Mälardalen University Press Licentiate Theses
No. 108

A RESOURCE-AWARE COMPONENT MODEL FOR EMBEDDED SYSTEMS

Aneta Vulgarakis

2009

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering

ISSN 1651-9256
ISBN 978-91-86135-37-9
Printed by Mälardalen University, Västerås, Sweden

Abstract

Embedded systems are microprocessor-based systems that cover a large range of computer systems from ultra small computer-based devices to large systems monitoring and controlling complex processes. The particular constraints that must be met by embedded systems, such as timeliness, resource-use efficiency, short time-to-market and low cost, coupled with the increasing complexity of embedded system software, demand technologies and processes that will tackle these issues. An attractive approach to manage the software complexity, increase productivity, reduce time to market and decrease development costs, lies in the adoption of the component based software engineering (CBSE) paradigm. The specific characteristics of embedded systems lead to important design issues that need to be addressed by a component model. Consequently, a component model for development of embedded systems needs to systematically address extra-functional system properties. The component model should support predictable system development and as such guarantee absence or presence of certain properties. Formal methods can be a suitable solution to guarantee the correctness and reliability of software systems.

Following the CBSE spirit, in this thesis we introduce the ProCom component model for development of distributed embedded systems. ProCom is structured in two layers, in order to support both a high-level view of loosely coupled subsystems encapsulating complex functionality, and a low-level view of control loops with restricted functionality. These layers differ from each other in terms of execution model, communication style, synchronization etc., but also in kind of analysis which are suitable. To describe the internal behavior of a component, in a structured way, in this thesis we propose REsource Model for Embedded Systems (REMES) that describes both functional and extra-functional behavior of interacting embedded components. We also formalize the resource-wise properties of interest and show how to verify whether the behavioral models satisfy them.

To my parents

Acknowledgements

I have always known that I wanted to get a higher degree than Master of Science, and I have always been fascinated by research. However, I never thought that I would ever live in Sweden. Coming from Macedonia, Sweden has always seemed to me just "too north". But, then I was offered a Ph.D. candidate position at Mälardalen University, which I simply could not refuse. That is how my research journey started. I can not say that the journey has at all times been "a piece of cake" for me, but I can definitely say that I had great support from many people that made it a lot easier.

The work presented in this thesis would not have been possible without the encouragement and guidance of my supervisors. 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 always 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. You have not been just my supervisor, but an invaluable friend that has helped me in so many ways. Thank you so much for this!

I have authored and co-authored 16 different papers. I would never have done that without the help of very capable and hard working co-authors. Many thanks go to Tomáš Bureš, Jan Carlson, Aida Čaušević, Michel Chaudron, Ivica Crnković, Séverine Sentilles, Jagadish Suryadevara, Cristina Seceleanu and Paul Pettersson.

I would like to thank PROGRESS-ers Andreas Ermedahl, Hans Hansson, Björn Lisper, Kristina Lundqvist, Christer Norström, Sasikumar Punnekkat, Mikael Sjödin, and Gunnar Widforss. Without you PROGRESS would not have progressed to the point it is today. I would also like to thank 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 and Monika Matevska Stier.

Next, I would like to thank my officemates, Séverine Séntilles and Hongyu Pei Breivold 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 and meta-printers. I want to thank Adnan Čaušević, Aida Čaušević, Aleksandar Dimov, Ana Petričić, Andreas Hjertström, Antonio Cicchetti, Batu Akan, Cristina Seceleanu, Dag Nyström, Damir Isović, Daniel Sundmark, Farhang Nemati, Hongyu Pei Breivold, Hüseyin Aysan, Iva Krasteva, Jan Carlson, Jagadish Suryadevara, Johan Fredriksson, Johan Kraft, Josip Maras, Juraj Feljan, Kathrin Dannmann, Lars Asplund, Leo Hatvani, Luka Lednicki, Nikola Petrović, Marcelo Santos, Mikael Åsberg, Mikael Åkerholm, Moris Behnam, Pasqualina Potena, Radu Dobrin, Séverine Séntilles, Stefan (Bob) Bygde, Thomas Nolte, Tiberiu Seceleanu and Yue Lu. Most of you have been more friends than colleges to me. Not surprisingly, I would especially like to thank Juraj for being there and making my days brighter!

Many thanks also to my Bulgarian friend Velemira Slaveykova, and my Macedonian friends Bojana Bislimovska and Marija Taškova.

To my sister Sofija, her husband Boris and my nephew Filip - you have given me positive energy when I needed it the most.

Finally, Mirjana and Janko, my parents. Thank you for always being with me, and guiding me through life. Your love and support means the world to me!

The journey continues...

Aneta Vulgarakis
Västerås, September, 2009

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

List of Publications

Publications Included in the Licentiate Thesis

Paper A: Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel Chaudron. *A Classification Framework for Component Models*. Accepted to IEEE Transactions on Software Engineering (in the process of revision).

Paper B: Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković. *A Component Model for Control-Intensive Distributed Embedded Systems*. In Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE), Karlsruhe, Germany, October 2008.

Paper C: Aneta Vulgarakis and Cristina Seceleanu. *Embedded Systems Resources: Views on Modeling and Analysis*. In Proceedings of the 1st IEEE International Workshop On Component-Based Design Of Resource-Constrained Systems (CORCS 2008), IEEE CS, Turku, Finland, July, 2008.

Paper D: Cristina Seceleanu, Aneta Vulgarakis, and Paul Pettersson. *REMES: A Resource Model for Embedded Systems*. In Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009), IEEE CS, Potsdam, Germany, June, 2009.

Paper E: Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. *Formal Semantics of the ProCom Real-Time Component Model*. In Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Patras, Greece, August, 2009.

Other publications, not included in the thesis

Conferences and workshops:

- Aneta Vulgarakis and Aida Čaušević. *Applying REMES behavioral modeling to PLC systems*. In Proceedings of the 22nd International Symposium on Information, Communication and Automation Technologies (ICAT 2009), Sarajevo, Bosnia Herzegovina, October 2009.
- Aida Čaušević and Aneta Vulgarakis. *Towards a Unified Behavioral Model for Component-Based and Service-Oriented Systems*. In Proceedings of the 2nd IEEE International Workshop On Component-Based Design Of Resource-Constrained Systems (CORCS 2009), IEEE CS, Seattle, Washington, July, 2009.
- Aneta Vulgarakis. *Towards a Resource-Aware Component Model for Embedded Systems*. In Proceedings of the Doctoral Symposium of 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009), IEEE CS, Seattle, Washington, July, 2009.
- Tomáš Bureš, Jan Carlson, Séverine Sentilles, and Aneta Vulgarakis. *A Component Model Family for Vehicular Embedded Systems*. In Proceedings of the 3rd International Conference on Software Engineering Advances (ICSEA), Sliema, Malta, October 2008.
- Ivica Crnković, Michel Chaudron, Séverine Sentilles, and Aneta Vulgarakis. *A Classification Framework for Component Models*. In Proceedings of the 7th Conference on Software Engineering and Practice in Sweden, Göteborg, Sweden, October 2007.
- Séverine Sentilles, Aneta Vulgarakis, and Ivica Crnković. *A Model-Based Framework for Designing Embedded Real-Time Systems*. In Proceedings of the Work-In-Progress (WIP) track of the 19th Euromicro Conference on Real-Time Systems (ECRTS), Pisa, Italy, July 2007.

MRTC reports:

- Jagadish Suryadevara, Aneta Vulgarakis, Jan Carlson, Cristina Seceleanu, and Paul Pettersson, *ProCom: Formal Semantics*, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-234/2009-1-SE, M Mälardalen Real-Time Research Centre, Mälardalen University, March, 2009

- Cristina Seceleanu, Aneta Vulgarakis, and Paul Pettersson. *REMES: A Resource Model for Embedded Systems*. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-232/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, October, 2008

- Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, and Aneta Vulgarakis. *ProCom – the Progress Component Model Reference Manual, version 1.0*. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-230/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, June 2008.

- Tomáš Bureš, Jan Carlson, Séverine Sentilles, and Aneta Vulgarakis. *Towards Component Modelling of Embedded Systems in the Vehicular Domain*. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-226/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April 2008.

- Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, and Aneta Vulgarakis. *Progress Component Model Reference Manual - version 0.5*. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-225/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April 2008.

Contents

I

Thesis

Chapter 1

Introduction

An embedded system is a microprocessor-based system that is built (embedded) in a larger system that may or may not be a computer system. Embedded systems can be found in an enormous range of electrical items such as cellphones and PDAs, instruments such as GPS automotive navigation systems, and also large engineering systems such as traffic control systems, or control systems of nuclear power plants. Virtually any electronic device designed and manufactured nowadays is an embedded system, and virtually all people are touched by this technology.

Embedded systems have tightly constrained heterogenous requirements [1, 2]. They must often have low cost, constantly react to changes in the system's environment, must compute certain results in real time without delay and satisfy reaction constraints, such as deadlines and throughput, must be sized to fit on a single chip and consume minimum resources, and similar. Like all computing systems, embedded systems consist of hardware and software integrations, in which the software reacts to the environment. Nevertheless, in difference to other computing systems, most of the requirements of embedded systems are related to extra-functional properties (such as reliability and safety), and to limited resources. As such, design space exploration and verification at an early design stage are desirable.

During recent decades, the vast majority of functionality of embedded systems is realized with software. For example, up to 40 percent of the development time for an upper-class car is spent in car-IT (such as driver assistance) [3]. Nowadays, a car may hold up to 80 control-units that are cross-linked. The existing theories and methods for software development, when applied to software design of embedded systems, reveal the two major challenges of embedded system design. The first challenge is to provide an artifact (an embedded computer system) that provides the specified services under given constraints. The second challenge is that relevant properties of this artifact need to be modeled at different levels of abstraction by models of adequate simplicity [4]. Accordingly, there is a need for improved software development techniques and processes that will let developers to tame software's growing complexity, while reducing time to market and development costs. A promising approach to handle the complexity, reduce time to market, introduce structure and abstractions, lies in the adoption of the component based software engineering (CBSE) paradigm. The central point of CBSE has been reuse, but for embedded systems the structure and abstractions introduced by components are equally important as a basis for construction of abstract formal models. In that sense, the CBSE paradigm facilitates the use of formal methods, in modeling and analyzing the used components, to tackle the need for early stage verification.

The goal of this thesis is to propose solutions for modeling modern real-time embedded systems, in a component-based fashion, in an attempt to manage the associated extra-functional properties including resource constraints. Following the CBSE spirit, this thesis introduces an analyzable component model for development of distributed embedded systems, which tries to meet the designer's needs for building vehicular embedded systems in particular. The component model is built in two layers, in order to address in same time loosely coupled subsystems (big parts) and control tasks (small parts) of a system. These parts differ from each other in terms of execution model, communication style, synchronization etc., but also in kind of analysis which are appropriate.

While a fully and semantically described interface of a component defines the intent of a component, that is, what the component does, the content of a component describes how the intent is realized [5]. Such information is hidden from the end user and becomes important only to those who intend to modify the component. Hence, in order to provide the designer with means for representing the internal behavior of a component, in a structured way, in this thesis we also introduce a model that describes both functional and certain class of extra-functional (such as timed behavior and resource consumption) behavior of components. Any modification of a component's internal description, even if gives rise to a functionally equivalent model, might alter the component's original properties wrt timing and resource usage. To prove that the desired properties are still exhibited by a modified component model, we formalize the resource-wise properties of interest and show how to verify such behavioral models against them.

The work has been carried out within PROGRESS [6], a Swedish national centre for development of predictable embedded systems. The main aim with PROGRESS is to promote the development of embedded systems to a mature engineering discipline. Thus, PROGRESS should provide theories, methods and tools, which will increase quality, reduce costs and complexity in the development of embedded systems.

The following section provides the background for the basic concepts of CBSE, and formal analysis, as a foundation for reading the remainder of the thesis. In the end of this chapter the overview of the thesis is presented.

1.1 Preliminaries

1.1.1 Component Based Software Engineering

The basic rationale for the field of CBSE [7, 8] is the idea of constructing systems by reusing existing 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 CBSE 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 others 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.

Although CBSE has been widely used for software development of desktop and distributed enterprise applications, there is still a lack of broadly adopted component technology standards which are suitable for embedded systems. Due to the specific characteristics of embedded systems, a component architecture for embedded systems must have low overhead, be flexible to accommodate application unique requirements, and be able to address relevant extra-functional issues (resource restrictions, timeliness, safety and dependability).

In CBSE the smallest functional building unit is a *component*. The idea behind components originates from a paper published by M.D. McIlroy [9] at the 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 same time contradictory directions. Up until today there is no generally accepted definition of what a component is. A definition that is commonly cited in publications is the one from Szyperski [8], which focuses on the key characteristics of components:

> *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.

A component based system is a composition of components, where a component is an open system that communicates with the environment through its interfaces. The behavior of an embedded system should be predictable, both functionally and with respect to timeliness and resource usage. 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 in embedded systems.

Component models are used in the development of components to define standards for their interfaces, illustrate their dependencies, specify their properties and composition mechanisms [7]. In other words, the component model embraces a set of rules regulating how the components may or may not be used.

Nowadays a number of component models for embedded systems exist [10–15], however they seldom provide support for relevant extra-functional properties.

1.1.2 Formal Analysis

Component technologies for embedded systems should support system development with high degree of predictability. Predictability concerns the possibility to guarantee absence or presence of certain properties, or to predict/guaranty a value of a property. The employed predictability analysis should guide the design and selection of hardware and software system components.

Formal analysis is a process 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 [16, 17]. 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 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, an automata model describing the possible system behaviors is fed into a model-checking tool, together with a desired property (requirement) 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. 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, 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, and the UPPAAL [18] 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 automata and the model of priced (or weighted) timed automata [19, 20], an extension of timed automata [21] with prices/costs on both locations and edges.

Timed Automata

The model of timed automaton (TA) [21] 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. 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. Edges are labeled with guard expressions, an action, and a reset set i.e., set of clocks to be reset. We say that an edge is enabled if the guard evaluates to true and the source location is active. Locations are labeled with clock constraints called invariants, which enforce that the location is left before they are violated. The semantics of TA is defined in terms of a timed transition system. A state of TA depends on its current location and on the current values of its clocks. The transitions between states can be of two kinds: *delay* and *discrete*. Delay transitions are result of passage of time while staying at some location. Discrete transitions are result of following an enabled edge in a TA to its destination location with the clocks in the reset set, set to zero. Systems comprising multiple concurrent processes are modeled by networks of timed automata, which execute with interleaving semantics and synchronize on channels.

UPPAAL is a tool set for modeling, simulation, and verification of networks of timed automata. The UPPAAL model checker supports verification of temporal properties, including safety and liveness properties. The simulator can be used to visualize counter examples produced by the model checker. UPPAAL automata extend timed automata by introducing bounded integer variables, binary and broadcast channels, and urgent and committed location.

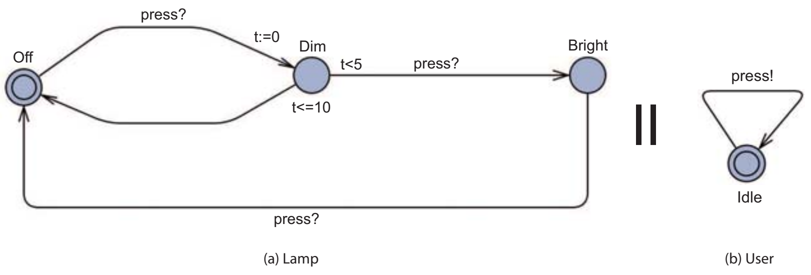


Figure 1.1: Timed automaton of a lamp and a user.

An example of a network of timed automata modeled in UPPAAL is shown in Figure 1.1. The timed automata consist 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 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 by sending and receiving events through channels. Sending and receiving via a channel press is denoted by press! and press?, respectively.

Priced Timed Automata

Priced timed automata extend timed automata with prices/costs on both locations and edges. The cost labeling a location represents the price per time unit for staying in that location, whereas the cost labeling an edge represents the price for taking the transition. As such, every run in the priced timed automation has a global cost, which is the accumulated price along the run of every delay and discrete transition. Multi priced automata [22] are extension to priced timed automata in which a timed automation is augmented with more than one cost variable. In this thesis, the framework of priced timed automata is used for formally analyzing resource consumption in embedded systems.

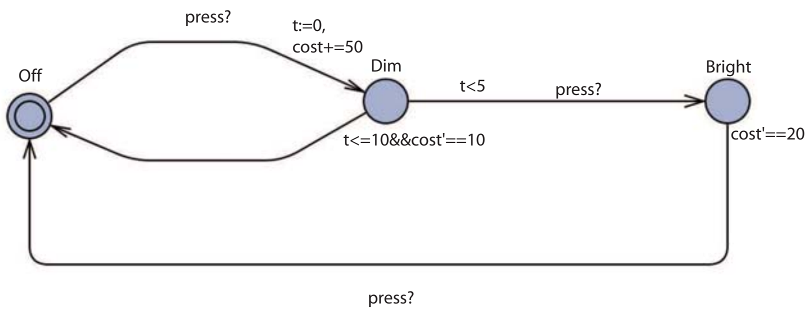


Figure 1.2: Priced timed automaton of a lamp.

Switching on a lamp and letting it burn uses energy, therefore in Figure 1.2 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 implicitly by specifying a rate. 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.

1.2 Thesis Overview

The thesis is divided into two distinct parts. The first part is a summary of the performed research. Chapter 1 describes the background and motivation of the research. Chapter 2 formulates the main research goal and introduces the research questions. Chapter 3 describes the research results and recapitulates the research questions. Chapter 4 presents the research method used. Chapter 5 surveys related work. Finally, Chapter 6 concludes the thesis, summarizes the contributions and outlines future work that formulates guidelines for further PhD studies.

The second part of the thesis presents a collection of peer-reviewed journal, conference and, workshop papers that contain details of the answers of the research questions, methods and, results presented in the first part of the thesis. The following five papers are included in the second part of the thesis:

Paper A. ”A Classification Framework for Component Models”. Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, Michel Chaudron. Accepted to IEEE Transactions on Software Engineering (in the process of revision).

Summary: This paper presents a survey of a number of component models, described and classified with respect to a four 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 mostly written with equal contribution of the first three authors. All the coauthors have contributed with ideas, discussions, and reviews. I was responsible mainly for the lifecycle section 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 CBSE and empirical software engineering researchers and experts from different engineering domains.

Paper B. ”A Component Model for Control-Intensive Distributed Embedded Systems”. Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, Ivica Crnković. In Proceedings of the 11th International Symposium on

Component-Based Software Engineering (CBSE2008), 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. I took part in the discussions and contributed with writing and improving parts of the paper, 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.

Paper C. ”Embedded Systems Resources: Views on Modeling and Analysis”. Aneta Vulgarakis, Cristina Seceleanu. In Proceedings of COMPSAC, the 1st IEEE International Workshop On Component-Based Design Of Resource-Constrained Systems Software and Applications Conference (CORCS), 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 all the authors. I was specifically working on the code-level and UML- based resource modeling and analysis.

Paper D. ”REMES: A Resource Model for Embedded Systems”. Cristina Seceleanu, Aneta Vulgarakis, Paul Pettersson. In Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009), IEEE CS, 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 [23] the TCS system presented as a case study in the paper.

Paper E. ”Formal Semantics of the ProCom Real-Time Component Model”. Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. In Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Patras, Greece, August, 2009.

Summary: In this paper, we define the formal 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 driver and principal author of this paper and contributed with defining a formal 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 the second author of this paper.

Chapter 2

Research Problems

This chapter presents the scope of our work by formulating the research goal, and introducing the research questions that address the goal.

2.1 Problem Description

Our research is in the area of component based development for embedded systems, and was driven by the problems coming from the embedded systems domain, such as managing complexity, distribution, resource limitations, analysis, managing the strong coupling between the components, the system and the target platform. In this thesis, we address the problem of modeling modern real-time embedded systems, in a component-based fashion, in an attempt to tackle the system complexity and manage the associated resource constraints. Concretely, the overall goal of the thesis is

> *design of an analyzable component model for real-time embedded systems*

This goal is broad and admits various answers. We approach the goal by answering to three research questions that address component models for embedded systems and behavioral modeling of embedded systems, which we formulate in the next section.

2.2 Research Questions

Research question 1.

Component models are indispensable to CBSE, as they define rules for constructing individual components and for assembling them into systems. There are various component models proposed in the literature as suitable for the development of real-time embedded systems. Some characteristics are shared among the component models, yet each of the latter has distinct characteristics too. Therefore, it is important to analyze and compare the existing component models in order to identify the most interesting for the development of embedded systems. Such motivation justifies our first research question:

> *What are the common characteristics and differences between existing component models?*
>
> *(Q1)*

Research question 2.

One of the main characteristics of embedded systems is the restriction of available resources. The diversity of approaches on resource modeling and analysis existing in the literature [24–31] indicate the difficulty 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 should use a modeling language that incorporates resources as primitive types, that is, built-in the model. Ideally, the same language should provide support for modeling and analyzing functional and timing behavior too, besides the resource-wise behavior of the embedded system. This would allow both separation of concerns as well as simple model-to-model transformations, for analysis purposes. Accordingly, the second research question can be formulated:

> *How can we model and formally analyze functional, timing and resource-wise behavior in a unified manner?*
>
> *(Q2)*

Research question 3.

The potential benefits of CBSE are as attractive in the domain of embedded systems as they are in other areas of the software industry. Beside component models, component technologies form another central concept of CBSE. 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., Koala [10], Robocop [14], BlueArX [15], AUTOSAR [32], COMDES-II [33], Pecos [12], Rubus [13], and SaveCCM [11]), CBSE 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 [34] 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). Hardware and software models annotated with performance, resource consumption or size information can be beneficial to embedded system designers. 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 them to reason about timed behavior, resource consumption and so on, without going down to to the instruction level. For instance, architectural models may be used for modeling basic functionality, and behavioral models for modeling functional and extra-functional behavior. Also, embedded system developers must verify that applications meet their functional and extra-functional specification. All these requirements should be reflected in the component model. However, the specifications of many component models are defined informally and component models suffer from incomplete and imprecisely defined syntax and semantics. Formalization of component models using formal methods can provide precise definitions. The formalization should be designed to unambiguously describe the elements of the component model. Thus the third question is as follows:

> *What is an appropriate component model for real-time embedded systems and how can we describe its elements in an unambiguous way?*
>
> *(Q3)*

Chapter 3

Research Results

The current chapter presents the main lines of our contribution and research results, starting from the research questions proposed in Chapter 2. The research does not provide complete answers to the questions, but only partially answers them and gives directions for further research. The following sections describe each research topic.

3.1 Classification of Component Models

Goal: The large number of existing component models having particularities, different aims, sometimes unclear concepts, but also many similarities, calls for a systematic analysis of such models. The goal of this research is to provide a classification framework that will identify and discuss the basic principles of component models. Later, according to this classification framework, existing component models may be classified and compared. This framework could also help in the design of new component models, since one of its goals is to identify the elements of a component model that would be important when designing a (new) component model.

Research process: The research was performed in several iterations of observations, analysis, classification, and validation. We have started with a large number of component models, by first studying the state of the art on general principles of CBSE, and the existing literature on classification of architecture description languages, quality attributes and component models. From this we have gained knowledge and made the first version of the classification framework. Then, we discussed this classification framework with several CBSE and empirical domain software engineering researchers and experts from different engineering domains. The discussions have led to a refinement of the framework. In the next iterations the refined framework was mapped on the studied and new component models and discussed with other researchers and practitioners. The research process has been completed when all studied component models complied well with the classification framework.

Results: The result of this research is a classification and comparison framework for component models. The classification framework includes four dimensions (lifecycle, constructs, extra-functional properties and domains) in which the basic characteristics and principles of component models are distinguished: (i) The lifecycle dimension identifies the support provided (explicitly or implicitly) by the component model during components' lifecycle, such as modeling of components and component based systems, implementation, packaging and distribution, and deployment of components into an executable system or some target environment. (ii) The constructs dimension identifies (i) the component interface used for the interaction with other components and external environment, and (ii) the means of component binding and communication. (iii) The extra-functional properties demension, identifies specifications of different property values, and means for management and their composition. (iv) The domain dimension classifies component models according to their usage domain: general-purpose, specialized or generative. Details about the classification framework can be found in the included paper A.

Limitations and future work: The proposed framework can always be extended since it does not comprise all the elements of all component models. Some component models have specific solutions related to particular models or technologies. Furthermore, we have not characterized the components themselves (e.g., internal behavior, whether components are active or passive, etc.) and the list of component models that we have studied can always be extended. This is subject to future work. However, to our knowledge the proposed framework identifies the minimal criteria for considering a model to be a component model and it groups the basic characteristics of the models.

3.2 The REMES Behavioral Model

Goal: The competing or inconsistent requirements of real-time embedded systems, such as minimizing memory consumption while still ensuring all deadlines are met at run-time, call for rigorous analysis of the system's resource consumption already at early design stages. Our goal is to propose a model for formal modeling and analysis of embedded resources. The envisioned outcome has been a behavioral model with support for reasoning about functional, timing and resource-wise behavior in a unified way.

Research process: The research included several iterations. First we have studied and grouped several representative frameworks that model and estimate resource consumptions of embedded systems. The studied frameworks indicate the possible difficulty of reasoning about all resource types within the same theoretical framework. While studying them, we developed our own view on how to model and, carry out analysis of embedded resources. In developing our behavioral model we were inspired by the CHARON [35] modeling language, used for specifying embedded systems as communicating agents, while relying on hybrid automata for the semantic translation. Our main contribution is the introduction of resource as a built-in data type, and the addition of other constructs (like the conditional connector) that facilitate the application of REMES to modeling both functional and extra-functional behavior of component-based embedded systems.

Results: The result of this research is the behavioral model REMES (REsource Model for Embedded Systems) and associated analysis techniques for performing resource-wise behavioral analysis, such as feasibility analysis, optimal/worst-case resource consumption, and trade-off analysis. In our studies we consider resources as quantities of finite size, and we classify them by their discrete or continuous nature, the way they are consumed and/or allocated and released, and whether they can be referred to, or not. The classification of resources is not tied to any particular formal semantic representation. Consequently, REMES can model number of generic resources (e.g., memory, CPU, energy, bandwidth, etc.). REMES is a dense time state-based hierarchical behavioral language with a notion of explicit entry- and exit points, continuous variables, flows and progress invariants. For formal analysis purposes, REMES can be semantically translated into timed automata or (multi) priced timed automata depending on the analysis goals (i.e., timing analysis, resource consumption, etc.). The analysis of REMES is based on a weighted sum in which the variables capture the accumulated consumption of resources, respectively.

Details about the variety of approaches on resource modeling and analysis existing in the literature and about REMES can be found in the included papers C and D, respectively.

Limitations and future work: REMES is tailored for embedded systems, but it is also suitable for modeling behavior of reactive systems. We have performed analysis in UPPAAL CORA [23], which can currently only handle priced timed automata models where the weighted sum is monotonically increasing. As future work, we plan to integrate REMES in the PROGRESS IDE, by first connecting REMES with ProCom, and second, by implementing the kinds of analysis that we are interested in, in UPPAAL CORA. The scalability and appropriateness of REMES for real-world industrial applications is unfortunately not exercised in this work. We plan to apply REMES on a series of complex systems, in order to better identify its weaknesses and limitations. The proposed cost analysis model for REMES is platform-aware. Hence, as future work, it could benefit from including abstractions of platform specific tools, such as the associated compiler, linker etc. We do believe that in order to derive the costs, one could apply static analysis techniques on already implemented components. We underline the fact that the values of the weights are a subjective matter; the way they are chosen depends mostly on the designer's experience, application domain and on the analysis goals.

3.3 The ProCom Component Model

Goal: The goal of this research is to propose a component model suitable for development of real-time embedded systems, in particular vehicular-, and telecommunication systems. The type of embedded systems found in the targeted domains typically have specific characteristics when considered at different levels of granularity. The loosely coupled subsystems differ from the control loops controlling a certain piece of hardware, with respect to execution model, communication style, synchronization, etc. Also, there are differences in the kind of information that must be available and the type of analysis that is appropriate. Our goal has been the design of a component model that supports both a high-level view of loosely coupled subsystems encapsulating complex functionality, as well as low-level view of control loops having dedicated, restricted functionality, simpler communication, which control a certain piece of hardware.

To enable formal analysis, the component model should be given a formal semantics. The formalization should not only describe the modeling elements of the component model in a rigorous way, but also provide support for reasoning about functional and extra-functional behavior of the modeling elements.

Research process: The research process included studying existing component models (in particular SaveCCM [11]) for the development of embedded systems and discussions with domain experts from the targeted domains. From this we have identified the following requirements that a suitable component model should fulfill:

- manage complexity;
- manage the strong coupling between the system and the targeted platform;
- deal with different types of components with respect to granularity, functionality and semantics;
- utilize resources efficiently;
- provide support for different kinds of analysis (functional behavior, timed behavior, resource usage, etc.).

Following these requirements, the ProCom component model has been developed in several iterations.

The formalization of ProCom's architectural elements included several iterative steps as well. We first started with studying the formal semantics of the SaveCCM [36] component model (ProCom's predecessor). The formal semantics of SaveCCM is based on timed automata (TA). While carrying out the formalization work, we have managed to comprise the necessary semantic descriptions in a simpler and more compact form than TA provides: a high-level FSM-like model that abstracts away some aspects present in the corresponding TA.

Results: The result from this research is the ProCom component model and an associated architectural semantics based on an FSM - like representation. In comparison to other component models targeting embedded systems, ProCom addresses quality attributes, resource consumption and distribution more systematically. 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. The upper layer, called ProSys, is intended for modeling the embedded system as a collection of complex active and concurrent subsystems, communicating via asynchronous message passing. The lower layer, ProSave, serves for modeling the internal design of a subsystem down to primitive functional components implemented by code. ProSave components are passive units, which communicate based on a pipe-and-filter architectural style with an explicit separation between data and control flow. In both layers, information about a component is stored in a repository, including requirements, textual documentation, formal semantics and REMES behavioral models.

The FSM language, used for the ProCom formalization, has notions of urgency, implicit timing and priorities. Its formal semantics, hence of the architectural elements of ProCom, is expressed in terms of TA with priorities [37] and urgent transitions [38]. The FSM language has graphic simplicity, making it simpler than the corresponding TA model, as it abstracts from real-valued variables and synchronization channels. The FSM models of ProCom systems can be analyzed both in a dense-time underlying framework, as well as in a discrete-time one, since TA has been recently given a sampled semantics [39]. Hence, tools such as UPPAAL [18] can be employed for early-stage verification of ProCom models, whereas discrete-time model-checkers, such as DT-Spin [40], could be used for later-stage analysis, as sampled time semantics is closer to the actual software or hardware system with a fixed granularity of time.

Details about the design and formal semantics of the ProCom component model can be found in the included papers B and E.

Limitations and future work: The current version of the ProCom component model is developed for the vehicular domain and focuses on the design of a class of distributed embedded systems that primarily perform real-time controlling tasks. In the future, ProCom may be extended for instance to the telecommunication domain. Additionally, ProCom has not yet been industrially verified on a real-world example case study but we plan to do this as future work. Although the FSM formalism sets the grounds for formal analysis, the semantic descriptions focus only on formalizing the correct behavior of ProCom architectural elements, without consideration for efficiency in formal analysis of the resulted models. As future work, we plan to integrate REMES with the formal semantics of ProCom.

3.4 Questions Revisited

In this section, we show how the research results and included papers give answers to the research questions.

Question *Q1: What are the common characteristics and differences between existing component models?*
From the research summary, we can see that this question is answered by the first research topic and by paper A in which a classification framework for component models is introduced. Among other things, the four-dimensional framework identifies the common characteristics and differences between existing component models.

Question *Q2: How can we model and formally analyze functional, timing and resource-wise behavior in a unified manner?*
The second research topic and included papers C and D contribute with answers to this question. It is our intention that REMES is used to model both functional and extra-functional behavior of interacting embedded components, while relying on the solid verification framework of priced timed automata.

Question *Q3: What is an appropriate component model for real-time embedded systems and how can we describe its elements in an unambiguous way?*
The second and third research topics give answers to this question. ProCom is targeting the development of embedded systems and the semantics of the ProCom architectural elements can be presented with the FSM language introduced in paper E. REMES can be used for internal behavioral modeling of ProCom-based systems, whereas the underlying PTA framework in which REMES is translated can be used for formal verification of functional and extra-functional requirements/properties.

Hence, all three questions have been at least partly answered. Needless to say, we have provided one possible answer to each question, out of a possibly large pool of valid answers.

Chapter 4

Research Method

Different research methods are suitable for different settings, and similarly different validation techniques are suitable for different types of results. The methodology that has been used in this research is based on the research steps presented in [41]. The main activities are:

1. Identification of the research problem from real-world software engineering issues.

2. Transferring the problem to a research setting, and defining the research questions. In this stage the research problem is often refined and narrowed down.

3. Analysis of the current state of the art addressing the research questions.

4. Answering the research questions and presenting the research results. This stage includes several iteration steps, such as observations, discussions, analysis and improvement of the research results.

5. Checking whether the research results adequately answer the research questions. This can be performed in several different ways, e.g., by formal proofs, by performing case-studies, by implementation of a prototype, by describing experiences etc.

6. Validating the research results in the sense of checking whether they are feasible for the real-world software engineering problem.

Following the abovementioned activities to a great extent, we have initially defined the research problem, as stated in Chapter 2. Second, from the research problem we have identified the research questions and presented them in Chapter 2. Later we have conducted a thorough investigation of the current state of the art addressing the research questions. This investigation has resulted in two papers: paper A and paper C. Further in papers B, D and E we have presented our research results on designing a component model suitable for development of embedded systems and a behavioral model for modeling and analyzing functional, timing and resource-wise behavior in a unified manner, which are summarized and discussed in Chapter 3. Since the research done so far does not yet offer a complete solution to the research problem (a thorough validation of the results is missing), the research results have been applied on simple yet relevant "research examples" presented in papers A, B and D. Accordingly, in paper A the classification framework was demonstrated on a considerable number of component models. In paper B we have exemplified the ProCom component model on an electronic stability control system of a car, but a deeper analysis of a real case is needed. Further in paper D, we have performed a small case study demonstrating the principles of our resource modeling and analysis approach. The case study has been conducted on an abstracted version of the internal design of a temperature control system for a heat producing reactor. Again, a more detailed case study for a complete evaluation is needed. Consequently, in the aforementioned research methodology, we have completed the first 3 activities. The methodology awaits the validation of the research results, which might entail improving and extending the research results towards applicability on real-world engineering problems.

Chapter 5

Related Work

This chapter relates the contributions presented in this thesis to relevant research and practice areas, subdivided into two sections. Paper A and Paper C contain extensive related work and state of the art so here we give only a short summary.

5.1 Component Models for Embedded Systems

Nowadays many component models exist, either general purpose or dedicated to specific domain. Still, only 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. Often the component models 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) [32] 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. 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. Moreover, 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 upcoming AUTOSAR 4.0 release should contain a meta-model extension for specifying timing properties and constraints of software components and it is expected that TIMMO [42] project results will strongly influence future AUTOSAR releases with respect to timing modeling.

BlueArX [15, 43] 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 an analytic interface which is used to store components’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 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 [44] to analyze object code and to extract the worst-case execution time of a component, and SymTA/S [45] 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 a 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) [33] 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. [46] show how a COMDES-II system can be equivalently transformed into UPPAAL timed automata, and verified with preservation of system operational semantics.

IEC 61499 [47] is developed by the International Electrotechnical Commission (IEC) to support the development of automation and control systems. It has evolved from IEC 61131-3 [48] standard that is widely used in the development of software for PLCs. IEC 61499 components are called function blocks. Similar to ProSave, the data between the blocks is transferred using pipe-and-filter paradigm and the execution of the function blocks is event driven. In comparison to ProCom, there is no support for specifying or reasoning about extra-functional properties.

Koala component model [10] 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. Since Koala components are delivered as source code, it is possible to statically analyze components and systems built by composing them. To some extent, Koala allows calculating and predicting resource consumption (e.g., static memory), but it lacks support for managing other extra-functional properties. Compared to ProCom, Koala is geared towards less safety-critical applications.

PECOS [12] 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 [49] component model is developed at Carnegie-Mellon University. Its purpose is to be used as a basis for PECTs (Prediction-Enabled Component Technologies), which are concerned with providing 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 interface and analysis theory. Analytical 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 [14] component model is a successor of the Koala component model, and is developed out 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 [13] 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.

SaveCCM [36] is a component model for embedded control applications of vehicular systems developed at Mälardalen University. 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. Component behavior modeling is done using timed automata extended with tasks. Nevertheless, ProCom has a clearer concept of composite components, and addresses distribution and extra-functional properties more systematically.

5.2 Resource Modeling and Analysis

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 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 research detailed in paper C, this section summarizes the related work on modeling and analyzing resources in component-based real-time embedded systems and compares them with the REMES behavioral model proposed in this thesis. The related work may be grouped into three categories.

First, research has been devoted to predicting code-level resource consumption of component assemblies. In Koala [27] component model, compositional ways of estimating static memory consumption have been performed for applications in which the instantiated components of a composition are known prior to run-time. The resource information is exposed through a spacial type of component's interface, called IResource. The interface contains information about different types of memory and a formula for estimating the memory size of each type of memory is added to the IResource implementation. The technique supports budgeting i.e., the expected values of the resource consumption of non implemented components can be also accounted for. In Robocop [28] component model is presented a scenario-based prediction of run-time resource consumption. Robocop resource model specifies the predicted resource consumption for all operations implemented by the services of an executable component. The resource consumption is given as a number of cost functions. The resources that are claimed and released are specified per operation. Similar to Koala, this method is also dealing with static resource consumption, since it is considered that the consumption of resources is constant per operation. Both of the aforementioned approaches deal with low-level code-driven resource estimates, which can only be used in cases when one has access to the components implementations. However, more abstract descriptions of expected resource usage may be needed for not-yet implemented components, or for guiding the selection of components from the repository. In such cases, the designer could first employ REMES for early resource usage analysis, and then apply the approaches mentioned above.

The second category is represented by the attempts of software modeling languages and profiles (e.g.,UML [50], UML/SPT [29] and MARTE [51]) to tackle the modeling and analysis of embedded resources. Amar et al. [30] 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) [29] 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 new profile, MARTE (Modeling and Analysis of Real-Time and Embedded systems) [51], which emerged from the UML/SPT profile is dedicated to complement UML with the required extensions for supporting modeling and analysis of embedded real-time systems. The new profile should address specification of not only real-time constraints but also other embedded extra-functional characteristics such as memory and power consumption and modeling and analysis of component-based architectures. It provides a basic framework for platform-based modeling, the General Resource Modeling (GRM). It 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. Although graphical and intuitive, these 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 [25, 26], proposed by Lee et al. They propose a family of process-algebraic formalisms, developed to unify formal modeling and analysis of embedded systems resources. 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, probability 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. The framework is theoretically rich, however it is not intuitive and the tool support is not equally mature. Ouimet et al. [31] 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.

Chapter 6

Conclusions and Future Work

In this thesis we have addressed the design, behavioral modeling and analysis of resource-aware component model for development of distributed embedded systems, vehicular embedded systems in particular. We have exemplified the applicability of the results presented on two small case studies. However, full validation of the research results in more realistic case studies is subject to future work.

6.1 Contributions

The main contributions of the presented research are summarized as follows:

A four dimensional classification framework. In this thesis we present a classification and comparison framework for component models. The classification framework consists of four dimensions (lifecycle, constructs, extra-functional properties and domains) in which the basic characteristics and principles of component models are distinguished.

A two-layered ProCom component model for embedded systems. Comparing with other component models targeting embedded systems, the ProCom component model addresses quality attributes, resource consumption and distribution more systematically.

An unambiguous and compact description of the modeling elements of ProCom. The description is based on an extension of finite-state machines and sets the ground for formal analysis of systems built out of ProCom elements. The proposed finite-state machine language has graphical appeal, making it simpler than the corresponding timed automata model, and it abstracts from real-valued variables and synchronization channels.

REMES **behavioral language.** 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. It is our intention REMES to be used for unified modeling and formal analysis of functional, timing and resource-wise behavior of embedded systems.

Performing resource-wise analysis. We present a method for encoding the resource-wise analysis problem as a weighted sum in which the variables capture the accumulated consumption of resources, respectively. Thus, we perform three types of analysis: feasibility analysis, optimal or worst-case resource consumption analysis, and trade-off analysis. Feasibility analysis checks 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. Optimal or worst-case resource consumption analysis returns the cost of the "cheapest", and/or most "expensive" trace that will eventually reach some goal. This analysis may help in resolving the possible non-determinism in a component implementation. Trade-off analysis is a systemic 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.

6.2 Future Research Directions

There are many possible future extensions of the work presented in this thesis. The current version of the ProCom component model is primarily targeting vehicular systems and focuses on design of a class of distributed embedded systems that execute real-time controlling tasks. As future work, ProCom component model may be extended for instance to the automation domain. Additionally, ProCom has not yet been industrially verified on an industry case study and we plan to do this as future work.

We have already started with studying the applicability of REMES to other domains related to embedded systems, such as service-oriented systems and programmable logic controllers. In future, we plan to integrate REMES and its notion of resources in the ProCom component model. Here mapping between ProCom ports and REMES entry/exit points should be formally defined. Further, we plan to perform automatization of the process of predicting the resource usage of components and systems. As such, REMES GUI should become part of the ProCom development IDE, that is currently in phase of development. We have performed analysis in UPPAAL CORA, which can currently only handle priced timed automata models where the weighted sum is monotonically increasing. Therefore, it is left for future work to conduct a case-study which tackles feasibility analysis problem for a system in which some of the edge prices are negative so that the global cost function is non-monotonically increasing. In addition, all of the resource-wise verification algorithms presented in this thesis need to be implemented in UPPAAL CORA and the scalability of the approaches should be checked. As future work, we also plan to integrate REMES with the formal semantics of ProCom. Another opportunity for future work is to investigate the compositional reasoning i.e., analyzing each component of the system in isolation and allowing global properties (such as resource consumption of the whole system) to be inferred about the entire system. This of course will leave us the obligation of proving that the component specifications in turn imply the specification of the entire system.

Bibliography

[1] Frank Vahid and Tony D. Givargis. *Embedded System Design: A Unified Hardware/Software Introduction*. Wiley, international student edition edition, October 2001.

[2] Thomas A. Henzinger and Joseph Sifakis. The Embedded Systems Design Challenge. In *Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science*, pages 1–15. Springer, 2006.

[3] Gerd Beneken, Ulrike Hammerschall, Manfred Broy, Mara Victoria, Cengarle Jan, Jrjens Bernhard Rumpe, and Maurce Schoenmakers. Componentware - state of the art 2003. background paper for the understanding components workshop, 2003.

[4] Hermann Kopetz. The Complexity Challenge in Embedded Systems Design. In *Proceedings of the 11th IEEE International Symposium on Object/Component/Service-Oriented Real-time Distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[5] Ben Whittle. Models and Languages for Component Description and Reuse. *SIGSOFT Softw. Eng. Notes*, 20(2):76–89, 1995.

[6] PROGRESS Project. `http://www.mrtc.mdh.se/progress/` (Last Accessed: 2009-08-12).

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

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

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

[10] 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.

[11] 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.

[12] M Winter, C Zeidler, and C Stich. The PECOS software process. *Workshop on Components-based Software Development Processes, ICSR*, 2002.

[13] 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. In *Proceedings of the 3rd IEEE International Symposium on Industrial Embedded Systems*, June 2008.

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

[15] 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. In *ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 307–311, Washington, DC, USA, 2008. IEEE Computer Society.

[16] 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. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 244–256, London, UK, 1996. Springer-Verlag.

[17] Thomas Stauner, Olaf Müller, and Max Fuchs. Using HYTECH to Verify an Automative Control System. In *HART '97: Proceedings of the International Workshop on Hybrid and Real-Time Systems*, pages 139–153, London, UK, 1997. Springer-Verlag.

[18] UPPAAL. `www.uppaal.com`, accessed August 2009.

[19] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. In Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Proceedings of the 4th International Workshop on Hybris Systems: Computation and Control*, number 2034 in Lecture Notes in Computer Sciences, pages 147–161. Springer–Verlag, 2001.

[20] Kim Guldstrand Larsen and Jacob Illum Rasmussen. Optimal Reachability for Multi-Priced Timed Automata. *Theor. Comput. Sci.*, 390(2-3):197–213, 2008.

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

[22] T. Brihaye, V. Bruyère, and J-F. Raskin. Model-Checking for Weighted Timed Automata. In *Proceedings of FORMATS-FTRTFT*, number 3253 in Lecture Notes in Computer Science, pages 277–292. Springer–Verlag, 2004.

[23] UPPAAL CORA. `http://www.cs.aau.dk/∼behrmann/cora/`, accessed August 2009.

[24] Insup Lee, Jin-Young Choi, Hee-Hwan Kwak, Anna Philippou, and Oleg Sokolsky. A Family of Resource-Bound Real-Time Process Algebras. In *FORTE*, pages 443–458, 2001.

[25] Insup Lee, Anna Philippou, and Oleg Sokolsky. A General Resource Framework for Real-Time Systems. In *RISSEF*, pages 234–248, 2002.

[26] Insup Lee, Anna Philippou, and Oleg Sokolsky. Resources in Process Algebra.

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

[28] Merijn de Jonge, Johan Muskens, and Michel Chaudron. Scenario-Based Prediction of Run-Time Resource Consumption in Component-Based Software Systems. In *Proceedings of the 6th ICSE Workshop on*

Component-based Software Engineering (CBSE6), pages 19–24. IEEE, 2003.

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

[30] Hany H. Ammar, Vittorio Cortellessa, and Alaa Ibrahim. Modeling Resources in a UML-Based Simulative Environment. In *AICCSA*, pages 405–410, 2001.

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

[32] AUTOSAR Development Partnership. AUTOSAR – Technical Overview V2.2.1, 2008. `http://www.autosar.org/download/AUTOSAR_TechnicalOverview.pdf`.

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

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

[35] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical Modeling and Analysis of Embedded Systems. *Proceedings of the IEEE*, 8(3):231–274, 1987.

[36] 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.

[37] Alexandre David, John Håkansson, Kim Guldstrand Larsen, and Paul Pettersson. Model Checking Timed Automata with Priorities using DBM Subtraction. In *4th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'06)*, pages 128–142. Springer-Verlag, September 2006.

[38] Johan Bengtsson, W. O. David Griffioen, Kre 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, July-August 2002.

[39] P. A. Abdulla, P. Krcal, and W. Yi. Sampled Universality of Timed Automata. In *10th International Conference Foundations of Software Science and Computational Structures, FOSSACS 2007, part of ETAPS 2007*, volume LNCS 4423, pages 2–16. Springer-Verlag, 2007.

[40] Dragan Bošnački and Dennis Dams. Discrete-Time Promela and Spin. In *FTRTFT '98: Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 307–310. Springer-Verlag, 1998.

[41] Mary M. Shaw. The Coming-of-Age of Software Architecture Research. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001, Toronto, Ontario, Canada*, pages 656–664a. IEEE Computer Society.

[42] TIMMO consortium. `http://www.timmo.org`, accessed August 2009.

[43] 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. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 28–38, 2009.

[44] aiT - Worst-Case Execution Time Analyzer. `http://www.absint.com/ait/`, accessed August 2009.

[45] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst. System Level Performance Analysis - the SymTA/S Approach. In *Proceedings of Computers and Digital Techniques*, 2005.

[46] Xu Ke, Paul Pettersson, Krzysztof Sierszecki, and Christo Angelov. Verification of COMDES-II Systems Using UPPAAL with Model Transformation. In *RTCSA '08: Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 153–160, Washington, DC, USA, 2008. IEEE Computer Society.

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

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

[49] 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.

[50] The Object Management Group. UML Version 2.1.2. http://www.omg.org/spec/UML/2.1.2/, accessed August 2009.

[51] Object Management Group. A UML Profile for MARTE, Beta 1, August 2007. Document number: ptc/07-08-04.

II

Included Papers

Chapter 7

Paper A: A Classification Framework for Component Models

Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, and Michel Chaudron
Accepted to IEEE Transactions on Software Engineering (in the process of revision)

Abstract

The essence of component-based software engineering is embodied in component models. Component models specify the properties of components and the mechanism of component compositions. In last decade a rapid growth, a plethora of different component models has been developed, using different technologies, having different aims, and using different principles. This has resulted in a number of models and technologies which have many similarities, but also principal differences, and in a lot cases unclear concepts. Component-based development has not succeeded in providing standard principles, as for example object-oriented development. In order to increase the understanding of the concepts, and to easier differentiate component models, this paper provides a Component Model Classification Framework which identifies and discusses the basic principles of component models. Further the paper classifies a certain number of component models using this framework.

7.1 Introduction

Component-based software engineering (CBSE) is an established area of software engineering. The inspiration for “building systems from components” in CBSE comes from other engineering disciplines, such as mechanical or electrical engineering, software architecture. The techniques and technologies that form the basis for component models originate mostly from object-oriented design and Architecture Definition Languages (ADLs). Since software is in its nature different from the physical world, the translation of principles from the classical engineering disciplines into software is not trivial. For example, the understanding of the term component has never been a problem in the classical engineering disciplines, since a component can be intuitively understood and this understanding fits well with fundamental theories and technologies. This is not the case with software. The notation of a software component is not clear: its intuitive perception may be quite different from its model and its implementation. From the beginning, CBSE struggled with a problem to obtain a common and a sufficiently precise definition of a software component. An early and probably most commonly used definition coming from Szyperski [1] (“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 party”) focuses on characterization of software component. In spite of its generality it was shown that this definition is not valid for a wide range of component-based technologies (for example those which do not support contractually specified interface or independent deployment). In the definition of Heineman and Councill [2] (“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”), the component definition is more general actually a component is specified through the specification of the component model. The component model itself is not specified. This definition can be even more generalized in respect to the component specification, but component model can be expressed more precisely [3]:

Definition: *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.*

This generic definition allows the existence of a wide spectrum of component models, which is also happening in reality; on the market and in different research communities, there exists many component models with different characteristics. However, it makes it more difficult to properly understand the Component-Based (CB) principles. In particular, this is true since CB principles are not clearly explained and formally defined. In their diversities component models are similar to ADLs; there are similar mechanisms and principles but many variations and different implementations. For this reason there is a need for having a framework which can provide a classification and comparison between different component models in a similar manner as it was done for ADLs [4, 5]. In addition, a framework can help in the selection of a particular component model or in the design of a new component model.

In this paper, we propose a classification and comparison framework for component models. Since component models and their implementations in component technologies cover a large range of different aspects of the development process, we group these aspects in several dimensions and build a multi-dimensional framework that counts different, yet equality important, aspects of component models. We have also analyzed a considerable number of component models, and compared their characteristics. The results of the comparison have led to some observations which are discussed in the paper.

Our research methodology was based on several iterations of (i) observations and analysis, (ii) classification, and (iii) validation; in the first iteration, based on the literature related to general principles of component- based software engineering and existing classification [1–11], the classification model was applied to a set of component models, and discussed with several CBSE and empirical software engineering researchers and experts from different engineering domains. The resulting analysis and discussions have led to a refinement of the framework. In the next iterations the refined framework was applied to new component models and discussed with new researchers. The process (which lasted more than one year) has been completed when in the last iteration all new component models complied well with the framework. Another important issue that we learned was related to a decision what to define as a component model and what not. This is discussed in section three.

The remainder of this paper is organized as follows. Section 7.2 motivates, explains and defines the different dimensions of the classification framework. Section 7.3 discusses the criteria for inclusion of different models/technologies into to component models survey and the classification framework. The comparison framework and observations from the comparison are presented in section 7.4. Related work is covered in section 7.5 and section 7.6 concludes the paper. A very brief overview of the selected component models on which the classification framework has been mapped is given in appendix 7.7.

7.2 The classification framework

The main concern of a component model is to (i) provide rules for the specification of component properties and (ii) provide rules and mechanisms for component composition, including the composition rules of component properties. These main principles hide many complex mechanisms and models, and have significant differences in approaches, concerns and implementations. For this reason we cannot simply list all possible characteristics to compare the component models; rather we want to group particular characteristics that have similar concerns i.e. that describe the same or related aspects of component models. Starting from the definition of component models, we distinguish specification of components from specification of communication. Component specifications express component functions (typically in a form of signatures), and extra-functional properties. Most of the component models include only specification of functions, in form of interfaces. Extra-functional properties, if specified at all, are defined either in a form of extended interface or as component metadata. The functional part of an interface is directly related to interaction between components and realized through construction mechanisms using different interaction (architectural) styles. Communication between components is usually not explicitly specified, but there are different types of communications that are assumed in component models.

Finally different component models cover different phases in a component lifecycle; while some support only the modeling phase, others also provide mechanisms supporting the implementation and run-time phase.

In this paper we divide the fundamental principles and characteristics of component models into the following dimensions.

D.1 **Lifecycle.** *The lifecycle dimension identifies the support provided (explicitly or implicitly) by the component model, in certain points of a lifecycle of components or component-based systems.* Component-Based Development (CBD) is characterized by the separation of the development processes of individual components from the process of system development. There are some synchronization points in which a component is integrated into a system, i.e. in which the component is being bound. Beyond those points, the notion of components in the system may disappear, or components can still be recognized as parts of the system.

D.2 **Constructs.** *The constructs dimension identifies (i) the component interface used for the interaction with other components and external envi-*

ronment, and (ii) the means of component binding and communication. In some component models, the interface comprises the specification of all component properties, including both functional and extra-functional, but in most cases, it only includes a specification of functional properties. Directly correlated to the interface are the components interoperability mechanisms. All these concepts are parts of the "construction" dimension of CBD.

D.3 **Extra-Functional Properties.** *The extra-functional properties dimension identifies specifications and support that includes the provision of property values and means for their composition.* In certain domains (for example real-time embedded systems), the ability to model and verify particular properties is equally important but more challenging than the implementation of functional properties.

D.4 **Domains.** *This dimension shows in which application and business domains component models are used or supposed to be used.* It indicates the specialization, or the generality of component models.

In these four dimensions, we comprise the main characteristics of component models but, of course, there are also other characteristics that can differentiate them. For example, since in many cases component models are built on a particular implementation technology, many characteristics come directly from this supporting implementation technology and are not visible in component models themselves. Still the intention with the classification and comparison model is to comprise the main characteristics of component models.

7.2.1 Lifecycle

While CBSE aims at covering the entire lifecycle of component-based systems, component models provide only partial lifecycle support and usually are related to the design, implementation and integration phases.

The overall component-based lifecycle is separated into several processes; building components, building systems from components, and assessing components [6]. Some component technologies provide certain support in these processes (for example maintaining component repositories, exposing interface, component deployment).

The component-based paradigm has extended the integration activities up to the run-time phase; certain component technologies provide extended support for dynamic and independent deployment of components into running systems. This support is reflected in the design of many component models. In contrast, in other component models components only exist as separate units in the development stage and become assimilated into a system when the system is built. In this case the system at run-time is monolithic. However not all component models consider this integration phase. We can clearly distinguish different component models that focus on one particular or more phases and such phases can be different for different component models. Some component technologies start in the design phase (e.g. Koala which has an explicit and dedicated design notation of components and other elements of the component model), while other component technologies focus on the implementation phase (e.g. COM, EJB). For this reason one important dimension of our component model classification lifecycle support. In our classification, we distinguish the lifecycle of components from the lifecycle of the component-based system, which are different [3, 7] and are not necessary temporally related they are ongoing in parallel and have some synchronization points.

Component lifecycle stages

We identify the following stages of the component lifecycle.

L.1 *Modelling stage.* The component models provide support for the modelling and the design of component- based systems and components. Models are used either for the architectural description of the systems and components (e.g. ADLs), or for the specification and the verification of particular system and component properties (e.g. statecharts, resource usage models, performance models).

L.2 *Implementation stage.* The component model provides support for production of code. The implementation may stop with the provision of the source code, or may continue up to the generation of a binary (executable) code. The existence of executable code is a precondition for the dynamic deployment of components (during run-time).

L.3 *Packaging stage.* Because components are the central unit in CBSE, there is a need for their storage and packaging either in a repository or for distribution. A component package is a set of metadata and code (source or executable). Accordingly, the result of this stage can be a file, an archive, or a repository in which the packaged components reside prior to decisions about how they will be run in the target environment. For example, in Koala, components are packed into a file system-based

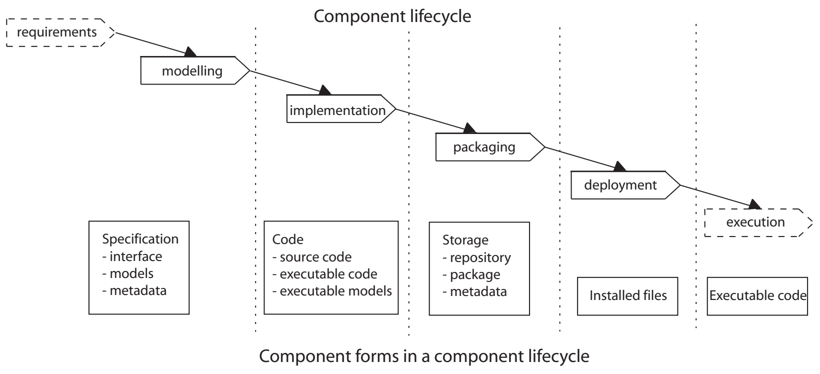


Figure 7.1: Component lifecycle and component forms

repository, with a folder per component. The folder includes a number of files: Component Description Language (CDL) file and, a set of C and header files, test file and different documents. Another example of packaging is achieved in the EJB component model. There, packaging is done through jar archives, called ejb-jar. Each archive contains XML deployment descriptor, component description, component implementation and interfaces.

L.4 *Deployment stage.* At a certain point of time, a component is integrated into a system. This activity may happen at different phases of the systems lifecycle. In general, the components can be deployed at:

(a) *compilation time*, so it is no longer possible to change the way the components interact with each other. For instance, Koala components are deployed at compilation time and they use static binding by following naming conventions and generated renaming macros.

(b) *run time* as separate units by using means such as registers (COM) or containers (CCM,EJB). For example, CORBA components are deployed at run time in a container by using information of the deployment descriptor packed with the component implementation.

Figure 7.1 illustrates different stages in a component lifecycle and the associated forms of the components. Through the stages some of the forms are transformed into new ones, some remains, while some disappear. In the figure the requirements and execution phase are denoted with the dashed lines which indicate that in these stages components do not necessary exist as independent units. The forms of the components will be different across phases for different component models.

7.2.2 The constructs

As defined in [12], the verb “construct” means “to form something by putting different things together”, so in applying this definition to the CBSE domain, we define by the Constructs dimension, the way components are connected together within a component model in order to provide communication means. But although this communication aspect is of primordial importance, it is not often expressed explicitly. Instead, it is reflected implicitly by some underlying mechanisms. This should be distinguished from specifications of functional and sometimes extra-functional properties in a form of component interfaces. Consequently, a component interface has a double role: It first specifies the component properties (functional and possibly extra-functional), and second, it identifies the connection points through which components are interconnected.

Interface

Interface specification is the characteristic “sine qua non” of a component model. Interfaces are defined either by using special languages, or elements of programming languages. Several languages exist that specify components interfaces and their connections: modeling languages, such as UML or different Architecture Description Languages (ADLs), particular specification languages, such as Interface Definition Languages (IDLs), programming languages such as Interface in Java, or abstraction class in C++, or some additions built directly in a programming language, such as pre-defined structs in C. In case of special languages, the interface specifications are translated to a programming language. In a few cases (e.g. COM), the interface is also defined in a binary format in order to have a standard representation at deployment and run-time. Some mechanisms such as introspection in Java are also used to discover the interfaces of a component at run-time.

The component models that use programming languages or their extensions for component specification, also inherit properties of these languages. For example the component models that use object-oriented languages utilize the concepts of classes and (interface) inheritance. Typically a component is expressed as a class in which the interface is defined as a set of operations/functions and attributes. However there exist other types of interfaces so called port-based where ports are entries for receiving/sending different data types and events. Note that this concept is different from the concept in UML 2.0 [13] in which a port is defined as a set of specifications.

Some component models distinguish also the "provides"-part (i.e. the specification of the functions that the component offers) from the "requires"-part (i.e. the specification of the functions the component requires) of an interface.

In order to ensure that a component will behave as expected according to its specification and operational mode, and in order to ensure that a component is supplied with expected input and environment the notion of contract has been adjoined to interfaces. According to [8], contracts can be classified hierarchically in four levels which, if taken together, may form a global contract. We only adopt the three first levels in our classification since the last level "contractualizes" only the extra-functional properties and this is not in direct relation with interoperability

- *Syntactic level*: describes the syntactic aspect, also called signature, of an interface. This level ensures the correct utilisation of a component. That is to say that the "calling-component" must refer to the proper types, fields, methods, signals, ports and handles the exceptions raised by the "respondingcomponent". This is the most common and most easy agreement to certify as it relies mainly on an, either static or dynamic, type checking technique.

- *Semantic level*: reinforces the previous level of contracts in certifying that the values of the parameters as well as the persistent state variables are within the proper range. This can be asserted by pre-conditions, post-conditions and invariants. A generalization of this level can be assumed as semantics.

- *Behaviour level*: dynamic behaviour of services. It expresses either the composition constraints (e.g., constraints on their temporal ordering) or the internal behaviour (e.g. dynamic of internal states).

Finally, the constructs dimension refers to the notions of reusability and evolvability, which are important principles of CBSE. Indeed many component models are endowed with diverse features for supporting them; one typical solution is the ability to add new interfaces to a component. This makes it possible to embody several versions or variants of functions in the component.

Composition of constructs

While compositions in general consider compositions of component properties, both functional and extra-functional, compositions of constructs are related to components interactions. Constructs compositions are implemented as connections of interaction channels and the process of this connection is called binding. The binding mechanism is related to the component lifecycle; it can occur at compilation time (when a compiler provides connections between components using programming language mechanisms), or at runtime, in which connection mechanisms are utilised that are provided by the underlying run-time infrastructure. Such a run-time infrastructure may consist of dedicated component middleware, and/or a component framework or of a common operating system or middleware.

A so-called "docking interface" method is commonly used when binding occurs at run-time. This docking interface does not offer any application functionality, but serves instead for managing the binding and subsequent interaction between a component and the underlying run-time infrastructure. In many component models (e.g. CCM, EJB) the composition specification is location-transparent; the run-time location of components (placed on a local or a remote node) is specified separately from the binding information. This information about the location is used in the deployment phase.

Connectors, introduced as distinct elements in ADLs, are not common among the first class citizens in most component models. Connectors are mediators in the connections between components and have a double purpose: (i) enabling indirect composition (so called exogenous compositions), and (ii) introducing additional functionality, especially for mediation between components. In the exogenous composition information concerning the binding resides outside of the components; the components have no knowledge of who they are connected to. Exogenous composition enables more seamless evolution because it separates changes to components from changes to their bindings. In several component technologies, connectors are implemented as special types of components, such as adaptors or proxies, either to provide additional functional or extra-functional properties, or to extend the means of intercommunication. In direct (endogenous) type of composition the components are connected directly through their interfaces. Information concerning the binding resides inside components.

The interface specification implicitly defines the type of interaction between components to comply with particular architectural styles. In most cases, a particular component models provide a single basic interaction style (for example, “request-response” or “pipe & filter”, but others, such as Fractal, Pin and BIP allow the construction of different architectural styles.

An important question related to the composability of components has concerned the research community [9]: Can the assemblies of components (by assemblies we assume a set of components mutually connected) be treated as components themselves, i.e. is he composition hierarchical? There are two kinds of assemblies supported by existing component technologies. The first is the first order assembly which is not treated as a component in the component model. This type of assembly is merely a set of components of an arbitrary form, creating an application or a part of an application. In terms of binding the component models refer to “horizontal composition” or “horizontal binding”. The second type of assembly is hierarchical which means that the assembly, created from components, again satisfies the properties that an individual component should satisfy according to the component model. In that case we refer to “hierarchical composition” or “hierarchical binding”. The criteria for vertical composition are related to constructs (interface specification and the interaction), and possibility extra-functional properties. Most of the component models support partial vertical composition. For example interfaces can be composed recursively in modeling phase, but not in the deployment phase (in particular when deployment is performed during run-time).

Constructs classifications

Following the observations and reasoning from above we identify the following classification characteristics for interfaces and connections in the constructs dimension.

C.1 *Interface specification*, in which different characteristics allowing the specification of interfaces are identified:

(a) The distinction of interface type: operation-based (e.g. methods invocations) and port-based interface (e.g. data passing).

(b) The distinction between the provides-part and the requires-part of an interface.

(c) The existence of some distinctive features appearing only in this component model (such as special type of ports, optional operations).

(d) The language used to specify the interface.

   (e) Interface levels which describes the levels of contractualisation of the interfaces, namely syntactic, semantic and/or behaviour level.

C.2 *Interactions*, which comprise the following characteristics:

   (a) Interaction style which describes the main underlying architectural style used.
   (b) Communication type which details mainly if the communication used are synchronous and/or asynchronous.
   (c) Binding type describes the way components may be linked together through the interfaces. It is realized in two subtypes:
      i. The exogenous/endogenous sub-category describing whether the component model includes connectors as architectural elements, and
      ii. The hierarchical sub-category expressing the possibility of having a hierarchical composition of components (horizontal composition is an intrinsic part of all component models, thus it is implicitly assumed, and not put in the classification framework).

7.2.3 Extra-Functional Properties

Properties are used in the most general sense as defined by standard dictionaries, e.g.: "a construct whereby objects and individuals can be distinguished" [9]. There is no unique taxonomy of properties, and consequently many property classification frameworks can exist. One commonly used classification is to distinguish functional from extra-functional properties. While functional properties describe functions or services of an object, extra-functional properties (EFPs) specify the quality, or in general a characteristic of interest, of objects. In CBSE, there is also a distinction between component properties and system properties. A property at the system level can result from the composition of the same properties of constituent components, but also from the composition of different properties. In latter case such property can exist only on a system level. Such properties are called emerging properties.

Composition of extra-functional properties

EFPs can be complex and abstract or, they can be tangible and concrete. Examples of abstract (and complex) properties are dependability or performance and examples of tangible properties are memory footprint or scalability. Complex properties are typically the result of the composition of several more tangible properties. An important concern of CBSE is composition of properties expressed in the following way. For an assembly A that is composed of component C1 and C2

$$A = C1 \circ C2$$

expresses a property of the assembly as a composition of properties of the components

$$P(A) = P(C1) \circ P(C2)$$

are specified in very different ways. Also computing the compositions of EFPs require different composition theories for different EFPs. In relation to composability, one of the challenges of CBSE is predictability. To enable analysis at the design stage and to avoid expensive, tedious and non-accurate tests and increase reusability, a lot of efforts has been made in CBSE research communities to design component models that enable predictability. According to [9], the properties can be classified according to types of compositions in the following basic categories.

- *Directly composable properties* (example: static memory): A property of an assembly is a function of, and only of, the same property of the components involved.

$$P(A) = f(P(C1), \ldots, P(Ci), \ldots, P(Cn))$$

- *Architecture-related properties* (example: performance): A property of an assembly is a function of the same property of the components and of the software architecture.

$$\begin{aligned} P(A) &= f(SA, \ldots P(Ci) \ldots), \\ i &= 1 \ldots n \\ SA &= softwarearchitecture \end{aligned}$$

- *Derived properties* (example: response time vs. execution time): A property of an assembly depends on several different properties of the components.

$$\begin{aligned} P(A) &= f(SA, \ldots Pi(Cj) \ldots), \\ i &= 1 \ldots m \\ j &= 1 \ldots n \\ Pi &= componentproperties \\ Cj &= components \end{aligned}$$

- *Usage-depended properties* (example: reliability): A property of an assembly is determined by its usage profile.

$$\begin{aligned} P(A, U) &= f(SA, \ldots Pi(Cj, U) \ldots), \\ i &= 1 \ldots m \\ j &= 1 \ldots n \\ U &= usageprofile \end{aligned}$$

- *System environment context properties* (example: safety): A property is determined by other properties and by the state of the system environment.

$$\begin{aligned} P(S, U, X) &= f(SA, \ldots Pi(Cj, U, X) \ldots), \\ i &= 1 \ldots m \\ j &= 1 \ldots n \\ S &= system \\ X &= systemcontext \end{aligned}$$

This idealised classification indicates the limitations of the compositions of EFPs. Determining the compositions of properties of components becomes feasible when restrictions are imposed on the design of individual components (by means of rules/constraints in of the component model) and system architecture. For example static memory usage of an assembly can be defined as the sum of static memory usage of involved components, but only using particular composition policies (e.g. no concurrency). In this way, we can obtain predictability of the considered property. Other properties are related to usage profile and if we cannot predict usage profile we cannot predict the system properties. Some other properties are not composable at all, and in that case we cannot predict their composition.

Management of extra-functional properties

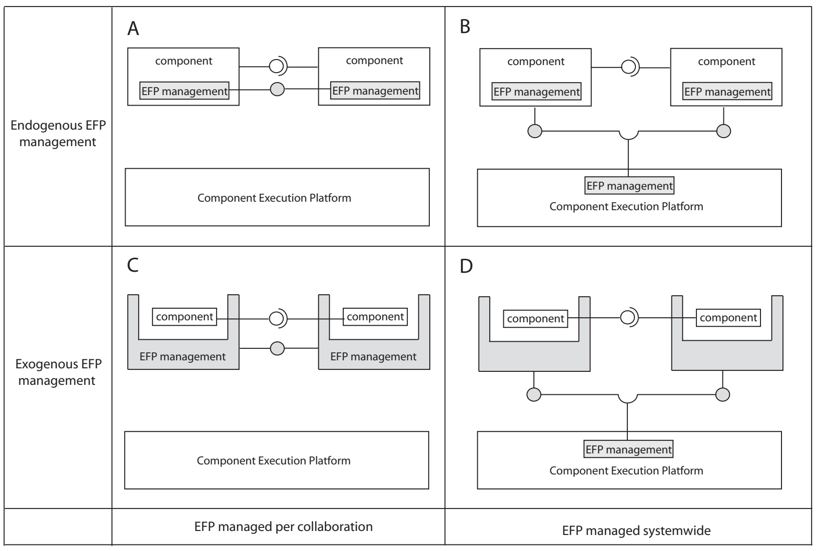


Figure 7.2: Management of extra-functional properties

Even if EFPs are not composable, they can be manageable, i.e. they can be obtained by using some solutions encapsulated in component models and standardized architectural solutions. Different types of EFP management exist according to the way the component models handle them. We distinguish two main dimensions Fig. 7.2:

1. A property is managed by the components (endogenous EFP management – approaches A and B), or by the system (exogenous EFP management – approaches C and D) or managed.

2. A property is managed on a system-wide scale (approaches B and D), or the property is managed on a per-collaboration basis (approaches A and C).

Approach A (*endogenous per collaboration*). A component model does not provide any support for EFP management, but it is expected that a component developer implements it. This approach makes it possible to include EFP management policies that are optimized towards a specific system, and also can cater for adopting multiple policies in one system. This heterogeneity may be particularly useful when COTS components need to be integrated. On the other hand, the fact that such policies are not standardized may be a source of architectural mismatch between components. This approach can hardly manage emerging properties.

Approach B (*endogenous systemwide*). In this approach, there is a mechanism in the component execution platform that contains policies for managing EFPs for individual components as well as for EFPs involving multiple components. The ability to negotiate the manner in which EFPs are handled requires that the components themselves have some knowledge about how the EFPs affect their functioning. This is a form of reflection.

Approach C (*exogenous per collaboration*) and *Approach D* (*exogenous systemwide*). In these approaches the components are designed such that they address only functional aspects and not EFP. Consequently, in the execution environment, these components are surrounded by a container. This container contains the knowledge on how to manage EFPs. Containers can either be connected to containers of other components (approach C) or containers can interact with a mechanism in the component execution platform that manages EFPs on a system wide scale (approach D). The container approach is a way of realizing separation of concerns in which components concentrate on functional aspects and containers concentrate on extra-functional aspects. In this way, components become more generic because no modification is required to integrate them into systems that may employ different policies for EFPs. Since these components do not address EFPs, another advantage is that they are simpler and hence cheaper to implement. A disadvantage of this approach might be a degradation of the system performance.

Extra-functional properties classification

For the EFPs we provide a classification in respect to the following questions:

E.1 *Management of EFPs*: Which type of management (if any) is provided by the component model?

E.2 *EFP specification*: Does the component model contain means for specification and management of specific EFPs. If yes, which properties or which types of properties?

E.3 *Composability of EFPs*: Does the component model provide means, methods and/or techniques for composition of certain extra-functional properties and/or what type of composition?

7.2.4 Domains

Some component models are aimed at specific application domains as for instance consumer electronics or information systems. In such cases, requirements from the application domain penetrate into the component model. The benefits of a domain-specific component models are that the component technology facilitates achieving certain requirements. Such component models are, as a consequence, limited in generality and will not be so easily usable in domains that are subject to different requirements.

Some component models are of general-purpose. They provide basic mechanisms for the specification and the composition of components, but do not assume any specific architecture beyond general assumptions (like interaction style, support for distributed systems, compilation or run-time deployment). A general solution that enables component models to be both generally applicable but to also cater for specific domains is through the use of optional frameworks. A framework is an extension of a component model that may be used, but is not mandatory in general. There is a third type of component models, namely generative; they are used for instantiation of particular component models. They provide common principles, and some common parts of technologies (for example modeling), while other parts are specific (for example different implementations). According to this, we classify the component models as

A.1 General-purpose component models;

A.2 Specialized component models;

A.3 Generative component models.

7.2.5 The classification overview

Fig. 7.3 summarizes the classification framework in a graph form.

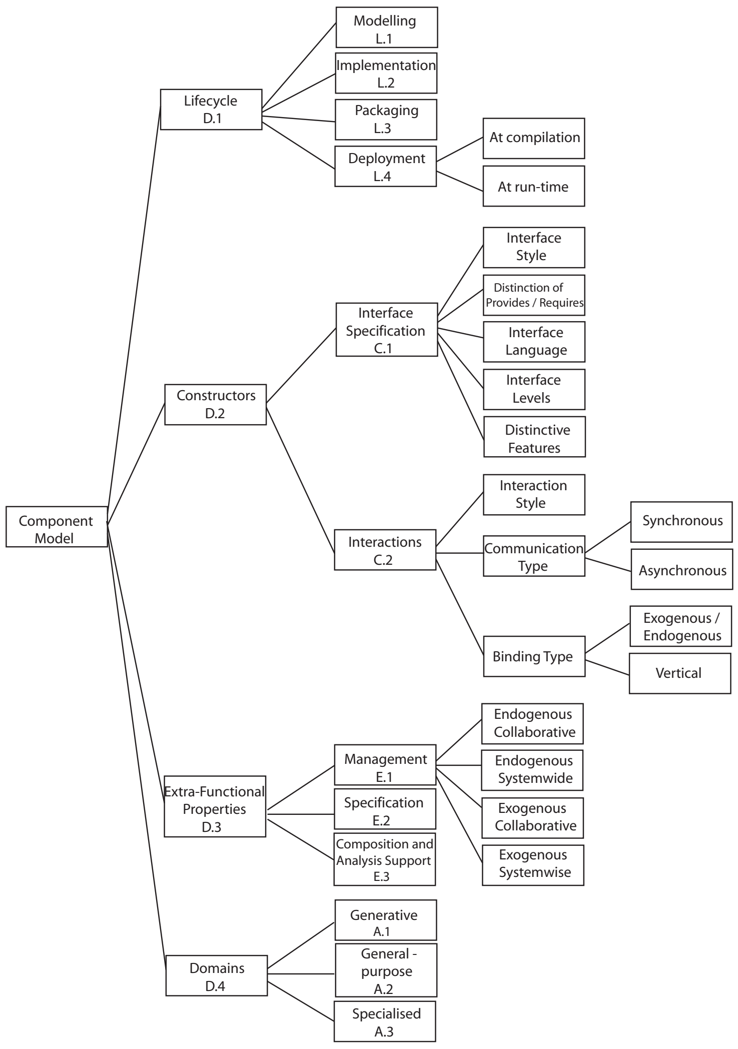


Figure 7.3: The hierarchical structure of the classification framework

7.3 Survey of component models

Nowadays a number of component models exist. They vary widely: in usage, in support provided, in concerns, in complexity, in formal definitions, etc.. In our classification of component models, the first question is whether a particular model (or technology, method, or similar) is a component model or not. Similar to biology in which viruses cover the border between life and non-life, there is a wide range of models, from those having many elements of component models but still not assumed as component models, via those that lack many elements of component models, but still are designated as component models, to those which are broadly accepted component models. Therefore, we identify the minimum criteria required to classify a model, or a notation as a component model. This minimum is defined by the definition of component models given in the introduction: A model that defines rules for the design and specification of components and their properties and means of their composition can be classified as a component model. It should be noted that this condition is mandatory, but not sufficient. We have identified several models that fulfill this condition, but still we have not included them in the survey. We can call them "almost" component models.

7.3.1 "Almost" component models

A wide range of modeling languages contains the term "component" and even (semi)formally specifies components and component compositions. For example in the classification of ADLs [5] one of the basic elements are components (and connectors as means for construction composition). UML 2.0 is even closer to component models since it provides a metamodel for components, interfaces and ports. Still we have deliberately chosen not to select them as component models, in difference to some other classifications (such as [11]). One reason is that their purpose is not component-based development but rather the specification of system architectures. ADLs and UML 2.0 are excellent language candidates for modeling component-based systems and components in the design phase, but are missing other characteristics to be declared as component models. Certain languages derived from UML, such as xUML [14] in which the component specification is translated to an executable entity, are even closer candidates for component models. However xUML and similar languages do not operate with components as first class citizens (for example components are not treated as separate development or executable entities), but components are only architectural elements.

On the other side of the lifecycle line are services. One can argue that services are special types of components. Services are focused on run-time retrieval and run-time deployment. Similar to components, services are specified by an interface, and provide support for constructs compositions [15]. Still we have not included services in the classifications for similar reasons as for ADLs their focus is not component-based development. In analogy to ADLs, services are not component models but rather use component models. Further, we have not included technologies such as Unix processes and "pipe & filter" mechanisms, or modeling environments such as Simulink or Ptolemy [16], as again the components are not the primary concern in these approaches.

Finally we have not included technologies like Eclipse or Photoshop that enable the integration of plugins from third parties and in this way suit well to a part of Szyperskis definition of components ("deployed independently and is subject to composition by third party"). However they do not provide mechanisms of compositions between components, rather mechanism between components and the underlying platform.

For these "almost component models" one can argue that they are component models or technologies, and that they could be included into the survey. Our position is that their inclusion will break the spirit of the component models as defined in this paper according to the arguments presented.

7.3.2 Component models

In our classification framework we have selected a number of component models that appeared in the research literature and in practice. While some of them are widely spread and proven, others are used as demonstrators or illustrations of ideas in research.

The classification framework does not show the success of particular component models, or any business model, but it is based on the technical characteristics only. The components models that we have included in the list are shortly referred to in the appendix 7.7.

It is worth to mention that for some of the component models that we found, our selection criteria were satisfied, however because of scarcity of available documentation it was impossible to get the needed detailed information (which usually is a sign that no activity around the model is going on). In these cases, we have decided to omit them from our list.

7.4 The comparison framework

The characteristics of the component models are collected in the tables below, following the dimensions in the classification framework, namely lifecycle (Table 7.1), constructs (Tables 7.2, and 3), extra-functional properties (Table 7.4), and the domains (Table 7.5) lined in the alphabetic order. Following each table, a short discussion gathering observations and their rationales is presented.

7.4.1 Life-cycle classification

From the observation of Table 7.1, one can notice that there is a group of component models that do not provide any support for modeling of components or component-based applications, but cover only implementation part (specification and deployment). All these component models belong to the state of the practice and most of them are widely used. Does that mean that the modeling of components is not supposed to be a part of a component model? Or does it mean that other tools, for example general-purpose modeling tools, such as UML or ADLs are used for modeling, while component technologies are used for the implementation? It is partially true that most of the practitioners do not model their systems using formal specification languages, but rather express their design in a non-formal way for documentation purpose only, or in a semi-formal way typically using UML. In both cases neither the precise definitions of components nor their interactions are assumed to be of high priority. This is also an indicator of differences between state of the art and state of the practice; many solutions that include modeling of components or their properties from the state of the art have still not been realized or scaled up in practice.

The second observation from Table 7.1 is the fact that most of the component models use object-oriented languages for the implementations with domination of Java. Still there exist component models using other languages, for example imperative programming languages such as C.

It seems that the packaging and component repositories are not in focus of component models. In most cases, certain standard archives are used (such as DLL or JAR packages). The lack of repositories indicates a low focus of reuse, in particular of COTS components.

Deployment at compile time and run-time occurs almost equally often. Deployment at compile time limits the flexibility at run-time, but on the other hand enables easier predictability, richer composition features (such as hierarchical composition), and more efficient reuse (such as deployment of implementation parts that will be used in the application). This might be a reason why this is the primary deployment style chosen by specialized component models (cf. Table 7.5).

7.4.2 Constructs classification

Tables 7.2 and 7.3 show interface and interaction specifications of the selected component models. Although the existence of interface is a "conditio sine qua non" for component models, and all selected component models identify the interface as an indispensable part of a component, Table 7.2 shows that interfaces can be of different types. Most interfaces are of operation type, thus using functions and parameters for defining elements of services the component provides and requires. Still, many component models use ports as interface elements using them for passing data. Such component models are typically used in embedded systems and have their grounds from the concept of hardware components. Some component models do not distinguish between required and provided interface, but the interface is identified with the provided interface, similar to the object-oriented approach. In port-based interfaces, input and output interfaces consisting of ports that receive and send data (often designated as sink and source) are distinguished, which corresponds to provided and required interface.

Since interfaces are an obligatory part of the component specification, all component models provide at least the first level, i.e. syntactic specification. A considerable number of component models also have behavior specifications, in most cases specified by a particular form of finite state machines (state charts, timed automata). Rather few of the component models identify semantic of the interfaces. If semantics are defined, then mostly pre- and post-conditions are used for this. It is worth to mention that interface semantics should not be mixed with other types of semantics that some component models can have (e.g. SaveCCM has execution semantics which defines the process of the component execution in respect to time).

In line with the type of an interface (operation vs. ports), from the information provided in Table 7.3 one can conclude that the dominating interaction styles in the component models are "request response" (typically used in client/server architectures), and dataflow and pipe & filter. Some component models have specific additions to interaction styles – event-driven, broadcast or rendez-vous.

Table 7.1: Lifecycle Dimension

Component Models	Modelling	Implementation	Packaging	Deployment
AUTOSAR	N/A	C	Non-formal specification of container	At compilation
BIP	A 3-layered representation: behavior, interaction, and priority	BIP Language	N/A	At compilation
BlueArX	N/A	C	N/A	At compilation
CCM	N/A	Language independent	Deployment Unit archive (JARs, DLLs)	At run-time
COMDES II	ADL-like language	C	N/A	At compilation
CompoNETS	Behavour modeling (Petri Nets)	Language independent	Deployment Unit archive (JARs, DLLs)	At run-time
EJB	N/A	Java	EJB-Jar files	At run-time
Fractal	ADL-like language (Fractal ADL, Fractal IDL), Annotations (Fractlet)	Java (in Julia, Aokell) C/C++ (in Think) .Net lang. (in FracNet)	File system based repository	At run-time
KOALA	ADL-like languages (IDL,CDL and DDL)	C	File system based repository	At compilation
KobrA	UML Profile	Language independent	N/A	N/A
IEC 61131	Function Block Diagram (FBD) Ladder Diagram (LD) Sequential Function Chart (SFC)	Structured Text (ST) Instruction List (IL)	N/A	At compilation
IEC 61499	Function Block Diagram (FBD)	Language independent	N/A	At compilation
JavaBeans	N/A	Java	Jar packages	At compilation
MS COM	N/A	OO languages	DLL	At compilation and at run-time
OpenCOM	N/A	OO languages	DLL	At run-time
OSGi	N/A	Java	Jar-files (bundles)	At compilation and at run-time
Palladio	UML profile	Java	N/A	At run-time
PECOS	ADL-like language (CoCo)	C++ and Java	Jar packages or DLL	At compilation
Pin	ADL-like language (CCL)	C	DLL	At compilation
ProCom	ADL-like language, timed automata	C	File system based repository	At compilation
ROBOCOP	ADL-like language, resource management model	C and C++	Structures in zip files	At compilation and at run-time
RUBUS	Rubus Design Language	C	File system based repository	At compilation
SaveCCM	ADL-like (SaveComp), timed automata	C	File system based repository	At compilation
SOFA 2.0	Meta-model based specification language	Java	Repository	At run-time

Table 7.2: Constructs – Interface Specification

Component Models	Interface type	Distinction of Provides/ Requires	Distinctive features	Interface Language	Interface Levels (Syntactic, Semantic, Bahaviour)
AUTOSAR	Operation-based Port-based	Yes	AUTOSAR Interface	C header files	Syntactic
BIP	Port-based	No	Complete interfaces, Incomplete interfaces	BIP Language	Syntactic Semantic Behaviour
BlueArX	Port-based	Yes	N/A	C	Syntactic
CCM	Operation-based Port-based	Yes	Facets and receptacles Event sinks and event sources	CORBA IDL, CIDL	Syntactic
COMDES II	Port-based	Yes	N/A	C header files State charts diagrams	Syntactic Behaviour
CompoNETS	Operation-based Port-based	Yes	Facets and receptacles Event sinks and event sources	CORBA IDL, CIDL, Petri nets	Syntactic Behaviour
EJB	Operation-based	No	N/A	Java Programming Language + Annotations	Syntactic
Fractal	Operation-based	Yes	Component Interface, Control Interface	IDL, Fractal ADL, or Java or C, Behavioural Protocol	Syntactic Behaviour
KOALA	Operation-based	Yes	Diversity Interface, Optional Interface	IDL, CDL	Syntactic
KobrA	Operation-based	N/A	N/A	UML	Syntactic
IEC 61131	Port-based	Yes	N/A	N/A	Syntactic
IEC 61499	Port-based	Yes	Event input and event output Data input and data output	N/A	Syntactic
JavaBeans	Operation-based	Yes	N/A	Java	Syntactic
MS COM	Operation-based	No	Ability to extend interface	Microsoft IDL	Syntactic
OpenCom	Operation-based	No	Interfaces additional to COM-interface managing lifecycle, introspections, etc.	Microsoft IDL	Syntactic
OSGI	Operation-based	Yes	Dynamic Interfaces	Java	Syntactic
Palladio	Operation-based	Yes	Possibility to annotate interface	UML	Syntactic Behaviour
PECOS	Port-based	Yes	Ability to extend interface	Coco language, Prolog query, Petri nets	Syntactic Semantic Behaviour
Pin	Port-based	Yes	N/A	Component Composition Language (CCL), UML statechart	Syntactic Behaviour
ProCom	Port-based	Yes	Data and trigger ports	XML based, Timed Automata	Syntactic Behaviour
Robocop	Port-based	Yes	Ability to extend different types of interface/annotations	Robocop IDL (RIDL), Protocol specification	Syntactic Behaviour
RUBUS	Port-based	Yes	Data and trigger ports	C header files	Syntactic
SaveCCM	Port-based	Yes	Data, trigger, and data-trigger ports	SaveComp (XMLbased),Timed Automata	Syntactic Behaviour
Sofa 2.0	Operation-based	Yes	Utility Interface, Possibility to annotate interface and to control evolution	Java, SPC algebra	Syntactic Behaviour

Table 7.3: Constructs – Interface Interaction

Component Models	Interaction Styles	Communication Type	Binding Type	
			Exogenous	Hierarchical
AUTOSAR	Request response, Messages passing	Synchronous, Asynchronous	No	Delegation
BIP	Triggering, Rendez-vous, Broadcast	Synchronous, Asynchronous	No	Delegation
BlueArX	Pipe&filter	Synchronous	No	Delegation
CCM	Request response, Triggering	Synchronous, Asynchronous	No	No
COMDES II	Pipe&filter	Synchronous	No	No
CompoNETS	Request response	Synchronous, Asynchronous	No	No
EJB	Request response	Synchronous, Asynchronous	No	No
Fractal	Multiple interaction styles	Synchronous, Asynchronous	Yes	Delegation, Aggregation
KOALA	Request response	Synchronous	No	Delegation, Aggregation
KobrA	Request response	Synchronous	No	Delegation, Aggregation
IEC 61131	Pipe&filter	Synchronous	No	Delegation
IEC 61499	Event-driven, Pipe&filter	Synchronous	No	Delegation
JavaBeans	Request response, Triggering	Synchronous	No	No
MS COM	Request response	Synchronous	No	Delegation, Aggregation
OpenCOM	Request response	Synchronous	No	Delegation, Aggregation
OSGi	Request response, Triggering	Synchronous	No	No
Palladio	Request response	Synchronous	No	No
PECOS	Pipe&filter	Synchronous	No	Delegation
Pin	Request response, Message passing, Triggering	Synchronous, Asynchronous	No	No
ProCom	Pipe&filter, Message passing	Synchronous, Asynchronous	Yes	Delegation
Robocop	Request response	Synchronous, Asynchronous	No	No
Rubus	Pipe&filter	Synchronous	No	No
SaveCCM	Pipe&filter	Synchronous	No	Delegation, Aggregation
SOFA 2.0	Multiple interaction styles	Synchronous, Asynchronous	Yes	Delegation

Table 7.3 shows that the dominant communication type in component models is synchronous. Component models that provide support for asynchronous type of communication also support synchronous communication. This indicates that component models are not concerned about architecture (architectural design), but rather targeting detailed design. This fact is also reflected in the use of connectors. Quite a few of the component models have connectors as first class entities, which indicates that components in many component models are implicitly assumed as fine-grained entities, in contrast to architectural components.

Finally, one can observe that many component models do not support vertical binding, i.e. the means for hierarchical composition. Composition of vertical binding is implemented either through delegated interfaces (i.e. selected interfaces from sub-components build up the interface of the composite components) or as aggregation in which the composite component (or in this case just an assembly) include all interfaces of the aggregated components.

7.4.3 Extra-functional properties classification

From Table 7.4 an interesting observation can be found: Many components provide certain support for management of EFPs, either system-wide or per container. However a significantly smaller number of component models have formalisms for EFPs specifications. Even smaller number provides means for composition of EFPs. This is particularly true for commercial component models. This is not surprising since many EFPs are either not formally defined, or are considered too complex.

Some of the component models provide architectural solutions (for example redundancy or authentication) which in general improve the quality of systems. These solutions have an impact on different properties (for example reliability and availability). The solutions are usually not part of components themselves but are built into the underlying platform, and added as additional service used in some particular domains (for example COM+ used in MS COM and .NET technologies). While these component models provide support for increasing quality, they still do not support EFP compositions and by this do not obtain “predictability by construction”. Clearly, composition of EFPs still belongs to research challenges. A vast majority of EFPs that are explicitly managed (specified and composed) belong to resource usage and timing properties.

Table 7.4: Extra-Functional Properties

Component Models	Management of EFP	Properties specification	Composition and analysis support
AUTOSAR	Endogenous per collaboration (A)	N/A	N/A
BIP	Endogenous system wide (B)	Timing properties	Behaviour compositions
BlueArX	Endogenous per collaboration (A)	Resource usage, Timing properties	N/A
CCM	Exogenous system wide (D)	N/A	N/A
COMDES II	Endogenous system wide (B)	Timing properties	N/A
CompoNETS	Endogenous per collaboration (A)	N/A	N/A
EJB	Exogenous system wide (D)	N/A	N/A
Fractal	Exogenous per collaboration (C)	Ability to add properties (by adding property controllers)	N/A
KOALA	Endogenous system wide (B)	Resource usage	Compile time checks of resources
KobrA	Endogenous per collaboration (A)	N/A	N/A
IEC 61131	Endogenous per collaboration (A)	N/A	N/A
IEC 61499	Endogenous per collaboration (A)	N/A	N/A
JavaBeans	Endogenous per collaboration (A)	N/A	N/A
MS COM	Endogenous per collaboration (A)	N/A	N/A
OpenCOM	Endogenous per collaboration (A)	N/A	N/A
OSGi	Endogenous per collaboration (A)	N/A	N/A
Palladio	Endogenous system wide (B)	Performance properties specification	Performance properties
PECOS	Endogenous system wide (B)	Timing properties, generic specification of other properties	N/A
Pin	Exogenous system wide (D)	Analytic interface, timing properties	Different EFP composition theories, example latency
ProCom	Endogenous system wide (B)	Timing and resources	Timing and resources at design and compile time
ROBOCOP	Endogenous system wide (B)	Memory consumption, Timing properties, reliability, ability to add other properties	Memory consumption and timing properties at deployment
RUBUS	Endogenous system wide (B)	Timing	Timing properties at design time
SaveCCM	Endogenous system wide (B)	Timing properties, generic specification of other propertie	Timing properties at design time
SOFA 2.0	Endogenous system wide (B)	Behavioural (protocols)	Composition at design

7.4.4 Domains classification

From Table 7.5 we see that the distribution between general-purpose component models and specialized component models is equal. We could expect more specialized; Probably in practice there are more specialized proprietary and not published component models. We have also observed a migration of certain component models. For example OSGI was originally designed for embedded systems, but later has been used as general-purpose component model in different domains. There is also an opposite trend to this. General-purpose component models have been adapted for particular domains by a combination of addition of new features and restriction of some functions. Such examples are CompoNETS and OpenCOM.

Specialized component models belong to two domains: a) embedded systems, and b) information systems. The component models from the embedded systems domain have some common characteristics: the use of the "Pipe & Filterdataflow" architectural style, components are usually deployable at compilation time, components are resource-aware and often there is support for management of timing properties. These component models are significantly different from general-purpose component models. The component models from the information systems domains are significantly more similar to general-purpose component models. Typically they have similar characteristics as general-purpose component models, such as use of "request response" interaction, support for run-time run-time deployment, expandable interface, implementation in object-oriented language but they can be distinguished from general purpose component models through specific support for distributed components, data transaction support, interoperability with databases, and some architectural solutions such as redundancy or location transparency.

Table 7.5: Domains

Domain	AUTOSAR	BIP	BlueArX	CCM	COMDES II	CompoNETS	EJB 3.0	Fractal	KOALA	KobrA	IEC 61131	IEC 61499	JavaBeans	MS COM	OpenCOM	OSGi	Palladio	PECOS	Pin	ProCom	Robocop	Rubus	SaveCCM	SOFA 2.0
General-purpose				X		X	X	X		X			X	X	X		X		X					X
Specialised	X	X	X		X				X		X	X				X		X		X	X	X	X	
Generative								X													X			X

7.5 Related work

Over the last decade, several attempts to identify key features of software components and component models have been proposed: classification or studies of components and interfaces ([17], [18]), interfaces, extra-functional properties ([9]), ADLs ([5]), component models ([11]), characteristics of component models for particular business domains ([10]), among others.

The models presented in [17] and [18] do not consider any component model but rather focus on practical issues of component utilization and reutilization. In [17], the interface classification is split into two categories: application interfaces and platform interfaces. Application interfaces describe the information about the interaction with other components (messages protocol, timing issues to requests) whereas the platform aspect is concentrates on the interaction between components and the executing platform. Similarly in [18] a model for characterizing components is proposed which reuses the classification model of interfaces from [17]. A component is there regarded as the description of three main items (informal description, externals and internals) each of them split into several subelements. The informal description is connected with a set of human-related features which can influence on the selection of a component such as its age, its provenance, its level of reuse, its context, its intent and if there is any related component solving a similar problem. The externals are concerned with interaction mechanisms both with other application artifacts and with the platform (application interfaces, platform interfaces, role, integration phase, integration frameworks, technology and non-functional features). Finally the internals are concerned with elements related to the potential information needed during the development process of a system (nature, granularity, encapsulation, structural aspects, behavioural aspects, accessibility to source code).

Similar to our work to some extent, a classification framework to classify each of the proposed models, frameworks, or standards is proposed in [19], trying to determine what the core features of a software component are. The classification approach is different from ours; it includes identification of a component by a set of elements/characteristics (unit of composition, reuse, interface, interoperability, granularity, hierarchy, visibility, composition, state, extensibility, marketability, and support for OO). The classification includes only business components and business solutions. One of the problems with this classification is the non orthogonality of some of the characterized items.

In [5], in which ADLs are classified, components are defined as basic elements of ADLs. The components are distinguished by the following features:

interface, types, semantics, constraints, evolution, and non-functional properties.

In [10], a classification model is proposed to structure the CBSE body of knowledge. All research results are characterized according to several aspects (concepts, processes, roles, product concerns and business concerns, technology, off-the-shelf components and related development paradigms). Here, the component model is only considered as one of the fifty elements in the CBSE items. However, in this work, a more precise taxonomy of application domains is proposed. The paper identifies the following application domains in which component-based approaches are utilized: avionics, command and control, embedded systems, electronic commerce, finance, healthcare, real-time, simulation, telecommunications and, utilities.

In [7], several component models (JB, COM, MTS, CCM, .NET and OSGI) are mainly described according to the following criteria: Interfaces and Assembly using ACME notation, Implementation, and Lifecycle. The models are not compared or valuated, but rather these characteristics are described for each component model.

In [11], a study of several component models is presented that considers the following aspects: syntax, semantics and composition through an idealized component-based development lifecycle,. A smaller number of component models are considered (also UML and ADLs are included). Based on this study, a taxonomy centered on the composition criterion is proposed, which clarifies at which steps of the development process of a given component model, components can be composed and whether they can be retrieved from a repository to be composed. Further the different types of bindings (compositions) of some of the component models are discussed in more details. This taxonomy does not consider EFPs.

7.6 Conclusion

In this survey, we have presented a framework for the classification and comparison of component models, which identifies issues related to component-based development. This survey indicates that many principles comprised in the component-based approach are not always included in every component model. Many of these principles are taken and further developed from other approaches (OO development, modeling using ADLs) which also contributes to an unclear understanding of component-based development.

The intention of this work is to increase the understanding of component-based approach by identifying the main concerns, common characteristics and differences of component models. The proposed framework does not include all the elements of all component models since many of them have specific solutions some related to models, some related to particular technology solutions. Further we have not characterized the component themselves (like implementation, internal behavior, whether components are active or passive, and similar). The framework however identifies the minimal criteria for assuming a model to be a component model and it groups the basic characteristics of the models.

From the results we can recognize some recurrent patterns, such as: general-purpose component models utilize the “request response” style, while in the specialized domains (mostly embedded systems) “pipe & filter/dataflow” is the predominate style. We can also observe that support for composition of extra-functional properties is rather scarce. There are many reasons for that: in practice explicit reasoning and predictability of EFPs is still not widespread, there are unlimited number of different EFPs, and finally the compositions of many EFPs are not only the results of component properties, but also a matter outside component models for example of system architectures, which makes EFP an aspect that is difficult to handle at the level of traditional implementation languages.

In similarity with other technologies we could expect a convergence of the main characteristics of component models, i.e. becomes more standardized, using more commonly accepted concepts and terminology, even if the number of different component models will not necessary decreased. The aim of this work is to provide a help in this convergence process.

7.7 Survey of component models

In this appendix, we provide a brief overview of component models taken in the survey and their main characteristics. The component models are listed in the alphabetic order. The list should be understood as a provision of some characteristic examples, or examples of widely used component models in Software Engineering.

Note that when listing the component models we have not provided their product name with edition number except for cases in which the edition numbers are part of the name or indicate significant difference from the previous version.

AUTOSAR (AUTomotive Open System ARchitecture) [20], the new standard in automotive industry is the result of the partnership between several manufacturers and suppliers from the automotive field. The main focus of AUTOSAR is standardization of architecture, architectural components and their interoperability, which allows a separation of development of component-based applications from the underlying platform. AUTOSAR supports both the client-server and sender-receiver communication types. An AUTOSAR software component instance is only assigned to one computer node - Electronic Control Unit (ECU). The AUTOSAR software components are implemented in C. The main focus of AUTSOAR is the architecture not the component model itself.

BIP (Behavior, Interaction, Priority) [21] framework developed at Verimag is used for modelling heterogeneous real-time components. This heterogeneity is considered for components having different synchronization mechanisms (broadcast/rendez-vous), timed components or non-timed components. BIP focuses on component behaviour through a model with a three-layer structure of the components (Behaviour, Interaction and Priority); a component can be seen as a point in this three-dimensional space constituted by each layer. In this model, compound components, i.e components created from already existing ones, and systems are obtained by a sequence of formal transformations in each of the dimension. BIP comes up with its own programming language but targets C/C++ execution. Some connections to the analysis tools of the IF-toolset [22] and the PROMETHEUS tools [23] are also provided.

BlueArX [24] [25] is a component model developed and used by Bosch for the automotive control domain. BlueArX defines a hierarchical component model with focus on design-time, which does not require additional run-time or memory resources on the target hardware. A BlueArX component consists of specification, documentation and implementation (as object or C source code). BlueArX components and interfaces are specified using MSRSW (Manufacturer Supplier Relationship SoftWare), a standardized XML format. Components communicate using client-server and sender-receiver interfaces. Besides name and type the interfaces specification lists additional details (e.g. mapping between internal and physical representation, value range, and physical unit). Other interfaces address component configuration (variation points), calibration data and extra-functional properties, like timing, memory usage or generic specification of other properties.

COMDES II [26], developed at University of Southern Denmark, defines various types of components to address both architectural and behavioral properties of control software systems. It employs a two-level model to specify system architecture. At the first (system) level a distributed control application is conceived as a network of communicating actors and at the second (actor) level an actor is specified as a software artifact containing a single actor task and multiple I/O drivers. The functional behavior is specified by a composition of different function block instances which implement concrete computation or control algorithms. COMDES II defines four kinds of functional blocks: basic, composite, modal and state machine. The former two can be used to model continuous behavior (data flow) and the later two describe the sequential behavior (control flow). All non-functional information such as physicality, real-time and concurrency is specified with respect to actors.

CompoNETS [27], developed at Universit Toulouse 1, is based on CCM where additionally the internal behavior of a software component and inter-component communication are specified by Petri Nets. Accordingly, a mapping from the constructs of the component models (e.g. facets, receptacles, event sources and sinks) to the constructs of Petri-net based behavioral formalism (e.g. places, transitions etc.) is defined. Other characteristics are the same (or very similar) to CCM.

CCM (CORBA Component Model) [28] evolved from Corba object model and it was introduced as a basic model of the OMGs component specification. The CCM specification defines an abstract model, a programming model, a packaging model, a deployment model, an execution model and a metamodel. The metamodel defines the concepts and the relationships of the other models. CORBA components communicate with outside world through ports. CCM uses a separate language for the component specification: Interface Definition Language (IDL). CCM provides a Component Implementation Framework (CIF) which relies on Component Implementation Definition Language (CIDL) and describes how functional and nonfunctional part of a component should interact with each other. In addition, CCM uses XML descriptors for specifying information about packaging and deployment. Furthermore, CCM has an assembly descriptor which contains metadata about how two or more components can be composed together.

EJB (Entreprise JavaBeans) [29], developed by Sun MicroSystems envisions the construction of object-oriented and distributed business applications. It provides a set of services, such as transactions, persistence, concurrency, interoperability. EJB differs three different types of components (The EntityBeans the SessionBean and the MessageDrivenBeans). Each of these beans is deployed in an EJB Container which is in charge of their management at runtime (start, stop, passivation or activation) and EFPs (such as security, reliability, performance). EJB is heavily related to the Java programming language.

Fractal [30] is a component model developed by France Telecom R&D and INRIA. It intends to cover the whole development lifecycle (design, implementation, deployment and maintenance/management) of complex software systems. It includes several features, such as nesting, sharing of components and reflexivity in that sense that a component may respectively be created from other components, be shared between components and can expose its internals to other components. The main purpose of Fractal is to provide an extensible, open and general component model that can be tuned to fit a large variety of applications and domains. Fractal includes different instantiations and implementations: a C-implementation called Think, which targets especially the embedded systems and a reference implementation, called Julia and written in Java.

Koala [31] is a component model developed by Philips for building software for consumer electronics. Koala components are units of design, development and reuse. Koala has a set of modeling languages: Koala IDL is used to specify Koala component interfaces, its Component Definition Language (CDL) is used to define Koala components, and Koala Data Definition Language (DDL) is used to specify local data of components. Koala components communicate with their environment or other components only through explicit interfaces statically connected at design time. Koala targets C as implementation language and uses source code components with simple interaction model. Koala pays special attention to resource usage such as static memory consumption.

KobrA (KOmponentenBasieRte Anwendungsentwicklung) [32] is a hierarchical component model that supports a model-driven, UML-based representation of components. In KobrA components are not physical components like in the contemporary physical technologies (e.g. CORBA, EJB, .NET) but logical building blocks of the software system. The components can be constructed in any UML modeling tool and deposited into a file system. They can be compared to subsystems in UML with additional behavior. KobrA uses UML class diagrams to specify structure, functional model to describe functionality and finally the behavioral model describes the component behavior. Composition of components is done in the design phase by direct method calls.

IEC 61131 [33] is a standard for the design of Programmable Logic Controllers approved by the International Electrotechnical Commission (IEC). In this standard, the software units are called function blocks and based on incoming events, they execute some algorithms to update the internal variables. This standard has been further extended to IEC 61499 [34] which provides distribution in the runtime environment through high-level abstraction of communication primitives. IEC 61499 is an open communication standard for distributed control systems.

JB (Java Beans) [35] developed by Sun Microsystems is based on Java programming language. In the JavaBeans specification a bean is a reusable software component that can be visually composed into applets, applications, servlets, and composite components, using visual application builder tools. Programming a Java component requires definition of three sets of data: i) properties (similar to the attributes of a class); ii) methods; and iii) events which are an alternative to method invocation for sending data. JavaBeans was primarily designed for the construction of graphical user interface. The model defines three types of interaction points, referred to as ports: (i) methods, as in Java, (ii) properties, used to parameterize the component at composition time, (iii) event sources, and event sinks (called listeners) for event-based communication.

COM (Microsoft Component Object Model) [36] is one of the most commonly used software component models for desktop and server side applications. A key principle of COM is that interfaces are specified separately from both the components that implement them and those that use them. COM defines a dialect of the Interface Definition Language (IDL) that is used to specify object-oriented interfaces. Interfaces are object-oriented in the sense that their operations are to be implemented by a class and passed a reference to a particular instance of that class when invoked. A concept known as interface navigation makes it possible for the user to obtain a pointer to every interface supported by the object. This is based on VTable. Although COM is primarily used as a general-purpose component model it has been ported for development of embedded software and extended for distributed information systems

OpenCOM [37] is a lightweight component model developed at Lancaster University which aims at exploiting component-based techniques within middleware platforms. It is built atop a subset of Microsofts COM. These include the binary level interoperability standard, Microsofts IDL, COMs globally unique identifiers and the IUnknown interface. The higherlevel features of COM such as distribution, persistence, transactions and security are not used. The key concepts of OpenCOM are capsules, components, interfaces, receptacles and connections. Capsules are runtime containers and they host components. Each component implements a set of custom receptacles and interfaces. A receptacle describes a unit of service requirement, an interface expresses a unit of service provision, and a connection is the binding between an interface and a receptacle of the same type.

OSGi (Open Services Gateway Initiative) [38] is a consortium of numerous industrial partners working together to define a service-oriented framework with an open specifications for the delivery of multiple services over wide area networks to local networks and devices. Contrary to most component definitions, OSGI emphasis the distinction between a unit of composition and a unit of deployment in calling a component respectively service or bundle. It offers also, at contrary to most component models, a flexible architecture of systems that can dynamically evolve during execution time. This implies that in the system, any components can be added, removed or modified at run-time. In relying on Java, OSGI is platform independent. There exists several additions of OSGi that provides additional characteristics.

Palladio Component Model [39], developed at University of Oldenburg and University of Karlsruhe, provides a domain specific modeling language for component-based software architectures, which is tuned to enable early life-cycle performance predictions. Palladio defines its own metamodel specified in EMF/Ecore and divided into several domain specific languages for each developer role (i.e. component developers, software architects, system deployers and domain experts). All specifications can be combined to derive a full Palladio component model instance. As a starting point for implementing the systems business logic, the instance can be converted into Java code skeletons via Model2Text transformation. Components are specified via provided and required interfaces which consist of a list of service signatures. In order to allow accurate performance prediction, a so called resource demanding service effect specification can be added to each provided service to describe the sequence of called required services, resource usage, transition probabilities, loop iteration numbers, and parameter dependencies. Components and their roles can be connected via assembly connectors to build an assembly.

Pecos [40] is a joined project between ABB Corporate Research and Bern University. Its goal is to provide an environment that supports specification, composition, configuration checking and deployment for reactive embedded systems built from software components. There are two types of components, leaf components and composite components. The inputs and outputs of a component are represented as ports. At design phase composite components are made by linking their ports with connectors. Pecos targets C++ or Java as implementation language, so the run-time environment in the deployment phase is the one for Java or C++. Pecos enables specification of EFPs such as timing and memory usage in order to investigate in prediction of the behaviour of embedded systems.

Pin [41] component model developed at Carnegie Mellon Software Engineering Institute (SEI) is used as a basis in prediction-enabled component technologies (PECTs). By using principles from PECT it aims at achieving predictability by construction i.e. constraining the design and the implementation to analyzable patterns. To achieve predictability of a particular property PECT proposes a building of a reasoning framework that includes a component technology powered by analytical interface used for a specification of a property of interest and analysis theory used in provision of the system property composed from component properties. Accordingly, in order to perform analysis, proper analysis theories must be found and implemented in a suitable underlying component technology. PECT currently supports three reasoning frameworks fro Pin Component model: ABA - for predicting average latency in assemblies with periodic tasks, ss - for predicting average latency in stochastic tasks managed by a sporadic server and ComFoRT -for formal verification of temporal safety and liveness. Pin Components are defined in an ADL-like language, in the component and connector style, so called Construction and Composition Language (CCL). Pin components are fully encapsulated, so the only communication channels from a component to its environment and back are sink and source pins. Composition of components is obtained by connecting source and sink pins and the behavior of the interaction, which is specified as executable state machines.

ProCom [42] is a component model for control-intensive distributed embedded systems being developed at PROGRESS Strategic Research Center at Mlardalen University, Sweden. ProCom consists of two layers, in order to address different concerns that exist at different levels of a distributed embedded system. The upper layer, ProSys, focuses on modeling of the whole system or large subsystems. It considers complex active subsystems as components and captures the message flow between them. The lower layer, ProSave, serves for modeling of ProSys components on a detailed level. It explicitly captures the data transfer and control-flow between the components using a rich set of connectors which makes a platform for modelling control loops in a way that allows them to be easily analyzed and synthesized. The analysis is facilitated by the explicit control-flow and by the abstraction provided by components (read-execute-write semantics, encapsulation). The model provides support for different types of analysis by making possible to attach various models (behaviour, timing, resource utilization, etc.) to different architectural elements such as components, connections, subsystems, etc. Further, it considers deployment as a specific activity which includes components allocations, transformation of components to the entities complied with the execution model, and synthesis, i.e. creation of a glue code.

Robocop [43] is a component model developed by the consortium of the Robocop ITEA project, inspired by COM, CORBA and Koala component models. It aims at covering all the aspects of the component-based development process for the high-volume consumer device domain. Robocop component is a set of possibly related models and each model provides particular type of information about the component. The functional model describes the functionality of the component, whereas the extra-functional models include modeling of timeliness, reliability, safety, security, and memory consumption. Robocop components offer functionality through a set of services and each service may define several interfaces. Interface definitions are specified in a Robocop Interface Definition Language (RIDL). The components can be composed of several models, and a composition of components is called an application. The Robocop component model is a major source of for ISO standard ISO/IEC 23004-1:2007 Information technology - Multimedia Middleware.

Rubus [44] component was developed as a joint project between Arcticus Systems AB and Mlardalen University. The Rubus component model runs on top of the Rubus real-time operating system. It focuses on the real-time properties and is intended for small resource constrained embedded systems. Components are implemented as C functions performed as tasks. A component specifies a set of input and output ports, persistent states, timing requirements such as releasetime, deadline. Components can be combined to form a larger component which is a logical composition of one or more components.

SaveCCM [45], developed within the SAVE project by several Swedish universities, is a component model specifically designed for embedded control applications in the automotive domain with the main objective of providing predictable vehicular systems. SaveCCM is a simple model that constrains the flexibility of the system in order to improve the analysability of the dependability and of the real-time properties. The model takes into consideration the resource usage, and provides a lightweight run-time framework. For component and system specification SaveCCM uses SaveCCM language which is based on a textual XMLsyntax and on a subset of UML2.0 component diagrams.

SOFA (Software Appliances) [46] is a component model developed at Charles University in Prague. A SOFA component is specified by its frame and architecture. The frame can be viewed as a black box and it defines the provided and required interfaces and its properties. However a framework can also be an assembly of components in a composite component. The architecture is defined a grey-box view of a component, as it describes the structure of a component until the first level of nesting in the component hierarchy. SOFA components and systems are specified by an ADL-like language, Component Description Language (CDL). The resulting CDL is compiled by a SOFA CDL compiler to their implementation in a programming language C++ or Java. SOFA components can be composed by method calls through connectors. The SOFA 2.0 component model is an extension of the SOFA component model with several new services: dynamic reconfiguration, control interfaces and multiple communication styles between the components.

Bibliography

[1] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, December 1997.

[2] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., 2001.

[3] Michel Chaudron and Ivica Crnkovic. *Software Engineering: Principles and Practice, 3rd Edition*, chapter chapter 18 in H. van Vliet, Component-Based Software Engineering. Wiley, 2008.

[4] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. Moving architectural description from under the technology lamppost. *Inf. Softw. Technol.*, 49(1):12–31, 2007.

[5] Medvidovic, Nenad and Taylor, Richard N. . A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, January 2000.

[6] Ivica Crnkovic, Michel Chaudron, and Stig Larsson. Component-based Development Process and Component Lifecycle. *Journal of Computing and Information Technology*, 13(4):321–327, November 2005.

[7] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-Based Software Systems*.

[8] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999.

[9] Ivica Crnkovic, Magnus Larsson, and Otto Preiss. Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes. pages 257–278. 2005.

[10] Gerald Kotonya, Ian Sommerville, and Steve Hall. Towards A Classification Model for Component-Based Software Engineering Research. In *EUROMICRO '03: Proceedings of the 29th Conference on EUROMICRO*, page 43, Washington, DC, USA, 2003. IEEE Computer Society.

[11] Kung-Kiu Lau and Zheng Wang. Software Component Models. *Software Engineering, IEEE Transactions on*, 33(10):709–724, 2007.

[12] Oxford advanced learners dictionary.

[13] The Object Management Group. UML Superstructure Specification v2.1, April 2009.

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

[15] H.P. Breivold and M. Larsson. Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles. pages 13–20, Aug. 2007.

[16] John Reekie, Stephen Neuendorffer, Christopher Hylands, and Edward A. Lee. Software Practice in the Ptolemy. Technical Report GSRC-TR-1999-01, Gigascale Silicon Research Center, April 1999.

[17] Sherif Yacoub, Hany Ammar, and Ali Mili. A Model for Classifying Component Interfaces. In *Second International Workshop on Component-Based Software Engineering, in conjunction with the 21 st International Conference on Software Engineering (ICSE99*, pages 17–18, 1999.

[18] Sherif Yacoub, Hany Ammar, and Ali Mili. Characterizing a Software Component. In *In Proceedings of the 2nd Workshop on Component-Based Software Engineering, in conjunction with ICSE99*, 1999.

[19] Klement J. Fellner and Klaus Turowski. Classification Framework for Business Components. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8*, page 8047, Washington, DC, USA, 2000. IEEE Computer Society.

[20] AUTOSAR Development Partnership. Technical Overview V2.2.1, February 2008. Available from `www.autosar.org`.

[21] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12. IEEE, 2006.

[22] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF Toolset. In *SFM*, pages 237–267, 2004.

[23] Gregor Gssler. Prometheus - A Compositional Modeling Tool for Real-Time Systems.

[24] 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. In *ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 307–311, Washington, DC, USA, 2008. IEEE Computer Society.

[25] 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. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009.

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

[27] Rmi Bastide and Eric Barboni. Component-Based Behavioural Modelling with High-Level Petri Nets. In *MOCA '04 - Third Workshop on Modelling of Objects, Components and Agents , Aahrus, Denmark , 11/10/04-13/10/04*, pages 37–46. DAIMI, octobre 2004.

[28] OMG CORBA Component Model v4.0. Available from `http://www.omg.org/docs/formal/06-04-01.pdf`.

[29] EJB 3.0 Expert Group. JSR 220: Enterprise JavaBeansTM,Version 3.0 EJB Core Contracts and Requirements Version 3.0, Final Release, May 2006.

[30] E Bruneton, T Coupaye, and J Stefani. The Fractal component model specification. *The ObjectWeb Consortium, Tech. Rep., Februar*, 2004.

[31] 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.

[32] Atkinson, Colin and Bayer, Joachim and Bunse, Christian and Kamsties, Erik and Laitenberger, Oliver and Laqua, Roland and Muthig, Dirk and Paech, Barbara and Wüst, Jürgen and Zettel, Jörg. *Component-based product line engineering with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

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

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

[35] Sun Microsystems. JavaBeans specification, 1997.

[36] D Box. *Essential COM. Object Technology Series*. Addison-Wesley, 1997.

[37] M Clarke, GS Blair, G Coulson, and N Parlavantzas. An Efficient Component Model for the Construction of Adaptive Middleware. *Proceedings of the IFIP/ACM International Conference on Middleware*, 2001.

[38] OSGi Alliance. OSGi Service Plaform Core Specification, V4.1, 2007.

[39] S Becker, H Koziolek, and R Reussner. Model-Based Performance Prediction with the Palladio Component Model. *the 6th international workshop on Software and performance*, 2007.

[40] M Winter, C Zeidler, and C Stich. The PECOS software process. *Workshop on Components-based Software Development Processes, ICSR*, 2002.

[41] 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.

[42] Séverine Sentilles, Aneta Vulgarakis, Tomas Bures, Jan Carlson, and Ivica Crnkovic. A Component Model for Control-Intensive Distributed

Embedded Systems. In Michel R.V. Chaudron and Clemens Szyperski, editors, *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008)*, pages 310–317. Springer Berlin, October 2008.

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

[44] Arcticus Systems. Rubus Software Components. `www.arcticus-systems.com`.

[45] 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.

[46] T Bure, P Hntynkal, and F Plil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. *Proceedings of SERA*, 2006.

Chapter 8

Paper B: A Component Model for Control-Intensive Distributed Embedded Systems

Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković

In Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008), Karlsruhe, Germany, October, 2008.

Abstract

In this paper we focus on design of a class of distributed embedded systems that primarily perform real-time controlling tasks. We propose a two-layer component model for design and development of such embedded systems with the aim of using component-based development for decreasing the complexity in design and providing a ground for analyzing them and predict their properties, such as resource consumption and timing behavior. The two-layer model is used to efficiently cope with different design paradigms on different abstraction levels. The model is illustrated by an example from the vehicular domain.

8.1 Introduction

A special class of embedded systems are control-intensive distributed systems which can be found in many products, such as vehicles, automation systems, or distributed wireless networks. In this category of systems as in most embedded systems, resources limitations in terms of memory, bandwidth and energy combined with the existence of dependability and real-time concerns are obviously issues to take into consideration.

Another problem when developing such systems is to deal with the rapidly increasing complexity. For example in the automotive industry, the complexity of the electronic architecture is growing exponentially, directed by the demands on the driver's safety, assistance and comfort [1]. In this class of systems, distribution is also an important aspect. The architecture of the electronic systems is distributed all over the corresponding product (car, production cell, etc.), following its physical architecture, to bring the embedded system closer to the sensed or controlled elements.

In this paper, we propose a new component model called ProCom with the following main objectives: (i) to have an ability of handling the different needs which exist at different granularity levels (provide suitable semantics at different levels of the system design); (ii) to provide coverage of the whole development process; (iii) to provide support to facilitate analysis, verification, validation and testing; and (iv) to support the deployment of components and the generation of an optimized and schedulable image of the systems. The focus of this paper is on the component model itself, described as means for designing and modelling system functionality and as a framework that enables integration of different types of models for resource and timing analysis.

The component model is a part of the PROGRESS approach [2] that distinguishes three key activities in the development: design, analysis and deployment. The *design* activity provides the architectural description of the system compliant with the semantic rules of the component model presented in this paper and enables the integration analysis and deployment capabilities. *Analysis* is carried out to ensure that the developed embedded system meets its dependability requirements and constraints in terms of resource limitations. The proposed component model provides means to handle and reuse the different information generated during the analysis activity. The *deployment* activity is specific for control-intensive embedded systems; due to timing requirements and resource constraints, the execution models can be very different from the design models. Typically, execution units are processes and threads of tasks.

The main focus of this paper is oriented towards system design. The two supplementary activities (analysis and deployment) are outside the scope of the paper. A component model that enables a reusable design, takes into consideration the requirements' characteristics for control-intensive embedded systems, and is used as an integration frame for analysis and deployment, is elaborated in the subsequent sections.

The ideas underlying ProCom emanate partly from the previous work on the SaveComp Component Model (SaveCCM) [3] within the SAVE project, such as the emphasis on reusability, a possibility to analyse components for timing behavior and safety properties. Several other concepts and component models have inspired the ProCom Design. Some of them are the Rubus component model [4], Prediction-Enabled Component Technology (PECT) [5], AUTOSAR [1], Koala [6], the Robocop project [7], and BIP [8].

8.2 The ProCom two layer component model

In designing our component model, we have aimed at addressing the key concerns which exist in the development of control-intensive distributed embedded systems. We have analyzed these concerns in our previous work [9], with the conclusion that in order to cover the whole development process of the systems, i.e. both the design of a complete system and of the low-level control-based functionalities, two distinct levels of granularity are necessary.

Taking into consideration the difference between those levels, we propose a two-layer component model, called *ProCom*. It distinguishes a component model used for modelling independent distributed components with complex functionality (called *ProSys*) and a component model used for modelling small parts of control functionality (called *ProSave*). ProCom further establishes how a ProSys component may be modelled out of ProSave components. The following subsections describe both of the layers and their relation. The complete specification of ProCom is available in [10].

8.2.1 ProSys — the upper layer

In ProSys, a system is modeled as a collection of concurrent, communicating *subsystems*, possibly developed independently. Some of those subsystems, called *composite subsystems*, can in turn be built out of other subsystems, thus making ProSys a hierarchical component model. This hierarchy ends with the so-called *primitive subsystems*, which are either subsystems coming from the ProSave layer or non-decomposable units of implementation (such as COTS or legacy subsystems) with wrappers to enable compositions with other subsystems. From a CBSE perspective, subsystems are the “components” of the ProSys layer, i.e., design or implementation units that can be developed independently, stored in a repository and reused in multiple applications.

The communication between subsystems is based on the asynchronous message passing paradigm which allows transparent communication (both locally or distributed over a bus). A subsystem is specified by typed input and output *message ports*, expressing what type of messages the subsystem receives and sends. The specification also includes attributes and models related to functionality, reliability, timing and resource usage, to be used in analysis and verification throughout the development process. The list of models and attributes used is not fixed and can be extended.

Message ports are connected via *message channels* — explicit design entities representing a piece of information that is of interest to several subsystems — as exemplified in Fig. 8.1. The message channels make it possible to express that a particular piece of shared data will be required in the system, before any producer or receiver of this data has been defined. Also, information about shared data such as precision, format, etc. can be associated with the message channel instead of with the message port where it is produced or consumed. That way, it can remain in the design even if, for example, the producer is replaced by another subsystem.

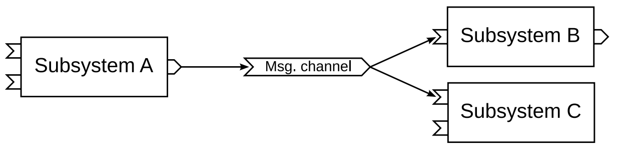


Figure 8.1: Three subsystems communicating via a message channel.

8.2.2 ProSave — the lower layer

The ProSave layer serves for the design of single subsystems typically interacting with the system environment by reading sensor data and controlling actuators accordingly. 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.

A ProSave component is of a collection of services, each providing a particular functionality. 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. Fig. 8.2 illustrates these concepts. The data of an output group are produced at the same time, at which the trigger port of that group is also activated. Having multiple output groups allows the service to produce time critical parts of the output early.

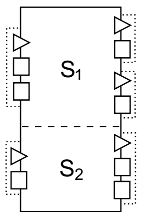


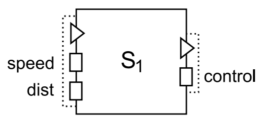
Figure 8.2: A ProSave component with two services; S_1 has two output groups and S_2 has a single output group. Triangles and boxes denote trigger- and data ports, respectively.

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 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. Before the service returns to the inactive state again, each of its output groups should be written exactly once.

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

A component also includes a collection of structured *attributes* which define simple or complex types of component properties such as behavioural models, resource models, certain dependability measures, and documentation. These attributes can be explicitly associated with a specific port, group or service (e.g. the worst case execution time of a service, or the value range of a data port), or related to the component as a whole, for example a specification of the total memory footprint. New attribute types can also be added to the model.

The functionality of a component can either be realized by code (*primitive component*), or by interconnected sub-components (*composite component*). For primitive components, in addition to a function called at system startup to initialise the internal state, each service is implemented as a single non-suspending C function. Fig. 8.3 shows an example of the header file of a primitive component.



```
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 8.3: A primitive component and the corresponding header file.

Composite components internally consist of *sub-components*, *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 analyse 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. Fig. 8.4 shows a typical usage of the selection connector together with *or* connectors.

ProSave follows the push-model for data transfers and the triggered service always uses the latest value written to each input data port. Since communication may eventually be realised over a physical connection, the transfer of data and triggering is not an atomic operation. For triggering and data appearing together at an output group, however, the semantics specify that all data should be delivered to their destinations before the triggering is transferred, to avoid components being triggered before the data arrives.

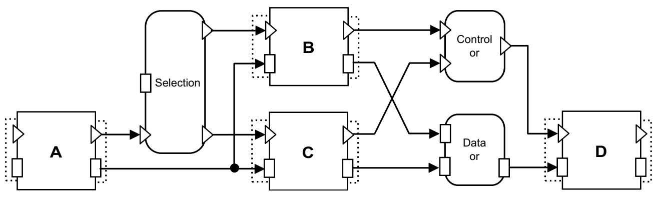


Figure 8.4: A typical usage of *selection* and *or* connectors. When component A is finished, either B or C is executed, depending on the value at the selection data port. In either case, component D is executed afterwards, with the data produced by B or C as input.

8.2.3 Integration of layers — combining ProSave and ProSys

ProCom provides a mechanism for integrating the low-level design of a subsystem described by ProSave into the high-level design described by ProSys. A ProSys primitive subsystem can be further specified using ProSave (as exemplified in Fig. 8.6). Concretely, in addition to ProSave components, connections and ProSave connectors, additional connector types are introduced to *(a)* map the architectural style (message passing used in ProSys to pipes-and-filters used in ProSave, and vice versa), and *(b)* specify periodic activation of ProSave components.

Periodic activation is provided by the clock connector, with a single output trigger port which is repeatedly activated at a given rate. 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 appearing on the ProSave level. An input message port corresponds to a connector with output ports. Whenever a message is received by the message port, it writes the message data to the output data port and activates the output trigger. Oppositely, output message ports correspond 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.

These composition mechanisms do not only allow a consistent design of the entire system by integrated pre-existing subsystems but also provide mechanisms for analysis of particular attributes such as timing properties or performance of the entire system using specifications or analysis results of the subsystems.

8.3 Example

To illustrate the ProCom component model we use as an example an electronic stability control (ESC) system from the vehicular domain. In addition to anti-lock braking (ABS) and traction control (TCS), which aim at preventing the wheels from locking or spinning when braking or accelerating, respectively, the ESC also handles sliding caused by under- or oversteering.

The ESC can be modeled as a ProSys subsystem, as shown in Fig. 8.5.

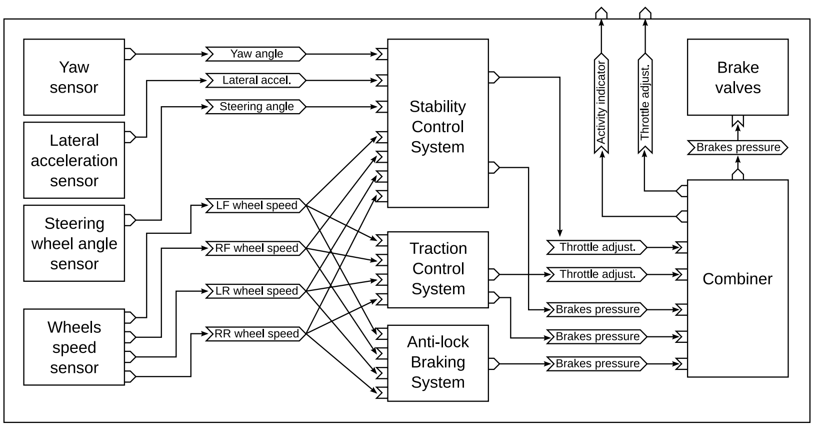


Figure 8.5: The ESC is a composite subsystem, internally modelled in ProSys.

Inside, we find subsystems for the sensors and actuators that are local to the ESC. There are also subsystems corresponding to specific parts of the ESC functionality (SCS, TCS and ABS). In the envisioned scenario, the TCS and ABS subsystems are reused from previous versions of the car, while SCS corresponds to the added functionality for handling under- and oversteering. Finally, the "Combiner" subsystem is responsible for combining the output of the three.

The internal structure of a SCS primitive subsystem is modeled in ProSave (see Fig. 8.6). The SCS contains a single periodic activity performed at a frequency of 50 Hz, expressed by a clock connector. The clock first activates the two components responsible for computing the actual and desired direction, respectively. When both components have finished their respective tasks, the "Slide detection" component compares the results (i.e., the actual and desired directions) and decides whether or not stability control is required. The fourth component computes the actual response, i.e., the adjustment of brakeage and acceleration.

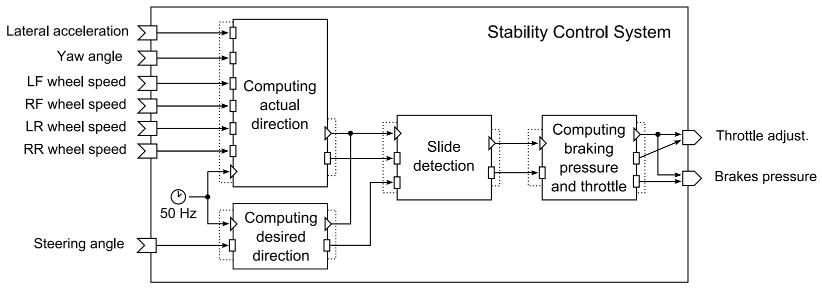


Figure 8.6: The SCS subsystem, modelled in ProSave.

8.4 Conclusions

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.

The future work on ProCom includes elaborating on advanced features of the component model (e.g. static configuration, mode shifting, error-handling, etc.), building an integrated development environment and evaluating the proposed approach in real industrial case-studies.

Acknowledgement

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS.

Bibliography

[1] AUTOSAR Development Partnership. Technical Overview V2.2.1, February 2008. Available from `www.autosar.org`.

[2] Hans Hansson, Mikael Nolin, and Thomas Nolte. Beating the Automotive Code Complexity Challenge. In *National Workshop on High-Confidence Automotive Cyber-Physical Systems*, Troy, Michigan, USA, April 2008.

[3] 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.

[4] Arcticus Systems. Rubus Software Components. `www.arcticus-systems.com`.

[5] Kurt C. Wallnau. Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC). Technical Report CMU/SEI-2003-TR-009, Carnegie Mellon, 2003.

[6] 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.

[7] Robocop project page. `www.extra.research.philips.com/euprojects/robocop`.

[8] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12. IEEE, 2006.

[9] Tomáš Bureš, Jan Carlson, Séverine Sentilles, and Aneta Vulgarakis. A Component Model Family for Vehicular Embedded Systems. In *The Third International Conference on Software Engineering Advances*. IEEE, October 2008.

[10] 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.

Chapter 9

Paper C: Embedded Systems Resources: Views on Modeling and Analysis

Aneta Vulgarakis and Cristina Seceleanu
In Proceedings of the 1st IEEE International Workshop On Component-Based Design Of Resource-Constrained Systems (CORCS 2008), IEEE CS, Turku, Finland, July, 2008.

Abstract

The conflicting requirements of real-time embedded systems, e.g. minimizing memory usage while still ensuring that all deadlines are met at run-time, require rigorous analysis of the system's resource consumption, starting at early design stages. In this paper, we glance through several representative frameworks that model and estimate resource usage of embedded systems, pointing out advantages and limitations. In the end, we describe our own view on how to model and carry out formal analysis of embedded resources, along with developing the system.

9.1 Introduction

Embedded systems (ES) 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. The *resources* that such systems use (CPU share, memory, energy, bus bandwidth, ports etc.) are limited in capacity, expensive and (usually) not extensible during the system's lifetime. In contrast to the fixed nature of available resources, software can be subjected to change. To ensure that the combination of components fits on a particular target platform, one needs to be able to estimate the resource consumed by the application software. Further, to facilitate reuse and fast integration of pre-designed components, the ES design techniques should cover systems having minimal resource requirements.

The limited nature of the available resources, especially memory size and computation resources, complicates meeting the real-time constraints. Hence, the ability to quantify and reason about *trade-offs* between various resources, under given technical constraints, is essential. *Prediction* methods for resource usage should be available throughout the whole system's development lifecycle. Access to such information at early stages of design can help the designer to prevent resource conflicts at run-time. This can, in turn, help to decrease the ES' development time and, consequently, reduce development costs. Analysis will then require models of resource usage and theories for composing resource usage models.

Extensive research has been recently devoted to modeling and analyzing ES resource consumption, in component-based design frameworks. Code-level memory estimation for, e.g. Koala- [1, 2] and Robocop- based [3] compositions, as well as higher-level formal approaches [4–7] aim to establish whether certain resource-related properties hold for a system model.

The main problem of building an ES is correlating its various models of different degrees of detail, which are related via abstraction or refinement. This impacts also on transferring the resource analysis results from one design stage to another. Even so, performing resource consumption analysis at design-time might guide the selection of the appropriate components from existing repositories, when adopting a bottom-up ES design method. Similarly, in a top-down approach, it could help in the correct decomposition of the system's specification into smaller parts (sub-systems) such that the latter could be easier matched by existing sub-systems.

Designing a *predictable* ES amounts, among other things, to establishing that the required resources do not exceed the available resources. For many

ES, costs and other constraints make it important to “minimize” resource usage. Here, we advocate a *deployment-oriented* view of ES resource modeling and analysis. For this, we argue that it is important, first, to keep the abstracted hardware model at similar level of detail as the one of the software model, such that the resource-usage analysis can become progressively refined, depending on the design stage; second, we believe that striving for a general formal framework that could be uniformly used throughout the design cycle could be of great help in solving the problem of correlating analysis results.

The variety of approaches existing in the literature indicates the possible difficulty in gathering all resource reasoning in one uniform theory. This calls for a fresh look on resource-aware design methods, based on the lessons drawn from the existing component-based approaches. In the following, we survey some of the current trends in analyzing ES resource accesses, and point out their limitations. We end by describing our view on what is needed to make such methods applicable on a variety of ES, and underline the essential demands of a resource-aware component-based design perspective.

9.2 Motivating Example

The systematic analysis of the resource consumption of an embedded system must include ways of semantic representation of various types of resources, be they of continuous, monotonic type (like energy), of continuous, non-monotonic type (like memory), or of discrete nature (e.g. I/O ports). A representative analysis goal would be to answer the *feasibility* question: does the composition of the worst-case resource requirements of components stay within the available resources provided by the implementation platform? Checking whether the resource-wise composition is feasible might not be always straightforward.

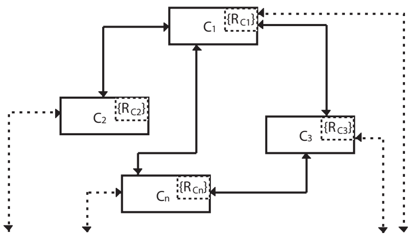


Figure 9.1: n-component Embedded System with resource annotations.

In practice, it may be often necessary to replace a component with another one having the same functionality, yet using a more sophisticated control algorithm that requires bigger memory resources. Alternatively, if we assume a repository of models, the designer might need, at some point, a refined component model, with modified behavior or more efficiently implementable data structures.

Let us now imagine the following scenario. Suppose that we start building up an embedded system for which we identify the interconnected software components as being $C_1, C_2, \ldots, C_n$ (see Figure 9.1). The dotted lines represent connections to other possible system components. We also assume that the hardware abstraction provides us with global available resources R. Consider that the computed resource requirement of C_1 is R_{C1}, of C_2, R_{C2} and so on. In Figure 9.1, the components are annotated with this information.

Suppose now that a different designer wants to use some component B, from the repository, instead of C_1 (for one of the reasons mentioned previously). So, we replace C_1 by B, both functionally and resource-wise. However, it so happens that B needs more resources than C_1 to perform its function: $R_B > R_{C1}$. Intuitively, the resource feasibility test will fail for the new composition, thus preventing us from accommodating B. In order to be able to include B in the system, we need to "fine-tune", in the sense of decreasing enough, the resource requirements of one or more components, for instance, by code-optimization. Then, by rechecking resource feasibility, we should get a positive answer.

A more challenging situation arises if we do not have access to the components implementation. How can we then accommodate B? One could think of trying to change the communication between components, or maybe the allocation of components to hardware units. We would be interested to assess, before deployment, how would any of these design decisions affect the system overall resource consumption. This amounts to finding an appropriate trade-off between different configuration requirements and constraints.

Performing all kinds of analysis of the embedded system's resource usage, starting at an early stage of design, and up to an as close to implementation stage as possible, 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 some repository, choosing among various admissible architectural designs, or transforming a component model into one with less resource requirements.

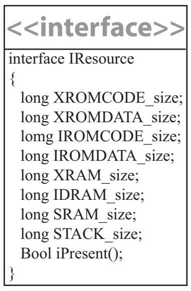
9.3 Modeling and Analyzing ES Resources: Representative Current Approaches

9.3.1 Koala and Robocop: Code-level Analysis

The importance of predicting resource consumption of component assemblies has motivated many researchers to investigate the issue. Compositional ways of estimating the static memory consumption of *Koala*-based embedded system models are already here to help us live up to the resource prediction challenge [1, 2]. Koala [8] is a software component model, introduced by Philips Electronics, designed to build product families of consumer electronics. In the mentioned approaches, resource information is exposed at the component's interface. The *provides* interface defines the operations offered by the component, whereas the *requires* interface defines the operations of other interfaces that the component needs to use. Since in a Koala model all the external functionality that is required by the component needs to go through the "requires" interface, it is somewhat straightforward to estimate the use of the system's resources, such as memory consumption. To estimate a Koala component's static memory consumption, one can assume that a special type of *reflection* provides the interface. For this purpose, Eskenazi et al. introduce the interface *IResource*, which contains the memory consumption demands of each component (see Figure 9.2). Each of this interface's members corresponds to a particular type of memory. A formula for estimating the memory size of each type of memory is added to the IResource implementation. Since the static properties of a compound component are specified in the reflection interface of its constituents, it follows that it is possible to reuse the respective components in a similar, yet different, composition, without changing its specification.

The above technique supports budgeting, that is, the expected values of the resource consumption of non-implemented components can also be accounted for. An important drawback of the approach is that it can only be used on specific, reduced-size scenarios and concrete component model for which the set of components instantiated in a composition is known before run-time. However, in real-world applications, the situation is much more complicated. If the set of instantiated components changes during run-time, the method will only estimate the memory consumption of the composition for a snapshot of components instantiated at that moment. A lot of experiments, measurements and simulations need to be carried out on the application.

Full state-space analysis of system models is most of the time accompanied by combinatorial complexity, as encountered by model checking approaches.



```
<<interface>>
interface IResource
{
  long XROMCODE_size;
  long XROMDATA_size;
  lomg IROMCODE_size;
  long IROMDATA_size;
  long XRAM_size;
  long IDRAM_size;
  long SRAM_size;
  long STACK_size;
  Bool iPresent();
}
```

Figure 9.2: Example of an ”IResource” interface of a Koala component.

In order to avoid such complexity, Jonge et al. [3] introduce a *scenario*-based prediction of *run-time* resource consumption, this time for the *Robocop* component model [9], a variant of the Koala component model. This approach delivers resource estimations for a set of scenarios that represent critical usages/executions of the system. The proposed resource model specifies the predicted resource consumption for all the operations implemented by the services of an executable component. As such, the model contains a number of cost functions that give the operations' costs. There can be multiple cost functions, for each resource. To increase the faithfulness of the prediction, the resources that are claimed and released are specified per operation. Figure 9.3 shows a revised example of how the service specification is done in [3]. Basically, it specifies service s_1 that requires interfaces I_2 and I_3, and provides interface I_1. Service s_1 implements the operation f that uses the operations g and h from interfaces I_2 and I_3, respectively. In Figure 9.3, we assume that operation f requires 1200 cpu cycles, without counting the invocations of I_2.g and I_3.h; also, operation f claims 100 bytes of memory before execution, and releases 100 bytes after finishing execution.

Similar to the reflection interface of the Koala component model, this method is also dealing with static resource consumption, since it is assumed that consumption of resources stays constant per operation. In reality, the former typically depends on parameters passed to operations and previous actions. In addition, the validity of the analysis results still depends on the scenario selection. Moreover, synchronization protocols for analyzing shared resource accesses are not supported. On the other hand, this approach fits very well within current system design practice, such as UML [10], where the dynamics of systems are modeled using scenarios.

```
service s1
  requires I2
  requires I3

provides I1 {
  operation f
    uses I2.g
    uses I3.h

  resource cpu
    require 1200
```

```
resource memory
  claim 100
  release 100

behavior
  operation f calls:
  I2.g*;
  I3.h}
```

Figure 9.3: Example of a Robocop component's service specification.

The research reviewed so far has been mainly dealing with estimating static memory usage. In many meaningful platform dependent applications, the problem of dynamic resource allocation is acute. Huh et al. [11] address such problem and solve it via *dynamic load balancing* techniques: in case the feasibility test fails, one can either increase the available budget of the current host, or migrate the application to the next best available host. However, this is mostly applicable on distributed, heterogenous systems.

9.3.2 UML-based Analysis

Low-level, code-driven resource estimates are invaluable when one has access to the implementation of the components, and especially when the components conform to a particular model. Nevertheless, more abstract descriptions of the expected resource usage are also needed in cases of not-yet implemented components or when the designer has to select components from existing repositories, and adapt them to fit the design.

Such abstract descriptions have to not only state what and how many resources are needed, but they should also include information of when and for how long must the resources be available. This extra requirement calls for system *specification* languages. The latter range from fully formal temporal logics [4] and process algebras [12], to the less formal, yet widely used, Unified Modeling Language (UML) [10]. Let us look at some of the UML-based attempts to tackle the analysis of embedded resources.

Baum et al. [13] present a structured approach of describing resource-usage scenarios. For this, they distinguish between two basic classes of resources: *timed-shared* and *space-shared*. Baum argues that any technique for modeling resource-usage scenarios has to consider three description aspects: service requirements, service provision and resource interaction. Service provision captures the characteristics of the services offered by the resource, whereas service requirements describe the resource's demands. Finally, resource interaction links service requirements with provisions. However, such a modeling approach lacks the ability to extend towards a formal description that could provide us with more accurate resource reasoning results.

The UML profile for Schedulability, Performance and Time (UML/SPT) [14] is a framework for modeling concurrency, resources and timing concepts, which eventually produces models for schedulability and performance analysis. From the user's point of view, UML/SPT provides a set of stereotypes and related tag values (i.e., "attributes" in UML 2.0) that can be used by the modeler for the annotation of the model elements and for performing analysis. The core of the profile is the *General Resource Modeling* (GRM) framework. The GRM describes resource types, their static and dynamic interaction with the system, and their management. Each resource offers services for which the effectiveness or quality of service (QoS) is measured. One advantage of the framework is that both static and dynamic resource requirements can be checked against; the disadvantage is the lack of a unique semantical interpretation.

Resources can also be modeled within UML-based simulative environments [15]. For this, Amar et al. extend the UML notation with new stereotypes for performance related items: resource types. The software architecture and the resources that the software components require are both represented in the same capsule diagram, which is split in two parts: the *software* side and the *resource* side.

We exemplify the modeling idea of such an approach through the following example.

Example: A Simple Light Switch System. Consider an ES composed of a display and a fan component, which are turned on/off by the same switch [16]. The software that implements both the light display's and the fan's behaviors utilize memory and processor computational resources.

The software architecture is described by the display and the fan capsules, and the resources that these components require are represented by two more capsules: the memory and the processor. (see Figure 9.4). The behaviors of the light display, the switch and the fan are depicted in Figure 9.5. The resource requests issued by the display and the fan application software include the amount of memory/processor needed by each to execute the respective software block.

Suppose that the display application needs 300 bytes of memory and a share

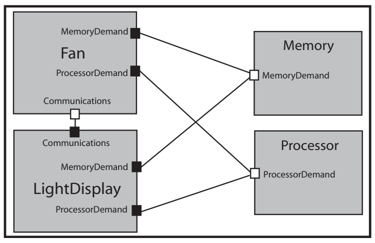


Figure 9.4: Two-sided capsule diagram.

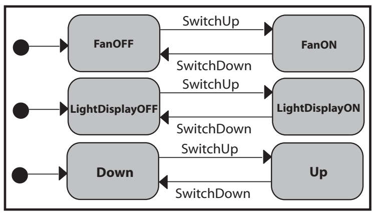


Figure 9.5: Behavioral diagrams.

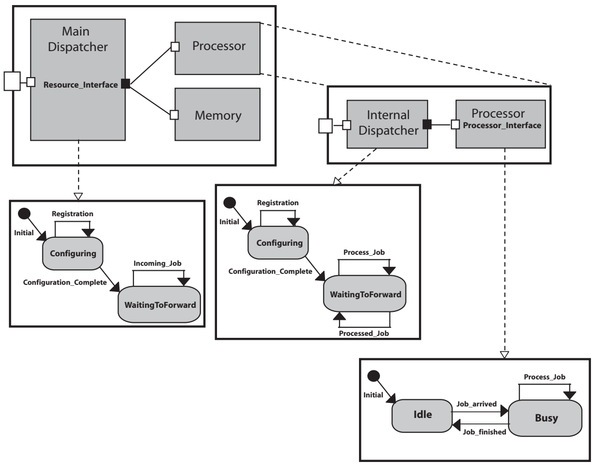


Figure 9.6: Resource model.

of 25% processor time to be turned (and while) "on", and just 100 bytes of memory and 5% processor time to be turned "off" (similar for the software that controls the fan). The resource side is composed by a *Main Dispatcher* and two resource types: memory and processor. In Figure 9.6 three new stereotypes have been introduced as capsules: a high-level Main Dispatcher, a low level *Internal Dispatcher*, and a *Processor resource*. The state diagrams of these stereotypes can be seen in the lower part of Figure 9.5. The processor is modeled by a simple state diagram: upon a job's arrival the system is in "busy" state, returning to "idle" when no job is to be served anymore. When the requested memory/processor share, needed for turning on or off the display and/or the fan, has been consumed, a notification is sent to the internal dispatcher and then forwarded to the main dispatcher. The latter checks whether the completed resource request has been satisfied, e.g. whether 300 bytes of memory have been provided to turn on the display, or if the processor utilization has been no more than 5%, for turning the same display off.

Although graphical and intuitive, the above approach is based on a simulative environment, hence one can not entirely guarantee the feasibility of the architecture, but rather provide a partial answer.

9.3.3 Formal Reasoning on Embedded Resources

Process Algebraic Approaches. In an attempt to unify formal modeling and analysis of ES resources, Lee et al. [7, 17, 18] open a new gate: they propose a family of process-algebraic formalisms that can theoretically account for various resource types. The family relies on algebra of communicating shared resources (ACSR), a discrete-time process algebra that extends classical process algebras with the notion of resource. The starting point of the modeling is the introduction of a resource as a *generic*, *first-class* modeling entity. This comes closer to our wish of being able to correlate various analysis results at different abstraction levels. The authors characterize the resource by a set of attributes, such as timing parameters, probability of failure (π, assumed constant), priority (pr, variable), power consumption (pc, variable) etc., which capture the resource's behavior. For instance, a class of resources that may experience failures, consume power and whose use can be regulated by properties is captured by the following model:

$$R\colon [\pi :< [0,1] : \mathsf{stat} >, pc :< \mathsf{int} : \mathsf{dyn} >, pr :< \mathsf{int} : \mathsf{dyn} >]$$

The authors consider sets of resource classes deemed useful 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. In addition, the general framework is instantiated to several progressively more complex application domains [18].

A first instantiation is Milner's calculus of communicating systems (CCS) [12], in which the resource constraints are equated to just communication constraints between concurrent processes. For example, one can formally enforce model correctness by composing in parallel two processes that send and receive on the same channel. Such an analysis is pretty restricted, as we need to account for other types of resources, as well. Consequently, the authors proceed to extending CCS towards ACSR, which considers time and priorities as resource attributes. An important restriction is the assumption that each action takes exactly one time unit, and that only one process may use a resource during a time step. However, the extension allows for more complex formal analysis, such as correct time reservation of concurrent processes, based on their synchronous execution. The semantic translation of the model gives rise to a transition system that captures the nondeterministic behavior of processes.

Fault-tolerance analysis of embedded real-time systems can be carried out within probabilistic resource failure in real-time process algebra (PACSR), the probabilistic extension of ACSR [18]. Last but not least, memory use is captured as a shared resource among concurrent processes, in the multi-capacity resources algebra (MCSR). Multi-capacity resources are introduced as a new class with two attributes: the capacity of a resource and the memory used by a process during one execution step. Such a rich resource model facilitates reasoning about the effects of reducing the memory use of a process at the expense of its longer execution.

Although the ACSR framework is theoretically rich, the resource analysis is not correlated with the steps towards ES deployment; the verification is independent of the design stage, which makes it difficult to actually use the gained information, when allocating components to the hardware units.

Algorithmic Methods. Quantifying resource usage, such as power consumption, size of message queues, net profit etc., can be done by augmenting *Discrete Time Markov Chains* (DTMCs) with real-valued quantities, called *costs*/*rewards*, assigned to states and/or transitions [6]. Properties to be verified are expressed in a probabilistic temporal logic (PCTL) extended with reward operators (R). Quantitative verification involves a combination of the traversal of the state-transition graph of the model and numerical computation.

If we consider the light switch example, the properties that could be verified with DTMCs, are:

$$\begin{array}{ll} \mathsf{R}_{\leq 300} & [\mathsf{C}^{\leq 150}] \\ \mathsf{R}_{\leq 5} & [\mathsf{F}\,(\mathsf{light} = \mathsf{off})] \end{array}$$

The first property says that the expected memory consumption within the first 150 time-steps of operation is less than or equal to 300. The second formula states that the expected processor share when the state “light = off” is reached is no more than 5. DTCMs are not really suitable for modeling real-time systems, since there is no notion of real-time, though reasoning about discrete time is possible through state variables “counting” transition steps [6].

A continuous-time approach to analyzing resource consumption is provided by the *Priced Timed Automata* (PTA) [19] framework. PTA are proper extensions of Timed Automata, with cost information on both locations and transitions. Although suited for real-time system modeling, PTA allow mainly continuous, monotonically increasing consumption of resources (e.g. energy) to be modeled and analyzed. How could one then handle non-monotonic resource models (e.g. memory), along with reasoning about, say, energy consumption? The solution might require employing *multi-priced* TA [20], which are PTA with multiple cost variables evolving according to given rates for each location. Even though multi-priced TA are already on the market, a general, unified PTA-based resource model is still missing, as are component-oriented algorithms for verification.

Correct-by-Construction Techniques. The issues of how to deal with reusable resources systematically, and how to convert a program into one requiring less resources may become mind-boggling if systems are complex and heavily resource-constrained. Naiyong and Jifeng [21] address these problems and introduce a resource calculus where comparing two programs that consume/reuse resources is possible. The algebraic laws include program transformation rules that let the designer change the initial program into a less-resource-requiring one. The limitations of the method stays in the fact that the proof-system is not proved complete and program iterations are not considered. Besides, real-time systems can not be covered, since timing information is missing from the models. Even if not component oriented, the approach sheds a light on the meaning of resource-wise program refinements.

A related class of correct-by-construction techniques is focused on the use of *component interfaces* [5]. A well-designed interface exposes exactly the information about a component which is necessary to check for composability with other components. In a sense, an interface formalism is a “type theory” for component composition. As we have seen in the above, recent trends are towards rich interfaces, which expose extra-functional information about a component, like resource consumption levels, besides the functional aspects. Interface theories are especially promising for incremental design under such quantitative constraints, because the composition of two or more interfaces can be defined as to calculate the combined amount of resources that are consumed by putting together the underlying components.

Timed Abstract State Machines (TASM) Approach. *Timed Abstract State Machines* (TASM) is a unified formalism for the specification of functional and non-functional properties of ES [16]. The model is made of two parts, a timed ASM and an environment. Resources are defined at the environment level, such that when a machine executes a step, the updated set of controlled variables contains the step duration and the amounts of resources consumed during the respective step execution. Hierarchical structures and parallel compositions of TASMs are supported, which makes the framework applicable to resource analysis of more complex ES. However, the model seems inadequate for modeling more detailed resource descriptions, since the resource information is a simple annotation, in the form of a real-valued variable assignment. Another limitation is the impossibility to carry out trade-off analysis of conflicting resource requirements.

9.4 Our Vision of Resource-aware ES Design

In order to be able to synthesize a predictable ES from components and compositions, a *resource-aware* design framework is a must.

Let us assume that the ES under design is real-time, and it is built from components existing at one of the following three levels of granularity (see Figure 9.7): *subsystem components* (coarse grained with restricted inter-subsystem interaction capabilities), *architectural components* (elements providing an internal decomposition and interconnection structure of subsystems), or *software components* (containers of software with specific interfaces and properties) [22].

Predictability amounts to establishing that the total, worst-case resource usage, in terms of memory, bandwidth, power etc., is within the bounds given by the resources of the selected execution platform. The employed predictability analysis should guide the design and selection of components, as well as the design and selection of the hardware and system software.

Our vision (Figure 9.7) relies on two pillars: *early stage resource-usage analysis* based on *abstract system models* (at subsystem and architectural levels) and *platform assumptions* (see virtual architecture in the figure), and *later*

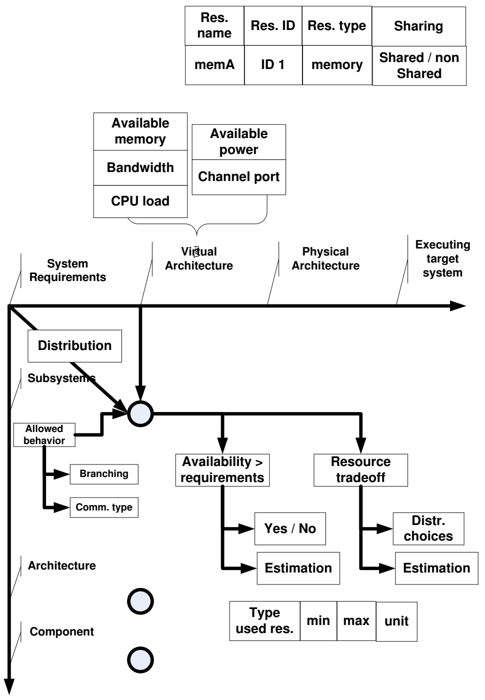


Figure 9.7: Resource analysis within our proposed ES design flow (courtesy of Severine Sentilles).

stage (when a specific system instance to be executed on a specific target is known) *verification* that the assumptions underlying the early analysis are still valid.

The common model-based development approach for ES assumes a "one-way" process, starting from requirements, followed by platform independent system models (PIM), platform specific system models (PSM), and final implementation. As opposed to such a method, we advocate a flexible (iterative) resource-aware analysis framework for real-time ES, which is tightly correlated with the deployment process, from the beginning. The correlation is ensured by the notion of *virtual architecture* on which components can be (virtually) mapped to [22]. The virtual architecture provides an abstraction of the targeted platform, which is gradually added with detail, at the same time with increasing the detail of the system representation. The elements of the virtual architecture (nodes and networks) have resource attributes (memory and power budgets per node, CPU load per node, bandwidth of the communication network, etc.) that allow feasibility analysis, w.r.t resource usage, at different levels of abstraction:

- **Subsystem/Architectural/Software Component-level Resource Analysis (Early stage analysis)**. Suppose that the components $C_1, \ldots, C_n$ of the motivating example (Section9.2) are either subsystems or software components (SC) mapped onto the virtual nodes. We can assume, at the beginning, a one-to-one mapping. Various design solutions can be investigated for feasibility, by checking whether the resource demands of the subsystems/SCs are smaller than the available ones of the virtual nodes, respectively.

  Early analysis requires a general, unified theoretical framework (like in [19], or [7]) for composing resource-usage models of storage, computation and communication resources. The model should be unified especially for being able to perform *trade-off* analysis between apparently conflicting resource requirements: memory vs. execution time, energy vs. memory etc. If we assume a PTA formal real-time ES model, trade-off analysis would require *multi-objective* model-checking algorithms. However, creating the suitable mathematical framework is work-in-progress, and involves other people than the authors too, hence we just sketch the initial ideas below.

  Within the PTA model, we can encode the notion of a *resource*, by constructing the weighted sum of all the objectives ($c_1, \ldots, c_n$), as the following cost function: $c = w_1{*}c_1{+}w_2{*}c_2{+}\ldots{+}w_n{*}c_n$. Here, $c_1, \ldots, c_n$ could describe, for instance, memory-usage cost, energy-consumption cost etc., whereas $w_1, \ldots, w_n$ (weights) could represent the relative importance of $c_1, \ldots, c_n$. The values of the weights are a subjective matter; the way they are chosen depends both on the application and on the analysis goals. For example, if we are designing a heavily resource-constrained soft real-time ES that might tolerate lateness at the expense of quality of service, and are considering trade-offs between memory consumption and (execution) time, we can assign higher weight to memory than to time. To derive the costs, one could apply static analysis techniques on the implementation of a previous version of the software component [23]. Finally, the ES resource-usage would then be described by equation: $\dot{c} = w_1 * \dot{c_1} + \ldots + w_n * \dot{c_n}$.

- **Task-level Resource Analysis (Later stage Verification)**. We now assume some grouping of components $C_1, \ldots, C_n$ into real-time tasks, which are assigned to virtual nodes. Next, virtual nodes are mapped onto physical nodes, according to the resource attributes assumed by the virtual architecture, which are now requirements that the physical architecture must satisfy. The later stage analysis requires a simple yet faithful (abstract) description of the mapping of components, grouped into real-time tasks, onto hardware units.

  If, on top of the hardware abstraction used in the early analysis, we also assume a simple task model (deadline, worst-case execution time, offset, priority), we could roughly estimate resource-usage bounds per tasks, respectively, and their compositions.

The resource analysis process described above is iterative, allowing feedbacks between steps, in case the verification fails. This lets one narrow the space of design solutions, at early stages in the design flow.

We hope that the reader will regard the above arguments as an incentive to looking at more practical ways of incorporating resource information into ES models.

Acknowledgements. The authors thank Séverine Sentilles for lending them Figure 9.7. This work was performed within the Swedish national strategic research centre PROGRESS, supported by the Swedish Foundation for Strategic Research (SSF) and Mälardalen University.

Bibliography

[1] D.K. Hammer M.R.V. Chaudron E.M. Eskenazi, A.V. Fioukov. Estimation of Static Memory Consumption for Source Code Components. In *Proceedings of Composing Systems from Components Workshop*, 2002.

[2] A. V. Fioukov, E. M. Eskenazi, D. K. Hammer, and M. R. V. Chaudron. Evaluation of Static Properties for Component-Based Architectures. *EUROMICRO Conference*, pages 33–39, 2002.

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

[4] P. Bellini, R. Mattolini, and P. Nesi. Temporal Logics for Real-Time System Specification. *ACM Comput. Surv.*, 32(1):12–42, 2000.

[5] Luca de Alfaro and Thomas A. Henzinger. Interface-based Design. In *Engineering Theories of Software-intensive Systems*, volume 195 of *NATO Science Series: Mathematics, Physics, and Chemistry*, pages 83–104. Springer, 2005.

[6] Marta Kwiatkowska. Quantitative Verification: Models Techniques and Tools. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 449–458, New York, NY, USA, 2007. ACM.

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

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

[9] H. Maaskant. A Robust Component Model for Consumer Electronic Products. In *Philips Research Book Series*, volume 3, pages 167–192. Springer Netherlands, 2005.

[10] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[11] Eui-Nam Huh, Lonnie R. Welch, Behrooz A. Shirazi, and Charles D. Cavanaugh. Heterogeneous Resource Management for Dynamic Real-Time Systems. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 287, Washington, DC, USA, 2000. IEEE Computer Society.

[12] Robin Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[13] Lothar Baum and Thorsten Kramp. Towards a Uniform Modeling Technique for Resource-Usage Scenarios. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1324–1329. CSREA Press, 1999.

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

[15] Hany H. Ammar, Vittorio Cortellessa, and Alaa Ibrahim. Modeling Resources in a UML-Based Simulative Environment. In *AICCSA '01: Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications*, pages 405–410, Washington, DC, USA, 2001. IEEE Computer Society.

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

[17] Insup Lee, Jin-Young Choi, Hee-Hwan Kwak, Anna Philippou, and Oleg Sokolsky. A Family of Resource-Bound Real-Time Process Algebras. In *FORTE '01: Proceedings of the IFIP TC6/WG6.1 - 21st International Conference on Formal Techniques for Networked and Distributed Systems*, pages 443–458, Deventer, The Netherlands, The Netherlands, 2001. Kluwer, B.V.

[18] Insup Lee, Anna Philippou, and Oleg Sokolsky. A General Resource Framework for Real-Time Systems. In *Radical Innovations of Software and Systems Engineering in the Future, 9th International Workshop*, volume 2941 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 2002.

[19] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, Judi Romijn, and Frits Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. In *Proceedings of the 4th International Workshop on Hybris Systems: Computation and Control*, pages 147–161. Springer–Verlag, 2001.

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

[21] J. Naiyong and H. Jifeng. Limited Resource Models and Specifications for Programming Languages. UNU/IIST Report. Number 277, 2004.

[22] I. Crnkovic H. Hansson and T. Nolte. The World according to PROGRESS, Draft paper, 2008.

[23] 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. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1307–1314, New York, NY, USA, 2007. ACM.

Chapter 10

Paper D: REMES: A Resource Model for Embedded Systems

Cristina Seceleanu, Aneta Vulgarakis, and Paul Pettersson
In Proceedings of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009), IEEE CS, Potsdam, Germany, June, 2009.

Abstract

In this paper, we introduce the model REMES for formal modeling and analysis of embedded resources such as storage, energy, communication, and computation. The model is a state-machine based behavioral language with support for hierarchical modeling, resource annotations, continuous time, and notions of explicit entry and exit points that make it suitable for component-based modeling of embedded systems.

The analysis of REMES-based systems 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. To formalize these problems and provide a basis for rigorous analysis, we show how to analyze REMES models using the framework of priced timed automata and weighted CTL. To illustrate the approach, we describe a case study in which it has been applied to model and analyze resource-usage of a temperature control system.

10.1 Introduction

The importance of resource awareness in embedded systems is growing rapidly [1–7]. The limited availability of computing resources is preventing the introduction of new product features and applications, especially in areas where high-performance embedded systems are required. Resources include energy, computational power, memory, and hardware components such as buses, input/output ports, etc.

The systematic analysis of the resource consumption of an embedded system must include ways of semantic representation of various types of resources, be they of continuous type (like energy), or of discrete nature (e.g. memory, I/O ports). A representative analysis goal is to verify the *resource-wise feasibility* property. 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.

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. Alternatively, if one assumes a repository of models, the designer might need, at some point, to replace a component model with a refined one, having modified behavior or more efficiently implementable data structures.

We would be interested to assess, before deployment, how would any design decision, such as, reallocation of components to hardware units, or replacement of components, affect the system's overall resource consumption. This amounts to finding an appropriate trade-off between different configuration requirements and constraints.

The 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.

In this paper, we propose a modeling framework and apply associated analysis techniques for performing quantitative analysis such as feasibility, trade-offs, and optimal/worst-case resource consumption analysis. The model, called REMES, is tailored for embedded systems, but it is generally suitable for reactive systems. It provides support for 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. A number of generic resources can be modeled in this way, including memory, ports, energy, CPU, and buses. As such, the characterized and classified abstract resource types are not tied to any particular formal semantic interpretation.

In brief, our contribution is threefold:

- A classification of embedded resources, based on their rate of consumption over time, and the attribute of being referable, or not (section 10.3.1).
- The behavioral language REMES (sections 10.3.2, 10.3.3).
- Encoding the resource-wise analysis problem as a weighted sum of consumed resources (section 10.4.1).

Since REMES allows the description of functionality and timing in a dense time state-based modeling language, we also show how the latter and a number of important resource analysis problems can be formalized in the framework of (multi-)priced timed automata (sections 10.4.2, 10.4.3, 10.4.4).

The main purpose of REMES is to narrow the gap between architectural modeling (e.g., architecture description languages (ADLs) [8]) and formal analysis models (such as priced timed automata [9, 10]). This claim is supported throughout a case study presented in section 10.5, in which a temperature control system is modeled and analyzed. Following the example, we discuss our approach and compare to related work in section 10.6, then conclude the paper and present a line of future work, in section 10.7.

10.2 Preliminaries

10.2.1 Priced Timed Automata

In the following, we recall the model of priced (or weighted) timed automata [9, 10], an extension of timed automata [11] with prices/costs on both locations and transitions.

Let X be a finite set of clocks and $\mathcal{B}(X)$ the set of formulas obtained as conjunctions of atomic constraints of the form $x \bowtie n$, where $x \in X$, $n \in \mathbb{N}$, and $\bowtie \in \{<, \leq, =, \geq, >\}$. The elements of $\mathcal{B}(X)$ are called *clock constraints* over X.

Definition 1. *A linearly Priced Timed Automaton (PTA) over clocks X and actions Act is a tuple* (L, l_0, E, I, P)*, where L is a finite set of locations,* l_0 *is the initial location,* $E \subseteq L \times \mathcal{B}(X) \times Act \times \mathcal{P}(X) \times L$ *is the set of edges,*

$I : L \to \mathcal{B}(X)$ assigns invariants to locations, and $P : (L \cup E) \to \mathbb{N}$ assigns prices (or costs) to both locations and edges. In the case of $(l, g, a, r, l') \in E$, we write $l \overset{g,a,r}{\to} l'$.

The semantics of a PTA is defined in terms of a timed transition system over states of the form (l, u), where l is a location, $u \in \mathbf{R}^X$, and the initial state is (l_0, u_0), where u_0 assigned all clocks in X to 0. Intuitively, there are two kinds of transitions: delay transitions and discrete transitions. In delay transitions,

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

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

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

correspond to taking an edge $l \overset{g,a,r}{\to} l'$ for which the guard g is satisfied by u. The clock valuation u' of the target state is obtained by modifying u according to updates r. The cost $p = P((l, g, a, r, l'))$ is the price associated with the edge.

A timed trace σ of a PTA is a sequence of alternating delays and action transitions

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

and the cost of performing σ 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).

A network of PTA $A_1|| \dots ||A_n$ over X and Act is defined as the parallel composition of n PTA over X and Act. Semantically, a network again describes a timed transition system obtained from those components, by requiring synchrony on delay transitions and requiring discrete transitions to synchronize on complementary actions (i.e. $a?$ is complementary to $a!$).

In order to specify properties of PTA, the logic Weighted CTL (WCTL) has been introduced [12]. WCTL extends Timed CTL with resets and testing of cost variables. We refer the reader to [12] for a thorough introduction of WCTL.

10.2.2 Multi Priced Timed Automata

An extension of PTA is the class of Multi Priced Timed Automata (MPTA) in which a timed automaton is augmented with more than one cost variable [12, 13]. 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) reaching a goal state. Note that the solution is a set of pairs, rather than a single pair, since the costs contributed from the individual costs can be incomparable, i.e., if for two traces (p_1, p_2) and (p_1', p_2') e.g., $p_1' < p_1$ and $p_2 < p_2'$. In this setting, the minimal cost reachability problem is to find the set of pairs with minimum cost reaching the goal state. Dually, the maximization problem is defined as finding the set of 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 [13] for a thorough description of optimization problems in MPTA.

10.3 REMES: The Proposed Resource Model

In this section, we define the resources of interest and introduce the model REMES intended for resource modeling and analysis.

10.3.1 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. We also classify resources depending on whether they are *referable* or *non-referable*. A representative example of a referable resource is the memory. Memory can be dynamically allocated, deallocated, addressed, and manipulated during run-time.

Taking all these into consideration, Table 10.1 shows three identified resource classes and their characteristics of interest. Resource consumption for resources that belong to class C is continuous, which is in opposition to the discrete resource consumption nature for the resources from class A and B. The consumption of the CPU can be modeled either by a discrete variable, denoting the number of accumulated clock ticks, or by a continuous variable, 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 either

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

Table 10.1: Resource classes/characteristics

class B or C (same applies to bandwidth). Only the resources from class A are referable and can be dynamically manipulated.

10.3.2 Introducing REMES

Our **RE**source **M**odel for **E**mbedded **S**ystems (REMES) is intended to describe the resource-wise behavior of interacting embedded components. REMES relies heavily on the modeling language CHARON [14], used for specifying embedded systems as communicating agents. Our main contribution is the addition of information regarding resource consumption and its rate, as well as other constructs that facilitate the application of REMES to modeling both functional and extra-functional behavior of (real-time) component-based systems.

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 10.1). 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 10.1, that the entry and exit points are drawn as blank and filled circles, respectively. The variables of mode M are partitioned into *local* variables, ($\mathsf{L_M}$), and *global* variables ($\mathsf{G_M}$), and can be of types boolean, natural, integer, array, or of an extra type clock that specifies continues variables evolving at rate 1. The global variables are in turn partitioned into *read* variables, $\mathsf{Rd_M}$, and *write* variables, $\mathsf{Wr_M}$, such that $\mathsf{G_M} = \mathsf{Rd_M} \cup \mathsf{Wr_M}$.

Read/Write Variable Accesses. The local variables of M, $\mathsf{L_M}$, can not be read or written by other modes, the set $\mathsf{Wr_M}$, written by M can be read by other modes, whereas the set $\mathsf{Rd_M}$ may be written by other modes. The sets $\mathsf{Wr_M}$

and $\mathsf{Rd_M}$ need not be disjoint; concurrent access to common write variables of modes can be regulated by specifying certain synchronization protocols in the REMES model.

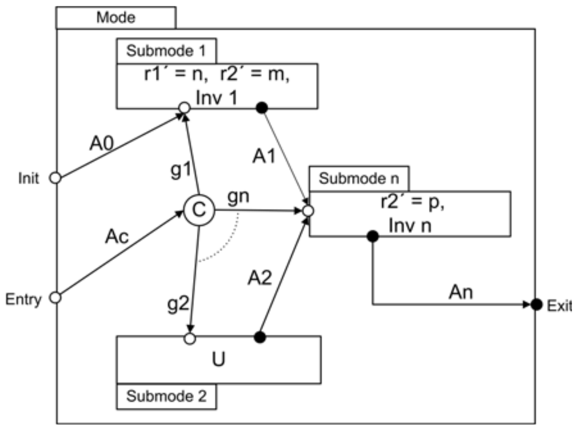


Figure 10.1: A REMES Composite Mode.

The atomic modes Submode 1 and Submode n in Figure 10.1 are annotated with their respective resource-wise continuous behavior, assuming that the corresponding component is consuming resources $(r1, r2)$ belonging to class C. Such consumption is expressed by the first derivatives of the typed resource variables $r1, r2 : T_C$, respectively, $r1', r2'$, 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 10.1, the composite mode takes the edge labeled A_0, in order to enter Submode 1, after initialization, and similarly, edges labeled $A_1, A_2, \ldots, A_n$, to further enter Submode 1, Submode 2, . . ., Submode n, respectively.

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 is decorated with letter U, 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 $A = (g, S)$ 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., Inv 1,..., 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.

Similar to Statecharts [15], REMES provides a *conditional connector* (depicted by C in Figure 10.1), which allows the selection of an outgoing edge, out of two or more possible ones, via the guarding boolean conditions (guards $g_1, g_2, \ldots, g_n$) 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.

We classify a mode's edges, and afferent discrete actions, as follows:

- *entry edges*: connect an entry point of the composite mode with the entry point of a submode (e.g., annotated with action A_0);
- *exit edges*: connect the exit point of a submode with the exit point of the composite mode (e.g., annotated with action A_n);
- *entry conditional top edges*: connect an entry point of the composite mode with a conditional connector (e.g., annotated with action A_C);
- *exit conditional top edges*: connect a conditional connector with the exit point of the composite mode;
- *entry conditional sub edges*: connect a conditional connector with the entry point of a submode;

- *exit conditional sub edges*: connect the exit point of a submode with a conditional connector;
- *internal edges*: connect the exit point of a submode with the entry point of another submode (e.g., annotated with actions A_1, A_2).

The control points of submodes should be connected by edges, such that the termination of the internal behavior of the composite mode is ensured (e.g., in case cycles exist, termination can be guaranteed by decreasing a chosen termination function each time the cycle is executed). Note that, in REMES, each mode describes the behavior of a component, and a composite mode is a way of encapsulating behavior and it describes a composite component.

Formal Definition of a Mode. A mode M is a tuple $(\mathsf{SM}, \mathsf{V}, \mathsf{In}, \mathsf{Out}, \mathsf{E}, \mathsf{RC}, \mathsf{Inv})$, 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, and, finally, Inv is the set of invariants.

For the submodes of M, the condition $\mathsf{G}_{\mathsf{SM}} \subseteq \mathsf{L}_{\mathsf{M}} \cup \mathsf{G}_{\mathsf{M}}$ should hold, for a local variable of a mode to be accessible only in its submodes, and not anywhere else.

Mode Execution. The top-level mode, which is activated when a corresponding event is received, enters execution for the first time through the special Init entry point, while 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 10.1, the execution of Mode 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 $\mathsf{Inv\ 1}$ 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 $\mathsf{Inv\ n}$ does not hold anymore, the mode takes a new discrete step corresponding to discrete action A_n. The next time when the control is passed to Mode, a discrete step corresponding to A_C is taken and the selection of a possible path is made through the conditional connector, etc.

10.3.3 Composition of REMES models

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: $\mathsf{L_M} \subseteq \cup_{\mathsf{sm} \in \mathsf{M}} \mathsf{G_{sm}}$, and $\mathsf{G_M} = \cup_{\mathsf{sm} \in \mathsf{M}} \mathsf{G_{sm}} - \mathsf{L_M}$. The mode composition is defined as follows.

Definition 2. *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* $\mathsf{Mode_A} \cup \mathsf{Mode_B}$.

In Definition 2, the parallel composition of composite modes subsumes the reunion of all the constituent submodes, corresponding edges and associated actions.

10.4 Analyzing REMES-based Systems

10.4.1 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:

$$r_{tot} \triangleq w_1 \times r_1 + w_2 \times r_2 + \ldots + w_n \times r_n,$$

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 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.

Informal Translation of REMES **into PTA.** 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 as the underlying semantic representation.

Informally, translating REMES into 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 transition in PTA's semantics. An atomic mode represents a PTA location, and global variables used for passing control in REMES become synchronization channels in PTA. The formal translation of the hierarchical REMES into a network of PTA and the associated tool are subject to future work. Next, we formalize some of the main analysis goals that we are interested in.

10.4.2 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 1 as the semantic translation of a REMES model, feasibility goals can then be formalized as the following WCTL properties that the PTA model can be checked against:

$$A\,F_{cost \leq n}\,v \tag{10.1}$$
$$A\,G\,(q \Rightarrow A\,F_{cost \leq n}\,v) \tag{10.2}$$
$$E\,F_{cost \leq n}\,v \tag{10.3}$$
$$A\,G\,(q \Rightarrow E\,F_{cost \leq n}\,v) \tag{10.4}$$

where G and F are the WCTL temporal operators "always" and "eventually", respectively [12].

The above properties are in fact liveness properties (10.1), (10.2), (10.4), and a reachability property (10.3), indexed by cost constraints. The first two properties specify *strong feasibility*: property (10.1) 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 (10.2) 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 (10.3) models *weak feasibility*: the target location v may be reached within a total cost of n. Finally, property (10.4) 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 [16]. For other PTA, one can only verify reachability properties of the form given by (10.3).

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.

10.4.3 Optimal and Worst-Case Resource Consumption

Optimal and worst-case resource consumption analyses 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 [13]. 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 example of section 10.5.

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 [17].

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.

10.4.4 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 10.1, 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 [18], 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 [18].

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.

10.5 Example: A Temperature Control System

As a case study (taken from [19]) demonstrating the principles of our resource modeling and analysis approach, we consider a temperature control system (TCS) for a heat producing reactor. It has two rods that can be inserted into the core of the reactor, to control the heat producing (chain) reaction. 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 of the TCS is to maintain the temperature in the reactor core between θ_{min} and θ_{max}. Whenever the core reaches temperature θ_{max}, it has to be cooled with one of the two rods. After a rod has been used for cooling, it is then unavailable for T time units.

10.5.1 A REMES Model of TCS

We model an abstracted version of the internal design of TCS in the SaveComp component model [20], with three components: HC controller, Rod selector, and Clock as depicted in Figure 10.2.

The interfaces of the components are described in terms of ports. SaveComp distinguishes between input and output ports, which can be of the types: *data* for transferring of data, *triggering* to trigger component executions, or *combined* to combine the two.

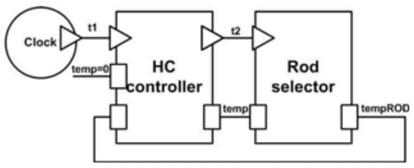


Figure 10.2: Component based TCS model.

The component HC controller activates the heating/cooling process of the core using trigger port t2. The Rod selector uses temperature data of the core conveyed through data port temp to control whether the core should continue to heat, or if a rod should be selected for insertion into the core to slow down the reaction. The latter must take the availability of rods into consideration, as a rod has to rest for at least T time units after its previous use. Finally, the Clock component, periodically generates the trigger event t1 that activates the HC controller. The temp value in the HC controller is updated by reading the value of variable tempROD that is assigned the cooling rate of the rods within the Rod selector component.

We model the resource usage of the TCS components as modes in REMES. The modes of the Clock, the HC controller, and the Rod selector are depicted in Figure 10.3, 10.4, and 10.5, respectively.

The modes communicate data between each other using the global variables: temp, tempROD, t1, and t2. The modes of HC controller and the Rod selector are made of submodes, conditional connectors, and edges, as described in Section 10.3.

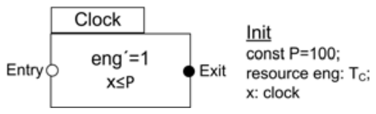
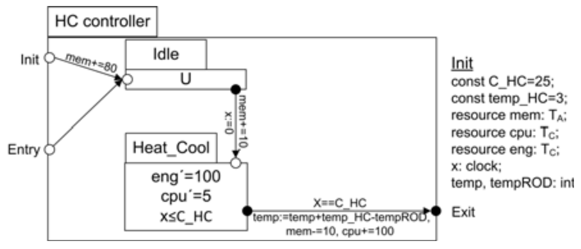


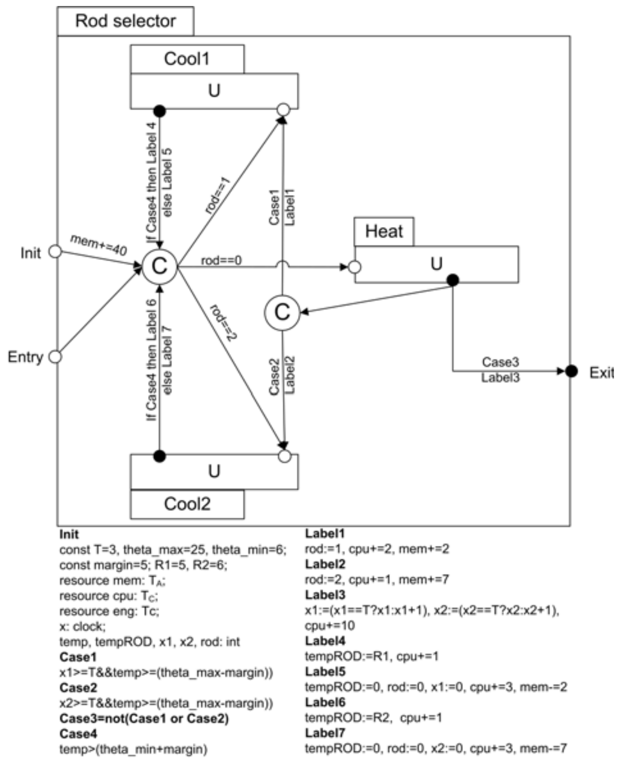
Figure 10.3: The Clock modeled in REMES.

In the TCS model, we make use of three resources: memory, energy, and CPU, which belong to two different classes of the taxonomy presented in Section 10.3.1. We assume that every simple cpu instruction utilizes one cpu tick. 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.



```
Init
const C_HC=25;
const temp_HC=3;
resource mem: T_A;
resource cpu: T_C;
resource eng: T_C;
x: clock;
temp, tempROD: int
```

Figure 10.4: The HC Controller modeled in REMES.

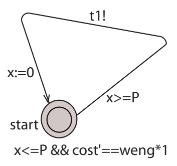


```
Init                                          Label1
const T=3, theta_max=25, theta_min=6;         rod:=1, cpu+=2, mem+=2
const margin=5; R1=5, R2=6;                   Label2
resource mem: T_A;                            rod:=2, cpu+=1, mem+=7
resource cpu: T_C;                            Label3
resource eng: Tc;                             x1:=(x1==T?x1:x1+1), x2:=(x2==T?x2:x2+1),
x: clock;                                     cpu+=10
temp, tempROD, x1, x2, rod: int               Label4
Case1                                         tempROD:=R1, cpu+=1
x1>=T&&temp>=(theta_max-margin))              Label5
Case2                                         tempROD:=0, rod:=0, x1:=0, cpu+=3, mem-=2
x2>=T&&temp>=(theta_max-margin))              Label6
Case3=not(Case1 or Case2)                     tempROD:=R2,  cpu+=1
Case4                                         Label7
temp>(theta_min+margin)                       tempROD:=0, rod:=0, x2:=0, cpu+=3, mem-=7
```

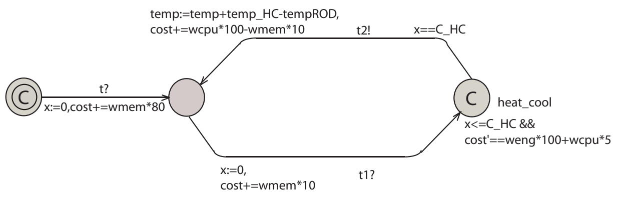
Figure 10.5: The Rod selector modeled in REMES.

10.5.2 A PTA model of TCS

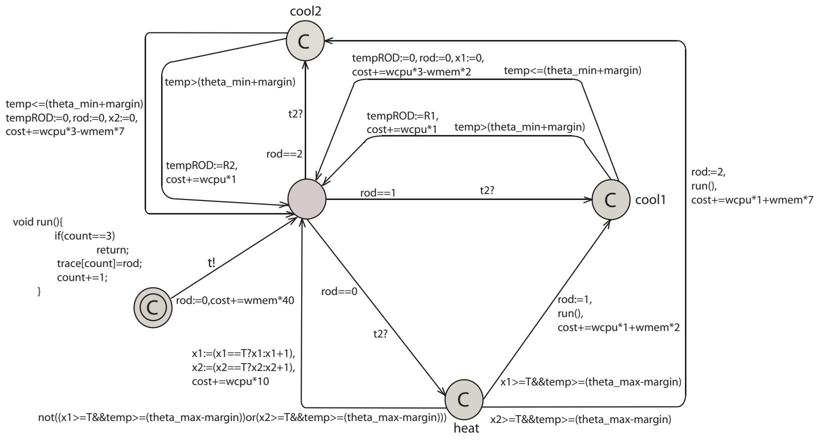
We have analyzed the REMES-based TCS system, as a network of three PTA models, in UPPAAL CORA[1]. The PTA models of the Clock, the HC controller, and the Rod selector are shown in Figure 10.6.



(a) The model of the clock component as a PTA.



(b) The model of the HC controller component as a PTA.



(c) The model of the Rod selector component as a PTA.

Figure 10.6: TCS modeled with three PTA.

The Clock is modeled as a simple PTA that, after every P time units, periodically synchronizes on channel t1 with HC controller. The HC controller PTA has three locations: Start, Idle, and Heat_Cool. The constant C_HC is the execution time of the HC controller. The difference (temp_HC −

[1]See the web page www.uppaal.org/cora for more information about the UPPAAL CORA tool.

tempROD), where temp_HC is the heating produced by the reactor, and tempROD is the current cooling effect of the rod, is used to update the reactor temperature.

The PTA Rod selector has five locations: Start, Select, Heat, Cool1, and Cool2. The execution of the Rod selector consumes 40 units of static memory. The locations Start, Heat, Cool1, and Cool2 are committed, as their actions are atomic. The synchronization with HC controller is modeled using channel t2. The selection of the rods is controlled by variable rod. From location Heat, based on the temperature of the core, temp, and the time since a rod has been previously used for cooling (i.e., x1 and x2 for rod1 and rod2, respectively), an available rod is selected for insertion into the core, and, consequently, the Rod selector enters location Cool1 or Cool2, or alternatively jumps back to location Select, provided that no rod needs to be used.

For analysis purposes, we have added the TCS model with the function run() (see Figure 10.6(c)) that merely stores the first few selections of rods, in an array of integers.

10.5.3 Formal Analysis of the PTA model

In the analysis model, we have encoded the relative importance of the resources energy, CPU, and memory. We consider CPU to be the most critical resource, followed by memory. Energy is not critical, yet it is taken into consideration in order to ensure higher energy efficiency in the system. Therefore, we give highest weight to CPU and lowest to energy. The cost of resource usage is influenced by the individual weights of each resource, and the consumed (utilized) resource on each transition (location). Currently UPPAAL CORA can only handle PTA models where the cost function is monotonically increasing. This means that in order to keep the cost function monotonically increasing we have to fine-tune the weights of the resources.

In the TCS system, we consider the following total cost function

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

where wcpu = 15, wmem = 2, and weng = 1, and c_{cpu}, c_{mem}, and c_{eng} are the accumulated consumed amounts of cpu, memory, and energy, respectively.

After having fed UPPAAL CORA with the PTA model of the TCS, we were able to analyze the minimum cost reachability problem, that is, to compute the lowest cost of satisfying a given reachability property, and a corresponding trace. However, we have first checked the model against the safety properties: $\mathsf{AG\,temp} \leq \mathsf{theta}_{\mathsf{max}}$ and $\mathsf{AG\,temp} \geq \mathsf{theta}_{\mathsf{min}}$. In our case, we are 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.

To illustrate the technique, we check for an optimal trace satisfying the property: $\mathsf{EF}(\mathsf{count} == 3)$, that is, a trace in which rods are inserted into the reactor three times. UPPAAL CORA has found that the second rod should be inserted two times, followed by the first one, the third time. Table 10.2 shows the cost of this best trace, and also the cost of another more expensive trace where only rod 2 has been used.

For TCS, we can only partially tackle the trade-off resource analysis problem, by giving higher weight to the most critical resource, the CPU, followed by memory and energy. Additionally, we have conducted optimal reachability resource usage analysis, by minimizing the memory consumption, while imposing upper bounds on the CPU consumption, in the TCS. For example, for 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.

Scenario	Order of execution	Cost
1	P_2-P_2-P_1	127499
2	P_2-P_2-P_2	127509

Table 10.2: Cost of execution for different rod insertion scenarios.

10.6 Discussion and Related Work

The REMES model presented in Section 10.3 can be employed in the design of embedded systems, for representing the internal behavior of the interacting embedded components. As such, it complements architectural description languages (ADLs) [8], 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, we believe that REMES is simple enough to 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 PTA [9, 10]).

The cost analysis model proposed in Section 10.4.1 is platform-aware. Hence, as future work, it could benefit from including abstractions of platform specific tools, such as the associated compiler, linker etc. We do believe 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. [21]. 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.

Last but not least, we 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.

Related Work. In a recent study [22], Vulgarakis and Seceleanu have presented related work on modeling and analyzing resources in component-based embedded real-time systems, and they have grouped it into three categories, as follows.

First, research has been devoted to code-level resource modeling and analysis, in component assemblies. In Koala [3] and Robocop [2] component frameworks, static memory estimation has been performed for applications in which the instantiated components of a composition are known prior to runtime. Such low-level code-driven resource estimates can only be used in cases when one has access to the components implementations. More abstract descriptions of expected resource usage may be needed for not-yet implemented components, or for guiding the selection of components from the repository. In such cases, the designer could first employ REMES for early resource usage analysis, and then apply the approaches mentioned above.

The second category is represented by the UML-based attempts [1, 4] that have been undertaken to tackle the analysis of embedded resources. Although graphical and intuitive, these approaches lack a formal description that could provide the designer with verified resource usage claims. In contrast, REMES provides both a graphical behavioral notation, as well as a rigorous underlying framework for formal analysis.

Third, higher-level formal approaches [5, 6], proposed by Lee et al., encompass a family of process-algebraic formalisms, developed to unify formal modeling and analysis of embedded systems resources. The framework is theoretically rich, yet the tool support is not equally mature. Ouimet et al. use timed abstract state machines [7] to describe resources 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.

Last but not least, as mentioned earlier, REMES focuses on component-based behavioral modeling, and, if paired with ADL descriptions [8], could provide the designer with a complete system representation.

10.7 Conclusions and Future Work

In this paper, we have introduced REMES — a language for resource modeling and analysis of embedded systems. The essence of REMES is a notion of resources that are characterized by their discrete or continuous nature, the way they are consumed and/or allocated and released, and whether they can be referred to, or not. Resources that can be easily modeled include memory, ports, energy, CPU, busses etc.

In order to express resource usage in a system, REMES has a graphical behavioral language influenced by CHARON, 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 illustrate analysis, we have shown in an example how REMES models can be analyzed in the framework of (multi) priced timed automata. The studied example is a temperature control system of a reactor that consumes CPU, energy, and memory resources. The system is architecturally modeled in the component modeling language SaveCCM, and REMES is used to describe function, timing, and resource usage of the included components. To synthesize the optimal resource usage of the system, we model the latter and the weighted sum of resource costs, as a network of PTA, and perform the analysis in the UPPAAL CORA tool.

As future work, we plan to apply the results of Bouyer et al. [17], in order to tackle the feasibility analysis problem for systems in which the global cost function is non-monotonic. We also plan to integrate REMES and its notion of resources in the recently proposed ProCom component model [23] and its associated tools.

Acknowledgments: This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS, the Swedish Research Council (VR), and the European Union via the Q-ImPrESS research project (FP7-215013).

Bibliography

[1] Hany H. Ammar, Vittorio Cortellessa, and Alaa Ibrahim. Modeling Resources in a UML-Based Simulative Environment. In *AICCSA*, pages 405–410, 2001.

[2] Merijn de Jonge, Johan Muskens, and Michel Chaudron. Scenario-Based Prediction of Run-Time Resource Consumption in Component-Based Software Systems. In *Proceedings of the 6th ICSE Workshop on Component-based Software Engineering (CBSE6)*, pages 19–24. IEEE, 2003.

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

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

[5] Insup Lee, Anna Philippou, and Oleg Sokolsky. A General Resource Framework for Real-Time Systems. In *RISSEF*, pages 234–248, 2002.

[6] Insup Lee, Anna Philippou, and Oleg Sokolsky. Resources in Process Algebra.

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

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

[9] Rajeev Alur. Optimal Paths in Weighted Timed Automata. In *In HSCC01: Hybrid Systems: Computation and Control*, pages 49–62. Springer, 2001.

[10] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. In Maria Domenica Di Benedetto and Alberto Sangiovanni-Vincentelli, editors, *Proceedings of the 4th International Workshop on Hybris Systems: Computation and Control*, number 2034 in Lecture Notes in Computer Sciences, pages 147–161. Springer–Verlag, 2001.

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

[12] T. Brihaye, V. Bruyère, and J-F. Raskin. Model-Checking for Weighted Timed Automata. In *Proceedings of FORMATS-FTRTFT*, number 3253 in Lecture Notes in Computer Science, pages 277–292. Springer–Verlag, 2004.

[13] Kim Guldstrand Larsen and Jacob Illum Rasmussen. Optimal Reachability for Multi-Priced Timed Automata. *Theor. Comput. Sci.*, 390(2-3):197–213, 2008.

[14] R. Alur, T. Dang, J. Esposito, Y. Hur, F. Ivančić, V. Kumar, I. Lee, P. Mishra, G. Pappas, and O. Sokolsky. Hierarchical Modeling and Analysis of Embedded Systems. *Proceedings of the IEEE*, 8(3):231–274, 1987.

[15] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 91(1), 2003.

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

[17] P. Bouyer, Th. Brihaye, V. Bruyère, and J.-F. Raskin. On the Optimal Reachability Problem. *Formal Methods in System Design*, 31(2):135–175, 2007.

[18] K. G. Larsen and J. I. Rasmussen. Optimal Conditional Reachability for Multi-priced Timed Automata. In *Proceedings of the 8th International Conference on Foundations of Software Science and Computa-*

tional Structures (FOSSACS 2005/ETAPS 2005), number 3441 in Lecture Notes in Computer Sciences, pages 234–249. Springer–Verlag, 2005.

[19] 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.

[20] 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.

[21] 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. In *ACM Symposium on Applied Computing (SAC '07), Seoul, Korea, March 11-15*, 2007.

[22] Aneta Vulgarakis and Cristina Seceleanu. Embedded Systems Resources: Views on Modeling and Analysis. In *COMPSAC*, pages 1321–1328, 2008.

[23] Sverine Sentilles, Aneta Vulgarakis, Tomas Bures, Jan Carlson, and Ivica Crnkovic. A Component Model for Control-Intensive Distributed Embedded Systems. In *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008)*, pages 310–317. Springer, Oct 2008.

Chapter 11

Paper E: Formal Semantics of the ProCom Real-Time Component Model

Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson

In Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Patras, Greece, August, 2009.

Abstract

ProCom is a new component model for real-time and embedded systems, targeting the domains of vehicular and telecommunication systems. In this paper, we describe how the architectural elements of the ProCom component model have been given a formal semantics. The semantics is given in a small but powerful finite state machine formalism, with notions of urgency, timing, and priorities. By defining the semantics in this way, we (i) provide a rigorous and compact description of the modeling elements of ProCom, (ii) set the ground for formal analysis using other formalisms, and (iii) provide an intuitive and useful description for both practitioners and researchers. To illustrate the approach, we exemplify with a number of particularly interesting cases, ranging from ports and services to components and component hierarchies.

11.1 Introduction

Designing embedded systems (ES) in a *component-based* fashion has become an attractive approach for embedded software development. With benefits ranging from simplification and parallel working to pluggable maintenance and reuse, the financial gains are significant. In this context, systems consist of identifiable, relatively independent and generally replaceable units of composition, called *components*, which encapsulate complex functionality.

Once a component is defined, it can be distributed and used in other applications. Examples of component models include JavaBeans [1], Koala [2], SOFA [3, 4], ProCom [5, 6] etc. Out of these, ProCom is a recently proposed component model tailored for developing *real-time* ES in the vehicular and telecom domains.

To achieve *predictability* throughout the development of the ES, the designer needs to employ a design framework equipped with analysis methods and tools that can be applied at various levels of abstraction, in order to provide estimations and guarantees of relevant system properties. Usually, embedded system designers deal with two kinds of requirements. *Functional* requirements specify the expected services, functionality, and features, independent of the implementation. *Extra-functional* requirements specify the use of available resources. For the same functional requirements, extra-functional properties can vary depending on a large number of factors and choices, including the overall system architecture and the characteristics of the underlying platform. Consequently, ES modeling must deal with both computation and physical constraints, which calls for an underlying semantic framework that abstracts away from both physical notions of concurrency and from all physical constraints on computation.

In this paper, we formalize the semantics of ProCom [5] architectural elements, while identifying potential trouble spots in modeling, which we describe in detail in Section 11.2.2. To tackle the mentioned modeling issues of ES, ProCom consists of two distinct, but related, layers, which expose a number of modeling characteristics that pose challenges to the system designer. The upper layer, called ProSys, serves the modeling of the ES as a number of active and concurrent subsystems, communicating by message passing. The lower layer, ProSave, addresses the internal design of a subsystem down to primitive functional components implemented by code. ProSave components are passive and the communication between them is based on a pipes-and-filters paradigm. Bridging the semantic gap between the two communication paradigms is one particular modeling challenge that we show how to solve within the proposed

ProCom formalization.

Another distinguishing characteristic of ProCom is the possibility to model both fully implemented components, described internally by code, and also design-time components, possibly modeled internally as inter-connected ProSave components that might co-exist with the implemented components.

In order to rigorously describe the above mentioned and all of the other behavioral features of ProCom models, and to provide support for formal analysis, we use an underlying *finite state machine* (FSM) formalism, with notions of urgency, timing and priority. The formal semantics of the FSM language, hence of the architectural elements of our component model, is expressed in terms of *timed automata* with priorities [7] and urgent transitions [8]. However, in the following, we chose to present just some of the most interesting cases, like the formal description of services, component hierarchy, and ProSys-ProSave linking. The formalism is intended to provide a high-level, abstract representation of ProCom semantics, understandable and appealing to both formalists and engineers. Our solution is based on a small semantic core to which the synthesis of ProCom-based models of real-time embedded systems should conform. Note that, although it sets the grounds for formal verification, our semantic descriptions focus only on describing the correct behavior of ProCom architectural elements, without consideration for efficiency in formal verification of the resulted models.

The remainder of the paper is organized as follows. In Section 11.2, we briefly recall the ProCom component model and identify some of its particularities. Section 11.3 presents our underlying formal notation and the actual formalization of the selected ProCom architectural elements. The comparison to related work is carried out in Section 11.4, whereas in Section 11.5, we conclude the paper.

11.2 The Component Model

11.2.1 ProCom

The ProCom component model [6] is specifically developed to address the particularities of the embedded systems domain, including resource limitations and requirements on safety and timeliness.

To achieve efficiency, ProCom components are design-time entities that can comprise information about interfaces, internal structure, code, models, attributes, etc., rather than discernable, concrete units in the final system. Applications are build as a collection of interconnected components, and in the later stages of development this component-based design is transformed into executable units, such as tasks that can be handled by traditional real-time operating systems.

Another basis of the ProCom development approach is that various types of analysis are carried out throughout the development process, in order to ensure that the application will meet requirements on resource usage, safety and timeliness. Early analysis is particularly emphasized, as it allows potential problems to be discovered when the cost of resolving them is relatively low. At early stages, analysis is mainly based on models and estimates, and in later stages on, for example, source code and concrete design parameters. A key concern is to provide means to perform analysis on systems where fully developed parts, for example reused components, co-exist with parts in an early stage of development.

To address the different concerns that exist on different levels of granularity, spanning from the overall architecture of a distributed embedded system, to the details of low-level control functionality, ProCom is organized in two distinct, but related, layers: ProSys and ProSave. In addition to the difference in granularity, the layers differ in terms of architectural style and communication paradigm.

In ProSys, the top layer, a system is modeled as a collection of communicating *subsystems* that execute concurrently, and communicate by asynchronous messages sent and received at typed output and input *message ports*.

Contrasting this, the lower lever, ProSave, consists of passive units, and is based on a 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. Data ports always appear in a group together with a single trigger port, and the ports in the same group are read and written together in a single atomic action.

Figure 11.1 (a) shows the graphical representation of a ProSys subsystem with one input port and two output ports, and (b) shows a simple ProSave component with one input port group and two output port groups. Triangles and boxes denote trigger- and data ports, respectively.

In addition to simple connections from output- to input ports, ProSave contains *connectors* that provide detailed control over the data- and control flow, including forking, joining and dynamically changing connection patterns.

Both layers are hierarchical, meaning that subsystems as well as components can be nested. The way in which the two layers are linked together is that

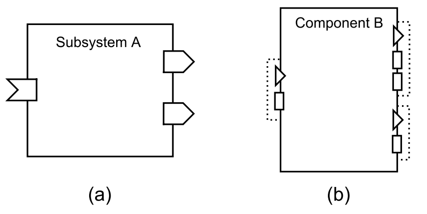


Figure 11.1: A ProSys subsystem and a simple ProSave component.

a primitive ProSys subsystem (i.e., one that is not composed of other subsystems) can be further decomposed into ProSave components. At the bottom of the hierarchy, the behavior of a primitive ProSave component is implemented as a C function.

For the purpose of analysis, it is possible to associate attributes with components and subsystems to specify different functional and non-functional characteristics. Some attributes can be represented by a single number, e.g., worst-case execution time or static memory usage, but in the case of more complex functional and extra-functional behavior (such as timing and resource consumption), a dense time state-based hierarchical modeling language called REMES [9] is used.

11.2.2 Particularities of ProCom

The ProCom component model imposes restrictions on the behavior of its constructs, which should be addressed and formally specified, in order to achieve predictable behavior. This section recalls the informal behavioral semantics of specific modeling constructs in ProCom: services, connections, component hierarchy and building active subsystems out of passive components.

The functionality of a ProSave component is captured by a set of *services*. The services of a component are triggered individually and can execute concurrently, while sharing only data. A service consists of one input port group and zero or more output port groups, and each port group consists of one trigger port and a number of data ports. An input port group may only be accessed at the very start of each invocation, and the service may produce parts of the output at different points in time. The input ports are read in one atomic step, and then the service switches to an executing state, where it performs internal computations and writes at its output port groups. The data and triggering of an output group of a service are always produced at the same time. Before the service returns to idle, each of the associated output port groups must have been activated exactly once. This restriction serves for tight read-execute-write behavior of a service. Since a service is a complex concept, its formalization is highly needed.

In the ProCom language, *connections* and *connectors* define how data and control can be transferred between ProSave components. 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 should hold also in case when the data passes through a connector. ProSave follows a push model for data transfer, so whenever there is data produced on an output port, it is forwarded by the connection to the input data port and stored there. In case more data (trigger) connections are enabled at the same time, the order in which they are taken is non-deterministic. Let us assume the following modeling scenario: three components A, B and C, are interconnected via a Data-Fork connector (see Figure 11.2). 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 component A has finished executing, component B should start executing. However, since the input trigger port of component B is directly connected to the output trigger port of component A, while the data is not transferred directly, but via a connector, there is a risk that the trigger signal may reach component B before the data has arrived. Hence, such a scenario in which trigger might arrive before data should be prohibited by the formalization.

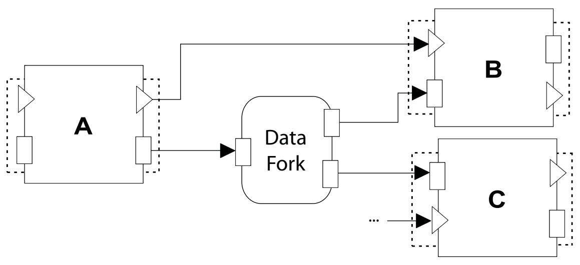


Figure 11.2: Example of a critical modeling of data and trigger transfer in ProCom.

Internally, a ProSave component may be described by code or other interconnected sub-components. When a trigger of an output group is activated internally, all the data (assuming it is ready internally) and the trigger are atomically transferred to the corresponding output port groups of the enclosed component. This contributes to the fact that, externally, there is no difference between components, which allows the coexistence of fully developed components and early design units.

ProSys systems are active entities that communicate via message passing. In contrast, the communication between ProSave components is based on the pipes-and-filters paradigm. Internally, a ProSys system can be built out of other ProSys (sub)systems. At the lowest level of ProSys hierarchy, a subsystem can be internally modeled by ProSave components. In order to build active subsystems out of passive components, we use *clocks*. A clock is a special type of construct that has one output trigger port, which is activated periodically at a given rate. Clocks are not allowed to drift, but it is not assumed that all clocks are initially synchronized. Additionally, a mapping is needed between the message passing in ProSys and the trigger/data communication used in ProSave.

Given the above, we identify the following issues that have motivated our formalism and that we show how to solve in Section 11.3:

- The data and triggering of an output group of a service must always be produced atomically, and each of the service output port groups must have been activated exactly once before the service returns to idle state.
- All the data must arrive to its end destinations before the trigger signal. This rule should also hold in cases when data is transferred through a connector.
- Coexistence of both fully implemented components having well known inner structure, and early design black box components, should be supported.
- Bridging the two communication paradigms: message passing in ProSys and pipes-and-filters in ProSave.

11.3 Formal Semantics of Selected ProCom Architectural Elements

To describe the behavioral semantics of ProCom architectural elements, we introduce a high-level formalism as an extension of finite state machine (FSM) notation and semantics. Our FSM formalism is enriched with additional notions of urgency, priority and implicit timing, necessary for modeling semantics of component-based architectures of real-time systems. The formalism is small, but powerful enough to grasp all the information that is needed for proper formalization of ProCom. In addition, we believe that the language is intuitive enough to be used by developers/engineers, but also formalists/researchers. Yet this has to be proved by experiments that we leave for future work.

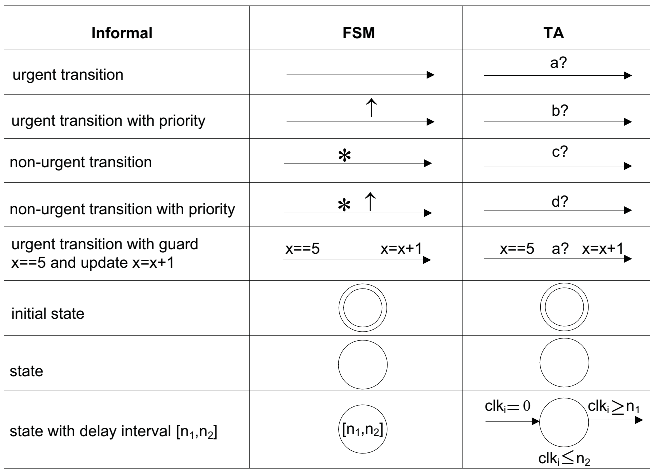
The FSM formalism and related graphical notation are introduced formally below.

11.3.1 Formalism and Graphical Notation

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. Our 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 11.3. A transition can be either *urgent* or *non-urgent*, and it can have *priority* or no priority. As shown in Figure 11.3, a transition may be decorated with the non-urgency symbol *, and/or the priority symbol $\uparrow$. 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.



Informal	FSM	TA
urgent transition	$\longrightarrow$	$\xrightarrow{a?}$
urgent transition with priority	$\xrightarrow{\uparrow}$	$\xrightarrow{b?}$
non-urgent transition	$\xrightarrow{*}$	$\xrightarrow{c?}$
non-urgent transition with priority	$\xrightarrow{* \uparrow}$	$\xrightarrow{d?}$
urgent transition with guard x==5 and update x=x+1	$\xrightarrow{x==5 \quad x=x+1}$	$\xrightarrow{x==5 \quad a? \quad x=x+1}$
initial state	◎	◎
state	○	○
state with delay interval [n_1,n_2]	[n_1,n_2]	$\text{clk}_i = 0$, $\text{clk}_i \geq n_1$, $\text{clk}_i \leq n_2$

Figure 11.3: 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.

11.3.2 Formal Semantics of the FSM Language

In this section, we formally define the semantics of our FSM language using timed automata (TA) [10] with priorities [7] and urgent transitions [8] as a semantic domain. The translation of each FSM element to TA is depicted in Figure 11.3. 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 11.3. The translated TA edges need a timed automaton offering synchronization on the complementary channels (e.g., $a!$ complementary to $a?$), depicted in Figure 11.4.

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 11.4.

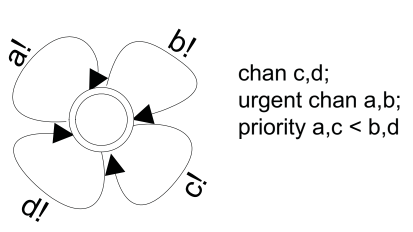


Figure 11.4: The automaton used for synchronization.

11.3.3 Overview of ProCom Formalization

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.

Additionally, 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 [11]. 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 and message ports.

11.3.4 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 11.5 (a)). The informal semantics of a service in ProSave is described in Section 11.2. The formal semantics of a service, in this case, S_1, is described below and shown in Figure 11.5 (b).

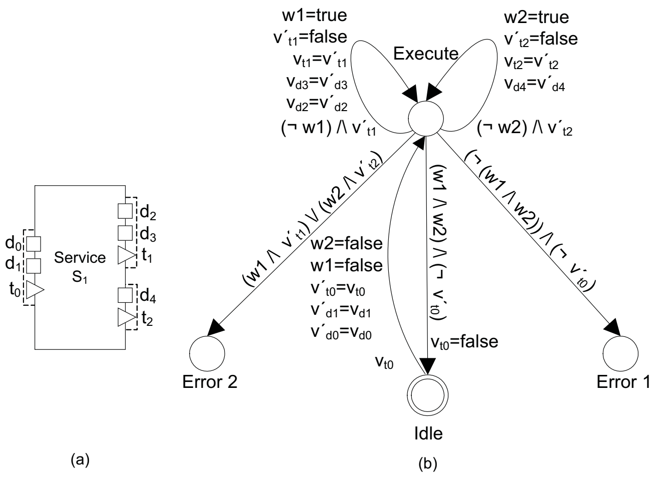


Figure 11.5: (a) A ProSave service S_1 and (b) its formal semantics.

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 two error scenarios:

- A service might try to go back to the Idle state before all output groups have been activated. In the formal semantics of a service this is depicted by the state Error 1.
- During execution, a service might try to activate an already activated output port group. This problem is captured by the state Error 2.

As such, the formal semantics, ensures the informal semantics described in Section 11.2 i.e., the triggering and data of a service is always produced atomically and each of the service output groups is activated exactly once before the service returns to the Idle state.

11.3.5 Data and Trigger Connections

We will now focus on the ProSave connections between two data ports d_0 and d_1 and two trigger ports t_0 and t_1. The formal semantics of ProSave connections is presented in Figure 11.6, for data connection, and in Figure 11.7, for trigger connection.

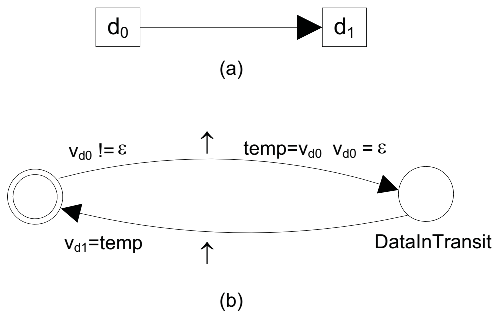


Figure 11.6: (a) A ProSave data connection and (b) its formal semantics.

To ensure that data is transferred prior to trigger, and to avoid undesirable consequences otherwise, the transitions in the FSM formalism (Figure 11.6) 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 [11]).

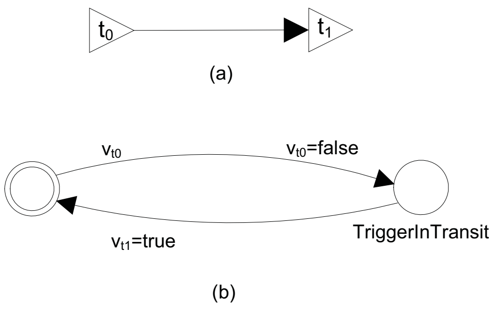


Figure 11.7: (a) A ProSave trigger connection and (b) its formal semantics.

11.3.6 Component Hierarchy

ProCom is a hierarchical component model, with each component being a parallel composition of services, executing concurrently and sharing data. The functionality of a ProSave component can be implemented by a single C function (primitive component) or 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 11.3.4). 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.

11.3.7 Linking Passive and Active Components

By definition, ProSave components are passive and they communicate via data exchange and triggering. ProSave components can be used to define the internals of an active ProSys subsystem with some additional connector types: *clocks* (see Figure 11.9 (a)) and *input*- and *output message ports* (see Figure 11.10 (a) and Figure 11.11 (a), respectively). These connectors are not allowed inside a ProSave component, so the coupling between ProSave and ProSys is done only at the top level in ProSave. The use of these connectors is exemplified in Figure 11.8.

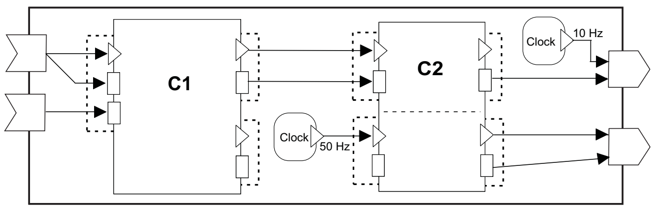


Figure 11.8: A ProSys subsystem internally modelled by ProSave.

A clock serves for generating periodic triggers. A ProSave component can be activated by receiving a periodic trigger with appropriate period. The formal semantics of a ProSave clock with period P is shown in Figure 11.9 (b). Thus, the formal semantics complies to the informal semantics of a clock, described in Section 11.2.

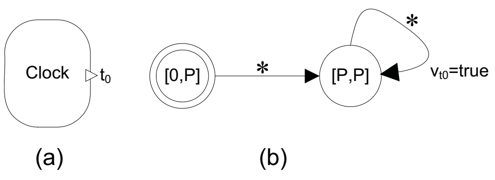


Figure 11.9: (a) A ProSave clock with period P and (b) its formal semantics.

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 semantics of an input message port and an output message port can be described as in Figure 11.10 (b) and Figure 11.11 (b), respectively.

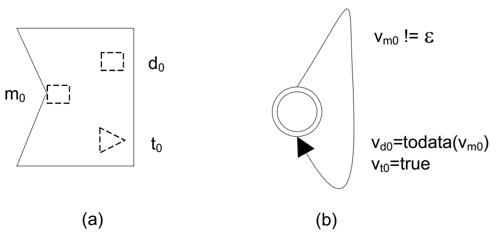


Figure 11.10: (a) A ProSave input message port and (b) its formal semantics.

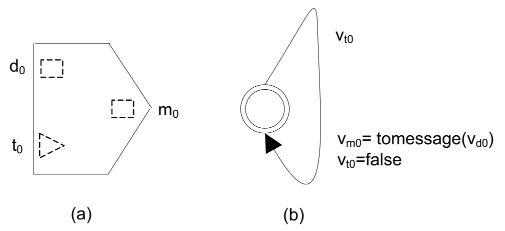


Figure 11.11: (a) A ProSave output message port and (b) its formal semantics.

11.4 Discussion and Related Work

As shown previously, the formalization of the relevant ProCom architectural elements can be subsumed by a small and simple FSM-like language, extended with an abstract representation of clocks, and also urgency and priority on transitions. To place our contribution in the right context and emphasize its strengths and weaknesses, in the following, we review some of the related work to which ours can compare.

The BIP (Behavior, Interaction model, Priority) component framework introduced by Gößler and Sifakis [12, 13] has been designed to support the construction of reactive systems. By separating the notions of behavior, interaction model, and execution model, it enables both heterogeneous modeling, and separation of concerns. The semantics of BIP is given in terms of Timed Automata (TA), on which priority rules are successively applied to enforce certain invariants of the expected real-time behavior. As opposed to our formal semantics, the BIP formalization targets directly the efficient verification of the considered models.

COMDES-II (Component-Based Design of Software for Distributed Embedded Systems) [14] is a development framework in which the functional units encapsulate one or more dynamically scheduled activities. Besides providing a clear separation of concerns (functional behavior from real-time behavior), in modeling, COMDES-II also offers support for formal analysis, by specifying the activity behavior in terms of hybrid state machines. The ProCom semantics presented in this paper does not focus on the transformational aspects of component and system behavior, but more on the reactive and real-time aspects, while emphasizing the co-existence of black-box and fully implemented components, via the component hierarchy.

The communication among SOFA components [3] can be captured formally, by traces, which are sequences of event tokens denoting the events occurring at the interface of a component. The behavior of a SOFA entity (interface, frame or architecture) is the set of all traces, which can be produced by the entity. Such a formalization can be hard to comprehend, but the proposed formalization of ProCom might, on the other hand, be more difficult to implement and exploit towards efficient verification, due to its higher-level of abstraction.

A process-algebraic approach to describing architectural behavior of component models is advocated by Allen and Garlan [15], and Magee et al. [16], who formalize the component behavior in CSP (Communicating Sequential Processes) and via a labeled transition system with a possibly infinite number of states.

Koala [2] is a software component model, introduced by Philips Electronics, designed to build product families of consumer electronics. 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 our FSMs into timed automata [7]), prior to implementation.

ProCom's precursor, SaveCCM, is also an analyzable component model for real-time systems [17]. SaveCCM's semantics is defined by a transformation into timed automata with tasks, a formalism that explicitly models timing and real-time task scheduling. The level of detail of such a formal model is higher than in our FSM notation, making it more suitable for formal verification; however, 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.

11.5 Conclusions

In this paper, we have presented the overall ideas and some lessons learned from defining a formal semantics of the ProCom component modeling language. The ProCom language is structured in two layers, and equipped with a rich set of design elements aimed to primarily support the application area of embedded systems. The ProCom language constructs include service interfaces, data and trigger ports, passive or active components, connections and connectors, hierarchies of components, timing, etc.

Clearly, a formalization of the language needs to deal with all concepts of the modeling language. Additionally, it has been our goal to make the formalization as simple and intuitive as possible, so that it can serve as a basis both for engineers using ProCom, as well as researchers developing analysis techniques, model-transformation tools, etc., within the ProCom framework. In order to meet these sometimes contradicting goals, we have used a small but powerful FSM language, in which the semantics of each ProCom element is described. 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. The language assumes an implicit notion of time, making it easy to integrate with various concurrency models (e.g., the synchronous/reactive concurrency model, or a discrete-event concurrency model) [18]. Its formal semantics is expressed in terms of TA with priorities and urgent transitions, as shown in Section 11.3.2. 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 [19]. Hence, tools such as UPPAAL can be employed for early-stage verification of ProCom models, whereas discrete-time model-checkers, such as DTSpin [20], 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.

To illustrate our approach, we describe in detail how the design constructs for services, data and trigger connections, component hierarchies, and passive and active components of ProCom have been formalized in this manner. These elements are 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.

As future work, we plan to develop support for model-based analysis techniques such as model-checking, based on the formalization given in this paper. In particular, we plan to integrate our recent work on modeling and analysis of embedded resources and the associated modeling language REMES [9] with the formal semantics of ProCom given in this paper.

Acknowledgment

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS.

Bibliography

[1] R. Englander. *Developing Java Beans*. O'Reilly, 1997.

[2] R. van Ommering, F. van der Linden, and J. Kramer. The Koala Component Model For Consumer Electronics Software, booktitle = IEEE Computer, organization = IEEE, month = "march", year = "2000", pages = "78-85",.

[3] T. Bureš, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Proceedings of SERA 2006*, pages 40–48. IEEE CS, August 2006.

[4] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In *Proceedings of ICCDS 98*. IEEE CS, May 1998.

[5] 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.

[6] T. Bureš, J. Carlson, S. Sentilles, and A. Vulgarakis. A Component Model Family for Vehicular Embedded Systems. In *Proceedings of the Third International Conference on Software Engineering Advances*. IEEE, October 2008.

[7] Alexandre David, John Håkansson, Kim Guldstrand Larsen, and Paul Pettersson. Model Checking Timed Automata with Priorities using DBM Subtraction. In *4th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'06)*, pages 128–142. Springer-Verlag, September 2006.

[8] Johan Bengtsson, W. O. David Griffioen, Kre 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, July-August 2002.

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

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

[11] J. Suryadevara, A. Vulgarakis, J. Carlson, C. Seceleanu, and P. Pettersson. ProCom: Formal Semantics. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-234/2009-1-SE, Mälardalen University, March 2009.

[12] G. Gößler and J. Sifakis. Priority Systems. In *Proceedings of FMCO'03*, volume LNCS 3188, pages 314–329. Springer-Verlag, 2004.

[13] G. Gößler and J. Sifakis. Composition for Component-based Modeling. *Science of Computer Programming*, 55(1–3):161–183, 2005.

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

[15] R.J. Allen and D. Garlan. A Formal Basis for Composing Components. *ACM Transactions on SW Engineering and Methodology*, 1997.

[16] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference*, 1995.

[17] 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.

[18] B. Lee and E. A. Lee. Interaction of Finite State Machines and Concurrency Models. In *32nd Annual Asilomar Conference on Signals, Systems, and Computers*, November 1998.

[19] P. A. Abdulla, P. Krcal, and W. Yi. Sampled Universality of Timed Automata. In *10th International Conference Foundations of Software Science and Computational Structures, FOSSACS 2007, part of ETAPS 2007*, volume LNCS 4423, pages 2–16. Springer-Verlag, 2007.

[20] Dragan Bošnački and Dennis Dams. Discrete-Time Promela and Spin. In *FTRTFT '98: Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 307–310. Springer-Verlag, 1998.