables of type boolean in a Remes mode, respectively. However, since we assume that the ProSave architectural model is in charge of signaling when the modes should be entered, we choose to omit such signals from the Remes modes. Similarly, we map the ProSave input- and output- data ports to interface read- and interface write- Remes variables, respectively. Note that a Remes mode should be associated with each service of a ProSave component, since services can be triggered independently and may run concurrently. This is due to the Remes "run-to-completion" semantics, where only one submode at a time can be active.

Definition 16. ***(Formal Definition of Connecting the ProSave and Remes interfaces).*** *Let* P *be the set of data ports of a ProSave component* C *made of one service. Each data port* $\mathsf{p}_{\mathsf{i}\in[1\ldots\mathsf{n}]} \in \mathsf{P}$ *is a tuple* $(\mathsf{Name}, \mathsf{Kind}, \mathsf{Type}, \mathsf{Value})$*, where:* Name *is the data port identifier,* Kind *is a tag denoting the kind of a data port, possible tag values are input data or output data,* Type *encodes the port's data type, and* Value *denotes a literal data value compatible with some type. Further, let* M *be a* Remes *mode that depicts the behavior of the component* C*, and* V *the set of all variables of mode* M *that correspond to the data ports of* C*. Each variable* $\mathsf{v}_{\mathsf{j}\in[1\ldots\mathsf{n}]} \in \mathsf{V}$ *is a tuple* $(\mathsf{Name}, \mathsf{Kind}, \mathsf{Type}, \mathsf{Value})$*, where:* Name *is an identifier of the variable,* Kind *distinguishes between interface read- (variable of the mode coming from the ProSave component's interface that may be written by other modes) and interface write- (variable of the mode coming from the ProSave component's interface that may be read by other modes) variables,* Type *encodes the variable's data type,*

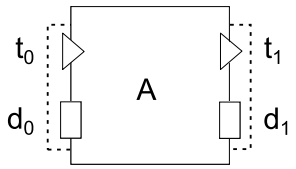
and Value *stores the actual variable value. The connection between mode* M *and the interface of a ProSave component* C *is given by a mapping function* $\mu : P \rightarrow V$ *that maps component data ports to* REMES *mode variables. Assuming a data port* $\mathsf{p_i}$ *the mapping is done as follows*

$$\mu(\mathsf{p_i}) = \mathsf{v_i}$$

such that the following boolean condition holds:

$$\begin{aligned}
&\mathsf{Name}(\mathsf{v_i}) = \mathsf{Name}(\mathsf{p_i}) \\
\wedge\ &\\
&((\mathsf{Kind}(\mathsf{p_i}) = \mathsf{inputdata} \wedge \mathsf{Kind}(\mathsf{v_i}) = \mathsf{interface\ read} \\
&\quad \wedge \mathsf{Type}(\mathsf{v_i}) = \mathsf{Type}(\mathsf{p_i}) \wedge \mathsf{Value}(\mathsf{v_i}) = \mathsf{Value}(\mathsf{p_i})) \\
&\vee \\
&((\mathsf{Kind}(\mathsf{p_i}) = \mathsf{outputdata} \wedge \mathsf{Kind}(\mathsf{v_i}) = \mathsf{interface\ write} \\
&\quad \wedge \mathsf{Type}(\mathsf{v_i}) = \mathsf{Type}(\mathsf{p_i}) \wedge \mathsf{Value}(\mathsf{v_i}) = \mathsf{Value}(\mathsf{p_i}))
\end{aligned}$$

■



ProSys port	REMES variables
d_0	*interface read int* d_0
t_0	*omitted from the* REMES *mode*
d_1	*interface write float* d_1
t_1	*omitted from the* REMES *mode*

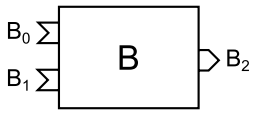
Figure 5.1: Example of how ProSave ports are mapped to REMES variables.

The parallel composition of the REMES modes associated to all ProSave components in the given system, together with representations of the ProSave connectors and connections, describe the whole system's behavior. Figure 5.1 exemplifies the mechanism of connecting the ProSave

and Remes interfaces. Component A receives a trigger via trigger port t_0 and a data of type int via input data port d_0, and it sends a trigger via trigger port t_1 and a data of type float through data port d_1.

5.1.2 Connecting ProSys and Remes

The connection between a Remes mode and the interface of a ProSys component is established in the following way. An input message of a component is mapped to two interface read variables of its mode, whereas an output message of a component is mapped to two interface write variables of its mode. In each case, one of the variables is a boolean that signals the receiving/sending of the message, respectively, while the other variable keeps the value of the message. If the message is empty, only the boolean variable is used.



ProSys port	Remes variables
B_0	*interface read boolean* B_0 and *float* B_0*_value*
B_1	*interface read boolean* B_1
B_2	*interface write boolean* B_2 and *int* B_2*_value*

Figure 5.2: Example of how ProSys ports are mapped to Remes variables.

Definition 17. ***(Formal Definition of Connecting the ProSys and Remes interfaces).*** *Let* P *be the set of message ports of a ProSys component* C*. Each port* $p_{i\in[1...n]} \in P$ *is a tuple* (Name, Kind, Type, Value)*, where:* Name *is the port identifier,* Kind *models the input/output feature of the message port,* Type *encodes the port's data type, and* Value *stores the port's actual data value. Further, let* M *be a* Remes *mode that depicts the behavior of the component* C*, and* V *the set of all variables of mode* M *that correspond to the ports of* C*. Each variable* $v_{i\in[1...n]} \in V$ *is a tuple* (Name, Kind, Type, Value)*, where:* Name *is an identifier of the variable,* Kind *distinguishes between interface read- and interface write variables,* Type *encodes the variable's data type, and* Value *stores the actual variable value. The connection between mode* M *and the interface of*

a ProSys component C *is given by a mapping function* $\mu : P \rightarrow V$ *that maps component ports to* REMES *mode variables. Assuming non-empty messages (*$\mathsf{Value}(\mathsf{p_i}) \neq \mathsf{NULL}$*), the mapping is defined as follows:*

$$\mu(\mathsf{p_i}) = \overline{\mathsf{v}}_\mathsf{i}, \quad \overline{\mathsf{v}}_\mathsf{i} = (\mathsf{v}_{\mathsf{i}_1}, \mathsf{v}_{\mathsf{i}_2}),$$

such that the following boolean condition holds:

$$\begin{aligned}
&\mathsf{Name}(\mathsf{v}_{\mathsf{i}_1}) = \mathsf{Name}(\mathsf{p_i}) \wedge \mathsf{Name}(\mathsf{v}_{\mathsf{i}_2}) = \mathsf{Name}(\mathsf{p_i}) + \text{“_value”} \\
\wedge\ &\\
&((\mathsf{Kind}(\mathsf{p_i}) = \mathsf{input} \wedge \mathsf{Kind}(\mathsf{v}_{\mathsf{i}_1}) = \mathsf{interface\ read} \wedge \mathsf{Type}(\mathsf{v}_{\mathsf{i}_1}) = \mathsf{boolean} \\
&\quad \wedge \mathsf{Value}(\mathsf{v}_{\mathsf{i}_1}) = \mathsf{false} \wedge \mathsf{Kind}(\mathsf{v}_{\mathsf{i}_2}) = \mathsf{interface\ read} \\
&\quad \wedge \mathsf{Type}(\mathsf{v}_{\mathsf{i}_2}) = \mathsf{Type}(\mathsf{p_i}) \wedge \mathsf{Value}(\mathsf{v}_{\mathsf{i}_2}) = \mathsf{Value}(\mathsf{p_i})) \\
&\vee \\
&(\mathsf{Kind}(\mathsf{p_i}) = \mathsf{output} \wedge \mathsf{Kind}(\mathsf{v}_{\mathsf{i}_1}) = \mathsf{interface\ write} \wedge \mathsf{Type}(\mathsf{v}_{\mathsf{i}_1}) = \mathsf{boolean} \\
&\quad \wedge \mathsf{Value}(\mathsf{v}_{\mathsf{i}_1}) = \mathsf{false} \wedge \mathsf{Kind}(\mathsf{v}_{\mathsf{i}_2}) = \mathsf{interface\ write} \\
&\quad \wedge \mathsf{Type}(\mathsf{v}_{\mathsf{i}_2}) = \mathsf{Type}(\mathsf{p_i}) \wedge \mathsf{Value}(\mathsf{v}_{\mathsf{i}_2}) = \mathsf{Value}(\mathsf{p_i})))
\end{aligned}$$

In case an empty message is received/sent ($\mathsf{Value}(\mathsf{p_i}) = \mathsf{NULL}$*), the mapping function returns* $\overline{\mathsf{v}}_\mathsf{i} = (\mathsf{v}_{\mathsf{i}_1}, \mathsf{NULL})$. ■

The parallel composition of the REMES modes associated to all ProSys components in the given system, together with representations of the ProSys message channels and connections, describe the whole system's behavior. Figure 5.2 exemplifies the mechanism of connecting the ProSys and REMES interfaces. Component B receives a message of type float via input port B_0 and an empty message via input port B_1, and it sends a message of integer type through output port B_2.

5.2 Packaging ProCom Components and REMES Modes Together

The packaging of ProCom components and their REMES behavioral models is managed by the Attribute Framework [92], in which extra-functional properties are represented by attributes consisting of an *attribute type* and one or more *attribute values*. The attribute type specifies how a given extra-functional property is represented, i.e., what data type is required for its values and how they should be manipulated. The complete list of

the attribute types that are available during the development is stored in an attribute registry together with the specification of each attribute, that is (i) the list of entities to which this attribute can be attached (i.e., *attributables*), and (ii) the valid storage format i.e., *data format* for its values (e.g., integer, interval, external models, complex types). Providing that it is authorized by its specification, an attribute can be associated with any entity of a component model, such as a component, a message port, a connection or even a component instance.

Table 5.1 gives some examples of possible attribute specifications. For instance, one of the attributes captures static memory usage, represented by a single integer. This property makes sense for a component or a subsystem, but not for instance a single message port, and thus the specification states that it can only be attached to those two entity types. The packaging of ProCom and Remes is achieved by defining a new attribute type attached to a ProCom component or a ProSave service in the attribute registry, which has a complex attribute value consisting of (i) a reference to the Remes model file in the component structure and (ii) a reference to the mapping file specifying the relation between the ports of the component and the variables of the Remes model, as we described in Sections 5.1.1 and 5.1.2.

TypeID	**Attributable(s)**	**Data format**	**Documentation (short)**
Static memory	Component	Int	The amount of memory (in kB) statically allocated by the ProSave component or the ProSys subsystem.
Average delay	Channel	Int	The average delay (in ms) for communication over the channel.
Remes model	Component, Service	<modelFilePath; mappingFilePath>	A reference to a Remes model, and to the mapping file between ProCom component's ports and the variables used in the Remes model.

Table 5.1: Examples of attributes.

5.3 Example Revisited: A Temperature Control System

In Chapter 4, we have introduced the temperature control system (TCS) example, and we have modeled its behavior with three REMES modes: Clock, HCController and RodSelector. Here, we first model the software architecture of the TCS in ProSave. Then, we exemplify the connection between ProSave and REMES on the TCS, and next, formally analyze the TCS.

5.3.1 Architecting the TCS in ProSave

The TCS system can be modeled in ProSave with two components HCController and RodSelector, and one clock connector Clock (see Figure 5.3). The Clock connector periodically generates the trigger activate that activates the HCController component. HCController is responsible for activating the heating/cooling process in the core by sending heat_cool trigger to RodSelector. When activated, RodSelector uses the temperature data of the core conveyed through data port temp to decide whether the core should continue to heat, or if a road should be inserted into the core to decrease the temperature. Finally, the temp value in the HCController is updated by the reading of the latest value of the variable tempROD that is assigned to the cooling effect of the rods within the RodSelector component.

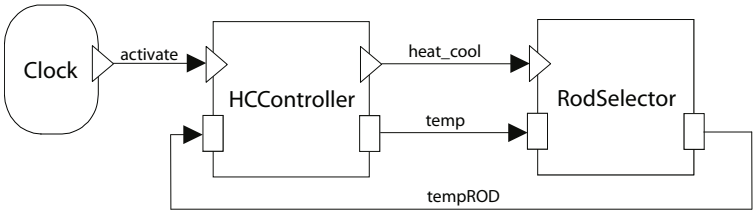


Figure 5.3: ProSave design of the temperature control system.

5.3.2 Behavioral modeling of the TCS in Remes

The connection between the ProSave HCController and RodSelector components, and the Remes modes describing their behavior, is done as described in Section 5.1.1. The input- and output data ports of the components are mapped to interface read- and interface write Remes variables of same type as the port's data type, respectively. As such, we have mapped the input data port tempROD from the HCController component to the interface read variable tempROD in the HCController Remes mode (see Figure 5.4(b)). On the other hand, we have mapped the output data port tempROD from the RodSelector to the interface write variable tempROD in the RodSelector Remes mode (see Figure 5.4(c)). Similarly, we have mapped the output data port temp from the HCController component to the interface write variable temp in the HCController Remes mode, and we have mapped the input data port temp of the RodSelector to the interface read variable temp in the RodSelector Remes mode.

5.3.3 PTA formal modeling and analysis of the TCS

We have analyzed the Remes–based TCS, as a network of PTA models, in Uppaal Cora. We have obtained the PTA models presented in Figure 5.5 by applying the transformation rules stated in Section 4.3, automatically, by using the Remes tool-chain introduced in Chapter 6. We have transformed the Remes modes Clock, HCController and RodSelector into PTA models depicted in Figures 5.5(b), 5.5(c) and 5.5(d), respectively. The Init automaton shown in Figure 5.5(a) serves for the system startup of the TCS. It has been added to the PTA model of the TCS as a result of the transformation from Remes into PTA. Note that we have used component triggering information from the ProSave TCS model, to insert appropriate synchronization channels into the resulting PTA. We have mapped the trigger ports activate and heat_cool into synchronization channels activate and heat_cool, respectively. For example, in the HCController automaton the synchronization channel activate has been added on the edge connecting the locations Init and Idle. This channel is used for synchronization between the automata Clock and HCController. The interface read variable tempROD from the HCController mode and the interface write variable tempROD from the RodSelector mode are together mapped to one global variable tempROD in the PTA model of the TCS. Similarly, the interface write variable temp from the HCController

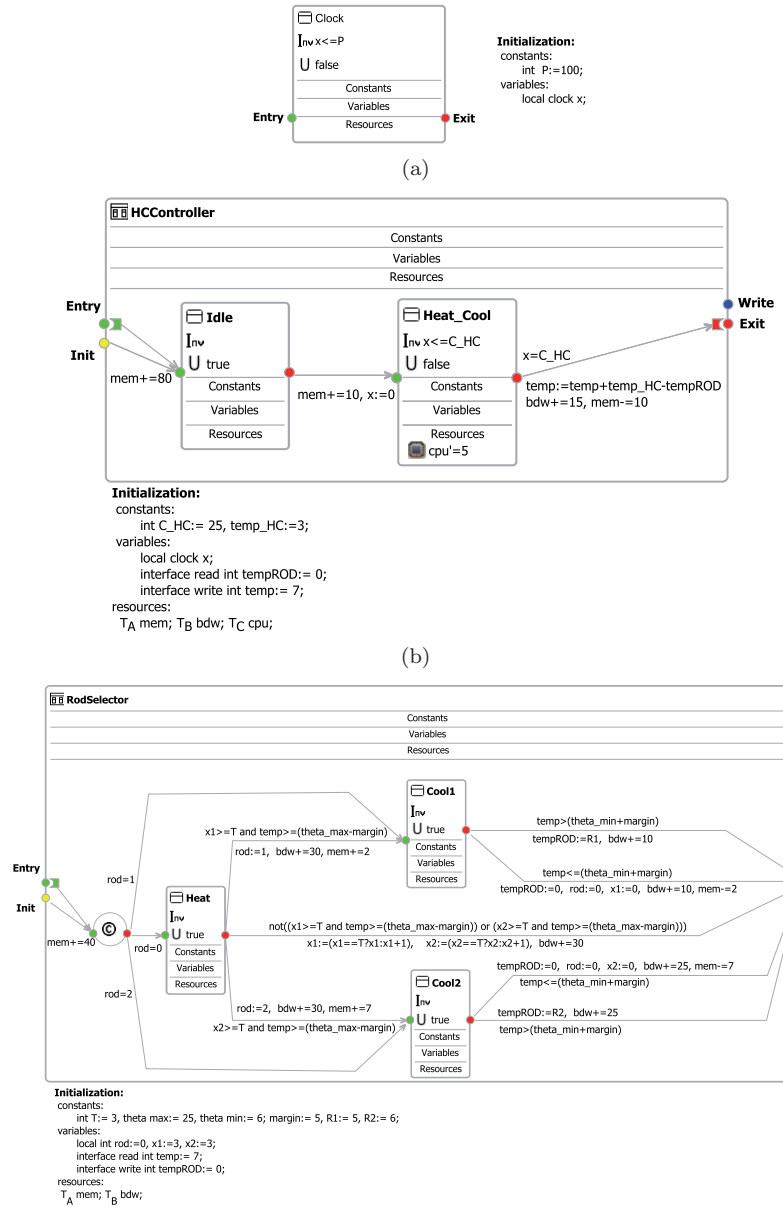


Figure 5.4: The Clock, HCController, and RodSelector modes of the TCS depicted in Figures (a), (b), and (c), respectively.

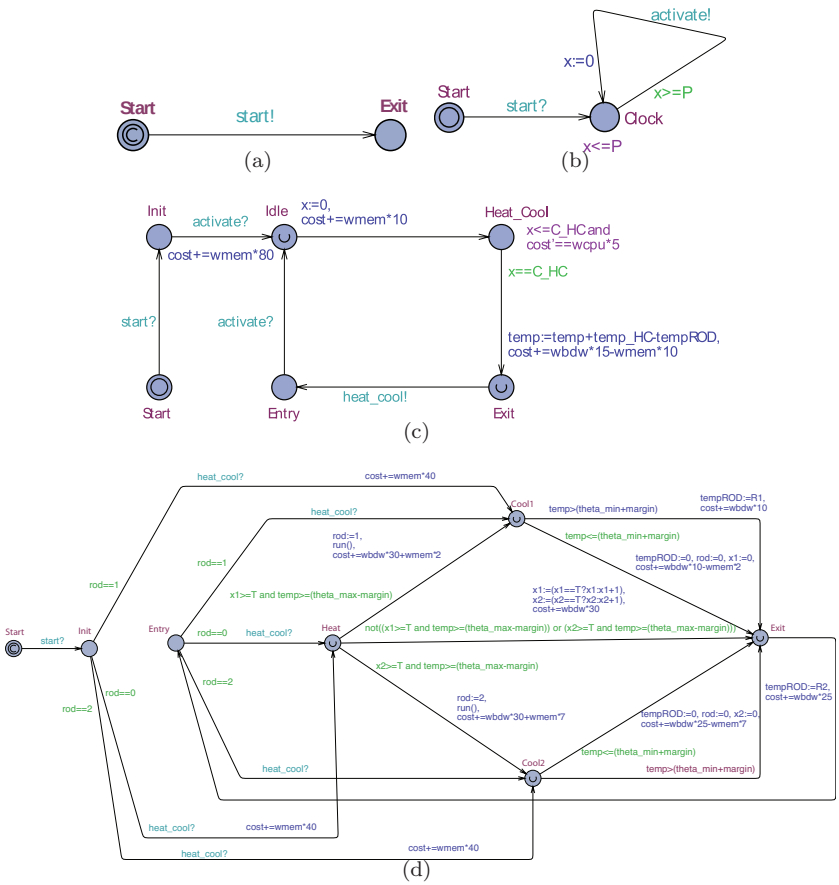


Figure 5.5: The TCS modeled with four PTA: (a) Automaton for initialization of the TCS system, added as a result of the automatic transformation from Remes into PTA, (b) The Clock connector as a PTA, (c) The HCController component as a PTA, and (d) The RodSelector as a PTA.

mode and the interface read variable temp from the RodSelector mode are together mapped to one global variable temp in the PTA model of the TCS. The TCS declared variables and their initial values are shown in Table 5.2.

Scope	Declarations

Global	const int wbdw = 2, wcpu = 5, wmem = 1;
	int temp = 7, tempROD = 0;
	chan heat_cool, activate;
	broadcast chan start;

Clock	const int P = 100; clock x;
HC controller	const int C_HC = 25, temp_HC=3 clock x;

Rod selector	const int T = 3, theta_max = 25;
	const int theta_min = 6;
	const int margin = 5, R1 = 5, R2 = 6;
	int rod = 0, count = 0, trace[3];
	int x1 = 3, x2 = 3;

Table 5.2: Declarations of the TCS PTA model.

Clock, HCController and RodSelector wait in location Start for the system startup. The locations Init, Entry and Exit have been created from the init-, entry- and exit points of the HCController and RodSelector composite modes. The return edge from Exit to Entry location ensures the cyclic behavior of the HCController and RodSelector automata. The conditional connector from the RodSelector mode has been removed in the transformation, and the edges entering the conditional connector have been combined with the edges exiting the conditional connector. As a result there are three outgoing edges from the Init and Exit locations of the RodSelector automaton (see Figure 5.5(d)).

For analysis purposes, we have added the TCS model with the function run() (see Figure 5.5(d)) that merely stores the first few selections of rods, in an array of integers.

In the weighted cost function that represents the analysis model for REMES, we have encoded the relative importance of the resources CPU, bdw and mem. We consider CPU to be the most critical resource followed by bdw and mem, so we give the highest weight to CPU in the cost

function. The cost of resource usage is influenced by the individual weights of each resource, and the consumed (utilized) resource on each edge or location. Currently, Uppaal Cora can only model-check PTA models where the cost function is monotonically increasing. Therefore, in order to keep the cost monotonically increasing we had to fine-tune the weights of the resources.

The analysis model of the TCS system is the following cost function:

$$c_{tot} = \mathsf{wcpu} \times c_{cpu} + \mathsf{wbdw} \times c_{bdw} + \mathsf{wmem} \times c_{mem}$$

where $\mathsf{wcpu} = 5$, $\mathsf{wbdw} = 2$ and $\mathsf{wmem} = 1$, and c_{cpu}, c_{bdw} and c_{mem} are the accumulated used amounts of CPU, bdw and mem, respectively.

Before embarking upon formal analysis of the TCS, we check the absence of deadlocks, property specified in Uppaal as follows:

$$A\,G\,\neg\, deadlock \tag{5.1}$$

We have also checked the model against the following safety properties, that basically capture the requirements set on the temperature, with respect to its upper and lower bounds:

$$A\,G\,(temp \leq theta_{max}) \tag{5.2}$$
$$A\,G\,(temp \geq theta_{min}) \tag{5.3}$$

After verifying the above properties on the PTA model of the TCS, by model-checking it with Uppaal Cora, we proceed to studying the minimum cost reachability problem, that is, to compute a model execution trace having the lowest possible resource cost. In our case, we have been interested in finding an execution order of the system (a cheapest sequence of rod insertions) that results in the lowest possible total resource cost, that is, to minimize c_{tot}. Such information extracted from the analysis could be used in the implementation stages of the TCS system, by resolving existing non-determinism in such a way that a specific execution trace, the cheapest with respect to total resource usage, is enforced.

For illustration, let us assume that both rods are available for cooling, and check for an optimal trace in which rods are inserted into the reactor three times, expressed as the following reachability property:

$$E\,F\,(count == 3) \tag{5.4}$$

Uppaal Cora has found that the second rod should be inserted two times in a row, followed by the first one, the third time. Table 5.3 shows the cost of this best trace, and also the cost of two other more expensive traces. Note that in our model the availability of both rods is always ensured by the chosen model parameters, presented in Table 5.2. We have noticed that, for the chosen parameters, it is always cheaper that the second rod is inserted in the core, even though the cost for selecting the second rod (trace Heat $\rightarrow$ Cool2 $\rightarrow$ Exit) is higher than the one for the first rod (trace Heat $\rightarrow$ Cool1 $\rightarrow$ Exit). This fact is a consequence of the higher cooling rate of the second rod as compared to the one for the first rod.

Scenario	Order of execution	Cost
1	P_2-P_2-P_1	15157
2	P_1-P_2-P_1	16357
3	P_1-P_1-P_2	17562

Table 5.3: Cost of execution for different rod insertion scenarios.

Discussion. For the TCS, we could only partially tackle the trade-off resource analysis problem, by giving the highest weight to the most critical resource, the CPU. We have also, by hand, conducted optimal conditional reachability resource usage analysis, by minimizing the mem consumption, while imposing upper bounds on the CPU consumption, in the TCS. For instance, for assuming three sequential insertions of the rods in the reactor's core, it might happen that it is necessary to insert the second rod three times in a row, in order to satisfy all constraints, even though the total cost is higher for such a trace than for the best execution trace.

5.4 Example: A Turntable Drilling System

As our second example, we consider the turntable drilling system previously described by, e.g., Bos and Kleijn [28] and Bortnik et al. [27]. The

system, depicted in Figure 5.6, consists of a rotating table that moves products between processing stations where they are drilled and tested, and then removed from the table once they pass the test.

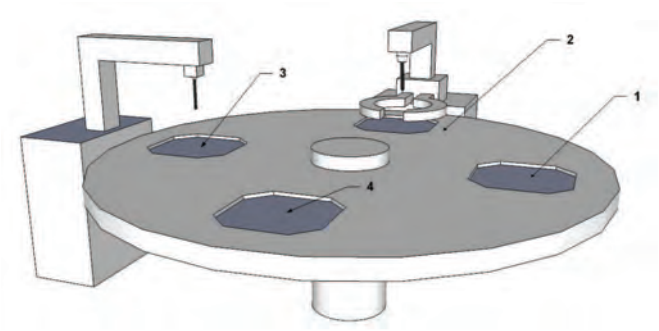


Figure 5.6: The turntable system (load and unload stations are not shown).

The load station places new products on the table (1), after which they are moved to the drill station (2) by rotating the turntable 90°. Drilling requires that the product is securely held in place by a clamp mechanism. After drilling, the product is moved to the testing station (3) where the depth of the drilled hole is measured. Finally, the unload station (4) removes the product from the table, provided that it passed the test. If not, it remains on the table to be drilled and tested again. The turntable has four slots, each capable of holding one product. Thus, the stations can operate in parallel, so that while the first piece is being tested, a second piece can be drilled, and a third piece loaded, etc.

5.4.1 Architecting the turntable in ProSys

We model the turntable drilling system in the ProCom component model, with five ProSys subsystems – Loader, Unloader, Turntable, Driller and Tester – as depicted in Figure 5.7. We assume that the Loader, Unloader and Turntable components can be reused from a previous project. In order to ensure the synchronization between the stations and the table, e.g., guaranteeing that the table turns only when no processing station is operating, an additional subsystem is needed: the Controller component.

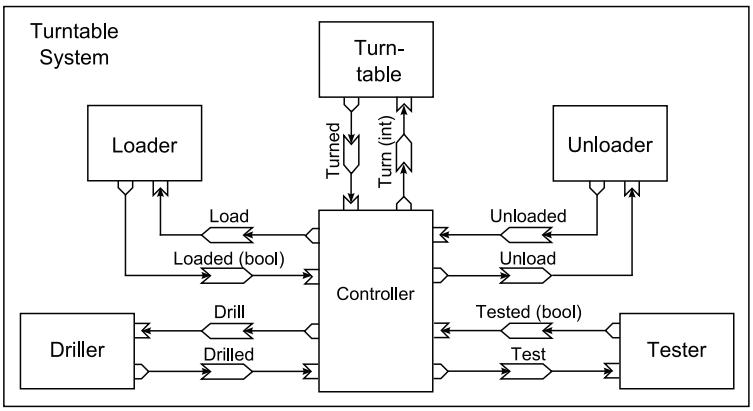


Figure 5.7: ProCom design of the turntable system.

The latter keeps track of the current status of the four slots, and activates the four stations and the turntable accordingly, by sending messages and receiving messages back once they are done.

The Tester and Driller components have similar interfaces; an incoming message telling the station to start processing, and an outgoing message indicating that it has finished. The output message of Tester also contains a boolean value representing if the test has succeeded or not.

It is possible to further decompose each of these ProSys components into either smaller ProSys components or into ProSave components according to the level of complexity of the functionality, and the potential for distribution. Before doing that, however, the developer may want to validate the feasibility of the design so far. Some properties can be analyzed from the ProCom design alone, for example that connected ports and channels match. However, in order to reason about properties, such as functional correctness, timing and resource consumption, we need to model the behavior of the components identified so far.

5.4.2 Behavioral modeling of the turntable in Remes

We model the functional, timing and resource usage behavior of the turntable components in Remes. Since the Loader, the Unloader and the Turntable components are reused, they already have behavioral models

in an assumed repository, whereas the remaining components await for Remes behavioral descriptions. In the following we present the Remes models of the Driller, the Tester and the Controller components, depicted in Figures 5.8, 5.9, 5.10, respectively.

The Driller component is responsible for moving the drill up and down, and for locking ad unlocking the clamp. In order to do this, it reads values from the drill and clamp sensors, modeled by boolean variables sdu (drill in upmost position), sdd (drill in downmost position), scl (clamp fully locked) and scu (clamp fully unlocked). We assume that these values are set or reset by hardware devices, whose behavior we do not model. We also assume that the Driller component contains three actuators modeled by variables drill_power (can be in states on or off), drill_position (can be in states down or idling), and drill_clamp (can be in states lock or unlock). Neither of the two message ports of Driller carries values, and thus they are mapped to two boolean variables Drill and Drilled, as described in Section 5.1.2.

The Driller remains in the non-lazy mode Idle until receiving a Drill message. When this happens, the component goes through a sequence of submodes: Clamp_locking, Driller_moving_down, Driller_moving_up and Clamp_unlocking. Each of these submodes is exited as the result of a sensor value turning true. When exiting the last submode, a Drilled message is sent, indicating that the operation is finished.

This Remes model also models Driller subsystem's energy consumption. We assume the following: powering the Driller consumes eng_pow units of energy per time unit, locking or unlocking the clamp consumes eng_clamp units of energy per time unit, and drilling consumes eng_drill units of energy per time unit. Moreover, we assume that the time of each Driller operation cycle is bounded to the interval $[\mathsf{tdrill1}, \mathsf{tdrill2}]$.

The Tester component is responsible for testing the quality of the drilled products. In order to do this, it reads values from the testing sensors, modeled by two boolean variables stu (tester is in up position) and std (tester is in down position). The input message port of the Tester does not carry a value, so we map it to a boolean variable Test. On the other hand, the output message port of the Tester carries a value of type boolean, so we map it to two boolean variables Tested and Tested_value. The Tester component remains in the non-lazy mode Idle until receiving a Test message. When this happens, Tester moves to mode Moving_down and starts testing the quality of a given product. We assume that the time spent in the mode Moving_down in bounded to

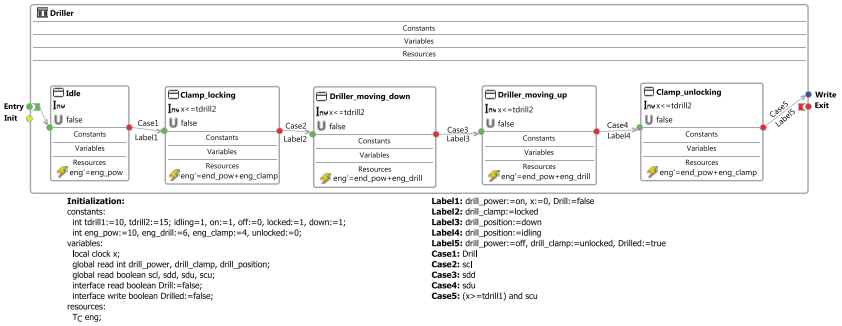


Figure 5.8: The behavior of the Driller component modeled in REMES.

the interval [move_down, timeout]. If the product is not properly drilled the testing finishes unsuccessfully. In that case the Moving_up1 mode is visited and when this mode is exited, a boolean message with a value false is sent via the message port Tested. If the testing finishes successfully (i.e., std and stu become true) a boolean message with a value true is sent to the Controller through the message port Tested.

The Controller component, depicted in Figure 5.10, keeps track of the states of the four slots and operates the stations and the turntable accordingly by exchanging messages with all of them. The behavior defined by the REMES mode consists of two main submodes, one in which the controller waits for messages from the stations, and one waiting for the turntable to finish turning.

The submode Wait_for_turning is exited when the Turned message arrives. Depending on the current state of the four slots, messages are sent out to the respective station. This is managed by the four urgent modes and the guards Case9, . . . , Case17. For example, the Load message is only sent if the first slot is empty, and the Drill message is only send if the second slot is occupied. The local variables, signal_loader etc., are used to keep track of what messages have been sent. When all messages are sent, the history variable h is assigned to the value Wait_for_stations, and the Write point is visited. Thus, the Controller will be immediately reentered in the submode Wait_for_stations.

In submode Wait_for_stations, the Controller waits until it receives a reply to one of the messages sent. Since this is a non-lazy mode, it must

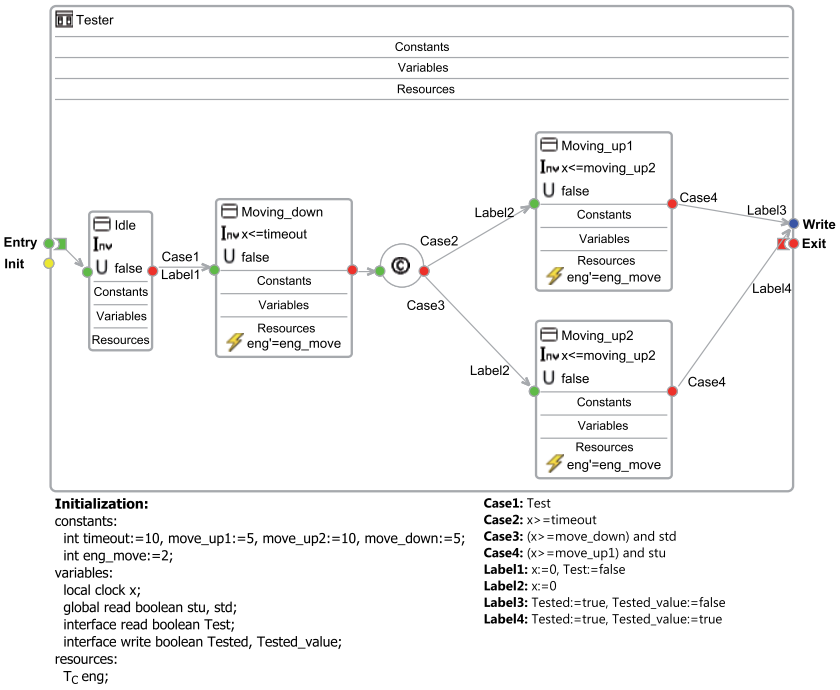


Figure 5.9: The behavior of the Tester component modeled in Remes.

be exited as soon as the guard of one of the outgoing discrete actions Case1, . . . , Case7 is satisfied. If the message carries a value (which is the case for Loaded and Tested), it is used to update the state of the corresponding slot. When all messages have been received, the message Turn is sent to the Turnable, and the history variable is set to Wait_for_turning before exiting through the Write point, meaning that the execution will be immediately resumed in that submode.

5.4.3 PTA formal modeling and analysis of the turntable system

We have analyzed the model of the turntable system, transformed into a network of PTA models, in Uppaal Cora. The semantic translation

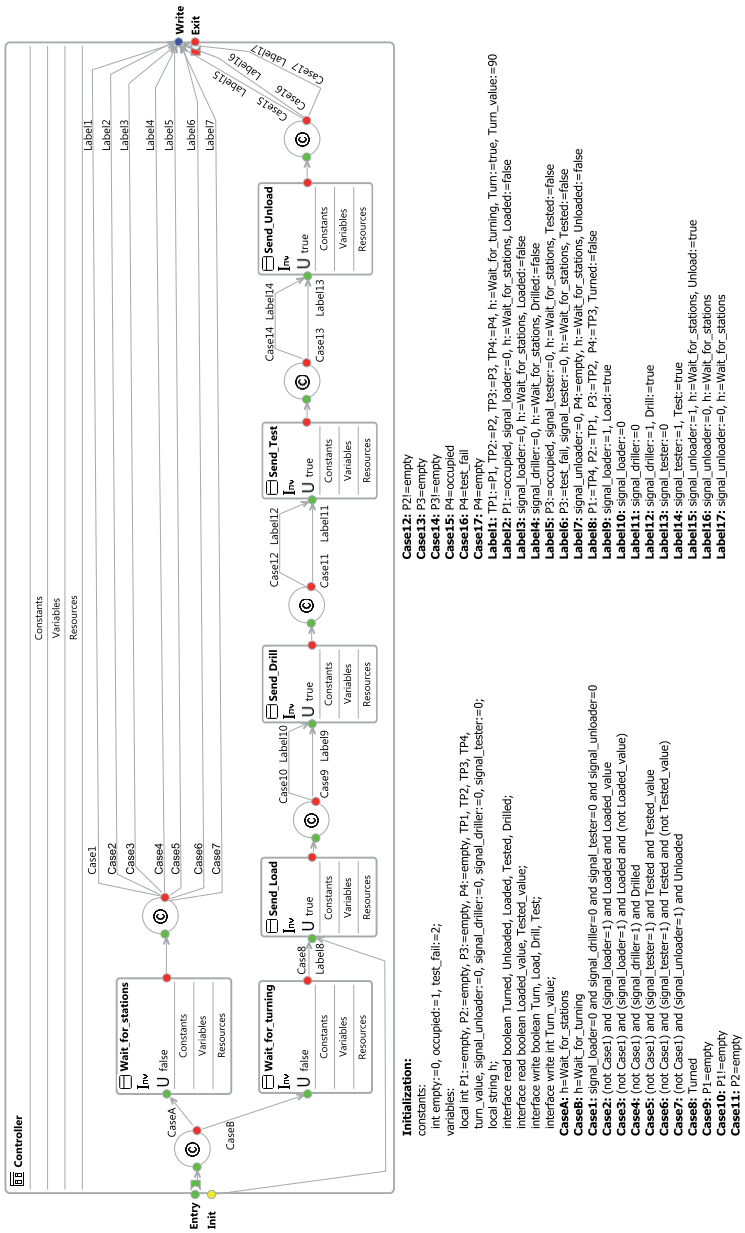


Figure 5.10: The behavior of the Controller component modeled in REMES.

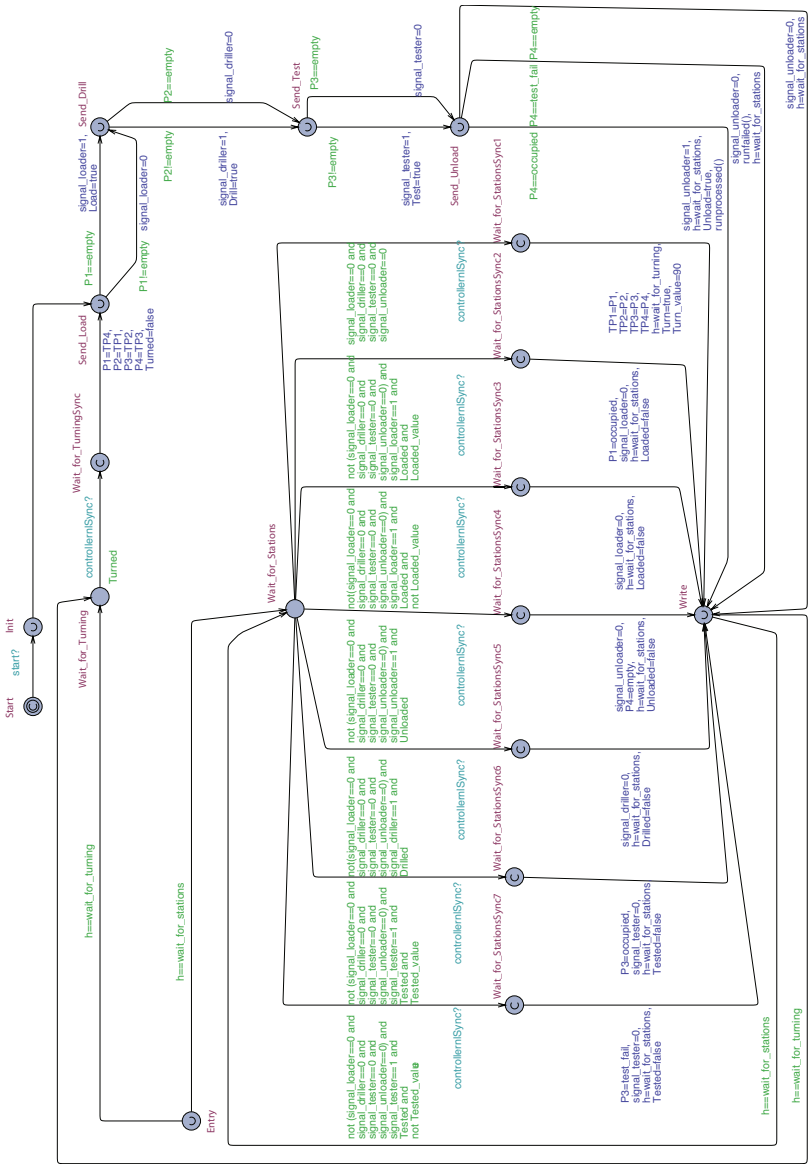


Figure 5.11: The Controller REMES mode translated to PTA.

from REMES to PTA is done by applying the transformation rules stated in Section 4.3. In the following, we present the transformed PTA model of the Controller mode, shown in Figure 5.11. The locations Init, Entry and Write have been created from the init-, entry- and write points of the Controller mode. The Exit location has been omitted from the model since there is no edge in the Controller mode that is connected with the exit point. Note that since Controller models a behavior of a ProSys component that does not wait for an activation from the environment, we have marked the Init and Entry locations as urgent ones . The return edge from Write to Entry location ensures the active behavior of the Controller component. The conditional connectors from the Controller mode have been removed in the transformation, and the edges entering a conditional connector have been combined with the edges exiting the same conditional connector. Every non-lazy submode has been translated to two locations: one location representing the submode and one additional committed location for each outgoing edge from the non-lazy submode. Observe, in Figures 5.10 and 5.11, that the Wait_for_turning mode has been translated in PTA into two locations: Wait_for_turning and Wait_for_TurningSync. The execution of Controller stays in the location Wait_for_turning until the guard Turned becomes true. This is ensured with the synchronization action controllernlSync? with an additional dummy PTA automaton (see Figure 5.10) over the urgent channel controllernlSync.

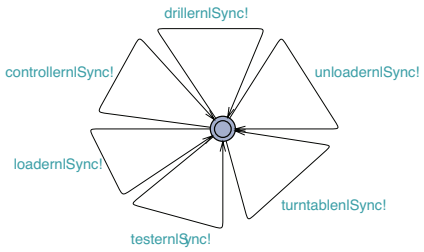


Figure 5.12: An additional automaton that ensures the semantics of the non-lazy modes in the turntable system.

For analysis purposes, we have added the sccmController PTA model with the functions runprocessed() and runfailed() (see Figure 5.12), which store the number of processed- and failed products, respectively.

After having provided Uppaal Cora with the PTA model of the turntable system, the last step before actually verifying the system design, is to formulate the desired system requirements as temporal logic formulas. Table 5.4 lists a few representative system requirements together with their temporal logic formulas. All properties are satisfied. Property 1 is a generic safety property, specifying the absence of a system deadlock, i.e., the system cannot reach a state from which it cannot continue operating. The turntable system is verified to be deadlock free. The next step is to verify that it satisfies the functional system requirements, here represented by properties 2 and 3. Properties 4 and 5 are examples of extra-functional properties, addressing time and resource usage, respectively. Uppaal Cora has calculated that the minimum energy consumption for processing five products is 1370 units.

♯	**System property**	**Temporal logic formula**
1	The system should be free from deadlocks.	$A\,G\,\neg\, deadlock$
2	A product must be clamped when drilled.	$A\,G\,(Driller.Driller\_moving\_down \Rightarrow$ $Driller.drill\_clamp == locked)$
3	The table should never turn when one of the stations is operating.	$A\,G\,(Turntable.Turn1 \vee Turntable.Turn2$ $\Rightarrow (Loader.Idle \vee Loader.Write)$ $\wedge\, (Unloader.Idle \vee Unloader.Write)$ $\wedge\, (Tester.Idle \vee Tester.Write)$ $\wedge\, (Driller.Idle \vee Driller.Write))$
4	Processing five products should never take more than 50 seconds (assuming at most one failed drilling).	$A\,G\,(\neg\, loaded\_failed \wedge time > 50 \wedge$ $failed\_products \leq 1 \Rightarrow processed\_products \geq 5)$
5	What is the minimum energy consumption for processing five products?	$E\,F\,(processed\_products == 5)$

Table 5.4: System properties of the turntable system.

Ideally, any analysis result of a component analyzed in isolation should be stored as an attribute of that component, as we have described in Section 5.2. In the turntable case, the second property in Table 5.4 holds for the Driller subsystem regardless of how the rest of the system behaves. The property could be packaged as a reusable attribute

of the Driller subsystem. However, the details of how such attributes are specified are outside of the scope of this thesis.

5.5 Summary

In this chapter, we have shown how the ProCom component model, specifically intended for architectural modeling of control-intensive embedded systems, can be combined with the Remes behavioral modeling language, in which functionality, timing and resource usage can be addressed together. In turn, this permits analysis of system level properties, while also supporting reuse of behavioral models when components are reused. To accomplish this, we have proposed a way of mapping the ProCom component interface onto the variables of Remes modes, such that the two models become connected. By packaging ProCom components together with their Remes behavioral models via a general attribute framework, we have addressed the important problem of model reuse. Transformations of Remes models into timed automata or priced timed automata, according to the transformation rules described in Chapter 4, allow for model-checking various properties, performed locally or at system level. We have demonstrated the connection between ProCom components and Remes modes on two examples: a temperature control system and a turntable drilling system. The relation between Remes models and other, simpler attributes should be investigated further, as well as the relation between the Remes model of a composite component and those associated with its subcomponents.

Chapter 6

The REMES Tool-chain

Based on the concepts described in Chapter 4, we have developed the REMES tool-chain [62, 87] that presents an Integrated Development Environment for construction and analysis of embedded systems behavior, modeled in the REMES language. It is built on the Eclipse Platform [98], which provides a common, familiar user interface. Our goal during the implementation of the tool-chain was to integrate it with the existing development environment concepts, and to reduce the learning effort. The REMES tool-chain consists of basic elements, such as meta-models, graphical editors, model transformations, and behavior testing support. This chapter gives an overview of the REMES tool-chain architecture and the main tools consisting the tool-chain (see Section 6.1), and presents the tool-chain workflow that can be used for modeling and analysis of REMES-based systems (see Section 6.2).

6.1 Overview of the REMES Tool-chain

The REMES tool-chain provides two sets of tools: (i) *behavioral modeling tools* for defining platform profiles (i.e., platform profile editor) and for building complex REMES models (i.e., REMES editor) and priced timed automata models (i.e., ULITE editor), and (ii) *analysis tools* for simulating REMES models and integration with UPPAAL and UPPAAL CORA tools to visually inspect transformed (priced) timed automata models and verify model properties. The tool-chain can be integrated with other architectural modeling- and external tools. Figure 6.1 shows the over-

all tool-chain structure. Most of the tools in the REMES tool-chain are integrated with the Eclipse environment and presented as actions and wizards to the user, e.g., wizards to run transformations. Some intervention is required to run certain tools, as the user interface is not yet fully developed.

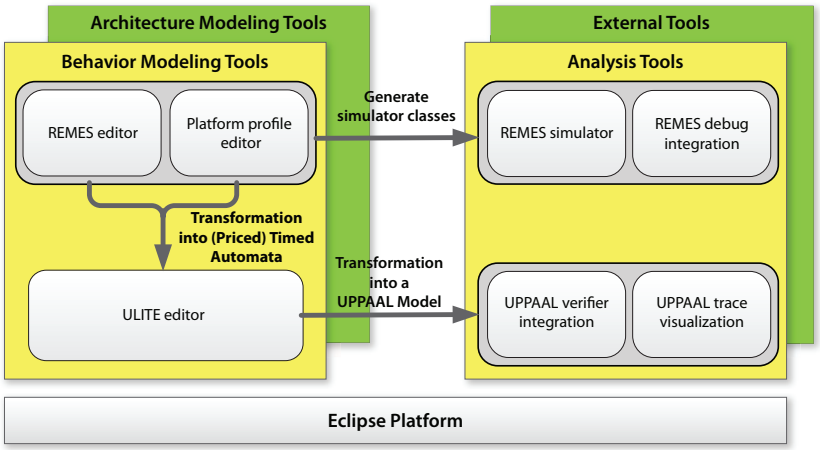


Figure 6.1: Overview of the REMES tool-chain.

6.1.1 Behavior Modeling Tools

The REMES tool-chain includes three editors: the REMES editor for modeling REMES behaviors, the ULITE editor for examining the generated priced timed automata (PTA) from the REMES into PTA transformation, and the platform profile text editor where the resources of the platform can be declared. The three editors are based on the Eclipse Graphical Modeling Framework (GMF) and the Eclipse Modeling Framework (EMF). The REMES and ULITE graphical modeling editors are presented in Figure 6.2, marked with ❶ and ❷, respectively.

Meta-models

We have used EMF to define the meta-models of REMES, ULITE and the platform profile. From the meta-models we have partially generated the

implementation code of the editors. We have specified two variants of the UPPAAL meta-model – the lightweight meta-model (ULITE) is used internally within the tool-chain, and the full UPPAAL meta-model is used to export models for verification in UPPAAL. ULITE is a subset of the full UPPAAL meta-model and includes just the timed automata model, without the diagram layout information presented in the full meta-model. ULITE contains resource cost declarations, which are transformed differently into models for UPPAAL and UPPAAL CORA. The two model variants, UPPAAL and UPPAAL CORA, share the same syntax, but UPPAAL CORA models include cost specification, which is not allowed in "plain" UPPAAL models. The REMES and ULITE meta-models can be found in Appendix A and B, respectively. Details about the platform profile specification can be found in Appendix C.

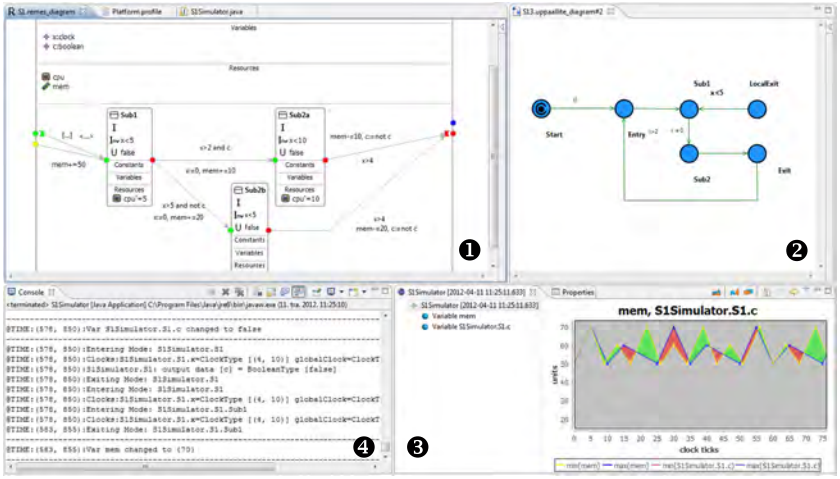


Figure 6.2: REMES ❶ and timed automata ❷ editors, simulator console output ❹ and simulator variable trace ❸.

The REMES editor

The REMES editor allows the user to easily create REMES artifacts, such as atomic and composite modes, edges or conditional connectors. Submodes and conditional connectors can be nested inside composite modes.

The user defines the control flow by creating edges between diagram elements. The action guards of the edges are defined by in-place editing in the diagram. The user can define typed variables, constants and resources in separate sections within the modes. The Remes diagram editor integrates with the Eclipse properties view, which displays and edits context sensitive information for the currently selected diagram element. Filters can be applied over the Remes diagram to outline a particular aspect of the Remes model – behavior, timing or resource usage. When the user saves a Remes diagram to a file, the guard and action expressions are parsed and checked for errors. The expression parser was built with the help of ANTLR [16] parser generator. Parsed expressions are stored with the model, ready to be used by other tools.

6.1.2 Analysis Tools

After modeling the behavior of a given system in the Remes editor, the Remes tool-chain can be used for analysis of that system behavior. The Remes tool-chain supports two types of analysis: simulation of Remes behaviors by using the Remes simulator, and formal analysis of Remes-based systems.

The Remes simulator

Simulating and testing system behaviors as they are being designed can provide valuable input to the system designer. Remes models can be tested with the Remes simulator (marked with ❹ in Figure 6.2), a standalone Java application integrated with the tool-chain using the Eclipse Platform Debug [99]. One of the steps of launching the simulator is to generate Java code used to represent the model within the simulator. Java code generation is performed with Acceleo [6]. When launching a Remes behavior in the Remes simulator, the system designer can specify the platform profile to be used (if not specified, a default profile will be applied). The system designer can then run the simulator which updates mode variables and resources in each simulation round, based on passed time. The system designer can visualize the mode transitions, the clock- and variable changes in the simulator output. In addition, the simulator can be configured to record changes of model variables to the trace database that can be visualized on a simple chart or used for later analysis. Figure 6.2 illustrates the user interface of the Remes

tool-chain. The screen-shot shows graphical editors for REMES ❶ and ULITE models ❷, the REMES simulator output in a console ❹ and variable value traces recorded from a previous simulator run ❸. Figure 6.3 shows the simulation started in debug mode when a behavior is executed step-by-step. Mode hierarchy ❺ displays active modes (highlighted in a model editor ❻), and variable values for a selected mode ❼. Since the REMES simulator is outside the scope of the thesis, for more information we refer the reader to [86, 87].

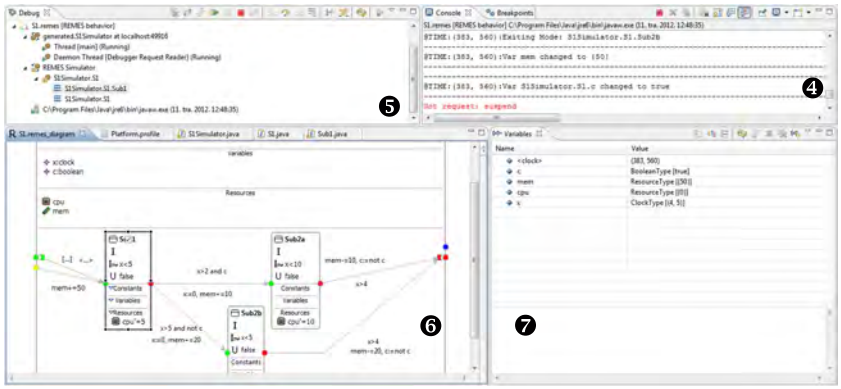


Figure 6.3: REMES testing interface, mode hierarchy ❺ and simulator console output ❹, mode highlight ❻ and variable inspector ❼.

Transforming REMES into priced timed automata

By following the transformation rules presented in Section 4.3 REMES diagrams can be transformed into ULITE diagrams to prepare for formal analysis. Model transformations, implemented with the ATL Transformation Language (ATL) [17, 64], transform models between different meta-models. We provide transformations between REMES and ULITE models, and between ULITE models and UPPAAL or UPPAAL CORA models. A given verification expert can inspect and tune the timed automata obtained from the transformation in the ULITE graphical editor. Once the model is ready for formal analysis, it can be exported to an UPPAAL model format, with the provided transformation actions. The verification engine distributed with UPPAAL can be started directly from the

tool-chain, with verification queries specified part of the Eclipse launch configuration. Simulation of the model, similar to the one in UPPAAL is also integrated, using the Java API for UPPAAL. Simulation traces are visualized and can be executed in steps, shown in Figure 6.4. Figure 6.4 displays part of the UPPAAL integration in the REMES tool-chain. UPPAAL simulator trace is shown as a list of state transitions ❽, and as a graphical trace ❿. Values of UPPAAL variables can also be inspected in ❾. The example model shown in the figure comes from a validation case-study that we describe in Chapter 7. We envision that the information about the relative importance of the resources of the platform (i.e., their weights) in the REMES into PTA transformation will be provided by the platform profile. However, this process is not yet automated, and currently we expect that the verification expert manually enters the information about the weights in the transformed UPPAAL priced timed automata model.

Figure 6.4: UPPAAL integration, showing a simulation trace with enabled transitions ❽ and variables ❾, and a graphical simulation trace ❿.

6.1.3 Integration between ProCom and REMES

Finally, we have integrated the REMES tools with the ProCom Integrated Development Environment (PRIDE) [26], which provides component triggering information from ProCom architectural models that can be used to insert appropriate synchronization channels in the transformed priced

timed automata models. The integration between the ProCom component model and the Remes behavioral language we have done via the Attribute Framework [92] and the QVT [40] transformation language, as we have described in Chapter 5. We had as goals to provide mapping between a ProCom component and a Remes model, defining the component's behavior, and to provide generation of a template Remes model out of the ProCom component. Therefore, in Pride we have used QVT model-to-model transformation to create blank Remes templates for ProCom components. Each template contains one empty composite mode with pre-created variables matching the ProCom component ports, resulting from the mapping by convention. The generated Remes model acts as a stub, which is later edited by a system designer or a verification expert. We refer the reader to [61] for a thorough description of the integration of the Remes tool-chain in Pride.

6.2 Workflow of the Remes Tool-chain

The Remes tool-chain implements a workflow based on two user roles: a system designer role, and a verification expert role. The system designer uses the Remes tool-chain for modeling and simulating Remes behaviors. As such, this role is similar to that of a software modeler/developer, and is focused on defining behavior models in Remes and simulating/testing these behaviors. The responsibility of the verification expert role is to use the Remes tool-chain for formal analysis of Remes behaviors. Note that the two roles are not necessarily always represented by different users. However, we envision that the verification expert role is dedicated to persons that have deeper knowledge in formal analysis.

Figure 6.5 shows the workflow of the tool-chain, split in four steps: when the tool-chain is used by the system designer for simulating/testing Remes behaviors, and when the tool-chain is used by the verification expert for formal analysis of Remes behaviors. We envision that the tool-chain will be first used for simulation of Remes behaviors to see whether they perform as expected, and once they perform correctly they will be transformed into PTA for formal analysis.

For illustration, suppose that the system designer works on defining a system's behavior. The system modeling starts with the specification of the system architecture, using an architectural language (such as the ProCom component model). The system designer specifies the archi-

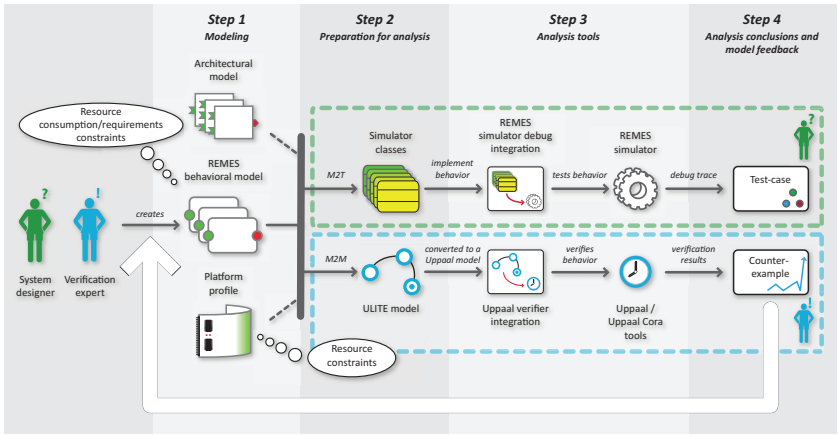


Figure 6.5: Workflow of the REMES tool-chain.

tectural elements of the system, and assigns a REMES behavioral model to each element. She models behaviors in the REMES graphical editor (**Step 1**). Once the behaviors are defined, the system designer can start the behavior similar to launching a program written in a programming language – either in run or in debug mode of the REMES simulator. Launching a behavior prepares the model for simulation and generates the code that implements the behavior in the simulator with a model-to-text transformation (M2T) (**Step 2** for the designer). The REMES simulator starts the behavior in run or debug mode, and displays mode transitions in a text form (**Step 3** for the designer). During debugging, a hierarchy of active modes is shown in the debugger interface, and the system designer can inspect mode variables (e.g., resources and clocks), execute a behavior step-by-step and track active modes in REMES diagrams. Finally, after each test run of the model, the system designer can conclude if the model performs as expected and correct it accordingly (**Step 4** for the designer).

Now let us present the part of the workflow when the REMES tool-chain is used by the verification expert. The verification expert starts by modeling the behavior of every component of the system in the REMES editor (**Step 1**). Once the system model is complete (**Step 1**), the verification expert can select to transform the REMES behavior into a network of priced timed automata using an automated model-to-model

(M2M) transformation. The verification expert then reviews the result of the transformation and modifies it if necessary, in the ULite graphical editor (**Step 2** for the verifier). Once custom changes are made, the verification expert proceeds with generation of the final Uppaal file. The transformation into Uppaal Cora is an extended version of the ULite into Uppaal transformation, where the costs are inserted in the generated Uppaal file. Resulting model files can be submitted for verification to the Uppaal verification engine directly from the tool-chain (**Step 3** for the verifier). The Uppaal verifier integration allows the verification expert to start the verification using the familiar Eclipse launch mechanism, specify verification queries, and inspect the automata trace within the tool-chain in an interface similar to the one implemented in Uppaal. Verification results can either confirm that the model conforms to the system requirements, or produce a counter-example to correct the model (**Step 4** for the verifier).

6.3 Summary

We have presented the Remes tool-chain, a set of tools that can be employed for construction and analysis of embedded system behavioral models. The core elements of the tool-chain are as follows: (i) the Remes editor for modeling behaviors of embedded components, (ii) the Remes simulator to test timing and resource behaviors prior to formal analysis, and (iii) an automated transformation from Remes into priced timed automata, needed for formal analysis. Our ongoing work on the Remes tool-chain includes adding deeper validation with helping tips, adding more strict variables declaration and usage, and improvements on the transformation rules implemented in the tool. The concrete usage scenarios in the practice might outline new requirements, where the transformed models might be enriched in order to decrease the user manual efforts.

Chapter 7

Case Study: Ericsson Nikola Tesla Demonstrator

The accuracy and effectiveness of modeling languages and analysis methods can be exercised by performing their validation against real-world application measurements. Validation loosely refers to the process of determining if a design is correct with respect to implementation requirements [41]. For model-based system design, validation establishes that the models capture the intended system behaviors accurately, and the analysis methods are effective if compared to measured system values (that is, their predictions match experimental/measured data). The two most usual model validation procedures are *simulation*, which traverses a subset of the system's behaviors, and *formal analysis*.

In this chapter we describe the modeling and formal analysis of a prototype industrial telecommunications system, a demonstrator developed by Ericsson Nikola Tesla, in Croatia [104, 111]. We present the applied verification and validation process in Section 7.1.

The Ericsson Nikola Tesla demonstrator is a proof-of-concept solution, intended to evaluate the horizontal system development paradigm. Hence, it has been developed by applying the component-based design paradigm, and by adding a newly developed authentication, authorization, and accounting service to a complex *basic service* telecom system.

The new service, called *extension service*, consists of several existing and reused components, such as the open-source load balancer called *Pen*, a number of servers, as well as standard communication protocols like, e.g., DIAMETER (see Figure 7.2). More precisely, the ENT demonstrator uses the basic service to issue call requests, to which the extension service delivers the authentication functionality on a request basis. The resulting system is a telecom system that must adhere to the characteristics of existing telecom legacy systems. Thus, two requirements are imposed on the demonstrator: capacity and optimal resource usage. We describe the demonstrator in Section 7.2.

Our validation effort consists of analyzing the above properties on a system model that uses code-measured values for its parameters, and compare the verification result to the one obtained directly on the code. The analysis effort is driven by both an academic, as well as an industrial interest. The former targets exercising the industrial applicability and validation of our REMES behavioral modeling language, and its underlying formal model in terms of priced timed automata networks. The industrial interest focuses on being able to use a virtual experimental "lab", in which various types of extra-functional analysis of the demonstrator are carried out, which could provide valuable feedback on the demonstrator's performance, and resource-usage, assuming various settings, respectively, prior to an actual implementation of the respective settings.

In Section 7.3, we model the demonstrator in a component-based fashion, by using the ProCom component model, whereas in Section 7.4.1 we model the functional, timing, and resource-wise behavior of the key components of the system in the REMES language. We build our model by using the timing and resource values extracted from the actual prototype implementation of the demonstrator, and provided by Ericsson researchers.

Regarding the system analysis, we consider, next to function and timing, a weighted sum of the resources CPU and memory, in which CPU is considered a more critical resource than memory (twice the relative weight of memory). Under this assumption, we derive an optimal system trace, the minimum time, and the minimal total accumulated weighted resource cost for processing a given number of system requests. We describe the results in Section 7.4.3.

7.1 Overview of the Verification and Validation Process

The verification and validation process that we have used in our case study is iterative, allowing feedback between steps. It consists of four steps (see Figure 7.1) as follows.

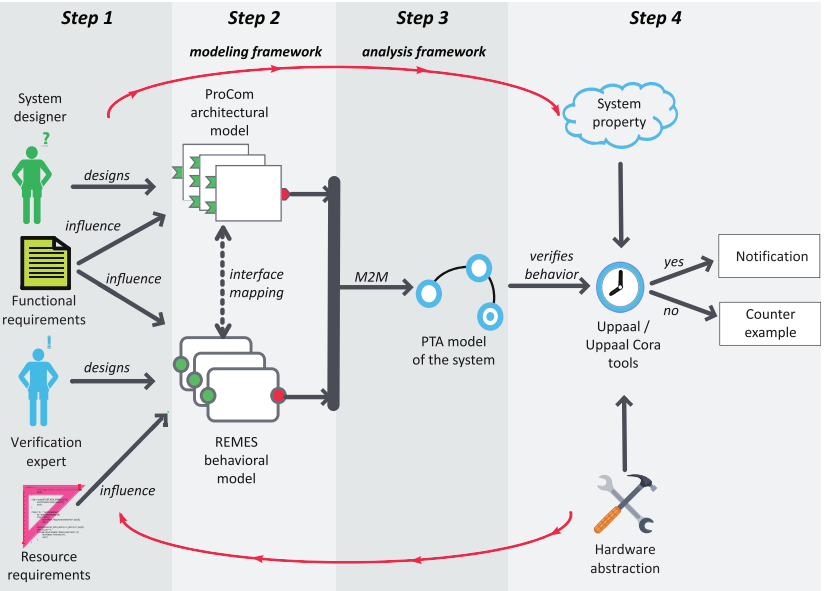


Figure 7.1: The system verification and validation process.

- **Step 1.** Based on the system functional requirements the system designer builds the ProCom architectural model of the system. Similarly, the verification expert uses both the functional- and resource requirements (such as timing, memory, etc.) to develop the REMES behavioral model of the system. In the ENT particular case, the timing constraints and resource usage information that we have annotated the REMES models with are those measured directly on the prototype implementation.

- **Step 2.** During this step an interface mapping between the ProCom architectural- and the REMES behavioral model is performed, as we have described in Chapter 5.

- **Step 3.** The ProCom architectural- and the REMES behavioral model are together transformed into a priced timed automata (PTA) model for formal analysis. The architectural model gives information about the order of execution of the REMES modes modeling the behavior of the components. The transformation is done by following the transformation rules presented in Section 4.3. After all needed transformation rules have been applied, one can visualize the transformation result and do eventual adjustments in the PTA model, if so needed.

- **Step 4.** In this step, we assume some hardware abstraction in the form of provided resources (e.g., memory budget, CPU load, bandwidth of the communication network, etc.). To perform model-checking, a PTA model of the system is fed into UPPAAL CORA, together with the hardware abstraction and a desired property (requirement) expressed in WCTL. UPPAAL CORA then verifies whether the property is satisfied or not, as we have described in Section 2.2. Out of the verification process we get timing information in the form of total time needed to handle a burst of calls, which is then compared to the measured value. This completes the validation process.

7.2 Description of the Demonstrator

Ericsson Nikola Tesla's (ENT) demonstrator is a prototype of a telecommunications system. It is designed according to current telecommunications industry's trends of adapting horizontal development (systems built from reusable components) methodologies instead of traditionally used vertical ones (systems built from ground-up in-house, now called legacy systems). The organization of the demonstrator is shown in Figure 7.2 from the perspective of its deployment architecture.

In the demonstrator, a new telecommunications service is created with horizontal development. This new service is added to the existing basic service that has been created over the years through vertical development. The basic service performs typical call control functionality:

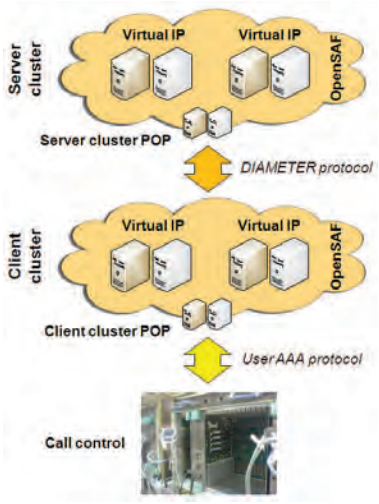


Figure 7.2: The deployment architecture of the demonstrator.

decoding of address information and routing calls from one end-point to another. When a special kind of processing is needed, the basic service generates events that result in requests (messages) that are being redirected into the extension service. The extension service processes messages generated by the basic service by performing an AAA (authentication, authorization and accounting) functionality that conforms to the widely accepted Internet standard called DIAMETER [81]. The result of the processing are also messages that are sent back to the basic service.

The extension service is realized by the *clients*- and *servers clusters*, which communicate via DIAMETER protocol. They should ensure high levels of performance through round-robin load balancing, and availability through redundancy. The implementation of high availability and reliability is facilitated by the use of an *OpenSAF* middleware. Previous experiments performed by Ericsson researchers show negligible impact of OpenSAF on the overall performance of the demonstrator [111]. Thus, we omit OpenSAF from experiments shown in this case-study.

Pen is a third-party open-source load balancer customized for balancing the load of stateful AAA protocol between the basic service and the

extension service. Pen maintains the information (e.g., IP addresses and ports) of which call control node is communicating with which DIAMETER *client* and uses round-robin method for choosing which client will serve a given request. DIAMETER client receives AAA requests through the AAA protocol between the basic service and the extension service, transforms them into DIAMETER-based AAA requests and sends the latter to the DIAMETER servers cluster.

DIAMETER *relay* is a DIAMETER protocol functionality that is used for balancing the load among DIAMETER servers. Similar to Pen, it uses the round-robin method to choose which server will serve a given request. Since each DIAMETER message contains full address information about communicating peers, it just transmits the response received from a DIAMETER server to the corresponding DIAMETER client that originated the initial request. DIAMETER *server* receives DIAMETER-based AAA requests sent by DIAMETER clients. It processes these requests and returns the results to the relay. Since the original request contains the information about which client has created it, the relay knows to which client the response must be sent to.

7.3 The ProCom Architecture of the Demonstrator

In this section, we describe the ENT demonstrator's software architecture, which complies to the ProCom component model. The architecture of the ENT demonstrator consists of two subsystems: Basic Service and Extension Service, as depicted in Figure 7.3. The interfaces of the subsystems are expressed in terms of message ports.

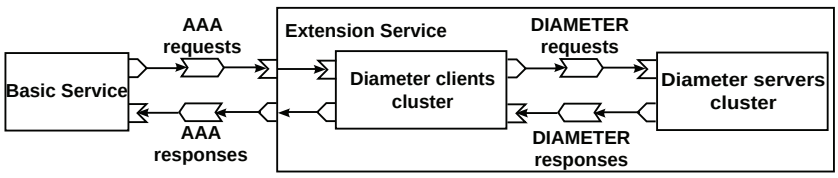


Figure 7.3: The ProSys model of the ENT demonstrator.

Basic Service is an existing legacy ProSys component. Extension Service is a subsystem composed of two smaller ProSys components: Diameter clients cluster and Diameter servers cluster. We consider that there are four clients, and four servers in Diameter clients cluster, and Diameter servers cluster, respectively. Each of these ProSys components may be further decomposed into either smaller ProSys components, or into ProSave components, depending on the level of complexity of the functionality, and the possibility for distribution. Accordingly, the Diameter clients cluster component can be internally modeled with a Pen ProSave component and four Client ProSave components. Similarly, the Diameter servers cluster component can be modeled with a Relay ProSave component and four Server ProSave components.

The communication between the components obeys the scheme that we have described in the previous section. The component Basic Service sends AAA requests to Extension Service. These requests are forwarded to Diameter clients cluster component. Inside this cluster, the Pen component forwards these messages in a round-robin fashion to each of the four clients. Inside Diameter clients cluster component AAA requests are transformed into DIAMETER requests and are forwarded to the Diameter servers cluster component. Relay, similarly to Pen, forwards the DIAMETER requests messages in a round-robin manner to each of the four servers. The servers process these requests and return DIAMETER responses to Relay that forwards them to Diameter clients cluster. In the end, Diameter clients cluster component transforms DIAMETER responses into AAA responses and sends them back to Basic Service.

Note that depending on the timing characteristics of the Pen component we can model it in two ways as follows.

- As a ProSave component with two services Pen_Input and Pen_Output, as we have described in Section 3.3.2. In this case, Pen_Input and Pen_Output can be triggered independently and they can run in parallel.

- As a ProSave composite component with one service that is internally built of two components Pen_Input and Pen_Output, and a selection connector that activates either Pen_Input or Pen_Output depending on the source of the input data. If a request has been sent from the Basic Service component then the Pen_Input is activated. On the other hand, the Pen_Output component is activated when one of the four Client components sends a response to the

Pen component.

Similar discussion applies for the Relay and Client components.

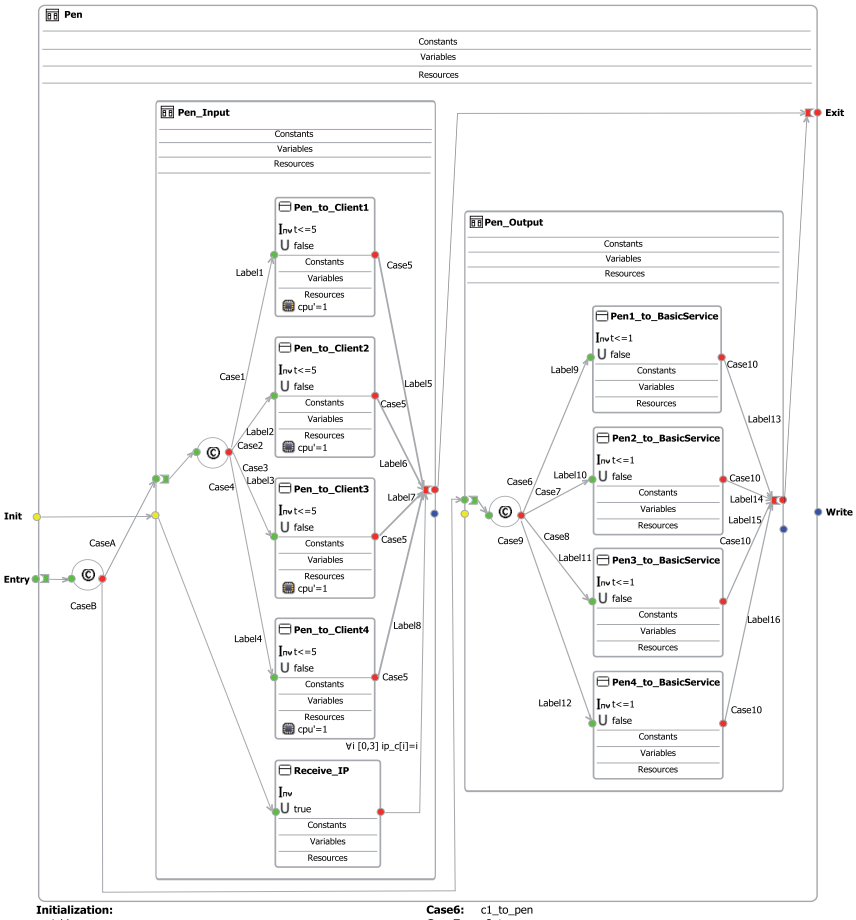
7.4 REMES Modeling and Formal Analysis of the Demonstrator

7.4.1 The REMES model of the ENT demonstrator

We model the functional, timing and resource usage behavior of the ENT components as REMES modes. In Figures 7.4, 7.5, and 7.6 we present the REMES models of the Pen component, one of the clients from Diameter clients cluster and one of the servers from Diameter servers cluster, respectively. The Relay component has a similar behavior to the Pen component.

When modeling the behavior of the ENT demonstrator, we have made a number of assumptions within the model, which we have discussed with the researchers at Ericsson and we have agreed upon. As such, we consider instantaneous reads for IP client addresses and other information by the Pen component. We also make an assumption on the linear response time increase per burst of requests (500) that lets us compute the total response time, hence the extension service capacity, once we have predicted the response time of an individual request. In the ENT demonstrator, we consider two resources in our analysis: memory and CPU. We assume CPU as a continuous resource, and we treat memory as a discrete resource. The timing constraints and resource usage information that we have annotated the REMES models with are those measured directly on the prototype implementation. Note that in the current version of the demonstrator the clients and the servers are homogenous, that is, the processing time is the same for each request, on any of them. Since the clients and the servers are homogenous, the Relay's round-robin load balancing protocol always sends messages coming from the first-, second-, third- and fourth- client to the first-, second-, third- and fourth server, respectively. From the measurements performed on the source code, we have concluded that Relay is the slowest component in the ENT demonstrator and the one that consumes the most resources.

The timing measurements performed on the source code of the ENT system have showed that the Pen component cannot receive requests and



Initialization:
variables:
  local clock t;
  local int counter;
  interface read boolean req, clients_to_pen;
  interface read int in_queue[], out_client1[], out_client2[], out_client3[], out_client4[], ip, inp, outp;
  interface write in_pen[], out_pen[], oc1, oc2, oc3, oc4;
  interface read boolean client1prio, client2prio, client3prio, client4prio;
  interface read c1_to_pen, c2_to_pen, c3_to_pen, c4_to_pen, penfree;
resources:
  T_A mem; T_C cpu;

CaseA: req
CaseB: clients_to_pen
Case1: client1prio and (counter mod 4==0)
Case2: client2prio and (counter mod 4==1)
Case3: client3prio and (counter mod 4==2)
Case4: client4prio and (counter mod 4==3)
Case5: t=5

Case6: c1_to_pen
Case7: c2_to_pen
Case8: c3_to_pen
Case9: c4_to_pen
Case10: t=1
Label1: in_pen[inp]:=in_queue[ip], t:=0, client1prio:=false, penfree:=false
Label2: in_pen[inp]:=in_queue[ip], t:=0, client2prio:=false, penfree:=false
Label3: in_pen[inp]:=in_queue[ip], t:=0, client3prio:=false, penfree:=false
Label4: in_pen[inp]:=in_queue[ip], t:=0, client4prio:=false, penfree:=false
Label5: mem+=2, ip++, penfree:=true, counter++, send_c1:=true
Label6: mem+=2, ip++, penfree:=true, counter++, send_c2:=true
Label7: mem+=2, ip++, penfree:=true, counter++, send_c3:=true
Label8: mem+=2, ip++, penfree:=true, counter++, send_c4:=true
Label9: out_pen[outp]:=out_client1[oc1], oc1++, t:=0
Label10: out_pen[outp]:=out_client2[oc2], oc2++, t:=0
Label11: out_pen[outp]:=out_client3[oc3], oc3++, t:=0
Label12: out_pen[outp]:=out_client4[oc4], oc4++, t:=0
Label13: mem+=2, c1_to_pen:=false
Label14: mem+=2, c2_to_pen:=false
Label15: mem+=2, c3_to_pen:=false
Label16: mem+=2, c4_to_pen:=false

Figure 7.4: The Pen component modeled in REMES.

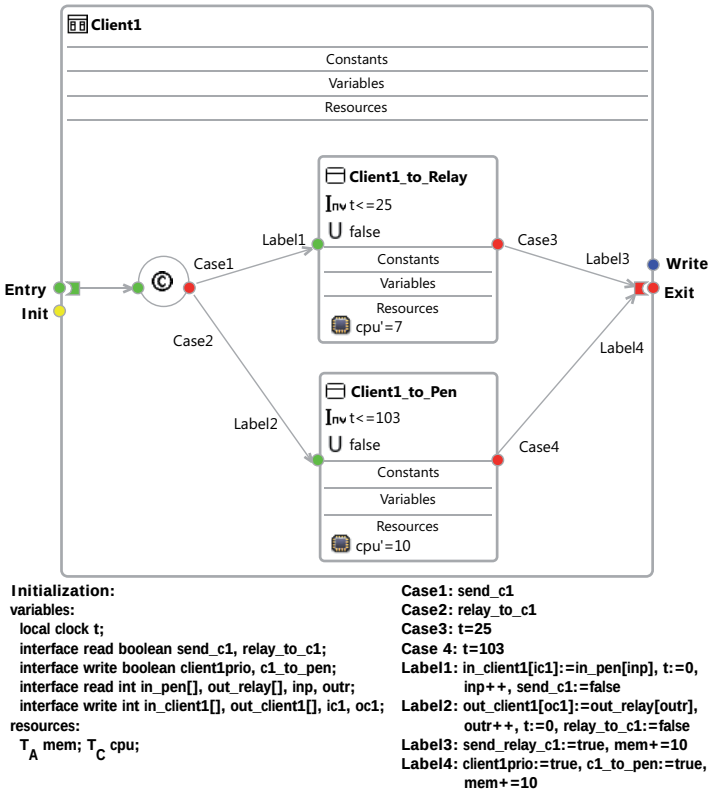


Figure 7.5: The Client1 component modeled in REMES.

send responses concurrently. Therefore, we have modeled the Pen mode as a composite mode containing two composite submodes: Pen_Input and Pen_Output. Pen starts executing by entering the Pen_Input mode and its atomic submode Receive_IP, where it reads instantaneously the addresses of the clients. Pen may be reentered in case the Basic Service component sends a new request (depicted with the boolean variable req) or in case one of the clients is ready to send a response (i.e., clients_to_pen evaluates to true). Basic Service may send requests to Pen only when Pen is free (captured by the boolean variable penfree).

The Pen_Input mode is responsible for sending requests to the clients in a round-robin fashion. We use the variable counter to ensure the round-robin principle. For example, Pen is ready to send a message to Client1 when the guard client1prio and (counter mod 4==0) evaluates to true. The Pen_Output mode receives responses from the clients and forwards them to Basic Service.

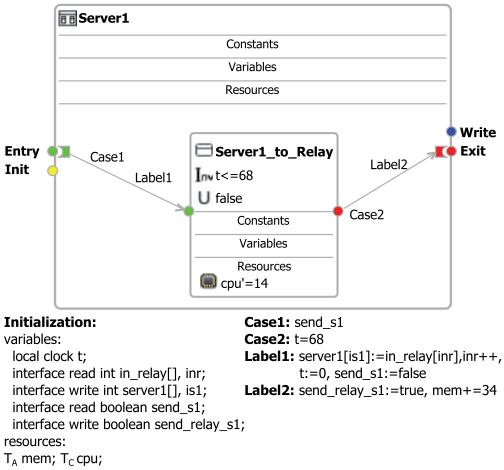


Figure 7.6: The Server1 component modeled in REMES.

The Client1 mode is a composite mode made of two atomic modes Client1_Relay and Client1_Pen. When send_c1 becomes true Client1 receives requests from Pen, processes them in the mode Client1_to_Relay, and forwards the processed requests to Relay. Later, when relay_to_c1 evaluates to true, Relay sends responses to the requests back to Client1. Client1 sends the responses to Pen. Client1 stays in the Client1_to_Relay or the Client1_to_Pen modes as long as their invariants hold (i.e., until $t \leq 25$ and $t \leq 103$, respectively). Note that Client1 can process only one request at a time. The fact that Client1 has to send back the response to Pen before receiving a new request is encoded by the boolean variable client1prio.

The Server1 mode receives requests from Relay, processes them, and sends them back to Relay. Server1 is entered when send_c1 becomes true. The Server1 mode is exited after 68 time units.

7.4.2 Formal analysis goals

The most important requirement imposed on the demonstrator is to ensure an acceptable performance of the extension service, that is, handling 100 calls per second. For this to happen, and assuming a linear timing behavior per bursts of requests, the end-to-end response time of, say, 500 AAA requests should be less or at most 5 seconds.

To verify this, and, at the same time, validate the abstract descriptions, we have considered in our behavioral models (REMES and the corresponding PTA) the actual source code measured values of the authorization request, and authorization answer response times, respectively, as well as their respective CPU load, and memory usage, for each component of the demonstrator: Pen, DIAMETER client, DIAMETER relay, and DIAMETER server.

Before embarking upon formal validation, we have checked the absence of deadlocks, property specified in UPPAAL as follows:

$$\mathsf{A}G \neg \mathit{deadlock}$$

By using the measured values of the demonstrator's extra-functional attributes, we aim at:

- model-checking the REMES system model's capacity (number of handled requests per second), as well as
- computing an optimal execution trace for the overall usage of resources (CPU and memory).

Verifying the demonstrator's capacity is a crucial performance requirement of the system, and we will next show that we actually deliver a performance guarantee, since model-checking is an exhaustive verification technique. To accomplish the response time verification, we define a global clock variable that stores the elapsed time from the start time of sending the authorization request to the Pen, until the request is served and returns to the call controller in the basic service. Computing optimal resource-aware traces relies on a weighted sum representation of the resource function, which accounts for both types of resources simultaneously, allowing the designer to set the level of criticality for each resource (identical weights meaning equal importance).

7.4.3 PTA model of the ENT demonstrator and analysis results

We have analyzed the REMES-based ENT demonstrator in UPPAAL CORA, by semantically translating it into a network of PTA models. The REMES to PTA transformation rules described in Section 4.3 cover a two-level hierarchy only, therefore we have carried out the translation of the REMES-based ENT demonstrator into PTA in three iterations. We have first flattened the hierarchical composite modes Pen and Relay into subcomposite modes made of atomic submodes. Then we have translated each of the subcomposite modes into PTA models, according to the rules introduced in Section 4.3. For instance, the composite mode Pen has been translated into two PTA models Pen_Input and Pen_Output. Finally, in the third iteration, we have compared the transformed PTA models with the respective REMES models, and performed some adjustments where necessary. Here, we present only the PTA models of the Pen_Input submode, the Pen_Output submode, the Client1 mode, and the Server1 mode shown in Figures 7.7, 7.8, 7.9, and 7.10, respectively. In the PTA model of the ENT system we have also added PTA models of different ProSave connectors. One such Control_Or connector that forwards each incoming trigger from the clients to the Pen component is depicted in Figure 7.11.

The PTA of Pen_Input has nine locations: Start, Init, Entry, Exit, Receive_IP, Pen_to_Client1, Pen_to_Client2, Pen_to_Client3 and Pen_to_Client4. The synchronization between Basic_Service and Pen_Input is modeled with the channel req. Similarly, the synchronization between Pen_Input and the four Client PTA models is modeled with the channel pen_to_clients. The selection of the clients is controlled by the variables client1prio, ..., client4prio, and counter.

The PTA of Pen_Output has seven locations: Start, Entry, Exit, Pen1_to_BasicService, Pen2_to_BasicService, Pen3_to_BasicService and Pen4_to_BasicService. The synchronization between the four clients and Pen_Output is modeled with the channel clients_to_pen. The synchronization between Pen_Output and Basic_Service, when a response has come from the Client components is modeled with the channel processed.

The PTA of Client1 consists of five locations: Start, Entry, Exit, Client1_to_Pen and Client1_to_Relay. The Control_Or PTA, presented in Figure 7.11, is in charge of forwarding the trigger signal c1_sending from Client1 to Pen_Output. The synchronization between the automata Client1 and Relay is modeled by using two channels: client1_to_relay (mod-

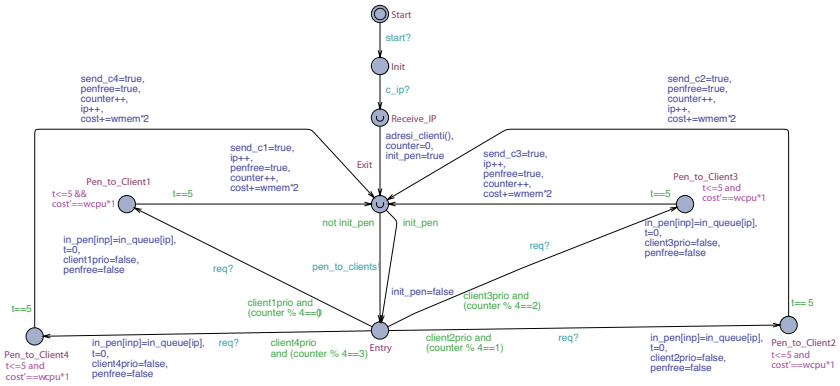


Figure 7.7: PTA model of the Pen_Input submode.

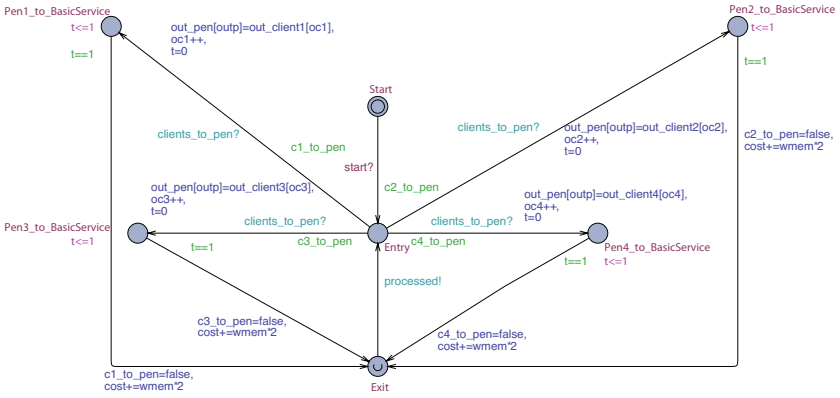


Figure 7.8: PTA model of the Pen_Output submode.

els requests sent from Client1 to Relay when boolean variable send_relay_c1 is true), and relay_to_clients (models responses sent from Relay to Client1 when boolean variable relay_to_c1 is true).

The PTA of Server1 consists of four locations: Start, Entry, Exit and Server1_to_Relay. The synchronization between Server1 and Relay is modeled by using two channels: relay_to_servers (for receiving requests from

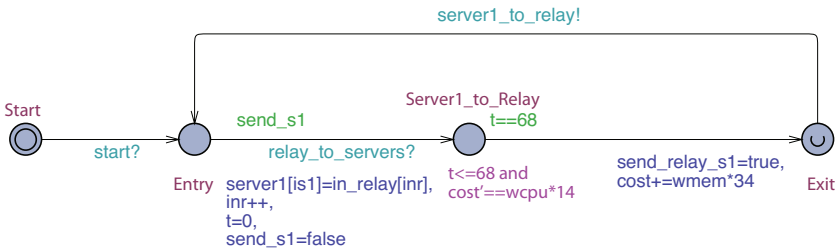


Figure 7.9: PTA model of the Client1 mode.

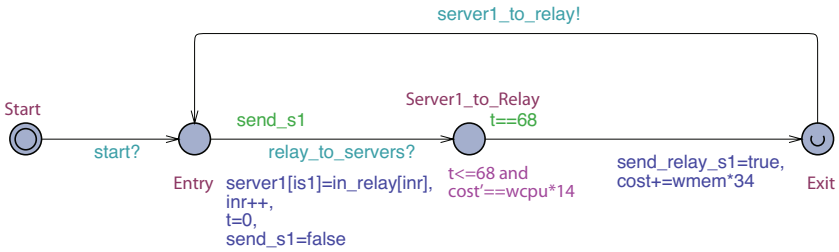


Figure 7.10: PTA model of the Server1 mode.

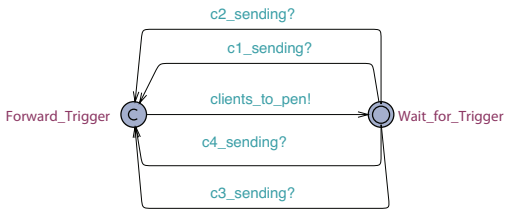


Figure 7.11: PTA model of the Control_Or connector that forwards each incoming trigger from the clients to the Pen component.

Relay) and server1_to_relay (for sending responses back to Relay).

In our analysis model, we consider CPU to be a more critical resource than memory. The cost model that we use is derived from the measurements carried out on the actual source code. The resource-usage cost is

influenced by the weights of CPU and memory, and the consumed resources of all edges and locations. In the ENT demonstrator, we consider the following total cost function

$$c_{tot} = \mathsf{wcpu} \times c_{cpu} + \mathsf{wmem} \times c_{mem}$$

where $\mathsf{wcpu} = 2$ and $\mathsf{wmem} = 1$, and c_{cpu} and c_{mem} are the accumulated consumed amounts of CPU and memory, respectively.

After providing Uppaal Cora with the PTA model of the ENT demonstrator, we were able to study the minimum cost reachability problem i.e., to find an execution trace of the system that uses the minimum possible total resource cost. For illustration, let's check for an optimal trace satisfying the reachability property:

$$\mathsf{EF}(\mathsf{processed\_messages}[3] == 4),$$

that is, a trace in which four requests are eventually processed by the ENT demonstrator. The cost of this best trace is 34628. We use the value 781 ms (from code measurements) as the time value needed for the basic service to process 500 requests.

From our analysis model we were also able to determine the time needed for processing a certain number of requests. For response time only, we have performed the analysis in Uppaal (in order to get the corresponding TA system model we have removed the costs from the PTA model). Uppaal has calculated that the time needed for handling 1 request is 3,42 time units (ms). If we consider that the processing time grows linearly, then for processing 500 requests our Uppaal model of the ENT demonstrator needs 1710 time units. This number is just slightly higher than the source code measured value, that is, 1690 ms. The verification result shows that the capacity of the demonstrator (extension service alone) is actually greater than the required 100 requests per second. Also, this result concludes our behavioral model formal validation, regarding the end-to-end response time, which has been a central design issue of the demonstrator.

7.5 Summary

In this chapter, we have presented a case study where our behavioral language Remes is applied to model and analyze a new telecommunication system by Ericsson Nikola Tesla. The new system is horizontally

developed by adding a newly developed authentication, authorization, and accounting service to a complex basic service telecom system consisting of several existing and reused components, such as a DIAMETER standard protocol, an open-source *Pen* load balancer, and a number of servers.

On the modeling front, we have shown how the system has been modeled in a component-based fashion using the ProCom architectural modeling language, and how the functional, timing, and resource-wise behavior of the key components of the system have been modeled in REMES. We have also shown how the combined model is semantically translated into a network of (priced) timed automata to enable model-checking in the tools UPPAAL and UPPAAL CORA.

On the system analysis front, we have, in addition to function and timing, considered a weighted sum of resources CPU and memory, for which designers have chosen to consider CPU as the most critical resource (twice the relative weight of memory). In this setting, we have computed an optimal system trace, with respect to the total accumulated resources, needed for processing a given number of system requests. In addition, we have also computed, by model-checking the TA models, the time needed for the extended ENT system to process 500 call requests, hence showing that the capacity requirement of the system is fulfilled. Allowing the system designers to gain deeper understanding in the system's resource behavior might prove valuable to further optimization steps, as well as in adjusting the resources provided by the underlying implementation platform, accordingly. Last but not least, the work and results described in this chapter have served as our framework validation basis, since most of our methods and tools have been exercised on a real-world industrial case-study.

Chapter 8

Related Work

This chapter relates the contributions presented in this thesis to relevant research and practice areas, subdivided into two sections. Our research work published in papers [37] and [103] contains extensive related work and state of the art so here we give only a short summary.

8.1 Component Models for Embedded Systems

Nowadays many component models exist, either general purpose or dedicated to a specific domain. Still, only a few component models target the development of embedded systems and most of them are dedicated to specific sub domains only. In these component models, component implementations are mostly given in C programming language and components are composed before compilation. The C-language provides more and easier access to details of operating system and underlying hardware platforms facilitating optimizations. Many component models targeting embedded systems use ports as the interface elements to exchange data. In port-based interfaces, input and output interfaces consist of ports that receive and send data, respectively (often designated as sink and source), hence corresponding to the concepts of provided and required interface. Often the component models for embedded systems are intended for applications of an algorithmic nature and these applications are commonly modeled as data- or signal-driven block diagrams. Another name for

this is pipe-and-filter architecture. Most component models targeting embedded systems focus primarily on "small" granularity components. Although they provide techniques for handling extra-functional properties there is still need for further research to improve the theories of specifying, modeling and analyzing extra-functional properties of components and composed systems, and to develop tool support. In this section we survey some component models that have been developed specifically for application in the embedded system domain and compare them with the ProCom component model proposed in this thesis. Special attention is dedicated to the capability of these component models to model and analyze extra-functional properties, in particular resource-related properties.

AUTOSAR (AUTomotive Open System ARchitecture) [18] component model has resulted from the cooperative research of a number of automotive manufactures and suppliers. The goal of AUTOSAR is to define a standardized platform for automotive systems facilitating the exchange of "elements" between different vehicle platforms and subsystem manufacturers. The main focus of AUTSOAR is the architecture, not the component model itself. Although some similarities with ProCom exist, such as the transparent communication between subsystems and components with the use of standardized interfaces, distribution of the functionalities provided by each subsystem across several nodes, some essential differences can also be noticed. In AUTOSAR, components are runtime entities whereas in ProCom they are considered at design time. In AUTOSAR subsystems are unaware of the characteristics of the underlying platform and not so much emphasis is put on analysis of the developed elements. The recent AUTOSAR 4.0 [19] release, influenced by the TIMMO project [1, 70], contains a meta-model extension for specifying timing properties and constraints of software components. As such, it allows expressing timing constraints, such as maximum delays, repetitions rates, synchronization, and data ages, by adding timing information to events and event chains.

BlueArX [67,68] is a component model developed and used by Bosch for automotive systems, such as engine control systems or chassis systems. Each component consists of specification, documentation and implementation and has interfaces, which are divided into two types import and export interfaces where import interface are required and export interfaces are provided by the component. There is also a special type of an interface, called analytic interface, which is used to store compo-

nents's extra-functional properties (such as worst-case execution times, code memory, stack memory, and data memory). Input to the analytic interface is the current semantic context, such as hardware dependencies, tool chains and the setting of constants and/or calibration parameters in which the component should be applied. Properties are specified in the service level of each component and the context information is specified for each property. Semantic context information is also specified by referring to the modes (such as initialization mode, cyclic executive mode or shut-down mode). Bosch uses static analysis tool aiT [7] to analyze object code and to extract the worst-case execution time of a component, and SymTA/S [54] tool as a reasoning framework that aids analysis and prediction of timing properties. The BlueArX concepts are close to to the ProSave layer of the ProCom component model, however ProCom uses an attribute management framework to associate extra-functional properties to components and others entities of the component model (component services, message ports, communication channels and component instances).

COMDES-II (COMponent-based design of software for Distributed Embedded Systems) [66] is a two-layered component model similar to ProCom, developed at University of Southern Denmark. At the system (first) layer, a distributed system is modeled as a network of communicating actors, and at the second level the functionality of individual actors is further specified by interconnected function blocks. COMDES-II supports modeling architectural and behavioral aspects of systems with a goal to analyze and verify system behavior at high abstraction level and to enable automatic code generation. In difference to ProCom, the timing behavior in COMDES-II is separated from the functional behavior. The timing behavior is verified by schedulability analysis, whereas functional properties are formally verified. Ke et al. [65] show how a COMDES-II system can be equivalently transformed into Uppaal timed automata, and verified with preservation of system operational semantics.

IEC 61499 [60] is developed by the International Electrotechnical Commission (IEC) to support the development of automation and control systems. It has evolved from IEC 61131-3 [59] standard that is widely used in the development of software for PLCs. IEC 61499 components are called function blocks that have a set of in- and out ports, and a hidden internal implementation. Similar to ProSave, the data between the blocks is transferred using pipe-and-filter paradigm and the execu-

tion of the function blocks is event driven. In comparison to ProCom, there is no support for specifying or reasoning about extra-functional properties.

Koala [101] presents a component model that is designed and used by Philips for the development of software in consumer electronics (such as TVs, VCRs, and DVDs). Components are connected via provided and required interfaces that depict a small set of semantically related functions. The Koala component model is hierarchical, so, compound components may be defined. For Koala compositions, the extra-functional information is exposed at the component's interface. The prediction of extra-functional properties is carried out by measurements and simulations at the application level. In contrast, the ProCom semantics sets the ground for achieving predictability via formal verification (by translating FSMs into timed automata), prior to implementation. Moreover, compared to ProCom, Koala is geared towards less safety-critical applications.

PECOS [108] is a component model developed conjointly by ABB Corporate Research and academia for development of small reactive embedded systems in automation applications (such as industrial field devices). The PECOS component model supports hierarchical component composition. Similarly to ProSave level, components interact via data ports, and the communication between them is based on the pipe-and-filter paradigm. A PECOS component can be active, passive or an event. Active and event components have their own thread of execution, and passive components cannot control their execution and are used as part of the behavior of another component being executed synchronously. Besides data ports, PECOS components have also interfaces to express extra-functional properties and constraints. In PECOS, as in ProCom, a strong importance is given to extra-functional properties, and there is possibility to specify component's meta-data, such as worst-case execution times and memory usage, but the techniques differ. The behavior of the components can be modeled with Petri nets.

Pin [56] component model is developed at Carnegie-Mellon University. Its purpose is to be used as a basis for PECTs (Prediction-Enabled Component Technologies), which provide predictability principles for the run-time behavior of assemblies of software components, such as performance, safety and security. In order to attain predictability of a given property PECT offers a reasoning framework that includes a component technology powered by analytical interfaces and analysis theory. Ana-

lytical interfaces are used for specification of the properties, which are n-tuples consisting of a name, value and additional property-specific information (e.g., confidence interval of the property value). Analysis theories are used to predict properties of component compositions. At this time PECT supports three reasoning frameworks: λ_{ABA} - for predicting average latency in assemblies with periodic tasks, λ_{ss} - for predicting average latency in stochastic tasks managed by a sporadic server and ComFoRT - for formal verification of temporal safety and liveness. Contrary to ProCom, Pin is not distributed, does not support hierarchical component nesting and does not have support for high-level design.

Robocop [78] component model is a successor of the Koala component model, and is developed out of the collaboration between Philips and Eindhoven Technical University. Similar to ProCom, a component is considered as "a whole", i.e., a collection of models gathering all the information needed and/or specified at different points of time of the development process (e.g., documentation, source code, functional model, resource model, simulation model and execution model). Models may be used as well for depicting extra-functional properties of Robocop components. These extra-functional models can include timeliness, resource consumption, reliability, safety and security. The resource model is based on resource predictions, which can not provide 100% guarantees if compared to formal methods. Therefore, it is not suitable for safety-critical systems. The functionality offered by a component is logically modeled as a set of "services". Similar to Koala, Robocop is dealing only with static resource consumption, since it is assumed that consumption of resources stays constant per operation of a service.

Rubus [52] is a component model developed in collaboration between Arcticus Systems AB and Mälardalen University, and is intended for development of distributed, resource-constrained, embedded control systems, with a mix of hard-, soft- and non real-time system requirements. Rubus components are called software circuits and each of these circuits is defined by its behavior, internal state, and interface. An interface is a set of input- and output ports. ProCom has been influenced by Rubus time- and event-triggering features and the ability to perform real-time analysis. In Rubus it is possible to specify timing properties and there is is a tool for schedulability analysis. Similar to ProSave, Rubus has data- and trigger ports, which capture data- and control flow, respectively. However, Rubus does not provide support for distributed implementation nor high-level design.

ProCom's precursor, SaveCCM [9], is also a component model for real-time systems. ProCom has inherited some concepts from SaveCCM, in particular in the ProSave layer, such as the emphasis on reusability, a strong degree of analyzability of component behavior wrt to timing behavior and safety due to the strong restrictions in the proposed syntax and semantics, and the decoupling of data- and control-flows. A new feature of ProSave, compared to SaveCCM, is that ports can be grouped into services, they are part of the component and allow external entities to make use of the component functionality. Services are triggered independently and can run concurrently. In SaveCCM, component behavior modeling is done using timed automata extended with tasks, a formalism that explicitly models timing and real-time task scheduling. The timed automata models of SaveCCM can be cluttered with variables whose interpretation is not necessarily intuitive, which makes the formal models less amenable to changes. In addition, ProCom has a clearer concept of composite components, and addresses distribution and extra-functional properties more systematically.

8.2 Resource-Aware Modeling and Analysis for Embedded Systems

Although, one may think of numerous extra-functional properties crucial for embedded systems, in practice, they often reduce to timing, memory, performance or throughput, and dependability/reliability-related aspects. These aspects may be addressed differently depending on the context or the application domain (e.g., timing aspects have to be more precise for safety-critical systems than for home-appliances). Thus, depending on the context, extra-functional properties can be modeled or built-in at different levels of formality, such as: informal level, which describes extra-functional aspects in natural language; semi-formal, which uses notations, such as the UML [49] or even more formal, which describes extra-functional aspects by using much more formal notations, such as temporal logics or process algebras. Using to a great extent the work we have presented in paper [103], this section summarizes the related work on modeling and analyzing resources in embedded systems and compares them with the Remes behavioral model proposed in this thesis. The related approaches we have classified into three categories as follows.

The first category of related work covers approaches that address predicting code-level resource consumption of component assemblies. Eskenazi et al. [43] and Fioukov et al. [45] present compositional ways of estimating static memory consumption of Koala-based embedded systems [101], in which the instantiated components of a composition are known prior to run-time. In the mentioned approached, the memory consumption demands of each component are exposed through a special type of component's interface, called IResource. The implementation of IResource contains a formula for estimating the memory size of each type of memory. The approach supports budgeting i.e., it is possible to take into account the estimates of memory demands for the components that are not implemented yet. Jonge et al. [39] introduce a scenario-based prediction of run-time memory consumption in component-based applications, this time for the Robocop component model [78]. Resource consumption in this approach, is specified for all operations implemented by the services of an executable component. Similar to Koala, this method is also dealing with static resource consumption, since it is assumed that resource claims and releases are constant per operation, whereas they typically depend on parameters passed to operations. Both of the aforementioned approaches have been mainly dealing with low-level, code-driven estimation of static memory usage, which can only be used in cases when the components implementations are avaialble. However, more abstract descriptions of expected resource usage may be needed for not-yet implemented components, or for selecting components from repositories, and adapting them to fit the design. In such cases, the designer can first use Remes for early resource usage modeling and analysis, and then apply the approaches described earlier.

The second category is represented by the software modeling languages and profiles attempts (e.g., UML/SPT [48] and MARTE [85]) to tackle the modeling and analysis of embedded resources. Amar et al. [15] model resources in UML-based simulative environment. They extend the UML notation with new stereotypes for resources types. In one capsule diagram are gathered the software architecture and the resources that the software components require. As such, the capsule diagram is spilt in two parts: the software side and the resource side. The resource side is composed by a Main Dispatcher, which is in charge of receiving resource requests from the software side and a set of resource types. Internally every resource type capsule contains an Internal Dispatcher and a set of actual resource instances. The UML profile for Schedulability,

Performance and Time (UML/SPT) [48] is a framework for modeling concurrency, resources and timing concepts, which eventually produces models for schedulability and performance analysis. The core of the profile represents the General Resource Modeling framework, which describes resource types (hardware or software) and their management. The UML/SPT profile provides set of stereotypes and tag values that can be used for annotation of the model elements and for performing analysis. The recent profile, MARTE (Modeling and Analysis of Real-Time and Embedded systems) [85], which emerged from the UML/SPT profile is dedicated to complement UML with the required extensions for supporting modeling and analysis of extra-functional properties of embedded real-time systems, such as memory and power consumption. It provides a basic framework for platform-based modeling and a high level concepts for specifying resource usage. This all is done through the Generic Resource Modeling (GRM) sub-profile that is based on a clear design pattern considering platforms as a set of resources containing possible sub-resources in hierarchical manner and offering at least one service. GRM is refined in Software Resource Model and Hardware Resource Model dedicated to describe software and hardware computing platforms, respectively. MARTE has a complex structure and it is tightly connected to UML. Hagner et al. [50] present a UML profile, called Power Consumption Analysis View Profile, for annotating power/energy consumption relevant parameters to a UML development model and a simple algorithm to analyze power consumption of that model. For the power consumption analysis of a system the power consumption of each task is calculated to find out the power consumption of a CPU. Although graphical and intuitive, the previously listed UML-based approaches are not precise and rigorous, and lack formally founded semantics. They can not entirely guarantee the feasibility of the architecture, but rather give partial answer. In contrast, Remes provides both a graphical behavioral notation, as well as a rigorous underlying framework for formal analysis.

The third category is mainly represented by the higher-level formal approaches [76, 77], proposed by Lee et al. They propose a family of process-algebraic formalisms, that rely on an algebra of communicating shared resources (ACSR), developed to unify formal modeling and analysis of embedded systems resources. Like Remes, their formalisms can theoretically account for various resource types and a resource is considered as a generic, first-class modeling entity. A resource may be characterized by a set of attributes, such as timing parameters, proba-

bility of failure, priority, power consumption, etc., which capture the resource's behavior. The authors take into account sets of resource classes important for embedded real-time systems: serially reusable shared resources, used to model processor units, communication resources, used to model synchronous and asynchronous communication channels, and multi-capacity resources that naturally correspond to memory modules. An important restriction of the ACSR framework is the assumption that every action lasts exactly one time unit, and that only one process may use a given resource during a time step. Although the ACSR framework is theoretically rich, however it is not intuitive, and the tool support is not equally mature. Kim et al. [69] have recently presented an extension of statecharts, called Timed and Resource-oriented Statecharts (TRoS), that provides the notation of timed action, adopted from ACSR. In TRoS each action is labeled with its consumption of resources. In contrast to ACSR, in TRoS it is assumed that an execution of a timed action takes one or more timed units in respect of a global clock. The system is modeled as a set of TRoSs that run in parallel and share limited resources. Similar like in ACSR, a situation where two or more processes with the same priority access to same resource is not permitted in TRoS. Ouimet et al. [88] use timed abstract state machines as a unified formalism to specify functional and extra-functional properties of embedded systems. The resources are described as simple annotations, in the form of real-valued variable assignments. Consequently, the framework can not support trade-off analysis of possibly conflicting resource requirements, which is supported by Remes. Schlatte et al. [90] describe an enhancement for the object oriented modeling language Creol [63] for supporting resource constraints modeling, specifically memory consumption, call stack depth and restrictions on parallelism. For modeling resource constraints, they assign each method an amount of required resources (through attribute RNeed), and each class an amount of provided resources (such as memory/processing capacity via attribute RLimit). Creol deals with modeling systems on a lower level of abstraction than Remes.

Chapter 9

Conclusion

Embedded systems are challenging to design since they must meet special type of constraints beyond those that apply for general-purpose computers, such as low cost, timeliness, resource usage efficiency, short time-to-market, etc. The demanding requirements of modern embedded systems coupled with the increasing complexity of the underlying software, demand techniques for managing complexity and for ensuring critical system properties. The thesis concerned the modeling and analysis of embedded systems, and argued that resource requirements need to be considered from the initial design stages of embedded systems. This chapter provides some concluding remarks on the presented work, and presents a number of directions in which this work could be extended in the future.

9.1 Summary and Contributions

The thesis proposes a resource-aware framework for designing predictable component-based embedded systems. The proposed framework consists of (i) the formally specified two-layered component model - ProCom (ii) the resource-aware behavioral modeling language - Remes (REsource Model for Embedded Systems), associated analysis techniques for various resource-wise properties, and a set of tools implementing the former.

The ProCom component model, introduced in Chapter 3, is specifically developed to target control-intensive embedded systems. The

model takes into account the most important characteristics of these systems and consistently uses the concept of reusable components throughout the development process, from early design to deployment. Reusing a component means reusing not only its concrete realization, but the whole collection of artifacts needed or produced during the development of this system element, such as source code, early design models, test results, architectural models, behavioral models and analysis results. At an early stage of the system design process, a new component could consist of just the interface specification, maybe with a rough behavior model. Additional information is then added gradually, e.g., internal structure or source code, detailed models, analysis results and documentation, and in the final development stage the components are synthesized together into an executable system.

Another characteristic feature of ProCom is that it is structured in two layers (ProSys and ProSave), to address the different concerns that arise when modeling on one hand a large distributed system, and on the other hand the detailed control functionality of each individual subsystem. These layers differ from each other in terms of communication style, execution model, synchronization etc., but also in kind of analysis which are suitable. The ProCom language constructs include services, data and trigger ports, passive or active components, connections and connectors, hierarchies of components, etc. All of these constructs have a precise execution semantics that we formally define.

Clearly, a formalization of the ProCom language needs to deal with all concepts of the modeling language. It has been our goal to make the formalization as intuitive as possible, so that it can serve as a basis both for engineers using ProCom, as well as researchers developing analysis techniques, model-transformations tools, etc., within the ProCom framework. In order to meet these sometimes contradicting goals, we have used a small but powerful finite-state machine (FSM) language, in which the semantics of each ProCom element we have defined as a translation relation from ProCom to the FSM language. The FSM language is essentially standard FSM, enriched with finite domain integer variables, guards and assignments on transitions, notions of urgency and priority, as well as time delays in locations. Hence, it can obviously be analyzed in a number of theoretical and tool frameworks including e.g., the Uppaal tool.

The possibly complex extra-functional behavior of ProCom components we model in Remes, a dense-time, state-based hierarchical lan-

guage. REMES, presented in Chapter 4, is suited for modeling timing and resource-wise behaviors of embedded system components described by modes. A mode can be either atomic or composite depending on whether it contains number of submodes or not. The crux of REMES is introducing resources as first-class modeling entities that are characterized by their discrete (e.g., memory, access to external devices) or continuous (like energy) nature. Each mode has a well-defined data interface consisting of typed global variables that is used for data transfer between modes, and also a well-defined control interface in terms of entry and exit points through which discrete control enters and exists the mode. Each mode may declare its own set of local variables that is hidden outside the mode, but accessible to its submodes. To analyze various scenarios of system's resource usage, in REMES models, we encode the resource-wise analysis problem as a weighted sum of consumed amounts of resources and their given weights. Assuming the encoding, we define three important resource analysis problems: feasibility analysis, trade-off analysis, and optimal/worst-case resource analysis. To be able to formally analyze REMES compositions we have provided a set of transformation rules that semantically translate REMES modes into priced timed automata.

By connecting REMES behavioral models to individual ProCom components, via a general attribute framework [92], in Chapter 5, we have addressed the important problem of model reuse. The packaging of ProCom and REMES is achieved by defining a new attribute type attached to a ProCom component in the attribute registry, which has a complex attribute value consisting of (i) a relative path to the REMES model file in the component structure and (ii) a relative path to the mapping file specifying the relation between the ports of a ProCom component and the variables of its associate REMES model. The connection between ProCom and REMES is done differently for ProSave and ProSys. The concepts of connecting and packaging a ProCom component with its REMES behavioral model we present on two examples: a temperature control system, when the architecture of the system is modeled in ProSave, and a turntable drilling system, when the architecture of the system is modeled in ProSys.

Based on the REMES language, in Chapter 6 we have presented the REMES tool-chain supporting construction and analysis of embedded systems behaviors. The core elements of the tool chain are: (i) the REMES editor for modeling behaviors of embedded components, (ii) the REMES

simulator to test timing and resource behavior prior to formal analysis, and (iii) an automated transformation from REMES into priced timed automata, following the transformation rules presented in Chapter 4, needed for formal analysis. The transformation rules applied to REMES diagrams result in ULITE models representing the same behavior. A graphical editor for ULITE models is provided, as a tool to visually inspect transformation results. Model files for both UPPAAL (timed automata) and UPPAAL CORA (priced timed automata) can be exported for formal analysis. When using the REMES tool-chain within PRIDE, an environment for ProCom component-based architecture development [26], the transformation uses component triggering information from ProSave components, to insert appropriate synchronization channels into the resulting priced timed automata. REMES models were integrated in PRIDE via the Attribute Framework [92]. We provide QVT model-to-model transformation that creates blank REMES templates for ProCom components. Each template contains pre-created variables matching the ProCom component ports, as we have described in Chapter 5.

Finally, in Chapter 7, we have demonstrated the applicability, and we have exercised the modeling and formal analysis concepts of our resource-aware framework on an industrial research prototype of an Ericsson Nikola Tesla telecommunication system. As such, we have modeled the system in a component-based fashion using the ProCom component model, whereas the functional, timing, and resource-wise behavior of the key components of the system we have modeled in REMES. We have also shown how the combined model (ProCom and REMES) can be semantically translated into a network of (priced) timed automata to enable model-checking in the tools UPPAAL and UPPAAL CORA which verification engine can be started directly from the REMES tool-chain. The validation of our models is ensured by the cost model that we use – derived from the measurements of timing, CPU, and memory usage – measured by Ericsson researchers on the prototype's source code.

9.2 Limitations and Future work

The current version of the ProCom component model focusses on the design of a class of embedded systems that primarily perform real-time controlling tasks. In the future ProCom might be extended to other types of embedded systems. In addition, the list of ProSave connec-

tors is presumably incomplete and may grow over time as additional data-/control-flow constructs prove to be needed. The semantics of ProCom, in future, will be extended with the formalization of actuators and sensors. Although the ProCom finite-state machine formalization sets the ground for formal analysis, the semantic descriptions focus only on formalizing the correct execution semantics of ProCom architectural elements, without consideration for efficiency in formal analysis of the resulted models. As future work we may develop support for model-based analysis techniques, such as model-checking, based on the formalization of the ProCom component model. In particular, it could be interesting and helpful to integrate our modeling language REMES with the formal execution semantics of ProCom.

Our REMES behavioral language can be used for modeling the internal behavior of interacting embedded components, not necessarily modeled in the ProCom component model. As such, REMES complements architectural description languages (ADLs) [83], which describe the software system's conceptual architecture as a collection of components, connectors and architectural configurations, by adding component behavior. If one attaches semantics to the connection points of the architectural elements of a system, REMES can then be used for modeling the behavior of a generic embedded system. Moreover, it was our intention to make REMES as simple as possible, so that it can be utilized by both formalists and engineers with different backgrounds, as an intermediate layer between abstract architectural modeling and very detailed behavioral modeling (e.g., by priced timed automata).

In the current version of the REMES language, top-level modes follow a "run-to-completion" semantics, where only one submode can be active at a time. Since services of a ProSave component can be triggered/activated independently and may run concurrently, we associate a REMES mode to each service. In future, we could extend the REMES language with new types of constructs (such as modes or connectors) that will allow several submodes of a REMES top-level mode to be simultaneous active. In turn, this will allow that only one REMES mode is associated to a ProSave component made of more than one service. It is worth pointing out that REMES has already been extended with new kind of AND and OR modes [102], which still have a "run-to-completion" semantics. For AND modes all services are entered at the same time, whereas in OR modes one or all constituent services are entered when the mode is activated. This type of modes are needed for describing the behavior

of service-oriented systems, where it is often the case that services need to synchronize their behaviors. Depending on the required synchronization type and starting time of the constituent services execution, both AND and OR modes can be employed when either "and" synchronization (both services should finish execution at the same time), or "max" synchronization (the composite mode finishes when the slowest service finishes) is required. In addition, the relation between the REMES model of a composite component and those associated with its subcomponents should be studied.

As future work, we plan to apply the results of Bouyer et al. [30], in order to tackle the feasibility analysis problem for systems in which the global cost function is non-monotonically increasing. In such situations, the usual branch-and-bound symbolic reachability algorithms, for priced timed automata, cannot be applied as such anymore, since minimal/maximal reachability analysis requires a monotonically increasing cost function. In addition, all of the resource-wise verification algorithms presented in this thesis need to be implemented in UPPAAL CORA.

The cost analysis model proposed in Section 4.2.1 is platform-aware. Hence, as future work, our simplistic platform profile (described in Appendix C) could benefit from including abstractions of platform specific tools, such as the associated compiler, linker etc. We consider that the cost model can be derived from the results provided by static analysis tools, which could be applied on already implemented components. A possible solution is presented by Bonenfant et al. [25]. In order to obtain provably correct static analysis results, the authors propose a formal source-level cost model, enriched with rules for deriving the execution cost of a subset of expressions belonging to the system-oriented language Hume. We also underline the fact that the selection of the weights in our resource model depends mostly on the designer's experience and decisions. However, by analyzing the results of model checking the chosen cost models, one could adjust the weights accordingly.

The concrete transformation rules for translating REMES integrated with ProCom into priced timed automata are subject of future improvements. The usage scenarios in the practise might outline new requirements, where the transformed models might be enriched in order to save the verification expert's manual effort. The transformation rules for transforming REMES into priced timed automata, presented in Section 4.3, cover two-level REMES mode hierarchy only. As future work, we plan to extend the transformation rules to cover hierarchy of arbitrary

depth. Another limitation of the transformation rules is that they do not handle different types of ProSave connectors.

Concerning the REMES tool-chain, note that all the theoretical contributions have not yet been implemented. For e.g., the rules for transforming non-lazy modes into priced timed automata are still not implemented. Also, the current version of the REMES editor does not support nesting of composite modes.

Throughout this thesis we have discussed and proposed solutions for modeling and formal analysis of component-based embedded systems. Unfortunately, the relatively small case-studies that we have performed can not answer questions about size and complexity of our methods when applied to large real-world systems. Scalability is clearly not exercised within this context. We can just hope that the presented examples have added some merit to the theoretical results. As future work we could as well exercise the scalability of REMES and associated analysis techniques. We also plan to look into compositional reasoning in the REMES language i.e., separately analyzing each component of the system and allowing global properties to be deduced for the entire system. This certainly will leave us obligation of proving that the component specifications in turn apply the specification of the whole system. Instead of generating a priced timed automata model of the entire system, compositional reasoning could be used to prove global system properties out of individual subsystems, or subsystem clusters properties. Another approach will envision developing specialized model checking optimizations, which exploit the topology of the architectural- and behavioral model, similar to the work on UPPAAL PORT [51]. In order to perform compositional reasoning of REMES, we could first give a trace semantics of a REMES mode. This would allow us to compute the set of traces of a composition of REMES modes from the traces of the constituent modes.

Another opportunity for future work is to investigate further the ENT demonstrator by modeling and verifying other telecommunication protocols for serving requests. By checking possible performance in such other cases (like the first-in-first-out protocol), we could feed Ericsson researchers with important insights on the systems behavior, which might save unnecessary implementation time. We can also consider the case of heterogeneous servers, that is, processing the same request takes different time on each server, respectively.

Appendix A

REMES Meta-model

The meta-model is a feature of the REMES behavioral language. It models its concepts as classes and shows the relations among them. The complete REMES meta-model is depicted in Figure A.1. For readability, in Figures A.2, A.3 and A.5 we focus on particular parts of the REMES meta-model.

The root element of the REMES meta-model is the class RemesDiagram (see Figure A.2), which contains zero or more (usually one) elements modes from type Mode. The class Mode is an abstract class that has to be replaced by either CompositeMode or SubMode. The internal of a composite mode can be modeled by submodes, conditional connectors, init-, write-, composite entry- and composite exit point. SubMode has attributes invariant (specifies for how long a mode can be executed) and isUrgent (when isUrgent is set to true the mode is exited right-away). Modes and conditional connectors represent a control path of execution (class ControlPath).

Four types of points (class Point) exist in the meta-model: InitPoint, EntryPoint, WritePoint and ExitPoint (see Figure A.3). An edge (class Edge) connects an exit- to an entry point, and has attributes actionGuard and actionBody. When the action guard of an edge is evaluated to true, the action body (i.e., a statement or a sequence of statements) is executed. ControlPath contains an entry- and an exit point, which are used to enter and exit modes or conditional connectors, respectively. In addition, a composite mode contains an init point (class InitPoint) through which the mode is entered for the first time of its execution, and a write

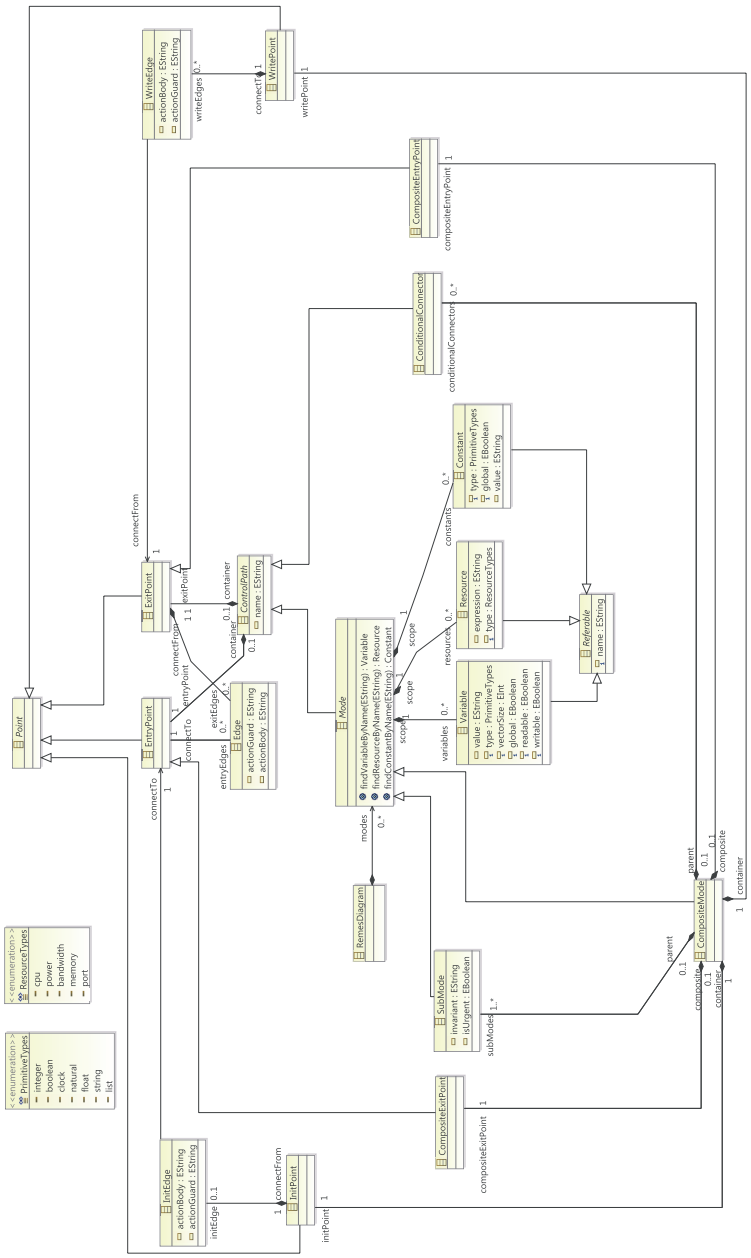


Figure A.1: The REMES meta-model for describing a REMES diagram.

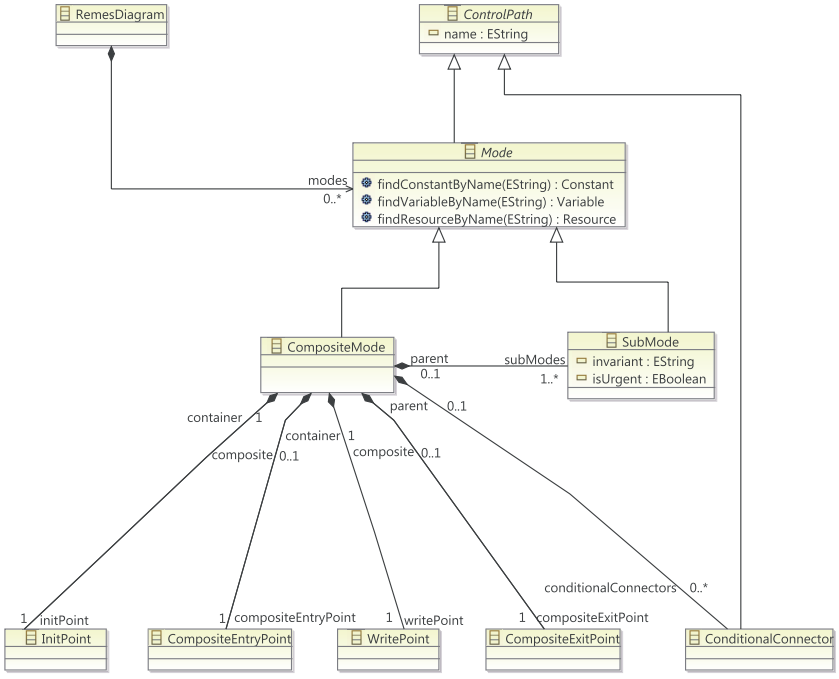


Figure A.2: Excerpt of the REMES meta-model for describing a REMES diagram.

point (class WritePoint), which is a local exit of the mode. An init edge (class InitEdge) connects an init- and an entry point, and a write edge (class WriteEdge) connects an exit- to a write point.

The composite entry- and exit point of a composite mode deserve a closer look. A composite mode is entered via its regular entry point. In model semantics it is not possible to connect this entry point to another entry point, for example of a submode. An edge only connects an exit- to an entry point. This imposed the need to introduce "intermediate" points in the REMES meta-model — composite entry point (class CompositeEntryPoint) and composite exit point (class CompositeExitPoint). Note, in Figure A.3, that the class CompositeEntryPoint extends ExitPoint to be used as a source of an edge. Similar applies for CompositeExitPoint

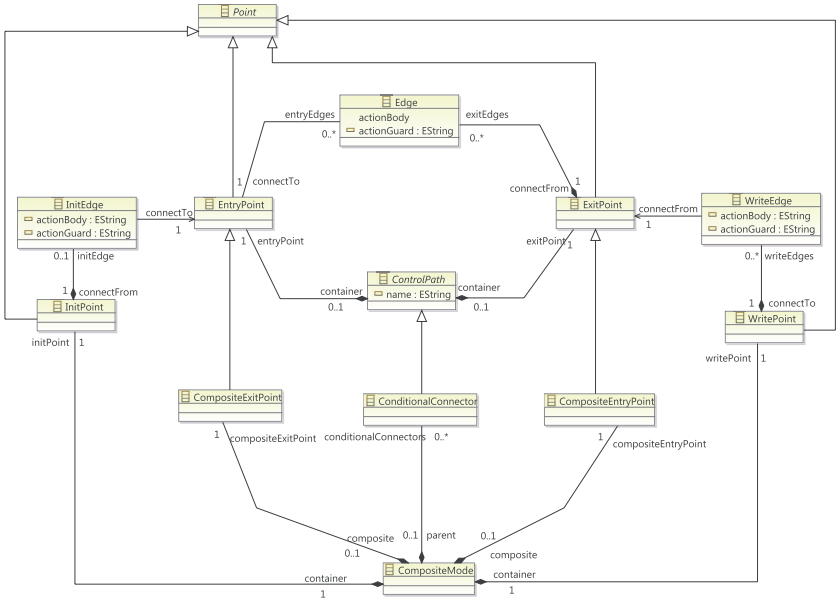


Figure A.3: Excerpt of the REMES meta-model that shows control point entities.

— it extends EntryPoint, so that it can be used as a target of an edge. Figure A.4 helps to get a better understanding how these parts of the meta-model map to a model instance.

The referable interface (class Referable) of a mode contains a single attribute name, and is extended by Variable, Resource and Constant (see Figure A.5). Variables are of primitive type. Attribute vectorSize is grater than zero if the variable is an array. The interface flag denotes a global variable mapped to an interface port (data, trigger or message) coming from the ProCom architectural model, as we have described in Chapter 5. The readable and writable flags determine whether a given REMES mode variable represents an input or output data or message port. Resources have type and expression attribute. If a certain resource is continuous, then the expression property defines the rate at which the resource is being consumed. The rate is defined by the first derivative of the resource.

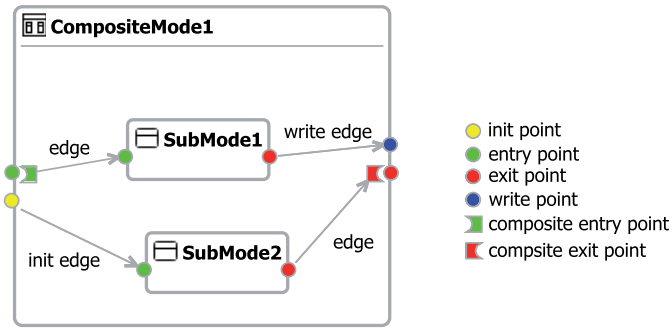


Figure A.4: Control points legend in REMES.

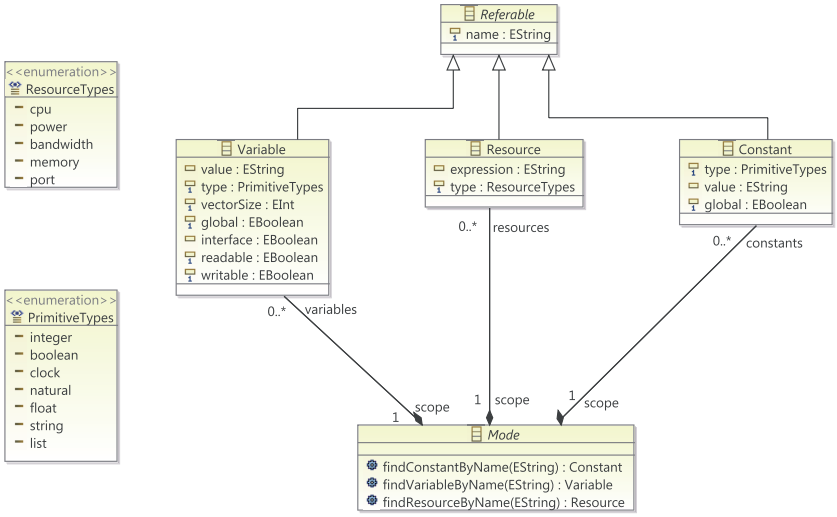


Figure A.5: Excerpt of the REMES meta-model that shows referable entities.

Appendix B

ULite Meta-model

The diagram in Figure B.1 depicts the main entities in the ULite meta-model and their relationships. The ULite meta-model is a subset of the Uppaal meta-model targeted for the transformation from Remes models into (priced) timed automata. When using the Remes tool-chain within the ProCom Integrated Development Environment (Pride) [26] the transformation uses component triggering information from ProCom architectural models, to insert appropriate synchronization channels into the resulting priced timed automata.

The root model entity is the diagram (class UppaalDiagram). It has declaration attribute, which carries the global variables declaration. The latter contains integer constants that determine the relative importance of different resources. Every UppaalDiagram presents a composition of timed automata. A timed automaton is represented with the element Template. It contains an attribute declaration for declaring the local variables. Each automaton is made of locations (class Location) with a set of transitions (class Transition) connecting the locations. Location has the following attributes: name, id, invariant, cost, initial, urgent and committed. Transition contains the following attributes: source location, target location, guard, assignment, sync (contains synchronization channel information) and cost for taking the transition.

Note that the ULite meta-model can be used for description of both timed automata and priced timed automata. In case of priced timed automata the cost for staying in a certain location is written in the extended invariant of a location (attribute invariant from class Location).

Similarly, the cost for taking a transition is written as part of the transition assignment (attribute assignment from class Transition).

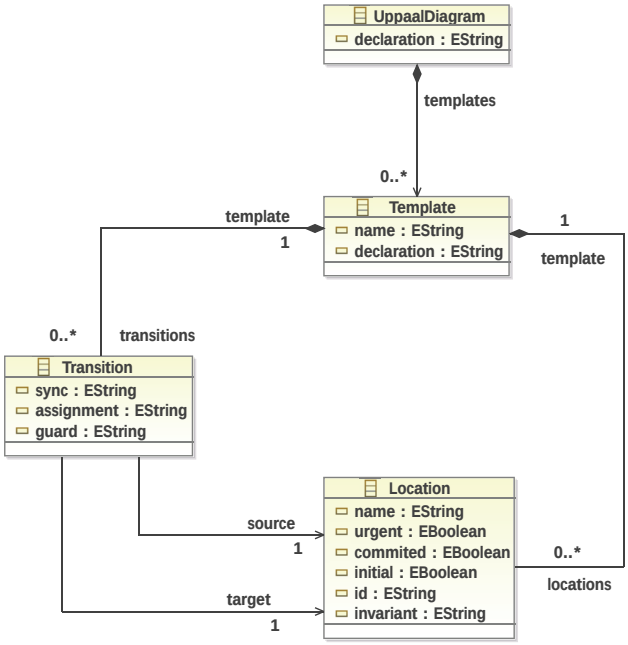


Figure B.1: The ULite meta-model.

Appendix C

Platform profile

Platform profiles can be used to distinguish between different software/hardware platforms, and different limitations they impose. In the REMES tool-chain a platform profile consists of resource- and constraint declarations, as in the following example:

```
profile Board1 {
  resources {
    proc: cpu @ 5,
    mem: memory @ 1,
    batt: energy @ 10
  }
  constraints {
    max(proc') <= 200,
    max(mem) <= 10240,

    max(batt) <= 150000,
    min(batt') >= 5, max(batt') <= 100
  }
}
```

In the above example, we declare a profile of a platform with three resources, proc, mem, and batt. Relative weights of resources are marked after the symbol @, with cpu having five times the weight of mem in our example platform. Constraints over these resources can be specified on resource consumption rates or on resource consumptions directly: $\mathsf{max}(\mathsf{proc}') <= 200$ constraints maximum rate of the resource proc to remain under 200, while $\mathsf{max}(\mathsf{mem}) <= 10240$ restricts the total consumption of the resource mem to 10240. For the resource batt, we restrict

the consumption rate between 5 and 100, and the total consumption to 150000.

This is a simplistic model of platform resources suitable for high-level analysis. The presented platform profile is not directly related to REMES, but to the implementation of the tool-chain. Relative weights of the resources defined in the profile are used to construct a weighted sum of resource consumptions during analysis, described in Section 4.2. The constraints described in the platform profile are similar to resource usage definitions of the Generic Resource Modeling package introduced in the MARTE profile [85].

Bibliography

[1] TIMMO (TIMing MOdel) consortium. http://www.timmo-2-use.org/timmo/index.htm, (Last Accessed: 2012-01-30).

[2] Uppaal. www.uppaal.com, (Last Accessed: 2012-03-27).

[3] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, page 1, 1990.

[4] *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[5] Parosh Aziz Abdulla, Pavel Krcal, and Wang Yi. Sampled Universality of Timed Automata. *Proceedings of the 10th International Conference Foundations of Software Science and Computational Structures*, LNCS 4423:2–16, 2007.

[6] Acceleo. http://www.acceleo.org/ (Last Accessed: 2012-04-14).

[7] aiT - Worst-Case Execution Time Analyzer. http://www.absint.com/ait/, (Last Accessed: 2012-01-30).

[8] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The SAVE approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, May 2007.

[9] Mikael Åkerholm, Jan Carlson, John Håkansson, Hans Hansson, Mikael Nolin, Thomas Nolte, and Paul Pettersson. The SaveCCM

Language Reference Manual. Technical Report MDH-MRTC-207/2007-1-SE, Mälardalen University, January 2007.

[10] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138:3–34, 1995.

[11] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[12] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-Checking in Dense Real-time. *Information and Computation*, 104:2–34, 1993.

[13] Rajeev Alur, Thao Dang, Joel Esposito, Yerang Hur, Franjo Ivančić, Vijay Kumar, Insup Lee, Pradyumna Mishra, George J. Pappas, and Oleg Sokolsky. Hierarchical Modeling and Analysis of Embedded Systems. *Proceedings of the IEEE*, 8(3):231–274, 1987.

[14] Rajeev Alur and David L. Dill. Automata For Modeling Real-Time Systems. *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, pages 322–335, 1990.

[15] Hany H. Ammar, Vittorio Cortellessa, and Alaa Ibrahim. Modeling Resources in a UML-Based Simulative Environment. *Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, pages 405–410, 2001.

[16] ANTLR (ANother Tool for Language Recognition). `http://www.antlr.org/` (Last Accessed: 2012-04-14).

[17] ATL (Atlas Transformation Language) - User Guide. `http://wiki.eclipse.org/ATL/User_Guide` (Last Accessed: 2012-04-14).

[18] AUTOSAR Development Partnership. AUTOSAR – Technical Overview V2.2.1, February 2008. `http://www.autosar.org/download/AUTOSAR_TechnicalOverview.pdf` (Last Accessed: 2012-04-11).

[19] AUTOSAR Development Partnership. AUTOSAR – Specification of Timing Extensions V1.2.0, September 2011. `http://www.`

`autosar.org/download/R4.0/AUTOSAR_TPS_TimingExtensions.pdf` (Last Accessed: 2012-04-11).

[20] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[21] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim Guldstrand Larsen, Paul Pettersson, Judi Romijn, and Frits Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control*, (2034):147–161, 2001.

[22] Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Automated Analysis of an Audio Control Protocol Using Uppaal. *Journal of Logic and Algebraic Programming*, 52–53:163–181, 2002.

[23] Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an Audio Protocol with Bus Collision Using Uppaal. *Proceedings of the 8th International Conference on Computer Aided Verification*, pages 244–256, 1996.

[24] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal - a Tool Suite for Automatic Verification of Real-Time Systems. *Proceedings of Workshop on Verification and Control of Hybrid Systems III*, pages 232–243, October 1995.

[25] Armelle Bonenfant, Zezhi Chen, Kevin Hammond, Greg Michaelson, Andy Wallace, and Iain Wallace. Towards Resource-Certified Software: A Formal Cost Model for Time and its Application to an Image-Processing Example. *ACM Symposium on Applied Computing*, March 2007.

[26] Etienne Borde, Jan Carlson, Juraj Feljan, Luka Lednicki, Thomas Leveque, Josip Maras, Ana Petricic, and Séverine Sentilles. PRIDE an Environment for Component-based Development of Distributed Real-time Embedded Systems. *Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture*, June 2011.

[27] Elena M. Bortnik, Nikola Trčka, Anton Wijs, Bas Luttik, J. M. van de Mortel-Fronczak, Jos C. M. Baeten, Wan Fokkink, and J. E. Rooda. Analyzing a χ model of a turntable system using Spin, CADP and Uppaal. *Journal of Logic and Algebraic Programming*, 65(2):51–104, 2005.

[28] Victor Bos and Jeroen J. T. Kleijn. Automatic verification of a manufacturing system. *Robotics and Computer-Integrated Manufacturing*, 17(3):185–198, 2001.

[29] Dragan Bošnački and Dennis Dams. Discrete-Time Promela and Spin. *Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 307–310, 1998.

[30] Patricia Bouyer, Thomas Brihaye, Véronique Bruyère, and Jean-François Raskin. On the optimal reachability problem of weighted timed automata. *Formal Methods in System Design*, 31(2):135–175, 2007.

[31] Patricia Bouyer, Kim G. Larsen, and Nicolas Markey. Model-Checking One-Clock Priced Timed Automata. *Logical Methods in Computer Science*, 4(2:9):1–28, 2008.

[32] Thomas Brihaye, Vèronique Bruyère, and Jean-François Raskin. Model-Checking for Weighted Timed Automata. *Proceedings of the joint conference on Formal Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault Tolerant System*, (3253):277–292, September 2004.

[33] Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, and Aneta Vulgarakis. ProCom – the Progress Component Model Reference Manual, version 1.0. Technical Report MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[34] Michel Chaudron and Ivica Crnkovic. Component-based Software Engineering. In *Software Engineering: Principles and Practice*, pages 605–628. Wiley, 3 edition, 2008.

[35] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[36] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-Based Software Systems.* Artech House publisher, 2002.

[37] Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel Chaudron. A Classification Framework for Software Component Models. *IEEE Transaction of Software Engineering*, 37(5):593–615, October 2011.

[38] Alexandre David, John Håkansson, Kim Guldstrand Larsen, and Paul Pettersson. Model Checking Timed Automata with Priorities using DBM Subtraction. *Proceedings of the 4th International Conference on Formal Modelling and Analysis of Timed Systems*, pages 128–142, September 2006.

[39] Merijn de Jonge, Johan Muskens, and Michel Chaudron. Scenario-Based Prediction of Run-Time Resource Consumption in Component-Based Software Systems. *Proceedings of the 6th ICSE Workshop on Component-based Software Engineering*, pages 19–24, 2003.

[40] Radomil Dvorak. Model Transformation with Operational QVT, 2008. `http://www.eclipse.org/eclipse/debug/` (Last Accessed: 2012-04-14).

[41] Stephen Edwards, Luciano Lavagno, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Design of Embedded Systems: Formal Models, Validation, and Synthesis. *Proceedings of the IEEE*, 85(3):366–390, march 1997.

[42] Ulrik Eklund and Carl Magnus Olsson. A Case Study of the Architecture Business Cycle for an In-Vehicle Software Architecture. *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pages 91–100, 2009.

[43] Evgeni M. Eskenazi, Alexandre V. Fioukov, Dieter K. Hammer, and Michel R. V. Chaudron. Estimation of Static Memory Consumption for Systems Built from Source Code Components. *Proceedings of the 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems*, April 2002.

[44] Peter H. Feiler, Bruce Lewis, and Steve Vestal. The SAE architecture analysis & design language (AADL) standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. *Proceeding of the IFIP World Computer Congress – Workshop on Architecture Description Languages*, 2004.

[45] Alexandre V. Fioukov, Evgeni M. Eskenazi, Dieter K. Hammer, and Michel R. V. Chaudron. Evaluation of Static Properties for Component-Based Architectures. *Proceedings of 28th EUROMICRO conference*, pages 33–39, 2002.

[46] David Garlan. Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events. *Third International School on Formal Methods for the Design of Computer, Communication and Software Systems*, 2804:1–24, 2003.

[47] Dawn G. Gregg, Uday R. Kulkarni, and Ajay S. Vinze. Understanding the Philosophical Underpinnings of Software Engineering Research in Information Systems. *Information Systems Frontiers*, 3(2):169–183, 2001.

[48] Object Management Group. UML Profile for Schedulability, Perfomance and Time Specification. Version 1.1, formal/05-01-02. 2005.

[49] The Object Management Group. UML Version 2.1.2. `http://www.omg.org/spec/UML/2.1.2/`, (Last Accessed: 2012-01-23).

[50] Matthias Hagner, Adina Aniculaesei, and Ursula Goltz. UML-Based Analysis of Power Consumption for Real-Time Embedded Systems. *Proceedings of the 8th IEEE International Conference on Embedded Software and Systems*, November 2011.

[51] John Håkansson, Jan Carlson, Aurelien Monot, Paul Pettersson, and Davor Slutej. Component-Based Design and Analysis of Embedded Systems with Uppaal Port. *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, pages 252–257, 2008.

[52] Kaj Hänninen, Jukka Mäki-Turja, Mikael Nolin, Mats Lindberg, John Lundbäck, and Kurt-Lennart Lundbäck. The Rubus Component Model for Resource Constrained Real-Time Systems. *Proceedings of the 3rd IEEE International Symposium on Industrial Embedded Systems*, June 2008.

[53] David Harel. Statecharts: A Visual Formalism For Complex Systems. *Science of Computer Programming*, 91(1), 2003.

[54] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System Level Performance Analysis – the SymTA/S Approach. *IEE Proceedings of Computers and Digital Techniques*, 152(2):148–166, March 2005.

[55] Thomas A. Henzinger and Joseph Sifakis. The Discipline of Embedded Systems Design. *Computer*, 40(10):32–40, 2007.

[56] Scott Hissam, James Ivers, Daniel Plakosh, and Kurt C. Wallnau. Pin Component Technology (V1.0) and Its C Interface. Technical Note: CMU/SEI-2005-TN-001, April 2005.

[57] Hilary J. Holz, Anne Applin, Bruria Haberman, Donald Joyce, Helen Purchase, and Catherine Reed. Research Methods in Computing: What are they, and how should we teach them? *Proceedings of the SIGCSE/SIGCUE Joint Conference on Integrating Technology into Computer Science Education*, 38:96–114, 2006.

[58] PROGRESS homepage. `http://www.mrtc.mdh.se/progress/`, (Last Accessed: 2012-01-23).

[59] IEC. Application and Implementation of IEC 61131-3. IEC, 1995.

[60] IEC. IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design. IEC, 2005.

[61] Dinko Ivanov. Integrating formal analysis techniques into the Progress-IDE. *Master thesis*, 2011.

[62] Dinko Ivanov, Marin Orlić, Cristina Seceleanu, and Aneta Vulgarakis. REMES Tool-chain - A Set of Integrated Tools for Behavioral Modeling and Analysis of Embedded Systems. *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, September 2010.

[63] Einar Broch Johnsen and Olaf Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and System Modeling*, 6(1):39–58, 2007.

[64] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. ATL: a QVT-like Transformation Language. *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720, June.

[65] Xu Ke, Paul Pettersson, Krzysztof Sierszecki, and Christo Angelov. Verification of COMDES-II Systems Using UPPAAL with Model Transformation. *Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 153–160, 2008.

[66] Xu Ke, Krzysztof Sierszecki, and Christo Angelov. COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems. *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 199–208, 2007.

[67] Ji Eun Kim, Rahul Kapoor, Martin Herrmann, Jochen Haerdtlein, Franz Grzeschniok, and Peter Lutz. Software Behavior Description of Real-Time Embedded Systems in Component Based Software Development. *Proceedings of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 307–311, 2008.

[68] Ji Eun Kim, Oliver Rogalla, Simon Kramer, and Arne Haman. Extracting, Specifying and Predicting Software System Properties in Component Based Real-Time Embedded Software Development. *Proceedings of the 31st International Conference on Software Engineering*, pages 28–38, 2009.

[69] Jin Hyun Kim, Inhye Kang, Jin-Young Choi, and Insup Lee. Timed and Resource-oriented Statecharts for Embedded Software. *IEEE Transactions on Industrial Informatics*, 6(4):568–578, November 2010.

[70] Kay Klobedanz, Christoph Kuznik, Andreas Thuy, and Wolfgang Mueller. Timing modeling and analysis for AUTOSAR-based software development: a case study. *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 642–645, 2010.

[71] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[72] Kim G. Larsen and Jacob I. Rasmussen. Optimal Conditional Reachability for Multi-priced Timed Automata. *Proceedings of the 8th International Conference on Foundations of Software Science and Computational Structures*, (3441):234–249, 2005.

[73] Kim Guldstrand Larsen and Jacob Illum Rasmussen. Optimal Reachability for Multi-Priced Timed Automata. *Journal Theoretical Computer Science*, 390(2-3):197–213, 2008.

[74] Luka Lednicki, Juraj Feljan, Jan Carlson, and Mario Zagar. Adding Support for Hardware Devices to Component Models for Embedded Systems. *Proceedings of the 6th International Conference on Software Engineering Advances*, pages 149–154, October 2011.

[75] Insup Lee, Jin-Young Choi, Hee-Hwan Kwak, Anna Philippou, and Oleg Sokolsky. A Family of Resource-Bound Real-Time Process Algebras. *Proceedings of the 21st International Conference on Formal Techniques for Networked and Distributed Systems*, pages 443–458, 2001.

[76] Insup Lee, Anna Philippou, and Oleg Sokolsky. A General Resource Framework for Real-Time Systems. *Proceedings of the 9th International Workshop on Radical Innovations of Software and Systems Engineering in the Future*, pages 234–248, 2002.

[77] Insup Lee, Anna Philippou, and Oleg Sokolsky. Resources in Process Algebra. *Journal of Logic and Algebraic Programming*, 72(1):98–122, 2007.

[78] Hugh Maaskant. *A Robust Component Model for Consumer Electronic Products*, volume 3 of *Philips Research*, pages 167–192. Springer, 2005.

[79] Esperanza Marcos. Software Engineering Research versus Software Development. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[80] Tiziana Margaria, Barry D. Floyd, and Bernhard Steffen. IT Simply Works: Simplicity and Embedded Systems Design. *Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference, Workshop on Component-Based Design of Resource-Constrained Systems*, pages 194–199, July 2011.

[81] Dinko Matijašević, Igor Gizdić, and Darko Huljenić. Mechanisms for Diameter service performance enhancement. *Proceedings of the 17th International Conference on Software, Telecommunications and Computer Networks*, 2009.

[82] Doug. Mcilroy. Mass-Produced Software Components. *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[83] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Language. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[84] Robin Milner. *Communication and Concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[85] Object Management Group. A UML Profile for MARTE. `http://www.omgmarte.org/`, (Last Accessed: 2012-03-09).

[86] Marin Orlić. Predvidjanje uporabe resursa u sustavima temeljenim na programskim komponentima (in Croatian). *Doctoral dissertation*, 2010.

[87] Marin Orlić, Aneta Vulgarakis, and Mario Zagar. Towards Simulative Environment for Early Development of Component-Based Embedded Systems. *Proceedings of the 15th International Workshop on Component-Oriented Programming*, June 2010.

[88] Martin Ouimet, Kristina Lundqvist, and Mikael Nolin. The Timed Abstract State Machine Language: An Executable Specification Language for Reactive Real-Time Systems. *Proceedings of the 15th International Conference on Real-Time and Network Systems*, 2007.

[89] Akshay Rajhans, Shang-Wen Cheng, Bradley R. Schmerl, David Garlan, Bruce H. Krogh, Clarence Agbi, and Ajinkya Bhave. An Architectural Approach to the Design and Analysis of Cyber-Physical Systems. *Electronic Communications of the EASST*, 21, 2009.

[90] Rudolf Schlatte, Bernhard Aichernig, Andreas Griesmayer, and Marcel Kyas. Resource Modeling for Timed Creol Models. *Electronic Notes in Theoretical Computer Science*, 266:63–75, October 2010.

[91] Cristina Seceleanu, Aneta Vulgarakis, and Paul Pettersson. REMES: A Resource Model for Embedded Systems. *Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems*, June 2009.

[92] Séverine Sentilles, Petr Stepan, Jan Carlson, and Ivica Crnkovic. Integration of Extra-Functional Properties in Component Models. *Proceedings 12th International Symposium on Component Based Software Engineering*, June 2009.

[93] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković. A Component Model for Control-Intensive Distributed Embedded Systems. *Proceedings of the 11th International Symposium on Component Based Software Engineering*, pages 310–317, October 2008.

[94] Mary Shaw. What Makes Good Research in Software Engineering? *Software Tools for Technology Transfer*, 4(1):1–7, 2002.

[95] Thomas Stauner, Olaf Müller, and Max Fuchs. Using HyTech to Verify an Automotive Control System. *Proceedings of the International Workshop on Hybrid and Real-Time Systems*, 1201, 1997.

[96] Jagadish Suryadevara, Aneta Vulgarakis, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. ProCom: Formal Semantics. Technical Report MDH-MRTC-234/2009-1-SE, Mälardalen University, March 2009.

[97] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[98] The Eclipse Foundation. Eclipse Platform. `http://www.eclipse.org/` (Last Accessed: 2012-02-01).

[99] The Eclipse Foundation. Eclipse Platform Debug. `http://www.eclipse.org/eclipse/debug/` (Last Accessed: 2012-04-14).

[100] Uppaal Cora. `http://people.cs.aau.dk/~adavid/cora/`, (Last Accessed: 2012-03-27).

[101] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.

[102] Aida Čaušević, Cristina Seceleanu, and Paul Pettersson. Modeling and Reasoning about Service Behaviors and their Compositions. *Proceedings of 4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, October 2010.

[103] Aneta Vulgarakis and Cristina Seceleanu. Embedded Systems Resources: Views on Modeling and Analysis. *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference, Workshop on Component-Based Design of Resource-Constrained Systems*, pages 1321–1328, July 2008.

[104] Aneta Vulgarakis, Cristina Seceleanu, Paul Pettersson, Ivan Skuliber, and Darko Huljenic. Validation of Embedded Systems Behavioral Models on a Component-Based Ericsson Nikola Tesla Demonstrator. *Proceedings of the 11th International Conference on Quality Software*, July 2011.

[105] Aneta Vulgarakis, Séverine Sentilles, Jan Carlson, and Cristina Seceleanu. Integrating Behavioral Descriptions into a Component Model for Embedded Systems. *Proceedings of the 36th Euromicro Conference on Software Engineering and Advanced Applications*, pages 113–118, September 2010.

[106] Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. Formal Semantics of the ProCom Real-Time Component Model. *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications*, August 2009.

[107] Farn Wang. Formal Verification of Timed Systems: A Survey and Perspective. *Proceedings of the IEEE*, pages 1283–1305, 2004.

[108] Michael Winter, Christian Zeidler, and Christian Stich. The PECOS software process. *Proceedings of the 7th International Conference on Software Reuse, Workshop on Components-based Software Development Processes*, 2002.

[109] Wayne Wolf. Embedded Computing - What Is Embedded Computing? *IEEE Computer*, 35(1):136–137, 2002.

[110] Marvin V. Zelkowitz and Dolores Wallace. Experimental Validation in Software Engineering. *Information and Software Technology*, 39:735–743, 1997.

[111] Marijan Zemljić, Ivan Skuliber, and Saša Dešić. Utilization of Open-Source High Availability Middleware in Next Generation Telecom Services. *Proceedings of the 17th International Conference on Software, Telecommunications and Computer Networks*, 2009.

Index

Errata

Page 64. The description of the CPU consumption should read "The consumption of the CPU can be modeled by a discrete variable, denoting the number of accumulated clock ticks, or processor load, or by a continuous variable, which represents the CPU usage in computerized systems."

Page 65. Table 4.1 should read

Resource Class	**Characteristics**
A (e.g. memory)	discrete: $\dot{c} = 0$ or $\dot{c} = \infty$ referable
B (e.g. CPU, bandwidth)	discrete: $\dot{c} = 0$ or $\dot{c} = \infty$ non-referable
C (e.g. CPU, energy)	continuous: $\dot{c} = n$, $n \in \mathbb{Z} - \{-\infty, +\infty\}$ non-referable

Table 1: Resource classes/characteristics.

Page 71. In Definition 8 max should read sum.