Mälardalen University Licentiate Thesis
No.132

Design and Analysis Support for Abstract Models of Component-based Embedded Systems

Jagadish Suryadevara

2011

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering

ISSN 978-91-7485-016-1
ISBN 1651-9256
Printed by Mälardalen University, Västerås, Sweden

Abstract

Developing industrial real-time software systems is challenging due to demands on system safety and reliability, through stringent system requirements in terms of functionality, timing, resource consumption etc. Due to this, the system development needs to ensure predictability *before* the actual implementation, through reliable engineering methods. To address these challenges, model-based engineering (MBE) combined with Component-based development (CBD) has emerged as a feasible solution. MBE supports system modeling and formal analysis through the development phases such as requirements, specification, and design. CBD supports reusability of software parts leading to faster development time, and reduced costs. However, an integrated approach needs to deal with various abstractions of the system during different phases of the development.

In this thesis, we present model-based techniques, for the development of predictable, component-based designs of embedded systems. We consider ProCom as the underlying component model and, as a first step, we define a formal semantics for its architectural elements. The given semantics provides a basis for developing analyzable embedded systems designs, associated analysis techniques, model transformations etc. Next, we describe some commonly-found behavioral patterns, in component-based designs. These patterns provide an abstract, and reusable specification of a real-time components functionality. Also, we define component-based design templates, intended to support the systematic development of component-based designs from abstract system models. Finally, we propose a formal framework to correlate statemachine-based system behavior with corresponding ProCom-based system designs. We validate our research contributions using case-studies and examples, and also by applying verification techniques, such as, model-checking.

Acknowledgements

I wish to thank my research advisors Paul Pettersson, and Cristina Seceleanu for the continuous support and valuable suggestions throughout this research work. It has been a great time of learning, and also fun working with you; and also many moments to cherish for times to come.

I thank co-phd students of my research group; Stefan Björnander, Aida Causevic, Leo Hatvani, and Aneta Vulgarakis for being helpful teammates and also for stimulating research presentations and discussions. I would like to thank Shuhao Li, for sharing interesting research thoughts during his work as visiting phd student.

I thank Jan Carlson, and Eun-Young Kang for valuable contributions as co-authors. I thank Thomas Nolte, Bernhard Schätz, and other anonymous reviewers for providing valuable insights of this research work.

I would like to thank the professors in PROGRESS research project, Ivica Crnkovic, Hans A. Hansson, Björn Lisper, Kristina Lundqvist, Sasikumar Punnekkat, and Mikael Sjödin for the critical observations and valuable discussions during project meetings and also at other occasions. Interacting with them has always been a great experience.

I thank PROGRESS researchers, Radu Dobrin, Andreas Ermedahl, Rikard Land, Frank Lüders, Dag Nyström, Daniel Sundmark, Jukka Mäki-Turja for making the research progress, through inspiring work and helpfulness.

I thank IDT staff; Susanne Fronnå, Åsa Lundkvist, Malin Rosqvist, Carola Ryttersson, and Gunnar Widforss, for making many things easier through their support and lots of patience.

I thank Abhilash, Adnan, Ana, Andreas G., Andreas H., Andreas J., Antonio, Barbara, Batu, Damir, Etienne, Farhang, Federico, Fredrik, Hang, Hongyu, Hüseyin, Håkan, Juraj, Josip, Johan L., Johan K., Karin, Kathrin, Rafia, Rikard Li., Saad, Shahina, Lars, Lilia, Luka, Mats, Mehrdad, Mikael, Mobyen, Moris, Nikola, Nima, Peter, Sara, Séverine, Stefan (Bob), Stefan C., Svetlana, Thomas

Le., Tiberiu (Tibi), Veronica, and Yue for all the fun and the great time together.

I am grateful to the Swedish Research Council (VR), and the Swedish Foundation for Strategic Research (SSF) for funding this research work.

Last but not least, I would like to acknowledge the love and warmth of my family for the success of all my efforts; my daughters Nandana, and Mahima for all the fun; my wife Anuradha for being a wonderful companion.

Jagadish Suryadevara
Västerås, June 2011.

Publications

Included in the thesis

Paper A. "Analyzing a Pattern-Based Model of a Real-Time Turntable System". Davor Slutej, John Håkansson, Jagadish Suryadevara, Cristina Seceleanu, and Paul Pettersson. In proceedings of the 6th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA), pages 161-178, UK, March 2009.

Paper B. "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), pages 478-485, Greece, August, 2009.

Paper C. "Bridging the Semantic Gap between Abstract Models of Embedded Systems", Jagadish Suryadevara, Eun-Young Kang, Cristina Seceleanu, Paul Pettersson, In proceedings of the 13th International Symposium on Component Based Software Engineering (CBSE), Springer LNCS, vol 6092, pages 55 - 73, Czech, June, 2010.

Paper D. "Pattern-driven Support for Designing Component-based Architectural Models", Jagadish Suryadevara, Cristina Seceleanu, Paul Pettersson, In proceedings of the 18th IEEE International Conference on Engineering of Computer-Based Systems (ECBS), USA, April, 2011.

Other Publications

Not included in the thesis

A) Journal

- Jagadish Suryadevara, Lawrence Chung, Shyamasundar R.K, *cmUML - A UML based Framework for Formal Specification of Concurrent, Reactive Systems*, Journal of Object Technology (JOT), vol 7, nr 4, ETH, Swiss Federal Institute of Technology, May, 2008.

- Jagadish Suryadevara, Shyamasundar RK, *UML based Approach for Secured, Fine-grained, Concurrent Access to Shared Variables*, Journal of Object Technology (JOT), vol 6, nr 1, p107-119, ETH, Swiss Federal Institute of Technology, Zurich, January, 2007.

B) Conference/ Workshop

- Jagadish Suryadevaracm, Shyamasundar R.K., *UML - A Precise UML for Abstract Specification of Concurrent Components*, Parallel and Distributed Computing and Systems, p 141-146, ACTA Press, USA, Dallas, Texas, USA, Editor(s): S. Q. Zheng, November, 2006.

- Jagadish Suryadevara, Paul Pettersson, Cristina Seceleanu, *Validating the Design Model of an Autonomous Truck System*, Mälardalen University Software Enginnering Workshop (MUSE'09), Mälardalen University, Västerås, Sweden, November, 2009.

C) Technical Reports

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

Contents

I

Thesis

Chapter 1

Introduction

In modern days, embedded systems have become an intrinsic part of human life. These include highly critical systems in domains, such as, automotive, avionics, and industrial automation. Embedded systems, in addition to being control-intensive and time critical, are increasingly becoming larger in size, and complex in functionality. Most often, different aspects of embedded system functionality are associated with hard real-time constraints, that is, the respective functions should be completed by certain deadlines, or respect specific orders of execution, specified delays etc. Since real-time systems faults may have serious consequences, possibly including loss of human life, their predictability should be guaranteed at design time.

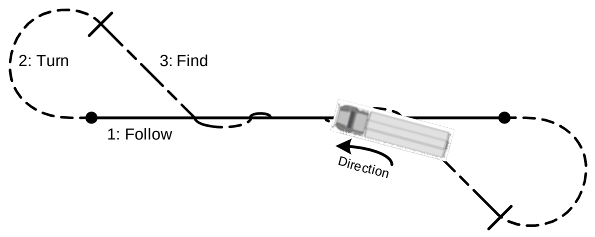

Figure 1.1: An autonomous truck control system application.

Figure 1.1 presents an example embedded system, an autonomous truck control system. It is part of a demonstrator project conducted at the PROGRESS research centre[1]. The truck moves along a specified path, as illustrated in the figure, according to the specified behavior in terms of the three operational models, as described below.

- *Follow*: in which the truck follows the line (the thick line in Fig. 1.1) using light sensors. When the end of the line is detected, it changes to *Turn* mode.
- *Turn*: the truck turns right for a specified time duration, and then changes to *Find* mode.
- *Find*: the truck searches for the line. When it is found, the truck returns to *Follow* mode.

An embedded system interacts with its environment through sensors and actuators. For the truck application described above, the path, as well as the end of the path, are detected with the help of light sensors. The development of an embedded system needs to establish the system predictability by ensuring the system design and implementation follows the specified behavior.

Component- and model-based approaches are emerging as promising solutions to cost-effective development of predictable embedded systems [1, 2]. Component-based approaches aim at increasing reusability of software "parts" i.e., *components*, and *sub-systems*, which is expected to lead to faster development time, and reduced costs. Several component-based methodologies (see Chapter 5) have been developed, for the design and analysis of embedded systems. On the other hand, the model-based approaches e.g., Unified Modeling Language (UML) [3], support modeling and analysis of systems throughout the development phases, such as, requirements, specification, and design. UML contains several modeling views captured by composite diagrams, statemachines, sequence diagrams etc, for structural, and behavioral modeling of complex systems. Such models facilitate both qualitative and quantitative analysis during system development, by describing functional and possibly extra-functional behavior, while omitting the implementation details. Hence, an integrated component- and model-based approach has become increasingly popular, as a high-level design solution for achieving predictability. In this thesis, we adopt this combined approach, and contribute to it, as described in Chapter 3.

[1]For more information about PROGRESS, see http://www.mrtc.mdh.se/progress/

1.1 Thesis Contributions: Overview

We present below an overview of the thesis contributions. Further details are presented in following chapters. In this thesis, we have aimed at meeting the following objectives:

- **Analyzable component-based designs**. We have defined a formal semantics for the ProCom component model, facilitating the design of unambiguous architectural models. Also, the semantics is described in an intuitive formalism, by design, without restricting the capabilities for formal analysis.

- **Behavior modeling of components**. We have proposed behavior patterns to support component modeling. The patterns are based on recurring behavior of real-time components, and provide abstraction mechanisms for increased reusability, and analyzability.

- **Design support for Component-based development**. We have proposed a design methodology to transform abstract system models into ProCom-based component designs. This is done by introducing component-based design templates for transforming abstract features, such as, events, triggers, timeouts, causality etc., into corresponding design elements.

- **Formal correlation between models, and designs**. We have developed a formal framework to correlate ProCom-based designs, and statemachine-based abstract specification models. This provides an important step in bridging the semantic gap between the underlying formalisms of the corresponding models.

1.2 Thesis Outline

This thesis is divided into two parts. The first part is an overview of the research work. In Chapter 1, we describe the background and motivation of the research work underlying the thesis. In Chapter 3, we present the main research goal, and related research questions. In Chapter 4, we discuss the thesis contributions in terms of stated research questions. The related work is described in Chapter 5. In Chapter 6, we summarize the thesis work, over a discussion of the future work . Finally, in Chapter 7, we give an overview of the included papers in the second part of the thesis.

The second part of the thesis contains a collection of four peer-reviewed conference and workshop papers that contain details of the underlying research work of the thesis.

Chapter 2

Background

In this chapter, we overview notions of model-based, and component-based development that are used in this thesis. Also, we briefly present the main ideas and techniques of formal analysis of systems.

2.1 Model-based Engineering

Model-based (sometimes used as model-driven) engineering (MBE), has proven to be an effective paradigm for developing complex systems. It facilitates system modeling through multiple abstractions or *views*, corresponding to its development phases. This enables the seamless integration of design and analysis techniques and tools, otherwise pertaining to specific system features / behaviors, throughout the system development.

In a model-based engineering approach, the main development phases are requirements, specification, and design. During the requirements phase, various system properties, attributes, constraints etc are identified. These are also categorized as functional, and extra-functional (timing, resource-efficient, performance, reliability etc.). In the specification phase, abstract models of system structure and behavior are developed. The structural models include high-level system architecture diagrams or component-based designs. The behavior models, if used at this phase, for e.g., statemachines or sequence diagrams, are generally abstract, and describe the system-level behavior. For example, Figure 2.1(b) presents a statemachine view of the application behavior of the autonomous truck system (Figure 2.1(a)) as described in Chapter 1. These

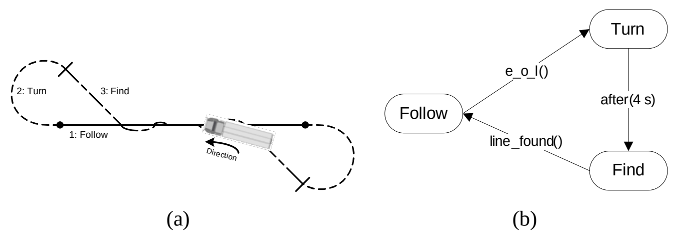

Figure 2.1: (a) Autonomous truck control system, and (b) an abstract specification model of the system.

models are useful for the formal analysis or validation during early phases of development with respect to the intended system properties, identified during the requirements phase.

During system design, the specification models, such as, architectural and behavioral descriptions are refined into detailed models, often hierarchical in nature. Different *parts* of the system, that is, both large-grained (e.g., subsystems), as well as small-grained (e.g., components), are integrated into precise design models, by specifying detailed mechanisms for communication, synchronization etc. Additionally, these models may be associated with deployment models, which specify the actual physical configuration of a system. The detailed design, and deployment models provide the valuable opportunity to verify critical timing properties, such as, *end-to-end* response time, or analyze for the best and the worst-case resource usage etc.

One of the strong points of MBE is the possibility of carrying out model-to-model transformations. This proves to be a powerful technique that enables both qualitative and quantitative analysis of system properties, by integrating existing methods, and tools. As a result, integrated development environments (IDE) have become the development norm for all kinds of complex systems, including real-time systems.

In support of MBE, the Unified Modeling Language (UML) [3, 4] has become a *de facto* industry standard modeling language for complex systems. UML consists of many formalisms, successfully used in industry and academia. Although UML lacks a unique precise semantics (as defined by OMG), several researchers have proposed various semantics to UML constructs, which enable rigorous reasoning [5, 6]. UML provides extension mechanisms to facilitate its application in different domains, including real-time systems, through customized sub-languages called UML *profiles*. For instance, MARTE (Modeling and Analysis of Real-Time Systems) [7] UML profile is intended for modeling and analysis of real-time systems. We have considered a subset of UML and MARTE in the model-based techniques presented in this thesis.

2.2 Component-based Development

The most important goal of component-based development (CBD) [1] is to tame the development of complex systems, by supporting reusability as a principle, aiming at improving cost-effectiveness of development. This is achieved through reuse of various system parts, such as, *subsystems*, and *components*. This may also include existing system models e.g., behavior models, when the development is combined with model-based engineering, as described previously.

Figure 2.2 presents design layers or phases in a component-based development. From a set of system requirements, higher-level system models, such as, behavior specification can be derived. Also, an initial hardware architecture may be obtained. These models are often input or provide guidance to more detailed models, such as, analysis, design, deployment etc before the final implementation phase. As shown in the figure, component-based software designs, such as, high-level *System Software Design*, *Subsystem Design*, and *Architectural Design* can be directly influenced or guided by a specification model e.g., statemachine view of the system behavior.

In CBD approach, the central element is the *component model*, for example ProCom component model [8]. A component model describes the syntax and semantics of a component-based design. A *component* encapsulates functionality, paving the way for its reuse. A component can be hierarchical (made of connected sub-components), or atomic / primitive (not containing any other components). Further, a component may be large-grained and system level e.g., *subsystem* components or small-grained components containing executables i.e., actual *code*. This diversity of component characteristics serves the purpose of reuse at different levels of granularity during design, but also the need for addressing different issues during different phases of system development. For example, ProCom consists of two sub-languages; ProSys for modeling a system as a collection of communicating *subsystem* components, while ProSave is based on *pipes-and-filters* architectural style.

Components communicate through ports. Ports are of different kinds e.g., message ports, data ports, control ports etc. Communication between components can be synchronous or asynchronous. The detailed semantics of com-

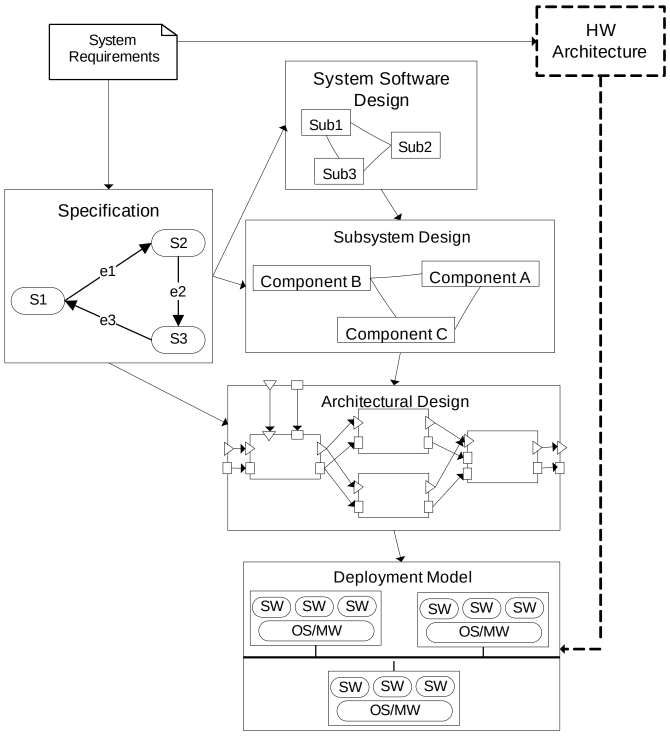

Figure 2.2: Design layers in component-based development.

ponent execution, communication etc, is given by the underlying *component model*. For ProCom, the sub-languages are based on different communication paradigms; while ProSys components are based on message passing, ProSave components are based on explicit separation between data and control flow. A component model defines the rules of execution behavior at the architectural level, and it is therefore most important for carrying out formal verification of components and global system properties.

CBD assumes the interleaving of system design with component development, with one influencing the other. An initial configuration of a system may be rapidly designed from existing fully-developed components, as well as partially-developed components. In component development, components may be created anew, or modified from those currently existing in the com-

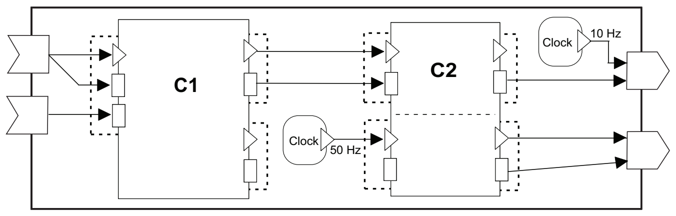

Figure 2.3: A composite subsystem component with detailed functional components (C1, C2).

ponent *repository*. While components may be developed in parallel, a partial system configuration provides a very useful context to conduct early analysis and system validation. The early system analysis guides the component selection during system development, might narrow the design space by ruling out infeasible choices, and provides rough estimations of the system's predictability.

Component-based development is generally supported by a component *framework* consisting of a specific component model, corresponding component technology, and a component repository. The framework may also include an integrated development environment (IDE) for both system and component development. Also, IDEs facilitate integration of other tools for system design, analysis, as well as synthesis. The ProCom language and the corresponding modeling framework constitutes the underlying context of this thesis work. Specifically, as shown in Figure 2.2, we focus on linking behavior specification models to architectural design in ProCom. However, the later phases i.e., deployment or hardware architecture design are outside the scope of this thesis.

2.3 Formal Analysis

Analytic methods for real-time systems, e.g., schedulability analysis, performance analysis are empirical methods based on various system parameters. On the other hand, computational methods, commonly known as "formal methods" [9], such as *model-checking*, and *theorem proving*, are based on exhaustive analysis of system behavior, by computing all possible execution states of a system representation. In this thesis, we focus on verification techniques based on formal methods.

Applying formal methods requires that both the specification and the sys-

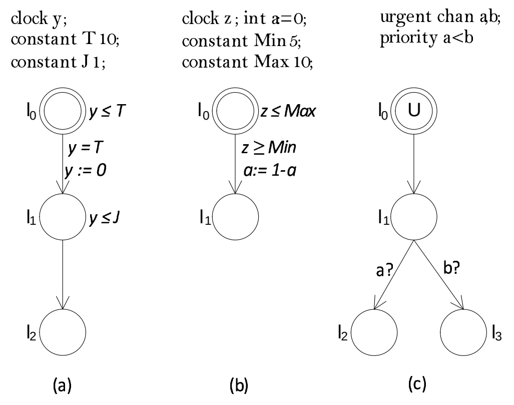

Figure 2.4: Examples of timed automata (TA) modeling; (a) A clock with period T, and jitter J, (b) A computation occurring between Min and Max time units, (c) Modeling urgent locations and priority-based synchronizations.

tem model are given in some precise mathematical notation.

- System model: a formal representation of system model in terms of structure, and behavior. The example system models are statecharts, timed automata etc.

- Specification: a formal description of the intended system behavior, or system properties. Example specification formalisms are temporal logics, statemachine, automata, etc. The important kinds of system properties that can be specified are functional, safety, liveness, timing etc.

Formal methods are applied to verify if the intended behavior or properties of a specification hold on the corresponding design or implementation. The specific formal technique that is applied, in general, is one of following two kinds: model-checking and theorem proving that we will briefly recall later in this section.

Timed automata formalism: Formal methods based on timed automata [10] are extensively used for modeling and analysis of real-time systems [11, 12]. Timed automata supports modeling of real-time concepts, such as, periodicity, jitter, timing, priority, urgency etc based on clock variables, and synchronization channels, as shown in Figure 2.4. A timed automata consists of locations

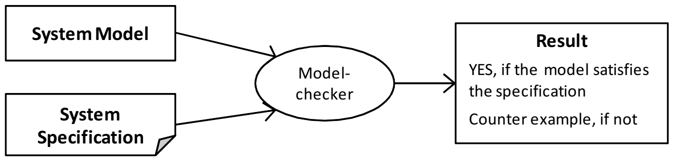

Figure 2.5: A schematic view of model-checking based verification

and edges. An edge is taken from the current location if its associated guard expression (e.g., 'y=T' in Figure 2.4.(a)), if any, consisting of integer and data variables becomes true. Further, edges are associated with update actions, and clock resets (see Figure 2.4.(a)). Locations can be *urgent* (e.g., l_0 in Figure 2.4.(c)) to disallow passing of time, or associated with invariant expression (e.g., l_0 in Figure 2.4.(a)) to allow only the maximum delay of the specified time units. Timed automata also supports priority-based channels for synchronization. An urgent location e.g., at location l_1 in Figure 2.4.(c)) must be exited with no time delay.

Model-checking: It is a formal technique for automatically, i.e., algorithmically, verifying correctness properties of a finite-state system. Given a model of a system e.g., a finite state machine, say M, model-checking verifies (see figure 2.5) whether the model satisfies a given specification e.g. a temporal logic formula, say ρ. This can be formally expressed as below.

$$M, s \models \rho \text{ that is, given model M, and initial state s, } \rho \text{ holds.}$$

In model-checking, the above problem reduces to a reachability problem or to temporal logic verification i.e., of verifying if the expression ρ is satisfied by a state in M, by algorithmically traversing the state transition graph of M. Further, model-checking can produce a counter-example i.e., a partial execution trace leading to a system state where the property is not satisfied by the model. Example model-checking tools are SMV [13], SPIN [14], UPPAAL [15], TIMES [16], and many others.

Thorem proving: It is an interactive formal technique, compared to model-checking. In this approach, both a specification, and corresponding implementation are represented as logic descriptions e.g., using first- or higher-order predicate logics. Then, a designer employs a theorem-proving tool through partially guided, rigorous proof steps, to show that the implementation *implies* the logical specification. Example theorem proving or automated reasoning tools are `PVS` [17], `HOL` [18], and many others.

Model-checking tools such as `UPPAAL`, and `TIMES` are based on extended forms of timed automata, and enable verification of safety (e.g, something bad never happens), liveness (e.g, something good eventually happens), and timing properties of system models as well as schedulability analysis using precise task models.

Application of formal methods, in general, require expertise in constructing mathematical models, applying the corresponding analysis techniques, analyzing the results, and improving the system design or implementation. However, these tasks can be simplified by choosing suitable abstractions for both specification and system models, and corresponding tool support that hide the intricacies of the underlying formalisms. Further, the higher level abstractions facilitate increased understandability, reusability of system features.

While formal methods have been successfully applied in hardware design, their application in system design, including software, has been only recently increasing. This is mainly due to the reason that the applicability of formal methods is constrained by the size, and complexity of the systems that can be verified. However, the increasing demands for system predictability imply the stronger need of applying formal, and systematic verification techniques. This can be done through suitable abstraction techniques, compositional design methodologies, and tailored verification techniques.

Chapter 3

Research Goals

In this chapter, we outline the scope of the research presented in this thesis. We begin with the description of the overall research goal, within the context of model- and component-based development of embedded systems. In relation to this, we also present some specific research questions that are addressed in the research work and are intended to serve the overall goal.

3.1 Problem Description

The combined component- and model-based approach is deemed feasible for ensuring reusability, maintainability and analyzability of predictable embedded software designs. However, for most of integrated methodologies, it involves different formalisms, languages, tools for system modeling across different development phases, such as, requirements, specification, and design. These paradigms address different concerns of a system behavior in terms of structure, functionality, timing etc, at various phases of development. The variety of paradigms and views give rise to many challenges that need to be addressed, with respect to the abstract system models, and the underlying formalisms with varying semantics.

The overall objective of the research behind this thesis work is to develop suitable methods for the design and analysis of component-based embedded systems, based on abstract models for both system, and its components. Our main research goal is to:

Develop suitable methods for designing and analyzing abstract *models of embedded systems.*

With respect to the above research goal, we present the following specific research questions.

3.2 Research Questions

Research question 1 In a component-based development style, an embedded software system is represented by *components* inter-connected using architectural elements, such as, *ports*, *connections*, and *connectors*. A component model for real-time systems, for example ProCom [8], contains many elements with critical real-time features such as period, urgency, priority etc. To develop unambiguous designs of a system, it is essential to associate the underlying component model and its constructs with a formal semantics to which any design should conform. Also, to support rigorous analysis, the design elements together with the underlying semantics should be easily transformed into established verification frameworks, such as, UPPAAL-based model-checking. Moreover, we would obtain an even higher gain if the formalization were intuitive and easy-to-use. Such features would be beneficial to engineers using the component model, as well as researchers developing analysis techniques, model-transformation tools etc. Based on these arguments, we state our first research question below.

> *How to formally describe the behavior of architectural elements of a real-time component model such that we provide a basis for rigorous analysis?*
>
> *(Q1)*

Research question 2 Abstract models, for example, statemachine-based behavior models are commonly used to represent system behavior during early phases of the development. For embedded systems, such models are based on features, such as, events, control states, timeouts, etc., and also associated with timing constraints, such as, *end-to-end* response time. However, these aspects are often considered in a rather ad hoc way, while developing a component-based system design. Applying a systematic approach in translating the above system features to component-based designs is desirable, to preserve the behavioral properties when generating component-based designs from abstract models of embedded systems. Also, in a related design aspect, that is, component development, the functionality of components is often represented only by code. This makes components difficult to understand and reuse.From this, we state the next research question as below.

> *How to develop component-based designs from abstract system models and component behaviors?*
>
> *(Q2)*

Research question 3 When embedded systems are designed using an MBE approach, it is quite often the case that different formalisms are used for system modeling during different phases of its development. These formalisms are normally based on events, or time triggering, or dataflow, or even a combination of these. These abstractions may be used within the same, or across different phases of system development, such as, requirements, specification, and design. In order to ensure predictable behavior of the system, the different abstractions used in development should be verified for behavior consistency. Specifically, in a component-based development, the behavior of a component-based design must be consistent with the system behavior specified using other abstractions. From this, we state the next research question as below.

> *How to relate component-based system designs with the abstract models of system behavior?*
>
> *(Q3)*

Chapter 4

Research Contributions

In this chapter, we give an overview of the thesis contributions with respect to the research questions presented in the previous chapter.

4.1 Formal Semantics of a Real-Time Component Model

Problem description. To achieve predictability of a component-based real-time system, the designer needs a development framework equipped with analysis methods and tools. This foremost requires a formalization of the underlying component model, which will give unambiguous meaning to their constituent elements, such that any claim regarding system and component properties becomes refutable. However, the formalization of a real-time component model needs to deal with issues like priority, urgency, timing etc. Further, it would be effective to make the formalization as simple and intuitive as possible, such that, it can serve as a basis both for designers using the language, and the researchers developing analysis techniques, model-transformation tools etc. Coming up with such a formalization is no trivial job.

ProCom [8] is a component model for real-time systems (recently developed at Mälardalen University, within the PROGRESS research centre). To address various modeling issues, ProCom consists of two distinct but related layers. The upper layer, called ProSys, supports the modeling of an embedded system as a collection 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 *pipes-and-filters* paradigm. However, ProCom has a number of modeling characteristics that pose challenges to the system designer. For example, bridging the semantic gap between the two communication paradigms is one particular modeling challenge that needs to be addressed by any 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 as inter-connected ProSave components), which might co-exist with the implemented components. The ProCom language constructs include service interfaces, data and trigger ports, passive or active components, connections and connectors, hierarchies of components, timing etc. Clearly, an intuitive formalization of the ProCom component language is essential to support system designers, as well as researchers developing analysis techniques for predictability.

Solution. We describe the formal semantics of the ProCom component model rigorously, as well as intuitively, using a *finite state machine* (FSM) formalism, extended with notions of urgency, timing and priority. The formal semantics of the FSM language itself is described using *timed automata* with priorities [19] and urgent transitions [20]. The FSM formalism is intended to provide a high-level, abstract description of ProCom semantics,based on a small semantic core to which the synthesis of ProCom-based models need to conform. However, the semantic descriptions focus only on describing the correct behavior of ProCom architectural elements, such as, *components*, *services*, *ports*, *connections*, *connectors* etc, but do not target goals, such as, achieving efficiency in formal verification of the resulting models.

The FSM language builds on standard FSM, enriched with finite-domain integer variables, guards and assignments on transitions, notions of urgency and priority, and 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 [21]. The FSM language has a graphical appeal and it is simpler than the corresponding TA model, as it abstracts from real-valued variables and synchronization channels. However, the FSM models of ProCom-based designs can be analyzed with timed automata tools like UPPAAL [15].

The details of the above research work can be found in Paper B, which is included in the second part of this thesis.

4.2 Design Support for Component-based Development

Problem description. When developing real-time systems in a CBD fashion, the state-of-practice is dominated by an ad-hoc mixture of methods and tools, and system validation is mostly done by extensive testing after the implementation phase. In general, components are introduced as *executable* software units that can be deployed into a system. This makes both the design model, and also individual components, incomprehensible and difficult to reuse. To support predictability, the system designs should reflect a clearly stated intent and structure, besides containing reusable, analyzable, and understandable component behaviors. Further, the structured design process should take into account the two parallel, but related, work-flows of component-based development, that is, the overall system-development, and component-development.

Solution. A general solution to the above demanding requirements is a structured, pattern-based design methodology for developing component-based embedded systems. For the overall system-development, the abstract system models e.g., specification, requirements etc, can be considered to guide the component based design process. In the parallel component-development activity, behavior patterns based on the recurring behavior of individual components, can be considered to support developing abstract, reusable component behaviors.

For component-development, we have defined behavior modeling patterns based on the common behaviors of real-time components. The patterns are described in a finite-state-machine (FSM) notation that we call *Pattern-FSM* (PFSM). The finite behaviors of components can be specified using two design patterns encoding $run - to - completion$ semantics, and *history* states. Timing is introduced using a third design pattern for specifying the response time of components. These patterns can be easily transformed to fit into specific formal frameworks for verification. To show the usefulness of these patterns, we have applied them in the component-based development of an industrial real-time turntable system [22]. The chosen analysis framework is the Timed Automata (TA) language of UPPAAL [23, 15]. Component behaviors have been specified using the patterns and manually transformed into timed automata models. Also, the complete design together with an environmental model has been translated into timed automata for property verification using UPPAAL.

In a related design activity, to support the transformation of abstract system models into corresponding component-based designs, we have proposed component design templates (referred as component patterns).The abstract system model, which we refer to as *modemachine* i.e. an extended form of UML statemachine, represents the mode configurations of a system and corresponding event-based mode changes. The constraints are specified using UML/ MARTE Clock Constraint Specification Language (CCSL), based on physical and logical clocks, and we also show how to specify periodic triggers, timeouts, causality etc.

For a given system, the modemachine represents an abstract specification of architectural features, while hiding the detailed internal behaviors. Based on this, one can derive architectural or component-based designs that satisfy the specified functional and timing constraints. To guide the transformation, we have proposed several component patterns: *timeout* pattern, *discrete-clock* pattern, *periodic-behavior* pattern, and *controller* pattern. While the first two patterns model the timing aspects, the other patterns model the time- and event-triggered executions of internal behaviors respectively.

The patterns are implemented within the ProCom framework, to support the development of ProCom-based designs. Based on the formal semantics of ProCom, the patterns are manually transformed into timed automata framework for verification of timing properties using UPPAAL model-checker.The usefulness of these design patterns is demonstrated on a temperature control system (TCS), for which we develop a ProCom-based design by applying our patterns.

The details of the research described above can be found in both Paper A and Paper D, included in the second part of the thesis.

4.3 Relating Abstract Models of Embedded Systems

Problem description. The predictable behavior of a real-time system can be ensured through extensive modeling and analysis during development phases, such as, specification and design. These phases are suitable for applying early predictability analysis techniques with respect to functionality, timing, resource consumption etc., over different models of system structure, and behavior. However, such models may use paradigms that cannot be immediately compared and related, due to their apparently incompatible nature.

There exist several paradigms for the specification of embedded systems.

For example, statemachine-based approaches, such as UML statemachines [4], are intended to specify system states and the corresponding system behavior in these states. The timed UML state-machines add the possibility of representing timing aspects. Statemachines often use an aperiodic, *event-triggered* representation of behavior, since such a paradigm facilitates easy changing of a model's configuration or set of events. On the other hand, design models i.e. architectural/ structural models might use a different modeling paradigm e.g., a periodic, *time-triggered* behavioral description. With time-triggered communication, the data is read from a buffer according to a triggering condition generated by e.g., a periodic clock. Although these modeling capabilities, in isolation, are invaluable to a mature development process tailored for predictability, when applied to embedded system development, they need to be proven consistent with each other.

Solution. In order to address the above goal of ensuring inter-model consistency, we have defined a methodology for relating event-based abstract models with time triggered, data-flow based design models. Such abstractions may be used within the same or different phases of system development. Concretely, we consider UML statemachines for modeling event-based system specification, and the ProCom language as the basis for modeling the system's architecture (the design model). Hence, the method may be only suitable for a specific class of embedded systems, which employ the above mentioned formalisms. However, the underlying methodology can be generalized to include other classes of systems.

The proposed approach of relating abstract models of embedded systems is based on comparison of execution trajectories of system models. To be able to carry out a meaningful comparison, the respective models need to rely on precise semantic grounds. To accomplish this, we define the formal semantics of both kinds of models i.e., statemachines and ProCom-based designs is defined in terms of the underlying state-transition systems. As the execution trajectories generated by these models can be extremely large and incomprehensible, they need to be reduced to more readable and analyzable forms. Consequently, we define two sets of inference rules, one for simplifying the specification trajectories, and the other one for simplifying the design ones. Moreover, in order to relate and compare the above two sets of simplified trajectories, we have also proposed a set of transformation rules from time-triggered to event-triggered trajectories, and vice-versa. To summerize the steps for bridging the gap between the paradigms consists of the following five steps:

- given a specification trajectory, generate a corresponding design trajectory by e.g, simulating the model
- simplify the specification trajectory (may be skipped)
- simplify the design trajectory
- transform the design trajectory into one comparable to the event-based specification trajectory
- compare the reduced specification and design trajectories

The above described methodology, which relies on rules that can be automated, has however been manually applied to a representative design trajectory suitable to demonstrate several simplification scenarios described previously.

Our initial experiences with applying the proposed technique to an autonomous truck control system indicate that the design model trajectories can sometimes be manually transformed into trajectories of the specification model. However, as this is not the case in general, the above framework should be extended with simulation relation checking methods, for proving conformance between the respective trajectories.

The details of the above research work can be found in Paper C included in the second part of this thesis.

4.4 Research Questions - Revisited

In this section, we discuss how the research contributions described in the previous sections correspond to the research questions presented in Chapter 3. Details can be found in the corresponding research papers included in the second part of this thesis.

Q1 How to formally describe the behavior of architectural elements of a real-time component model such that we provide a basis for rigorous analysis?

The formalization described in Section 4.1, which resulted in Paper B, shows a way of giving formal semantics to architectural elements of our real-time component model ProCom. The given semantics not only follows an intuitive approach but provides basis for easier translation of ProCom-based designs into corresponding models in semantic domains, for example timed automata, for formal verifications.

Q2 *How to develop component-based designs from abstract system models and component behaviors?*

The proposed approach as described in Section 4.2 has resulted in two research papers, that is, Paper A, and D. These papers describe approaches for providing design support for development of embedded systems. In Paper A, we have proposed behavior modeling patterns for components and demonstrated their usefulness by applying them to an example industrial turntable system. In Paper D, we presented a few component-based design templates to develop architectural designs from abstract system models. The approach is applied in developing a component-based design of a temperature control system.

Q3 *How to relate component-based system designs with the abstract models of system behavior?*

The proposed methodology as described in Section 4.3, has resulted in Paper C. In this paper, we have defined a methodology based on inference rules for simplifying, and comparing the execution trajectories of specification, and architectural models. The approach is demonstrated by applying it on an example autonomous truck system.

From the above, one can conclude that we have addressed the research questions to some extent. Although the research questions are much wider in scope, we have chosen ProCom-based design framework for our research work. Consequently, our work is not a general solution to the research problems, yet it provides particular answers. We discuss the limitations of our contributions, along with possible future lines of research, in Chapter 6 .

Chapter 5

Related Work

In this chapter, we describe both the state-of-the-art related to the model- and component-based development of embedded systems.

5.1 Formalizations of Real-time Component Models

In order to support the component-based development of embedded systems, several researchers, as well as practitioners have devised a variety of component models and corresponding.development frameworks.

COMDES-II (Component-Based Design of Software for Distributed Embedded Systems) [24] 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 behavior in terms of hybrid state machines. The ProCom semantics presented in this thesis 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 BIP (Behavior, Interaction, Priority) component framework, introduced by Gößler and Sifakis [25, 26], has been designed to support the construction of reactive systems. By separating the notions of behavior, interaction, 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 compared to our approach for ProCom formal semantics, the BIP formalization targets directly the efficient verification of the considered models.

In SOFA component model [27], the communication among components 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 [28], and Magee et al. [29], who formalize the component behavior in CSP (Communicating Sequential Processes) and via a labeled transition system with a possibly infinite number of states.

Koala [30] 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 [31]), prior to implementation.

ProCom's precursor, SaveCCM, is also an analyzable component model for real-time systems [32]. 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 for ProCom semantics, 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.

5.2 Design Support for Component-based Development

The Statemate toolkit [33] is an early working environment for the development of complex reactive systems. Modularity of the system development is provided in terms of different *views*, such as, structure, functionality, and behavior. Our approach for behavior specification of components (*modules* in Statemate) is similar to the Statecharts [34], the behavioral language of Statemate. Though not hierarchical, our FSM notation for component behaviors (see Section 8.3), combined with the patterns proposed in this paper, is similar to the Statechart features “run-to-completion” and “execution history”.

The BIP framework and the toolkit IF [35] are intended for predictable embedded systems development by supporting *correctness-by-construction* and compositional verification. While BIP offers bottom-up design of systems, our approach supports CBD in a bit more pragmatical traditional top-down design, with support of modeling in Save-IDE [36] and formal verification using the UppaalPort toolkit [37, 15].

The CHARON toolkit [38] supports modular specification of embedded systems, based on the notions of *agents* and *modes*, for architectural and behavioral specifications, respectively. Our behavioral specification language of components shares some features of the modes in Charon, but without hierarchy, and in our approach the execution history of a component is provided by using a simple design pattern.

The case study of the Turntable production system, presented in this thesis, has previously been analyzed using different methods and tools. Bos and Kleijn [39] have specified the turntable model in χ [40], a simulation language for industrial systems, and translated into Promela, the input language of the Spin model-checker to verify several properties of the model. Bortnik et al. [41] have translated a χ model of the turntable system into the specification languages of three model-checkers: CADP, Spin, and Uppaal comparing the ease of conversion, the expressiveness of each of the specification languages, and the abilities and performances of the respective model-checkers. Ke et al. [42] have implemented the turntable production system in COMDES-II, a component-based framework. They have developed a semantic transformation of the COMDESS-II model into an UPPAAL timed automata model, allowing for formal verification of a set of properties similar to those verified by Bortnik et al [41].

In the domain of synchronous languages [43], mode automata and the notion of running modes have been introduced, to reduce the gap between the initial design of a system and the program written for it. The formalism has been proposed to support both dataflow, and imperative styles. The *modemachine* proposed in this thesis corresponds to the event-based, hierarchical, high-level control structure of the system and associated timing constraints.

Sandén proposes the “state-machine” pattern [44], for designing concurrent real-time software in Ada [45]. Many possible implementations of the pattern, corresponding to concurrent, reactive, and time-triggered behaviors, are described. Also, patterns for non-functional aspects such as resource usage, quality-of-service have been proposed [46]. However, such patterns focus on the design or implementation phase of the system. The patterns proposed in this thesis support the design process, by transforming the specification aspects, with associated timing constraints, into the corresponding design elements.

Maxwell et al. have proposed a formal framework [47] for heuristics-based transformation of architectural designs. The authors capture heuristics in a structured and formal manner, such that the architectural transformations can be performed for optimizing the non-functional qualities of a system. Denford et al. have proposed an architectural refinement method [48] that focuses on non-functional requirements e.g., reliability, performance, while still addressing the functional requirements. While these works focus on non-functional aspects, such as, performance, we address architectural designs through timing constraints of embedded systems.

The UML profile MARTE is extensively used in the context of AADL (Architecture analysis and design language [49]) for component-based designs of real-time, embedded systems [50, 51]. AADL supports the modeling of both software components such as *thread*, *subprogram*, *process*, and platform components, such as, *bus*, *memory*, *processor*, and *device*. However, AADL introduces avoidable redundancies that obscure the model and may even lead to design inconsistency. To address this deficiency, the MARTE clock constraints have been used [51] to precisely specify both event, and time triggered communications for AADL models, and to compute end-to-end flow latency. While these works focus on models related to software and platform mapping, in this thesis, we address specification, design mappings and corresponding behavior correlations.

EastADL [52] is a layered architecture language for model-based development of automotive software. To address various concerns of system’s lifecycle development, it provides abstraction layers, such as, feature level, requirements, analysis, design, and implementation. Mallet et al. have described MARTE specification of EastADL timing requirements [53]. This enables the use of MARTE tools for timing verification of EastADL requirements.

5.3 Relating Abstract Models of Embedded Systems

The problem of relating design to specification models is a topic with a growing interest in the research community.

For synthesizing executable programs from timed models, Krcal et al. [54] have proposed a timed automata based semantic framework, relying on non-instant observability of events. Time-triggered automata (TTA), a sub class of timed automata (TA), is used to model finite state implementations of a controller that services the request patterns specified by a TA. This technique enables deciding whether a TTA correctly implements a TA specification. In comparison, although ProCom oriented, our methodology can be applied within a generic component-based framework, and is not being tied to any particular formal verification framework either.

Sifakis et al. propose a methodology for relating the abstractions of both real-time application software and corresponding implementation [55]. The related formal modeling framework integrates event-driven, and time triggered paradigms by defining *untiming* functions. Problems of correctness, timing analysis, and synthesis are considered in the methodology. In contrast to our approach, this one does not address the intermediate design layer commonly used in system development.

Plasil and Visnovsky describe a formal framework based on *behavior protocols*, in order to formally specify the interplay between components [56]. This allows for formal reasoning about the correctness of the specification refinement and about the correctness of an implementation, in terms of the specification. Further, the framework is validated in the SOFA component model environment [57]. While the approach provides much needed formal correctness in component-based development, it does not address timing issues and vertical layers of abstractions in real-time system development.

Schätz et al. [58] have described a model-transformation based approach using constraints as transformation rules guiding a mechanized exploration of possible design alternatives. The approach has been demonstrated for the incremental deployment of logical architectures to hardware platforms.

Chapter 6

Conclusions and Future Work

In this chapter, we present a summary of the thesis contributions, as well as corresponding limitations. Finally, we conclude the thesis work with presenting possible lines of future work.

6.1 Summary and Discussion

In this thesis work, we have tackled some design challenges of the development of real-time systems, in the context of model-based engineering (MBE) and component-based development (CBD). While models provide very useful abstractions and corresponding predictability analysis techniques, these however increase the development complexity due to multiplicity of models, underlying formalisms, tools etc that are generally employed. Similarly, while CBD enables faster development and reduced costs through reusability of system designs, it provides limited capabilities regarding formal verification. We have addressed some of these challenges in this thesis work.

As a first step, we have chosen the ProCom -based design framework as the basis for developing real-time systems, due to its particular characteristics, on the one hand useful for real-time design, yet on the other hand challenging to formally analyze. The framework is associated with many directions of real-time research, such as, software engineering, formal analysis, schedulability, execution time analysis, etc. However, in the context of software engineering and formal analysis, we have aimed at meeting the following objectives:

- Analyzable designs based on a formalized component model.
- Providing design support for behavior modeling of components.
- Providing design support for component-based development.
- Correlating specification models, and component-based designs.

Even if our work tackles some of the embedded system development challenges coined by the research questions, there certainly are limitations to our work, which we present in the following. To begin with, the formalization of ProCom has not been validated on a real-world example such that we could assess its verification capabilities. Also, while the formalization clearly attempts to avoid design ambiguities, by formalizing otherwise informal descriptions of ProCom semantics, the resulting designs might still be incomplete due to inherent limitations of the ProCom itself.

Even if the contribution with respect to the second objective, that is, our proposed component behavior patterns have been applied on an industrial case study, the turn-table system, further investigations would be useful. Also, the proposed patterns may not be sufficient for the abstract specification of complex functional behaviors of components. The same holds for the pattern-based design methodology for the overall system development.

Finally, the formal correlation framework proposed in the thesis has only been manually applied on a simple case study. For complex systems, the manual approach is clearly not feasible. Also, while the correctness of the inference rules proposed has been informally checked, this should be formally verified, for example, by logic-based reasoning.

The limited validations of the contributions made in the thesis are in the spirit of providing *proof-of-concepts*; however, all our solutions requiring further investigations and more extensive validation in order to establish their applicability, not to mention their potential benefits for the development of component-based industrial real-time systems.

6.2 Future Work

The initial focus of the future work will be to address the limitations of the contributions, as described in the previous section.

Next, we plan to extend the pattern-based design methodology, proposed in this thesis, for component development, as well as the complete system development. Additional useful patterns will be investigated and integrated within the design process. Also, the design methodology will be rigorously validated by applying it to industrial-strength case studies.

Due to the timed automata-based semantics, the ProCom designs can be analyzed in a dense-time underlying framework, as well as in a discrete-time one, since timed automata has been given a sampled semantics [59]. Hence, tools such as UPPAAL can be employed for early-stage verification of ProCom models, whereas discrete-time model-checkers, such as DTSpin [60], 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. We plan to consider these aspects for the future work related to formal verification of ProCom-based system designs.

Last but not least, we plan to extend the inference-rule driven methodology for relating abstract models towards verifying behavioral consistency between different embedded system abstractions. This involves applying simulation relation checking to prove (or disprove) conformance between non-identical trajectories. Further, we plan to investigate the integration of suitable proof assistant tools to support the underlying formal techniques.

Chapter 7

Overview of Papers

In this chapter, we present an overview of the research papers included in the second part of the thesis.

Paper A. "Analyzing a Pattern-Based Model of a Real-Time Turntable System". Davor Slutej, John Håkansson, Jagadish Suryadevara, Cristina Seceleanu, and Paul Pettersson. In proceedings of the 6^{th} International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA), pages 161-178, UK, March 2009.

Abstract: Designers of industrial real-time systems are commonly faced with the problem of complex system modeling and analysis, even if a component-based design paradigm is employed. In this paper, we present a case-study in for- mal modeling and analysis of a turntable system, for which the components are described in the SaveCCM language. The search for general principles underlying the internal structure of our real-time system has motivated us to propose three modeling patterns of common behaviors of real-time components, which can be instantiated in appropriate design contexts. The benefits of such reusable patterns are shown in the case-study, by allowing us to produce easy-to-read and manageable models for the real-time components of the turntable system. Moreover, we believe that the patterns may pave the way toward a generic pattern-based modeling framework targeting real-time systems in particular.

Contribution: This paper was written with equal contribution from all the authors. I specifically contributed to section three of the paper, proposing the behavior modeling patterns for components and also partly in section four in applying the proposed patterns to the case study.

Paper B. “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), pages 478-485, 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.

Contribution: The core of the paper, that is, section three, was written with equal contribution from the first two authors. I was the main author for the corresponding extended version of this paper, published as a (MRTC) technical report [61] .

Paper C. “Bridging the Semantic Gap between Abstract Models of Embedded Systems”. Jagadish Suryadevara, Eun-Young Kang, Cristina Seceleanu, and Paul Pettersson, In proceedings of the 13th International Symposium on Component Based Software Engineering (CBSE), Springer LNCS, vol 6092, Czech Republic, June, 2010.

Abstract: In the development of embedded software, modeling languages used within or across development phases e.g., requirements, specification, design, etc are based on different paradigms and an approach for relating these is needed. In this paper, we present a formal framework for relating specification and design models of embedded systems. We have chosen UML statemachines as specification models and ProCom component language for design models. While the specification is event-driven, the design is based on time triggering and data flow. To relate these abstractions, through the execution trajectories of corresponding models, formal semantics for both kinds of models and a set of inference rules are defined. The approach is applied on an autonomous truck case-study.

Contribution: I was the main author of this paper.

Paper D. "Pattern-driven Support for Designing Component-based Architectural Models", Jagadish Suryadevara, Cristina Seceleanu, Paul Pettersson, In proceedings of the 18^{th} IEEE International Conference on Engineering of Computer-Based Systems (ECBS), USA, April, 2011.

Abstract: The development of embedded systems often requires the use of various models such as requirements specification, architectural (component-based), and deployment models, across different phases. However, there exists little design support for obtaining suitable component-based designs that satisfy specified requirements and timing constraints. In order to provide guided support for the design process of embedded systems, we introduce several component templates, referred as patterns, which we also formally verify against relevant properties. To illustrate the usefulness of the approach, we have applied the proposed patterns to obtain a component-based design of a temperature control system.

Contribution: I was the main author of this paper.

Bibliography

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

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

[3] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[4] Object Management Group. UML 2.0 Superstructure Specification, The OMG Final Adopted Specification, 2003.

[5] Michael von der Beeck. Formalization of uml-statecharts. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, &#171;UML&#187; '01, pages 406–421, London, UK, UK, 2001. Springer-Verlag.

[6] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19:1:45–80, 2001.

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

[8] T. Bureš, J. Carlson, I. Crnković, S. Sentilles, and A. Vulgarakis. ProCom – the Progress Component Model Reference Manual, version 1.0. Technical Report MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[9] P.E. Black, K.M. Hall, M.D. Jones, T.N. Larson, and P.J. Windley. A brief introduction to formal methods [hardware design]. In *Custom Integrated*

Circuits Conference, 1996., Proceedings of the IEEE 1996, pages 377 –380, May 1996.

[10] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[11] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and symbolic model-checking of real-time systems. In *In Proc. of the 16th IEEE Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, 1995.

[12] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:394–406, 1992.

[13] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[14] G.J. Holzmann. The model checker spin. *Software Engineering, IEEE Transactions on*, 23(5):279 –295, May 1997.

[15] K.G. Larsen, Paul Pettersson, and Yi. Wang. Uppaal in a nutshell. *Int. J. on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[16] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *Proc. of 1st International Workshop on Formal Modeling and Analysis of Timed Systems, Lecture Notes in Computer Science. Springer–Verlag, 2003*. Springer-Verlag, 2003.

[17] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[18] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.

[19] 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.

[20] 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.

[21] 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.

[22] E. Bortnik, N. Trcka, A.J. Wijs, B. Luttik, J.M. van de Mortel-Fronczak, J.C.M. Baeten, W.J. Fokkinkc, and J.E. Rooda. Analyzing a x model of a turntable system using spin, cadp and uppaal. *Journal of Logic and Algebraic Programming*, 65(2):51–104, November-December 2005.

[23] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[24] X. Ke, K. Sierszecki, and C. 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 Computer Society, 2007.

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

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

[27] 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.

[28] R.J. Allen and D. Garlan. A formal basis for composing components. *ACM Transactions on SW Engineering and Methodology*, 1997.

[29] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, 1995.

[30] R. van Ommering, F. van der Linden, and J. Kramer. The Koala component model for consumer electronics software. In *IEEE Computer*, pages 78–85. IEEE, March 2000.

[31] 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.

[32] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli. The SAVE approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, May 2007.

[33] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-trauring, and D Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16, 1991.

[34] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[35] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3–12, 2006.

[36] Sverine Sentilles, John Håkansson, Paul Pettersson, and Ivica Crnkovic. Save-ide an integrated development environment for building predictable component-based embedded systems. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, September 2008.

[37] John Håkansson, Jan Carlson, Aurelien Monot, Paul Pettersson, and Davor Slutej. Component-based design and analysis of embedded systems

with uppaal port. In *6th International Symposium on Automated Technology for Verification and Analysis*, pages 252–257. Springer–Verlag, October 2008.

[38] R. Alur, D. Thao, J. Esposito, H. Yerang, F. Ivancic, V. Kumar, P. Mishra, G.J. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1):11–28, January 2003.

[39] V. Bos and J.J.T. Kleijn. Automatic verification of a manufacturing system. *Robotics and Computer Integrated Manufacturing*, 17:185–198, 2001.

[40] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and consistent equation semantics of hybrid chi. *Journal of Logic and Algebraic Programming*, 68(1-2):129 – 210, 2006.

[41] E. Bortnik, N. Trčka, A.J. Wijs, S.P. Luttik, J.M. van de Mortel-Fronczak, J.C.M. Baeten, W.J. Fokkink, and J.E. Rooda. Analyzing a χ model of a turntable system using Spin, CADP and Uppaal. *Journal of Logic and Algebraic Programming*, 65(2):51–104, 2005.

[42] Xu Ke, P. Pettersson, K. Sierszecki, and C. Angelov. Verification of comdes-ii systems using uppaal with model transformation. *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*, pages 153–160, Aug. 2008.

[43] Florence Maraninchi and Yann Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46:219–254, March 2003.

[44] Bo I. Sandén. The state-machine pattern. In *Proceedings of the conference on TRI-Ada '96: disciplined software development with Ada*, pages 135–142, New York, NY, USA, 1996. ACM.

[45] A. Burns and A. Wellings. *Concurrency in Ada*. Cambridge University Press, 1995.

[46] Joseph P. Loyall, Paul Rubel, Richard Schantz, Michael Atighetchi, and John Zinky. Emerging patterns in adaptive, distributed real-time, embedded middleware. In *9th Conference on Pattern Language of Programs*, September 2002.

[47] Cameron Maxwell, Tim O'Neill, and John Leaney. Formal architecture transformation using heuristics. In *Engineering of Computer-Based Systems, 2007. ECBS '07. 14th Annual IEEE International Conference and Workshops on the*, pages 15 –24, March 2007.

[48] M. Denford, John. Leaney, and Tim. OŃeill. Non-functional refinement of computer based systems architecture. In *Proceedings of the 11th IEEE International Conference and Workshop on Engineering of Computer-Based Systems*, pages 168–, Washington, DC, USA, 2004. IEEE Computer Society.

[49] Society of Automotive Engineers (SAE). Architecture analysis and design language (AADL), June 2006.

[50] M. Faugere, T. Bourbeau, R. De Simone, and S. Gerard. MARTE: Also an UML profile for modeling AADL applications. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 359 –364, 2007.

[51] F. Mallet, R. de Simone, and L. Rioux. Event-triggered vs. time-triggered communications with UML MARTE. In *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pages 154 –159, 2008.

[52] ATESST (Advancing Traffic Efficiency through Software Technology). East-ADL2 specification, March 2008.

[53] F. Mallet, M.-A. Peraldi-Frati, and C. Andre. Marte CCSL to execute East-ADL timing requirements. In *Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC '09. IEEE International Symposium on*, pages 249 –253, March 2009.

[54] Pavel Krčál, Leonid Mokrushin, P.S. Thiagarajan, and Wang Yi. Timed vs time triggered automata. In Philippa Gardner and Nobuko Yoshida, editors, *Proc. of CONCUR'04.*, number 3170 in Lecture Notes in Computer Science, pages 340–354. Springer–Verlag, 2004.

[55] Joseph Sifakis, Stavros Tripakis, and Sergio Yovine. Building models of real-time systems from application software. In *In Proceedings of the IEEE Special issue on modeling and design of embedded*, pages 100–111. IEEE, 2003.

[56] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.

[57] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[58] Bernhard Schätz, Florian Hölzl, and Torbjörn Lundkvist. Design-space exploration through constraint-based model-transformation. In *Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '10, pages 173–182, Washington, DC, USA, 2010. IEEE Computer Society.

[59] 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.

[60] 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.

[61] 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.

II

Included Papers

Chapter 8

Paper A: Analyzing a Pattern-Based Model of a Real-Time Turntable System

Davor Slutej, John Håkansson, Jagadish Suryadevara, Cristina Seceleanu, Paul Pettersson
In proceedings of the 6th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2009), pages 161-178, March 2009.

Abstract

Designers of industrial real-time systems are commonly faced with the problem of complex system modeling and analysis, even if a component-based design paradigm is employed. In this paper, we present a case-study in formal modeling and analysis of a turntable system, for which the components are described in the SaveCCM language. The search for general principles underlying the internal structure of our real-time system has motivated us to propose three modeling patterns of common behaviors of real-time components, which can be instantiated in appropriate design contexts. The benefits of such reusable patterns are shown in the case-study, by allowing us to produce easy-to-read and manageable models for the real-time components of the turntable system. Moreover, we believe that the patterns may pave the way toward a generic pattern-based modeling framework targeting real-time systems in particular.

8.1 Introduction

Developing industrial real-time systems is difficult and sets high requirements to system safety and reliability. The short development cycles demand a reliable engineering method, with predictable costs. The state-of-the-art is dominated by an ad-hoc mixture of methods and tools, and system validation is mostly done by extensive testing at the implementation level. However, testing is done already too late in the design process, and bugs may still exist even in well-tested models. In this context, techniques for managing complexity and ensuring critical system properties during design become a necessity.

A promising design approach is to employ a *formal component-based* development technique. In such an approach, components are introduced as executable software units that can be deployed into a system. One of the key issues of realizing the component-based software paradigm is to ensure that the separately specified components do not conflict with each other when composed, resulting in blocking the system. A potential solution to this issue is *formal modular verification* of component-based software via *model checking*.

In this paper, we present a case-study in formal modeling and analysis of a real-time, component-based turntable system, for which the components are described in the SaveCCM language [1]. For verification, we use an integrated development environment for SaveCCM, connected via a plug-in with UPPAAL PORT, an extension of the model-checker UPPAAL, which implements a partial order reduction technique [2] for efficient model-checking. The technique exploits the topology of the network of components and consequently improves the scalability of the verification method.

Our experience with this case-study and other similar examples is that, beside making the model-checking efficient, an as demanding task is to produce manageable and easy-to-grasp design models for components and their composition. This has motivated us to try to extract some common behavioral patterns that occur frequently in the design of real-time systems, and represent them in a finite-state-machine like notation. Such notation lets us apply these patterns at high-levels of software development, as shown in the paper, while simplifying the produced models. We believe that employing patterns in designing component-based systems might also help in documenting the associated software, through pattern-based reverse engineering. However, this is out of the scope of this paper.

General purpose program design patterns are well-known in the object-oriented design community for a while now [3]. Nevertheless, in the design of component-based real-time systems, some different aspects might need to be represented in the modeling patterns; for instance, the semantics of our SaveCCM components is a read-execute-write semantics, hence a run-to-completion pattern can prove beneficial in the design. Similarly, the reusable modeling of the sequence of visited states during the execution of a component, or reducing the time-wise non-determinism of the real-time component behavior, by providing systematic means to associate a *deadline* with the behavior, through a pattern, might also help the designer in the modeling phase. In this paper, we introduce the just mentioned abstractions of common real-time component behaviors, as the run-to-completion, history, and execution-time patterns, respectively. Next, we apply them in modeling the component-based turntable production cell.

The remainder of the paper is organized as follows. In section 8.2, we briefly recall the basics of the SaveCCM language used for modeling the components in our case-study. The three modeling patterns are introduced and described as finite state machines in section 8.3, after which we present the real-time turntable production cell example, including the formal models of the constituent components, in section 8.4. The system's formal requirements and verification results are displayed and discussed in sub-section 8.4.3. We compare our approach to related ones, in section 8.5. Finally, section 11.9 concludes the paper and outlines possible directions for future work.

8.2 SaveCCM

In this section we briefly present the Save component modeling language [1], which will be used in the case study of this paper. The language is part of a larger framework, called SaveCCM, for component-based design of real-time and embedded system [4]. The SaveCCM language consists of a graphical syntax and an associated formal semantics. Due to space limitation, the presentation in this section is restricted to a short informal overview of SaveCCM. For a complete description of the language we refer the reader to [1].

In SaveCCM, systems are built from interconnected components with well-defined interfaces consisting of input and output ports. The communication style is based on the pipes-and-filters paradigm, but with an explicit separation of data transfer and control flow. The former is captured by connections between *data ports* where data of a given type can be written and read, and the latter by *trigger ports* that control the activation of components. Figure 8.1(a) shows an example of the graphical SaveCCM notation. Triangles and boxes denote trigger ports and data ports, respectively.

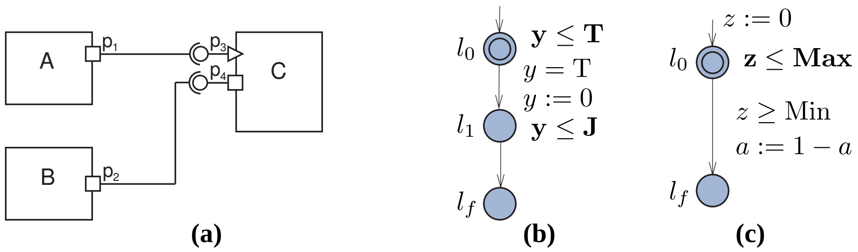

Figure 8.1: An example of **(a)** a composition where components A, B and C are composed by connecting port p_1 to p_3, and p_2 to p_4, and timed behaviors: **(b)** a clock with period T and jitter J, **(c)** a computation updating data variable a after between Min and Max time units.

A component remains passive until all input trigger ports have been activated, at which point it first reads all its input data ports and then performs the associated computations over this input and an internal state. After this, the component writes to its output data ports, activates the output trigger ports, and returns to the passive state again. This strict "read-execute-write" semantics ensures that once a component is triggered, the execution is functionally independent of any concurrent activity.

Components are composed into more complex structures by connecting output ports to input ports of other components. In addition to this "horizontal" composition, components can be composed hierarchically by placing a collection of interconnected components inside an enclosing component. From the outside, such a composite component is indistinguishable from other component where the behavior is given by a single model or piece of code.

To support analysis of SaveCCM models, it is required that each component is associated with a behavioral model consisting of a timed automaton [5] with a distinct exit location (see Figure 8.1(b-c)), and a mapping between component data ports and the internal automata variables. When a component is triggered, the port values are copied to the internal variables of the timed behavior which then proceeds as specified in the timed automaton. Whenever it reaches the exit location, variable values are copied to the output ports according to the given mapping, and the output trigger port is activated.

The timed automata modeling language used in SaveCCM is based on the language used in the UPPAAL tool [6]. It extends the timed automata language originally introduced by Alur and Dill [5] with a number of features that will be used in the case study, including: global and local bounded integer variables and arithmetic operations over such variables, arrays, and a small C-like programming language that can be used to define functions and predicates. For a detailed description of the timed automata language, we refer the reader to [7].

8.3 Component Modeling Patterns

A modeling pattern is a way of designing a model with a clearly stated intent and structure. In this section, we propose three modeling patterns for common behaviors of real-time components, in order to ultimately provide the designer with useful abstraction mechanisms for the high-level modeling and analysis of CB real-time systems. We chose to define the patterns by a finite-state-machine like (FSM) notation, which we call *Pattern-FSM* (or PFSM) in this paper. The patterns can be instantiated, separately or in combination, in specific formal frameworks, to increase the readability of the models and their suitability for verification. To justify our claim, in section 8.4, we apply the proposed patterns, as combinations, to the CB modeling of an industrial real-time turntable system (see for instance Figure 8.10). The analysis framework is the Timed Automata (TA) language of UPPAAL [7, 6].

Generic PFSM Definition and Graphical Notation. Let V be a set of data variables, G be a set of boolean conditions (*guards*) over V, and A a set of actions that update the variables. Then PFSM is a tuple $\langle S, \mathsf{start}, \mathsf{exit}, E, Att\rangle$, where S is a set of states, start is the *entry* state, exit is the *exit* state, $E \subseteq S \times G \times A \times S$ is the set of transitions between states, and *Att* is a set of timing attributes, e.g. execution time, deadline, etc.

The execution of a PFSM starts in the special control state start. At a given state, an outgoing transition may be executed only if its associated guard evaluates to true; in this case we say that the transition is *enabled*. In case more than one outgoing transitions are enabled, one can be executed non-deterministically. A filled circle denotes the start control state and a semi-filled circle denotes the exit control state (see Figure 8.2). Different attributes of a PFSM, e.g. execution time, deadline etc. can be added to the graphical representation of a PFSM model (e.g. Figure 8.7).

8.3.1 Run-to-Completion Pattern

In the run-to-completion (RTC) execution model, the component is executing in indivisible steps, without interruption from any concurrent activity. The key advantage of the RTC semantics is simplicity and guaranteed absence of

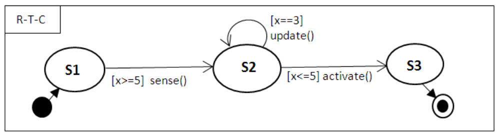

Figure 8.2: PFSM specification of a component behavior

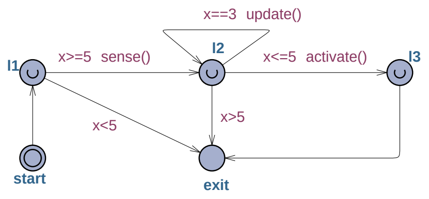

Figure 8.3: An equivalent timed automata model with run-to-completion pattern

deadlocks. Another advantage is that it might prune away unnecessary interleavings, thus speeding up formal verification and bringing the model closer to implementation. The pattern is commonly used in high-level behavioral modeling languages like Statecharts and its variants [8, 9]. In Statecharts, the events are handled in an RTC manner, along possibly compound transitions (i.e., paths of adjacent arrows).

Pattern description. In this pattern, we assume that the component execution proceeds with changing states by firing enabled transitions until it reaches a state for which no outgoing transitions are enabled. At such a point, the execution terminates.

To implement the pattern, one needs to translate the corresponding PFSM into a timed automaton (TA). Run-to-completion can be implemented by introducing new edges in the automaton, which describe termination of component execution. Let L be the set of locations $l_i,\ i \in \{1,..,n\}$ in the corresponding TA. For each location $l_i \in L$, we assume that $g_j,\ j \in \{1,..,m\}$ are the guards of the respective outgoing edges. The exit edge from l_i connects l_i with the exit location. The guard of the l_i exit edge is $\neg(\bigvee_j g_j)$.

Example. Figure 8.2 represents a PFSM specification of a simple component behavior obeying our run-to-completion pattern. Figure 8.3 describes the equivalent behavior as a timed automaton, which serves as the pattern implementation. The states S1, S2, and S3 of the PFSM are mapped onto locations

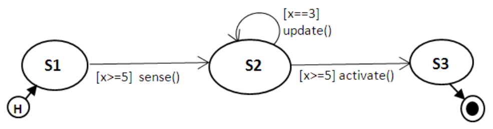

Figure 8.4: PFSM specification of a component behavior with history

l1, l2, and l3, respectively, in the equivalent TA.

8.3.2 History Pattern

Execution history is a core feature of behavior modeling techniques [10, 8]. The history mechanism of a behavior remembers which state was last visited during execution, before exiting. This state can then be re-entered next time the execution re-starts. In the hierarchical state-machine modeling of Statecharts [8], an inner state may be exited and re-entered directly, by using the history mechanism. A similar approach is adopted in CHARON, a formal modeling framework for hybrid systems [10].

Pattern description. The pattern provides a mechanism to remember the execution history in the behavioral models of components. Assuming the execution as a sequence of states, the pattern has means of remembering the last state, or a particular state for that matter, reached during execution. Hence, the next time, the execution can resume from the state stored through the history mechanism. Similar to Statecharts, in a PFSM representation, the history mechanism is denoted as an H within a circle, and acts as the start state.

The pattern is implemented as a TA, by using an integer variable H, which is updated along each edge connecting any states different from the start, and exit states, with the corresponding location identifier. Special edges connect the start state to each of the states of interest, while appropriately testing the variable H. In addition, exit edges connect each state of interest to the exit control state. Variable H can be re-initialized appropriately when entering a specified final location.

Example. Figure 8.4 represents a component behavior with history pattern. The history is denoted by the encircled H symbol, in the start state. In Figure 8.5, we give the equivalent behavioral model as a TA, which implements the history pattern. The states in Figure 8.4 are mapped onto locations 1, 2, 3 in the TA. Variable H is initialized to an initial location, i.e., H = 1. The

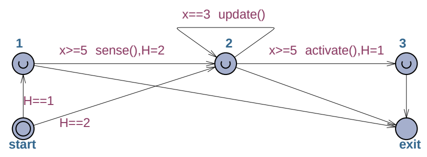

Figure 8.5: A timed automata behavior with history pattern

edges that connect the start location to locations 1, and 2 are due to the pattern, and are guarded by conditions H==1, and H==2, respectively. Also, the history variable H is updated with the location identifier along each edge entering that respective location (edges that leave and enter the same location may be skipped, e.g., location 2 in Figure 8.5). Finally, H is re-initialized at location 3 of Figure 8.5.

8.3.3 Execution-Time Pattern

For embedded and real-time systems, it is often interesting to specify and analyze the best or worst execution time of components. The variation in execution time also gives rise to, e.g., non-deterministic timing, jitter, and varying end-to-end timing, which represent phenomena that are important to analyze (and master) at design time. In the following, we introduce a pattern for specifying the best and worst execution times of components.

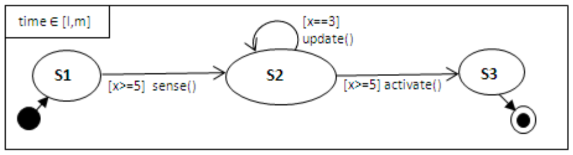

Figure 8.6: Annotation of time attributes on PFSM models for execution-time pattern

Pattern description. In this pattern, we assume that the total accumulated time of executing a component is within an interval where the lower and upper bounds are the shortest and longest possible execution times, respectively. Hence, the component will produce output (data and trigger) at some time instance, in the interval.

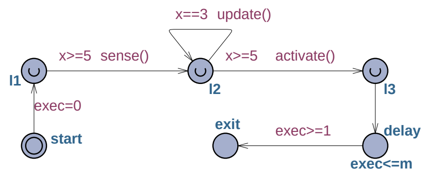

Figure 8.7: A timed automata behavior with execution-time pattern

We also assume that the component is annotated with an interval specifying the lower and upper bound on the execution time. To implement the pattern, we use a dedicated clock, say exec, which is used to measure the time since the component was triggered. The clock is reset on the edge outgoing from location start. We further introduce a location, say delay, and an edge from location delay to the exit location. Location delay is annotated with an invariant over exec, corresponding to the upper bound of the execution interval, whereas the exit edge is decorated with a guard corresponding to the lower execution bound.

Example. Figure 8.6 represents a PFSM specified using the execution time pattern. Its execution time is in the (closed) interval $[l, m]$. Figure 8.7 shows a timed automaton implementing the pattern. Note that when the exit location is reached, the value of clock delay is in the interval $[l, m]$.

8.4 Turntable Production Cell

In industry automation, a production cell is a part of an overall production system — a factory. In this section, we present a formal model of a turntable production cell, previously described in [11, 12]. The case study is designed using the component framework described in Section 8.2 and the patterns introduced in Section 8.3. By employing the patterns, we get simple and understandable component models for our case-study, as shown in the following subsections.

The turntable cell is illustrated in Figure 8.8. It consists mainly of a rotary disc with four product slots. A product is *loaded* into a slot at position 0, and is then rotated to position 1 where it is *drilled*. It is then rotated into position 2 where it is *tested*, and finally to position 3 where it is *unloaded* (or possibly left to be redrilled in the next cycle). The positions are aligned with various tools for loading, drilling, testing, and unloading.

Drilling and testing are the most critical tool positions, as the overall pur-

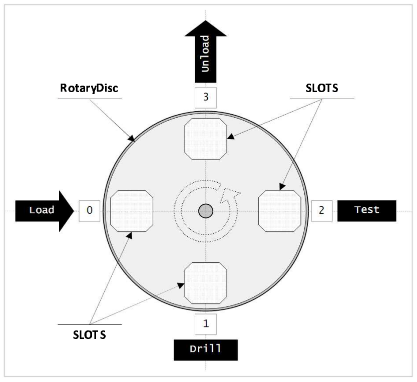

Figure 8.8: Schematic diagram of a Turntable system

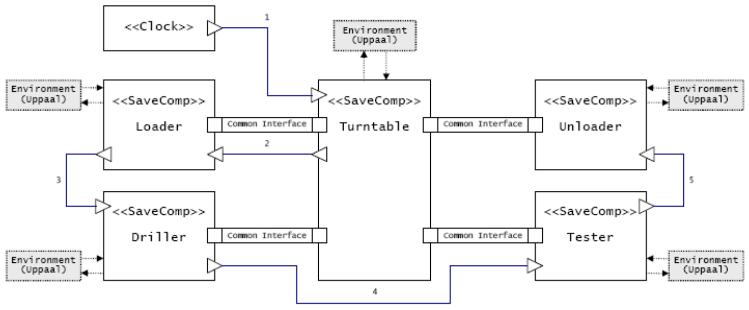

Figure 8.9: Software architecture design layout of Turntable system

pose of the production cell is the verified drilling of products that flow through the cell. All slots of the rotary disc may be occupied at the same time, and products are processed in parallel. When a cycle completes, meaning that all positions complete their functionality, the rotary disc rotates 90 degrees thus positioning the products for the next phase of processing. As the rotation is initiated by signals from tools that are not time deterministic, there is no fixed period between rotation of the slots.

Table 8.1: Common interface for components Loader, Driller, Tester, and Unloader

Port	Data type	Description
status	int	An input representing the current known status of the product in the tool position (0 indicates an empty slot).
result	int	An output that holds the status of the product after processing.
start	bool	An input that initiates tool processing.
finished	bool	An output that signals when the tool controlled by the component has completed its processing.

8.4.1 System Design

Following the informal description of the system, we can identify the system as consisting of five main software components: Turntable, Loader, Driller, Tester, and Unloader, corresponding to the functionalities of the cell. The components interact with several sensors and actuators, such as position sensors, clamping, and drilling devices, which do not require explicit modeling. Further, as we focus on modeling and analysis of the functional and timing behavior of the system, we make assumptions regarding error situations, e.g., no fault situations like broken tools, etc. This simplifies the system model without loss of generality.

We now describe in detail the software components in terms of their interfaces and behaviors. Figure 8.9 shows the software architecture of the turntable system. An interface of a component defines the access point to its behavior, in our case in terms of data ports and trigger ports. The Turntable component acts as a central controller in the system, and all other components are independent of each other and have a similar interface with Turntable. The common interface approach supports reuse, as well as the flexibility to extend or modify the system architecture. We define a common interface for each component, except Turntable, as shown in Table 8.1.

Data flow is defined by connections between data ports, within the common interfaces and with external sensors and actuators. The control flow is modeled separately from the data flow, by connections between triggering ports. As illustrated by Figure 8.9, the flow starts from the Clock component and ends at the Unloader.

The component behaviors are modeled as finite state machines under the assumption of the modeling patterns defined in previous section. The history and the run-to-completion patterns are combined to achieve the modeled finite state machine behavior of the components, eventhough the components will be executed in a time-triggered fashion. The execution time pattern is applied to model the time required to execute each component. As such, the models present intuitive conceptual modeling retaining the analysis capability of the underlying formalism, i.e., timed automata. The modeled behaviors execute under the semantics of SaveCCM component model and the semantics of the patterns. In the following, we describe each of the component behaviours along with their associated functions and predicates, defined in terms of variables associated with the data and trigger ports of the corresponding component.

The Turntable Component

The interface of the turntable controller consists of two trigger ports, a sensor input, an actuator output, and four instances of the *common interface*. A clock component generates trigger signals to periodically activate Turntable, which in turn activates the Loader component. The actuator output aRotate is connected to a motor turning the rotary disc, and the sensor inputsRotated senses when the rotation is completed. The behavior of the Turntable component coordinates the rotation of the disc with the execution of other components.

Initially it rotates the disc, and sets ports of other components appropriately. It then waits for the other components to signal that their processing has stopped, before restarting the main loop by turning the disc again[1]. Starting from an empty system, it will take at least four rotations for all components to work in parallel. The first rotation only starts processing of the Loader, which then loads the first product onto the table. In addition to controlling the rotation of the disc, the component also maintains status information for each position. The status information is shifted one step each time the table rotates. The detailed behavior is modeled in Figure 8.10, in terms of associated functions and predicates (listed in Figure 8.11). The internal variables $\mathsf{status}_i, \mathsf{start}_i, \mathsf{finished}_i, \mathsf{result}_i$ represent the data values of the corresponding *common* interface ports of position i.

[1]Hence, even though Turntable is triggered periodically, the period of the rotation of the disc depends on the processing time in the four slots.

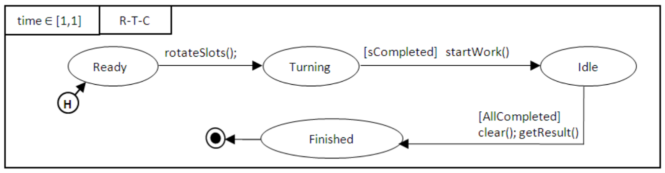

Figure 8.10: Behavioural model Turntable component.

```
rotateSlots() **is**
    temp : **int** := $\text{status}_0$
    aRotate := true
    $\text{status}_0$ := $\text{status}_3$ ; $\text{status}_3$ := $\text{status}_2$
    $\text{status}_2$ := $\text{status}_1$ ; $\text{status}_1$ := temp
**end**

startWork() **is**
    **for positions** $i$ **do** $\text{start}_i$ := true
**end**

getResult() **is**
    **for positions** $i$ **do** $\text{status}_i$ := $\text{result}_i$
**end**

clear() **is**
    **for positions** $i$ **do** $\text{start}_i$ := false
**end**

allCompleted **iff** $\forall i$ : $\text{finished}_i$
```

Figure 8.11: Functions and predicates used by Turntable.

The Loader Component

As mentioned, Loader shares a common interface with, and receives a trigger, from Turntable. It also has a trigger output to the Driller, sensor input sLoaded, and actuator output aLoad. The behavioral model is shown in Figure 8.12. When triggered the component checks the status of the slot at position 0. If a previous product is present, forwarded by the Unloader for reprocessing, the product is left in the slot for repeated drilling. Otherwise a new product is loaded into the slot, to be drilled in the next cycle.

The Driller Component

Figure 8.13 shows a model of the Driller component behavior, which interacts with actuators and sensors for clamping and drilling the product. When triggered the component checks the status of the slot at position 1. If empty, the driller does nothing, otherwise the product in the slot is fixated (clamped), the drill starts spinning and is lowered. When the drilling is completed, the drill is

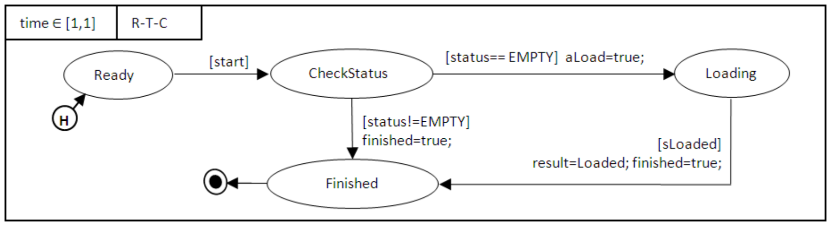

Figure 8.12: Behavioral model of Loader component.

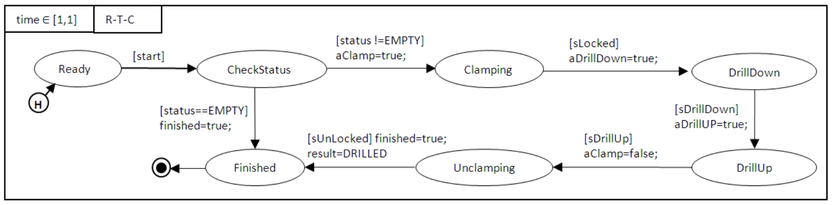

Figure 8.13: State machine model of the Driller component.

lifted and stopped, and the status of the slot is updated accordingly.

The Tester Component

The behavioral model of Tester is shown in Figure 8.14. Its input trigger is received from Driller, and its output trigger output is sent to Unloader. Similar to the driller, it interacts with actuators and sensors to move a tool into the product. The tool of the tester is a sensor sTesterDown, that measures the hole within 2 time units since the beginning of the test process. When triggered the

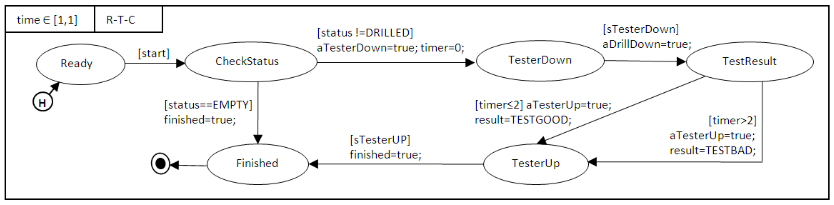

Figure 8.14: State machine model of the Tester component.

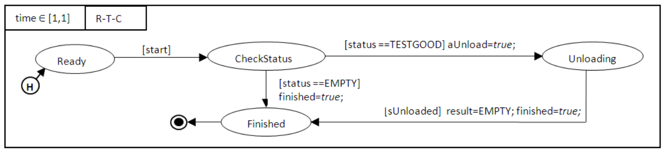

Figure 8.15: State machine model of the Unloader component.

component checks the status of the slot at position 2. If empty, it does nothing, otherwise it measures the hole drilled in the product, and updates the status according to its verdict.

The Unloader Component

Figure 8.15 shows a model of the Unloader behavior. The status of the drilled product at position 3 indicates the verdict determined by the previous tester component. If the product was faultily drilled, it is not unloaded, otherwise, the component activates an actuator to unload the product. If the slot is empty, as in initial rotations, the Unloader does nothing.

8.4.2 Modeling a Closed System

For verification purposes we define a closed system, that is, a system with no inputs or outputs. A closed model of the turntable is created by composing the turntable controller software with an UPPAAL timed automata model of the environment that is affected by actuators, and affects sensors. The software architecture of the turntable controller is presented in Figure **??** (as it appears in the SaveCCM syntax in the Save-IDE). The behavior of each component, as modeled in the previous section, is translated into TA, following the modeling patterns presented in section 8.3.

The environment of the turntable control software is modeled with appropriate abstractions of the complex real world aspects, in such a way that the behavior (and timing) of the real physical environment is included in the model. Further, as mentioned earlier, the model is done under the assumption of normal behavior, meaning no exception handling or error conditions such as faulty sensors or actuators may occur. The environment of the turntable system is modeled as timed automata (TA) in the UPPAAL tool. The environment essentially consists of the actuators and sensors associated with the system and its components. Due to space limitation, we leave out some of the environment

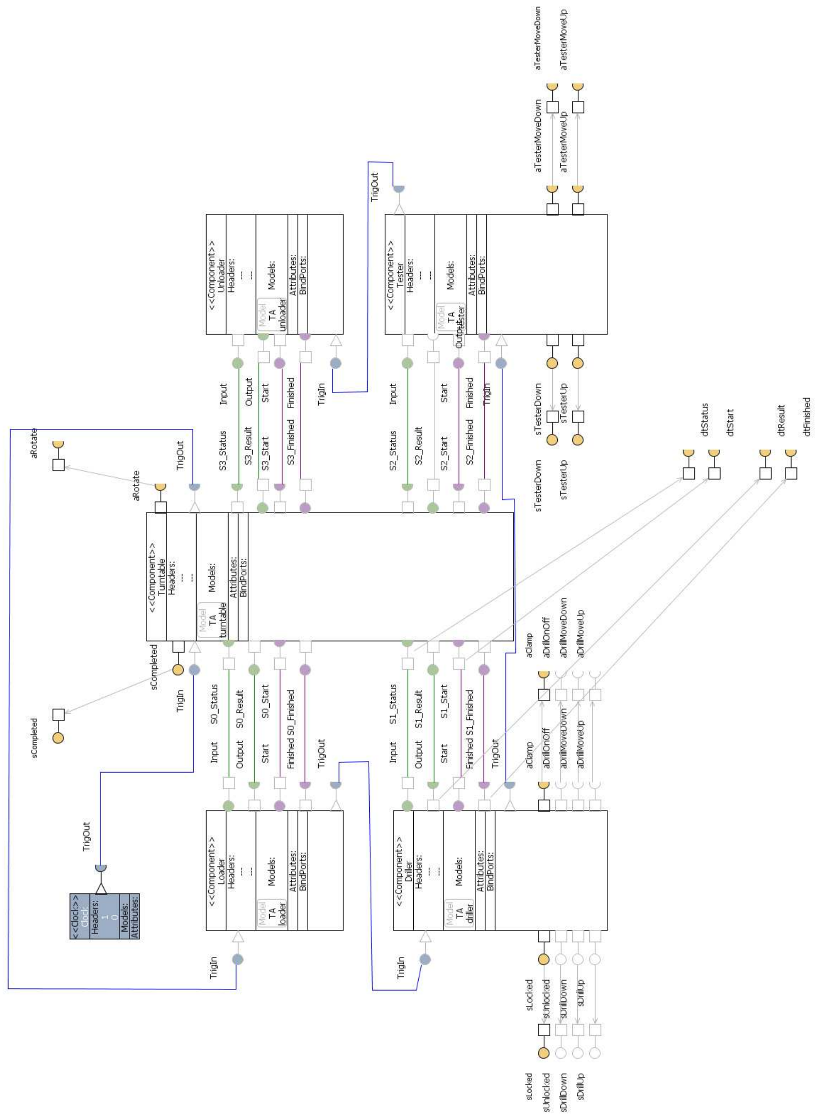

Figure 8.16: Control structure and system architecture of the turntable system as modeled in Save-IDE.

automata, and we refer the reader to our recent work [12] for a more detailed environment model.

The communication interface between the system and its environment is facilitated by shared variables. These variables correspond to the communication ports between the modeled system software and its sensors and actuators, as well as test automata that drive the verification process. The interface, and its initialization, is given in Table 8.2. To simplify the modeling process, and reduce the state space of the model, all aspects of a system are not modeled explicitly. Instead, models focus on critical aspects of the system. The environment model used for the formal verification of the turntable consists of the behaviors Disc, Clamp, Drill, and TestTool.

The drilling tool is modeled in terms of its two controllable parts: Clamp and Driller. The behavior of these environment models are presented in Figures 8.17 and 8.18, respectively. The function of the clamp is to lock the product in place so that the drilling can be carried out. The timed automaton is initially in the location UnLocked, and transitions to the location Locking when the edge guard aClamp goes high (value becomes 1). It can remain in the location Locking as long as the associated invariant $\mathsf{claCLK} \leqslant \mathsf{ClampTime}$ holds. The same happens when the clamp is in location UnLocking. This models the continuous behavior of the Clamp.

The function of Driller is to make holes in the product. The timed automaton (Figure 8.18) is initially in the DrillUp location, and transitions to DrillerMovingDown when the guard aDrillMoveDown goes high. It can remain in this location as long as the associated invariant $drillCLK \leqslant \mathsf{MaxDownTime}$ holds to model the maximum time the drilling can take place. The same happens when the drill is in location DrillerMovingUp. The driller moves out from

Table 8.2: Interface of the environment components

TA	*Variables*	*Data type*	*Initially*
Disc	aRotate, sCompleted	bool	false
Clamp	aClamp, sLocked, sUnlocked	bool	false
Drill	aDrillDown, aDrillUp sDrillDown, sDrillUp	bool	false
TestTool	aTesterDown, aTesterUp sTesterDown, sTesterUp	bool	false

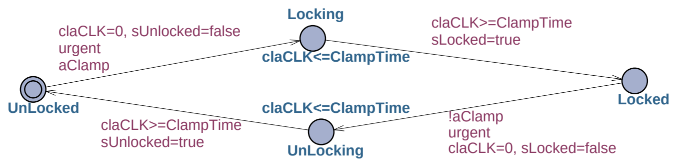

Figure 8.17: Behavior of the Clamp environment model.

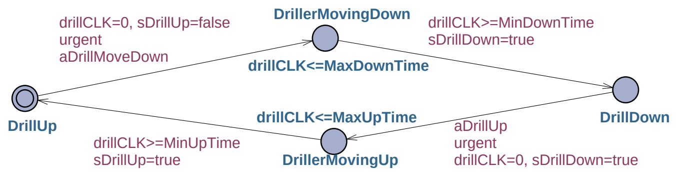

Figure 8.18: Behavior of Drill of the environment model.

the continuous behavior of drilling down or drilling up after MinDownTime or MinUpTime, respectively.

The TestTool works similarly to the drill, moving down by command from an actuator until a sensor is activated, and then moving up again by command from a different actuator until the corresponding sensor is activated. Also Disc is modeled with two states, wait and turning. The transition from wait to turning is initiated by the actuator aRotate, clears the sensor value sCompleted, and resets a clock ensuring the transition back to wait within TURN_TIME time units, when the sensor value sCompleted is also set.

8.4.3 Requirements and Verification

In this section, we present the verification aspects of the turntable system. The work has been performed in the SAVE-IDE, an integrated development environment for SaveCCM. For modeling, the Save-IDE provides graphical editors for architectural and behavioral modeling. For system (symbolic) simulation and verification by model-checking, the tool UPPAAL PORT [13, 2], an extension of UPPAAL [6], is integrated through a plug-in. The representation of the system architecture and component behaviors is represented in the SaveCCM XML file format [1], and the environment is stored in an UPPAAL XML file. UPPAAL PORT connects system inputs and output to global variables in the environment model.

A set of properties concerning the safety and liveness of the Turntable control system have been verified. In UPPAAL, liveness properties can be specified as *leads to* properties in the form $P \leadsto P'$, meaning that if a system has reached a state with P satisfied, it will eventually reach a state where P' is satisfied. We discuss a few representative properties below. The first property specified is:

$$A\Box\, \neg\textbf{deadlock} \tag{8.1}$$

Property 8.1 is a safety property, specifying the absence of deadlock situations. A deadlock occurs when the system can not progress further. In a real-time system, this is often caused by two tasks mutually excluding each other from acquiring a resource (e.g. semaphore). It can also be caused by a fault in the environment model. The property is verified as listed above. The A is a universal quantifier, and refers to the property to be verified on all execution paths of the statespace. The box $\Box$ is a universal quantifier over all states in a path. The states are defined by values of all variables as well as locations of automata. The keyword *deadlock* represents a state in the execution where there is no outgoing (delay or action) transition. The turntable system is verified to be *deadlock free*.

The absence of a deadlock does not mean that the system is guaranteed to make progress. The control system could be continuing with the component trigger without the components progressing through their respective finite state machines. The following set of properties verify that the turntable system is progressing. It checks that the central component Turntable continuously moves between Idle and Turning states. This is specified using *leads to* properties. The diamond $\Diamond$ is an existential quantifier over states in the path, meaning that the property is eventually satisfied by a state in the path (all paths in this case).

$$\begin{aligned} A\Diamond\, \mathsf{Turntable.Turning} \qquad & \mathsf{Turntable.Turning} \leadsto \mathsf{Turntable.Idle} \quad (8.2)\\ & \mathsf{Turntable.Idle} \leadsto \mathsf{Turntable.Turning} \end{aligned}$$

The properties 8.2 establishes that the component Turntable always progresses. This is possible only when the individual components too are progressing following the design strategy. The progress of individual components can be verified as below.

$$\mathsf{Loader.Ready} \leadsto \mathsf{Loader.Finished} \tag{8.3}$$

The above leads-to property 8.3 verifies that Loader always progresses. We can verify a similar property for all other components. Further, we verify an important safety property stating that when the Turntable component is executing, no other components are executing:

$$A\Box(\mathsf{Turntable.Turning} \Rightarrow (\mathsf{Loader.Ready} \wedge \mathsf{Tester.Ready} \wedge \mathsf{Unloader.Ready} \wedge \mathsf{Driller.Ready})) \quad (8.4)$$

Property 8.4 models the fact that while the Turntable is turning the other components are just waiting in their Ready location, according to the design strategy.

Property 8.5 establishes a state correspondence between an environment component and the corresponding SaveCCM component. The property ensures that whenever the Turntable is not turning, the Disc component is not turning either:

$$A\Box(\neg\mathsf{Turntable.Turning} \Rightarrow \neg\mathsf{Disc.Turning}) \quad (8.5)$$

The next property (8.6) specifies that the control model never sends two conflicting signals to its environment. Here, it checks that the system does not activate both actuators associated with the Driller component, simultaneously, as they move the Drill in opposite directions:

$$A\Box\,\neg(\mathsf{Driller.aDrillDown} \wedge \mathsf{Driller.aDrillUP}) \quad (8.6)$$

8.5 Related Work

There are a number of component based development (CBD) frameworks for embedded systems described in the literature. The BIP framework and the toolkit IF [14] are intended for predictable embedded systems development by supporting *correctness-by-construction* and compositional verification. While BIP offers bottom-up design of systems, our approach supports CBD in a bit more pragmatical traditional top-down design, with support of modeling in Save-IDE [15] and formal verification using the UPPAAL PORT toolkit [13, 6].

The Charon toolkit [10] supports modular specification of embedded systems, based on the notions of *agents* and *modes*, for architectural and behavioral specifications, respectively. Our behavioral specification language of components shares some features of the modes in Charon, but without hierarchy, and in our approach the execution history of a component is provided by using a simple design pattern.

The Statemate toolkit [16] is an early working environment for the development of complex reactive systems. Modularity of the system development is provided in terms of different *views*, such as structure, functionality, and behavior. Our approach for behavior specification of components (modules in Statemate) is similar to the Statecharts [17], the behavioral language of Statemate. Though not hierarchical, our FSM notation for component behaviors (see Section 8.3), combined with the patterns proposed in this paper, is similar to the Statechart features run-to-completion and execution history.

The case study of Turntable production system, presented in this paper, has previously been analyzed using different methods and tools. In [18], a turntable model is specified in χ [19], a simulation language for industrial systems, and translated into Promela, the input language of the Spin model-checker to verify several properties of the model. In [11], a χ model of the turntable system was translated into the specification languages of three model-checkers: CADP, Spin, and UPPAAL comparing both the ease of conversion, the expressiveness of each of the specification languages, and the abilities and performances of the respective model-checkers. In [20], the turntable production system was implemented in the COMDES-II component-based software framework. The authors developed a semantic transformation of the COMDESS-II model into an UPPAAL timed automata model, allowing for formal verification of a set of properties similar to those in [11].

8.6 Conclusion

In this paper, we have presented how the SaveCCM component-based approach for development of embedded systems has been applied in a case study, to model and verify an industrial turntable production system. We have presented a component-based system architecture model, as well as the detailed behavioral models of the system components. To produce a manageable and easy-to-grasp design model of the turntable, we have used three simple, but useful, design patterns. The finite behaviors of components are specified in a finite state machine notation, using two design patterns for encoding run-to-completion semantics, and history states. Timing is introduced using a third design pattern for specifying the execution time and order of components. We also describe how the design specifications are syntactically transformed into the modeling framework used in SaveCCM, for further analysis using UPPAAL PORT.

Throughout the case study, we have been using Save-IDE and its connection to UPPAAL PORT, for editing models, as well as for performing (symbolic) simulation, and verification by model-checking. As a modeling result, we believe that we have produced a very intuitive component-based model of the turntable system. As verification results, we have shown that the system model satisfies all the requirements specified for the system, formalized as safety and liveness properties in TCTL.

As future work, we intend to develop an enriched behavioral modeling language and formal analysis support for the successor of SaveCCM, called ProCom. The language will be based on the design patterns described in this paper, and possibly on other newly developed, more involved patterns that might prove useful in simplifying both the formal models and their verification.

Bibliography

[1] J. Carlson, J. Håkansson, and P. Pettersson. SaveCCM: An analysable component model for real-time systems. In *Proceedings of the 2nd Workshop on Formal Aspects of Components Software (FACS 2005)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

[2] J. Håkansson and P. Pettersson. Partial order reduction for verification of real-time components. In *Proc. of 1st International Workshop on Formal Modeling and Analysis of Timed Systems*, 2007.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing. AddisonWesley Publishing Company, Reading, Massachusetts, 1995.

[4] 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.

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[6] K.G. Larsen, Paul Pettersson, and Yi. Wang. Uppaal in a nutshell. *Int. J. on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[7] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[8] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, July 1997.

[9] Bran Selic. An efficient object-oriented variation of the statecharts formalism for distributed real-time systems. In *Proceedings of the 11th IFIP International Conference on Computer Hardware Description Languages and their Applications - CHDL '93*, volume A-32 of *IFIP Transactions*, pages 335–344. North-Holland, 1993.

[10] R. Alur, D. Thao, J. Esposito, H. Yerang, F. Ivancic, V. Kumar, P. Mishra, G.J. Pappas, and O. Sokolsky. Hierarchical modeling and analysis of embedded systems. *Proceedings of the IEEE*, 91(1):11–28, January 2003.

[11] E. Bortnik, N. Trčka, A.J. Wijs, S.P. Luttik, J.M. van de Mortel-Fronczak, J.C.M. Baeten, W.J. Fokkink, and J.E. Rooda. Analyzing a χ model of a turntable system using Spin, CADP and Uppaal. *Journal of Logic and Algebraic Programming*, 65(2):51–104, 2005.

[12] Davor Slutej. Component-based modeling and analysis of embedded systems. Master's thesis, Department of Computer Science and Engineering, Mälardalen University, September 2008.

[13] John Håkansson, Jan Carlson, Aurelien Monot, Paul Pettersson, and Davor Slutej. Component-based design and analysis of embedded systems with uppaal port. In *6th International Symposium on Automated Technology for Verification and Analysis*, pages 252–257. Springer–Verlag, October 2008.

[14] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in BIP. In *SEFM*, pages 3–12, 2006.

[15] Sverine Sentilles, John Håkansson, Paul Pettersson, and Ivica Crnkovic. Save-ide an integrated development environment for building predictable component-based embedded systems. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, September 2008.

[16] David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-trauring, and D Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16, 1991.

[17] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[18] V. Bos and J.J.T. Kleijn. Automatic verification of a manufacturing system. *Robotics and Computer Integrated Manufacturing*, 17:185–198, 2001.

[19] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and consistent equation semantics of hybrid chi. *Journal of Logic and Algebraic Programming*, 68(1-2):129 – 210, 2006.

[20] Xu Ke, P. Pettersson, K. Sierszecki, and C. Angelov. Verification of comdes-ii systems using uppaal with model transformation. *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*, pages 153–160, Aug. 2008.

Chapter 9

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

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

In proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pages 478-485, 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.

9.1 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 9.2.2. 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 9.2, we briefly recall the ProCom component model and identify some of its particularities. Section 9.3 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 9.4, whereas in Section 10.6, we conclude the paper.

9.2 The Component Model

9.2.1 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

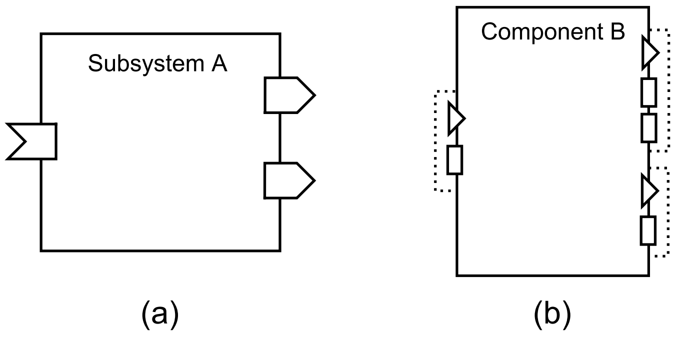

Figure 9.1: A ProSys subsystem and a simple ProSave component.

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.

9.2.2 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 9.2). 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.

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

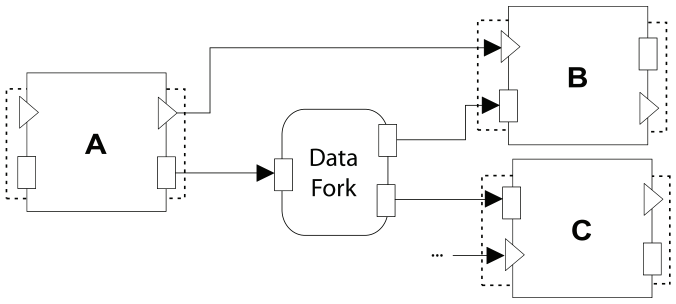

Figure 9.2: Example of a critical modeling of data and trigger transfer in ProCom.

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 9.3:

- 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.

9.3 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.

9.3.1 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 9.3. A transition can be either *urgent* or *non-urgent*, and it can have *priority* or no priority. As shown in Figure 9.3, 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

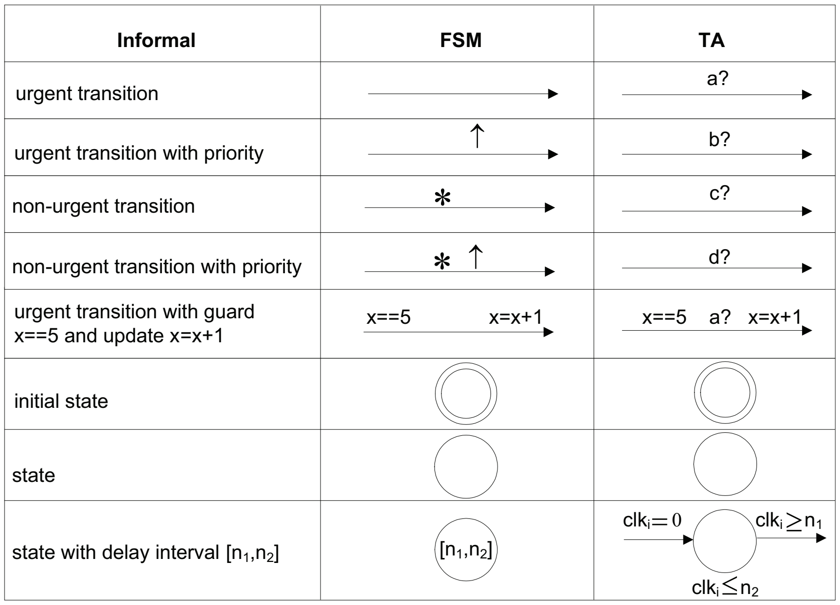
Informal
FSM
TA
urgent transition
a?
urgent transition with priority
↑
b?
non-urgent transition
*
c?
non-urgent transition with priority
* ↑
d?
urgent transition with guard x==5 and update x=x+1
x==5 x=x+1
x==5 a? x=x+1
initial state
state
state with delay interval [n1,n2]
[n1,n2]
clki= 0
clki≥n1
clki≤n2

Figure 9.3: The graphical notation of the FSM elements and their translation into TA.

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.

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.

9.3.2 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 9.3. 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 9.3. The translated TA edges need a timed automaton offering synchronization on the complementary channels (e.g., $a!$ complementary to $a?$), depicted in Figure 9.4.

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 9.4.

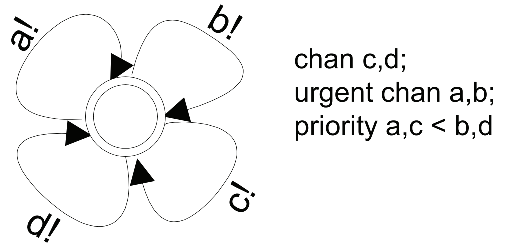

Figure 9.4: The automaton used for synchronization.

9.3.3 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.

9.3.4 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 9.5 (a)). The informal semantics of a service in ProSave is described in Section 9.2. The formal semantics of a service, in this case, S_1, is described below and shown in Figure 9.5 (b).

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

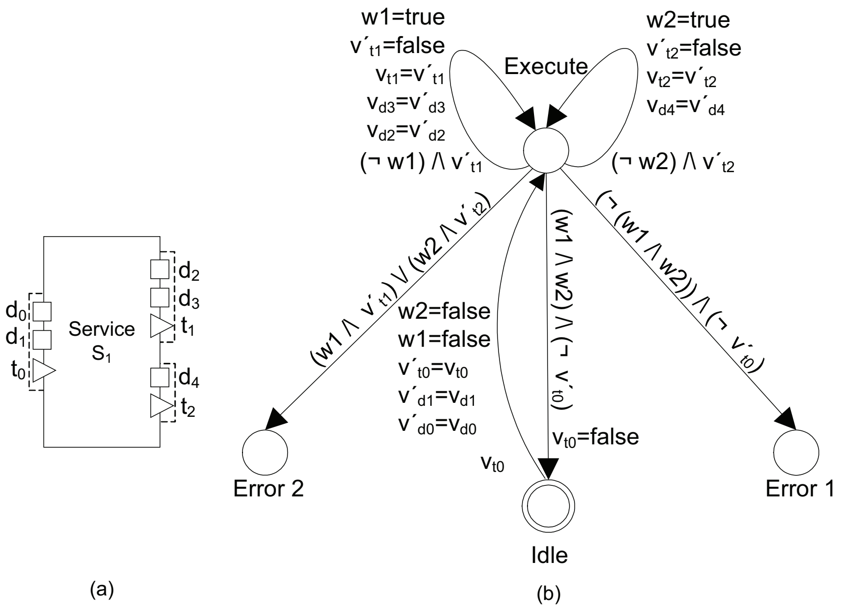

Figure 9.5: (a) A ProSave service S_1 and (b) its formal semantics.

As such, the formal semantics, ensures the informal semantics described in Section 9.2 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.

9.3.5 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 9.6, for data connection, and in Figure 9.7, for trigger connection.

To ensure that data is transferred prior to trigger, and to avoid undesirable consequences otherwise, the transitions in the FSM formalism (Figure 9.6) 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]).

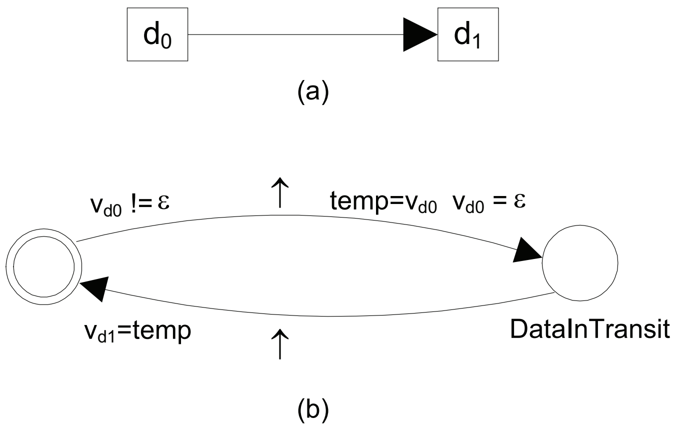

Figure 9.6: (a) A ProSave data connection and (b) its formal semantics.

9.3.6 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 9.3.4). This emphasizes the fact that, from an external

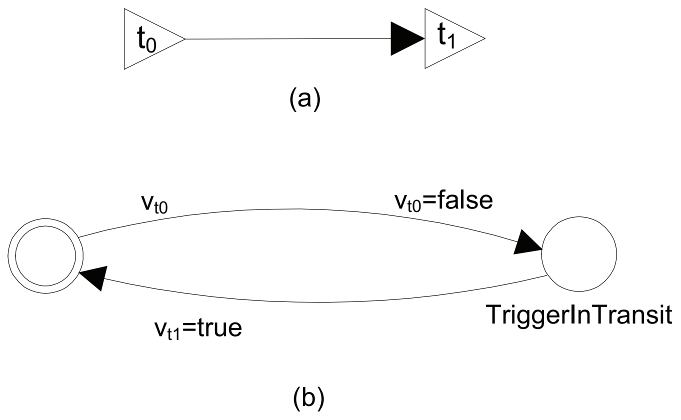

Figure 9.7: (a) A ProSave trigger connection and (b) its formal semantics.

observer's point of view, there is no difference between early design black box components and fully implemented components.

9.3.7 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 9.9 (a)) and *input*- and *output message ports* (see Figure 9.10 (a) and Figure 9.11 (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 9.8.

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 9.9 (b). Thus, the formal semantics complies to the informal semantics of a clock, described in Section 9.2.

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

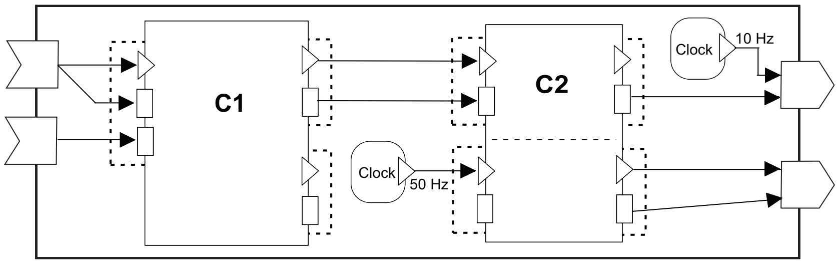

Figure 9.8: A ProSys subsystem internally modelled by ProSave.

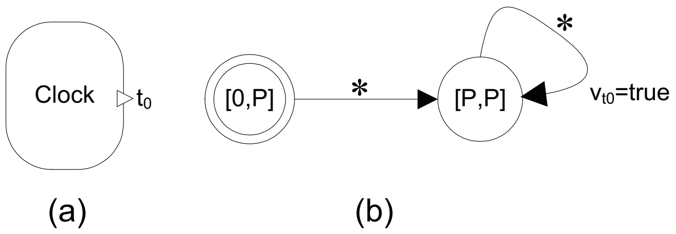

Figure 9.9: (a) A ProSave clock with period P and (b) its formal semantics.

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 9.10 (b) and Figure 9.11 (b), respectively.

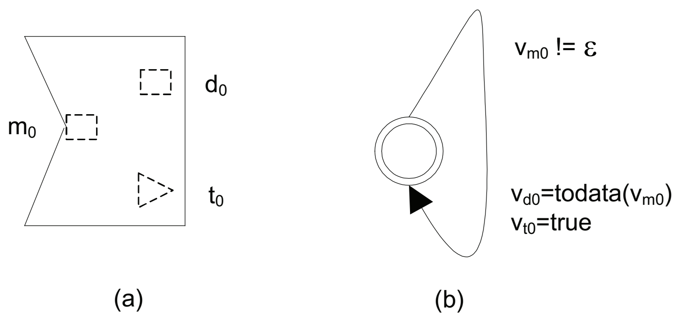

Figure 9.10: (a) A ProSave input message port and (b) its formal semantics.

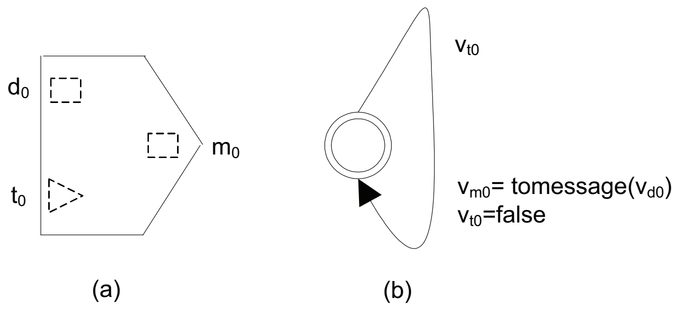

Figure 9.11: (a) A ProSave output message port and (b) its formal semantics.

9.4 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.

9.5 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 9.3.2. 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.

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. In *IEEE Computer*, pages 78–85. IEEE, March 2000.

[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] T. Bureš, J. Carlson, I. Crnković, S. Sentilles, and A. 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] X. Ke, K. Sierszecki, and C. 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 Computer Society, 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] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. 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.

Chapter 10

Paper C: Bridging the Semantic Gap between Abstract Models of Embedded Systems

Jagadish Suryadevara, Eun-Young Kang, Cristina Seceleanu, Paul Pettersson

In proceedings of the 13th International Symposium on Component Based Software Engineering (CBSE), pages 55-73, June, 2010.

Abstract

In the development of embedded software, modeling languages used within or across development phases e.g., requirements, specification, design, etc are based on different paradigms and an approach for relating these is needed. In this paper, we present a formal framework for relating specification and design models of embedded systems. We have chosen UML statemachines as specification models and ProCom component language for design models. While the specification is event-driven, the design is based on time triggering and data flow. To relate these abstractions, through the execution trajectories of corresponding models, formal semantics for both kinds of models and a set of inference rules are defined. The approach is applied on an autonomous truck case-study.

10.1 Introduction

Embedded systems (ES) are increasingly becoming control intensive, and time sensitive. To ensure predictable behaviors, the development phases of an ES require extensive modeling and analysis. These development phases/ abstraction layers e.g., requirements, specification, design, and implementation, provide opportunities for applying different predictability analysis techniques. Such models have to be precise enough to support formal analysis, and must ensure inter-operability during design. However, they may use paradigms for describing behavior that cannot be immediately compared and related, due to their apparently incompatible nature.

There exist several paradigms for behavior specification of embedded systems. For example, statemachine based approaches, such as UML statemachines [1], are intended to specify timed aspects of computation and communication, besides functionality. They often use an aperiodic, *event-triggered* representation of behavior, since such a paradigm facilitates easy changing of a model's configuration or set of events. On the other hand, behavior models might use a different modeling paradigm, e.g., a periodic, *time-triggered* behavioral description, instead of an event-triggered representation. With time-triggered communication, the data is read from a buffer, according to a triggering condition generated by, e.g., a periodic clock. Although these modeling capabilities are invaluable to obtaining a mature ES development process tailored for predictability, in order to ensure the correctness of the process, one needs to guarantee that the behavioral models are indeed consistent.

In this paper, we present a formal framework and a methodology for relating event-based and time triggered, data-flow driven models of behavior, which may be used at the same abstraction layer, e.g., at specification level, or across various layers of abstraction, from specification, to, e.g., the design level of embedded system development. Concretely, we consider UML statemachines [1] for event-based specification models and the ProCom component language [2] for design models. Hence, as it stands now, the framework is tailored to a specific class of embedded systems, which employ the above mentioned formalisms for modeling behavior. However, the framework and the methodology could be generalized to include other similar classes of systems (e.g., component based systems) and other behavioral paradigms (e.g., finite state machines).

The proposed framework is based on comparison of execution trajectories of corresponding behavior models. To accomplish this, the formal semantics of both kinds of models is defined in terms of underlying transition systems.

As the execution trajectories generated by above described models can be extremely large and incomprehensible, they need to be reduced to more readable and analyzable forms. Hence, we propose two sets of inference rules, one for simplification of specification trajectories and other for simplification of design trajectories. Moreover, in order to be able to relate and compare the above two sets of simplified trajectories, we introduce a set of transformation rules that lets one relate an event-triggered trajectory with corresponding time-triggered one.

We apply our approach on an autonomous truck system, by comparing some trajectories of its specification with those of corresponding component-based design model. By virtually simulating the models, we show a "run" of each model, respectively, by outlining corresponding sets of representative trajectories. Then, we show that, by applying our rules, we can first simplify the design model trajectory and then transform it into a trajectory equivalent to the one generated by the specification model. The timing aspects of both runs are also apparent in the respective trajectories, hence we show how to relate them too. For creating the truck's design model, we use the development environment of SaveIDE [3], an integrated design environment for ES. SaveIDE is developed as part of the PROGRESS project [4] for component-based development of predictable ES in the vehicular domain. It supports the subset of ProCom modeling language used for the case study design of the paper.

The rest of the paper is organized as follows. In Section 10.2, we describe event-based, and time triggered formalisms for modeling embedded systems. Corresponding to these formalisms we formally define semantics of a subset of both UML statemachines and ProCom design languages. In Section 10.3, we present the case study details. In Section 10.4, we describe our methodology, and introduce three sets of inference rules for simplification and comparison of trajectories of specification and design models. Some related work is discussed in Section 11.8. In section 10.6, we make conclusions and some aspects of the future work of the paper.

10.2 Abstract Models of Embedded Systems

In this section, we define the modeling formalisms for model-based specification and design of embedded systems used in this paper. As specification language, we will consider UML statemachine notation with timing annotations [1], and for design models, we will use the ProCom component modeling language [5].

10.2.1 Specification model of embedded systems

We specify embedded systems using the UML statemachine notation [1]. In order to model timing, we will use the notion of timeouts provided in UML. An example of a UML statemachine is shown in Fig. 10.1. We now give a formal definition of the model:

Statemachine Syntax A statemachine is a tuple $\langle L, l_0, A, E, M\rangle$ where

- L is a finite set of locations,
- $l_0 \in L$ is the initial location,
- $A = \{a_0, ..., a_n, tm\}$ is a set of events, where
  - a_i is an external event with zero or more parameters,
  - tm is a timeout representing the expiry of a timer, and
- $M : L \to \{\varepsilon\} \cup \mathbb{N}$ is a mapping from locations to the natural numbers (including zero), or ε denoting absence of timeout,
- $E \subseteq L \times A \times L$ is a set of edges. $\Box$

Fig. 10.1 shows a UML statemachine with the three locations Follow, Turn, and Find. The edges from Follow to Turn and from Find to Follow are labeled with the external events e_o_l() and line_found(), respectively. The edge from Turn to Find is labeled with event after(4), intuitively denoting a timeout that expires after four time units[1].

We now give the semantics of a UML statemachine specification model defined in terms of a finite state transition system.

Statemachine Semantics The semantics of a statemachine is defined as a transition system $\langle S, s_0, T\rangle$ where

- S is a finite set of states of form $\langle l, m\rangle$ with $l \in L$ and $m \in \{\varepsilon\} \cup \mathbb{N}$,
- $s_0 \in S$ is the initial state $\langle l_0, M(l_0)\rangle$,
- $T \subseteq S \times A \cup \{tick\} \times S$ where $tick$ is a periodic internal event, is a transition relation such that

[1] In the figures, we use timeout events of the form $after(n)$, where $n \in \mathbb{N}$, instead of annotating the source location (e.g., location Turn in Fig. 10.1) with timeout value n.

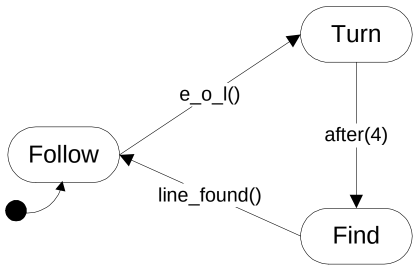

Figure 10.1: A UML statemachine specification model of the autonomous truck.

- $\langle l, m\rangle \xrightarrow{a_i} \langle l', m'\rangle$ if $\langle l, a_i, l'\rangle \in E$, and $m' = M(l')$
- $\langle l, m\rangle \xrightarrow{tick} \langle l', m'\rangle$ if $l = l'$, $m \neq 0$, and $m' = \begin{cases} \varepsilon & \text{if} \quad m = \varepsilon \\ m - 1 & otherwise \end{cases}$
- $\langle l, m\rangle \xrightarrow{tm} \langle l', m'\rangle$ if $\langle l, tm, l'\rangle \in E$, $m = 0$, and $m' = M(l')$

□

Intuitively, the initial state represents the initial location, and its timeout value, in the statemachine. The first rule describes the state change when an external event specified over an edge from current location, and in the current state, occurs. By second rule, if a timeout is defined at current location, the current value of the timeout decreases in steps of one corresponding to each occurrence of an internal periodic *tick* event. The *tick* event is ignored in the current state if no timeout is associated with the corresponding location. The third rule describes the occurrence of timeout event, and hence the location and corresponding state change, when the timeout duration associated with the current location expires i.e. becomes zero.

A *trajectory* of a UML specification model is an infinite sequence

$$\tau = \langle l_0, m_0\rangle \xrightarrow{\lambda_0} \langle l_1, m_1\rangle \xrightarrow{\lambda_1} \langle l_2, m_2\rangle \; ...$$

where $\langle l_0, m_0\rangle$ is the initial state, and $\langle l_i, m_i\rangle \xrightarrow{\lambda_i} \langle l_{i+1}, m_{i+1}\rangle \in T$ and $\lambda_i \in \{a_0, ..., a_n, tick, tm\}$ for all $i \in \mathbb{N}$.

10.2.2 Design model of embedded systems

As design modeling language we will use ProCom [5], a component model for embedded systems. It consists of the two sub-languages: ProSys, which is designed to model systems at high level (i.e., in terms of large-grained components called *subsystems*), and ProSave [2] which is designed to model detailed functionality of the subsystems. In this paper, we will focus on the ProSave model as it is better suited for our purposes. A ProSave model consists of atomic or composite components connected through ports (divided into input and output ports), and connections. Ports and connections represent data flow between components.

Component Syntax A component C is a tuple $\langle I, O, P, in, out, f, e\rangle$, where

- I, O, and P are mutually disjoint sets of input, output, and private variables respectively,
- $in : I \rightarrow Bool$ is a boolean expression over input variables I that triggers the execution of the component,
- $out : O \rightarrow Bool$ is a boolean expression over output variables O that indicates that the component has completed its execution,
- $f : I \times P \rightarrow P \times O$ is a function that maps input and private values to the private and output values, and
- $e \in \mathbb{N}$ is a constant representing the execution time of the component.

$\square$

We denote by $X = I \cup O \cup P$ the set of all variables with size $|X| = |I| + |O| + |P|$. We will further use $C.n$ to denote the elements of a component, hence e.g., $C.I$ denotes the input variables of component C. We now introduce the formal syntax of the ProSave model.

ProSave Syntax A ProSave design model is a tuple $\langle \mathbb{C}, \rightarrow\rangle$, where

- $\mathbb{C} = \{C_0, ..., C_n\}$ is a set of components,
- $\rightarrow \subseteq \mathbb{C} \times \mathbb{C}$ is a set of component connections, such that output variables $C_i.O$ may be connected to input variables $C_j.I$ $\square$

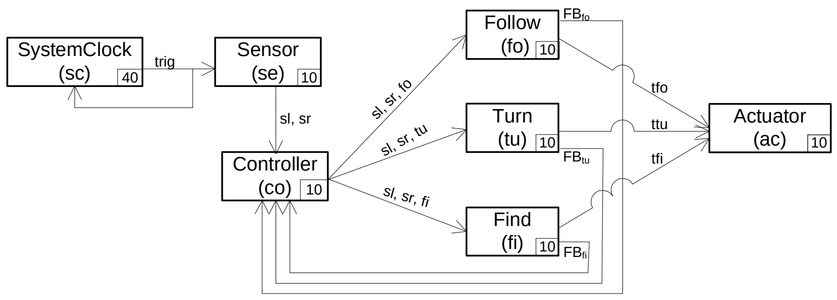

Figure 10.2: Schematic view of a ProSave design model of the autonomous truck.

We will write $C_i.O_m \rightarrow C_j.I_n$ to represent the connection from output variable m of component C_i to input variable n of component C_j.

A ProSave system is typically driven by a periodic clock which periodically generates a control (or trigger) signal. A clock component is defined as follows:

Clock Component A component $C = \langle I, O, P, in, out, f, e\rangle$ is a clock component with period p iff $| I |=| O |= 1$, $e = p$, and $C.O \rightarrow C.I$. $\Box$

Fig. 10.2 shows a ProSave design model consisting of seven components (depicted as boxes) interconnected by data and control flow connections (depicted as solid arrows indicating the flow direction). Component SystemClock is a clock component with period 40. The other six components have execution time 10. Their internal behavior may be specified using a formalism based on statecharts [6] or timed automata [7], which we are not explicitly concerned with in this paper. A component starts execution when it receives control input. It then reads its input and proceeds with internal computation. When the internal execution is completed, data and control output is generated for other components.

We will now give the formal semantics of the subset of ProSave used in this paper. For the semantics of the full ProCom language, we refer the reader to [2].

For a ProSave model consisting of components $C_0, ..., C_n$, we use V to denote the set of all variables in a model, i.e., $V = X_0 \cup ... \cup X_n$. The semantics is defined using valuations α mapping each variable in V to values in the type (or domain)[2] of V, and vectors $\bar{\beta}$ of $\beta_i \in \{0, ..., e_i, \perp\}$ representing

[2]We assume all variables in V are of type Boolean or finite domained integers.

the remaining execution time of all components C_i.

We use $f_i(\alpha)$ to denote the valuation α' in which $\alpha'(x_i)$ for each $x_i \in P_i \cup O_i$ is the value obtained by applying the function $C_i.f$ in the valuation α, and $\alpha'(x') = \alpha(x')$ for all other variables x'. To update the execution time vector $\bar{\beta}$ we use $\bar{\beta}[\beta_i := n]$ to denote the $\bar{\beta}'$ in which $\beta'_i = n$ and $\beta'_j = \beta_j$ for all $j \neq i$, and we write $\bar{\beta} \ominus n$ to denote the $\bar{\beta}'$ in which $\beta'_i := \beta_i - n$ for all $\beta_i \geq n$.

ProSave Semantics The semantics of a ProSave design model $\langle \{C_0, ..., C_n\}, \rightarrow \rangle$ is defined as a transition system $\langle \Sigma, \sigma_0, \mathcal{T} \rangle$ where

- Σ is a set of states of the form of a pair $\langle \alpha, \bar{\beta} \rangle$,
- $\sigma_0 \in \Sigma$ is the initial state $\langle \alpha_0, \bar{\beta}_0 \rangle$ which is such that $\alpha_0 \models C_i.in$ for all clock components C_i and $\alpha_0 \models \neg C_j.in$ for all other component C_j, and $\bar{\beta}_0 = \bar{\bot}$,
- $\mathcal{T} \subseteq \Sigma \times \{CD_i, CS_i, TP\} \times \Sigma$ is a set of transitions such that the following conditions hold:
  - (component start) $\langle \alpha, \bar{\beta} \rangle \xrightarrow{CS_i} \langle \alpha', \bar{\beta}' \rangle$ if $(C_i.in \wedge (\beta_i = \bot))$, $\bar{\beta}' = \bar{\beta}[\beta_i := e_i]$, and for all $i \neq j : \beta_j \neq 0$,
  - (component done) $\langle \alpha, \bar{\beta} \rangle \xrightarrow{CD_i} \langle \alpha', \bar{\beta}' \rangle$ if $\beta_i = 0$, $\alpha' = f_i(\alpha)$, and $\bar{\beta}' = \bar{\beta}[\beta_i := \bot]$,
  - (time passing) $\langle \alpha, \bar{\beta} \rangle \xrightarrow{TP} \langle \alpha', \bar{\beta}' \rangle$ if for all $i : \neg(C_i.in \wedge (\beta_i = \bot))$ and $\beta_i \neq 0$, $\bar{\beta}' = \bar{\beta} \ominus 1$, and $(\alpha' = \alpha)$.

where $CS_i \in \{CS_0, ..., CS_n\}$ and $CD_i \in \{CD_0, ..., CD_n\}$. $\square$

Intuitively, in the initial state only the clock components are triggered and the remaining execution time of all components are undefined. The "component start" rule describes how components are started. A component C_i may start its execution provided that all completed components have written their output. When C_i starts, its execution time is set to e_i. The "component done" rule describes that when a component C_i completes its execution, its output values are generated and mapped to the input values of the connected components according to connection relation $\rightarrow$, and its remaining execution time is updated to $\bot$ to reflect that it is inactive. Rule "time passing" describes how time progresses in the design model. As time progresses the remaining execution time β_i of each active component C_i is decremented by 1.

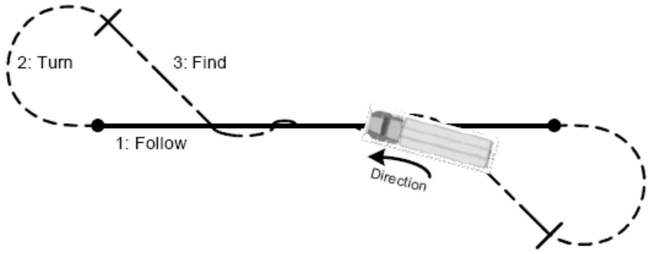

Figure 10.3: Path of the truck movement.

A *trajectory* of a design model is an infinite sequence

$$\pi = \langle \alpha_0, \beta_0 \rangle \xrightarrow{\gamma_0} \langle \alpha_1, \beta_1 \rangle \xrightarrow{\gamma_1} \langle \alpha_2, \beta_2 \rangle \ ...$$

where $\langle \alpha_0, \beta_0 \rangle \in \sigma_0$ is an initial state, and $\langle \alpha_i, \beta_i \rangle \xrightarrow{a_i} \langle \alpha_{i+1}, \beta_{i+1} \rangle \in \mathcal{T}$ is a transition such that $\gamma_i \in \{CD_i, CS_i, TP\}$ for all $i \in \mathbb{N}$.

10.3 Case Study: Autonomous Truck

The autonomous truck is part of a demonstrator project conducted at the Progress research centre[3]. The truck moves along a specified path (as illustrated in Fig. 10.3), according to a specified application behavior. In this section we give an overview of the truck application followed by a specification, and a design model, described in the modeling languages introduced in the previous section.

We will study a simplified version of the case study, in which the truck should simply follow a line. When it reaches the end of the line, it should try to find back to the line, follow the line again in the opposite direction, and repeat its behavior. The truck will have the following operational modes (see also Fig. 10.1):

- *Follow*: in which the truck follows the line (the thick line of Fig. 10.3) using light sensors. When the end of the line is detected, it changes to *Turn* mode.
- *Turn*: the truck turns right for a specified time duration, and then changes to *Find* mode.

[3]For more information about Progress, see http://www.mrtc.mdh.se/progress/.

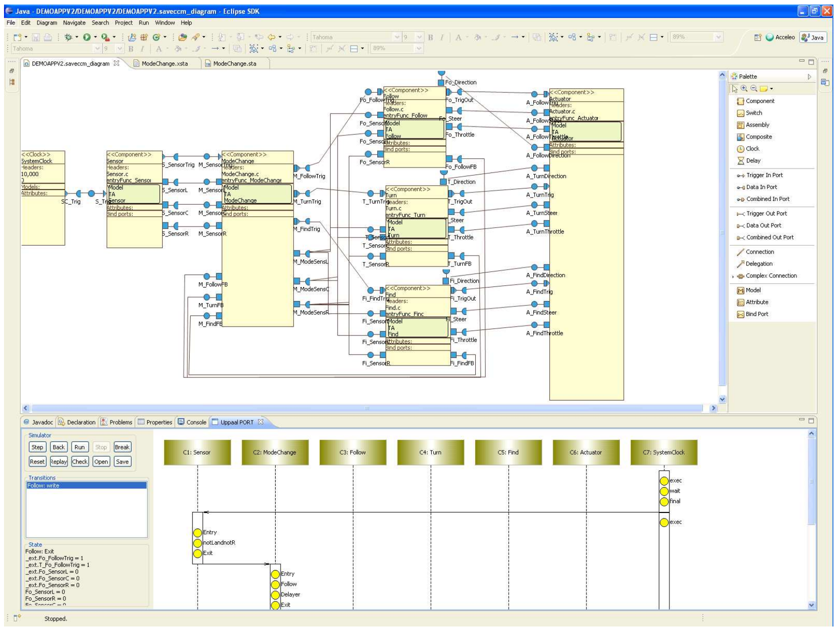

Figure 10.4: The design model of the autonomous truck in SaveIDE.

- *Find*: the truck searches for the line. When it is found, the truck returns to *Follow* mode.

A specification model of the case study is given in Fig. 10.1. It starts in location Follow. The end of the line is modeled using external event e_o_l(). In location Turn, it turns for four seconds, and then proceeds to location Find when the timer expires. The external event line_found() models that the line is found and control switches back to the initial location Follow.

The schematic view of a ProSave deign model of the case study is given in Fig. 10.2. The original model (as shown in Fig. 10.4) was developed using SaveIDE [3], an integrated development environment supporting the subset of ProSave used in this paper. As shown in Fig. 10.2, the design model consists of components SystemClock (a periodic clock), Sensor, Controller, Follow, Turn, Find and Actuator. Component SystemClock triggers the complete model periodically through the component Sensor which reads the light sensors of the truck. The sensor values (left, right) are communicated through the data ports sl and sr. Note, a connection between two components as shown in Fig. 10.2, denotes a collection of independent port connections between corresponding data or trigger ports of the components. Component Controller acts as a control switch for triggering the components Follow, Turn, and Find selectively , through control ports fo, tu, fi respectively, which contain the functionality of the corresponding modes of the truck behavior. The completion of execution of each operational mode (the corresponding component) is indicated by data (port) values FB_{fo}, FB_{tu}, and FB_{fi} respectively. Component Actuator, triggered by control port tfo, ttu, or tfi, actuates the corresponding hardware to cause the physical activity of the truck movement. As discussed previously, the periodicity of the SystemClock is 40 time units and the execution times of each of other components is 10 time units.

10.4 Methodology Description

In Section 10.2, we have described the syntax and semantics of two models used in the development of embedded system software: the event-based model of UML statemachines, and the time-triggered and data-flow oriented model of ProCom. These are examples of modeling languages that are aimed at providing different views of embedded systems, used in different stages or at different abstraction levels during system development. The common use of different models creates a need for comparing descriptions of systems made in different modeling languages.

In this section, we propose a method for comparing event-based and time-triggered models of embedded systems. The method will be described and illustrated on UML statemachines and ProCom models of the autonomous truck case study described in the previous section. Constructing a semantic bridge between the two models requires a series of steps that need to be systematically applied. Our methodology for bridging the gap between the paradigms consists of the following five steps: (i) given a specification trajectory, generate a corresponding design trajectory by e.g, simulating the model; (ii) simplify the specification trajectory (can be omitted); (iii) simplify the design trajectory; (iv) transform the design trajectory into one comparable to the event-based specification trajectory; (v) compare the reduced specification and design trajectories.

To support above described steps (ii) to (iv) of the method we will present in Sections 4.1 to 4.3 a number of inference rules for simplifying specification and design trajectories, and for transforming between the two. In the latter transformation step, we need to take two crucial steps. One is to relate events in the UML statemachine model to the data-flow of the ProCom model. This is done by mapping events observed in the specification trajectories to predicates over the data variables used in the design model. We expect that a designer will easily be able to provide this mapping based on his insights and knowledge in the models. For the autonomous truck system, we can assume a mapping given in Table 10.4.2 in section 10.4.2. A second important step in relating two models of embedded systems regards the different time scales that may be used. We take a rather straightforward approach and assume a δ, as defined in section 10.4.3, for characterizing the sampling period in design models, in comparison to the time base used in the specification model.

10.4.1 Specification simplification inference rules

In the following rules, we denote by $s_i \in S, i \in \mathbb{N}$, the states of an arbitrary specification model trajectory.

Skip time rule. This rule states that a sequence of *tick* transitions corresponding to a location without an associated timeout can be ignored.

$$\frac{s_i \xrightarrow{tick} s_i \xrightarrow{tick} \quad \dots \quad \xrightarrow{tick} s_i}{s_i} \qquad \text{(skip)}$$

By applying this rule to the original specification trajectory of the Autonomous truck (omitted due to space limitations), we get the simplified trajectory shown in Fig. 5.(a).

Time passing rule. The intuition behind this rule is that one can collapse a sequence of *tick* transitions corresponding to a timeout location in the specification model, into a single transition that collects all the ticks. Consequently, the intermediate states generated by the individual ticks become hidden.

$$\frac{s_i \xrightarrow{tick} s_{i+1} \xrightarrow{tick} \dots \xrightarrow{tick} s_{i+n}}{s_i \xrightarrow{n.tick} s_{i+n}} \qquad \text{(n\_tick)}$$

To show the rule at work, we have used it to reduce the sequence of tick transitions (s_1 to s_5) displayed in Fig. 5.(a), to the corresponding sequence in Fig. 5.(b).

Timeout start rule. Here, we introduce the virtual event tm_start needed to distinguish the transition leading to the corresponding timeout annotated location, from the one fired when the timeout countdown starts. Although not a simplification rule by itself, its usefulness is shown in the rules skip and n_TP, presented later.

$$\frac{s_i \xrightarrow{event\_label} s_{i+1} \quad m = value \quad m' \neq \varepsilon \wedge m' \neq 0}{s_i \xrightarrow{event\_label} s_{i+1} \xrightarrow{tm\_start} s_{i+2}} \qquad \text{(tm\_start)}$$

In the above rule, $value \in \{0, \varepsilon\}$. In case $value = 0$, that is, $m = 0$, it follows that $event\_label = tm$; on the other hand, if $value = \varepsilon$, that is, $m = \epsilon$, then $event\_label = a$.

Timeout rule. A sequence of *n-tick* transitions beginning at a location having timeout n that is then followed by a timeout transition can be reduced to a single transition denoted by $tm(n)$, as shown below:

$$\frac{s_i \xrightarrow{n.tick} s_{i+1} \xrightarrow{tm} s_{i+2}}{s_i \xrightarrow{tm(n)} s_{i+2}} \qquad \text{(tm)}$$

After applying the timeout rule, the sequence of the *4-tick* transitions (s_1 to s_5) followed by the tm transition (s_5 to s_6), depicted in Fig. 5.(b), is reduced to transition (s_1 to s_6), as in Fig. 5.(c).

10.4.2 Design simplification inference rules

As already mentioned, in order to be able to relate the specification and design models formally, we require the detailed mapping of the external and timeout events of the specification model onto predicates over data values of the corresponding design model. In addition to the observable events, such mapping should also include the virtual timeout start event, tm_start. We assume that such a mapping is provided by the ProSave designer, as he/she "implements" the specification model. For the current design model of the autonomous truck, one such mapping is given in Table 10.4.2.

Below, we denote by $\sigma_i \in \Sigma, i \in \mathbb{N}$, the states of an arbitrary design model trajectory. In Fig. 10.6, we give an excerpt of a design trajectory of the autonomous truck, and, on the right-hand side of the figure, we explain the used notation in terms of Definition 10.2.2 of Section 10.2.

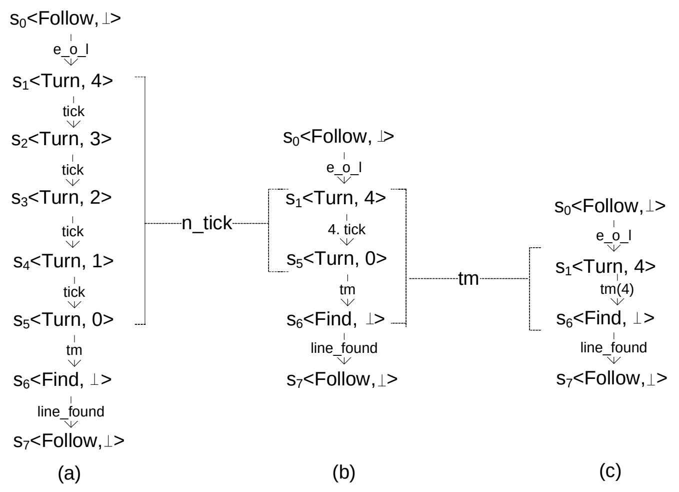

Figure 10.5: Examples of specification trajectories simplifications of the autonomous truck.

Skip time rule. This rule states that a sequence of TP-transitions from states that do not satisfy the predicate corresponding to the virtual event **tm_start** can be ignored. Such transitions correspond to time passing in the design trajectory, which are of no interest, that is, not related to observable timeout events.

$$\frac{\sigma_i \xrightarrow{TP} \sigma_{i+1} \quad \sigma_i \not\models Pred_{tm\_start}}{\sigma_i} \tag{Skip}$$

We apply the skip time rule on a design trajectory of the autonomous truck (see Fig. 10.7), and, as a result, we simplify the trajectory by reducing states $\sigma_1, \sigma_2, \sigma_3$, and σ_4, to state σ_1 only. The complete trajectory is given in the Appendix.

Hide CS rule. By this rule, a CS-transition, hence the target state, can always be ignored.

$$\frac{\sigma_i \xrightarrow{CS_i} \sigma_{i+1}}{\sigma_i} \tag{hide\_CS}$$

Assuming a design trajectory of our case-study, the application of the above rule on this trajectory is shown in Fig. 10.8.

Events	Predicates
e_0_l	$sl \wedge sr \wedge FB_{fo}$
$line\_found$	$(sl \vee sr) \wedge FB_{fi}$
tm(timeout event)	FB_{tu}
tm_start	ttu

Table 10.1: Events and corresponding predicates of the autonomous truck models.

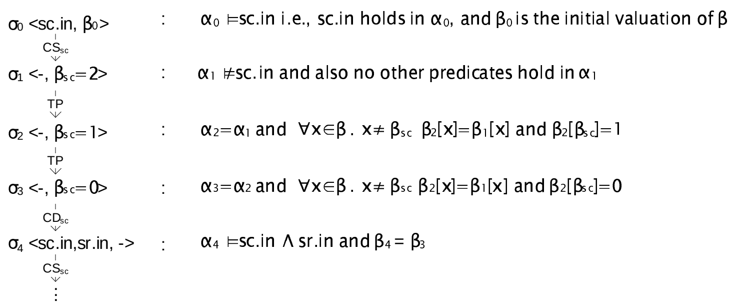
σ_0 <sc.in, β_0>
: $\alpha_0 \vDash$sc.in i.e., sc.in holds in α_0, and β_0 is the initial valuation of β
CS_{sc}
σ_1 <-, β_{sc}=2>
: $\alpha_1 \nvDash$sc.in and also no other predicates hold in α_1
TP
σ_2 <-, β_{sc}=1>
: $\alpha_2=\alpha_1$ and $\forall x \in \beta .\ x \neq \beta_{sc}$ $\beta_2[x]=\beta_1[x]$ and $\beta_2[\beta_{sc}]=1$
TP
σ_3 <-, β_{sc}=0>
: $\alpha_3=\alpha_2$ and $\forall x \in \beta .\ x \neq \beta_{sc}$ $\beta_2[x]=\beta_1[x]$ and $\beta_2[\beta_{sc}]=0$
CD_{sc}
σ_4 <sc.in,sr.in, ->
: $\alpha_4 \vDash$sc.in $\wedge$ sr.in and $\beta_4 = \beta_3$
CS_{sc}
⋮

Figure 10.6: Interpretation of example design trajectory notation w.r.t. Definition 10.2.2.

Hide CD rule. This rule stipulates that a CD-transition and the corresponding source state can be ignored if the target state does not satisfy any event occurrence predicate.

$$\frac{\sigma_i \xrightarrow{CD_i} \sigma_{i+1} \quad \forall a \in A \cdot \sigma_i \nvDash Pred_a}{\sigma_{i+1}} \qquad \text{(hide\_CD)}$$

An example application of the above rule is given in Fig. 10.8.

Time passing rule. A sequence of TP transitions starting in a state satisfying the predicate corresponding to *tm_start*, and ending in a state where the corresponding timeout occurs, can be collected into a single transition, while the

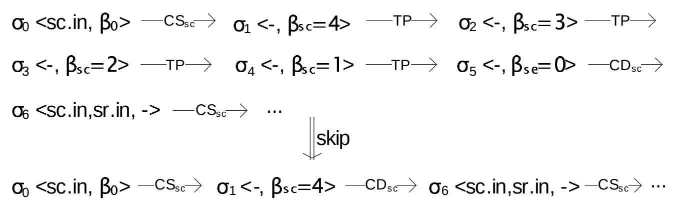
σ0 <sc.in, β0> —CSsc→ σ1 <-, βsc=4> —TP→ σ2 <-, βsc=3> —TP→
σ3 <-, βsc=2> —TP→ σ4 <-, βsc=1> —TP→ σ5 <-, βse=0> —CDsc→
σ6 <sc.in,sr.in, -> —CSsc→ ···
skip
σ0 <sc.in, β0> —CSsc→ σ1 <-, βsc=4> —CDsc→ σ6 <sc.in,sr.in, -> —CSsc→ ···

Figure 10.7: Application of *skip rule* on a design trajectory of the autonomous truck model.

intermediate states are ignored.

$$\frac{\sigma_i \xrightarrow{TP} \sigma_{i+1} \ldots \xrightarrow{TP} \sigma_{i+n} \xrightarrow{CD_j} \sigma_{i+n+1} \quad \sigma_i \models Pred_{tm\_start} \quad \sigma_{i+n+1} \models Pred_{tm}}{\sigma_i \xrightarrow{n.TP} \sigma_{i+n+1}} \qquad \text{(n\_TP)}$$

We have applied the above rule on a design trajectory of our Autonomous Truck, in Fig. 10.8. The rule works on the states $\sigma_{42}, \sigma_{46}, \sigma_{50}, \sigma_{53}, \sigma_{55}$ and σ_{56}.

Precedence of inference rules. In order to get the correct simplified design trajectory, we assume the following precedence rule when applying the above inference rules over design trajectories (rule hide_CS binds the strongest):

hide_CS **precedes** hide_CD **precedes** n_TP **precedes** Skip

10.4.3 Rules for transforming the design model trajectories

The following rules let one obtain design trajectories that are comparable to the event-based specification model trajectories. The first rule focuses on relating the time scales in both models; in order to achieve the goal, we assume a fixed quanta of time (number of time units), called δ, which can be viewed as the minimum amount of time guaranteed to be free of events. Then, this smallest amount of time becomes the basic time-unit that all time-related elements in both trajectories can be expressed by.

TimeOut Rule. We assume that a *TP*-transition "consumes" δ time units, the time duration associated with a *tick* event is *(m* δ)*($m \in \mathbb{N}$), and an '*n*' time units timeout in a specification trajectory, *tm(n),* equals (*n* tick*). Then,

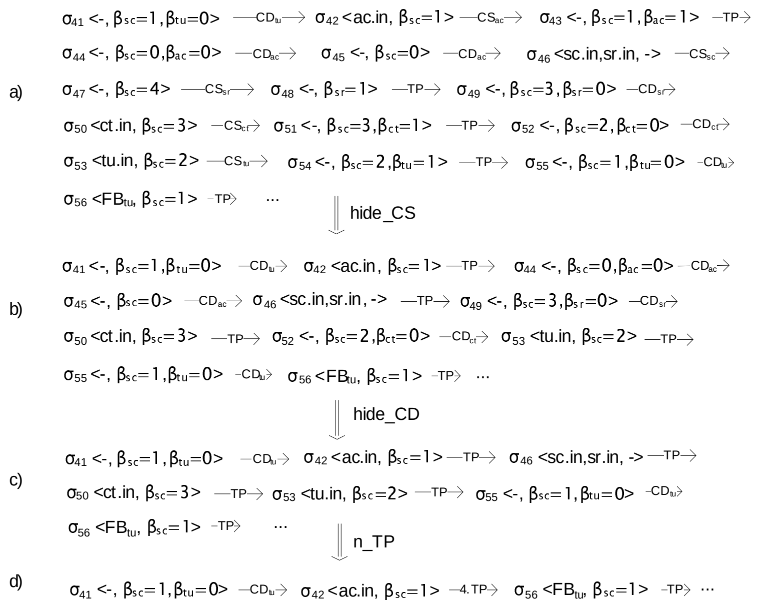
a)
σ41 <-, βsc=1,βtu=0> —CDtu→ σ42 <ac.in, βsc=1> —CSac→ σ43 <-, βsc=1,βac=1> —TP→
σ44 <-, βsc=0,βac=0> —CDac→ σ45 <-, βsc=0> —CDac→ σ46 <sc.in,sr.in, -> —CSsc→
σ47 <-, βsc=4> —CSsr→ σ48 <-, βsr=1> —TP→ σ49 <-, βsc=3,βsr=0> —CDsr→
σ50 <ct.in, βsc=3> —CSct→ σ51 <-, βsc=3,βct=1> —TP→ σ52 <-, βsc=2,βct=0> —CDct→
σ53 <tu.in, βsc=2> —CStu→ σ54 <-, βsc=2,βtu=1> —TP→ σ55 <-, βsc=1,βtu=0> —CDtu→
σ56 <FBtu, βsc=1> —TP→ ...
hide_CS
b)
σ41 <-, βsc=1,βtu=0> —CDtu→ σ42 <ac.in, βsc=1> —TP→ σ44 <-, βsc=0,βac=0> —CDac→
σ45 <-, βsc=0> —CDac→ σ46 <sc.in,sr.in, -> —TP→ σ49 <-, βsc=3,βsr=0> —CDsr→
σ50 <ct.in, βsc=3> —TP→ σ52 <-, βsc=2,βct=0> —CDct→ σ53 <tu.in, βsc=2> —TP→
σ55 <-, βsc=1,βtu=0> —CDtu→ σ56 <FBtu, βsc=1> —TP→ ...
hide_CD
c)
σ41 <-, βsc=1,βtu=0> —CDtu→ σ42 <ac.in, βsc=1> —TP→ σ46 <sc.in,sr.in, -> —TP→
σ50 <ct.in, βsc=3> —TP→ σ53 <tu.in, βsc=2> —TP→ σ55 <-, βsc=1,βtu=0> —CDtu→
σ56 <FBtu, βsc=1> —TP→ ...
n_TP
d)
σ41 <-, βsc=1,βtu=0> —CDtu→ σ42 <ac.in, βsc=1> —4.TP→ σ56 <FBtu, βsc=1> —TP→ ...

Figure 10.8: (a) a partial design trajectory of the autonomous truck, and (b) to (d) corresponding reduced trajectories after application of the inference rules of Section 10.4.2.

it follows that an *n.m.TP* transition in the design trajectory is equivalent to the '*n*' timeout event, *tm(n)*:

$$\frac{\exists \quad \sigma_k, \sigma_{k+1} \quad . \quad \sigma_i \xrightarrow{n.m.TP} \sigma_{i+1}}{\exists \quad s_k, s_{k+1} \quad . \quad s_k \xrightarrow{tm(n)} s_{k+1}} \tag{TO}$$

EventOccur Rule. An event occurrence in a specification trajectory corresponds to a CD-transition in the design trajectory, such that the predicate associated to the event holds in the target state of the design trajectory.

$$\frac{\exists \quad \sigma_i, \sigma_{i+1} \quad . \quad \sigma_i \xrightarrow{CD_j} \sigma_{i+1} \quad \sigma_{i+1} \models Pred_a}{\exists \quad s_k, s_{k+1} \quad . \quad s_k \xrightarrow{a} s_{k+1}} \tag{EO}$$

Next, we apply the above rules on a (simplified) design trajectory of our example, in order to obtain a trajectory comparable to the corresponding specification trajectory.

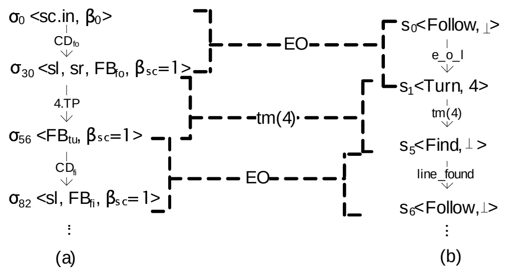

Figure 10.9: Comparison of completely reduced trajectories of (a) the design model, and (b) the specification model, of the autonomous truck.

10.4.4 Applying the methodology

Here, we show our methods at work, on the Autonomous Truck case study, presented in Section 10.3. We do this by transforming a trajectory of the design model (Fig. 10.2), which we present in the Appendix, into one that is comparable to the corresponding specification model (Fig. 10.1) trajectory. First, the design trajectory is simplified by applying the inference rules introduced in section 10.4.1. Similarly, a trajectory of the specification model is then simplified to a minimal form by applying inference rules in 10.4.2. Both simplified trajectories are shown in Fig. 10.9. Next, we relate these trajectories by using the inference rules of transformation (section 10.4.3), as follows:

- by EO rule, the CD_{fo}-transition to state σ_{30} corresponds to the occurrence of event e_o_l, since the (e_o_l) predicate, that is, $FB_{fo} \wedge sl \wedge sr$, holds in the target state σ_{30}. Further, σ_{30} corresponds to the completion of the **Follow** mode of the truck behavior, as FB_{fo} holds (by design).
- similarly, by EO rule, the CD_{fi}-transition to state σ_{82} corresponds to the occurrence of event $line\_found$, since $Pred_{line\_found}$, that is, $FB_{fi} \wedge sl$, holds in the target state σ_{82}. Further, σ_{82} corresponds to the completion of the **Find** mode of the truck behavior, as FB_{fi} holds (by design).
- by TO rule, the timeout event, $tm(4)$, between specification states s_2 and s_6 corresponds to the $4.TP$-transition between design states σ_{30} and σ_{56} that satisfy $Pred_{tm\_start}$, and $Pred_{tm}$, respectively. Further, σ_{56}

corresponds to the completion of the **Turn** mode, as FB_{tu} holds (by design).

By applying the rules on the truck example, we have shown that, at least with respect to this example, it is possible to transform and compare a simplified design model trajectory of the Autonomous Truck with a simplified specification model trajectory. The transformation correlates also the time scales in both models. In this particular case, we have reduced the design model trajectory to an event-based trajectory identical to the specification one.

The above steps are necessary in proving the correctness of design with respect to specification, however they are not sufficient. To get closure, one has to first consider all possible design behaviors for transformation, and then possibly apply refinement techniques to prove that the design does implement the functional and timing requirements represented by the specification model (see Section 10.6).

10.5 Related Work

The problem of relating design to specification models is a topic with a growing interest in the research community. For synthesizing executable programs from timed models, a timed automata [7] based semantic framework, relying on non-instant observability of events is proposed [8]. Time-triggered automata (TTA) - a sub class of timed automata (TA) - are used to model finite state implementations of a controller that services the request patterns specified by a TA. This technique enables deciding whether a TTA correctly implements a TA specification. In comparison, although ProCom oriented, our methodology can be applied within a generic component-based framework, and is not being tied to any particular formal verification framework either.

Sifakis et al. propose a methodology for relating the abstractions of both real-time application software and corresponding implementation [9]. The related formal modeling framework integrates event-driven, and time triggered paradigms by defining *untiming* functions. Problems of correctness, timing analysis, and synthesis are considered in the methodology. In contrast to our approach, this one does not address the intermediate design layer commonly used in system development.

In recent years, component and architecture based developments have been recognized as a viable way of building software systems [10]. Plasil and Visnovsky describe a formal framework based on *behavior protocols,* in order to formally specify the interplay between components [11]. This allows for formal reasoning about the correctness of the specification refinement and about the correctness of an implementation, in terms of the specification. Further, the framework is validated in the SOFA component model environment [12]. While the approach provides much needed formal correctness in component-based development, it does not address timing issues and vertical layers of abstractions in real-time system development.

UML has emerged as an industrial standard notation in system development and provides various sub-languages namely statemachines, sequence diagrams, etc [1]. For specification and design of real-time systems, a sub-language called UML/MARTE has been proposed [13]. In [14], the expressiveness of MARTE for event-triggered, and time-triggered communication is described. MARTE-based approaches facilitate various analytical methods for analysis, e.g., schedulability, system performance analysis; however, it falls short in providing formal support for comparing models at different abstraction levels.

Egyed, A. et al. [15] develop a methodology to mainly bridge the information gap created by heterogeneous models across the software life-cycle by transforming architecture description into (high-level) UML designs. The latter are then further refined into lower level designs. In contrast to our approach, their work does not provide details on the behavioral transformations. Indeed, a formal approach for establishing the semantic links between different terminologies and concepts across an architectural and a number of design models is not sufficiently addressed during the transformation.

10.6 Conclusions and Future work

In this paper, we have presented a formal approach for relating system models used in different design stages of embedded systems. For the early specification phases, we chose the UML statemachine language in which system behaviors are described in terms of abstract states, event triggered state changes, and timeouts relating to the external system and timing behavior. For the later design stages, we use the ProCom component design model in which systems are specified using data-flow connectors and time-triggered component behaviors closer to the timing granularity and behavior exhibited on the target platform.

As a main result, we have described a formal way of comparing behavioral models of a system modeled in the two different languages. The solution is based on a set of inference rules that can be applied to gradually transform trajectories of a ProCom design model into a trajectory of a UML specification model. This enables a designer to make sure that a component-based and time-triggered ProCom design model implements the behavior of a more abstract and event-triggered UML specification of the same system.

Our initial experiences from applying the proposed technique to a truck control system, indicates that the design model trajectories can often be manually transformed into trajectories of the specification model. However, as this is not the case in general, we plan as future work to apply simulation relation checking to the specification trajectories, to prove (or disprove) conformance between non-identical trajectories. We will apply proof assistant tools to support these techniques.

Appendix

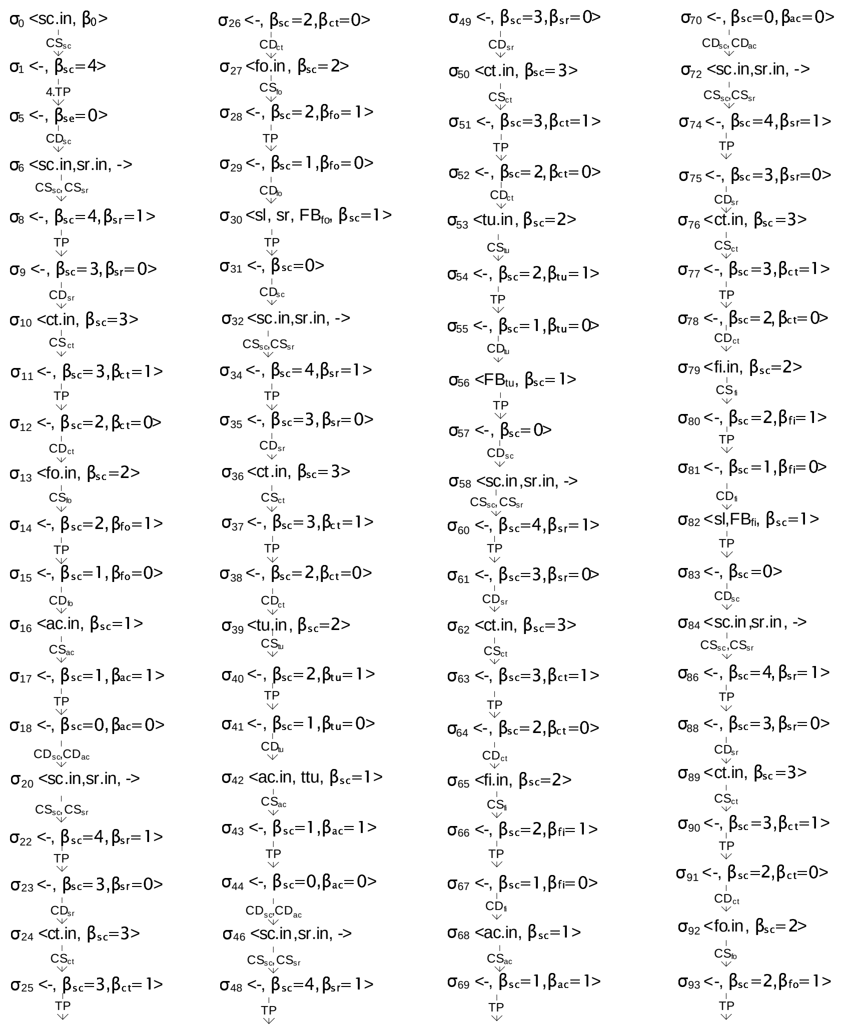
σ_0 <sc.in, β_0>
CS_{sc}
σ_1 <-, $\beta_{sc}=4$>
4.TP
σ_5 <-, $\beta_{se}=0$>
CD_{sc}
σ_6 <sc.in,sr.in, ->
CS_{sc}, CS_{sr}
σ_8 <-, $\beta_{sc}=4, \beta_{sr}=1$>
TP
σ_9 <-, $\beta_{sc}=3, \beta_{sr}=0$>
CD_{sr}
σ_{10} <ct.in, $\beta_{sc}=3$>
CS_{ct}
σ_{11} <-, $\beta_{sc}=3, \beta_{ct}=1$>
TP
σ_{12} <-, $\beta_{sc}=2, \beta_{ct}=0$>
CD_{ct}
σ_{13} <fo.in, $\beta_{sc}=2$>
CS_{fo}
σ_{14} <-, $\beta_{sc}=2, \beta_{fo}=1$>
TP
σ_{15} <-, $\beta_{sc}=1, \beta_{fo}=0$>
CD_{fo}
σ_{16} <ac.in, $\beta_{sc}=1$>
CS_{ac}
σ_{17} <-, $\beta_{sc}=1, \beta_{ac}=1$>
TP
σ_{18} <-, $\beta_{sc}=0, \beta_{ac}=0$>
CD_{sc}, CD_{ac}
σ_{20} <sc.in,sr.in, ->
CS_{sc}, CS_{sr}
σ_{22} <-, $\beta_{sc}=4, \beta_{sr}=1$>
TP
σ_{23} <-, $\beta_{sc}=3, \beta_{sr}=0$>
CD_{sr}
σ_{24} <ct.in, $\beta_{sc}=3$>
CS_{ct}
σ_{25} <-, $\beta_{sc}=3, \beta_{ct}=1$>
TP
σ_{26} <-, $\beta_{sc}=2, \beta_{ct}=0$>
CD_{ct}
σ_{27} <fo.in, $\beta_{sc}=2$>
CS_{fo}
σ_{28} <-, $\beta_{sc}=2, \beta_{fo}=1$>
TP
σ_{29} <-, $\beta_{sc}=1, \beta_{fo}=0$>
CD_{fo}
σ_{30} <sl, sr, FB_{fo}, $\beta_{sc}=1$>
TP
σ_{31} <-, $\beta_{sc}=0$>
CD_{sc}
σ_{32} <sc.in,sr.in, ->
CS_{sc}, CS_{sr}
σ_{34} <-, $\beta_{sc}=4, \beta_{sr}=1$>
TP
σ_{35} <-, $\beta_{sc}=3, \beta_{sr}=0$>
CD_{sr}
σ_{36} <ct.in, $\beta_{sc}=3$>
CS_{ct}
σ_{37} <-, $\beta_{sc}=3, \beta_{ct}=1$>
TP
σ_{38} <-, $\beta_{sc}=2, \beta_{ct}=0$>
CD_{ct}
σ_{39} <tu.in, $\beta_{sc}=2$>
CS_{tu}
σ_{40} <-, $\beta_{sc}=2, \beta_{tu}=1$>
TP
σ_{41} <-, $\beta_{sc}=1, \beta_{tu}=0$>
CD_{tu}
σ_{42} <ac.in, ttu, $\beta_{sc}=1$>
CS_{ac}
σ_{43} <-, $\beta_{sc}=1, \beta_{ac}=1$>
TP
σ_{44} <-, $\beta_{sc}=0, \beta_{ac}=0$>
CD_{sc}, CD_{ac}
σ_{46} <sc.in,sr.in, ->
CS_{sc}, CS_{sr}
σ_{48} <-, $\beta_{sc}=4, \beta_{sr}=1$>
TP
σ_{49} <-, $\beta_{sc}=3, \beta_{sr}=0$>
CD_{sr}
σ_{50} <ct.in, $\beta_{sc}=3$>
CS_{ct}
σ_{51} <-, $\beta_{sc}=3, \beta_{ct}=1$>
TP
σ_{52} <-, $\beta_{sc}=2, \beta_{ct}=0$>
CD_{ct}
σ_{53} <tu.in, $\beta_{sc}=2$>
CS_{tu}
σ_{54} <-, $\beta_{sc}=2, \beta_{tu}=1$>
TP
σ_{55} <-, $\beta_{sc}=1, \beta_{tu}=0$>
CD_{tu}
σ_{56} <FB_{tu}, $\beta_{sc}=1$>
TP
σ_{57} <-, $\beta_{sc}=0$>
CD_{sc}
σ_{58} <sc.in,sr.in, ->
CS_{sc}, CS_{sr}
σ_{60} <-, $\beta_{sc}=4, \beta_{sr}=1$>
TP
σ_{61} <-, $\beta_{sc}=3, \beta_{sr}=0$>
CD_{sr}
σ_{62} <ct.in, $\beta_{sc}=3$>
CS_{ct}
σ_{63} <-, $\beta_{sc}=3, \beta_{ct}=1$>
TP
σ_{64} <-, $\beta_{sc}=2, \beta_{ct}=0$>
CD_{ct}
σ_{65} <fi.in, $\beta_{sc}=2$>
CS_{fi}
σ_{66} <-, $\beta_{sc}=2, \beta_{fi}=1$>
TP
σ_{67} <-, $\beta_{sc}=1, \beta_{fi}=0$>
CD_{fi}
σ_{68} <ac.in, $\beta_{sc}=1$>
CS_{ac}
σ_{69} <-, $\beta_{sc}=1, \beta_{ac}=1$>
TP
σ_{70} <-, $\beta_{sc}=0, \beta_{ac}=0$>
CD_{sc}, CD_{ac}
σ_{72} <sc.in,sr.in, ->
CS_{sc}, CS_{sr}
σ_{74} <-, $\beta_{sc}=4, \beta_{sr}=1$>
TP
σ_{75} <-, $\beta_{sc}=3, \beta_{sr}=0$>
CD_{sr}
σ_{76} <ct.in, $\beta_{sc}=3$>
CS_{ct}
σ_{77} <-, $\beta_{sc}=3, \beta_{ct}=1$>
TP
σ_{78} <-, $\beta_{sc}=2, \beta_{ct}=0$>
CD_{ct}
σ_{79} <fi.in, $\beta_{sc}=2$>
CS_{fi}
σ_{80} <-, $\beta_{sc}=2, \beta_{fi}=1$>
TP
σ_{81} <-, $\beta_{sc}=1, \beta_{fi}=0$>
CD_{fi}
σ_{82} <sl,FB_{fi}, $\beta_{sc}=1$>
TP
σ_{83} <-, $\beta_{sc}=0$>
CD_{sc}
σ_{84} <sc.in,sr.in, ->
CS_{sc}, CS_{sr}
σ_{86} <-, $\beta_{sc}=4, \beta_{sr}=1$>
TP
σ_{88} <-, $\beta_{sc}=3, \beta_{sr}=0$>
CD_{sr}
σ_{89} <ct.in, $\beta_{sc}=3$>
CS_{ct}
σ_{90} <-, $\beta_{sc}=3, \beta_{ct}=1$>
TP
σ_{91} <-, $\beta_{sc}=2, \beta_{ct}=0$>
CD_{ct}
σ_{92} <fo.in, $\beta_{sc}=2$>
CS_{fo}
σ_{93} <-, $\beta_{sc}=2, \beta_{fo}=1$>
TP

Figure 10.10: An execution trajectory of the design model of the autonomous truck.

Bibliography

[1] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[2] Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. Formal semantics of the procom real-time component model. In *35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, August 2009.

[3] Sverine Sentilles, Anders Pettersson, Dag Nyström, Thomas Nolte, Paul Pettersson, and Ivica Crnkovic. Save-IDE - a tool for design, analysis and implementation of component-based embedded systems. In *Proceedings of the Research Demo Track of the 31st International Conference on Software Engineering (ICSE)*, May 2009.

[4] 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.

[5] Tomas Bures, Jan Carlson, Ivica Crnkovic, Sverine Sentilles, and Aneta Vulgarakis. ProCom - the progress component model reference manual, version 1.0. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[6] Davor Slutej, John Håkansson, Jagadish Suryadevara, Cristina Seceleanu, and Paul Pettersson. Analyzing a pattern-based model of a real-time turntable system. In Barbora Zimmerova Jens Happe, editor, *6th International Workshop on Formal Engineering approaches to Software Components and Architectures(FESCA), ETAPS'09, York, UK, March*, pages

161–178. Electronic Notes in Theoretical Computer Science (ENTCS), Vol 253, Elsevier, September 2009.

[7] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[8] Pavel Krčál, Leonid Mokrushin, P.S. Thiagarajan, and Wang Yi. Timed vs time triggered automata. In Philippa Gardner and Nobuko Yoshida, editors, *Proc. of CONCUR'04.*, number 3170 in Lecture Notes in Computer Science, pages 340–354. Springer–Verlag, 2004.

[9] Joseph Sifakis, Stavros Tripakis, and Sergio Yovine. Building models of real-time systems from application software. In *In Proceedings of the IEEE Special issue on modeling and design of embedded*, pages 100–111. IEEE, 2003.

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

[11] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.

[12] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

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

[14] Frédéric Mallet, Robert de Simone, and Laurent Rioux. Event-triggered vs. time-triggered communications with UML Marte. In *FDL*, pages 154–159, 2008.

[15] Nenad Medvidovic, Paul Grünbacher, Alexander Egyed, and Barry W. Boehm. Bridging models across the software lifecycle. *J. Syst. Softw.*, 68(3):199–215, 2003.

Chapter 11

Paper D: Pattern-driven Support for Designing Component-based Architectural Models

Jagadish Suryadevara, Cristina Seceleanu, Paul Pettersson

In proceedings of the 18^{th}IEEE International Conference on Engineering of Computer-Based Systems (ECBS), April 2011.

Abstract

The development of embedded systems often requires the use of various models such as requirements specification, architectural (component-based), and deployment models, across different phases. However, there exists little design support for obtaining suitable component-based designs that satisfy specified requirements and timing constraints. In order to provide guided support for the design process of embedded systems, we introduce several component templates, referred as patterns, which we also formally verify against relevant properties. To illustrate the usefulness of the approach, we have applied the proposed patterns to obtain a component-based design of a temperature control system.

11.1 Introduction

To achieve behavioral predictability of an embedded system, one might need to use, during system development phases, extensive modeling and analysis, prior to the actual system implementation. In general, these phases rely on various models, such as, requirements specification, design, and deployment models, before the implementation stage. As different models are based on different semantics, and focus on different aspects of the system, a guided design process from one phase to another is needed. Further, the design process should ensure that the system aspects, such as, functional requirements, timing constraints etc, are met by all system models, from any design phase to the subsequent ones.

Designing an embedded system in a *component-based* style has become an attractive approach. With benefits ranging from simplification and parallel working to pluggable maintenance and reuse, the advantages are significant. In this context, an embedded system consists of identifiable, relatively independent and generally replaceable units of composition, called *components*, which encapsulate complex functionality. A *component model* defines syntax and semantics of a component language through its architectural level elements, such as, components, ports, connections, and connectors to build system parts and their compositions. There exist several component models such as JavaBeans [1], Koala [2], SOFA [3, 4], and ProCom [5, 6], to name a few.

In this paper, we present a pattern-based design process to develop component based designs of an embedded system that preserve the specified functional requirements and related timing constraints. We propose a set of component templates, called component patterns in this paper, to transform a specification model together with functional and timing constraints, into a corresponding component design. The proposed patterns are described in ProCom [5, 6], a language for component-based design of embedded systems. Further, the patterns are formally verified to satisfy relevant timing properties. This is done by translating the pattern specifications in ProCom, into corresponding timed automata models, and model-check the resulting models using UPPAAL [7].

To specify the functional requirements, and related timing constraints of a system, we use an extended form of UML statemachines [8] together with UML/Marte timing profile [9], as the specification models (also referred as *modemachines* in this paper). The timing constraints are specified using the standard constructs of Marte CCSL (Clock Constraints Specification Language).

Finally, to illustrate the applicability of our approach, we apply the patterns in the development of a ProCom based component design for a Temperature Control System (TCS).

The rest of the paper is organized as follows. In Section 11.2, we present an overview of the ProCom component language. As a running example, we describe a Temperature Control System (TCS), in Section 11.3. In Section 11.4, we present the specification language for modeling the functionality, and timing constraints of an embedded system. In Section 11.5, we propose a set of component patterns for modeling “timers”, “clocks”, “controllers”, as well as the periodic and sequential behaviors. Also, the formal verification of the patterns with respect to relevant properties is described in Section 11.6. In Section 11.7, a complete ProCom design of TCS is presented. Related work is discussed in Section 11.8. Finally, in Section 11.9, we conclude the paper with future directions of work.

11.2 ProCom Component Model: An overview

In this section, we present an overview of ProCom[1] [5, 6], a recently developed component model for designing *real-time* embedded systems in the vehicular and telecom domains. To address the different concerns that exist on different levels of granularity or various phases of system development, ProCom is organized into two distinct layers: ProSys and ProSave. The layers also differ in terms of architectural style and communication paradigm. In ProSys, 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*.

On the other hand, the lower layer, i.e., 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. 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. For detailed description of these elements, we refer to ProCom language reference manual [5].

[1]Developed at MRTC, Mälardalen University, Sweden.

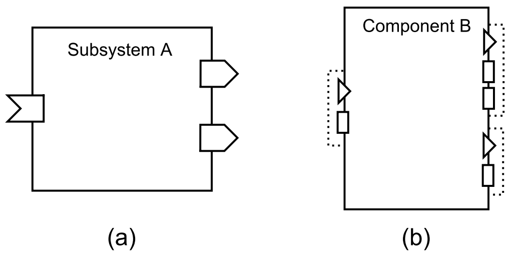

Figure 11.1: a) A ProSys subsystem and b) A ProSave component.

Fig. 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.

ProCom arcitectural elements have a precise formal semantics [10]. The semantics is described in terms of finite state machines extended with notions of urgency, timing, and priorities. Below, we informally describe the semantics of ProSave elements used in this paper. For through details, we refer the reader to [11].

- $Components$: internally, a ProSave component may be described by code or other inter-connected sub-components. The functionality of a component is captured by a set of services.

- $Services$: 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. When triggered, 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 (atomically) at its output port groups. 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.

- $Connections$: the migration of data or trigger over a ProSave connection is loss-less, atomic, and follows a push model. 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 data passes through a connector. In case more data (trigger) connections are enabled at the same time, the order is non-deterministic.

- $Connectors$: together with connections, connectors can be used to define complex data and control flow for a ProSave composition. ProSave defines different kinds of connectors such as *data fork, control fork, data or, control or, control join* and *selection*. A connector is a stateless component and executes atomically.

- $Clocks$: it 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.

Both layers of the ProCom are hierarchical, meaning that subsystems as well as components can be nested. The way in which the two layers are linked together is that a primitive ProSys subsystem (i.e., one that is not composed of other subsystems) can be further decomposed into ProSave components. Thus, a mapping has been defined between the message passing in ProSys and the trigger/data communication used in ProSave. At the bottom of the hierarchy, the behavior of a primitive ProSave component is implemented as a C function.

11.3 Example: Temperature Control System (TCS)

As the running example of the paper, we consider a Temperature Control System (TCS), for a heat producing reactor [12]. It has a collection of control rods that can be inserted into the core of the reactor, to control the heat producing (chain) reaction. If inserted, a control rod absorbs neutrons and consequently the reaction is slowed down, with the temperature inside the core decreasing at a fixed rate, depending on the rod inserted. When pulled out, the reaction speeds up, and temperature increases in the core. The main functionality of the TCS is to maintain the temperature in the reactor between the specified `MIN` and `MAX` values. However, when a rod has been used for cooling for a fixed duration, say "T" time units, it is then unavailable for a certain time duration, proportional to T.

In the next section, we propose a language for specification of functional, and timing properties of an embedded system. We use the language to specify the functionality, and timing constraints of the TCS system.

11.4 Our Specification Language: Modemachine + Marte CCSL

A specification is a way of explicitly stating system requirements and behavior. In this section, we propose a language for the abstract specification of system functionality, and related timing constraints of an embedded system. We use an extended form of statemachines that we call *Modemachines* (see Fig. 11.2). A modemachine adds to the original statemachine the system behavior, defined externally, which could be in turn a finite statemachine, a timed automaton etc. Also, in a modemachine, one can specify clock constraints, by using UML/Marte CCSL (Clock Constraint Specification Language) [9]. Graphically, a Modemachine is similar to UML statemachines [8].

11.4.1 Modemachine Definition and Graphical Notation

A *modemachine* is a tuple $\langle \mathcal{M}, \mathcal{B}, \mathcal{T}, \mathcal{C}, \mathcal{A}, \mathsf{s} \rangle$, where $\mathcal{M}$ is a set of *modes*, s is the *entry* mode, $\mathcal{B}$ is a set of externally defined *behaviors*, $\mathcal{A}$ is a set of *events*, $\mathcal{T} \subseteq \mathcal{M} \times \mathcal{A} \times \mathcal{M}$ is the set of transitions between *modes*, and $\mathcal{C}$ is a set of *clock constraints*.

If a mode contains other modes, it is called a *composite* mode. A mode with no internal modes is called an *atomic* mode. The elements of a modemachine are further described below, informally.

11.4.2 Modes, and Behaviors

A mode consists of a set of *behaviors*, where a *behavior* denotes the specific functionality of the system. A mode instance is the set of active behaviors at a particular instance of time. Behaviors can be externally specified, for example using external modeling tools such as Matlab/Simulink, UML Rhapsody, etc, or denote the reusable code of system functionalities. Within a mode or submode, behaviors execute concurrently, sequentially, or periodically, based on the associated mode constraints. Mode changes occur when a corresponding event or timeout occurs, or implicitly when all behaviors in the mode terminate. Further, an enabled mode change due to a timeout, has higher precedence over other simultaneously enabled mode changes, if any.

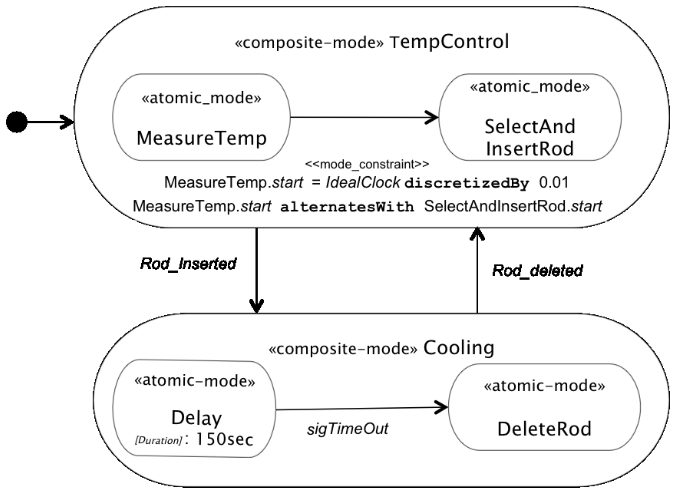
«composite-mode» TempControl
«atomic_mode» MeasureTemp
«atomic_mode» SelectAnd InsertRod
<<mode_constraint>>
MeasureTemp.start = IdealClock discretizedBy 0.01
MeasureTemp.start alternatesWith SelectAndInsertRod.start
Rod_Inserted
Rod_deleted
«composite-mode» Cooling
«atomic-mode» Delay [Duration]: 150sec
sigTimeOut
«atomic-mode» DeleteRod

Figure 11.2: Modemachine specification of a temperature control system.

11.4.3 Events, Triggers, and Timeouts

The execution of a behavior is triggered by the occurrence of an external event or time event. For an embedded system, the external *events* are generated by its *environment* consisting of sensors and actuators. A *trigger* denotes a periodic time event, and it is usually generated by "system clocks" (e.g., IdealClock in UML/Marte, for measurement of discrete chronometric time). Triggers can be used to specify periodic behaviors within a mode. A *timeout* denotes the expiry of the specified amount of (discrete) time duration. Timeouts are useful to model delays associated with an embedded system. A timeout can be associated with atomic modes, making them delay in particular states of the system model. The expiry of a timeout is denoted by the internally signaled `sigTimeOut` event.

11.4.4 Mode constraints using UML/Marte CCSL

The recently adopted UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE)[9] provides necessary and relevant features for modeling software of the real-time and embedded domain. Further, it aims at bringing interoperability between existing languages and formalisms of the real-time embedded domain. MARTE defines an expressive *Time-Model* for a generic timed interpretation of UML models. CCSL is a language annexed to MARTE specification. It is a declarative language that specifies constraints imposed on the clocks, i.e., both physical and logical, denoting the activation conditions of a model. Some of the constraints used in the paper are briefly discussed below:

- $discretizedBy$: specifies a discrete clock from a dense chronometric clock (e.g., *IdealClock* defined in Marte Time package). Expression (11.1) below defines a clock, whose period is 0.01 `s`, where `s` is the time unit of the IdealClock.

$$IdealClock\ \texttt{discretizedBy}\ 0.01 \tag{11.1}$$

- $isPeriodicOn$: specifies a discrete clock from another discrete clock of finer granularity (or faster clock). Expression (11.2) below defines a discrete clock that ticks 10 times slower than C (a tick of C' comes with every 10th tick of C):

$$C'\ \texttt{isPeriodicOn}\ C\ \texttt{period}\ 10 \tag{11.2}$$

- $alternatesWith$: implies causality between two clocks. Expression (11.3) states that each instance of clock C precedes and causes the corresponding instance of clock C'.

$$C\ \texttt{alternatesWith}\ C' \tag{11.3}$$

- $NFP\_duration$: supports the description of duration values with respect to an ideal chronometric clock. A *NFP_Duration* value is defined, in the non-functional types model library in Marte (i.e., MARTE::BasicNFP_Types), as a tuple containing a real value and a time unit.

Marte constraints in (11.1) and (11.2) are related to the basic synchronous constraint `coincidesWith` (also denoted by "=") which can also be used in specifying a mode constraint. The Marte constraint `alternatesWith` (see (11.3) above), is useful to specify constraints over logical clocks (e.g., non-periodic event occurrences, beginning and termination of behavior paths, etc), for instance, to specify causality dependencies between behavioral paths. Another useful Marte constraint for component models is *delayedFor* (e.g., "a `delayedFor` n `on` b", i.e., every n^{th} tick of b following a tick of a). Together with *precedes* relation ($\preceq$), it can be used to specify complex timing constraints of particular behaviors, like timing relationships between the start and the end of a behavior (e.g., B1.*finish* $\preceq$ (B1.start `delayedFor` 3 `on` C)).

11.4.5 Example Specification: TCS Modemachine

A *modemachine* specification of TCS is given in Fig. 11.2. At the top level, it contains two composite modes TempControl, and Cooling. Transitions between the modes are enabled with occurrence of events Rod_inserted, and Rod_deleted.

The composite mode TempControl contains two atomic submodes MeasureTemp, and SelectAndInsertRod. Further, the submode MeasureTemp contains a periodic behavior as specified by the associated Marte constraint `discretizedBy`. Also, the sequential dependency i.e., causality between behaviors of MeasureTemp, and SelectAndInsertRod is specified by the Marte constraint `alternatesWith`. SelectAndInsertRod contains the detailed behavior based on the data inputs received, for e.g., a rod selection and insertion is skipped when the current temperature value, communicated by the MeasureTemp component, is within the specified interval `MIN` and MAX, as described in Section 11.3.

The duration of the delay is specified using the Marte NFP_duration property. For instance, the composite mode Cooling contains a *delay* mode Delay characterized by a duration of 150 sec. When the timeout expires, it triggers the atomic mode DeleteRod.

Now let us assume a repository of ProCom components, which should be used for the architectural design of the TCS. Therefore, we will transform the above modemachine into a component-based design. To accomplish this, we need to tackle the following design issues/challenges:

- How to transform a composite mode with periodic, and sequential behaviors, into a component-compliant description?
- How to transform the control structure of a modemachine e.g. event, and signal based transitions?
- How to represent timeout in a component based-design?
- How to integrate different design aspects into a complete system design?

In order to address the above design issues, we introduce a set of component patterns that guides a designer in transforming a modemachine-based specification, e.g, the specification model of the temperature control system (TCS) in Fig. 11.2, into a corresponding ProCom based architectural design. The patterns are described in the next section.

11.5 Component Patterns

The component patterns, proposed in this section, provide simple mechanisms for modeling the time, and event based executions of system *behaviors* through reusable, easy to understand component designs. The patterns are described in ProCom component language (see Section 11.2). To illustrate our approach, we apply the proposed pattern-based support, in transforming the elements of the *modemachine* specification of the TCS system (see Fig. 11.2), into a corresponding design aspects in ProSave.

For ProCom-based pattern descriptions, we assume that the components are triggered, where necessary, by a clock, say *MainClock*, of fixed periodicity, say "P". In turn, the *MainClock* itself can be defined by the clock pattern (described below) using the *IdealClock* from the Marte time library. The *MainClock* is denoted by the conventional clock-icon symbol in the pattern descriptions below. Further, we specify additional constraints on the resulting designs (referred as "pattern constraints"), if any, by the pattern description (in a dotted text box, e.g., Fig. 11.3).

The set of patterns proposed below, address the design issues identified earlier, in the previous section: the "Timer Pattern" characterizes a time out in a component based design; the "Discrete Clock Pattern" addresses the clock synchronization problem between clocks of different granularity; the "Periodic Behavior Pattern" represents the design of periodically executed components; finally the "Controller Pattern" addresses event-triggered executions in a component-based design, and the development of a complete system design.

11.5.1 Timer Pattern

Timers, and timeouts constitute important aspects of an embedded system behavior. The pattern models a timeout (or delay) behavior of a system or its parts. It is triggered by a discrete, chronometric clock e.g., the `IdealClock` in UML/Marte time package. When triggered, time is internally measured (using a state variable) until the specified duration/delay units are expired. The output, i.e., the timeout `sigTimeOut`, is indicated as both data and control. This facilitates using the timeout as either sampled data or reactive trigger (de-

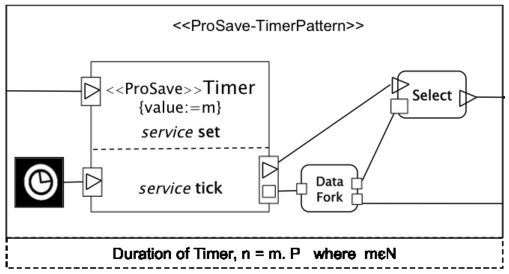

Figure 11.3: The timer pattern in ProSave.

sign choice based on the specified timing constraints). Further, the pattern specifies the timer mechanism set to assign the timer value (the value itself can be assigned statically or dynamically).

A ProSave description of the pattern is given in Fig. 11.3. The component Timer contains two services set, and tick. The service set, when triggered, sets the timer value (based on the statically assigned duration value through corresponding data port). The service tick, the periodic behavior triggered by the *MainClock*, decrements the timer value, if set, during each execution cycle and generates a timeout (denoted by `sigTimeOut`) when the value becomes zero. The connectors Select, DataFork are required to differentiate the final timeout output from trigger outputs corresponding to individual executions of Timer component (due to the semantics of a ProSave component).

The timer pattern corresponds to the Marte clock constraint in (11.4) below, where *n* is a natural number, and **s** is the time unit. However, the timer pattern suffers from a *jitter* of one period of the triggering clock i.e., the *MainClock* (as verified in Section 11.6). This implies the need for choosing a suitable granularity for the *MainClock*. Further, this should be taken into consideration while evaluating other timing aspects of the design, such as, end-to-end latency.

$$NFP\_duration = n \ \mathbf{s} \tag{11.4}$$

Application of the timer pattern to TCS: The pattern can be applied to transform a *delay* mode (i.e. an atomic mode with NFP_duration value) of a modemachine into a corresponding design in ProSave. For example, in TCS specification, the internal mode Delay (within the composite mode Cooling, see Fig. 11.2), is translated into a ProSave design, as shown in Fig. 11.4. Further, if the delay mode is not connected by a transition to any other internal

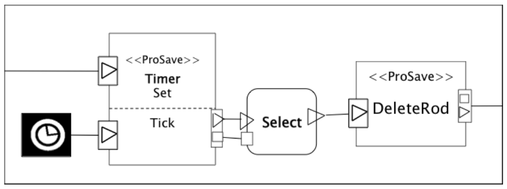

Figure 11.4: Transformation of the composite mode Cooling of TCS into a ProSave Design, by applying the timer pattern.

mode, its timeout i.e., `sigTimeOut` is communicated to the controller (described below, by the *controller pattern*) of a containing composite mode. For TCS, the *time-out* from Delay mode triggers the mode DeleteRod.

11.5.2 Discrete Clock Pattern

Clocks are central to embedded system behavior. The pattern models a coarse-grained discrete clock (i.e., a slower clock) triggered by a finer-grained clock (e.g., the *MainClock*). Also, the pattern facilitates the synchronization of various clocks within a component-based design.

The pattern is similar to the timer pattern described above, but does not require a set operation (as the state variable is simply incremented, when triggered). Further, unlike the timer pattern, the output of a discrete clock pattern is always a trigger rather than data as this is justified by the fact that clock *ticks* represent causality, between the clock and the triggered component, in a component-based design.

A ProSave description of the pattern is given in Fig. 11.5. The service tick, when triggered by the finer-grained clock, e.g., *MainClock*, increments the value of the state variable, modulo **m** (see the associated pattern constraint). The connector Select is needed to output the trigger only when the specified period expires, indicated by the associated boolean data output port. Hence, the final output trigger corresponds to a *tick* for every **m** ticks of the triggering clock.

The discrete clock pattern corresponds to the following Marte clock constraints as shown in (11.5), (11.6) below. Except for the initial *tick*, the discrete clock pattern does not suffer from any *jitter* (as verified in Section 11.6). This is consistent with the ProSave clock semantics.

$$MainClock\ \texttt{discretizedBy}\ n \tag{11.5}$$

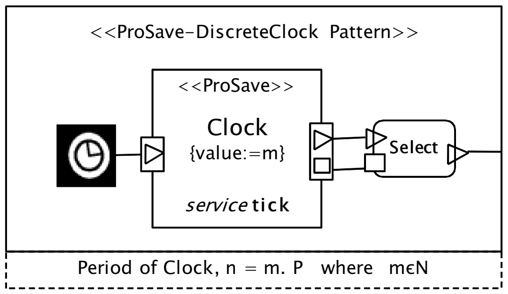
<<ProSave-DiscreteClock Pattern>>
<<ProSave>>
Clock
{value:=m}
service tick
Select
Period of Clock, n = m. P where m∈N

Figure 11.5: The discrete clock pattern in ProSave.

$$\texttt{isPeriodicOn}\ MainClock\ \texttt{Period}\ P \tag{11.6}$$

Application of the discrete clock pattern to TCS: The pattern can support the design of periodic behaviors in a component-based design, also described by the following patterns below. Additionally, it can be used to synchronize different clocks in a design. This not only simplifies the design, increasing its readability, and understandability, but avoids clock jitters and corresponding unpredictable delays, if any, which can be caused by different clocks. In the case of TCS component-based design, as shown in Fig. 11.11, the MainClock triggers both the Controller component, and the Timer component. Additionally, it could also trigger the Clock component, which instead is triggered by the Controller, for further simplification of the design.

11.5.3 Periodic Behavior Pattern

An embedded system is commonly described as a collection of periodic *behaviors*. The pattern describes two *behaviors*, say B1, and B2, where the periodic behavior B1 triggers the execution of B2. This causality makes the behavior B2 sequential, and also periodic. However, it is generally important for behavior B2 to act on the output generated from B1, which entails the constraint that "B2 *must be* at idle state when B1 completes the execution".

In Fig. 11.6, we give the ProSave description of this pattern. The component B1 (containing the *behavior* B1) is triggered by a clock of corresponding periodicity (can be designed using the discrete clock pattern described above). Further, the output of B1 triggers the component containing the *behavior* B2. However, the design must ensure that the specified pattern constraint is preserved. That is, B2 must be *idle,* when B1 completes its execution. The formal

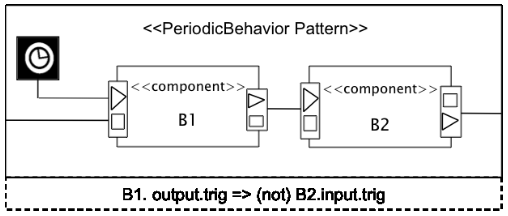
<<PeriodicBehavior Pattern>>
<<component>>
B1
<<component>>
B2
B1. output.trig => (not) B2.input.trig

Figure 11.6: The periodic behavior pattern in ProSave.

verification of the pattern (see Section 11.6), verifies the conditions for the constraint to hold, in terms of the period of B1, and also the end-to-end response time of B1, and B2.

The pattern corresponds to the following Marte clock constraint in (11.7) below.

$$B1.\mathit{finish}\ \texttt{alternatesWith}\ B2.\mathit{start} \qquad (11.7)$$

Application of the periodic behavior pattern to TCS: The pattern can be used in transforming a mode with periodic behaviors into corresponding component-based design. In the TCS modemachine (Fig. 11.2), the composite mode TempControl contains atomic submodes with periodic, and sequential behaviors MeasureTemp, SelectAndInsertRod, respectively. Thus, the composite mode TempControl can be translated into a ProSave design as shown in Fig. 11.7. The Clock component is designed by the application of the discrete clock pattern, and based on the periodicity of MeasureTemp mode behavior (represented by TempControl component), as specified by the corresponding timing constraint (see Fig. 11.2). When triggered by the Clock component periodically, the TempControl executes by reading the temperature data and provides output, temperature deviation within the allowed interval. This value is read by SelectAndInsertRod component to determine if a control rod is required to be inserted or not.

11.5.4 Controller Pattern

The behavior of an embedded system consists mainly of event-, or time-triggered *behaviors*. We have already covered the time-triggered behaviors by the patterns described above. Here, we introduce the controller pattern, to describe the event triggered execution of system *behaviors*. This corresponds to the reactive part of the system behavior.

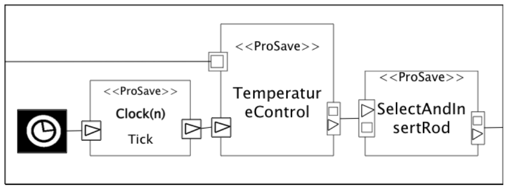

Figure 11.7: Transformation of composite mode TempControl of TCS into a ProSave design, by applying the periodic behavior pattern.

In principle, a component-based design is based on time-triggered, and control-, data-flow semantics. Hence, the pattern transforms event-based execution of mode *behaviors* into the executions based on sampling of the environment data corresponding to sensors, and actuators. Within a component-based design, events can be represented by pre-defined predicates over the environment data [13]. When triggered by a system clock (e.g., the *MainClock* discussed previously), the data is sampled to determine the occurrence of events, through the evaluation of corresponding predicates.

Fig. 11.8 shows the ProSave design of the pattern. The Controller component is triggered by a system clock, e.g., *MainClock*, periodically (in an implementation, the controller thus becomes a periodic task in the system). The periodicity of the clock is to be determined by the periodicity of data occurrences or their criticality. Also, there can exist multiple clocks of different periodicity (can be or-ed using ProSave connector ControlOr). Further, the controller implements the mode change behavior of a modemachine(e.g., Fig. 11.2). It also implements a datastructure representing the predicate-event mapping ([13]) described above. The controller can be triggered by internal signals i.e., `sigTimeOut`, when the signal represents a trigger rather than data (as described in the Timer pattern previously).

Application of the controller pattern to TCS: The pattern can be used in transforming the control structure of a modemachine specification into corresponding component-based design. For example, the event-based transitions corresponding to the top-level control structure of the TCS modemachine specification (Fig. 11.2), is transformed into the corresponding component-based design in Fig. 11.9. When triggered, the Controller evaluates the predicates corresponding to the event occurrence Rod_inserted, and Rod_deleted, respectively. In this case, no timeout is communicated to the controller, hence using the control-or connector, shown in the pattern, is not required.

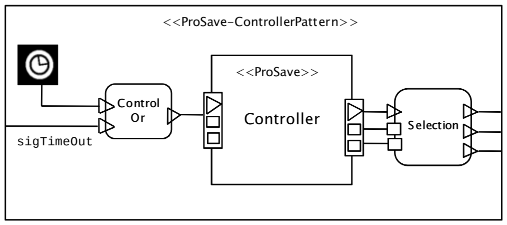

Figure 11.8: The controller pattern in ProSave

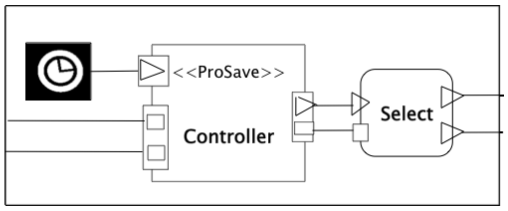

Figure 11.9: Transformation of the top level mode transitions of TCS into a ProSave design, by applying the controller pattern.

11.6 Pattern Verification

In this section, we describe the formal verification with respect to component patterns, presented in the previous section. The approach is based on the formal semantics of the architectural elements of the ProSave language [10, 11]. The formal semantics is based on an extension of finite state machine formalism with the notions of urgency, priority etc. The semantics of the formalism itself was given in terms of timed automata (TA) [14]. This provides a mechanism to formally verify ProSave designs using UPPAAL, the timed automata based model-checker [7] .

For formal verification, a component pattern is translated into the corresponding network of timed automata, based on the underlying semantics of constituting ProSave elements.

11.6.1 Verification of periodic behavior pattern

For periodic behavior pattern (in Fig. 11.6), the corresponding network of timed automata is given in Fig. 11.10. Each of the timed automata ClockTA,

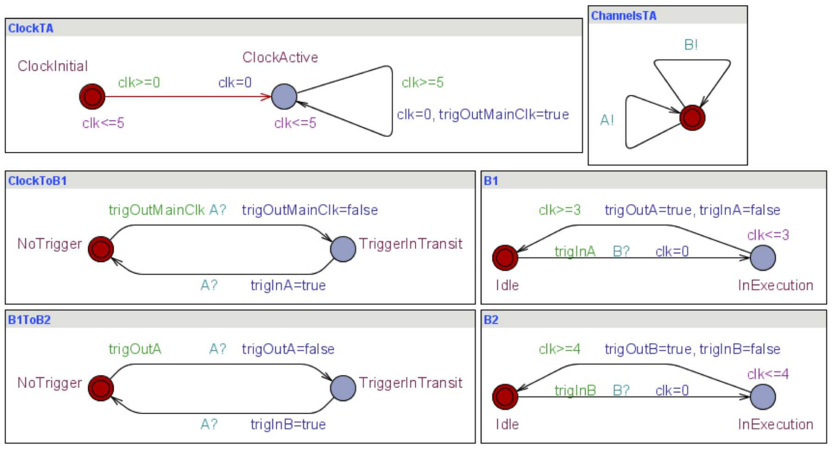

Figure 11.10: Translation of the periodic behavior pattern in ProSave into the corresponding network of timed automata.

ClockToB1, B1, B1ToB2, B2 correspond to the periodic trigger to B1, trigger connection to B1, component B1, trigger connection from B1 to B2, and component B2, respectively. Also, the end-to-end response time (say, R) of components B1, B2 are modeled in corresponding TA (3, 4 in this example). ChannelsTA denotes the timed automaton that contains complementary channels, corresponding to urgent channels A, B of other TA.

On the above models, we have verified with UPPAAL [7], that the following properties are satisfied by the periodic behavior pattern:

$$\texttt{A[] not deadlock} \tag{11.8}$$

$$\texttt{A[]}\ B1.trigOut\ \texttt{imply}\ (\texttt{not}\ B2.InExecution) \tag{11.9}$$

Property (11.8) states that there is no deadlock in the pattern. Though basic, this is a very important model feasibility property. Property (11.9) verifies the main constraint of the pattern i.e., that B2 is idle, that is, ready to begin execution, whenever B1 terminates its execution. However, it is observed that this property is satisfied, provided that the following conditions hold:

Periodicity of B1= $\begin{cases} \leq E_{B1} & \text{if} \quad E_{B1} > E_{B2} \\ > E_{B2} & \text{if} \quad E_{B2} > E_{B1} \end{cases}$ where

E_{B1}, E_{B2} denote the response time of B1, B2 respectively.

11.6.2 Verification of other Patterns

In addition to "no deadlock", we have verified other properties like the ones expressed by (10 - 12), this time for the Timer pattern (see Fig. 11.3). Expression (11.10) describes a liveness property (also called *leads to*, or *response* property [7]), as follows: when the timer is set, it eventually performs a timeout. Formulae (11.11), (11.12), verified to be satisfied by the pattern, indicate that the timer duration has a jitter of one period of the *MainClock*.

$$timerValue == Set.N \rightsquigarrow timerValue == 0 \tag{11.10}$$

$$\texttt{A[]}\ (TimeOut\ \texttt{imply}\ (obsClk >= (m-1) * P)) \tag{11.11}$$

$$\texttt{A[]}\ (TimeOut\ \texttt{imply}\ (obsClk <= (m+1) * P)) \tag{11.12}$$

For the discrete clock pattern, we have model-checked the corresponding network of automata against the following properties: deadlock freedom, liveness (as given by (11.10)), and jitter freedom (see (11.13)). The latter means that the Clock discretizes the *MainClock* perfectly, without any jitter, unlike the timeout duration of the timer pattern:

$$\texttt{A[]}\ TimeOut\ \texttt{imply}\ (obsClk = m * P) \tag{11.13}$$

11.7 Temperature Control System: A Complete ProCom Design

In this section, we complete the approach introduced in sections 11.4 and 11.5, by describing the final steps leading to a complete ProCom system design.

We have applied the *Timer* pattern, of the Fig. 11.3, to transform the composite mode Cooling of the TCS specification (Fig. 11.2), which contains the delay mode Delay, and an atomic mode DeleteRod with its corresponding *behavior*. The expiration of the timeout, signaled by the event `sigTimeOut` from the Timer component, triggers the execution of the *behavior* in DeleteRod.

To recall, Fig. 11.5 presents a ProSave design corresponding to the composite mode TempControl in the TCS specification. The corresponding design in ProSave, is obtained by applying the discrete clock pattern to the composite mode.

Also, the top level control structure, corresponding to the reactive behavior of the TCS modemachine (Fig. 11.2), with respect to the events Rod_inserted, and Rod_deleted is translated into the corresponding ProSave design in Fig. 11.8, through the controller pattern.

The complete ProCom design of the TCS is presented in Fig. 11.11. For simplicity, the complete system is represented as a single ProSys subsystem (see Section 11.2). For integrating different design parts, for instance, those described above, the following design steps are applied:

- *Synchronize the clocks using discrete clock pattern*: different clocks in the component-based design can be synchronized by applying the discrete clock pattern, and a finest-grained clock, e.g., the *MainClock*. This not only simplifies the component design, but also minimizes clock jitters, if any. For TCS, the different clocks, due to Controller, PeriodicBehavior, and Timer patterns, are synchronized using the MainClock.

- *Interconnect ProSys message ports with ProSave ports*: the system can be designed as a basic ProSys system (also called ProSave Subsystem) by connecting message ports to ProSave control, and data ports (as shown in Figure 11.11). Sensor and other data values, received as messages through ProSys message ports are forwarded to the internal ProSave components through their ports.

11.8 Related work

In the domain of synchronous languages [15], mode automata and the notion of running modes have been introduced, to reduce the gap between the initial design of a system and the program written for it. The formalism has been proposed to support both dataflow, and imperative styles. The *modemachine* described in this paper corresponds to the event-based, hierarchical, high-level control structure of the system and associated timing constraints.

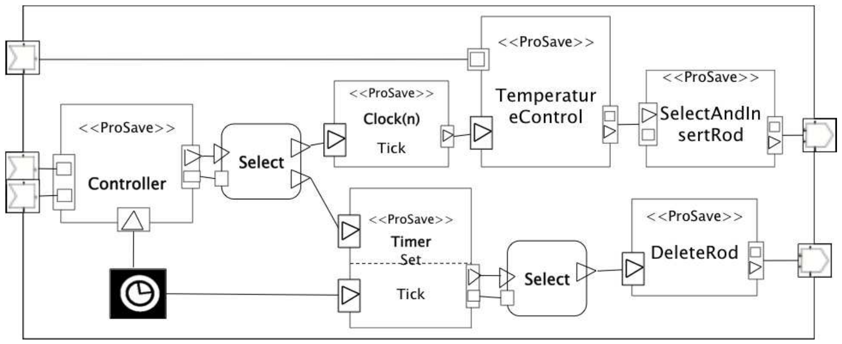

Figure 11.11: The Temperature Control System in ProCom: a ProSys component made of ProSave components.

Sandén proposes the “state-machine” pattern [16], for designing concurrent real-time software in Ada [17]. Many possible implementations of the pattern, corresponding to concurrent, reactive, and time-triggered behaviors, are described. Also, patterns for non-functional aspects such as resource usage, quality-of-service have been proposed [18]. However, such patterns focus on the design or implementation phase of the system. The patterns proposed in this paper support the design process, by directly mapping the specification aspects, with associated timing constraints, into the corresponding design elements.

Maxwell et al. have proposed a formal framework [19] for heuristics-based transformation of architectural designs. The authors capture heuristics in a structured and formal manner, such that the architectural transformations can be performed for optimizing the non-functional qualities of a system. Denford et al. have proposed an architectural refinement method [20] that focuses on non-functional requirements e.g., reliability, performance, while still addressing the functional requirements. While these works focus on non-functional aspects such as performance, we address architectural designs through timing constraints of embedded systems. However, this is done by including the functional requirements also.

UML/Marte profile is extensively used in the context of AADL (Architecture analysis and design language [21]) for component-based designs of real-time, embedded systems [22, 23]. AADL supports the modeling of both software components such as thread, subprogram, process, and platform components, e.g., bus, memory, processor, and device. However, AADL introduces avoidable redundancies that obscure the model and may even lead to design inconsistence. To address this deficiency, the Marte clock constraints have been used [23] to precisely specify both event, and time triggered communications for AADL models, and to compute end-to-end flow latency. These works focus on models related to software and platform mapping. In this paper, we offer formally verified support for component-based system design, in the form of patterns based on timing constraints.

EastADL [24] is a layered architecture language for model-based development of automotive software. To address various concerns of system's lifecycle development, it provides abstraction layers such as feature level, requirements, analysis, design, and implementation. Mallet et al. have described Marte CCSL specification of EastADL timing requirements [25]. This enables the use of Marte tools for timing verification of EastADL requirements. The work is similar to model driven aspects underlying the proposed patterns in this paper.

11.9 Conclusions

In this paper, we have proposed a set of component based patterns for developing embedded system designs. The patterns are based on the specification of reactive, and time-triggered behaviors of an embedded system. An extended form of statemachine, referred as modemachine, combined with UML/Marte clock constraints is used as the specification language. We have proposed component patterns for clocks, timers, periodic, and reactive behaviors. Also, we have described the implementation of the proposed patterns in the ProCom language, in order to support the design process based on the specification of functionality, and timing constraints. Further, we have described the correspondence of the proposed patterns with related UML/Marte clock constraints.

To guarantee timing correctness aspects, we have formally verified our patterns, by model checking their corresponding timed automata models, in UPPAAL. This facilitates the development of component based-design models with precise timing aspects. We have demonstrated the approach, by transforming the modemachine specification of an example temperature control system, into a corresponding design in ProCom component model. The explicit representation of running modes in the design, by application of the proposed patterns, may be useful for developing efficient deployment models. However, this requires further validation. Also, we intend to extend the approach to other Marte constraints, and validate the approach with complex systems. Further, we plan to work on the compositional verification of timing properties of the resulting component-based system designs.

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. In *IEEE Computer*, pages 78–85. IEEE, March 2000.

[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] T. Bureš, J. Carlson, I. Crnković, S. Sentilles, and A. 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] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[8] Object Management Group. UML 2.0 Superstructure Specification, The OMG Final Adopted Specification, 2003.

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

[10] Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. Formal semantics of the procom real-time component model. In *35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, August 2009.

[11] Jagadish Suryadevara, Aneta Vulgarakis, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. ProCom: Formal semantics. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-234/2009-1-SE, Mälardalen University, March 2009.

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

[13] Jagadish Suryadevara, Eun-Young Kang, Cristina Seceleanu, and Paul Pettersson. Bridging the semantic gap between abstract models of embedded systems. In Lars Grunske and Ralf Reussner, editors, *13th International Symposium on Component Based Software Engineering (CBSE)*. Springer LNCS, vol 6092, June 2010.

[14] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[15] Florence Maraninchi and Yann Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46:219–254, March 2003.

[16] Bo I. Sandén. The state-machine pattern. In *Proceedings of the conference on TRI-Ada '96: disciplined software development with Ada*, pages 135–142, New York, NY, USA, 1996. ACM.

[17] A. Burns and A. Wellings. *Concurrency in Ada*. Cambridge University Press, 1995.

[18] Joseph P. Loyall, Paul Rubel, Richard Schantz, Michael Atighetchi, and John Zinky. Emerging patterns in adaptive, distributed real-time, embedded middleware. In *9th Conference on Pattern Language of Programs*, September 2002.

[19] Cameron Maxwell, Tim O'Neill, and John Leaney. Formal architecture transformation using heuristics. In *Engineering of Computer-Based Systems, 2007. ECBS '07. 14th Annual IEEE International Conference and Workshops on the*, pages 15 –24, March 2007.

[20] M. Denford, John. Leaney, and Tim. OŃeill. Non-functional refinement of computer based systems architecture. In *Proceedings of the 11th IEEE International Conference and Workshop on Engineering of Computer-Based Systems*, pages 168–, Washington, DC, USA, 2004. IEEE Computer Society.

[21] Society of Automotive Engineers (SAE). Architecture analysis and design language (AADL), June 2006.

[22] M. Faugere, T. Bourbeau, R. De Simone, and S. Gerard. MARTE: Also an UML profile for modeling AADL applications. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 359 –364, 2007.

[23] F. Mallet, R. de Simone, and L. Rioux. Event-triggered vs. time-triggered communications with UML MARTE. In *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pages 154 –159, 2008.

[24] ATESST (Advancing Traffic Efficiency through Software Technology). East-ADL2 specification, March 2008.

[25] F. Mallet, M.-A. Peraldi-Frati, and C. Andre. Marte CCSL to execute East-ADL timing requirements. In *Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC '09. IEEE International Symposium on*, pages 249 –253, March 2009.