

Hierarchical Modeling and Analysis of Timed Systems

Alexandre David

A Dissertation submitted
for the Degree of Doctor of Philosophy
Department of Information Technology
Uppsala University

November 2003



IT Technical report series 2003-050

ISSN 1404-3203

Dissertation for the Degree of Doctor of Philosophy in Computer Science with specialization in Real Time Systems presented at Uppsala University in 2003.

Abstract

David, A. 2003: Hierarchical Modeling and Analysis of Timed Systems.
IT Technical report series 2003-050 178 pp. Uppsala. ISSN 1404-3203.

UPPAAL is a tool for model-checking real-time systems developed jointly by Uppsala University and Aalborg University. It has been applied successfully in case studies ranging from communication protocols to multimedia applications. The tool is designed to verify systems that can be modeled as networks of timed automata. But it lacks support for systems with hierarchical structures, which makes the construction of large models difficult. In this thesis we improve the efficiency of UPPAAL with new data structures and extend its modeling language and its engine to support hierarchical constructs.

To investigate the limits of UPPAAL, we model and analyze an industrial field-bus communication protocol. To our knowledge, this case study is the largest application UPPAAL has been confronted to and we managed to verify the models. However, the hierarchical structure of the protocol is encoded as a network of automata without hierarchy, which artificially complicates the model. It turns out that we need to improve performance and enrich the modeling language.

To attack the performance bottlenecks, we unify the two central structures of the UPPAAL engine, the passed and waiting lists, and improve memory management to take advantage of data sharing between states. We present experimental results that demonstrate improvements by a factor 2 in time consumption and a factor 5 in memory consumption.

We enhance the modeling capabilities of UPPAAL by extending its input language with hierarchical constructs to structure the models. We have developed a verification engine that supports modeling of hierarchical systems without penalty in performance. To further benefit from the structures of models, we present an approximation technique that utilizes hierarchy in verification.

Finally, we propose a new architecture to integrate the different verification techniques into a common framework. It is designed as a pipeline built with components that are changed to fit particular experimental configurations and to add new features. The new engine of UPPAAL is based on this architecture. We believe that the architecture is applicable to other verification tools.

© Alexandre David 2003

ISSN 1404-3203

Printed in Sweden by Nina Tryckeri HB, Uppsala 2003.

Distributor: Department of Information Technology, Uppsala University, Box 337,
S-751 05 Uppsala, Sweden.

Acknowledgments

First of all I thank my supervisor Wang Yi. Without his guiding and support this thesis would never have been completed. I thank all current and former members of the UPPAAL team in Uppsala who are Tobias Amnell, Johan Bengtson, Elena Fersman, John Håkansson, Annika Karlsson, Pavel Krčál, Fredrik Larsson, Leonid Mokrushin, and Paul Pettersson for being such a stimulating group. I thank Gerd Behrmann and Kim Larsen, members of the UPPAAL team in Aalborg, for the fruitful collaboration we had. I thank Pedro R. D'Argenio for his precious help in the beginning of the case study project, as well as Ulf Hammar and Thomas Lindström who devoted their time to explain the protocol and the code. I am grateful to Parosh Abdullah, Bengt Johnsson, Martin Leucker, and Sergei Vorobyov who took the time to review my thesis. I thank all the people from the department for their kindness. Finally, I thank my family for their support and encouragements during these years spent in Sweden.

This work has been mainly supported by the Swedish Foundation for Strategic Research (SSF) via ARTES and partially by the Swedish Board for Technical Development (NUTEK).

This thesis is based on the following published papers. Parts of these papers are included in the chapters of the thesis.

- Alexandre David and Wang Yi. Modeling and Analysis of a Commercial Field Bus Protocol. In *Proceedings, 12th Euromicro Conference on Real-Time Systems, 2000*. IEEE Computer Society. Pages 165–172.
I participated in the project, developed the models, and wrote the paper.
- Alexandre David, M. Oliver Möller, and Wang Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In *Proceedings, Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002*. LNCS number 2306. Pages 218–232.
I participated in the discussions and wrote the sections on syntax and semantics.
- Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL Implementation Secretes. In *Proceedings, Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002*. LNCS number 2469. Pages 3–22.
I wrote the sections on the passed and waiting list unification and implemented the structure.
- Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, and Wang Yi. A Tool Architecture for the Next Generation of UPPAAL. In *Proceedings, 10th Anniversary Colloquium, Formal Methods at the Cross Roads: From Panacea to Foundational Support, 2003*.
I wrote the sections on the PW-List and the storage and implemented these structures.
- Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, and Wang Yi. Unification & Sharing in Timed Automata Verification. In *Proceedings, SPIN 2003*. LNCS number 2648. Pages 225–229.
I made the experiments and I participated in the discussions and the writing of the paper.
- Alexandre David, M. Oliver Möller, and Wang Yi. Verification of UML Statecharts with Real-Time Extensions. Technical report, Uppsala University. Number 2003-009. Revised and augmented version of the FASE paper.
I participated in the discussions and wrote the sections on syntax and semantics.

Contents

1	Introduction	1
1.1	The Theme of the Thesis	3
1.1.1	Industrial Case Study	3
1.1.2	Hierarchical Modeling and Analysis	4
1.1.3	Performance Issues in Verification	6
1.1.4	Tool Architecture	7
1.2	A Brief Introduction to UPPAAL	7
1.2.1	Timed Automata	8
1.2.2	Extended Timed Automata	10
1.2.3	The Engine	13
1.2.4	The Query Language	14
1.3	Contributions	15
2	An Industrial Case Study Using UPPAAL	17
2.1	The Protocol	18
2.1.1	Overview	18
2.1.2	Field Interface: The Transport Layer	18
2.1.3	Bus Coupler: The Data Link Layer	22
2.2	The Modeling Process	27
2.2.1	Modeling the Bus Coupler	29
2.2.2	Modeling the FI	38
2.2.3	Related Work	42
2.3	The Verification Process	43
2.3.1	The Classes of Properties	44
2.3.2	The Bus Coupler	45
2.3.3	The Field Interface	50
2.4	Conclusion	53

3	Improvements on the Current UPPAAL	55
3.1	Reachability Algorithm	55
3.1.1	PW-List	56
3.1.2	Sharing	60
3.2	Implementation	63
3.2.1	The PW-List Structure	63
3.2.2	The Storage Structure	64
3.3	Experimental Results	67
3.4	Related Techniques	68
3.5	Conclusion	72
4	Hierarchical Timed Automata	73
4.1	Introduction	74
4.1.1	Statecharts	74
4.1.2	UML Statecharts	77
4.1.3	HTA	79
4.2	Syntax	82
4.2.1	Data Components	83
4.2.2	Structural Components	84
4.2.3	Constraints for Well-Formed HTA	85
4.3	Operational Semantics	88
4.4	Pacemaker Example	95
4.5	Simplified HTA	98
4.5.1	Simplified HTA Syntax	99
4.5.2	Simplified HTA Semantics	99
4.5.3	Expressiveness	101
4.6	Case Study Revisited	102
4.7	Conclusion	104
5	A Verification Engine for Hierarchical Timed Models	105
5.1	Representation and Computation of States	106
5.1.1	Data Structures for Representing States	107
5.1.2	Computation of States for the Simplified HTA	110
5.1.3	Experimental Results	112
5.2	An Abstraction Technique for HTA	114
5.2.1	Background	114
5.2.2	Abstracting Away from Hierarchy	116
5.2.3	Spurious Traces	128
5.2.4	Implementation	128
5.3	Conclusion	135

6	A New Tool Architecture	137
6.1	A Pipeline Architecture for UPPAAL	138
6.1.1	Implementation	138
6.1.2	Typed Data Flow	141
6.1.3	Parser Library	143
6.2	Extensions	143
6.2.1	Plugin	143
6.2.2	Hierarchy	143
6.2.3	New Algorithms	144
6.3	Conclusion	144
7	Conclusions	147
8	Appendix	151
8.1	Automata Figures of the Case Study	151
8.2	Grammar of Hierarchical Timed Automata	156
	References	161

Chapter 1

Introduction

This thesis elaborates on improvements of a technique called *model-checking* to verify software and hardware. It is known that software is not error-free and the term “bug” is used for errors. Upon the release of Windows 2000 there were about 64000 known bugs but the software was released and it proved to work at the cost of occasional crashes and frequent updates to fix these bugs. For this kind of software such factors as time-to-market and usability are often more important than safety and correctness. There are other kinds of software concerned with control of safety critical systems such as the brake system in cars or navigation systems in planes. For such cases it is definitely not acceptable that the system has occasional crashes or does not behave as expected. The correctness of these systems is defined not only in terms of functionality but timeliness, i.e., the correct output must be produced when it is required. These systems are called *real-time* systems.

There are several techniques to check if a real-time system works as intended, or more formally, meets its specifications. Testing is used to simulate usage conditions and to check for correct outputs. This method makes use of test scenarios and test cases. The goal is to cover as much behavior as possible with a limited set of cases. Another method is to systematically check that the system satisfies a set of properties. This is the verification method using theorem proving or model-checking techniques. Theorem proving requires interactivity in the sense that theorem prover needs to be guided to prove properties. In the model-checking technique one models the system in a given language, gives it properties, and asks the model-checker to verify them. It is a fully automatic technique in the sense that no interaction is required once the model and the properties have been given. Figure 1.1 illustrates how model-checkers work. Given a model and a set of properties,

the model-checker checks them and may answer *yes* or *no* possibly with a trace explaining why the model satisfies the properties or not.

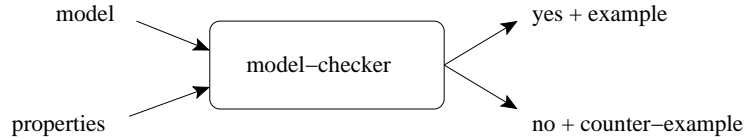


Figure 1.1: Model-checking principle.

The problem with model-checking comes from the cost of the verification. For most applications, the memory and time requirements in the verification process are exponential in the size of the model, that is, the verification suffers from the combinatorial explosion in the statespace. This is known as the *state explosion problem*. Fortunately, embedded real-time systems usually have limited amount of software. The limited complexity and the importance of correctness for embedded real-time systems make them good candidates for formal verification.

Several formalisms have been used to model real-time systems such as automata [HU01, ACD90, ACH92, AD90, HNSY92], process algebra [Yi91, Cer92, CGL93, HLY92], or timed Petri nets [Zub80, Zub85, RP85, BD91, Abd01, AN01]. One of the most successful approaches is the theory of timed automata of Alur and Dill [AD94]. Timed automata are an extension of the classical finite state automata with clock variables to model timing aspects. This model is further extended with integer variables (extended timed automata) and implemented in the tool UPPAAL [LPY97]. It has been successfully applied to a number of case studies [HSL97, LP97, DY00, LPY01] and it is appropriate for systems that can be modeled as networks of communicating processes. Improvements to the model-checker engine and faster computers allow us to handle larger and larger models. However, the input language lacks support to deal with larger models. Hierarchical variants of state machines such as Statecharts [Har87] are appropriate for large systems. These variants are appropriate for untimed systems and timed variants such as real-time UML [Dou99] have limited capabilities to model timed behaviors. The problems we are tackling are how to extend the well-known timed automata with hierarchy to obtain a rich timed language with hierarchical structures and how to improve efficiency of the model-checker UPPAAL.

In the following we present the theme of this thesis followed by a brief introduction to the UPPAAL tool. Finally, we detail the contributions of this

thesis and how the thesis is organized.

1.1 The Theme of the Thesis

The quick pace of computer hardware development (Moore's law) makes model-checkers run faster. However, improvements of the algorithms and the modeling language are more important: using a quick-sort algorithm to sort large sets on any ordinary computer is far better than using a bubble-sort algorithm on the top-of-the-line computer. The model-checker UPPAAL has improved considerably from its first version in 1995 to its current version 3.4. These improvements, along with new hardware, allow us to handle increasingly larger models but the language lacks structure to deal with such models. This thesis aims at improving the modeling and analysis capabilities of the model-checker UPPAAL. The introduction of hierarchy in the modeling language gives structured models and allows the use of an abstraction technique exploiting the hierarchy. Furthermore, we improve the performance and the architecture of the current engine.

1.1.1 Industrial Case Study

In recent years, various industrial case studies in model-checking have been reported. They are aiming at evaluating the industrial applicability and usefulness of existing verification techniques and tools. SDL [BHS91] has been applied mainly in telecommunication case studies [WC99, HM99] and has proven to be appropriate for these applications. Feature interaction in telecommunication systems [dB99, HS00] is also used to show how formal methods fare in practice. Safety critical applications, such as information display systems for air-traffic, are also in need of formal verification [Hal96]. Industrial case studies where UPPAAL is involved are reported in, e.g., [HSL97, LP97, DKRT97, BFK⁺98, HLS99, KLPW99, HLP00, LPY01]. We use these case studies to evaluate new algorithms for the tool.

The case study we present in this thesis, reported initially in [DY00], started with the idea to spread formal methods in a company and to evaluate the usefulness of our tool in a real-world situation. The company's interest in this project was to improve the development process, reduce the maintenance costs, and to improve the quality of the product with the help of formal methods. As for us, we aimed at evaluating what our tool needed in terms of graphical capabilities and verification. It became clear that the modeling language needed more structure and also the early models showed the weaknesses of the tool in its memory management. In this thesis we

propose hierarchical extensions and better data structures to improve on these two aspects.

This case study is the largest one reported so far where the UPPAAL tool has been applied. We model and verify different layers of a communication protocol in different steps. The study found no critical error in the sense that the protocol worked as intended and it turned out that it could be improved to avoid unnecessary retransmissions. We also suggested better message feedback from the lower layer to improve the implementation. The tool proves itself capable to handle this model despite its complexity and the hierarchical model gives better results.

1.1.2 Hierarchical Modeling and Analysis

The limits of the (flat) timed automata language become clear when you use the UPPAAL tool on large systems. One has to scroll the window up and down in the GUI to manage 20 or more processes placed in parallel in a moderate size application. It is difficult to manage this for a modeler and information concerning the structure is lost for the model-checker. Hierarchical state machines have been used to address both these issues. They give more structure and allow us to keep the original design of the systems in the models. Statecharts [Har87] have been proposed for this purpose and they are appropriate in conjunction with object-oriented modeling [HG97] for untimed systems. They are extensions of finite-state automata where the locations contain nested automata communicating via events.

To be more concrete we give a small TV set example and explain the basics of hierarchical modeling. In a statechart the locations are either *basic* locations that are not further refined, or *superlocation* that contain nested automata. The type of a superlocation may be *XOR* or *AND*. A *XOR* location has exactly one sublocation active at a time, whereas an *AND* location always has all of its sublocations active. Figure 1.2 shows a simple TV set example taken from [LMS97].

The system starts in the state (TV.OFF.STANBY). Depending on the events received, it can switch to the state (TV.ON.IMAGE.SHOW, TV.ON.SOUND.ON) or the state (TV.OFF.DISCONNECTED). The TV and the OFF automata are *XOR* superlocations where only one location is active at a time. The ON automaton is an *AND* location where all its sublocations (IMAGE and SOUND) are active if ON is active. In the state (ON.IMAGE.SHOW, ON.SOUND.ON), it is possible to switch videotext or mute the sound independently. It is also possible to switch off the TV set or disconnect it.

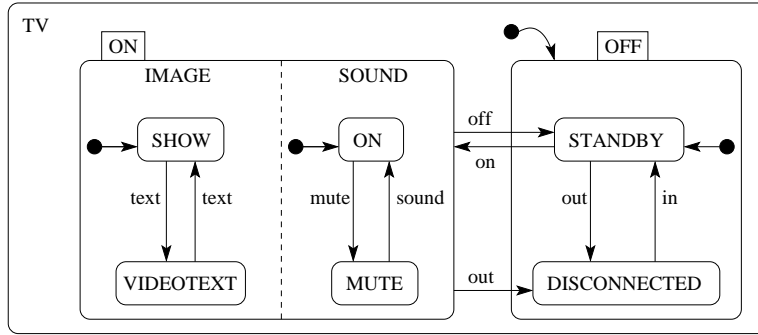


Figure 1.2: TV set statechart example.

Statecharts are in general models for event-driven systems, that is, transitions are driven by external (or internal) events. In the timed automata approach, we have channel synchronizations that are similar to events. The hierarchical approach allows us, as seen in the TV set example, to describe the model in a compact and structured manner. This is also a benefit for model-checkers that can organize their data to save time and memory. Furthermore, when the same automaton is used to describe several sublocations, it is possible to reuse previous exploration information [BLA⁺99, AMY02].

For the type of systems we model in UPPAAL, we need similar constructs in the language as in statecharts and also constructs to deal with time. We have not found these two features together in other formalisms that meet our needs. Modeling languages for timed systems in general have poor support for structuring of models and those with hierarchy have limited support for modeling of time. Therefore, our goal is to extend the timed automata language to include hierarchical constructs, thus having a language featuring strong timing modeling capabilities and hierarchy. There are other attempts where time is introduced to a hierarchical model: a profile for schedulability, performance, and time [IIC⁺02] has been proposed to extend the UML [BJR97]. The OMEGA project¹ aims at developing a methodology in UML for embedded and real-time systems based on formal techniques. In this context they define a real-time profile [GO03] and a kernel language for UML [DJVP03]. Most of the problems when dealing with UML lie in reducing it to a manageable subset to give well-defined semantics, and extending it with custom features. In our case time, we add hierarchy to the well-defined timed automata instead.

¹<http://www-omega.imag.fr>

1.1.3 Performance Issues in Verification

As the quick-sort versus bubble-sort example shows, algorithms are vital keys to efficiency. It turns out that the hierarchical extension we need gives a compact model, which is good for modeling but introduces challenges for model-checking: we have to handle models that are more compact but we have to keep at least the same efficiency. This means we need to improve model-checkers to handle larger systems. Dealing with more interesting problems often implies facing greater complexity. In the years of development of the UPPAAL tool we have improved considerably the algorithms and the used data structures. In this thesis we propose further refinements. These improvements were initially aimed at a UPPAAL engine for analyzing hierarchical models. It turns out that they are applicable for the current UPPAAL engine or any model-checker for timed systems.

The primary bottleneck of modern computers is the memory system. The CPUs are gaining power at an incredible pace (Moore's law) but the memory speed lags behind. Memory gets faster and cheaper but the gap behind CPUs is still growing. From this simple fact we looked into the model-checker to improve the memory footprint. It turns out that much of the used data can be shared. This was discovered by instrumenting UPPAAL. A special shared storage structure is proposed to take advantage of this. In addition we simplify the reachability algorithm by unifying its two main structures, the *waiting* and the *passed* lists. This improves speed and memory consumptions. There are many other algorithms to improve model-checkers. Compression [Hol97], selective state storing [LLPY97], and deallocation optimization [LPY00] are some of the existing memory optimizations. There has been progress on the symbolic representation of states [Dil89, YPD94, BLP⁺99] and associated reduction techniques [Yov97, LLY97]. Partial order [ABH⁺97, Pag96, Pel96, BJLY98] and symmetry reduction [ES97, ID96] are common techniques to reduce the statespace exploration.

When dealing with complex problems, e.g., EXPSPACE-complete for the hierarchical model, nothing can prevent the so-called *statespace* explosion. Theoretically the problem is intractable, no matter which algorithm or machine is used. In such cases where verification is not possible, approximation techniques are used. For this purpose we propose an abstraction based on the hierarchical model. This abstraction is used to cut down parts of the models by keeping some records of them. Other approximation techniques may under-approximate the statespace by using hashing [Hol91, WL93, DU95] or over-approximate it by using convex-hulls [LBB⁺01].

1.1.4 Tool Architecture

To obtain a reasonably efficient and robust tool, we have to follow a good software engineering approach to develop and maintain this software. After all, we write software that is used to check other software so we have to be careful about ours in the first place. There are other tools adapted for particular formalisms: Kronos [Yov97] and RED [Wan01] for timed automata, HyTech [HHWT97, AHH96] for hybrid systems, Murphi [DDHY92] and Spin [Hol97] for untimed systems. These tools follow their own designs but there are few publications on the architecture and design of these tools. When the source code is available, e.g., for Spin, it is possible to read but in practice it is hard to extract the architecture and design of such optimized code. The same holds for the algorithms: publications in the area are rich in algorithm descriptions, theoretical results, and experimental results [TAKB96, TY01, CCK⁺02, CGL94, CVWY92, CK97, ABH⁺97, DGKK98, Val90, Pel93, DU95, LNAB⁺98, BFH⁺01] but there is little information on how these techniques fit together into a common efficient architecture. We propose a flexible tool architecture that answers the needs of (i) easy configuration for experiments, (ii) maintainability with its pipeline made of separate components, and (iii) efficiency.

During the development of UPPAAL, we realized that the monolithic kernel of the engine is difficult to maintain and combinations of certain features are hindered due to the lack of flexibility. The new architecture is modular and is well adapted to model-checkers in general. We detail the different components (filters) of this architecture and we show how this allows us to improve performance, in particular to reduce the number of copies of data.

1.2 A Brief Introduction to UPPAAL

UPPAAL is a tool suite for validation and symbolic model-checking of real-time systems based on timed automata. It consists of a graphical user interface (GUI) and a stand-alone model-checker. The tool is appropriate for systems that can be modeled as a network of communicating processes. Communication is modeled with shared variables or channel synchronization. Clock variables are used to model time delays and integer variables to manipulate data. In the following we give the background on timed automata in UPPAAL followed by a brief introduction to UPPAAL.

1.2.1 Timed Automata

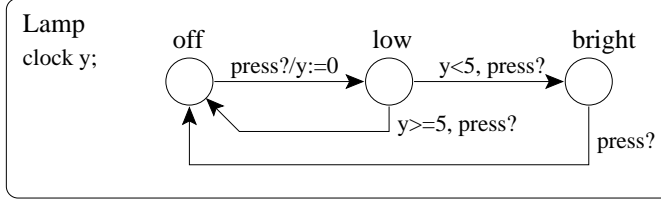
A timed automaton is a finite-state machine extended with clock variables used to measure delays. It uses a dense-time model where a clock variable can evaluate to a real number. All the clocks progress synchronously. We put several such timed automata in parallel to obtain a network of automata. Every automaton may fire an edge separately. The state of the system is then described by the different locations active in the automata. We prefer to call them locations instead of states to make the difference from the state of the system.

Two automata may synchronize using channels. When two synchronization actions match on enabled edges (e.g., $c!$ and $c?$) both edges are taken in the same transition leading to a new state where the locations in these two automata may be updated. The model is further extended with bounded integer variables that are part of the state. These variables are used as in programming languages: they are read, written, and are subject to common arithmetic operations.

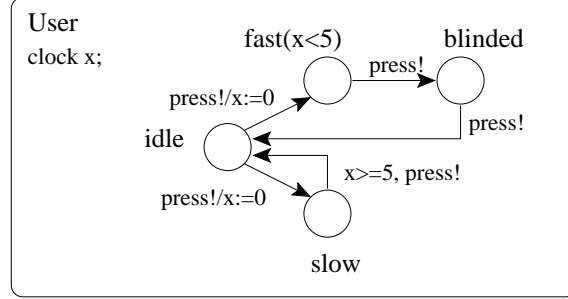
Figure 1.3(a) shows an example timed automaton modeling a simple lamp. The lamp has three locations: **off**, **low**, and **high**. If the user presses a button, i.e., synchronizes with **press?**, then the lamp is turned on. If the user presses the button again, the lamp is turned off. However, if the user is fast and quickly presses the button twice, the lamp is turned on and becomes bright. The user model is shown in Figure 1.3(b). The user may be slow or fast, which is modeled by the locations **slow** and **fast**. The clock variable x checks for time delays. The clock constraint $x \geq 5$ is used to force at least a delay of 5 time units from the **slow** location. The invariant $x < 5$ is used to force progress so that the location **fast** may stay active no more than 5 time units. The user decides non-deterministically to be fast or slow. If she is fast then she becomes blinded by the bright lamp.

We give now the basic definitions of the syntax and semantics for timed automata. We use the following notations: let C be a set of clocks, $B(C)$ the set of conjunctions over simple conditions of the form $x \bowtie c$ or $x - y \bowtie c$, where $x, y \in C$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. A timed automaton is a graph annotated with conditions and resets of non-negative real valued clocks in the form $x := 0$.

Definition 1 (Timed Automaton (TA)) A timed automaton over a set of clocks C is a tuple (L, l_0, E, I) , where L is a set of locations, $l_0 \in L$ is the initial location, $E \subseteq L \times (B(C) \times 2^C) \times L$ is a set of edges between locations with guards and clocks to be reset, and $I : L \rightarrow B(C)$ assigns invariants to



(a) Lamp model.



(b) User model.

Figure 1.3: Timed automata example.

locations. □

In UPPAAL, invariants are restricted conditions of the form $x \bowtie c$ for $\bowtie \in \{<, \leq\}$. To keep the notation simple at this stage we omit this detail.

We define now the semantics of timed automata. A clock valuation is a function $u : C \rightarrow \mathbb{R}_{\geq 0}$ from the set of clocks to the non-negative reals. Let \mathbb{R}^C be the set of all clock valuations. Let $u_0(x) = 0$ for all $x \in C$. We will abuse the notation by considering guards and invariants as sets of clock valuations, writing $u \in I(l)$ to mean that u satisfies $I(l)$.

Definition 2 (Semantics) The semantics of a timed automaton (L, l_0, E, I) over C is defined as a transition system $\langle S, s_0, \rightarrow \rangle$, where $S \subseteq L \times \mathbb{R}^C$ is the set of states, $s_0 = (l_0, u_0)$ is the initial state, and $\rightarrow \subseteq S \times S$ is the transition relation such that:

- $(l, u) \rightarrow (l, u + d)$ if $u \in I(l)$ and $u + d \in I(l)$, and
- $(l, u) \rightarrow (l', u')$ if there exists $e = (l, g, r, l') \in E$ s.t. $g(u)$, $u' = [r \mapsto 0]u$, and $u' \in I(l')$,

where for $d \in \mathbb{R}_{\geq 0}^C$, $u + d$ maps each clock x in C to the value $u(x) + d_x$, and $[r \mapsto 0]u$ denotes the clock valuation which maps each clock in r to 0 and agrees with u over $C \setminus r$. □

The semantics of timed automata results in an uncountable transition system. It is a well known fact that there exists an exact finite state abstraction based on convex polyhedra in \mathbb{R}^C called zones [Pet99, LPY95] (a zone can be represented by a conjunction in $B(C)$). This abstraction leads to the following symbolic semantics.

Definition 3 (Symbolic Semantics) Let $Z_0 = I(l_0) \wedge \bigwedge_{x,y \in C} x = y = 0$ be the initial zone. The symbolic semantics of a timed automaton (L, l_0, E, I) over C is defined as a transition system $\langle \mathcal{S}, f_0, \Rightarrow \rangle$ called the *symbolic reachability graph*, where $\mathcal{S} \subseteq L \times B(C)$ is the set of symbolic states, $f_0 = (l_0, Z_0)$ is the initial state, \Rightarrow is the transition relation and is defined by the following rules:

- $(l, Z) \xrightarrow{\delta} (l, \text{norm}(M, (Z \wedge I(l))^\dagger \wedge I(l)))$, and
- $(l, Z) \xrightarrow{e} (l', r(g \wedge Z \wedge I(l)) \wedge I(l'))$ if $e = (l, g, r, l') \in E$,

where $Z^\dagger = \{u + d \mid u \in Z \wedge d \in \mathbb{R}_{\geq 0}\}$ (the *future* operation), and $r(Z) = \{[r \mapsto 0]u \mid u \in Z\}$ (the *reset* operation). The function $\text{norm} : \mathbf{N} \times B(C) \rightarrow B(C)$ normalizes the clock constraints with respect to the maximum constant M of the timed automaton. \square

The relation $\xrightarrow{\delta}$ represents the delay transitions and \xrightarrow{e} the edge transitions. The classical representation of a zone is the Difference Bound Matrix (DBM) [Rok93, WT94]. The normalization problem and the algorithms to solve it are treated in [Ben02].

1.2.2 Extended Timed Automata

The UPPAAL modeling language extends timed automata with the following additional features:

- **Broadcast channels:** a broadcast channel synchronizes with as many edges as possible. In other words, an edge with $c!$, c being a broadcast channel, will synchronize with all possible edges with $c?$, possibly none if there is no such enabled edge.
- **Urgent channels:** a channel may be declared as urgent, in which case delays may not occur if a synchronization transition on that channel is enabled.
- **Urgent locations:** a location may be tagged as urgent, in which case delays may not occur when this location is active, i.e., present in the state.

- Committed locations: a location may be declared as committed, in which case it must be left immediately. This means that states having these locations active must take transitions to leave these locations without delay.
- Arrays of variables: this is a standard feature of programming languages.
- Arrays of channels: the channel synchronizations may be controlled by an integer expression. This feature enhances the compactness of models.
- Arrays of clocks: clock constraints and resets may be parameterized by integer expressions. This gives more flexibility to clock expressions.

As timed automata are extended with integer variables and synchronization channels, we add the following notations: let V be a set of integer variables and Ch a set of channels. Let $G(V, C)$ denote the set of conjunctions over boolean expressions of variables V and simple conditions of $B(C)$ as defined previously. Let $A(V)$ be the set of assignments over V , and $S(Ch)$ the set of channel synchronizations $c?$ or $c!$ where $c \in Ch$ and τ the internal action. In the following we omit urgency, committed locations, and broadcast communication for the sake of simplicity. These features extend the definitions we give in a natural way.

Definition 4 (Extended Timed Automata (XTA)) An extended timed automaton over C , V , and Ch is a tuple (L, L_0, E, I) , where L is a set of locations, $L_0 \subseteq L$ is the set of initial locations, $E \subseteq L \times (S(Ch) \times G(V, C) \times A(V) \times 2^C) \times L$ is the set of edges between locations with synchronizations $S(Ch)$, guards $G(V, C)$ over variables and clocks, assignments $A(V)$, and clocks to be reset. We use $l \xrightarrow{cgar} l'$ to denote such an edge. $I : L \rightarrow B(C)$ assigns invariants to locations. \square

We abuse the notation for I with $I(\{l_i\}) = \wedge_i I(l_i)$ for some subset of locations $\{l_i\}$. The synchronization on an edge may be empty in which case we note $(l, g, a, r, l') \in E$. A variable valuation is a function $v : V \rightarrow \mathbb{Z}$ from the set of variables to the set of integers. Let \mathbb{Z}^V be the set of all variable valuations and $v_0(i) = 0$ for all $i \in V$. We abuse the notation by considering $a(v)$ the updated valuation after an assignment $a \in A(V)$. We write $\bar{l}[l'_i/l_i]$ to denote the vector where the i th element l_i of \bar{l} is replaced by l'_i .

Definition 5 (Semantics for XTA) The semantics of an extended timed automaton (L, L_0, E, I) over C, V , and Ch is defined as a transition system $\langle S, s_0, \rightarrow \rangle$, where $S = 2^L \times \mathbb{R}^C \times \mathbb{Z}^V$ is the set of states, $s_0 = (\bar{l}_0, u_0, v_0)$ is the initial state, and $\rightarrow \subseteq S \times S$ is the transition relation defined by:

- $(\bar{l}, u, v) \rightarrow (\bar{l}, u + d, v)$ if $u \in I(s)$ and $u + d \in I(s)$.
- $(\bar{l}, u, v) \rightarrow (\bar{l}[l'_i/l_i], u', v')$ if there exists $l_i \xrightarrow{\tau gar} l'_i$ s.t. $g(u)$, $u' = [r \mapsto 0]u$ and $u' \in I(s)$, $v' = a(v)$.
- $(\bar{l}, u, v) \rightarrow (\bar{l}[l'_j/l_j, l'_i/l_i], u', v')$ if there exist $l_i \xrightarrow{c?g_i a_i r_i} l'_i$ and $l_j \xrightarrow{c!g_j a_j r_j} l'_j$ s.t. $g_i(u) \wedge g_j(u)$, $u' = [r_i \mapsto 0, r_j \mapsto 0]u$ and $u' \in I(s)$, $v' = a_i \circ a_j(v)$. \square

We omit conditions on S that define location vectors of 2^L as valid, that is, the locations may not belong to the same automaton. We aim at staying simple in the definitions at this stage. In the following, we refer extended timed automata as timed automata (TA).

As an example of an extended timed automaton, we give the model of the well-known Fischer's protocol. It is a mutual exclusion protocol for n processes that ensures that the location CS (critical section) is active only in one process at a time. Figure 1.4 shows one *template* automaton of the model. The actual automaton is made of the collection of the automata $P(1), P(2), P(3)$ for three modeled processes.

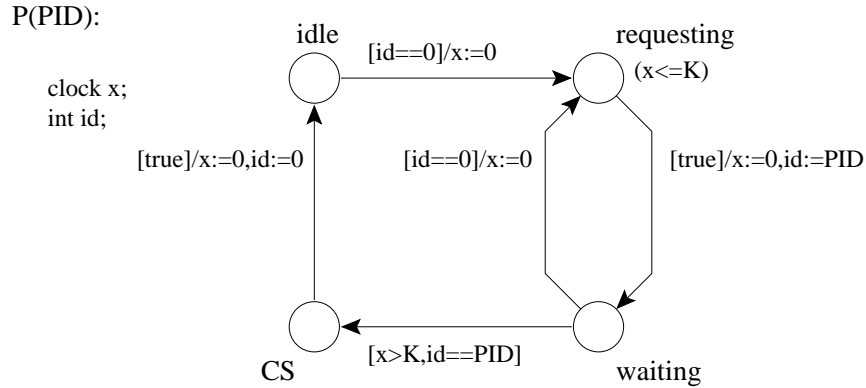


Figure 1.4: Template automaton of the Fischer's protocol.

Guards are written as $[expr]$, assignments and resets as $x := value$, and invariants as $(condition)$. K is a constant that represents the delay to wait,

the same for all processes. PID is the process identity constant, different for every process.

1.2.3 The Engine

The overall structure of UPPAAL is shown in Figure 1.5. There are two ways to use UPPAAL: one can use the graphical user interface (GUI) or the command line program *verifyta*. The GUI is a client program communicating with the server via the network. Having a client-server architecture allows us to run the server on a powerful station/server and use it remotely. The server and *verifyta* share the same engine written in C++. The GUI is written in Java and the engine is compiled for Sun/Solaris and x86/Linux/Windows.

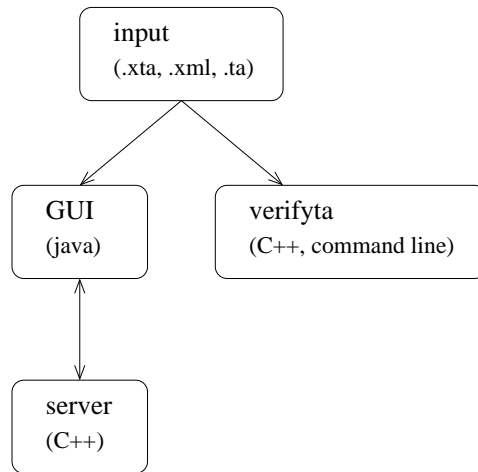


Figure 1.5: The UPPAAL tool.

The model-checking engine uses a number of optimization algorithms to get better time or memory performance. The mostly used algorithms are:

- *Active clock reduction* [Yov97] that detects locations where certain clocks are irrelevant.
- *Bitstate hashing* and *hash-compaction* [Hol98] that are under-approximation algorithms.
- *Convex-hull* approximation [Bal96] that is an over-approximation algorithm.
- *Guiding* [LBB⁺01] to speed up reachability.

- *Minimal graph reduction* [LLPY97] to reduce memory usage of zones.

1.2.4 The Query Language

The UPPAAL model-checker is able to verify if properties hold for a given extended timed automaton. These properties are defined in a subset of TCTL (timed computation tree logic).

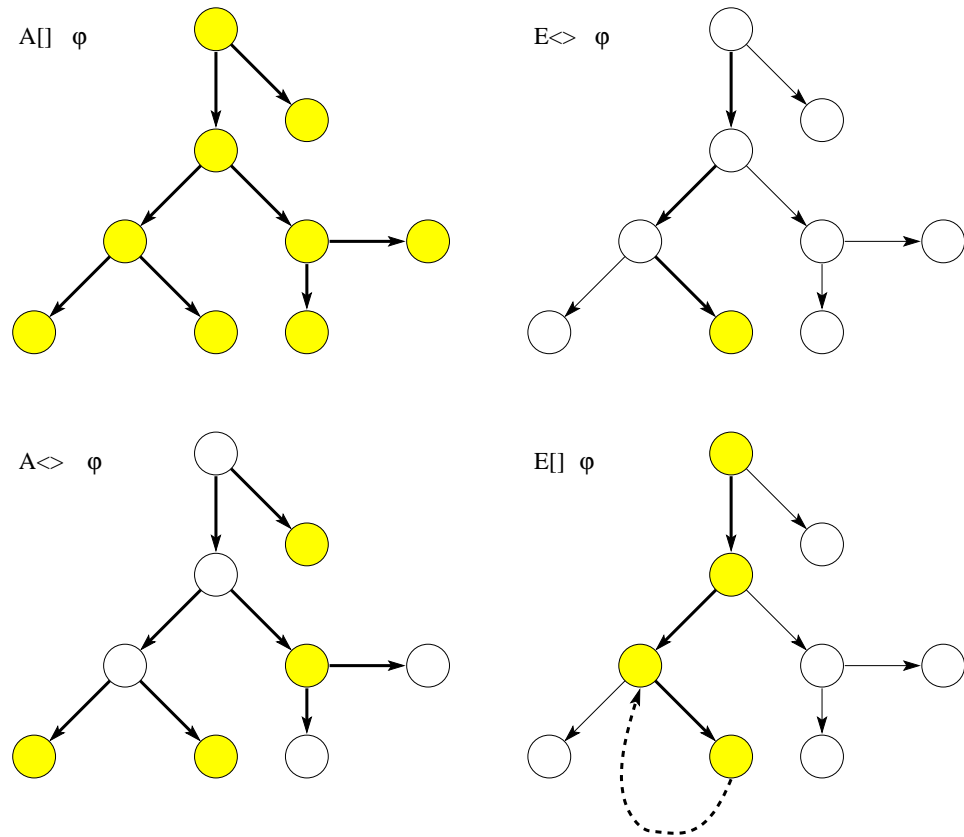


Figure 1.6: Basic CTL formulae.

A query is of the form:

- $A[] \phi$ "always globally ϕ ",
- $E <> \phi$ "exists eventually ϕ ",
- $A <> \phi$ "always eventually ϕ ",

- $E[] \phi$ “exists globally ϕ ”, or
- $A[](\phi \rightarrow A \langle \rangle \psi)$ “ ϕ always leads to ψ ”,

where ϕ and ψ are boolean expressions over locations, variables, and clocks. These queries are defined on paths: A applies for all paths and E for one existing path. $[]$ queries all states along paths and $\langle \rangle$ queries one state along paths. Figure 1.6 shows traces of states and paths for which CTL formulae hold. The filled states are those for which a given ϕ holds. Bold edges are used to show the paths the formulae evaluate on. The time part (**T**CTL) comes from the clock constraints used in ϕ (and ψ).

The formulae $A[] \phi$ and $E \langle \rangle \phi$ are *reachability* properties and are symmetric: $A[] \phi = \neg E \langle \rangle \neg \phi$. The $A[] \phi$ properties are also called safety properties because they check for a formula to hold for all the states. If such a property is not satisfied then $E \langle \rangle \neg \phi$ characterizes counter-example paths.

The formulae $A \langle \rangle \phi$ and $E[] \phi$ are *liveness* properties and are symmetric as well: $A \langle \rangle \phi = \neg E[] \neg \phi$. These properties involve a loop detection algorithm because $E[] \phi$ holds for an infinite trace where every state satisfies ϕ . As the (symbolic) statespace is finite, we are looking for loops.

1.3 Contributions

The main contributions of this thesis include (1) an industrial case study, (2) a number of improvements to the current UPPAAL model-checker, (3) a theory of hierarchical timed automata, and (4) a new tool architecture for the next generation of UPPAAL.

Case Study. The case study is the result of a joint project with industry based on a real-life product. To investigate the limits of UPPAAL, we model and analyze an industrial fieldbus communication protocol. This case study is the largest application UPPAAL has been successfully confronted to. To handle a complex system, we model it in several steps. As a result a number of improvements are proposed for the product.

Improvements to UPPAAL. To attack the performance bottlenecks, we propose improvements of (i) the reachability algorithm with help of a unified passed and waiting list called the *PW-List*, and (ii) the memory management using a *shared* storage structure. These structures give significant gains in memory and speed.

Hierarchical Timed Automata. We have designed a language for description of hierarchical models: Hierarchical Timed Automata (HTA). Furthermore, we have developed a verification engine that supports modeling of hierarchical systems without penalty in performance. The semantics addresses the complex issues caused by the constructs to deal with both time and hierarchy. We propose an abstraction model built upon the hierarchical model that approximates nested operations on integer variables and clock variables. We define new operations on zones to handle time.

Tool Architecture. We propose a new architecture to integrate the different verification techniques into a common framework. It is designed as a pipeline built with components that are changed to fit particular experimental configurations and to add new features. The new engine of UPPAAL is based on this architecture. We believe that the architecture is applicable to other verification tools. It is important to develop theories and algorithms for model-checking but the implementation, often left without detail, is as important. This architecture is modular, flexible, and suitable for the *PW-List/storage* structures we developed.

Outline. This thesis focuses on practical issues of model-checking. We start with Chapter 2 that gives a practical start point with a case study. This case study shows the limits of UPPAAL and the need for hierarchical modeling. We propose improvements to UPPAAL in Chapter 3 to tackle the performance issue. In Chapter 4, we give the hierarchical timed automata model with its syntax and semantics to address the modeling issue. As an application, we adapt the model of the bus coupler protocol in the case study to our hierarchical language. In Chapter 5 we present details on the implementation of the hierarchical model, and in particular the abstraction technique. To evaluate the technique we reuse the case study. Finally, Chapter 6 gives the architecture of our tool and shows how different implementation techniques fit together.

Chapter 2

An Industrial Case Study Using UPPAAL

In this chapter we report on an application of the UPPAAL tool to model and debug a commercial fieldbus communication protocol. To our knowledge, this case study was the largest one reported so far where the UPPAAL tool had been applied. This protocol is developed and implemented for safety-critical application, e.g., process control. During its life-cycle this protocol has encountered spurious time-outs and retransmissions. Due to the complexity it has been time and resource consuming to troubleshoot these behaviors.

The company's interest in this project is to improve the development process, reduce the maintenance costs, and to improve the quality of the product with the help of formal methods. The goal of the project is not to verify the correctness of the protocol in any sense of *completeness*, which is basically impossible due to the size and complexity of the system, but to localize the sources of potential imperfect behaviors. The first goal of the analysis is to find bugs and the second goal is to push the limits of UPPAAL and investigate to which extent it is applicable in an industrial context.

We extract the models from the source code of the protocol. The whole protocol involves more than 27000 lines of code and we focused our efforts on 5541 lines of Modula-2¹ that represent the core of the protocol together with a document of 151 pages for the specification of the protocol. The properties and observer automata are derived from the protocol specification. We study the protocol in two parts to tackle its size and complexity. The larger models are built incrementally on top of simplified models studied separately.

¹figures obtained with `wc`.

2.1 The Protocol

2.1.1 Overview

The protocol is a fieldbus communication protocol. Fieldbus is a generic term that refers to digital, bi-directional, serial-bus, communication networks used to link field devices, such as controllers, actuators, and sensors. Figure 2.1 illustrates this concept. In our case, we call these devices “stations” and the protocol is designed for 80 communicating stations.

A station acting as a “master” may initiate a dialog with up to 79 other stations acting as “slaves” in this dialog. The master requests information from a slave that only responds to it, thus the names master and slave. In fact the dialog is established between applications on stations. Each station may have several applications running, acting as masters or slaves.

The protocol has two main layers that are the field interface (FI) to access the protocol from the application, and the bus coupler layer to access the bus from the FI layer. These layers correspond respectively to the transport and data link layers in the OSI protocol standard [Tan96]. In this particular implementation, the protocol is divided in four layers shown in Figure 2.2. The field interface (FI) covers the service data transfer protocol (SDP) and partly the message transfer protocol (MTP). The bus coupler covers mainly the MTP and partly the packet transport protocol (PTP). The lowest layer corresponds to the physical layer and is depicted as “Bus” in the figure. The part of the PTP layer not covered by the bus coupler is taken into account in the study only for its capacity to store packets, i.e., to generate delays.

A typical scenario is as follows: a client application uses the master part of the FI to send requests to another station where a server application will respond through the slave part of the FI. Figure 2.3 shows an overview with four stations over a bus. Each of them has the described layers: the application, the FI, the bus coupler and the bus queue.

The different layers communicate with a specific protocol. We are interested in the bus coupler and in the FI protocols. The bus coupler protocol is the communication between the FI layer and the bus coupler layer. The FI protocol is the communication between the two FI entities on two stations. The low-level protocols for the communication with the bus and between the two bus coupler entities are not studied.

2.1.2 Field Interface: The Transport Layer

This interface provides the services depicted in Figure 2.2 to the application running on the station. From the master point of view, the services are

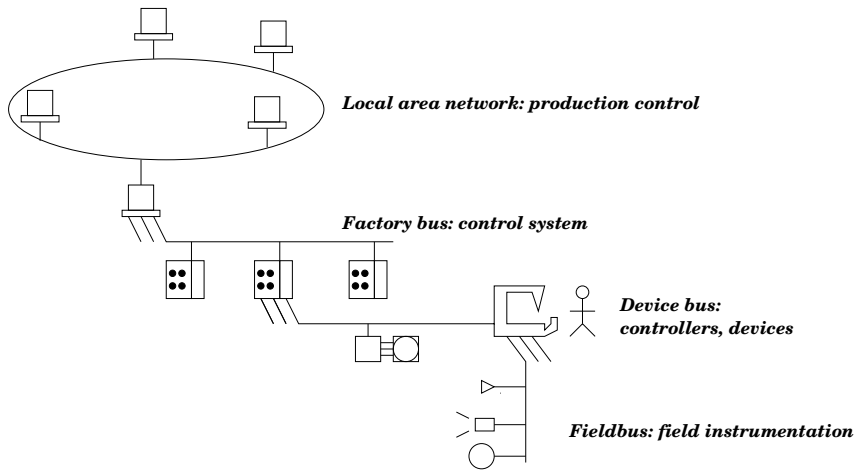


Figure 2.1: Factory communication networks.

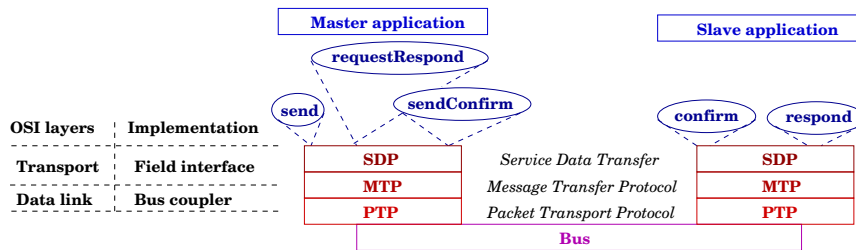


Figure 2.2: Protocol layers.

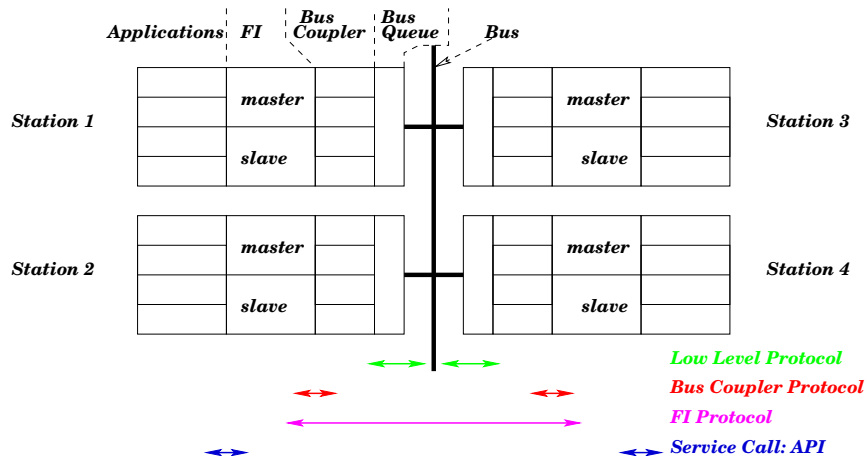


Figure 2.3: An overview of the protocol.

of two types: the *sendConfirm/requestResponse* and the *sendMessage*. The first one waits for an answer coming from the slave. The answer may be simple for a confirm of type yes/no, or more complex for a full response. The slave part has the corresponding services to answer when necessary. Message passing through the protocol, depicted in Figure 2.3, is as follows:

1. The master sends a message to the field interface.
2. The field interface (master side) decomposes the message into packets and sends them to the bus coupler.
3. The bus coupler (master side) sends the packets to the next bus coupler via the bus.
4. The next bus coupler (slave side) sends the packets to the field interface.
5. The field interface (slave side) receives the packets, rebuilds the message and sends it to the application that is waiting on a signal.

In addition, an acknowledgment mechanism ensures at every interface that messages are transmitted correctly.

Figure 2.4 shows the tasks involved in the field interface and the layer organization. The left hand side of the figure depicts a master station communicating with a slave station (right hand side). In a client/server scheme the master acts as the client and the slave as the server. The flow of control of the client application goes to the functions of the field interface API. Other tasks are involved, but not at this level. The same applies for the server.

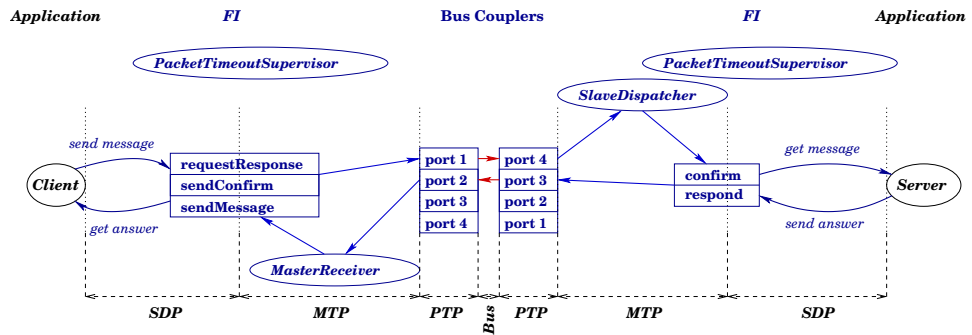


Figure 2.4: Tasks of the FI with respect to the layers. Real tasks are represented as circles, functions as rectangles.

The protocol reserves four ports for the communication. Ports 1 and 2 are reserved for the master part and ports 3 and 4 for the slave parts. The communications port 1 to port 4 and port 3 to port 2 are identical and concern the bus coupler and lower layers. A bus coupler task is associated to each of these ports. The ports are used for communication between the bus coupler entities over the bus.

The field interface has three main parts, both for the master and the slave:

- The *application programming interface* (API). For the master, it is the different send functions *sendMessage*, *requestResponse*, *sendConfirm*, and the *receive* function call that will block until an answer arrives or a time-out occurs. For the slave it is the equivalent *receive* function call and the *sendMessage* function that sends a *confirm* or a *response*².
- The *packet time-out supervisor* that monitors time-outs. The master and the slave are monitored. This is a task that wakes up periodically to decrement and check a time-out counter. When a time-out occurs, the global state variable of the master or the slave can be changed. A time-out may be (re)set by another task, which is viewed as a normal time-out by the supervisor.
- The *receiver task*, that is, *MasterReceiver* for the master and *SlaveDispatcher* for the slave. This task runs separately, listening to the bus coupler, assembling messages. When a message is ready, it puts it into a queue and *signals* a semaphore. As we are not interested in the message itself, only the semaphore is modeled. The different “receive” functions wait for this semaphore.

The master and the slave have a state machine each that describes how they should behave. These correspond to the specification of the behaviors of the master and the slave. The slave has three states and the master five. These states are global states that can be modified by different tasks. Priority and mutual exclusion are used to keep consistency and the study focuses on these global states.

In addition to the control tasks corresponding to the protocol, both the master and the slave have a monitoring process that accepts all transitions between these states. The monitoring process checks valid transitions and

²The difference between these two is minor. The confirm is used for an answer of type yes/no and the response for more detailed information. The other technical differences are out of scope for this study.

detects bad ones. This is a way to check that the implementation follows the specification.

The FI Protocol. The field interface protocol concerns communication between two applications, i.e., one master and one slave. It is an alternating bit protocol at the message level and a sliding window protocol at the packet level. The window varies depending on the success of transmissions. Packets have a sequence number to detect losses when reconstructing messages. A *transparent* bit marks the packets for the receiver to send back an acknowledgment or not: the first and the last packets of a window need acknowledgment and the others do not. The packets that do not require acknowledgement are called *transparent*. Transmission errors are detected at the end of the window. In this case the whole slide is retransmitted and the window size is reduced by one to adapt to transmission errors.

The modeling of the protocol concerns the tasks described in Figure 2.4 that implement the send/receive of this sliding window protocol. Figures 2.5 and 2.6 show the state machines described in the documentation. They specify the protocol at a high level, i.e., retransmissions and sliding windows are not mentioned. The master is used this way: it is normally in the *dormant* state and it goes to the *awaiting first packet* state when a request is transmitted and it waits for the first packet of the answer. It goes to the state *receiving* while receiving the answer and then back to *dormant*. The slave has to answer requests from the master, so it waits in the state *idle (I)* or *idle after error (IAE)* depending on previous errors. If a multiple-packet message is received it goes to the *active (A)* state to receive the whole message and when the last one comes in the slave goes to the state *wait for receive task (WFR)* to wait for the answer from the master. The answer is processed in the state *answer outstanding (AO)* to know if an error occurred or not.

2.1.3 Bus Coupler: The Data Link Layer

The bus coupler is the layer below the field interface. The tasks of the bus coupler run on a different board than the tasks of the FI. The operating systems are different as well. The design is motivated by the fact that this protocol is implemented on different hardwares and the lower-level layers can therefore be changed. This is for flexibility purposes.

The bus coupler corresponds to the data link layer in the ISO standard. It communicates with the FI via an interface implemented as a shared buffer. Each bus coupler entity serves a port that is used by the field interface. The

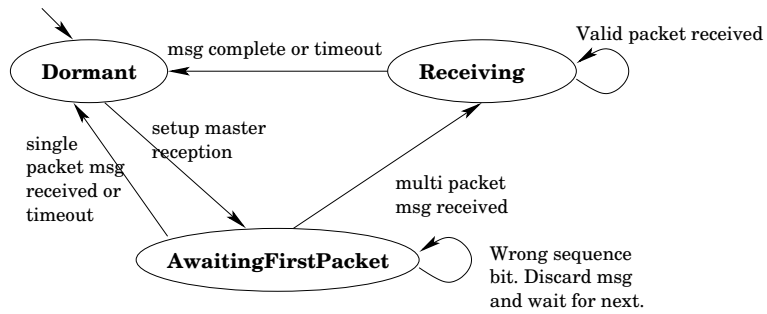


Figure 2.5: Master protocol state machine specification.

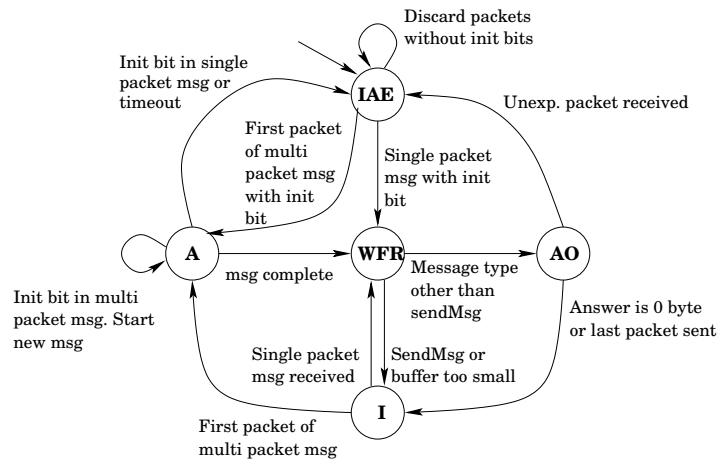


Figure 2.6: Slave protocol state machine specification.

different FI entities communicate with each other through their ports, via the bus coupler that makes the link. These ports are used in the following way:

- A *request* from the master is sent to *port 1*.
- This *request* is received by the slave from *port 4*.
- A response from the slave is sent to *port 3*.
- This response is received by the master from *port 2*.

Ports 1 and 2 are dedicated to the master and ports 3 and 4 to the slave. The ports are used to define the two communication channels port 1 → port 3 and port 4 → port 2. They are identical from the bus coupler point of view. This means that the tasks serving ports 1, respectively port 4, are identical to those serving port 3, respectively port 2. The bus coupler model takes into account only one of these symmetrical communications, that is master sends via port 1 to slave receiving via port 4.

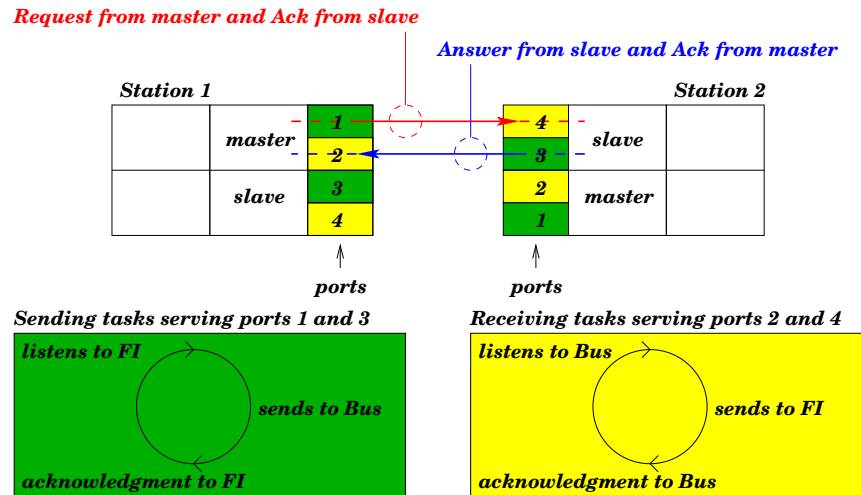


Figure 2.7: Bus coupler communication scheme with the different tasks.

Figure 2.7 illustrates this communication scheme and the tasks that we are focusing on. Figure 2.4 shows the same communication, though from the FI point of view. The tasks serving the ports 1 and 4 are symmetrical in the sense that they perform identical tasks, only the receiver and sender are switched.

The task serving port 1 listens to the FI. When it gets a packet, it forwards it to the bus and acknowledges the FI to notify that the packet was sent. Note the role of the transparent bit at this stage: if the packet is transparent, the acknowledgment will be positive (ACK). If the packet is not transparent the bus coupler has to wait for a real acknowledgment from the other side. Depending on the answer it will acknowledge the FI positively (ACK) or negatively (NACK).

The task serving port 4 listens to the bus. When it gets a packet, it forwards it to the FI and waits for an acknowledgment from the FI. If the packet is transparent, the acknowledgment is always positive. It is similar to the other task.

Communication between the FI and the bus coupler entities at a port is achieved via a buffer. This buffer is separated into fields writable by only one side, but readable by the other one. The synchronization is based on these bits and a signal mechanism that uses interrupts through the two operating systems. The different bits used for synchronization are:

- *Mailbox reserved* to access the buffer.
- *Data read* to notify that data has been read.
- *Data written* to notify that data has been written.
- *Data lost* to return positive or negative acknowledgment.

From the FI, these bits have in practice slightly different names, which is unfortunate. We will only consider the bus coupler view. In addition to these bits, a data field for the useful information is used. The FI side is referred as *cpu* because it is the application side at a high level. The bus coupler is referred as *dev* because it is the device side with low level communication with the bus. In practice the FI requests an interrupt to the OS. The OS notifies the other board, which generates an interrupt to the other OS. The interrupt is handled by an interrupt handler that signals a semaphore. The concerned task is waiting on that semaphore. The model considers only the semaphore.

Figure 2.8 illustrates the configuration of the buffer interface.

The BC Protocol. The bus coupler protocol has two main parts: the communication with the FI through the buffer interface and the communication with the other bus coupler. The communication between bus couplers involves a minimum control of packets with management of re-sending packets and associated acknowledgments. The layer below is used via a simple

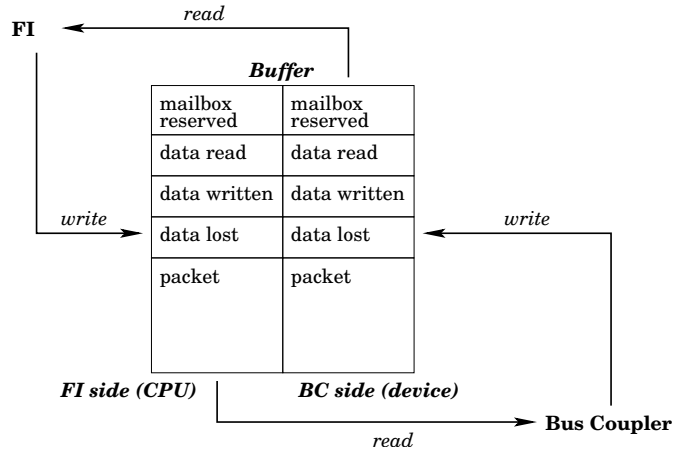


Figure 2.8: The configuration of the interface between the bus coupler and the FI.

API with send and receive primitives. This part of the protocol is known to be robust so we will not treat it in this study. Figure 2.9 shows the protocol we are interested in, namely the communication with the FI. Note that when waiting for a bit, a time-out value is specified and a time-out result can be returned, leading to a reset and another try.

The implementation of the protocol uses signals to notify the reading side when a bit has been written. The mechanism with interrupts and signals is specific to this implementation and uses semaphores on both sides (different boards with their own OS each) and the modeling stresses this feature.

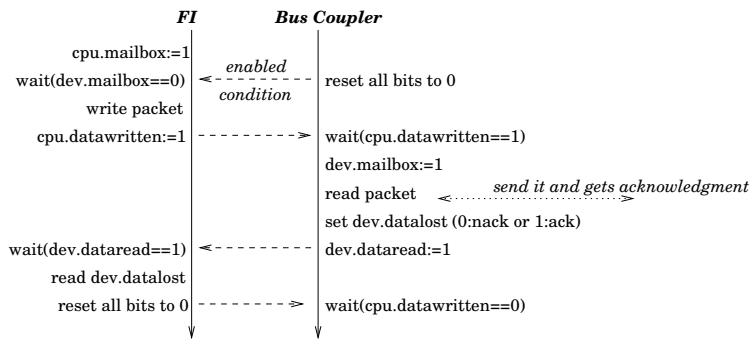


Figure 2.9: Communication protocol from the FI to the bus coupler.

2.2 The Modeling Process

We adopt a top-down approach first to find and understand the relevant components of the system and then a bottom-up approach with progressive abstractions that allows us to build up several abstract models for verification. At the beginning of the project it was not planned to model the bus coupler but it turned out that this was necessary because of its timed behavior. This motivates the following steps we take in the modeling process as illustrated in Figure 2.10:

1. Model the bus coupler, based on the source code. This gives the *detailed bus coupler model*.
2. Simplify the bus coupler model with classical abstraction techniques.
3. Model the FI master and slave sides separately, based on the source code.
4. Derive tests for the master and the slave, combine bus coupler abstraction, master/slave test and the slave/master models.
5. Validate results on the two partial models with the help of the complete master and slave model which contains the bus coupler abstraction.

Step 1 is to construct a detailed model based on the source code of the bus coupler, which is presented in Section 2.2.1. This model is called the “detailed bus coupler model”. Step 2 is to derive an abstract model presented in Section 2.2.1 using abstraction techniques such as hiding and the abstraction features of UPPAAL. Step 3 is to construct detailed models of the master and the slave separately, based on the source code, in Section 2.2.2. Step 4 is to derive test automaton that simulates outputs of these components (input is ignored). The generated messages follow the logic of the protocol and can send negative acknowledgments randomly. These test automata are used against the partial master and slave models. Step 5 is to validate properties that are not satisfied, that is, the counter examples found in the partial models are validated for the detailed model in Section 2.2.2.

Figure 2.11 shows with respect to the overview of Figure 2.3 what is modeled in the two steps of the study. First the bus coupler is modeled (transparent ellipses) with abstractions of lower and upper layers. Second the FI is modeled with abstraction of the previously studied bus coupler and an abstraction of the application using the FI.

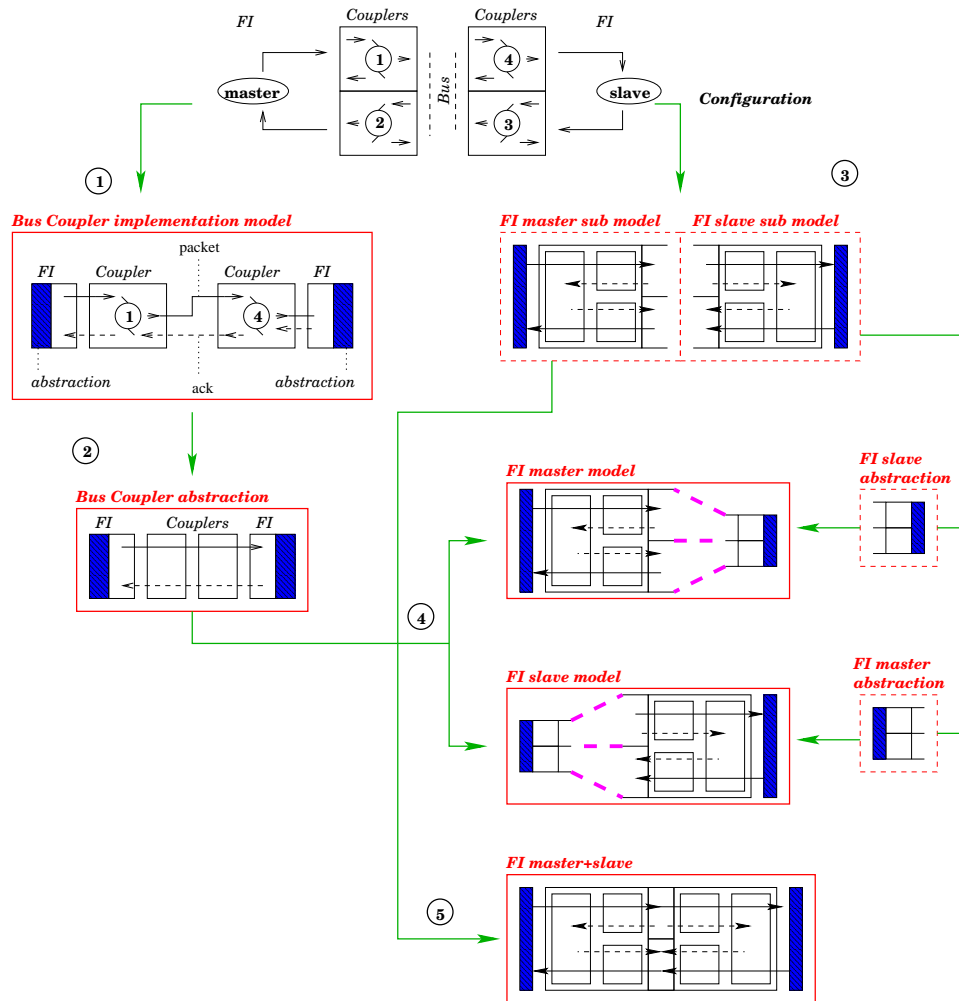


Figure 2.10: Modeling framework.

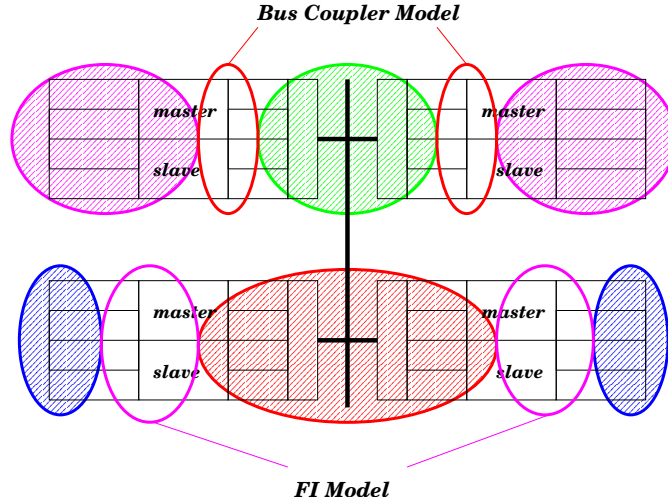


Figure 2.11: The two-steps modeling.

2.2.1 Modeling the Bus Coupler

Detailed Model of the Bus Coupler

This model is the detailed model from which the whole bus coupler study is carried out. It is close to the code and it is possible to map the model to the code. It is of great interest for the industrial partner that traces obtained from the verifier can be mapped to an execution trace of the real code for debugging purposes. To validate a trace is to validate the associated execution trace in the implementation. The engineers validate the traces using the source code or real execution cases.

The UPPAAL model of this part of the implementation consists of 14 automata, 4 clocks and 32 integer variables modeling 2 processes sending, 2 processes receiving, 4 semaphores and 6 functions. We consider here one master application on one station communicating with one slave application on another station. The structure of the model follows the description given in Figure 2.7. The structure of the model is given in Figure 2.12. Circles denote processes and rectangles functions. The functions are modeled as processes in UPPAAL due to size consideration but they behave like functions.

The data content of the packets is irrelevant to the correct behavior of the protocol since it is a layered protocol. The data carried by a layer has no meaning for this layer. There is however an exception, namely the transparent bit. This information bit is checked at the field interface and at

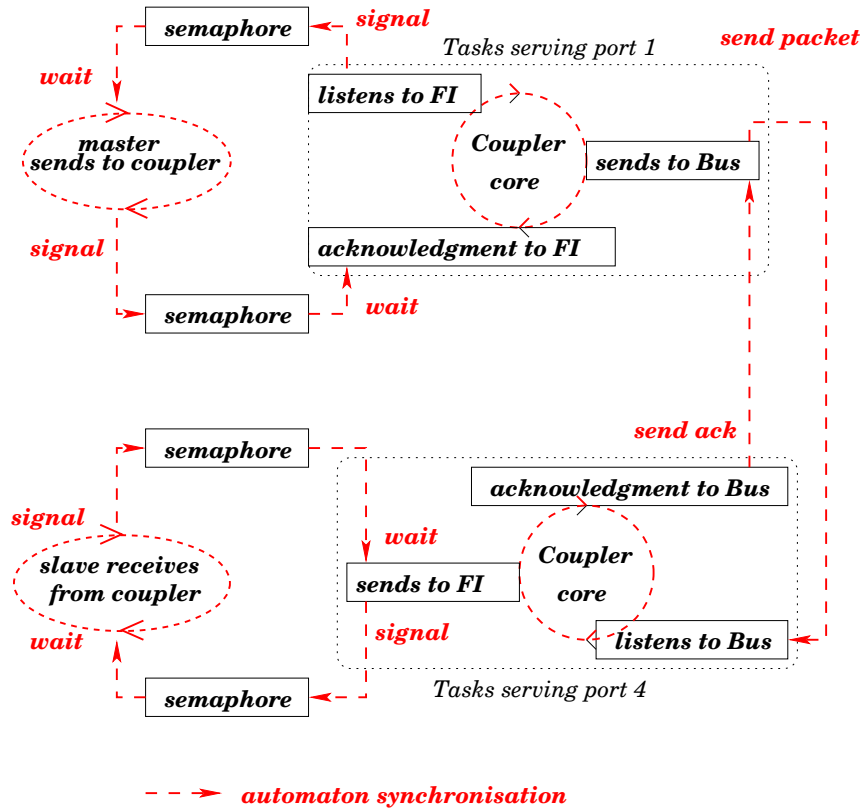


Figure 2.12: Static structures of the implementation model. Tasks are represented as circles and functions/semaphores as rectangles.

the bus coupler layers, even though it is encapsulated as data for the bus coupler packets. This is due to the handling of transparent packets that do not require acknowledgments. In the model, this bit is the data of the packets. It may have the value -1 to mark corrupted data that should not be read, or $0/1$ as valid values.

The stations have several applications, slaves or masters. They all use the same bus coupler layer serving the ports 1-4. A mutual exclusion mechanism is used to gain access to the bus coupler at the field interface level. This is not modeled here. At the bus coupler layer, the incoming packets have a random sequence of $0, 1$ or -1 since accesses may come from several applications in different states. This is modeled by non-deterministic choices.

The same idea is used in the receiver part where the upper layer may accept or reject a packet, which is positive or negative acknowledgment. These acknowledgments are non-deterministic since the FI is abstracted. Note that -1 is not a value part of the protocol but used for verification purposes.

The model consists of two bus couplers, one on the master side and one on the slave side. They are connected in practice via a lower layer to the bus. The communication protocol has another low-level that is partly modeled. The bus is modeled as a lossy channel preserving the ordering of data packets. There is a bus queue on each side of the bus, between the bus coupler and the actual bus. The queue only introduces delay when a message is to be sent. The delay varies if the queue is full or not, which depends on the accesses done by several applications and the traffic. This is therefore random in the model. The model for sending to the bus is a non-deterministic sending within a time window. Time-out may occur. Transmission delays are neglected with respect to the time-out values controlling the protocol.

Modeling Techniques

In addition to the UPPAAL modeling mechanisms, such as committed locations, we apply a statespace bounding technique and an abstraction technique on the detailed model to study different variations and to derive a simpler model without implementation details.

Modeling Mechanisms in UPPAAL. In the modeling process, the models are refined by various modeling mechanisms implemented in UPPAAL including:

- *Committed* states: the state must be left immediately with no delay. Interleaving is allowed only between the committed states. Atomicity

in a sequence of states may be achieved, thus reducing the statespace. The main goal of committed states is to reduce explicitly interleaving.

- *Urgent* states: time is not allowed to progress in such a state, but all interleavings are allowed. It is useful to model race condition and non-determinism.
- *Urgent* transitions: they should be taken whenever the guards become true. It is useful to model progress.
- Shared variables: they are set and read atomically³ by processes running concurrently, thus suppressing the need for explicit mutual exclusion on them.

The goal of this low level abstraction is to control the level of non-determinism of the model.

Error Pruning. We study the detailed model in four different variants. The variations express different levels of assumptions on the program. The idea of the study is to use an *error pruning* technique, which is to detect an error but to stop exploration of the statespace from the detected error state. The idea is to use those error states to bound the statespace by causing a deadlock. It is important to notice that the deadlock in this case is part of the model only, not the protocol, and is used only for studying the protocol. We call this state of states the *error border*. The interpretation of the verification is as follows: if such a state is reached then the property is *partially* verified for a system that does not contain the “error” states. However, we know that an error occurs; so we make another model with less pruning. Thus we have different refinement levels of the model with different levels of assumptions with associated partial results. This is useful to track bugs. The different variations are:

1. Semaphore counters limited to 1, pruning error space.
2. Semaphore counters limited to 2, pruning error space.
3. Semaphore counters limited to 3, full space.
4. Semaphore counters limited to 3, checks added to correct the model.

³on one transition

The limitation on the counter is still kept to bound statespace generation. In the modeling process it was proved at a stage that one semaphore behaved badly, i.e., its counter grew beyond 3. It turned out that the model was not accurate enough and did not filter the synchronization properly. The model was refined and this behavior disappeared. The limit of 3 comes from these experiments where the goal was to include the case where one semaphore is at 3 and the others at 2. The corrected version is a modification of the original model, to patch the implementation. The models are constructed so that the following inclusions between the statespaces hold:

$$\begin{aligned} space_1 &\subseteq space_2 \subseteq space_3 \\ space_{1 \setminus EB} &\subseteq space_{2 \setminus EB} \subseteq space_4 \subseteq space_3, \end{aligned}$$

where $space_{i \setminus EB}$ denotes $space_i$ excluding the error border EB . The experiments in Section 2.3.2 are consistent with these inclusions. These statespaces are comparable because $space_i \subseteq space_j$ comes from the fact that model j is a relaxed version of model i . The error border is meant to detect some states and it cuts the statespace from these states. Removing these detection states and allowing further exploration gives the natural inclusions with $space_{i \setminus EB}$.

The corrections of model 4 concern checking bits when a signal is received. This is actually done in the function *receive* from the bus coupler side, but not on the FI side.

More formally, the error pruning technique used allows us to partition the statespace. Considering one semaphore, the values taken into account in the model form 3 classes: $[0][1][2 \dots \infty]$. The model actually takes the semaphore into account up to 3. This is enough since the values of the variables and the clock regions are the same if there is a loop that makes the counter grow.

Hiding. To debug the protocol logic, we simplify the detailed model (which is based on the source code) using abstraction techniques and the modeling mechanisms listed above, in particular, the notions of *committed* and *urgent* states. The derivation of the abstract models takes away specific parts related to implementation which are the signal implementation and the way to wait on the bits, that is the interrupt handling and the semaphore management to signal a write or to ensure mutual exclusion. The implementation uses the sequence $interruption \rightarrow OS \rightarrow port \rightarrow interruption \rightarrow OS \rightarrow interrupthandler \rightarrow semaphore$. The abstraction allows a direct write/wait/read synchronization mechanism without semaphore, with the

help of urgent synchronizations. The abstraction is independent from the implementation in the sense that this synchronization may be implemented in a different way. Therefore, we hide all the semaphores and the corresponding variables. In addition to this we hide intermediate variables, i.e., result variables. This has the effect to collapse states and reduce the number of processes. Table 2.1 compares the detailed bus coupler and the abstract version.

	Full	Abstract
Variables	32	15
Clocks	4	4
Processes	16	4
Locations ⁴	5.8e11	11520

Table 2.1: Comparison of detailed/abstract models.

Atomicity and Delays. We use different variants of the detailed bus coupler model to study the behavior. The variations of the model are in two dimensions: breaking atomicity of transitions and allowing delay in reading bits. We obtain 5 models:

Model 1 is the simplest model where some transitions are considered to be atomic to study their consequences.

Model 2 relaxes model 1, by removing the atomicity of the transitions performing data-reading.

Model 3 relaxes model 2 by allowing delays when a bit is set to the expected value.

Model 4 also relaxes model 2 but by converting committed states related to data reading and writing to urgent states.

Model 5 relaxes model 4 by allowing delays as in model 3.

The models that are not relaxed do not allow delay when waiting on a bit to be set or reset. This is achieved in UPPAAL by an urgent synchronization that is always enabled, but in order to take the transition, the guard (the bit the component is waiting for) must be true. When relaxing the models, i.e.,

⁴product of locations

enabling delay, this synchronization is removed, allowing time to progress even if the guard is true. This models *eager* or *lazy* synchronization.

Atomicity in a sequence of transitions is modeled by UPPAAL *committed* locations. Locations that do not consume time but still do not have atomic transitions are marked *urgent*.

The models are derived so that the following inclusions hold:
 $space_1 \subseteq space_2 \subseteq space_4$
 $\quad \quad \quad \sqcap \quad \quad \quad \sqcap$. The idea is to derive models 3 and 5 to
 $space_3 \subseteq space_5$

stress delays and models 4 and 5 to stress race conditions. These inclusions hold by construction of the automata, which is, $space_i \subseteq space_j$ because model j relaxes model i by adding delays, or allowing interleavings (commit to urgent) *without* disabling the previous transitions. The basic model is the same, the set of variables is the same but there are more reachable states. The verification results are consistent with these inclusions.

Refining the Models. Refinement of the models is the opposite of abstraction. It gives more details and adds accuracy to the models. It is a trade-off since abstraction is needed to reduce state explosion and refinement is needed for accuracy of the model. In the study both were used side by side and refinement concerning the semaphore mechanism was used when a trace was not judged valid by the engineers.

Abstract Models of the Bus Coupler

The UPPAAL templates for the model 5 are given in Figures 8.1, 8.2, 8.3, and 8.4 (Appendix 8.1). The template automata for the other variants are similar and differ with respect to the described variations.

These automata are close to the protocol description given in Figure 2.9. We recognize the master waiting for `devmbr==0` and `devdataR==1` with possible time-outs, corresponding to `wait(dev.mailbox==0)` and `wait(dev.dataread==1)` on the figure. The random transparent bit of the packets, as explained previously, is modeled by non-deterministic transitions. States marked with `u` are urgent and those marked with `c` are committed.

The master bus coupler is the counter part of the master. It forwards data to the slave bus coupler and waits for an acknowledgment (non-transparent packets). Time-outs are possible here as well. The slave counterpart receives packets from the master bus coupler and it can choose to send back acknowledgments (positive or negative) or not at all. The slave receives from the bus coupler, it acknowledges positively or negatively. The

waiting is similar to the first coupler: wait for `devdataW==1` and then for `devdataW==0`.

We now investigate the relations between the different models.

Reduction. We are interested in properties at an abstract level from which we can infer conclusions on the concrete model. The idea is as follows: the concrete model is a particular implementation of a protocol, the reduced model is the protocol itself where particular implementation details have been abstracted away. We use the term *reduction* that is more appropriate for our purposes.

The inference rule that we are looking for is:

$$\frac{R(M) \models \phi}{M \models \phi},$$

where M is the original model, $R(M)$ the reduced one, ϕ a formula of the form $\exists \diamond p$.

The point is to know if the protocol is correct without considering the implementation, but if something wrong is found then it is wrong in the implementation. This is a debugging process and our approach allows us to localize errors isolated from the implementation.

The reduction relation used here is the simulation: $R \triangleleft M$ the reduced model R can be simulated by the concrete model M , with respect to observable actions. The inference rule comes from this. The definition is:

Definition 6 (Simulation) $P \triangleleft Q$ if for all actions $\alpha \in Action$ whenever $P \xrightarrow{\alpha} P'$, then for some $Q', Q \xrightarrow{\hat{\alpha}} Q'$ and $P' \triangleleft Q'$. \square

We have to define now the reduction relation \mathcal{R}_r that R has to satisfy.

Reduction Relation. We want to establish a relation allowing us to remove a component and hide non observable variables. The hiding part of the problem is a standard hiding operation. It includes collapsing states when transitions between them are not observable with respect to a set of hidden variables. Removing a component is more difficult and in our case is restricted.

The scheme to remove a process is as follows: we have three processes in a model M such that P_1 communicates with P_3 via P_2 . P_2 is the implementation of the communication. The static relation \mathcal{R}_r that $P_1|P_2|P_3$ satisfies is such that:

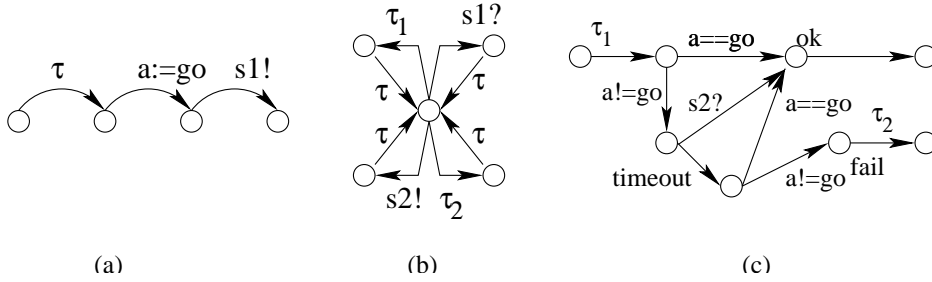


Figure 2.13: Complex automaton patterns.

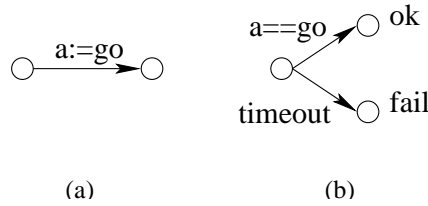


Figure 2.14: Reduced automaton patterns

- P_1 implements the behavior shown in Figure 2.13(a) where τ is an internal transition to prepare the communication, typically a reset ; $a := go$ sets the shared control variable a to the expected value go that P_3 is waiting for ; $s!$ is hand shaking synchronization.
- P_2 implements the behavior shown in Figure 2.13(b) where τ_1 can be related to τ in P_1 to prepare communication, and τ_2 can be a reset asked by P_3 . The τ transitions are internal actions that ensure $P_3 notified \implies P_1 sent$. $s_1?$ synchronizes with P_1 and $s_2!$ with P_3 .
- P_3 implements the behavior shown in Figure 2.13(c) where τ_1 is initialization, typically clock reset, τ_2 post synchronization or cleaning actions with P_2 .

These constraints are abstract and automata that simulate these satisfy \mathcal{R}_r as well and they will clearly yield the expected property as well.

The reduced system is then P'_1 with the reduced sub part Figure 2.14(a) and P'_3 with the reduced sub part 2.14(b). The transition labelled $a == go$ is urgent iff $s_1 \dots s_2$ is urgent. This reduction is in fact a busy waiting with time-out although the implementation is not. This is equivalent to the behavior of the protocol.

The result is then

$$\frac{\mathcal{R}_r(P_1, P_2, P_3) \quad R(P_1)|R(P_2)|R(P_3) \models \phi}{M \models \phi},$$

with ϕ of the form $\exists \diamond p$. We notice that $R(P_2)$ is empty.

Generalization. The generalized property becomes:

$$\frac{R(M) \triangleleft M \quad R(M) \models \exists \diamond p}{M \models \exists \diamond p},$$

which is straightforward but the point is to have the most relevant R as possible to keep interesting properties and our \mathcal{R}_r allows us to reduce M to that interesting model which keeps the logic of the protocol. Other \mathcal{R}'_r should verify $R' \rightsquigarrow M$. Such a relation is used for the FI models.

Relations between the Models. We have two basic models with variations in each. We note I_i the implementation models and R_i the reduced ones. As stated in Section 2.2.1:

$$I_{1 \setminus EB} \subseteq I_{2 \setminus EB} \subseteq I_4 \subseteq I_3$$

in term of space and from Section 2.2.1 we have

$$\begin{array}{ccccc} R_1 & \subseteq & R_2 & \subseteq & R_4 \\ & & \cap & & \cap \\ & & R_3 & \subseteq & R_5. \end{array}$$

The relations are $R_1 \triangleleft R_2 \triangleleft I_1 \triangleleft I_2 \triangleleft I_3$. I_4 is not present because it does not contain the error states.

2.2.2 Modeling the FI

The field interface model follows the protocol description given in Figure 2.4. The master side has a sender process implementing the sliding window with the transparent packet protocol. A receiver process listens to incoming packets and rebuilds messages that are responses from the slave. A time-out supervisor process takes care of the master time-outs. A status process monitors the state transition for verification purposes. The master part has 3 main running processes and one monitor process. The slave part is similar to the master part and has a sender, a receiver (called dispatcher) and a time-out supervisor as main processes. A state process monitors the state transitions here as well. In addition to these 8 main processes, 2 semaphores

for synchronization, 1 semaphore for mutual exclusion, and one forwarder process are used.

The model implements only the service request response since it is the most complete and it contains the others. The sending parts of the model as well as the receiving parts (though the forwarder) contain an abstraction of the bus coupler depicted in Figure 2.15. The model components are

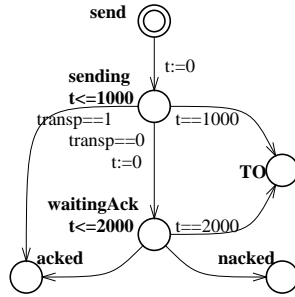


Figure 2.15: Bus coupler abstraction used.

depicted in Figure 2.16. The figure shows the communication between the components. As one guesses, the model is suitable to be cut into one master side and one slave side with a test in the place of the forwarder. The semaphores and the mutex are not depicted in the figure.

Detailed Models of the FI Master

This model is verified with the components from the master side and the slave test. The sender automata are adapted at the bus coupler level because no real receiver is modeled. It is important to notice that these changes are minor and consist in removing the channel synchronizations on the sending transitions. The test slave is depicted in Figure 8.6 (Appendix 8.1). This test models the strict minimum of a slave station. It reacts to acknowledgments from the master, generates messages (replies) with correct sequence. This test automaton is derived from the slave sending function.

The master is monitored with the state monitor process depicted in Figure 8.5 (Appendix 8.1). This monitor has the three central states corresponding to the protocol states. Valid transitions are drawn directly between

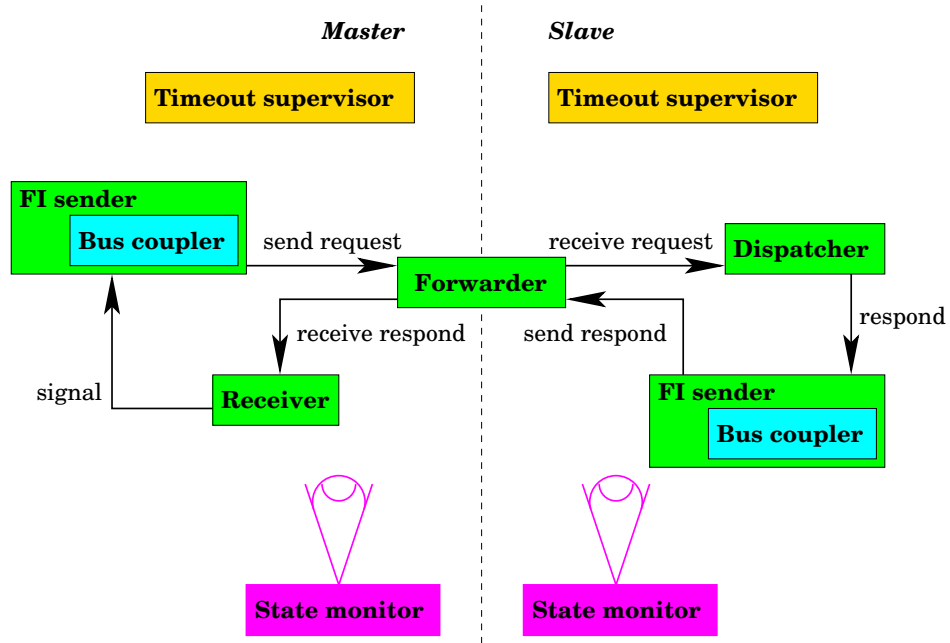


Figure 2.16: FI model overview.

these states as described in the protocol. All the other undefined transitions are present as well, though they go through an error state (committed⁵ in UPPAAL) to detect that the transition is taken. The specification of the master protocol state machine is given in Figure 2.5. We recognize the three central states. The labels are the original ones.

The implementation has a state variable per application. This state variable is shared between all the components of the master that run concurrently. It is important to keep the integrity of this variable by a controlled access, which is done via mutual exclusion and means of preemptive scheduling. Reading or changing this variable is done directly by variable manipulation. The model defers to the state monitor when changing it. When setting the variable to a specific value, a channel synchronization for this specific value is used. The state monitor takes the transition corresponding to this assignment, does the assignment, and depending on defined conditions in the protocol, it will take a legal or an illegal transition. The monitor states have the particularity that the disjunction of all outgoing transitions is always true to avoid artificial deadlocks. Reading is immediate and is

⁵that is left immediately, but detectable for verification

always legal.

Detailed Models of the FI Slave

The slave model is similar to the master model. It has a master test process generating messages as the slave test, though this one may change the init bit. The master test is depicted in Figure 8.7 (Appendix 8.1). This test automaton is derived from the master sending function. The slave is monitored as the master by a state monitor process that works as the master's one. This monitor is depicted in Figure 8.8. The specification of the slave protocol state machine is given in Figure 2.6. These states correspond to the five central states in the monitor automaton.

Validation of the FI Models

The master and the slave models were verified against a test automaton. These models satisfy the simulation relation with the complete model, with respect to the observable events of the master, respectively slave part:

$M_{master} \triangleleft M_{complete}$: the model with the master part is simulated by the complete model.

$M_{slave} \triangleleft M_{complete}$: the model with the slave part is simulated by the complete model.

This holds with respect of the visible events in the master, respectively the slave. The events concerning actual sending of messages are hidden, though not the acknowledgments. This holds because of the way the tests were constructed. These tests are able to produce all the outputs of the hidden component. However, we are interested in the observation equivalence relation. Unfortunately, the models are not equivalent since the tests are less constrained than the complete models. The point is to validate the models so that their behavior is not too general with respect to the detailed models. Experimentally, we rechecked the properties that were violated in the test models, i.e., those that gave counter-examples, to validate the counter-example in the detailed model. This is to ensure that the test behavior does not deviate from the real behavior.

We now strengthen the simulation relation [Mil89] towards the observation equivalence relation with respect to a set of properties, which is, for a subset of real behaviors the observation equivalence relation holds.

Definition 7 (ϕ -Observation equivalence) $P \approx^\phi Q$ if

- (1) $P \models \phi$ and $Q \models \phi$,
- (2) $\forall \alpha \in Act$:
 - (i) whenever $P \xrightarrow{\alpha} P'$ such that $P' \models \phi$, then for some Q' , $Q \xrightarrow{\hat{\alpha}} Q'$ and $P' \approx^{\phi} Q'$ and $Q' \models \phi$,
 - (ii) whenever $Q \xrightarrow{\alpha} Q'$ such that $Q' \models \phi$, then for some P' , $P \xrightarrow{\hat{\alpha}} P'$ and $Q' \approx^{\phi} P'$ and $P' \models \phi$. □

In our case we have:

$$\begin{aligned}
M_{master} &\approx^{\phi_1} M_{complete} \\
M_{slave} &\approx^{\phi_2} M_{complete},
\end{aligned}$$

which is confirmed by experimental results: formally, ϕ characterizes the subset of behaviors of the two models and on these two subsets are equivalent. In practice we have a set of properties verified or violated by both the abstract and the complete models. So experimentally, one could think that the models are equivalent, but by construction the abstract model is not limited in the generation of messages and can generate sequences that the complete model would not generate. Here, ϕ characterizes the valid traces of the abstract model and is not given explicitly. By executing the models and respecting these sequences, the models are equivalent with respect to these sequences.

Figure 2.17 shows the differences between trace equivalence, simulation and ϕ equivalence: X and Y are trace equivalent, they generate the same traces. X can simulate Y but Y can not simulate X: if Y takes the branch A-B-D, X has still the choice of E and C. The sub-tree verifying some ϕ are equivalent.

2.2.3 Related Work

Levi [Lev01] presents an abstraction technique for μ -calculus model-checking. The idea is to obtain an approximate semantics by substituting the domain of computation and its basic operations with an abstract simpler domain and corresponding operations. Fundamentals on Galois connections, the main abstraction formalization, are presented. In our work we consider a reduced set of variables (obtained from hiding).

Alur et al. [TAKB96] tackle the problem of proving that a refined description is a correct implementation of an abstract one. State homomorphism is used as a way of specifying correspondence between two modules. This is

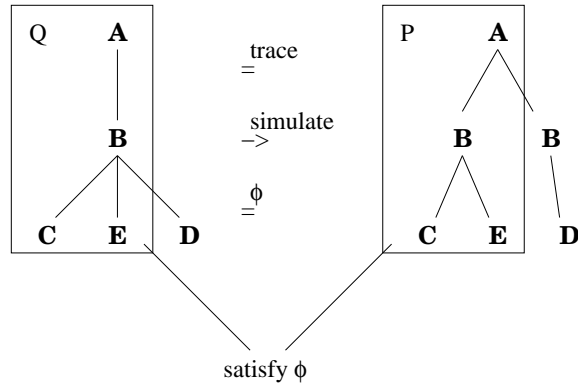


Figure 2.17: Trace equivalence, simulation and ϕ -equivalence.

implemented in the verifier COSPAN. In our case UPPAAL does not provide such capabilities and checking that a given automaton is the abstraction of another one reduces to checking time trace inclusion, which is undecidable in the general case for undeterministic timed automata [AD94]. We check however that the same properties (modulo renaming to appropriate domains) hold. Also we construct abstract models from the more detailed ones, which is opposite to refinement.

Tripakis and Yovine [TY01] present a method to verify dense-time systems modeled as timed automata with untimed verification techniques. Exact time delays are abstracted away and this abstraction is formalized under the concept of timed-abtracting bisimulation. In our example time is important and trying to abstract time requires to basically cross-product the automata with a zone transition automaton, which is done on-the-fly by the model-checker. Chapter 5 gives more details on abstraction. We focus more on practical issues here.

2.3 The Verification Process

In this section we present the correctness properties checked and their results. They are either reachability properties of the form $\exists \diamond \phi$ or invariants of the form $\forall \square \phi$. The predicate ϕ is defined over states, variables, and time.

2.3.1 The Classes of Properties

Finding the properties to check is a problem in itself. Furthermore, the completeness of a verification depends on the completeness of the set of properties we check. This is the limitation of our verification as we have to define these properties. For the bus coupler models, 82 properties for the detailed models (and 35 for the reduced models) are checked. These properties are classified into 4 classes:

- 6 correctness properties for both the detailed and the reduced models related to the logics of the protocol
- 25 functional properties for the detailed models and 5 for the reduced ones, related to the synchronization of the components. Violating these properties could induce bad/wrong behavior. The properties of the implementation models are classified as follows: 8 related to the implemented semaphores, 10 to detection of possibly bad states belonging to the *error border* and 7 related to precedence between states. The abstract models properties were based only on precedence.
- 19 behavior properties for the detailed models and 5 for the reduced ones, which are intuitively believed to hold with respect to the protocol. This is expected behavior which has only performance impact.
- 32 validation properties for the detailed models and 19 for the reduced ones, related to the model itself. The protocol works in practice and the model must work the same. A more complex model requires more validation hence the difference in the number of properties.

Concerning the field interface, 98 properties are checked. The classification is different: we have correctness properties and validation properties. There is no equivalent of the functional and behavior properties as defined for the bus coupler. However, the field interface has different priority tasks that are modeled. We check that this modeling is correct as well as the consistency between the state monitors and the state of the system:

- 32 correctness properties based on the state monitors.
- 50 simple validation properties related to the model itself to check that it works. These properties reflect the model of the implementation.
- 16 consistency properties related to the decoration of the model, i.e., parts of the models that are not originally part of the implementation.

This checks that the priority model holds, as well as the consistency of the state monitors.

We do not intend to present all the properties but rather the important ones. Verification was conducted on a Sun Ultra-SPARC-II 400MHz equipped with 4GB of physical memory. UPPAAL version is 3.0.39⁶. Options were reuse statespace, breadth-first search, active clock reduction and no trace generation to save memory⁷.

2.3.2 The Bus Coupler

Detailed Models

The resources consumed to verify the properties are given in Table 2.2. These figures show the size of the complete statespace because the properties need complete search. They are consistent with the inclusions $space_1 \setminus EB \subseteq space_2 \setminus EB \subseteq space_4 \subseteq space_3$ that were given in Section 2.2.1. Figure 2.18 illustrates the space inclusions.

Model	Size	Verification
1	129 MB	12:31 min
2	136 MB	14:41 min
3	149 MB	14:40 min
4	140 MB	11:11 min

Table 2.2: Resources used for verification.

The correctness properties are:

- 1: $A[]$ `FItoCoupler_1P1.written imply fiTrans1!=-1`
- 2: $A[]$ `(CouplerFromFI_1P1.done and resultC11==0) imply bcTrans11!=-1`
- 3: $A[]$ `CouplerToBus_1P1.sent1 imply bcTrans11!=-1`
- 4: $A[]$ `CouplerFromBus_2P4.received imply bcTrans24!=-1`
- 5: $A[]$ `CouplerToFI_2P4.step2w0 imply bcTrans24!=-1`
- 6: $A[]$ `FIFromCoupler_2P4.dataTaken imply fiTrans2!=-1`

where $A[]$ stands for $\forall \square$. They concern the transparent bit (data modeled) which should not be written/read when not valid (-1) by the FI (fiTrans) and the Bus Coupler (bcTrans). The full state model 3 does not satisfy property 6. More models used to violate more properties here but that was

⁶distributed on the web

⁷exact options given to verifyta are: -CDSTaqs

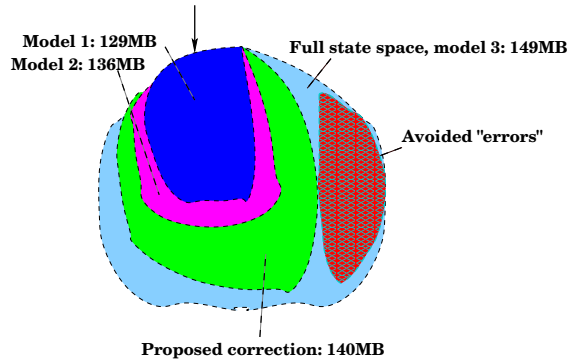


Figure 2.18: Overview of the statespaces and inclusions.

due to the granularity of the models themselves. Furthermore, the trace of property 6 exploits an approximation of the model and is not judged valid by the engineers. Although the model fails to show a real error here, it shows that the components may reach a state where they are unsynchronized.

4 of the properties concerning semaphores are:

```

43: A[] not SemFIToCoupler24.signalNotTaken
44: A[] not SemCouplertoFI24.signalNotTaken
45: A[] not SemFIToCoupler11.signalNotTaken
46: A[] not SemCouplertoFI11.signalNotTaken

```

They mean that whenever a signal is sent, the previous one should have been accepted otherwise “it has not been taken”. If there is a wait on that signal, it will not make much sense since the semaphore stores previous signals. These properties are not verified for model 1 but hold for models 2, 3, and 4. The counters may reach 2 but not 3, as pointed out in Section 2.2.1.

An interesting functional property (because of its result) is:

```

57: A[] not FIToCoupler_1P1.OKwhenMBR

```

It states that the FI side should not be in a success state after the first synchronization step if the mailbox receive flag is on (it should be off). This property is not satisfied by all the models. The interesting point here is that it is acknowledged by the engineers but it is not considered important because in this precise case, the communication concerns acknowledgment and not data. This is to be documented in the implementation. Other functional properties are similar to this one.

2 precedence properties addressing synchronization are:

75: A[] not (FIToCoupler_1P1.testOK and (CouplerFromFI_1P1.step1w0 or CouplerFromFI_1P1.step2))
 76: A[] not (CouplerToFI_2P4.endWait2 and (FIFromCoupler_2P4.waited or FIFromCoupler_2P4.wait0))

Only model 3 does not satisfy 76. Property 75 checks that one part should not be at the end of sending a packet with success while the other side still waits for acknowledgment. Property 76 checks that one part should not be sending an acknowledgment while the other part is going to begin to send a packet.

4 behavior properties are:

10: A[] not (FIToCoupler_1P1.done and resultV1!=0 and bcTrans11==1)
 16: A[] not (Coupler_1P1.sentT0 and bcTrans11==1)
 33: A[] not (Coupler_2P4.acking and saveTrans24==1)
 78: A[] (FIToCoupler_1P1.testOK and fiTrans1==1) imply devdatalost11==0

Property 10 states that sending a transparent packet should never fail and this is false for all models. Property 16 states that timeout should not occur on transparent packet which is true for all models. Property 33 states that acknowledgment is not sent after transparent packets which is true. Property 78 states that the coupler “lies” properly to the FI when a transparent packet is sent, which is true for all models.

Validity properties are simple reachability properties to check that the model does what it should do. One of these is: packets are transmitted successfully.

2: E<> FIFromCoupler_2P4.done and resultV2==0

In conclusion, the protocol is subject to desynchronization, though it is not fatal. The origin comes from race conditions when reading from and writing to the buffer.

Abstract Models

The resources consumed to verify the properties are given in Table 2.3. They are consistent with the inclusions

$$\begin{aligned} space_1 &\subseteq space_2 \subseteq space_4 \subseteq space_5 \\ space_2 &\subseteq space_3 \subseteq space_5 \end{aligned}$$

Model	Size	Time
1	3.8 MB	8 sec
2	4.1 MB	9 sec
4	5.0 MB	10 sec
3	11 MB	32 sec
5	14 MB	37 sec

Table 2.3: Resources used for verification.

referred in Section 2.2.1. 35 properties are verified.

Due to the way we construct the models, we believe that $space_2 = space_3 \cap space_4$ though we did not prove it. Experiments confirm this by exhibiting this common behavior with a number of properties. These different models are interesting when properties are verified in one model (case for $space_1$) but not in others ($space_2$, thus $space_3$ and $space_4$). This is used to pinpoint behavior differences and see what the protocol is sensitive to.

The correctness properties are:

- 1: `A[] master.waitDataR imply fitrans1!=-1`
- 2: `A[] coupler1P1.sending imply bctrans11!=-1`
- 3: `A[] coupler2P4.getMsg imply store24!=-1`
- 4: `A[] slave.read imply fitrans2!=-1`
- 5: `A[] master.OK imply devdatalost11!=-1`
- 6: `A[] coupler2P4.readnottrans imply cpudatalost24!=-1`

These are of the same type as the implementation properties. They state that wrong data should not be read because they are received too early or too late. We add here the explicit test on the acknowledgment answer from the coupler or the FI-slave with `dev/cpudatalost`. This is present in the implementation as well.

Properties 1 and 5 are satisfied by all the models. Properties 2 and 3 are satisfied only when no delay is allowed, which is the case for the models 1, 2 and 4. When delay is allowed, a timeout may occur concurrently leading to an unwanted change that leads to a race condition. To interpret this as realistic or not, the hardware and runtime environment have to be taken into consideration. In our context of non-preemptive multitasking on the Bus Coupler side, this situation is possible if the coupler blocks while sending.

Property 4 is satisfied only for the first model. This property is sensitive to race condition. Property 6 is satisfied only for the 3 first models. Models 4 and 5 introduce new interleavings and a race condition is enabled by changing *commit* states to *urgent* states.

The functional properties are:

```
31: A[] not (coupler2P4.readnottrans and slave.read)
32: A[] not (coupler2P4.readtrans and slave.read)
33: A[] not (master.OK and coupler1P1.sending)
34: A[] not (coupler2P4.sending and cpumbr24==1 and slave.read)
35: A[] not (master.waitMBR and devmbr11==1 and coupler1P1.sending
and ck11==0)
```

They concern desynchronization, when a component is one cycle late on the other. Properties 31 and 32 state that the coupler should not be in a state ready to read the acknowledgment from the slave while this one has not written it and is about to do it: models 4 and 5 violate these properties. This result is similar to property 6.

Property 33 states that the master should not have read the acknowledgment from the coupler when this one has not written it yet. Model 5 does not satisfy this one, which means that this property is related to delay *and* race condition.

Property 34 states that the coupler should not be in a state waiting for the mailbox being available in order to write data while the slave has read data and not reserved yet the mailbox. This is satisfied by all the models.

Property 35 states that the coupler should not be in a state when it has just reserved the mailbox and read data from the master though this one is waiting for the mailbox to be freed in order to write data. Models 3 and 5 do not satisfy this one. This property is sensitive to delays.

The behavior properties are:

```
26: A[] not (master.timedout2 and fitrans1==1)
27: A[] not (coupler1P1.waitanswer and bctrans11==1)
28: A[] not (coupler1P1.acking and ck11>0 and bctrans11==1)
29: A[] not (slave.timedout2 and fitrans2==1)
30: A[] not (coupler2P4.timedout2 and bctrans24==1)
```

These properties are related to the nature of the packets: if they are transparent, timeout should not occur. This is an expected behavior, but not a critical property. Properties 26 and 30 are not satisfied, which comes from a possible delay from the bus queue. The acknowledgment is sent to the master *after* having sent a message on the bus. If the queue is full and introduces delay, the transparent packet is delayed.

Property 27 is satisfied, which is straightforward with respect to the automaton. Property 28 is not satisfied by models 3 and 5, which comes directly from the possible delays while reading bits. Property 29 is satisfied.

The conclusion on the abstract model is that the protocol is implementable since the first model is valid. However, the implementation has to avoid some possible race conditions as well as some delays in order to work.

The bus coupler part does not show any major flaw. As it is a rather low level implementation, the models are sensitive to the underlying modeling assumptions. However, the models proved to be useful with their identified limits. The consequence is a series of improvement requests on the implementation, i.e., the product will be improved as a result of the study.

2.3.3 The Field Interface

The resources consumed to verify the properties are given in Table 2.4.

Model	Size	Verification
Master, all properties	817M	1h 32 min 4 sec
Slave, all properties	1.46G	7h 34 min 22 sec
Complete, without satisfied safety properties	2.42G	6h 1 min
Complete, all properties, bit state hashing	5.3M	2h 12 min

Table 2.4: Resources used for the verification.

The method used was to validate the complete model with the simulator, check the master sub-model (against a slave test) with the master properties, similarly for the slave, and check the complete model with the validation properties and the violated safety properties. Furthermore, the complete model is checked with the full set of properties, but with bit state hashing.

Master Model

The validation properties are of two kinds: reachability properties to check the functionality of the models. 2 of them are:

- 2: E<> Master_Send.sendOK
- 4: E<> Master_Send.NackTO

They check for the success of a sending and a time-out. These check the master. The other type is for the slave test process, there is only one, to check that a complete message is sent:

- 31: E<> Slave.send and Slave.size==0

All these properties are satisfied.

The consistency properties are of type $A[] \phi$ to check priority handling (one property) in the model and the consistency of the state monitor (3 properties, one for each state):

```
10: A[] (Master_Send.idle or Master_Send.sending1 or
Master_Send.waiting1 or Master_Send.waitReceive or
Master_Send.sendingx or Master_Send.waitingx) imply MP4==0
20: A[] ((Master_Status.D or Master_Status.DbadR) imply
(MStatus==Dormant)) and ((MStatus==Dormant) imply Master_Status.D or
Master_Status.DbadR))
```

All these properties are satisfied.

The safety properties of the state monitors check for bad transitions. In UPPAAL, it is not possible to check for a transition. The trick is to use a committed state just for the detection. There are 9 of these. Two of them are:

```
14: A[] not Master_Status.RbadAFP
18: A[] not Master_Status.DbadR
```

These two properties are not satisfied. We show that unknown transitions may occur. Some of them were acknowledged by the engineers and they are investigating them.

Slave Model

The model-checking of the slave model is similar to the master model. The validation properties are of two kinds, reachability and safety. Two reachability properties are:

```
2: E<> Slave_Send.sent1
3: E<> Slave_Send.NackT0
```

They check for sending successfully one packet and getting one time-out. One property to test The master:

```
56: E<> Master.send and Master.size==0
```

These reachability properties are satisfied. There are 5 safety properties concerning the state monitor (one for each state) that check for consistency of the model. One of them is:

44: $A[] ((\text{Slave_Status.AO or Slave_Status.AObadAO or Slave_Status.AObadWFR or Slave_Status.AObadA}) \text{ imply } ((\text{SStatus==AnswOuts})) \text{ and } ((\text{SStatus==AnswOuts}) \text{ imply } (\text{Slave_Status.AO or Slave_Status.AObadAO or Slave_Status.AObadWFR or Slave_Status.AObadA})))$

The consistency property concerning the priority is:

9: $A[] (\text{Slave_Send.respond or Slave_Send.send0 or Slave_Send.sending1 of Slave_Send.waiting1 or Slave_Send.idle or Slave_send.waitingMutex or Slave_Send.toI or Slave_Send.sendingx or Slave_Send.waitingx}) \text{ imply } \text{SP4==0}$

All these properties are satisfied.

There are 26 safety properties concerning the state monitor. Two of them are:

18: $A[] \text{ not Slave_Status.IAEbadI}$

26: $A[] \text{ not Slave_Status.AbadI}$

These two properties are not satisfied.

For the slave model we show here as well that undocumented transitions may occur. Some of them were acknowledged and are under investigation.

Complete Model

The bit state hashing experiment on the complete model proved to be unsuccessful. It is an over-approximation method where positive results are not reliable, whereas negative results are. Unfortunately, the verification gave too many false positive answers.

The full verification of the properties was successful. We succeeded in proving that properties violated on the partial models were still violated on the detailed model. The detailed model satisfied also all the simple validation properties. For our model and our set of properties the relation given in Section 2.2.2 holds.

The conclusion for the FI part is that we identified unknown behaviors on both the slave and the master parts. The models are accurate enough to reproduce real code execution. Engineers are analyzing these traces to improve the code.

2.4 Conclusion

This study is regarded as a success from the academic and the industrial point of view. For the academic side, we succeeded in modeling and analyzing a real product, not a toy example, and we suggested a number of improvements to the code and the documentation. For the industrial side, they were pleased to use formal methods, in particular the graphical interface of UPPAAL was intuitive.

The modeling method was to divide the protocol by its layers and to model them separately. We applied abstraction techniques to model the parts of the higher layer using the lower layer. The modeling process showed the limits of the input language of UPPAAL, demonstrating the lack of support for structured models. In particular, some parts of the models were used as sub-components. Furthermore, we pushed the limits of the verification engine, making this case study a suitable benchmark for new algorithms due to its size, timing behavior, and real-life nature.

Chapter 3

Improvements on the Current UPPAAL

Model-checking suffers from the so-called *state explosion* problem, which is, in the worst case the needs of resources for verification grow exponentially with the model to verify. For timed systems, the problem is more serious. It is known that the model-checking problem for timed systems is PSPACE-complete [ACD90, ACD93]. The case study of Chapter 2 illustrates the need for resources, in particular for memory.

When it comes to implementation, one has to consider that the memory system of modern computers is the main bottleneck. While the speed of processors increases at a fast pace, the speed of the memory system lags far behind. Although model-checkers benefit from faster computers, their performance is seriously impaired by the memory system because of the large statespace that needs to be explored, and thus stored in memory.

In this chapter we propose two improvements to the current implementation of UPPAAL. The first is the unification of the two main structures used in the reachability algorithm: the passed and the waiting lists. This unification reduces by half the number of zone inclusion checks. The second concerns memory management where we eliminate all duplicate data by using sharing. We present our implementation with experimental results and conclude with a survey on other techniques related to UPPAAL.

3.1 Reachability Algorithm

In this section we propose two techniques to improve the reachability algorithm implemented in UPPAAL. The first is the unification of the two main

structures in the algorithm, namely the *waiting* and the *passed* list, to the *PW-List* structure. The second technique is the use of sharing.

3.1.1 PW-List

The Passed and Waiting Lists

The reachability algorithm used in UPPAAL is shown in Figure 3.1. The states we are considering are symbolic states of the form (l, Z) where l is the location vector and Z a zone [Ben02]. For simplicity we omit the data variables.

```

waiting =  $\{(l_0, Z_0 \wedge I(l_0))\}$ 
passed =  $\emptyset$ 
while waiting  $\neq \emptyset$  do
   $(l, Z)$  = select state from waiting
  waiting = waiting  $\setminus \{(l, Z)\}$ 
  if testProperty( $l, Z$ ) then return true
  if  $\forall (l, Y) \in \textit{passed} : Z \not\subseteq Y$  then
    passed = passed  $\cup \{(l, Z)\}$ 
     $\forall (l', Z') : (l, Z) \rightarrow (l', Z')$  do
      if  $\forall (l', Y') \in \textit{waiting} : Z' \not\subseteq Y'$  then
        waiting = waiting  $\cup \{(l', Z')\}$ 
      endif
    done
  endif
done
return false

```

Figure 3.1: The reachability algorithm for timed automaton. The function *testProperty* evaluates the state property that is being checked for satisfiability. The while loop is referred to as the exploration loop.

The reachability algorithm uses two structures, namely the *passed* list and the *waiting* list. The *passed* list records all the explored states and the *waiting* list keeps track of the states to be explored. The waiting list is initialized with the initial state, where $I(l_0)$ represents its invariant constraint. The algorithm is a loop where a state (l, Z) is selected from the waiting list, its successor states computed and inserted in the waiting list. The successors are computed only if the state was not visited before, i.e., present in the passed list. The purpose of the passed list is to ensure termination and

to avoid exploring a given state twice.

Implementation of the Passed and Waiting Lists

One crucial performance optimization is the state inclusion checking. When adding a new state to a set of states, we check that the new state is not included in the states of the set. Furthermore, the states that are included in the new state are removed. The inclusion checking is computed between zones of states with the same discrete part (location vector and data variables). A simple implementation would implement a hash table [CLRS01] only for the passed list to access the state quickly and then perform an inclusion check. The waiting list does not need the check in principle. However, this solution leads to an unnecessary large waiting list. This implementation is refined in UPPAAL where the waiting list is also equipped with a hash table and inclusion check, which keeps the waiting list small. This solution is still not satisfactory because the separated structures contain unnecessary states. There may be states in the waiting list that are included in the passed list, which is a waste of resources since these states will be removed. Conversely, there may be states in the passed list that are included in the waiting list. Such states are guaranteed to be removed later, which also wastes memory. Furthermore, there are two inclusion checks per generated state.

The unified PW-List structure allows the removal of the redundant states and eliminates one inclusion check per state. All states in this structure are considered explored, though some of them are not yet explored. These waiting states will be explored if they are not replaced by larger states. The benefit of this is to have one check for all states, which allows to get rid of redundant states and detect state inclusion earlier.

Comparison with Discrete States

The passed/waiting list unification has been applied to Petri Nets [CN97] for the purpose of distributed model-checking and it concerned discrete states only. Figure 3.2 illustrates a unified structure for discrete states. When exploring the states with a breadth first search order (a), the states are put in a linear list with a moving pointer that marks the waiting states at the end of the list. When generating a new state, we need to test if it is present or not in the list and if it is not, we just put it at the end of the list. When a state is explored, the pointer is moved to the next state to be explored. We skip the details of the hash table necessary to find states

quickly. The simplicity for the discrete case comes from the fact that once a state is put in the list, it will not be removed. Symbolic states do not have this linear property: states in the passed or the waiting part of the list may be removed and we have to compute an inclusion check instead of a simple equality check. Furthermore, the search order is changed because states are replaced. A unified structure for symbolic states has to take into account these constraints.

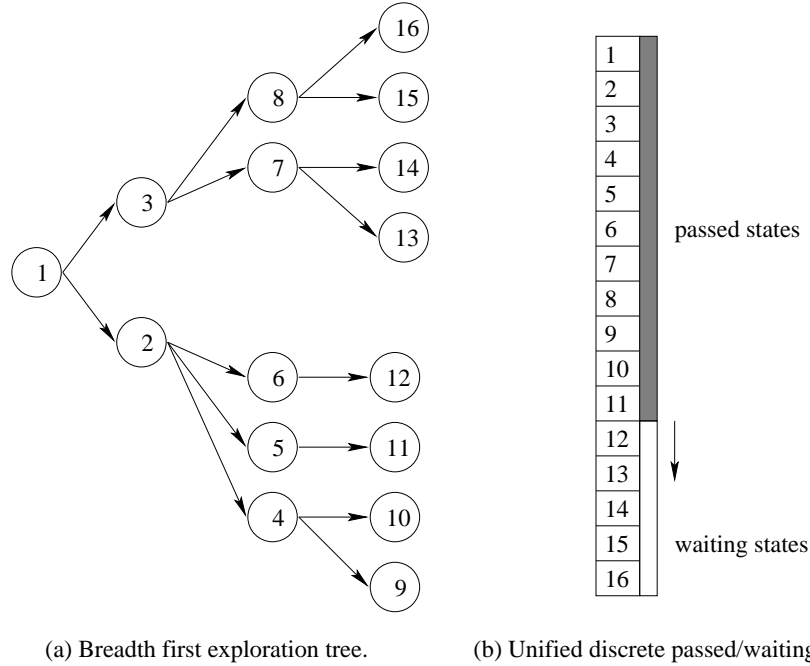


Figure 3.2: Unification of passed and waiting lists for discrete states.

Unification of the Passed and Waiting Lists

We use for the unified structure the notation (P, W) . P denotes the list of all states that are considered as passed and W marks a subset of these as waiting states. A PW-List is described as a pair $(P, W) \in 2^S \times 2^S$, where S is the set of symbolic states, $W \subseteq P$, and the two functions $put : 2^S \times 2^S \times S \rightarrow 2^S \times 2^S$ and $get : 2^S \times 2^S \rightarrow 2^S \times 2^S \times S$, such that:

- $get(P, W) = (P, W \setminus \{(l, Z)\}, (l, Z))$ for some $(l, Z) \in W$.
- $put(P, W, (l, Z)) =$

$$\begin{cases} (P \setminus I, W \cup \{(l, Z)\}) & \text{if } \forall (l, Y) \in P : Z \not\subseteq Y \\ (P, W) & \text{otherwise,} \end{cases}$$

where $I = \{(l, Y) \in P \mid Y \subset Z\}$.

The *get* function removes states from W and leaves them in P . The *put* function removes the states of P that are included in the new state (the set I) and add this new state to W . Removing states from P implicitly removes them from W too because $W \subseteq P$. Similarly, states added to W are also added to P .

Figure 3.3 illustrates the idea behind the PW-List structure. All states $\langle s1 \rangle \dots \langle s5 \rangle$ of the statespace are considered passed (in P) and some of them are marked waiting. From the structure point of view, getting a state from W corresponds to clearing the w flag. From a set point of view, this corresponds to shrinking the sub-set W . This structure allows us to simplify the reachability algorithm to the one in Figure 3.4.

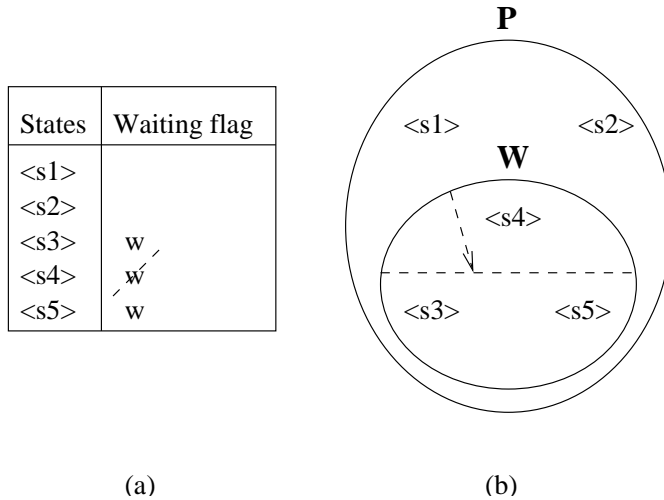


Figure 3.3: PW-List viewed as a structure or a set. Getting $\langle s3 \rangle$ from W results in clearing the flag w (a) or shrinking the set W (b).

The simplification comes from the observation that the states obtained from the *get* function are guaranteed not to have been previously explored. There is no need for zone inclusion checking in this step because the *put* function ensures that states of W are not included in other states of P . Furthermore, it is possible, as we show in Section 3.2 to represent the PW-List as one unique structure. There is no duplicate state and only one hash table lookup is needed. We also note that it was possible previously to have states

```

(P, W) = {(l0, Z0 ∧ I(l0)), (l0, Z0 ∧ I(l0))}
while W ≠ ∅ do
  (P, W, (l, Z)) = get(P, W)
  if testProperty(l, Z) then return true
  ∀(l', Z') : (l, Z) → (l', Z') do
    (P, W) = put(P, W, (l', Z'))  done
done
return false

```

Figure 3.4: Reachability algorithm using the unified PW-List.

in the waiting list though they were also in the passed list. This behavior is suppressed now because states marked as waiting are considered passed when inclusion checking is used. This allows us to remove states in the waiting or passed list earlier than in the previous case. It is specially important for the states of the passed list since further inclusions will consider the latest states only, whereas the old algorithm considers inclusion checking with states in the passed list that can be outdated by other states in the waiting list.

From an implementation point of view, the PW-List is a buffer that has the property to automatically eliminate states that are subsets of added states, or to eliminate the added states if they are subsets themselves. There is a need for a queue to keep track of the waiting states but this is straightforward and no copy is necessary. This structure, considered as a buffer, fits naturally into the pipeline architecture of Chapter 6.

3.1.2 Sharing

Data sharing is a technique used to reduce memory usage by sharing common sub-structures. It is the basis of BDDs efficiency [Bry86]. In this section we apply this technique to our model-checker. We first conduct experiments to evaluate data redundancy in UPPAAL. These results show that most of the data can be shared. Therefore, we implement two levels of data sharing to eliminate data redundancy.

Evaluating Sharing

When computing successor states during the statespace exploration it is often the case that some part of the state will be preserved. For extended timed automata (XTA) a state is noted as (\bar{l}, u, v) as defined in Section 1.2.2

of the introduction. A successor computation will do for example:

$$\begin{aligned}
 (\bar{l}, u, v) &\rightarrow (\bar{l}_1, u, v) \\
 &\rightarrow (\bar{l}_2, u_2, v) \\
 &\rightarrow (\bar{l}, u, v_3) \\
 &\rightarrow (\bar{l}_4, u, v_4).
 \end{aligned}$$

We observe that sub-parts of the original state are often preserved in the successor states. This also holds for the hierarchical timed automata of Chapter 4. If we consider the structure representing the statespace, i.e., the PW-List, it contains unique states (\bar{l}, u, v) but these states can share their sub-components.

We first investigate how much data can be shared by instrumenting UPPAAL. State information is printed out when states are stored after the inclusion check and it is analyzed by a Perl script. Table 3.1 shows the amount of unique data found in the states. States are of the form (\bar{l}, u, v) and the table shows how much of all the \bar{l} , u , and v data parts are unique among all the states. The models are: (1) Engine, a gear box controller [LPY01], (2) Audio, an audio/video protocol [HSSL97], (3) Dacapo, a TDMA protocol start-up mechanism [LP97], (4) Fischer4, the Fischer’s mutual exclusion protocol [AL92, KLL⁺96] for 4 processes, and (5) BC, an early version of the bus coupler model of Chapter 2 that generates more states than the version used for verification in the case study.

Model	Unique locations	Unique variables	Unique DBMs
Engine	71.7%	21.6%	8.6%
Audio	52.7%	25.2%	17.2%
Dacapo	4.3%	26.4%	12.7%
Fischer4	9.9%	0.6%	64.4%
BC	7.2%	8.7%	1.3%

Table 3.1: Results from instrumented UPPAAL: smaller figures correspond to more copies.

It appears that most of the stored data is redundant for all the examples. The engine example has states that depend heavily on the location combinations, which explains the 71.7%. For the Dacapo example, 4.3% of unique locations means that all location vectors are copied 23 times in average. The Fischer example is known to behave particularly badly with respect to timing constraints, which explains its odd results for the DBMs. In all the cases, data can be shared to gain in memory consumption. This

is an important result that shows the potential of sharing for UPPAAL. The implementation of a shared structure in Section 3.2.2 confirms this result.

Figure 3.5 illustrates sharing of data between 4 states. A state needs $5 + 5 + 9$ memory units so representing 4 states requires 76 memory units. With sharing we need 50 memory units. We do not take into account the overhead needed for the implementation because the example is too small to gain anything in practice.

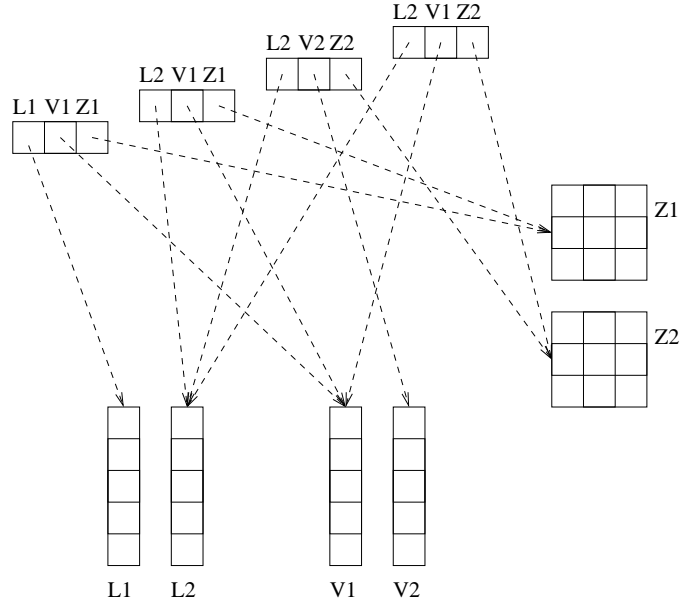


Figure 3.5: Example for sharing data between states.

Although these results show a high amount of redundant data, they tell us nothing about the overhead in time and memory that is taken by an implementation exploiting this property. We can expect a performance hit due to extra computation and gains due to the reduced memory consumption. To emphasize the potential gains of such an implementation, we note that all the numbers in the table would be replaced by 100%.

Exploiting Sharing

We define two levels of data sharing to eliminate data redundancy and exploit the sharing property. The first level is on the location, variable, and zone data: any given location vector, variable vector, or DBM is uniquely stored. The second level is by grouping the discrete parts of a state to

optimize the inclusion check. Let us consider how the inclusion check is defined:

$$(D, S) \subseteq (D', S') \Leftrightarrow D = D' \wedge S \subseteq S',$$

where D is the discrete part of the state, i.e., (\bar{l}, u) or (ρ, μ, θ) for respectively XTA or HTA, and S the symbolic part of the state, i.e., v or ν for respectively XTA or HTA. From this definition and its use in the reachability algorithm, it is natural to consider a more compact symbolic state that shares the discrete part D : $\langle D, \{S_1, \dots, S_n\} \rangle$. This representation of a state as a discrete part and a list of zones (or zone union) allows us to save both memory and time. Memory is saved from the fact that only one discrete representative needs to be stored. Time is saved because only one $D = D'$ positive test is necessary.

The zone union used in our state representation is a simple list of zones. This union can be implemented as a list as defined here, or it can be more elaborate and use the CDD [BLP⁺99] representation that can be used efficiently for analysis. Section 3.2 details the implementation of these different sharing levels and shows how well they perform. They save 80% of the memory consumption and 60% of the time needed for verification (in combination with the PW-List).

3.2 Implementation

The PW-List is the major data structure used in the reachability algorithm as described in Section 3.1.1. The PW-List stores explored and to be explored states. It is built on top of a *storage* structure that stores low level data such as the location vectors, the variable vectors, and the DBMs. The PW-List represents states as $\langle D, \{S_1, \dots, S_n\} \rangle$, with one discrete part D associated to several symbolic parts S_1, \dots, S_n . The storage implements sharing and ensure the uniqueness of the stored data.

3.2.1 The PW-List Structure

The implementation is depicted in Figure 3.6. When sending a state to the PW-List, a hash value is computed on the discrete part of the state, i.e., the location vector and the variable vector. This hash value is used in a discrete state look-up hash table to obtain a discrete state. We use a doubly linked list to manage hash collisions and to remove states that are not kept once they are explored, typically the committed states. The discrete state has *keys* to access the data itself via the *storage* structure. The purpose is

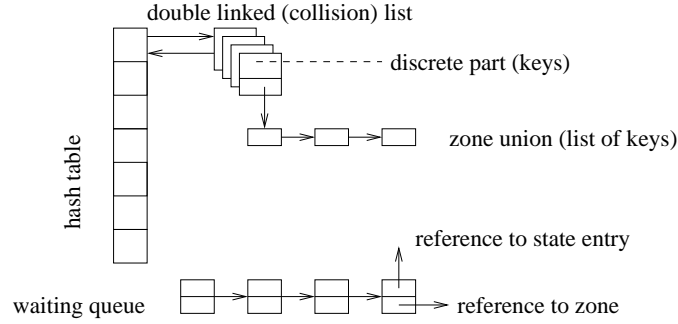


Figure 3.6: PW-List implementation.

to store in any format we wish without affecting the PW-List. Attached to the discrete state is a list of zones, or zone union. The list contains keys to the right data, as the discrete part. If such a state is found with a matching discrete part then the zone inclusion check is performed between the new zone and the zone union. Zones in the list may be removed or the new zone may be declared included. If the state is accepted, i.e., its zone is not included, then a reference is added to the waiting queue whose order depends on the chosen algorithm (depth-first search, breadth-first search, or other searches).

When getting a state from the PW-List, a reference is popped from the waiting queue. The reference is checked to be still valid because it could expire if its corresponding zone is deleted (because of a zone inclusion check). Then this reference is used as the internal *global state reference*, as described in the pipeline architecture in Section 6.1.2. The reference is used to copy the discrete part or the symbolic part of the state on demand.

This structure implements the PW-List as it is defined: the double linked list stores all passed and waiting states in the same way and the waiting queue refers to waiting states among them. The *put* operation corresponds to adding a state to the structure and checking for zone inclusion. The *get* operation consists in popping a state reference from the waiting queue. It happens that states removed from the hash table are referred in the waiting queue. These states are discarded later when their references are popped from the queue. We do not go into this technical detail.

3.2.2 The Storage Structure

The storage structure implements the management of simple data with a minimum of high level operations such as equality testing, copy, and inclu-

sion checking for zones. We need to store such data and these operations are dependent on the representation chosen by a particular implementation. This storage abstracts from different possible implementations, thus allowing us to share, compress, or minimize data with different algorithms without affecting the rest of the code. Figure 3.7 shows the interface of the storage structure. Locations and variables are represented by vectors and the zones by DBMs (difference bound matrix).

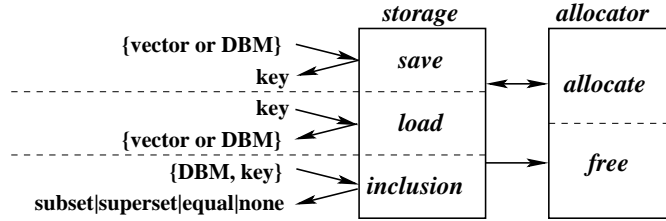


Figure 3.7: Storage interface.

The structure is based on keys like for a database: data is sent to the storage that returns a key used later to retrieve back the data. This mechanism is used to abstract from the actual internal representation. A state is then represented as a tuple of such keys. Furthermore, to test for state inclusion, we need to test for equality of the discrete part and inclusion of the symbolic part. Using a copy from a key to perform these operations is inefficient and this is the reason why the storage has these capabilities in the interface: the comparisons are made with internal representations optimally for particular implementations.

The storage itself is built on top of a specialized data allocator. This allocator has the ability to allocate memory in large chunks and is optimized to deliver many small memory blocks of few different sizes. This means that the memory allocation has little overhead and is efficient for allocating and deallocating memory blocks of the same size. This is justified by the nature of the data we are storing: there are few types of vectors but their numbers are huge.

The purpose of the storage structure is to be independent from particular choices of data representations. The PW-List can use keys to access these data. We give details on two specific implementations, namely using simple copy and using data sharing. Other particular algorithms aimed at reducing memory footprint such as compression or minimal constraint representation [LLPY97] are in integration stage with the code.

Copy Implementation

To evaluate the impact of sharing, we implement a *simple copy* version of the storage. This allows us to isolate the effects of the PW-List. This implementation simply copies data from states, that is, location vector, variables, and DBMs. The memory is managed by our custom allocator. This is similar to the default implementation of UPPAAL.

Sharing Implementation

The sharing implementation uses a hash table to store unique location vectors, variable vectors, and DBMs. Reference counters are used to know when data may be deallocated. The hash table represents a significant overhead and if we add up the overhead from the key reference then the structure may pay off only if there is a substantial amount of shared data. The overhead has two sources: (i) the hash table itself that grows dynamically depending on its filling rate, starting at 1M entries, and (ii) the extra memory needed for the linked lists.

A particular choice is made concerning the deletion of stored data: we deallocate only the DBMs. Due to the high expected sharing for the variables it is unlikely that the states that do not need to be stored, e.g., the committed states, will be the only states to use a given variable combination. Concerning the locations, it is likely that the ones that should be deleted will be reused. Since deallocating location and variable vectors is rarely necessary, we never deallocate them. We save the reference counter and we use a singly linked list instead of a doubly linked list. Concerning the DBMs, the inclusion checking often discards DBMs so it is necessary to be able to deallocate them.

The storage structure described in this way may further be extended to incorporate other optimizations. These may be integer compression or minimal graph reduction [LLPY97] for zones. This sharing is different from the *state compression* used in Spin [Hol97]. In Spin a *global state descriptor* represents a state and it holds a descriptor for the variables, followed by descriptors for every processes and channels. The user may choose the number of bits for these descriptors, which naturally limits the range of these descriptors. Our representation holds one descriptor (or reference) for the locations, one for the variables, and one for the zones. The variable sharing is the only similarity. Locations and variables are treated equally as data vectors and are shared as such. It is important to notice that compression is orthogonal and compatible with this representation. Our approach is similar

to the one in [CK97] for hierarhical colored Petri nets.

3.3 Experimental Results

We use the following examples in the experiments: (1) Audio, Engine, Fischer4, Dacapo, and BC are the same as mentioned in Section 3.1.2; (2) Fischer6 is the Fischer’s protocol with 6 processes, (3) Cups is a combinatorial problem where water is poured from cups to cups until a given goal is reached [DBL02], (4) Master and Slave are the FI master and slave models (the earlier version), and (5) Plant is a production plant with three batches [HLP00].

We conduct the experiments on the development version 3.3.24 of UPPAAL on an Ultra SparcII 400MHz equipped with 4GB of memory. This version incorporates the pipeline presented in Chapter 6 and is already twice as fast than the 3.2.x official versions due to memory optimizations such as the reduced number of copies of Section 6.1.2. We compare results with and without the described PW-List implementation.

Model	No PW-List	PW-List - copy	PW-List - shared
Audio	0.5s/2M	0.5s/2M	0.5s/2M
Engine	0.5s/3M	0.5s/4M	0.5s/5M
Fischer4	0.5s/3.1M	0.5s/3.8M	0.5s/5M
Dacapo	3s/7M	3s/5M	3s/5M
Cups	43s/116M	37s/107M	36s/26M
Fischer6	110s/43M	65s/29M	63s/24M
BC	428s/681M	359s/641M	345s/165M
Master	306s/616M	277s/558M	267s/153M
Slave	440s/735M	377s/645M	359s/151M
Plant	>4G	9207s/2771M	8513s/1084M

Table 3.2: PW-List experimental results.

Table 3.2 shows the time in seconds (s) and the memory used in mega bytes (M) to verify the property `A[] true` (to generate the whole statespace), except for Cups where the reachability property `E<> (cups[2] ==4 and y <= 30)` is verified. Time consumptions less than 0.5s are reported as 0.5s in the table. The result >4G means that the verifier ran out of memory. We choose the options `-Ca` to use DBM representation with active clock reduction. As UPPAAL does not support DBMs of dynamic size, our implementation is limited here. The PW-List is designed for dynamic data

and will give better results in this case if DBMs of dynamic size are used for zones. For the large examples we used the flag `-H273819,273819` (no PW-List) to increase manually the sizes of the hash tables for the passed and the waiting lists. The default sizes give verification times twice longer.

Our implementation consumes up to 80% less memory and if we take into account the factor 2 in speed and the gain in these experiments then it needs 60% less time. The memory gain is expected due to the sharing property of the data. The speed gain comes from only having a single hash table and from the zone union structure: the discrete test is done only once and then inclusion is done for the zone union. The results of the simple copy support these points. The plant example has 9 clocks and 28 integer variables and the simple copy implementation shows the gains attainable when the discrete part of the state is important. The implementation using shared data should be slower because of the overhead of hash computation and equality checking. However, the memory footprint is smaller and in recent computers the computation power is cheap, whereas memory is expensive in terms of resources, so the gain coming from memory exceeds the computation overhead.

The small examples are here to show the overhead of these structures. They also have low data sharing as shown in Table 3.1 and they represent the worst case. Even in the Fischer4 example where the overhead is not compensated the amount of memory used is low and does not matter. The results scale well with the size of the models, in particular the property of shared data holds well. Fischer has unexpected time/memory results compared to Cups, but Fischer is a benchmark for the clock constraints and operations on them are the most expensive in the reachability algorithm.

3.4 Related Techniques

Apart from general algorithms [Ski98, CLRS01], special algorithms and tricks have been developed specially for model-checking. The most recent overview on the optimizations implemented in the tool UPPAAL is found in [BBD⁺02]. The tool uses general techniques for memory management and it has techniques specially developed to handle the symbolic time representation. These techniques are orthogonal to the techniques we have presented in this chapter and can be combined with them.

Memory Management. In [SD98] a scheme to store the generated statespace on disk is proposed. Instead of keeping all data in main memory,

data is sent to disk. This allows the model-checker Murphi to manage larger statespaces. The point is to make the disk accesses in a sequential manner to reduce the huge disk access overhead. The cost in time is of the magnitude of 15%. We are planing to implement such a feature for UPPAAL but the main difference in the technique comes from the symbolic representation: Murphi handles discrete states and UPPAAL symbolic states for which zone inclusion checks are needed.

State compression [Hol97] is a technique to reduce the memory footprint of states. Data values are stored using as few bits as possible. The new storage structure proposed in this chapter can implement this feature. Experimental code shows no performance loss due to the reduced amount of memory used. Another method to reduce memory consumption is to avoid storing states: Larsen et al. [LLPY97] propose to store only states that contain loop-entry locations. This is implemented in UPPAAL. Larsson et al. [LPY00] propose a memory deallocation optimization. It appears that deallocating memory in the reverse order of its allocation improves performance, and in particular when *swapping* is involved. This is also implemented in UPPAAL.

An optimization of timed automata models [HL02] can be performed to avoid the fragmentation of the symbolic statespace. The proposed adjustment of the model is exact in the sense that it does not alter reachability properties. The fragmentation of the symbolic statespace occurs when many zones are generated and cannot be compared, i.e., the inclusion checking concludes the zones are just different. However, these zones often overlap and it is possible to reduce this. This optimization is currently done by the user and it may be implemented as a model optimizer.

Symbolic Model-Checking. Symbolic model checking for timed systems makes use of clock constraints represented by zones [Dil89, Hen94, AHH96, ACH⁺95, YPD94]. In practice these zones are often represented by the DBM structure (difference bound matrix). Rokicki [Rok93] presents the DBM coding and algorithms used to implement clock constraint operations. The implementation of zone manipulations in UPPAAL is based on these algorithms. Another zone representation is the CDD [LPWY99, BLP⁺99] (clock difference diagram) whose idea is borrowed from the BDD [Bry86] (binary decision diagram). CDDs are mainly used in UPPAAL for the deadlock checker because it can handle non-convex zones that result from substractions. Larsen et al. [LLPY97] present a technique to reduce the clock constraint representation. Clock constraints can be seen as a graph and this

technique reduces the graph to a minimal one. This technique is implemented in UPPAAL.

Wong [WT94] describes model-checking techniques using DBM and OBDD (ordered BDD). The OBDD is used to represent sets of states (without time) and the DBM to represent clock constraints. In such techniques [BCM⁺92] the automata are encoded as propositional formulae manipulated as BDDs. Bounded model checking [BCCZ99] is a technique where model-checking is applied for bounded traces of some length decided in advance. Research efforts in symbolic representation go in the direction of fully symbolic representations. Seshia and Bryant [SB03] have recently presented an unbounded, fully symbolic model-checking technique for timed automata. A technique is said to be fully symbolic when it has a single symbolic representation that handles both the finite and the infinite components of the statespace. Bounded techniques unfold the transition relation d times and are limited to check for paths of length d , whereas unbounded techniques guarantee the correctness for any length. Wang [Wan03] has implemented in the tool RED a fully symbolic representation based on CRD (clock restriction diagram) to represent zones and BDD (binary difference diagram) to represent discrete data.

Yovine [Yov97] presents two clock reduction algorithms: the active clock reduction and the equal clock reduction. The technique of the active clock reduction consists in detecting on a model which clock values actually matter in which locations and then to keep only these clocks in the exploration for these locations. The equal clock reduction technique is to detect which clocks have the same value and to keep only one of them. The active clock reduction is implemented in UPPAAL and the equal clock reduction is given for free by the graph reduction algorithm used to store zones.

In [LNAB⁺98] a backward and compositional model-checking technique using ROBDD (reduced ordered BDD) is presented. The analysis is for untimed systems and it has the particularity to check for determinism. UPPAAL dropped support for backward reachability from version 2 because the added features of the language made it very difficult to handle. One of the weaknesses of UPPAAL is that it can not do compositional verifications.

Partial Order. Alur et al. [ABH⁺97] present basic and general partial order theory and practice. The idea behind partial order reduction is to avoid to explore parts of the statespace that lead eventually to the same states as by using other paths. These states are equivalent with respect to an ordering relation and one needs to explore only one representant of these

equivalent classes. In other words, the exact ordering of events often does not affect the properties examined or the “future” of the system. Usual methods [God90, Val90, Pel93] take advantage of this. A survey [CGMP99] presents basic principles and implementations of partial order reduction. The problem of partial order reduction is more difficult for timed systems. An attempt was done by Bengtsson et al. [BJLY98] to implement it for UPPAAL. The method makes use of a local time semantics and it applies for TCTL reachability analysis over timed automata. These results are extended in [Min99] to LTL model-checking. It is difficult to implement this technique efficiently in UPPAAL and the current implementation still needs work.

Godefroid [God95] presents the theory, algorithms, and practice of partial order. It does not apply for timed system though and the module is implemented for SPIN [Hol91]. Pagani [Pag96] defines a partial order based on [God95] adapted to timed graph. In particular an algorithm for deadlock detection is presented. The technique is improved in [DGKK98]. Peled [Pel96] introduces the ample set technique as an algorithm for partial order reduction. The linear and branching time logics are treated. Chou and Peled [CP99] formally verify the correctness of a partial order reduction technique. Whereas it is common to define and apply a reduction algorithm, this paper focuses on the correctness of the reduction.

Symmetry Reduction. The symmetry reduction technique [HJJJ84, ES97] applies for systems having identical components. These components may have their role interchanged so that the verification does not need to explore all the possible combinations. Symmetry reduction can be seen as a special form of partial order. Dill and Ip [ID96] present symmetry reduction in the tool Murphi [DDHY92]. Equivalent classes are used to represent symmetric states and a bisimulation is shown between these equivalence classes. An implementation in UPPAAL to support symmetry reduction is currently in progress.

Approximation Methods. For systems that are too large for exact model-checking, under and over-approximation techniques can be used. The *supertrace* under-approximation technique [Hol91] is based on hashing: every state is represented as one bit in a large hash table. Its effectiveness is analyzed in [Hol98]. A similar technique called hash-compactness [WL93, DU95] stores hash values of states instead of the states themselves. As an over-approximation technique the convex-hull [Bal96] of time zones uses convex

unions of zones from the symbolic states. UPPAAL supports the bit-state hashing, hash-compaction, and convex-hull approximations.

Guiding. The reachability algorithm can be accelerated if the search may be guided towards a goal state. Behrmann et al. [BFH⁺01] present a minimal cost for uniformly priced timed automata. Via special decoration the search is greatly improved to find feasible “recipes” in a steel plant case study. The results are extended for linearly priced automata in [LBB⁺01]. Guiding is supported by a special “guided” UPPAAL version.

3.5 Conclusion

We have proposed two improvements for the reachability analysis of UPPAAL: (1) the PW-List unifies the basic structures passed and waiting lists, which saves time and memory by eliminating states earlier and reducing the number of state inclusion checks; and (2) the implementation of sharing that eliminates all redundant stored data. These techniques reduce memory consumption by 80% and time consumption by 60%. We have shown that these results hold for models of different sizes and that they tend to improve for larger models. Finally, this implementation fits well in the more general pipeline architecture presented in Chapter 6.

Chapter 4

Hierarchical Timed Automata

The case study in Chapter 2 shows the limits of flat networks of timed automata. For the modeler it becomes difficult to manage large number of components and for the model-checker structural information is lost. Hierarchical state machines have been used to deal with these issues. They are usually called statecharts and many variants have been developed from the statecharts of the tool STATEMATE [Har87] to the ones of UML [BJR97].

Our goal is to extend the TA language to include hierarchical constructs. The hierarchical timed automata (HTA) we define can be used to encode other statecharts formalisms naturally due to similar structures. The TA language has a well-understood semantics and it has been used over the past years successfully within the tools UPPAAL [HSL97, LP97, DKRT97, BFK⁺98, HLS99, KLPW99, IKL⁺00, HLP00, LPY01] and Kronos [DOY94, DY98, TY98, NY01]. The TA language fits well for addressing real-time problems and the statecharts for complex concurrency problems. As the different statecharts have little support for real-time we extend TA to HTA and support it in our tool.

In this chapter we give an overview on the statecharts formalisms (Section 4.1) and define the syntax (Section 4.2) and the semantics (Section 4.3) of HTA. We simplify this general formalism (Section 4.5) and revisit the case study model with the HTA language (Section 4.6).

4.1 Introduction

Statecharts were proposed as a modeling language in [HP85] for reactive systems. Since then many variants on the semantics have been proposed. Model-based development has become dominant and makes heavy use of statecharts to describe behaviors. Attempts to agree on a standard modeling language has lead to the rise of UML. We discuss the different variants of statecharts, the UML statecharts, and places where our HTA variant fits.

4.1.1 Statecharts

Let us recall the statecharts formalism. It consists of a collection of finite automata with locations and edges. Automata may be nested inside a given location. These locations may be AND-locations, where all the sublocations are disjoint automata, or XOR-locations, where the automaton may be a connected graph. We use here “location” (for vertex) and “edge” to describe the syntax in graph-theoretical term. People usually use “state” and “transition” but this leads to confusion when semantics is described. We keep “state” for the semantics. When the system is in a state where an AND-location is active, then all its sublocations are active. When it is an XOR-location, only one sublocation is active. When a location is left, all its sublocations are left too. AND-locations are used to model concurrency. A traditional network of concurrent automata is equivalent to one AND-location with these automata as sublocations. Edges have conditional expression and/or triggers. Events may trigger a transition where edges matching the trigger are taken. Edges may generate other events.

From this short and intentionally imprecise description, the reader may imagine several variants on how to queue events, how/when to dispatch them, which edges to choose, etc. Harel [HN96] defined the semantics for the tool STATEMATE. Other people define their own semantics when they develop their statecharts tools. Beeck [vdB94] makes a comparative study on the different statecharts flavors. He classifies them using 19 different semantics criteria. We recall briefly these criteria to classify our HTA model and refer to Beeck’s paper for details:

1. Perfect synchrony hypothesis: this term was defined by the developers of Esterel [BG92]. The system reacts immediately to inputs and generates outputs at the same time. Computation time is negligible.
2. Self-triggering, causality: self-triggering means a transition is taken without being caused by an external event, which is, two edges may

trigger themselves with: (i) receive a /send b , (ii) receive b /send a . Intuitively, causality is not respected in this example.

3. Negated trigger event: to avoid non-determinism where one of two edges may be chosen, the non occurrence of a given event is used. From a location A , having $t_1 : a$ and $t_2 : b$ outgoing edges triggered respectively by a and b is non-deterministic. $t_1 : a \wedge \neg b$ makes it deterministic.
4. Effect of a transition execution is contradictory to its cause: this describes contradiction between a trigger and its action, e.g., *receive $\neg e$ /sends e* on an edge. It is related to causality.
5. Inter-level edges: these are edges that cross the border of a location, thus connecting different automata at different levels.
6. Location reference: this is to test if a location is active in the current system state.
7. Compositional semantics, self-termination: in Beeck's terms, "a semantics of a language L is compositional if the semantics of a compound component of L is only defined by the semantics of its subcomponents, i.e., that no access to the internal syntactical structure of subcomponent is allowed". Maraninchi [Mar89] addresses the causality issue and restricts the language Argos with no inter-level edges. State reference and history mechanism are obstacles too. Self-termination is used to remove inter-level transitions to allow for an active composite location to be exited via a "send event to itself", which is an edge is triggered from a location from a nested edge in that location.
8. Operational versus denotational semantics: denotational style is used to have compositionality but it is purely mathematical and more difficult to use than the operational one. Operational semantics is based on computational models.
9. Instantaneous state: such a state may be simultaneously entered and exited. This is generally forbidden [HPSS87]. This may occur in the situation where an incoming edge to a location generates e and an outgoing edge from the same location is triggered by e .
10. Durability of events: most often events are instantaneous. They occur at a given instant and exist only at this instant of time. One has to

define then which edges have to be taken, i.e., all possible edges or just one.

11. Parallel execution of edges: edges in parallel automata are taken in the same transition. This is generally the case, in particular in the tool Rhapsody (www.ilogix.com).
12. Edge refinement: an edge may be refined in a sequence of edges. The problem is to decide if the sequence of edges is to be taken in one transition or several ones.
13. Multiple entered or exited instantaneous state: if instantaneous states are allowed it may be possible to take an infinite number of edges in a transition.
14. Infinite sequence of edge executions at an instant of time: one can forbid to go through the same location twice as a solution to the “infinite number of edges taken in one transition” problem.
15. Determinism: apart from Argos [Mar92], statecharts allow for non-deterministic constructions.
16. Priorities for transition execution: priority between transitions reduces non-determinism. [PS91] gives a higher priority to edges at a higher hierarchical level, so a transition involving such an edge at a higher level will be preferred to another involving an edge at a lower level if both are enabled. The level is based on where the source location is.
17. Preemptive versus non-preemptive interrupt: this refers to the case of an edge whose source is a composite location with a nested automaton. It is similar to the priority problem, though the edges here have different triggers and some events or edges may be preemptive.
18. Distinguishing internal from external events: internal events are generated on edges and a transition may involve several so-called *micro-steps* with several edges taken consecutively. The problem is to sense or not external events during such a transition, also called *macro-step*.
19. Time specification, timeout event, timed transition: progress of time is modeled on states since transitions are taken in no time in all variants. The question is how to model time and timing mechanisms such as timeout events or timing conditions.

To this list we can add the optional use of *history*, a special memory location that remembers the last active location. Beeck classifies 21 different variants with these criteria.

We complete the classification with additional references. Huizing et al. [HGdR88] model statecharts in an abstract way and gives them a denotational semantics. Uselton [US94] characterizes the statecharts step semantics of Pnueli and Shavel [PS91] and shows that the step semantics is not compositional; in addition a new semantics with a richer structure is defined and is compositional. Damm et al. [DJHP98] give a reference semantics for a verification tool to verify temporal properties of models using the tool STATEMATE. Levi [Lev97a, Lev97b] proposes a compositional proof system for the verification of a discrete timed process language TSP with minimal and maximal delays associated to actions. Mikk, Lakhnech et al. [MLPS97] formalize the rigorous (but informal) description of Harel [HN96]. They extend automata as an intermediate format to facilitate the linking of new tools to the STATEMATE environment [LMS97]. In particular inter-level transitions are reformulated to be handled as ordinary transitions (edges to be accurate). They also propose in [MLSH98] two frameworks to implement statecharts in Promela (SPIN language [Hol97]) that lead to parallel or sequential code. Huizing [Hui91] gives a complicated semantics of statecharts with inter-level edges. Other semantics of reactive system models are discussed. Beeck, Luetzgen, and Cleaveland [LvdBC99] propose a process-algebraic semantics of Harel's statecharts that involves a new process algebra called Statecharts Process Language (SPL). Pnueli [KP92] gives a structured operational semantics for statecharts without inter-level edges or location reference.

4.1.2 UML Statecharts

UML (Unified Modeling Language) is a standard graphical language born from the effort to unify different modeling languages and their methods such as OMT [RBL⁺95, BC95]. The Fusion method [CAB⁺94] was another attempt to take the best parts from different methods and to combine them, but it did not succeed as UML. In short, UML [OMG01, BJR97, BJR99] is a collection of standardized diagrams to describe a system from different views. The different views are:

- The structural view with class diagrams.
- The behavioral view with statecharts, sequence, activity, and collaboration diagrams.

- The environment view with deployment diagrams.
- The implementation view with component diagrams.
- The user view with use case diagrams.

UML has gained support in industry and has received much criticism from the academics for its lack of precise semantics. The UML annual conference exists with affiliated workshop on its different aspects. We will focus on the statecharts part only. The UML statecharts has its own flavor and has the particularity of not being precise. Much effort has been put in giving semantics to UML. The issue of the semantics in UML is so well-known that there is a “frequently asked questions” on it semantics [KER99]. We give here a short overview of the effort:

Porres [Por01] proposes a design method using UML models. He treats the use cases and statecharts analysis. He gives semantics to the statecharts and translate them to Promela with his vUML tool [LP99b]. The statecharts is formalized in terms of operational semantics in [LP99a]. Latella et al. [LMM99a] propose another operational semantics of a subset of UML with its translation to Promela. In [LMM99b] they set the basis to model-check UML statecharts. They map the statecharts to an intermediate format of extended hierarchical automata and they define an operational semantics for these. In its follow-up [GLM02] they give a formal semantics to a subset of the UML statecharts with extension to branching time logic. They prove the correctness of their semantics with respect to major UML semantics requirements and they use their model in the JACK verification environment. Kwon [Kwo00] treats the problem of model-checking UML statecharts with SMV¹. The hierarchical structure is not respected and there are problems with inter-level edges.

[FELR98] is the launch paper of the precise UML project (pUML²). It presents an approach to develop a precise semantics for the UML. It presents also a good discussion of the strengths and weaknesses of Object Oriented methods. It is continued in [EFLR98]. The response to the OMG RFP (request for proposal) on action semantics [AILKC⁺00] gives semantics for actions and describe how they fit into state machines. It does not give semantics for state machines. The OMEGA project³ aims at developing a methodology in UML for embedded and real-time systems based on formal techniques. In this context, Damm et al. [DJVP03] define a kernel language

¹<http://www-2.cs.cmu.edu/modelcheck/smv.html>

²<http://www.cs.york.ac.uk/puml>

³<http://www-omega.imag.fr>

for UML and they give formal semantics for it. They use inter-level edges in the statecharts and provide a flattening procedure. Furthermore, Graf et al. [GOO] attack the modeling of time in UML and they show that the use of timed events provides the right level of abstraction for reasoning about timed computations. The semantics is based on timed automata with urgency. In this project the modeling of time is an extension of a well-defined kernel language.

Bruel and France [BF98] present the benefits of integrating Fusion modeling techniques and Z formal specification notation. The technique is called FuZed and the final result is not UML anymore. Evans and Lano [LE99] propose a rigorous development method for UML, which is illustrated using a small traffic lights problem. The interesting point there is that the three steps of this process can be verified in principle. These steps are: (i) enhancement transformations, (ii) reductive transformation, and (iii) refinement transformations. Beeck [vdB01] gives a syntax and semantics definition for the UML statecharts, in particular for the entry/exit actions and the history mechanism. Action semantics is not treated and event dispatching is treated partially.

Most of the existing work is a mapping of UML statecharts to a known formalism. Furthermore, these mappings are always subsets of the full UML statecharts. This shows the limit of desirable features that the industry needs, as well as the complex process of giving a clear semantics to an unprecise description.

To give more flexibility to the UML, a standard extension mechanism called *profile* is developed. Cook [Coo00] presents an overview on UML and its extension mechanisms. Atkinson and Kühne [AK00] discuss model level inheritance and instance-of relationship between meta models.

We were involved in the process of defining a profile appropriate for verification within the context of the AIT-WOODDES project⁴. This profile makes it possible to define timing constraints and to use our tool UPPAAL to check them. It is based on another profile: UML profile for Schedulability, Performance, and Time Specification [IIC⁺02]⁵.

4.1.3 HTA

Our formalism is an extension of UPPAAL timed automata (TA) to hierarchical timed automata. Whereas standard formalisms use event queues for

⁴<http://wooddes.intranet.gr>

⁵We had access to earlier draft versions. We refer here to the final official version.

communication, we use channel communication. Event queues can be modeled within our formalism. There are many variants of statecharts and our goal is not to define yet another variant. Our goal is to extend the language used by our tool UPPAAL to accommodate useful features of statecharts. As a consequence, the obtained formalism resembles other statecharts, in particular UML, and it makes it appropriate as a target language for verification. As we mentioned previously, most of the semantics work for UML statecharts consists in defining a mapping to a target language. The choices in the UML flavors define that mapping. Our HTA is a good candidate for such a target language. Use of specialized UML profiles facilitate such mappings.

Furthermore, statecharts variants are not comparable and are customized for a particular language, tool, or purpose. There is no satisfactory formalism for our needs, that is, a hierarchical variant based on timed automata with features as urgency and history. Finally, to make the task of adopting an existing formalism even more difficult, statecharts tend to be used as an extension to programming languages with C++ [Str97] embedded in the statecharts. This is a serious obstacle to be considered. We characterize the HTA defined in this chapter with respect to Beeck's list of criteria:

1. Perfect synchrony hypothesis: in our formalism time is modeled with clocks. Action transitions do not affect time, so we respect the synchrony hypothesis.
2. Self-triggering, causality: there is no self-triggering and causality is respected.
3. Negated trigger event: we use channels instead of events. As we have no event queue, the nearest notion is to test for impossibility to synchronize on some channels. We do not support this.
4. Effect of a transition execution is contradictory to its cause: we cannot “send” and “receive” on a given edge, so this does not apply.
5. Inter-level edges: we forbid these edges.
6. Location reference: although it is not in the syntax, nothing forbids to have this as a boolean expression. Such expressions are valued on a given state so variables or locations could in principle be read.
7. Compositional semantics, self-termination: our formalism is not compositional and we would have to restrict and eliminate useful features

such as side-effect on shared variables and global clock access to have compositional semantics. Self-termination is allowed in the simplified HTA we propose.

8. Operational versus denotational semantics: we give an operational semantics since it is more natural to implement it.
9. Instantaneous state: we do not support simultaneous entry and exits of a given location. However, it is possible to enter and exit locations without time delays. The closest notion of instantaneous state in UPPAAL is the *committed state*. We will not elaborate on this feature.
10. Durability of events: since we are using channel synchronizations, the communication is instantaneous.
11. Parallel execution of edges: this is not the case for us. The only way to take in parallel several edges is to use a synchronization on them. There is an extension of the language we do not discuss here that is appropriate for this: broadcast channel synchronization. However, as actions take no time, a sequence of actions can correspond to a “parallelization” of concurrent processes. We support concurrency but not at the edge level.
12. Edge refinement: edges can be broken into several edges that connect *committed* locations. Several transitions are still needed but these edges will have priority. We only mention this feature of the language.
13. Multiple entered or exited instantaneous state: instantaneous states are not allowed.
14. Infinite sequence of transition executions at an instant of time: the “instant of time” here is associated to the taking of a transition. In that sense it is not possible for us. Time is modeled by clocks and it is possible to take an infinite number of transitions in finite time. This behavior is referred as *zeno* [AL92].
15. Determinism: our formalism is non-deterministic.
16. Priorities for transition execution: the only priority mechanism we have is with *committed* locations. This is a special feature of UPPAAL. We skip these in the general syntax and semantics and we do not define priorities. We stress that urgency is related to time delays and is different from priority between action transitions.

17. Preemptive versus non-preemptive interrupt: edges from composite locations are defined but they do not have priorities. Channels do not have priorities either.
18. Distinguishing internal from external events: we do not distinguish them, we have only channels and there is no sequence of consecutive edges taken in a transition, i.e., no micro and macro steps.
19. Time specification, timeout event, timed transition: progress of time is modeled on states. In a given system state time may progress and the clocks are used to measure time. Invariants are used on locations and timing constraints on edges.

Beeck does not include the support of the history mechanism in his list. History may not be necessary supported by all variants of statecharts. We provide a history mechanism.

We propose HTA as an improvement over TA for both the user and the verification engine. The properties of HTA defined in a subset of TCTL are decidable, which is shown by the translation from HTA to TA [DM01].

Composition. In terms of hierarchical structures, the semantics of a language is compositional if the semantics of a compound component is only defined by the semantics of its sub-components, i.e., that no access to the internal syntactical structure of sub-components is allowed [vdB94]. In practice the interest is to be able to deduce properties of a composed model from already verified properties of its sub-components. Argos [Mar92] is an example where hierarchy is defined in a compositional way. However, inter-level edges, state references, and the history mechanism are obstacles to the definition of a compositional statecharts semantics. Our HTA is therefore not compositional. Our purpose here is to define a useful language. As a refinement of the language, one could restrict these features that break composition. It is not our purpose here.

4.2 Syntax

We describe the syntax in graph-theoretical terms with “locations” (for vertices) and “edges”. Composite locations that have nested locations are called superlocations. We use the terms “state” and “transition” in the semantics to refer to the state of the system and transitions between such states. In

the literature state and transition are used for both syntax and semantics and may lead to confusion.

We introduce the data components of HTA that are used in guards, synchronizations, resets, and assignment expressions. Data accesses respect the scope in which they are declared. We give the HTA structure and constraints describing a well-formed HTA.

4.2.1 Data Components

Integer Variables. Let Var be a finite set of integer variables. $Var(S) \subseteq Var$ is the set of integer variables local to a superlocation S . We abuse the notation using Var on sets: $Var(\{u, v\}) = Var(u) \cup Var(v)$.

Clocks. Let $Clocks$ be a finite set of real clock variables. The set $Clocks(S) \subseteq Clocks$ denotes the clocks local to a superlocation S .

Channels. Let $Chan$ a finite set of synchronization channels. $Chan(S) \subseteq Chan$ is the set of channels that are local to a superlocation S , i.e., there cannot be synchronization on a channel $c \in Chan(S)$ between one transition inside S and one outside S .

Synchronizations. $Chan$ stands for a finite set of channel synchronizations, called $Sync$. For $c \in Chan$, $c?, c! \in Sync$.

Guards and Invariants. A data constraint is a boolean expression of the form $E \bowtie E$, where E is an arithmetic expression over Var and $\bowtie \in \{<, >, =, \leq, \geq\}$. A clock constraint is an expression of the form $x \bowtie n$ or $x - y \bowtie n$, where $x, y \in Clocks$ and $n \in \mathbb{Z}$ with $\bowtie \in \{<, >, =, \leq, \geq\}$. A clock constraint $x \bowtie n$ is downward closed if $\bowtie \in \{<, =, \leq\}$. A guard is a finite conjunction over data constraints and clock constraints. An invariant is a finite conjunction over downward closed clock constraints. $Guard$ is the set of guards and $Invariant$ is the set of invariants. Both contain additionally the constants **true** and **false**.

Assignments. A clock reset is of the form $x := 0$, where $x \in Clocks$. A data assignment is of the form $v := E$, where $v \in Var$ and E an arithmetic expression over Var . $Reset$ is the set of clock resets and data assignments.

History. If a superlocation is of type history, it is possible to declare some of its integer and clock variables as part of the history. Such variables are accessible only in the scope they are declared but are treated in the semantics as global variables, i.e., as belonging to the root superlocation. We will not mention history for variables in the following since we consider it as a syntactic convention.

4.2.2 Structural Components

We give now the formal definition of our HTA.

Definition 8 (Hierarchical Timed Automaton (HTA)) A hierarchical timed automaton is a tuple $\langle \mathcal{S}, \mathcal{S}_0, \eta, type, Var, Clocks, Chan, Inv, T \rangle$ where:

- \mathcal{S} is a finite set of locations.
- $\mathcal{S}_0 \subseteq \mathcal{S}$ is a set of initial locations.
- $\eta : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ maps S to all possible sublocations of S . The mapping η is required to give rise to a tree structure where a special superlocation $root \in \mathcal{S}$ is the root. We abuse the notation by using η on sets of locations, e.g., $\eta(\{l_1, l_2\}) = \eta(l_1) \cup \eta(l_2)$.
- $type : \mathcal{S} \rightarrow \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$ is the type function for locations. Superlocations are of type *AND* or *XOR*.
- $Var, Clocks, Chan$ are sets of variables, clocks, and channels. They give rise to *Guard*, *Reset*, *Sync*, and *Invariant*, as described in Section 4.2.1.
- $Inv : \mathcal{S} \rightarrow Invariant$ maps every locations S to an invariant expression, possibly to the constant **true**.
- $T \subseteq \mathcal{S} \times (Guard \times Sync \times Reset \times \{\mathbf{true}, \mathbf{false}\}) \times \mathcal{S}$ is the set of edges. An edge connects two locations S and S' , has a guard g , an assignment r (including clock resets), and a boolean urgency flag u . S is called the *source* and S' is called the *target* of the edge. We use the notation $S \xrightarrow{g,s,r,u} S'$ for this and omit g, s, r, u , when they are absent (or **false**, in the case of u). \square

Figure 4.1 shows an example of the syntax: 4.1(a) depicts a statechart graphically and 4.1(b) shows its tree representation. We note that B is an AND superlocation. The initial locations for every superlocation are marked with a small arrow.

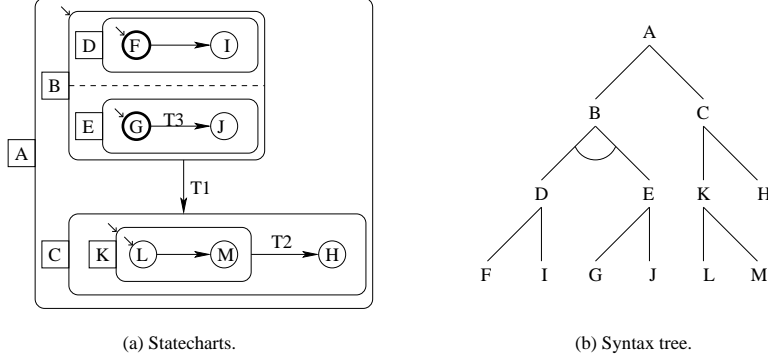


Figure 4.1: Example of the syntax.

Notational Conventions. We use the predicate notation $TYPE(S)$ for $TYPE \in \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$, $S \in \mathcal{S}$. For example, $AND(S)$ is true, exactly if $type(S) = AND$. The type $HISTORY$ is a special case of an entry. We use $HENTRY(S)$ to capture simple entry or history entry, i.e., $HENTRY(S)$ stands for $ENTRY(S) \vee HISTORY(S)$.

We define the parent function:

$$\eta^{-1}(S) = \begin{cases} b, \text{ where } S \in \eta(b) & \text{if } S \neq \text{root} \\ \perp & \text{otherwise.} \end{cases}$$

We extend η^{-1} to operate on sets of locations, i.e., for $\mathcal{S}' \subseteq \mathcal{S}$: $\eta^{-1}(\mathcal{S}') = \{\eta^{-1}(S) \mid S \in \mathcal{S}'\}$. Furthermore, we use $\eta^*(S)$ to denote the set of all nested locations of a superlocation S , including S . $\eta^{-*}(S)$ is the set of all ancestors of S , including S .

We introduce $\tilde{\eta}$ to refer to the sublocations that are proper locations:

$$\tilde{\eta}(S) = \{b \in \eta(S) \mid BASIC(b) \vee XOR(b) \vee AND(b)\}.$$

We use $Var^*(S)$ to denote the variables in the scope of superlocation S : $Var^*(S) = \bigcup_{b \in \eta^{-*}(S)} Var(S)$. $Clocks^*(S)$ and $Chan^*(S)$ are defined analogously.

4.2.3 Constraints for Well-Formed HTA

We give a set of constraints to ensure consistency of an HTA, grouped as the syntactic categories locations, initial locations, variables, entries, and edges.

Location Constraints. We require a number of properties on locations:

1. The function η gives a proper tree rooted at $root$ with $\mathcal{S} = \eta^*(root)$.
2. Only superlocations contain other locations:
 $AND(S) \vee XOR(S) \Leftrightarrow \eta(S) \neq \emptyset$.
3. Sublocations of AND superlocations are not basic:
 $AND(S) \wedge b \in \eta(S) \Rightarrow \neg BASIC(b)$.
4. No invariants on pseudo-locations:
 $HENTRY(S) \vee EXIT(S) \Rightarrow Inv(S) = \mathbf{true}$.
5. For every superlocation S , at most one exit can be declared to be the *default exit*. If the default exit is present then it is reachable from every location in S .

Initial Location Constraints. \mathcal{S}_0 is the set of initial locations and corresponds to the initial location tree. We have $root \in \mathcal{S}_0$ and for every $S \in \mathcal{S}_0$ the following holds:

1. $BASIC(S) \vee XOR(S) \vee AND(S)$: S is a proper locations.
2. $S = root \vee \eta^{-1}(S) \in \mathcal{S}_0$: the location tree is respected.
3. $XOR(S) \Rightarrow |\eta(S) \cap \mathcal{S}_0| = 1$: there is one initial location per XOR superlocation.
4. $AND(S) \Rightarrow \eta(S) \cap \mathcal{S}_0 = \tilde{\eta}(S)$: all sublocations of AND superlocations are initial locations.

Variable Constraints. We forbid conflict in assignments in synchronizing edges (1) and impose scope (2):

1. The following implication holds:

$$S_1 \xrightarrow{g,c!,r,u} S_2, S'_1 \xrightarrow{g',c?,r',u'} S'_2 \in T \Rightarrow \text{vars}(r) \cap \text{vars}(r') = \emptyset,$$

where $\text{vars}(r)$ is the set of integer variables involved in r . We require an analogous constraint to hold for the pseudo-edges originating in the entry of an AND superlocation.

2. For $S_1 \xrightarrow{g,s,r,u} S_2 \in T$, the guard g and reset r are defined over $Var^*(\eta^{-1}(S_1)) \cup Clocks^*(\eta^{-1}(S_1))$ and s is defined over $Chan^*(\eta^{-1}(S_1))$.

Entry Constraints.

1. Let $e \in \mathcal{S}$, $HENTRY(e)$. If $XOR(\eta^{-1}(S))$, then T contains exactly one transition $e \xrightarrow{r} S'$. If $AND(\eta^{-1}(S))$, then T contains exactly one transition $e \xrightarrow{r} e_i$ for every proper sublocation $B_i \in \tilde{\eta}(\eta^{-1}(S))$, and $e_i \in \eta(B_i)$.
2. In case of $HISTORY(e)$, outgoing transitions declare the *default history locations*.
3. At most one entry of a superlocation can be declared to be the *default entry*. If a superlocation S has a history entry, then every sublocation B of S has to provide a history entry or a default entry.

Edge Constraints.

1. Edges have to respect the structure given in η and cannot cross levels in the hierarchy, except one level to entries or exits that make the interface of superlocations. The set of legal edges is given in Table 4.2. Note that edges cannot lead directly from entries to exits. The internal edges are the ones defined inside a superlocation: from a location to a location, from a location to an exit or from an entry to a location. The constraint expresses that the parent location must be the same. The entering edge is from a location to an entry and the fork edge is from an entry to an entry. The exiting and join edges are symmetric to entering and fork. The changing edge is from the exit of a superlocation to the entry of another superlocation. The constraint states that both superlocation must have a common parent.
2. Edges $S \xrightarrow{g,s,r,u} S'$ with $HENTRY(S)$ or $EXIT(S')$ are called *pseudo-edges*. They are restricted in the sense that they cannot carry synchronizations or urgency flags, and only either guards or assignments. For $HENTRY(S)$, only pseudo-edges of the form $S \xrightarrow{r} S'$ are allowed. For $EXIT(S')$, only pseudo-edges of the form $S \xrightarrow{g} S'$ are allowed. For $EXIT(S) \wedge EXIT(S')$, this is further restricted to be of the form $S \rightarrow S'$ (with true as the guard).
3. The syntax does not support directly edges to a composite state such as XOR or AND state. As a notation an edge having a superlocation as target corresponds to its *default entry*. The *default entry* is connected to the initial location of a given superlocation. For edges

having superlocations as source, this is the same as having the source being a default exit connected to all internal locations.


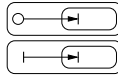
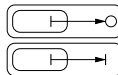
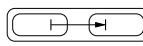
	Comment	S	S'	Constraint
 Internal transitions	Internal	<i>BASIC</i>	<i>BASIC</i>	$\eta^{-1}(S) = \eta^{-1}(S')$
		<i>BASIC</i>	<i>EXIT</i>	
		<i>HENTRY</i>	<i>BASIC</i>	
 Entering transitions	Entering and fork	<i>BASIC</i>	<i>HENTRY</i>	$\eta^{-1}(S) = \eta^{-2}(S')$
		<i>HENTRY</i>	<i>HENTRY</i>	
 Exiting transitions	Exiting and join	<i>EXIT</i>	<i>BASIC(S)</i>	$\eta^{-2}(S) = \eta^{-1}(S')$
		<i>EXIT</i>	<i>EXIT</i>	
 Changing transitions	Changing	<i>EXIT</i>	<i>HENTRY</i>	$\eta^{-2}(S) = \eta^{-2}(S')$

Figure 4.2: Legal edges $S \xrightarrow{g,s,r,u} S'$.

4.3 Operational Semantics

We define now the operational semantics of the HTA formalism. Legal steps between states of a HTA define a set of traces. A *state* captures a snapshot of the system, i.e., the active locations, the integer variable values, the clock values, and the history of some superlocations. States are of the form (ρ, μ, ν, θ) , where:

$\rho : \mathcal{S} \rightarrow 2^{\mathcal{S}}$ gives the active locations. ρ can be understood as a partial, dynamic version of η that maps every superlocation S to the set of active sublocation. If a superlocation S is not active, $\rho(S) = \emptyset$. We define $Active(S) = S \in \rho^*(root)$, where $\rho^*(S)$ is the set of all active sublocations of S including S . We note that for $S \neq root$: $Active(S) \Leftrightarrow S \in \rho(\eta^{-1}(S))$, where $\eta^{-1}(S)$ is the parent location of S .

$\mu : Var \rightarrow \mathbb{Z}$ maps integer variables to their values. If $\neg Active(S)$ then for $v \in Var(S)$, $\mu(v)$ is undefined, which is denoted $\mu(v) = \perp$.

$\nu : Clocks \rightarrow \mathbb{R}_{\geq 0}$ maps clock variables to their values. If $\neg Active(S)$ then for $c \in Clocks(S)$, $\nu(c)$ is undefined, which is denoted $\nu(c) = \perp$.

$\theta : \mathcal{S} \rightarrow \mathcal{S}$ represent the history. As stated previously, history is treated only for locations. The case for variables is only a syntactic convention. $\theta(S)$ returns the last visited sublocation of S or an entry of the sublocation in the case where the sublocation is not basic.

We call a state where all S in $\rho^*(root)$ are of type *BASIC*, *XOR*, or *AND* a *proper state*. Figure 4.3 shows an example of the semantics, in particular the manipulation of the location tree ρ . From the initial state 4.3(a), we fire a transition involving the edge T1 from the superlocation B to the superlocation C. The tree is cut from B and grafted from C to obtain 4.3(b). From there we fire a transition involving the edge T2, which cuts the tree from K and activates the leaf H. We note that the semantical location tree is a subtree of the syntactical location tree.

History. We capture the existence of a history entry with the predicate $HasHistory(S) = \exists b \in \eta(S). HISTORY(b)$. If $HasHistory(S)$ holds, the term $HEntry(S)$ denotes the unique history entry of S . If $HasHistory(S)$ does not hold, the term $HEntry(S)$ denotes the default entry of S . If S is basic $HEntry(S) = S$. If none of the above is the case, then $HEntry(S)$ is undefined. Initially, $\forall S \in \mathcal{S}. HasHistory(S) \Rightarrow \theta(S) = HEntry$.

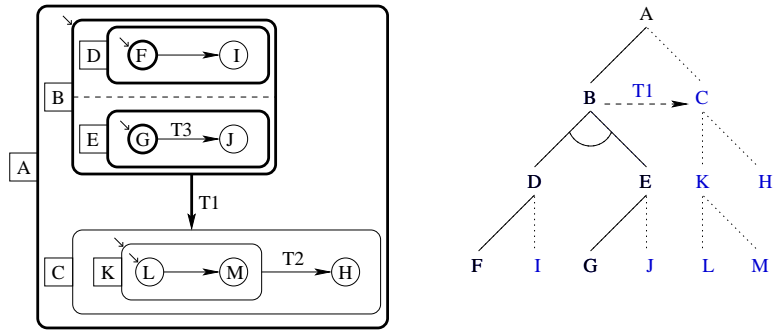
Reached Locations by Forks. In order to describe the set of locations reached by following a fork, we define the function $Targets_\theta : 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$ relative to θ .

$$Targets_\theta(L) = L \cup \bigcup_{S \in L} \{b \mid b \in \theta(\eta^{-1}(S)) \wedge HISTORY(S)\} \cup \{b \mid S \xrightarrow{r} b \wedge ENTRY(S)\}.$$

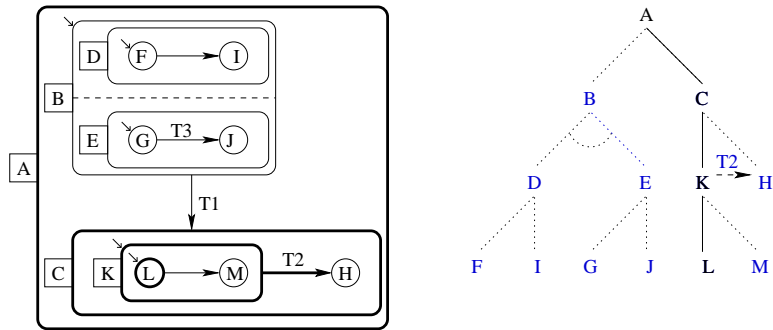
If the argument is a singleton, we use the notation $Targets_\theta(S)$ for $Targets_\theta(\{S\})$. $Targets_\theta^*$ is the reflexive transitive closure of $Targets_\theta$.

State Transformation. Taking an edge $t : S \xrightarrow{g,s,r,u} S'$ entails in general (1) executing a join to exit S , (2) taking the edge t itself, and (3) executing a fork at S' . If S (respectively S') is a basic location, part 1. (respectively 3.) is trivial. Together, 1–3 define a *transition*. We represent a transition formally by a transformation function \mathcal{T}_t , which depends on a particular edge t . The three parts are described as follows. We use $l = \eta^{-1}(S)$ if $EXIT(S)$, $l = S$ otherwise, and $l' = \eta^{-1}(S')$ if $ENTRY(S')$, $l' = S'$ otherwise. This is due to the fact that only proper locations are in the tree ρ .

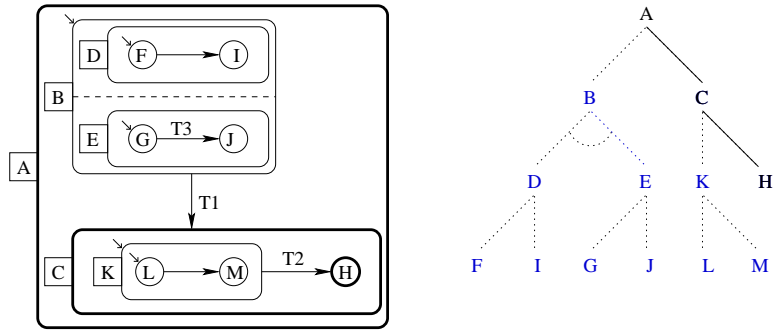
1. *Join*: (ρ, μ, ν, θ) is transformed to $(\rho^1, \mu^1, \nu^1, \theta^1)$ as follows:
 - ρ is updated to $\rho^1 = \rho[\forall b \in \rho^*(l) \setminus \{l\}. b \mapsto \emptyset]$.
 - μ is updated to $\mu^1 = \mu[\forall v \in \text{Var}(\rho^*(l)). v \mapsto \perp]$.
 - ν is updated to $\nu^1 = \nu[\forall c \in \text{Clocks}(\rho^*(l)). c \mapsto \perp]$.



(a) Initial state: statecharts and tree view.



(b) State after taking transition T1.



(c) State after taking transition T2.

Figure 4.3: Example of the semantics.

Let $H = \{h \in \rho^*(l) \mid \text{HasHistory}(h)\}$, if $H \neq \emptyset$ then $\theta^1 = \theta[\forall h \in H. h \mapsto \text{HEntry}(\rho(h))]$, otherwise $\theta^1 = \theta$.

2. *Edge part*: $(\rho^1, \mu^1, \nu^1, \theta^1)$ is transformed to $(\rho^2, \mu^2, \nu^2, \theta^2) = (\rho^1[S'/S], r(\mu^1), r(\nu^1), \theta^1)$. $r(\mu^1)$ denotes the updated values of the integers after the assignments and $r(\nu^1)$ the updated clock evaluation after the resets.
3. *Fork*: $(\rho^2, \mu^2, \nu^2, \theta^2)$ is transformed to $(\rho^3, \mu^3, \nu^3, \theta^3)$ by moving the control to all proper locations reached by the fork, i.e., those in $\text{Targets}_{\theta^2}^*(S')$. We note that $\rho^2(b) = \emptyset$ for all $b \in \eta^*(S') \setminus \{S'\}$ due to the join.

We compute ρ^3 as follows:

$$\begin{aligned} & \rho^3 = \rho^2 \\ & \text{FORALL } b \in \text{Targets}_{\theta^2}^*(S') \\ & \quad \text{IF } \text{ENTRY}(b) \\ & \quad \quad \text{THEN } \rho^3(\eta^{-2}(b)) = \rho^3(\eta^{-2}(b)) \cup \{\eta^{-1}(b)\} \\ & \quad \quad \text{ELSE } \rho^3(\eta^{-1}(b)) = \{b\}. \end{aligned}$$

We derive μ^3 from μ^2 by first initializing all local variables of the superlocations B in $\text{Targets}_{\theta^2}^*(S')$, i.e., $\forall v \in \text{Var}(HSUB) : \mu^3(v) = 0$. Then all variable assignments and clock-resets along the pseudo-edges belonging to this fork are executed to update μ^3 and ν^3 . The history does not change: $\theta^3 = \theta^2$.

Note that parts 1 and 3 correspond to the identity if S and S' are basic locations.

Definition 9 (State transformation) We define the state transformation \mathcal{T}_t for an edge $t = S \xrightarrow{g,s,r,u} S'$ as the result of the *join*, *edge*, and *fork* transformations:

$$\mathcal{T}_t(\rho, \mu, \nu, \theta) = (\rho^3, \mu^3, \nu^3, \theta^3).$$

If the context is unambiguous, we use $\rho^{\mathcal{T}_t}$ and $\nu^{\mathcal{T}_t}$ for the parts ρ^3 (respectively, ν^3) of the transformed state for the edge t .

Starting Points for Joins. A superlocation S can only be exited, if all its parallel sublocations can synchronize on this exit. For an exit $e \in \eta(S)$ we recursively define the family of sets of exits $\text{PreExitSets}(e)$. Each element E of $\text{PreExitSets}(e)$ is itself a set of exits. If the guards of the edges to all exits

in E are true, then all sublocations can synchronize. We use the notation $\eta^+(l) = \eta^*(l) \setminus \{l\}$.

$$PreExitSets(e) = \left\{ \begin{array}{l} \bigcup_{b_1, \dots, b_k} \boxtimes_{1 \leq i \leq k} PreExitSets(b_i), \text{ where} \\ \left. \begin{array}{l} k = |\tilde{\eta}(\eta^{-1}(e))|, \{b_1, \dots, b_k\} \subseteq \eta^+(\eta^{-1}(e)), \\ \forall i. EXIT(b_i) \wedge b_i \rightarrow e \in T \\ \eta^{-1}(\{b_1, \dots, b_k\}) = \tilde{\eta}(e) \end{array} \right\} \text{ if } \begin{array}{l} EXIT(e) \wedge \\ AND(\eta^{-1}(e)), \end{array} \\ \bigcup_{m \in \eta(\eta^{-1}(e))} PreExitSets(m), \text{ where } m \xrightarrow{g,r} e \in T \\ \cup \{\{e\}\} \\ \{\{\}\} \end{array} \right\} \text{ if } \begin{array}{l} EXIT(e) \wedge \\ XOR(\eta^{-1}(e)), \\ \text{if } BASIC(e). \end{array}$$

Here, the operator $\boxtimes : (2^{2^S}) \times (2^{2^S}) \rightarrow 2^{2^S}$ is a product over families of sets, i.e., it maps $(\{A_1, \dots, A_a\}, \{B_1, \dots, B_b\})$ to $\{A_1 \cup B_1, A_1 \cup B_2, \dots, A_a \cup B_b\}$ and is extended to operate on an arbitrary finite number of arguments in the obvious way.

Rule Predicates. To give the rules, we need to define predicates that evaluate conditions on the dynamic tree ρ . We introduce the set of active leaves (in the tree described by ρ), which are the innermost active locations in a superlocation S :

$$Leaves(\rho, S) = \{b \in \rho^*(S) \mid \rho(b) = \emptyset\}.$$

The predicate expressing that all the sublocations of a location S can synchronize on a join is:

$$JoinEnabled(\rho, \mu, \nu, S) = BASIC(S) \vee \\ \exists E \in PreExitSets(S). \forall b \in Leaves(\rho, S). \\ \exists b' \in E. b \xrightarrow{g} b' \wedge g(\mu, \nu).$$

Note that *JoinEnabled* is trivially true for a basic location S . For the invariants of a location we use the function $Inv_\nu : \mathcal{S} \rightarrow \{\mathbf{true}, \mathbf{false}\}$ that returns the invariant of a given location with respect to a clock evaluation ν . We use the predicate $Inv(\rho, \nu)$ to express, that for control locations ρ and clock valuation ν all invariants are satisfied.

$$Inv(\rho, \nu) = \bigwedge_{b \in \rho^*(root)} Inv_\nu(b).$$

For all legal states, the invariants of all active locations have to evaluate to **true**. We use the predicate *EdgeEnabled* over edges $t = S \xrightarrow{g,s,r,u} S'$, that evaluate to **true**, if t is enabled.

$$\begin{aligned} \text{EdgeEnabled}(t, \rho, \mu, \nu) = \\ g(\mu, \nu) \wedge \text{JoinEnabled}(\rho, \mu, \nu, S) \wedge \text{Inv}(\rho^{\mathcal{I}_t}, \nu^{\mathcal{I}_t}) \wedge \neg \text{EXIT}(S'). \end{aligned}$$

Since urgency has precedence over delay, we have to capture the global situation, where some urgent edge is enabled. We do this via the predicate *UrgentEnabled* over a state.

$$\begin{aligned} \text{UrgentEnabled}(\rho, \mu, \nu) = \exists t. \text{EdgeEnabled}(t, \rho, \mu, \nu) \wedge u \\ \vee \exists t'_1, t_2. (u_1 \vee u_2) \\ \wedge \text{EdgeEnabled}(t_1, \rho, \mu, \nu) \\ \wedge \text{EdgeEnabled}(t_2, \rho, \mu, \nu), \end{aligned}$$

where $t = S \xrightarrow{g,r,u} S'$, $t_1 = S_1 \xrightarrow{g_1,c^!,r_1,u_1} S_1$, and $t_2 = S_2 \xrightarrow{g_2,c^?,r_2,u_2} S_2'$.

Rules. We give now the action rule. It is not possible to break it in join, action, and fork because the join can be taken only if the action is enabled and the action is taken only if the invariants still hold after the fork.

$$\frac{\text{EdgeEnabled}(t, \rho, \mu, \nu)}{(\rho, \mu, \nu, \theta) \xrightarrow{t} \mathcal{I}_t(\rho, \mu, \nu, \theta)} \text{ action.}$$

This rule applies for action transitions involving basic locations as well as superlocations. In the latter case, this includes the appropriate joins and/or fork operations.

The delay transition rule is:

$$\frac{\forall d' \leq d : \text{Inv}(\rho, \nu + d') \quad \neg \text{UrgentEnabled}(\rho, \mu, \nu + d')}{(\rho, \mu, \nu, \theta) \xrightarrow{d} (\rho, \mu, \nu + d, \theta)} \text{ delay,}$$

where $\nu + d$ stands for the current clock assignment plus the delay $d \in \mathbb{R}_{\geq 0}$ for all the clocks. Time elapses in a state only when all invariants are satisfied and there is no urgent edge enabled. We omit for simplification the case where we have a pair of urgent synchronized edges enabled.

If an edge with $c!$ is enabled we consider that another with $c?$ is enabled too.

The last transition rule reflects the situation, where two edges synchronize via a channel c :

$$\frac{\begin{array}{l} \text{EdgeEnabled}(t_1, \rho, \mu, \nu) \quad S_1 \notin \eta^*(S_2) \\ \text{EdgeEnabled}(t_2, \rho, \mu, \nu) \quad S_2 \notin \eta^*(S_1) \end{array}}{(\rho, \mu, \nu, \theta) \xrightarrow{t_1, t_2} \mathcal{T}_{t_2} \circ \mathcal{T}_{t_1}(\rho, \mu, \nu, \theta)} \text{sync},$$

where $t_1 = S_1 \xrightarrow{g_1, c!, r_1, u_1} S'_1$ and $t_2 = S_2 \xrightarrow{g_2, c?, r_2, u_2} S'_2$.

We choose the order first t_1 , then t_2 here. This could be inverted, since the constraints for well-formed HTA ensure that the assignments cannot conflict with each other. The side conditions $S_1 \notin \eta^*(S_2)$ and $S_2 \notin \eta^*(S_1)$ prevent synchronization between a superlocation and its own descendants. For example, in Figure 4.4 The $a?$ edge exiting SUB cannot synchronize with the $a!$ edge in P.

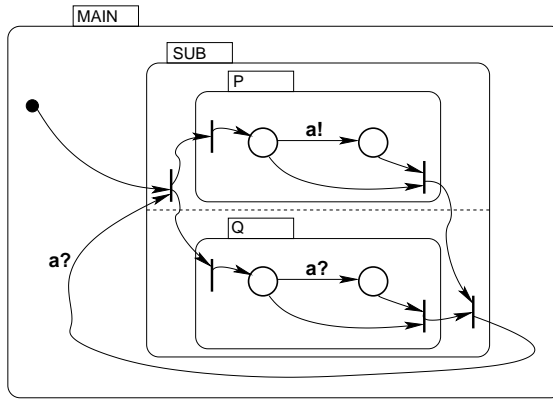


Figure 4.4: The $a?$ edge exiting SUB cannot synchronize with $a!$ in P.

If no action transition is enabled or becomes enabled when time progresses, we have an *action deadlock* state, which is typically a bad situation. If in addition an invariant prevents time to elapse, this is a *time stopping deadlock*. Usually this is an error in the model, since it does not correspond to any real world behavior.

We define a set of timed traces for an HTA that captures its behavior. We explicitly exclude sequences that are zero or not maximally extended.

Definition 10 (Timed Trace Semantics for HTA)

Let $M = \langle \mathcal{S}, \mathcal{S}_0, \eta, type, Var, Clocks, Chan, Inv, T \rangle$ be a hierarchical timed automaton. A *timed trace* of M is a sequence of states $\{(\rho, \mu, \nu, \theta)\}^K = (\rho, \mu, \nu, \theta)^0, (\rho, \mu, \nu, \theta)^1, \dots$ of length $K \in \mathbb{N} \cup \{\infty\}$ if

- (i) It starts at the initial configuration, i.e, for $(\rho, \mu, \nu, \theta)^0$:
 ρ describes \mathcal{S}_0 , μ and ν map the integer (respectively, clock) variables to 0.
- (ii) Every step from $(\rho, \mu, \nu, \theta)^k$ to $(\rho, \mu, \nu, \theta)^{k+1}$ is derived from the rules *action*, *delay*, and *sync*.
- (iii) Maximally extended finite sequences:
 if $K < \infty$, then for $(\rho, \mu, \nu, \theta)^K$ no further step is enabled.
- (iv) Non-zeno:
 if $K = \infty$ and $\{(\rho, \mu, \nu, \theta)\}^K$ contains only a finitely many k such that $(\rho^k, \mu^k) \neq (\rho^{k+1}, \mu^{k+1})$, then eventually every clock value exceeds every bound ($\forall x \in Clocks \forall c \in \mathbb{N} \exists k. \nu^k(x) > c$).

The set of timed traces, denoted by $Tr(M)$, is the *timed trace semantics* for M . □

4.4 Pacemaker Example

We give an example illustrating the use of HTA. It is the cardiac pacemaker case study of [DMY02]. This case study was modeled using the HTA language and was then translated into flat automata. We present it as an example and we do not treat its flattening. We will use this model for the experiments in Chapter 5. This model is motivated by the often-used corresponding UML design example [Dou99]. The hierarchical model is a parallel composition of three *XOR* superlocations: the human heart (Figure 4.5), the cardiac pacemaker (Figure 4.6), and a programmer setting up the pacemaker (Figure 4.7).

Heart Model. The human heartbeat is a complex sequence of chamber contractions where two *atrial* and two *ventricular* chambers collaborate to establish blood circulation. We use a simplified model of a human heart that may require pacing. We consider only two chambers, namely the (left) atrial and ventricular ones. A healthy heart contracts these in a steady rhythm. We mimic this by the time delays `DELAY_AFTER_V` and `DELAY_AFTER_A` and

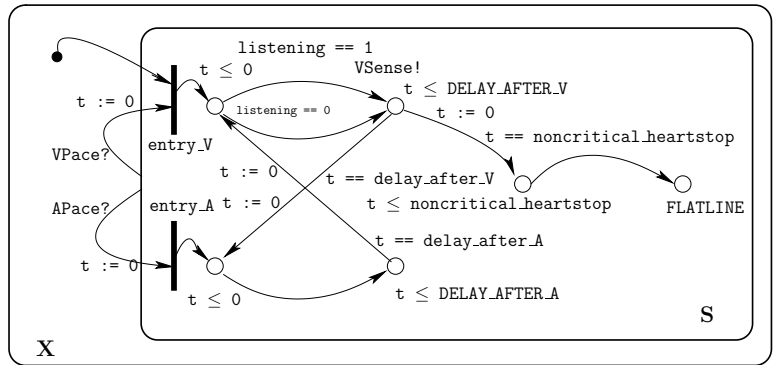


Figure 4.5: Model of a human heart.

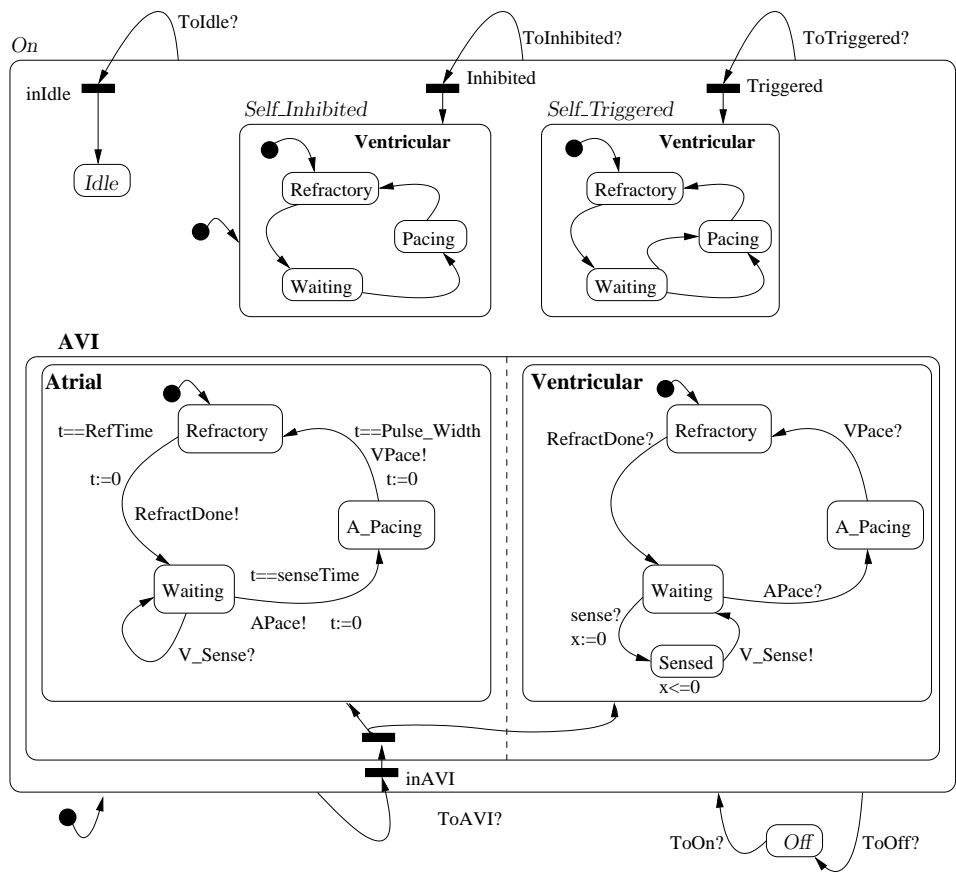


Figure 4.6: Model of the pacemaker. Labels on some edges are missing for simplicity.

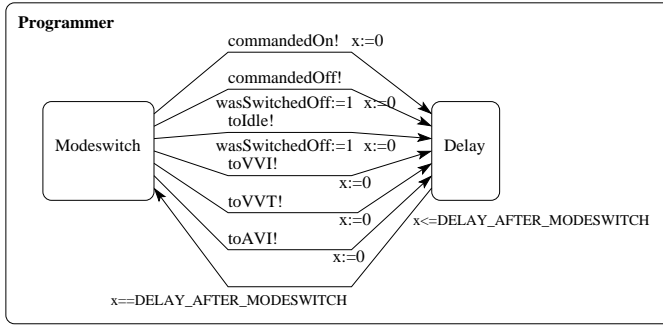


Figure 4.7: Model of the programmer.

the local clock τ . In our example we only monitor the ventricular chamber. The part after `entry_V` synchronizes on `VSense`, in case that anybody is listening (indicated by `listening == 1`).

After the contraction of the ventricular chamber, our heart model may non-deterministically stop beating on its own. If it does so for too long, the critical state `FLATLINE` is reached.

The pacemaker can send an impulse either to the atrial or ventricular chamber, i.e., synchronize on the channels `APace` or `VPace`. The particular heart chamber then is scheduled for contraction in the next moment, regardless on when these signals occur. This is modeled by using the default exit and re-entering at one of the leftmost locations.

Since in our example we only monitor the ventricular chamber, this one synchronizes on `VSense`, in case that anybody is listening (indicated by `listening == 1`). After the contraction of the ventricular chamber, our model may non-deterministically stop beating on its own. If it does so for too long, the critical state `FLATLINE` is reached. A pacemaker can send a signal either to the atrial or ventricular chamber, i.e., synchronize on channels `APace` or `VPace`. The particular heart chamber then is scheduled for contraction in the next moment, no matter when these signals occur. This is modeled by using the default exit and re-entering at one of the leftmost locations.

Pacemaker Model. The main component of the pacemaker is a *XOR* superlocation with the two sublocations *Off* and *On*. If the pacemaker is on, it can be in the different modes *Idle*, *AAI*, *AAT*, *VVI*, *VVT*, and *AVI*. The first letter indicates, to which chamber of the heart an electrical pacing pulse is sent (articular or ventricular). The second letter indicates, which chamber

of the heart is monitored (articular or ventricular). In the *Self-Inhibited* (I) modes, a naturally occurring heartbeat blocks a pulse from being sent. In the *Self-Triggered* (T) modes, a pacing pulse will always occur, triggered either by a timeout or by the heart contraction itself.

For simplicity we restrict to the operation modes *Idle*, VVT, VVI, and AVI. Of particular interest is the AVI mode, which is described as an *AND* superlocation with two parallel sublocations. In our example only the ventricular chamber is observed, but a pace signal may be sent to the ventricular or atrial chamber.

Programmer Model. A medical person—here called the *programmer*—is responsible for switching the pacemaker on/off and for selecting the operation mode. This the programmer does via the signals `commandedOn!`, `commandedOff!`, `toIdle!`, `toVVI!`, `toVVT!`, and `toAVI!`. We do not make assumptions, on how or in which order she issues the signals. However, we require a time delay of at least `DELAY_AFTER_MODESWITCH` after each signal. If one of the signals `commandedOff!` or `toIdle!` was issued this is recorded in the binary variable `wasSwitchedOff`. Note that we equipped the pacemaker with default exits, thus it can *always* synchronize with these signals.

The programmer is modeled by a *XOR* superstate with two locations. In the initial location, `Modeswitch`, any signal can be issued while entering the second location. The second location is left after exactly `DELAY_AFTER_MODESWITCH` time units.

4.5 Simplified HTA

The proposed HTA provides features similar to other statecharts formalisms such as entries and exits. Although inter-level edges are forbidden, they are in fact translated via entries and exits. The purpose of this is to be able to interchange superlocations that have compatible entries and exits. However, the semantics is rather complicated due mainly to the fork and join mechanisms. This is generally the case when dealing with forks and joins, even worse with inter-level edges.

From a model point of view it is useful to be able to have entries and exits since we forbid inter-level edges. We show in this section that these entries and exits are in fact not necessary. We simplify the syntax and semantics, and then we give the minor syntax additions that provide the power of fork and join.

4.5.1 Simplified HTA Syntax

Data Components

We use the same definitions as in 4.2.1.

Structural Components

We simplify the structural components by removing the entries and exits. History serves to mark a location as special. We redefine the HTA as:

Definition 11 (Hierarchical Timed Automaton(2)) A hierarchical timed automaton is a tuple $\langle \mathcal{S}, \mathcal{S}_0, \eta, type, Var, Clocks, Chan, Inv, T \rangle$ as defined before except for the type function $type : \mathcal{S} \rightarrow \{AND, XOR, BASIC, HISTORY\}$ is the type function for locations. Entries and exits are not used.

We reuse notations and constraints for well-formed HTA, though without the entries and exits since there is no pseudo-location anymore. The edge constraints are simplified to: an edge may be defined between two locations only if these locations belong to the same superlocation of type *XOR*, i.e., $S \xrightarrow{g,s,r,u} S'$ if $\eta^{-1}(S) = \eta^{-1}(S')$ and $XOR(\eta^{-1}(S))$.

Contrary to the previous constraints, edges defined between superlocations are legal and are not translated to anything but part of the syntax. The grammar of the language is given in Appendix 8.2 where we give the syntax for the HTA language and the query language.

4.5.2 Simplified HTA Semantics

We redefine the semantics of our simplified HTA without entries and exits. States are of the form (ρ, μ, ν, θ) with the same definitions.

History. History is handled in the same way as before although there is no entry. Default entry is equivalent to initial location of a given superlocation so we substitute default entry to initial location.

Initial locations. Instead of *Targets* we define the set of target initial locations:

$$Init_{\theta}(L) = L \cup \bigcup_{S \in L} \left\{ \begin{array}{l} b \mid b \in \theta(\eta^{-1}(S)) \wedge HISTORY(S) \\ b \mid b \in S \wedge AND(S) \\ b \mid b \in S \wedge b \in \mathcal{S}_0 \wedge XOR(S) \end{array} \right\} \cup$$

If the argument is a singleton, we use the notation $Init_\theta(S)$ for $Init_\theta(\{S\})$. $Init_\theta^*$ is the reflexive transitive closure of $Init_\theta$. We notice that *XOR* locations have only one initial location. In the following we will need $Init_\theta^*$ with the history locations removed:

$$HInit_\theta(L) = Init_\theta^*(L) \setminus \{b \mid b \in Init_\theta^*(L) \wedge HISTORY(b)\}.$$

State Transformation. As there is no entry and exit anymore, the terms *join* and *fork* do not apply as in classical statecharts. ρ describes a tree, so we use the terms *cut* and *graft*.

Taking an edge $t : S \xrightarrow{g,s,r,u} S'$ entails (1) cutting the tree from the source S , (2) taking the edge t , and (3) grafting the tree from the target S' . If S is a basic location then there is no cut. Similarly if S' is a basic location then there is no grafting. It is important to notice that the cut step does not involve any edge, thus data values are not modified. We define the transformation with these three steps. We reuse previous definitions where we substitute $EXIT(S)$ and $ENTRY(S')$ by *false*.

1. *Cut*: it is identical to the previous *join* definition.
2. *Edge part*: it is identical to the previous *edge part* definition.
3. *Graft*:
 $(\rho^2, \mu^2, \nu^2, \theta^2)$ is transformed to $(\rho^3, \mu^3, \nu^3, \theta^3)$ by resolving history and moving the control to initial sublocations, i.e., locations in $HInit_\theta(S')$:
 $\rho^3 = \rho^2$ with the updates $\rho^3(\eta^{-1}(b)) = \{b\}$ for all $b \in HInit_\theta(S')$.

State Transformation. We redefine the state transformation \mathcal{T}_t for an edge $t : S \xrightarrow{g,s,r,u} S'$ using the new ρ^3, μ^3, ν^3 , and θ^3 . It is as before: $\mathcal{T}_t(\rho, \mu, \nu, \theta) = (\rho^3, \mu^3, \nu^3, \theta^3)$.

Remarks. There are two ways to use history with the new syntax: either we use an edge from any location or the initial location of a superlocation is marked as history. In both cases the *graft* will resolve history. As history is updated when exiting a given superlocation, this is well-defined. Also, we notice that states obtained after the *graft* do not contain history locations.

Furthermore, we wish to use self-termination of superlocations. If we recall the *edge part* of the state transformation, we substitute there the source with the destination. However, in the case of self-termination, the source is cut from the *cut* step. This is not a problem and the substitution is not done because the source is no longer in ρ .

Rule Predicates. We need to define fewer and simpler predicates now. We reuse the same $Inv(\rho, \nu)$ definition for invariant. We simplify $EdgeEnabled$ to

$$EdgeEnabled(t, \rho, \mu, \nu) = g(\mu, \nu) \wedge Inv(\rho^{\mathcal{T}_t}, \nu^{\mathcal{T}_t}).$$

$UrgentEnabled$ has the same definition with the new $EdgeEnabled$.

Rules. The *action* and *delay* rules are the same with the updated predicates. We simplify the *sync* rule to allow for self-termination:

$$\frac{EdgeEnabled(t_1, \rho, \mu, \nu) \quad EdgeEnabled(t_2, \rho, \mu, \nu) \quad S_1 \neq S_2}{(\rho, \mu, \nu, \theta) \xrightarrow{t_1, t_2} \mathcal{T}_{t_2} \circ \mathcal{T}_{t_1}(\rho, \mu, \nu, \theta)} \text{sync},$$

where $t_1 = S_1 \xrightarrow{g_1, c^1, r_1, u_1} S'_1$ and $t_2 = S_2 \xrightarrow{g_2, c^2, r_2, u_2} S'_2$. We keep $S_1 \neq S_2$ otherwise the target locations conflict. The self-termination is well-defined because the conflict in changing locations is automatically resolved by the *cut* step. We notice that the user has to rename channels if she wants to use constructions like in Figure 4.4. In this updated semantics we do not have to take care of the *PreExitSets* as before and we simplify or do not use previous predicates.

4.5.3 Expressiveness

Now we provide ways to code the previous forks and joins. Forks via entries are replaced by a choice from initial locations. The choice is coded by “entry” variables that are just normal variables. To obtain the atomicity of forks, *committed* locations can be used. They add a level of priority on which edges may be chosen to take transitions and forbid delays.

Concerning joins, we can decorate the locations with “location functions”, similarly to methods in object oriented languages. Such functions are accessible from outgoing edges only and are used normally in guards. These functions define internal conditions that may range from variable to internal locations and include other nested location functions. This does not affect the semantics or the syntax. It is important to notice that such function are *side-effect* free. What we do here is to move the complexity of the previous rules to calculate joins to a flexible function encoding in the automaton.

It is now more natural to refine existing locations or to change automata nested in locations. Figure 4.8 shows how to translate some conventional statecharts constructs with inter-level edges to the simplified HTA model. This example shows the basics for default entry, selective entry, synchronized join, preemptive join, and default join. We use the (c) notation to refer to a *committed* state. We have to define the location functions for the superlocations M, A, and B. Notice that changing any superlocation and giving the appropriate functions does not change the rest of the automaton. Refinement is intuitive and the model is appropriate for abstraction since the user may provide approximate functions for incomplete superlocations.

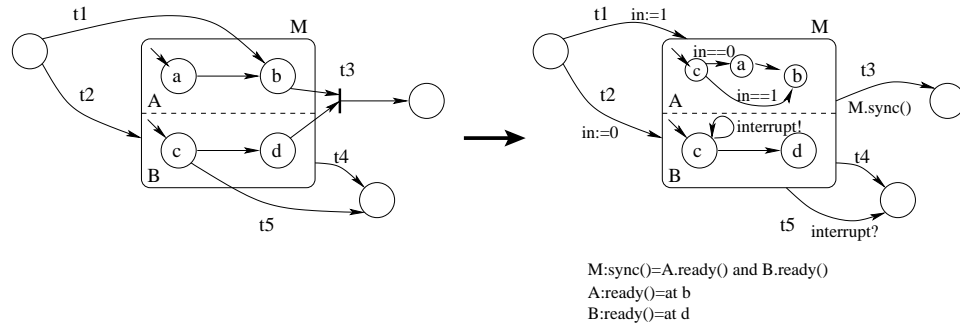


Figure 4.8: Example of translating statecharts to the simplified HTA.

In the following HTA will refer to the simplified HTA model only.

4.6 Case Study Revisited

In this section we adapt the model of the case study presented in Chapter 2 to the HTA model. The FI part of the case study is not appropriate for hierarchical modeling. The automata have reasonable size, have a flat straightforward structure, and adding hierarchy is artificial and does not give anything. The bus coupler part is appropriate for the HTA.

Figure 2.12 shows the different parts involved in the model. The tasks serving the ports 1 and 4 are updated by moving the automata used as functions inside locations of these tasks. Figure 4.9 details the transformation of the model. The c is the UPPAAL *committed* location notation. Locations marked with double circles are initial locations for a given automaton (UPPAAL notation). We use self-termination to control when the superlocation should be left. The *call* synchronization is removed since the nested automaton is automatically activated upon entry of the superlocation. We

note that the target location of a self-terminating synchronization does not matter for the nested location because the target is never entered.

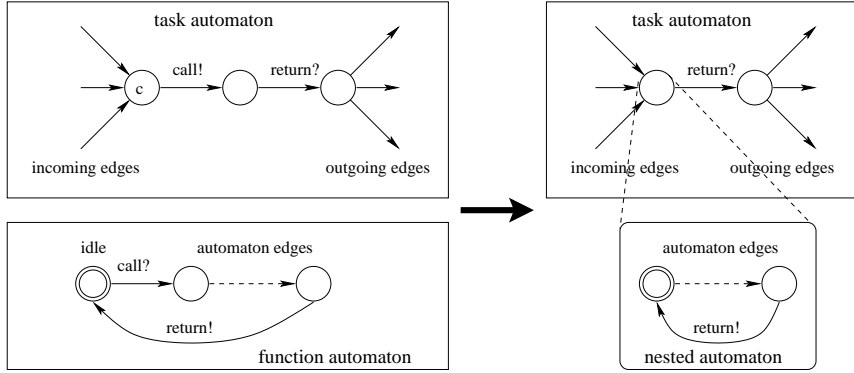


Figure 4.9: Model transformations from flat to HTA.

Consequently, the obtained model has 6 nested automata. In addition to the simplification due to hierarchy, we have suppressed intermediate locations to connect directly superlocations with each other. These are obvious optimizations that are given by the HTA model and the two models can be compared fairly. We have to rewrite properties to accommodate the renaming of the locations that are now nested. The verification is conducted with the prototype described in Section 5.1. We use the CVS internal version 3.3.36 and a branch from that version that implements our HTA. This version has other improvements that we describe in Chapter 3. Verification is run on the same Sun Ultra-II 400MHz as previously. We obtain the same results for the properties, as expected. Results of the statespace generation are given in Table 5.1 in Section 5.1.3. The HTA models outperforms the TA model by 14% in memory at no speed cost. The direct support of the hierarchical model allows the engine to get rid of intermediate states that were artificial for the flat model. Furthermore, when the model has more structure, as for the pacemaker, then the advantage of the transitions involving edges from superlocations shows up as a speed improvement.

This case study shows the benefits of our HTA language: it is more natural to use as a modeling language and it improves model-checking performance.

4.7 Conclusion

We defined a timed statecharts based on timed automata. This hierarchical timed automaton (HTA) is given a formal syntax and semantics. This formalism is then simplified to keep the essential hierarchical feature while removing the entries and exits that clutter both the syntax and semantics. Finally, it is shown that this language is useful from the modeling point of view. It allows for a more natural description for the case study. From the verification point of view, a prototype (not yet optimized) achieves better performance.

Chapter 5

A Verification Engine for Hierarchical Timed Models

Finite state machines are a widely used model in computer science. Several extensions have been proposed to enhance the expressive power and to describe more complex system more efficiently. Such extensions include addition of concurrency (communicating finite state machines), variables (extended finite state machines), time (timed automata), and hierarchy (hierarchical state machines). Improving expressiveness and model succinctness comes to the cost of computational complexity [Pap95] for the verification problem and the addition of hierarchy is the source of theoretical exponential blow-ups.

Alur and Yannakakis [AY98] show that for flat Kripke structures, deciding reachability between two states is in NLOGSPACE. For hierarchical structures, the reachability problem becomes PTIME-complete and the LTL and CTL model-checking problems are PSPACE-complete. Adding only hierarchy gives an exponential blow-up in time. For time, Alur, Courcoubetis, and Dill [ACD90, ACD93] show that the model-checking problem for determining the truth of a TCTL formula with respect to a timed graph is PSPACE-complete. Yannakakis [Yan00] shows that model-checking hierarchical state machines with arbitrary level of concurrency at any depth is EXPSPACE-complete. Furthermore, the model is exponentially succinct in term of expressiveness with respect to the size of the model: there is an exponential blow-up from finite state machines to hierarchical state machines, and from there a double exponential blow-up to concurrent hierarchical state machines.

Our HTA model suffers both from the complexities due to hierarchy and

time. This makes the implementation of a verification engine for hierarchical timed models a challenge. In this chapter we address implementation issues to verify hierarchical systems. We first discuss the representation and computation of states adopted for UPPAAL. Then we present an abstraction technique using hierarchical information to approximate verification. The abstraction is defined to preserve safety properties: if such a property holds for the abstract model, then it holds for the concrete model.

5.1 Representation and Computation of States

In this section we discuss different approaches to implement the hierarchical structures that represent states and in particular the one we choose in UPPAAL. The main problems in implementing hierarchy are (i) the dependency of the variables and clocks on the location vector, and (ii) the transitions between superlocations.

Variables are defined within the scope of superlocations. Changing superlocations may involve removing some variables from the state and adding new ones. This holds for the integer variables and the real clock variables. The set of variables is dynamically changing depending on the location vector in this respect. Furthermore, operations defined on edges refer to variables. Such references are implemented as indices but as the set of variables changes, these references are dynamic and must be computed on-the-fly.

Transitions may involve fork and join edges. They result in manipulating the tree structure of the locations, as described in the semantics of Chapter 4. We have referred locations as “location vector”, which is the representation used in our implementation. The tree is encoded in this vector. The tree manipulation may occur at different levels, from a node or a leaf. The operations are the “cut” and “graft”, if we see the location representation as a tree, which is the same as “join” and “fork” in the statecharts naming conventions.

An implementation handling only flat models does not need to consider these problems. It is then a challenge to keep the same level of efficiency for a hierarchical implementation compared to a non-hierarchical one. In particular for UPPAAL, we have to maintain backward compatibility with the current TA language at almost no cost otherwise the new implementation will have a limited interest.

5.1.1 Data Structures for Representing States

We present different approaches to implement the data structures representing states, i.e., the locations, the variables, and the clocks.

Locations

As we are working on a tree representation and we want to save memory, it is natural to try to work on the smallest possible structure. With this idea, we have implemented a representation that saves only the leaves of the tree, i.e., only the basic locations at the lowest nested level. This implementation follows the semantics of the HTA with entries and exits. In particular, the set of locations to be exited, see Section 4.3, is difficult to compute in practice. During exploration, the view that the model-checker has on the data structure is local, which is we “see” one basic location at a time with one edge at a time, and this is the root of the problem because joins are not local.

Figure 5.1 shows how the location tree is updated: the example is based on the statecharts 5.1(a) with the corresponding syntax description 5.1(b). From the initial state graphically depicted with thick locations in 5.1(a), the tree 5.1(c) is updated to 5.1(d). This transition corresponds to the default exit of the location B and it is taken at a high level in the tree. Working on the leaves implies that we have to propagate the information up and down the tree to compute the exit of B and the entry of C. A case of a transition between leaves (L to M) updates 5.1(d) to 5.1(e). Finally, 5.1(f) is obtained from a join at an intermediate level in the tree (K) to a leaf (H).

Let us consider an implementation based on leaves: the internal representation is a multi-graph with locations (or vertices) connected by multi-edges. The locations correspond to the leaves in the tree. Resolving a fork is straightforward since it is enough to follow all the edges iteratively. Resolving a join efficiently needs more work. The basic principle when computing transitions is to try all outgoing edges from all active locations. The problem with the joins comes from the fact that some edges must be taken only once and they require other locations to be active. We implemented a stamp algorithm to handle the joins: every join computation is associated with a stamp to validate the edges participating in the join. When there are precisely n edges enabled with the same stamp for a join of size n (defined thereafter) then the join can be taken. The stamp is important to avoid to use edges more than one time: as edges may have several sources and all sources are examined, they may be used several times. The algorithm

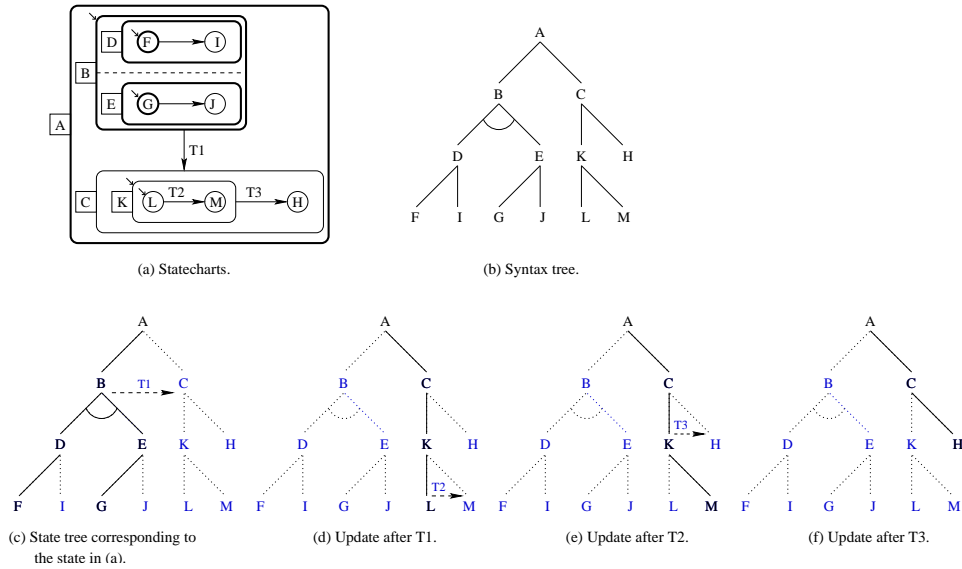


Figure 5.1: Updates of the location tree at different hierarchical levels.

is adapted for synchronization computations: we may need one join for the sending part, e.g. $c!$, and one more for the receiving, e.g. $c?$. We use two stamps to distinguish these two parts.

The size of a join is the number of active source locations that are required to execute it. Figure 5.2 shows that the size is dynamic and has to be computed dynamically. In the first case (a) the join involves 3 locations and the second case (b) there are 2. An alternative representation (c) separates these two cases with two multi-edges: the problem with this representation is that the graph needs to be “unfolded”, which destroys its compactness. The trade-off is to compute the size of the join on-the-fly.

From the implementation based on leaves and from the fact that our model was not simpler from the other existing statecharts, we simplified the model to the HTA without entries and exits. This gave back a simple graph to work on. The simpler data structure is an extension of the basic location vector of UPPAAL. The location vector represents the whole tree this time. This makes computation of transitions using edges from superlocation more natural. However, we have to take care of not active superlocations (or types *AND* and *XOR*). In addition to this we need a cut/graft filter¹ to execute the corresponding cut/graft operations of the semantics. This filter allows

¹a block in the pipeline architecture, see Section 6.1.1.

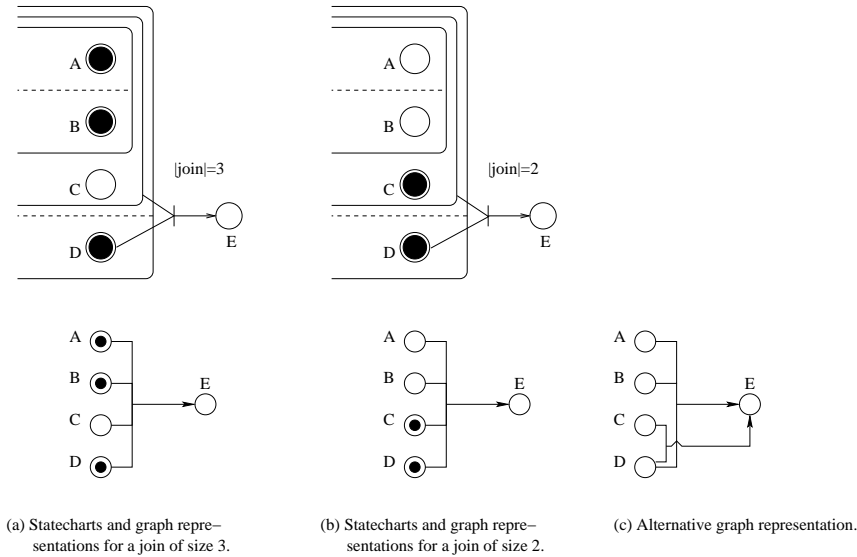


Figure 5.2: Dynamic size for joining multi-edges.

us to use the self-termination channel synchronization. We detail this filter in the next section.

We are working on a third approach to take the best from these both approaches, namely saving the leaves only and decompressing to the whole tree for successor computation.

Variables

The current implementation does not take advantage yet of the variables marked inactive even if the library is implemented. This has no impact in practice because the models we use do not take advantage of local variables yet. There are mainly two choices we face for the representation of the variables:

1. The variables are represented as one large global array where every declared variable has its cell. The active variables, depending on the active locations², are saved on a smaller vector and a mapping is computed and used upon load and save. The operations are applied directly on the large array.

²some local variables “exist” only for some given configurations.

2. The variables are represented as a small array containing only the active variables. In this case the mapping to the globally indexed variables is used upon access, which is, within the operations.

If we analyze these two solutions, it appears that we always need the mapping, so the difference is when we use it. In the first case we use it for all the active variables twice. In the second case we use it depending on the accesses. Now considering that the models always access a few variables on the edges, (2) is better. Another advantage for (2) is that it allows simpler code for the different load/save algorithms, and there are many optimizations (and implementations) possible there. Finally, the mapping has to be updated for some transitions. The cost of updating the mapping and updating the small variable array (shrink or grow) has to be taken into account too. However, it is small and it occurs rarely with respect to all the computed transitions because this concerns only some of the transitions involving a cut/graft.

Clocks

The current implementation uses the *free* operation in UPPAAL to free unused clocks. We have similar choices as for the variable case to optimize this. However, the clock representation for n clocks needs $(n + 1)^2$ integers in memory for the DBMs we are using³. Furthermore, common operations cost $O(n^2)$ in time and one $O(n^3)$ (Floyd’s algorithm to tighten the constraints [Flo62]). With these considerations the approach (2) for variables applied to clocks is better.

We note that the dimension of the zone, i.e., the number of clocks used, is dynamic and depends on the active locations. This is similar to the notion of active clocks, which is more fine grained. Furthermore, using the compact zone representation stores exactly the constraints of active clocks.

5.1.2 Computation of States for the Simplified HTA

The computation of states for the simplified HTA as defined in Chapter 4 the successor computation of UPPAAL with a back-end implementing the HTA semantics. The architecture of the engine is presented in Chapter 6 and is based on a data pipeline. Computing states for HTA is done by inserting a new “filter” block that implements the graft and cut operations. The graft and cut basic algorithms are given as pseudo-code in Figure 5.3. The corresponding **graft** function is given in Figure 5.4.

³+1 comes from the reference clock


```

repeat
  stabilized = true
  for loc in location_vector do
    if loc != idle then
      if loc.parent != idle or justLeft(loc) then
        loc = idle
        reset(loc.parent.var)
        reset(loc.parent.clocks)
        stabilized = false
      endif
      if loc.nested == idle then
        graft(loc.nested)
        stabilized = false
      endif
    endif
  done
until stabilized

```

Figure 5.3: Graft/cut stabilization algorithm.

```

graft(loc):
  if AND(loc) then
    for l in loc do fork(l) done
  else
    loc.init()
  endif
  reset(loc.var)
  reset(loc.clocks)

```

Figure 5.4: Fork algorithm.

The basic algorithm is a fix-point computation to stabilize the state to a valid configuration. The current configuration may contain a started graft or a cut that both need to be completed. As the cut part accesses the state of the parent location, this may require additional loops if the parent location is not stabilized itself. The first **if** block corresponds to the cut case. The location of a superlocation is set to idle if its parent is idle or if the location was just left. The additional or-condition is used to resolve transitions involving edges from a superlocation to itself where the superlocation has to be left and re-entered. The second **if** block corresponds to the graft case. The superlocation is entered recursively to its initial location.

In practice the locations are declared in hierarchical order with the top-level first. This leads to a simplification of the basic algorithm where the **repeat until** loop may be removed with the *stabilized* flag. We need then to separate the cut and graft in two different loops to resolve the special case of edges starting from and ending to the same superlocation. The algorithm is then correct because when a location is tested for cut, it is guaranteed that its parent's cut was resolved before. Furthermore, the graft algorithm is forward and it does not need more loops. The algorithm is optimal in the sense that for a location vector (tree description) of length n we test for exactly n cuttings and graftings.

5.1.3 Experimental Results

To test the hierarchical engine we use our hierarchical model from the case study of Chapter 2 and the pacemaker example presented in Section 4.4.

The Fieldbus

Table 5.1 shows the memory and time consumptions to generate the whole statespace obtained with the property `A[] true` for the case study model represented in the HTA and the TA language. We use the CVS version 3.3.36 that incorporates a default PW-List modified with the HTA implementation. The machine used is the same 400MHZ Ultra Sparc station as in previous experiments.

Engine version	HTA model	TA model
TA	-	98M / 113s
HTA	84M / 112s	98M / 118s

Table 5.1: Resource consumption for the whole statespace generation.

Reading the table vertically shows the small overhead due to the cut and graft operations. However, as the HTA language is better adapted for this model, the gain compensates this overhead. We gain about 14% in memory at no speed cost. Although the implementation has room left for optimizations it outperforms the TA engine for an equivalent non hierarchical model. Furthermore, the test was run in this case with the graft/cut turned on for the TA model on purpose to evaluate its overhead separately. This can be turned off for simple TA models as it is a filter that can be skipped on the pipeline, in which case the result is identical to the TA version. This overhead is negligible and is even compensated when a HTA model is used.

The Pacemaker

We check the following properties:

- (i) $A \square \text{Heart.Detail.Flatline} \text{ imply } \text{wasSwitchedOff} == 1$,
- (ii) $A \square \text{Pacemaker.On.Idle} \text{ imply } \text{wasSwitchedOff} == 1$,
- (iii) $E \langle \rangle \text{Heart.Detail.Flatline}$, and
- (iv) $E \langle \rangle \text{Pacemaker.On.Idle}$,

where (i) and (ii) are safety properties saying that the heart is allowed to stop beating only if the pacemaker is switched off. (iii) checks that the heart may stop to beat to check that property (i) is not trivially true. (iv) checks that the pacemaker may go idle to check that (ii) is not trivially true. We check these properties on the hierarchical model described in the simplified HTA language and on the flat model obtained from the automatic translation HTA-to-flat-TA described in [DMY02]. More details on the translation are found in [DMY03].

The pacemaker is sensitive to timing constraints and the results of the verification depend on the constants used to tune the model. Figure 5.5 shows the constants for which the safety properties hold. If we change `MODE_SWITCH_DELAY` to 65 then (i) does not hold any more, meaning that changing mode too quickly may put the patient in a dangerous situation. We allow the programmer (in the model) to switch the pacemaker off to generate more

```
REFRACTORY_TIME = 50
SENSE_TIMEOUT   = 15

DELAY_AFTER_V = 50
DELAY_AFTER_A = 5

HEART_ALLOWED_STOP_TIME = 135
MODE_SWITCH_DELAY = 66
```

Figure 5.5: Constants for which the properties hold.

behaviors and we check the models for this delay set to 65 and to 66. Table 5.2 shows the results. We use a Celeron 633MHz equipped with 192M of memory under Linux 2.4.18 for this experiment. The verifier used is the same HTA engine as in the previous experiment. We give time and memory consumptions to verify the four properties in the same run.

Model	Delay=66	Delay=65
TA	5.3s/13M	15.5s/14M
HTA	0.5s/4M	0.4s/4M

Table 5.2: Results to verify the pacemaker.

This example gives good results for memory and speed. This is due to the fact that the pacemaker was designed from the beginning with hierarchy and makes heavy use of *XOR* and *AND* locations with edges defined from superlocations. The experiments show the cost of translating these edges while they are cheap for a native hierarchical model.

5.2 An Abstraction Technique for HTA

5.2.1 Background

Abstract Interpretation

The Galois connection [CC92a] is appropriate for interpretation of programs where the concrete and abstract domains are naturally defined as integers. This relation must be defined between two partially ordered sets (posets). In our case where the predominant structures are the control location and the clocks (whose representation is already symbolic) it is artificial to define the partial order. This will not help us in model-checking properties, though we use similar techniques to manipulate the variables of our automaton. This technique can be used to define precise abstract functions and in our case of model-checking we are interested in the properties. It is costly to define one abstraction for every property we want to verify. We prefer to focus on abstracting the model based on its hierarchical structure and then check several properties on it. The extensive work of Cousot & Cousot [CC77, CC79, CC92a, CC92b, CC94, CC99, Cou00, CC02] presents abstract interpretation of programs, abstract model-checking, and refinement techniques.

Levi [Lev01] presents an abstraction for μ -calculus model-checking. The idea is to obtain an approximate semantics by substituting the domain of

computation and its basic operations with an abstract simpler domain. This is similar to our work though we apply the technique to timed automata.

Cleaveland et al. [CIY95] develop a framework based on state abstraction. They use the *democratic* Kripke structures to abstract Kripke structures and show that a class of them that are safe approximations in the sense that CTL* properties satisfied for the abstract model also hold for the concrete model. Our abstraction is preserves only safety properties and we focus our efforts in abstracting time.

Hiding and Refinement

This technique consists of hiding variables or locations from the original model and using abstract values instead in the abstract model. To do this a mapping (surjection) from the concrete states to the abstract states is defined. For safety properties $\widehat{M} \models \phi \Rightarrow M \models \phi$ with M the concrete model, \widehat{M} the abstract model, and ϕ the property to check.

Usually the variables to keep are chosen within the *cone of influence* [BBD⁺99] of the properties to check, where one tracks data dependency relative to properties to abstract the other data. The variables we chose to abstract are obtained from the hierarchical structure we want to abstract.

This technique is by essence an over-approximation and spurious traces may be obtained. These traces or counter-examples do not have concrete equivalents in the concrete model. In such cases further refinement of the abstract model is needed. In [CGJ⁺00] Clarke et al. show how to identify spurious traces and to use them to guide refinement. In [CCK⁺02] Clarke et al. present a SAT based automatic abstraction refinement framework where the abstract model uses variable hiding. The same authors presented also a BDD approach previously [CGL94].

Tripakis and Yovine [TY01] show how to verify dense-time systems modeled as TA with untimed verification techniques. There the TA is transformed into another where exact time delays are abstracted away. This abstraction is formalized under the concept of timed-abstracting bisimulation [LY93]. In practice the algorithm generates a graph (minimization step) whose vertices correspond to the zones of the symbolic time representation. The interest is to be able to have more or less precise time graphs with different refinements. In our work we keep time information and the different levels on refinements are defined in function of the hierarchy of the model. It is also interesting to quote the opposite work that consists in refining abstract descriptions into more precise implementations. Alur et al. [TAKB96] tackle the problem of proving the correctness of such refinements. This re-

finement checking is implemented in the verifier COSPAN.

Preservation

We apply abstraction techniques to approximate a complex model by a simpler one. To make this technique useful we have to preserve some properties between the abstract and the concrete models. We refer the reader to Dams' thesis [Dam96] for more details on preservation theory. Jensen [Jen99] gives general conditions for preservation of properties based on simulation relations for untimed and timed systems. We also base our abstraction on simulation and we are interested in safety properties.

Abstraction in the Case Study

In the case study of Chapter 2 we use variable and state hiding to code an under-approximation model for debugging purposes. The abstraction is carried out manually on implementation specific data. The abstract model is then the protocol logic as described in the implementation that corresponds to the one in the specification.

5.2.2 Abstracting Away from Hierarchy

There has been extensive work on abstraction techniques and our goal is not elaborate on the theory but to use it and define an abstraction naturally adapted for our model. Our abstraction is based on variable, clock, and location hiding. The hiding is guided by the hierarchical structure of the automaton. Our proposed abstraction is innovative in the sense that it proposes an abstraction on symbolic states. Clock constraints are approximated according to the part of the automaton that is abstracted. In contrast to [TY01] where time is abstracted away, we keep here time information.

Idea of the Abstraction

The idea of our abstraction is to remove nested automata and to replace them by information that describes what the removed automata may do. Figure 5.6 illustrates this idea. We want to abstract the superlocation A , which is, to remove its nested automata. There are 3 locations with invariants on the clock x , so the maximal possible upper bound for x is $x \leq 4$ in this example. There is no upper bound for the clock y so we assume y is not bounded. For the lower bounds, we note that x is reset but not y , which gives the constraints describing what may happen to the clocks: $0 \leq x \leq 4$

for x and $y_0 \leq y$ for y with y_0 being the clock value of y when entering A . For the variables, we note that a may be updated. It is expensive to know the exact value of a since we have to analyze the conditions of exit of A , and analyze the nested automaton. Instead we extract the information “ a is *unknown*”. Finally, the nested automaton may synchronize with another automaton, in this example on $c!$.

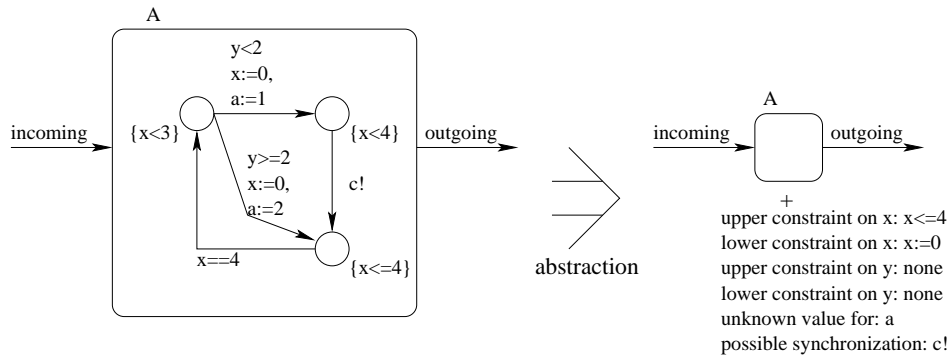


Figure 5.6: The idea of the abstraction for HTA.

Abstracting the nested automaton of A results in a basic location A with the additional information: $0 \geq x \leq 4$, $y \leq \infty$, a is unknown, and $c!$ is possible. When A is active, instead of applying a conventional delay operation on the zone (symbolic representation of time), we use the additional information given by the abstraction on the clock constraints. Furthermore, if there are additional constraints (from other locations) then they are also applied. For the variables, successor states will have the *unknown* value for a and the operations on integers take this into account. Finally, when computing channel synchronizations, it is assumed that $c!$ is available.

This abstraction uses the hierarchical structure to compute an over-approximation of the verification and it is based on a set of superlocations to abstract, in this example A . The goal of the over-approximation is to preserve safety properties. We use the notations of Chapter 4 to formalize this abstraction. In particular, we need to characterize the set of variables affected by the *unknown* value, the new operations on zones, and the synchronization.

Notations

We use the notations of Chapter 4 where states of hierarchical automata are of the form (ρ, μ, ν, θ) where ρ describes the active locations, μ the

values of the variables, ν the values of the clocks, and θ the history. $AND(l), XOR(l), BASIC(l)$ are predicates that characterize the type of the locations. η describes the structure of the statecharts ($\eta(l)$ is the set of sublocations of l), η^* is the reflexive transitive closure, and $\eta^{-1}(l)$ returns the parent location of l .

When we describe clock constraints we write $x_i - x_j \bowtie c_{ij}$ where $\bowtie \in \{<, \leq\}$. The cases $i = 0$ and $j = 0$ correspond to the reference clock and we replace it in the examples by 0.

Formalization of the Abstraction

Symbolic States. As we are interested in applying the abstraction in an existing engine using symbolic state representation, we describe this abstraction with symbolic states. Therefore, we define a mapping from the concrete states to the abstract states. The states are symbolic, that is, we use clock constraints to represent sets of clock values. We consider for this purpose the symbolic state $(\rho, \mu, D, \theta) \in Statespace$ where D is the clock constraint, instead of ν previously used in Chapter 4. We use notations and definitions from Section 4.5 and reuse known results on symbolic model-checking.

We recall the basic definition and property of clock constraints [Pet99]. Let A and A' be the solution sets of the clock constraints D and D' :

$$\begin{aligned} A^\uparrow &= \{w + d \mid w \in A \text{ and } d \in \mathbb{R}_+\}, \\ \{x\}A &= \{x[w] \mid w \in A\}, \text{ and} \\ A \wedge A' &= \{w \mid w \in A \text{ and } w \in A'\}, \end{aligned}$$

with $x[w]$ being the assignment that maps the clock x to 0. The *closure property of clock constraints* states that for clock constraints D and D' with solution sets A and A' respectively, there exist clock constraints D_1, D_2, D_3 with solution sets $A^\uparrow, \{x\}A, A \wedge A'$ respectively. This allows us to define operations on clock constraints directly and avoid to use the solution set. We will use clock constraints in the form $\{x_i - x_j \bowtie c_{ij}\}$ because this corresponds to standard implementations that use DBMs.

In the following we describe how to compute abstract states. We characterize the abstract locations that define the abstraction. We describe the operations we use for the abstracted integer and clock variables.

Locations. The abstraction is based on a set of superlocations. We replace these superlocations by basic locations associated with extra information that describes which side-effects the superlocation may have on the rest of

the system. To identify these locations, we define the function $\mathcal{A} : \mathcal{S} \rightarrow \{\mathbf{true}, \mathbf{false}\}$ that returns **true** if the superlocation is abstracted and **false** otherwise. In practice the user marks a set $\widehat{\mathcal{S}}$ of superlocations and all its sublocations as abstracted, which is, for $l \in \mathcal{S}$, $\mathcal{A}(l) = \mathbf{true}$ if $l \in \widehat{\mathcal{S}}$, **false** otherwise. $\widehat{\mathcal{S}}$ must be well defined in the sense that (i) all sublocations of abstracted superlocations are also abstracted and (ii) a basic location is abstracted only if its parent location is abstracted:

$$\begin{aligned} (i) s \in \widehat{\mathcal{S}} &\Rightarrow \eta^*(s) \subseteq \widehat{\mathcal{S}} \\ (ii) s \in \widehat{\mathcal{S}} \wedge \mathit{BASIC}(s) &\Rightarrow \eta^{-1}(s) \in \widehat{\mathcal{S}} \end{aligned}$$

We note the *minimal* subset of abstracted locations $\widehat{\mathcal{S}}_m = \widehat{\mathcal{S}} \setminus \{l \in \widehat{\mathcal{S}} \mid \eta^{-1}(l) \notin \widehat{\mathcal{S}}\}$. These are the superlocations defined as abstract at the highest level of the location hierarchy.

Integer Variables. When we abstract a superlocation and replace it with a basic location, we have to compute the possible side-effects the original nested automaton has on the system. The abstraction checks if variables are changed and assigned them to the special value *unknown* to represent these changes. We define the function $\mathit{VarSet}_\rho : 2^{\mathit{Var}} \rightarrow 2^{\mathit{Var}}$ to identify the subset of variables *possibly modified* in such active superlocations for a given ρ (ρ gives the locations in a state). A variable is *possibly modified* in a superlocation if an expression assigns any value to it on an edge inside that superlocation. This is written: for $l \in \widehat{\mathcal{S}}$, $\mathit{ModifVar}(l)$ is the set of variables possibly modified in $\eta^*(l)$ over \mathcal{S} . VarSet_ρ is then given by:

$$\mathit{VarSet}_\rho(V) = \{v \in V \mid \rho(\eta^{-1}(l)) = l \wedge v \in \mathit{ModifVar}(l)\}.$$

In this definition, ρ can be used for the concrete or the abstract model. We remind that $\rho(\eta^{-1}(l)) = l$ tests if l is active. In the following, V will be the subset of variables relevant for the abstraction. This subset is $\widehat{\mathit{Var}} = \mathit{Var} \setminus \{v \in \mathit{Var} \mid \exists l \in \widehat{\mathcal{S}}. v \in \mathit{Var}(l)\}$: we take the set of variables from which we remove the particular variables belonging to abstracted locations.

Similarly, we use $\mathit{ModifClk}_\rho$ and $\widehat{\mathit{Clocks}}$ for clocks to identify the set of possible modified clocks and the subset of used clocks in the abstracted model. The expressions we are interested in for clocks are the clock resets $x := 0$ for a clock x . We also use $\mathit{ChanSync}_\rho$ and $\widehat{\mathit{Chan}}$ to identify the set of abstracted channel synchronizations and the set of used channels in the abstracted model. The expressions we are interested in for channel synchronization are of the form $c!$ or $c?$ for a channel c .

Arithmetic Operations on the Abstract Domain. As shown in Figure 5.6, we record possible side-effects on variables with a special \mathcal{U} value. We need to modify the semantics of arithmetic operations on the abstract model to incorporate this special value:

- Binary operators: $x \oplus y = \mathcal{U}$ if $x = \mathcal{U} \vee y = \mathcal{U}$ for x, y integer variables and \oplus any binary operator. We have two exceptions: $\mathbf{true} \vee x = \mathbf{true}$ and $\mathbf{false} \wedge x = \mathbf{false}$ for any value of x .
- Unary operators: $\ominus x = \mathcal{U}$ if $x = \mathcal{U}$ for any unary operator.
- Conditional expressions: $(c?x : y) = \mathcal{U}$ if $c = \mathcal{U} \vee x = \mathcal{U} \vee y = \mathcal{U}$.

\mathcal{U} is used for integers and booleans. If an expression holds for concrete variables, then it still holds or it is *unknown* if at least one variable is set to \mathcal{U} . Conversely if an expression does not hold for concrete variables, then it does not hold or it is *unknown* if at least one variable is *unknown*. As a particular case, the expression $a \wedge \neg a$ is always false on the concrete domain but if a has the *unknown* value then the expression is valuated to *unknown*.

Clock Variables. As shown in Figure 5.6, we keep clock information and we approximate their values symbolically. When removing a nested automaton we keep track of which clocks may be reset and the maximal upper bound they may reach. Concerning zone manipulation, this gives two new operations that are *stretch-down* denoted \Downarrow and the *stretch-up* denoted \Uparrow . The stretch-down stretches the zone down for some clocks instead of projecting the zone as the reset does. The stretch-up stretches the zone up and then we restrict it to maximal upper bounds with the weakest invariants.

We use the following notations:

$$\begin{aligned} u \leq_C w &\equiv u(x) \leq w(x) \text{ for } x \in C \text{ and } u(y) = w(y) \text{ for } y \notin C \\ u \geq_C w &\equiv u(x) \geq w(x) \text{ for } x \in C \text{ and } u(y) = w(y) \text{ for } y \notin C \end{aligned}$$

Let A be the solution set of a clock constraint D . The operation *stretch-down* \Downarrow on a subset C of clocks is defined as:

$$A^{\Downarrow(C)} = \{u \mid \forall w \in A. u \leq_C w\}.$$

The result of the operation for a DBM D is computed as follows:

$$\begin{aligned} D^{\Downarrow(C)} &= \{x_i - x_j \bowtie c_{ij} \in D \mid x_j \notin C\} \cup \\ &\quad \{x_0 - x_j \leq 0 \mid x_j \in C\} \cup \\ &\quad \{x_i - x_j < \infty \mid x_j \in C \wedge x_i \neq x_j \wedge i \neq 0\}, \end{aligned}$$

where the union of constraints corresponds to their conjunctions. The first term keeps the upper bounds, the second term removes the lower bound, and the third term removes the lower bounds from diagonal constraints. In practice the third term is $x_i - x_j \leq \max(x_i)$ to keep the DBM in a canonical form.

Figure 5.7 shows the difference between the reset and the stretch-down operations. The zone used in this example is characterized by the constraints $\{y \geq 2, y \leq 2, y - x \leq 0, x - y \leq 1, x \geq 1, x \leq 3\}$ where we omit constraints of the form $x_i - x_i \leq 0$. The reset on y results in $\{y \geq 0, y \leq 0, y - x \leq \infty, x - y \leq \infty, x \geq 1, x \leq 3\}$. The stretch-down operation on y is computed as follows: $\{x \geq 1, y - x \leq 0, x \leq 3, y \leq 2\}$ for the first expression, $\{y \geq 0\}$ for the second, and $\{x - y \leq \infty\}$ for the third.

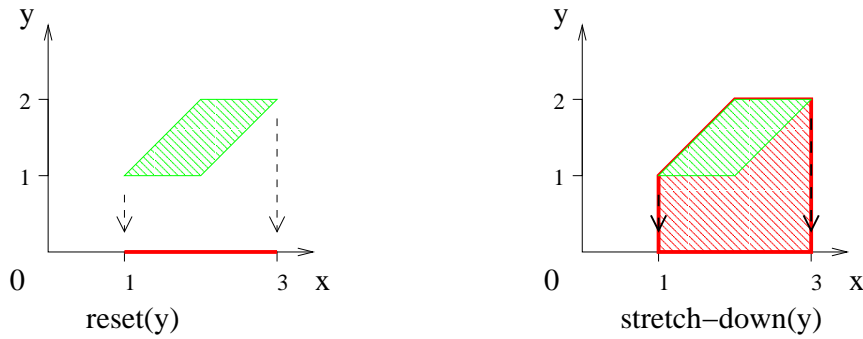


Figure 5.7: Reset and stretch-down operations.

The wanted properties that come directly from the definition are for any (x_1, \dots, x_n) :

- (i) $\{x_1, \dots, x_n\}D \subseteq D^{\Downarrow(\{x_1, \dots, x_n\})}$.
- (ii) $D \subseteq D^{\Downarrow(\{x_1, \dots, x_n\})}$.

For the solution set A of a clock constraint D , the *stretch-up* \Uparrow operation is defined as:

$$A^{\Uparrow(C)} = \{u \mid \forall w \in A. u \geq_C w\},$$

which corresponds to the following operation for D as a DBM:

$$D^{\Uparrow(C)} = \{x_i - x_j \bowtie c_{ij} \in D \mid x_i \notin C\} \cup \{x_i - x_j < \infty \mid x_i \in C \wedge x_i \neq x_j\},$$

where the first term keeps the lower bounds and the second term removes the upper bounds. We note that $(D^{\Downarrow(x)})^{\Uparrow(x)} = \text{free}(D, x)$ for a clock x with the *free* operation as defined in [Ben02]. The free operation removes all

constraints for a given clock. Figure 5.8 compares the stretch-up, stretch-down, and the free operations. The stretch-up operation on y is computed as follows: $\{x \geq 1, y \geq 2, x \leq 3, x - y \leq 1\}$ for the first expression, and $\{y - x \leq \infty, y \leq \infty\}$ for the second.

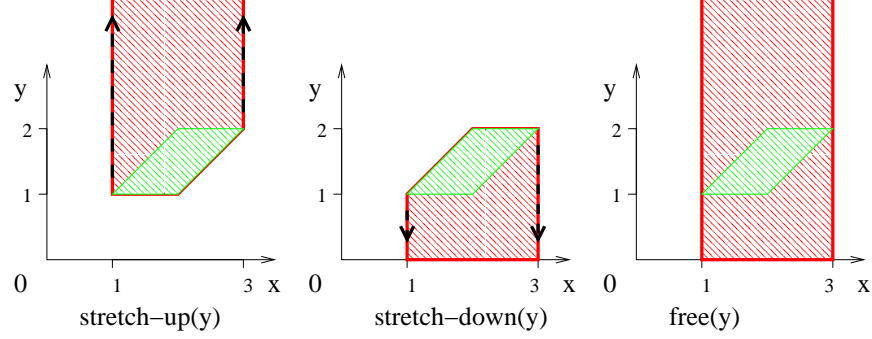


Figure 5.8: \uparrow , \downarrow , and *free*.

The property we obtain is $D \subseteq D^{\uparrow(\{x_1, \dots, x_n\})}$ for any (x_1, \dots, x_n) . In the following we use the notation D_C to restrict the clock constraint D to a subset of clocks C .

Having defined how to stretch up zones we define how to bound them with maximal upper bounds. We compute these maximal upper bound as the weakest invariant $W(s)$ for a superlocation $s \in \mathcal{S}.AND(s) \vee XOR(s)$:

$$W(s) = Inv(s) \vee \bigvee_{l \in \eta(s)} W(l),$$

where we define $Inv(s) \vee Inv(s')$: for $\{x_i - x_j \bowtie c_{ij}\}$ constraints of $Inv(s)$ and $\{x_i - x_j \leq c'_{ij}\}$ constraints of $Inv(s')$:

$$Inv(s) \vee Inv(s') = \{x_i - x_j \bowtie c''_{ij} \mid x_i - x_j \bowtie c_{ij}, x - y \bowtie c'_{ij}, c''_{ij} = \max(c_{ij}, c'_{ij})\}.$$

We consider that Inv is total over the clocks in the sense that it contains $x_i - x_j \leq \infty$ for “free” clock constraints $x_i - x_j \bowtie c_{ij}$. Furthermore, the definition omits details on \bowtie for simplicity, e.g.:

$$\{x - y \leq 3\} \vee \{x - y < 3\} = \{x - y \leq 3\}.$$

We use $W_C(s)$ to restrict W over a set of clocks C . The idea is then to stretch up a zone and then to restrict it the least possible, i.e., with W to keep the over-approximation small and correct.

Abstract States. Our abstraction is based on superlocation: a set of superlocation to be abstracted defines the abstraction. The abstracted variables and clocks are handled with respective arithmetic and symbolic operations that take into account the original behavior. We define the notion of abstract state with respect to a particular set of abstracted location $\widehat{\mathcal{S}}$ to map concrete states to the abstract domain:

Definition 12 (Abstract State) $(\widehat{\rho}, \widehat{\mu}, \widehat{D}, \widehat{\theta})$ is an abstract state of (ρ, μ, D, θ) iff

- (i) $\widehat{\rho} = \rho[\forall l \in \widehat{\mathcal{S}}_m. l \mapsto \emptyset]$.
- (ii) $\widehat{\mu} = \mu[\forall v \in V_\rho. v \mapsto \mathcal{U}]$ with $\text{VarSet}_\rho(\widehat{\text{Var}}) \subseteq V_\rho \subseteq \widehat{\text{Var}}$.
- (iii) $D_{\widehat{\text{Clocks}}} \subseteq \widehat{D}$ such that \widehat{D} satisfies the invariant

$$I_{\widehat{\rho}} \wedge \bigwedge_{s \in \widehat{\mathcal{S}}. \rho(\eta^{-1}(s))=s} W_{\widehat{\text{Clocks}}}(s).$$

- (iv) $\widehat{\theta} = \theta[\forall l \in \widehat{\mathcal{S}}. l \mapsto \emptyset]$. □

The location tree $\widehat{\rho}$ is given by removing abstract locations from ρ . The variable mapping $\widehat{\mu}$ is derived from μ by assigning \mathcal{U} to a subset of variables. It is important to notice that this subset must contain at least the variables that can be changed in the abstracted locations. The abstract zone includes the concrete zone and satisfies the invariant on ρ . The history has its abstracted locations removed. A mapping from concrete states to abstract states is a particular choice for V_ρ and \widehat{D} .

In practice, we choose a particular initial abstract state that corresponds to $V_\rho = \text{VarSet}_\rho(\widehat{\text{Var}})$ for the variables and

$$\widehat{D} = (D_{\widehat{\text{Clocks}}}^{\Downarrow(\text{ModifClk}_\rho)})^{\Uparrow(\widehat{\text{Clocks}})} \wedge I_{\widehat{\rho}} \wedge \bigwedge_{s \in \widehat{\mathcal{S}}. \rho(\eta^{-1}(s))=s} W_{\widehat{\text{Clocks}}}(s)$$

for the zone. This initial zone satisfies the initial invariants and includes the corresponding concrete zone on the concrete domain. We note this particular initial abstract state $\widehat{\mathcal{S}}_0$. The rules are then used to deduce successor states that are abstract states of the concrete ones when the corresponding transitions are taken.

Simulation Relation

To show that our abstraction preserves safety properties, we establish a simulation relation between the concrete and the abstract model. To do so

we have to update the transition rules for the abstract model to take into account the new operations we have defined. We now define the transition relation for the abstract system. The rules of Section 4.5.1 are adapted for the abstract model when an abstract location is active as described in the following.

Action Rule. We add to the action of a transition (i) the variable assignment to *unknown* and (ii) a \Downarrow operation for additional clocks. We define these in terms of update of the previously computed state:

$$(i) \quad \widehat{\mu}' = \widehat{\mu}[\forall v \in \text{ModifVar}_{\widehat{\rho}}(\widehat{\text{Var}}).v \mapsto \mathcal{U}].$$

$$(ii) \quad \widehat{D}' = \widehat{D}^{\Downarrow(\text{ModifClk}_{\widehat{\rho}})\Uparrow(\widehat{\text{Clocks}})} \wedge I_{\widehat{\rho}} \wedge W_{\widehat{\rho}},$$

where $W_{\widehat{\rho}} = \bigwedge_{s \in \widehat{\mathcal{S}}. \rho(\eta^{-1}(s))=s} W_{\widehat{\text{Clocks}}}(s)$. We update \widehat{D} with the stretch-up and stretch-down operations to enlarge the zone and apply the following constraints: $I_{\widehat{\rho}}$, the current invariant, and $W_{\widehat{\rho}}$, the weakest invariant of the abstracted superlocations. For states where the locations are not abstracted the \Uparrow operation is not necessary and is skipped. For these states, the additional constraints $W_{\widehat{\text{Clocks}}}(s)$ will be empty. In addition an edge is said to be enabled if its guard evaluates to **true** or \mathcal{U} .

Delay Rule. We enforce the invariant term with information coming from the removed automata. We rewrite the rule symbolically as:

$$\frac{\text{Inv}'(\widehat{\rho}, \widehat{D}^{\uparrow^d}) \quad \neg \text{UrgentEnabled}(\widehat{\rho}, \widehat{\mu}, \widehat{D}^{\uparrow^d})}{(\widehat{\rho}, \widehat{\mu}, \widehat{D}, \widehat{\theta}) \xrightarrow{\uparrow^d} (\widehat{\rho}, \widehat{\mu}, \widehat{D}^{\uparrow^d}, \widehat{\theta})} \text{delay},$$

where \uparrow^d is the symbolic delay operation bounded by d and $\text{Inv}'(\widehat{\rho}, \widehat{D}^{\uparrow})$ stands for

$$\text{Inv}(\widehat{\rho}, \widehat{\nu} + d) \wedge \bigwedge_{s \in \widehat{\mathcal{S}}. \widehat{\rho}(\eta^{-1}(s))=s} W_{\widehat{\text{Clocks}}}(s).$$

Sync Rule. We use the previous *sync* rule and we add another rule to enable the abstracted synchronizations. An edge guarded by a synchronization can be taken in the abstract model if its matching channel is present in $\text{ChanSync}_{\widehat{\rho}}$. c is used for $c!$ or $c?$ and \bar{c} is the matching $c?$ and $c!$ respectively.

$$\frac{\text{EdgeEnabled}(t, \widehat{\rho}, \widehat{\mu}, \widehat{\nu}) \quad \bar{c} \in \text{ChanSync}_{\widehat{\rho}}(\widehat{\text{Chan}})}{(\widehat{\rho}, \widehat{\mu}, \widehat{\nu}, \widehat{\theta}) \xrightarrow{t} \mathcal{T}_t(\widehat{\rho}, \widehat{\mu}, \widehat{\nu}, \widehat{\theta})} \text{Async},$$

where $t = S \xrightarrow{g, c, r, u} S'$.

Urgency. If we consider a state and an abstraction of it, we have by definition of the abstract operations the property: guard of concrete edge satisfied implies guard of corresponding abstract edge satisfied, given that the abstract edge was not abstracted. The same holds for the invariant conditions since it is relaxed in the abstract model. When considering synchronization this holds too.

Now if we take into account urgency, some delays might be disabled. If we examine Figure 5.9 urgency may conflict in the abstract model. Let us consider the concrete model with S having a nested automaton and the abstract model without the nested automaton. It is possible in $S.w$ to wait. If the state is $S.A$ and g is disabled then it is possible to wait too. However, if g is enabled, it is not possible to wait due to urgency. The abstract model has $c?$ always enabled if S is active so we need to disable urgency when urgency may be enabled by some automaton that has been abstracted.

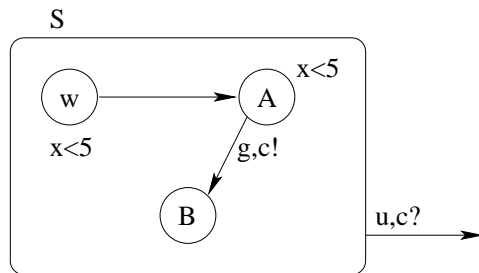


Figure 5.9: Urgency conflicts.

Let us consider what our rules state for this case. The delay rule is enabled if there is a pair of edges with $c!$ and $c?$ enabled. For the abstract case there is no such pair so the delay will be enabled. If there is such a pair not coming from the abstracted automaton there will be no conflict in terms of allowed delays. The *Async* rule will enable the edge as it should be and if there is another edge (not abstracted) the *sync* rule may enable it too.

In conclusion, if any abstracted edge may enable urgency in the concrete model, this urgency is always disabled in the abstract model: if no synchronization is used then the edge disappears ; if synchronization is used then it is ignored by the *delay* rule since the edge disappears as well. We have then the property that any delay enabled in the concrete model will be enabled in the abstract model for a state and any of its abstractions.

Theorem 1 (Simulation) $\forall s \in \text{Statespace}, s$ is simulated by any of its abstract states. We note \widehat{s} one of them: $\forall \alpha$, whenever $s \xrightarrow{\alpha} s'$, then $\widehat{s} \xrightarrow{\widehat{\alpha}} \widehat{s}'$ and $s' \triangleleft \widehat{s}'$. \square

Proof of Theorem 1. Let s be any concrete state and \widehat{s} an abstract state of s . For any transition $s \xrightarrow{\alpha} s'$, it is possible to take the corresponding transition $\widehat{s} \xrightarrow{\widehat{\alpha}} \widehat{s}'$ on the abstract domain by executing the corresponding edges if they are not abstracted, or a τ action if they are abstracted. In particular, the *Async* rule takes care of synchronizations where one of the two edges involved is abstracted. Concerning the delay, the abstract delay rule is weaker than the concrete one, which allows delays of at least the same duration. By definition, the obtained state is an abstract state of s' . \square

Corollary 1 The abstracted model is a safe over-approximation of the concrete model: $S_0 \triangleleft \widehat{S}_0$. \square

Corollary 2 For safety properties ϕ : $\widehat{S}_0 \models \phi \Rightarrow S_0 \models \phi$. \square

Particular Cases. We examine what happens in particular delay/reset cases. Figure 5.10 shows what happens in two scenarios. In the first scenario, we start from a given zone (1) resulting from a transition to a state with the location $S.a$ active. The upper bound on the clock y depends on the invariant on S . We apply delay bounded by the invariant (2). Then we can take a transition taking the edge that leads to $S.B$ and we delay again (3). These are transitions involving internal edges of S . Let us consider what the abstract model (abstracting S) does: y may be reset so $\text{stretch-down}(y)$ is applied. There is no bound on x and y has $y < 2$ so stretch-up is bounded only for y . For the further abstracted internal actions, the obtained zone (4) includes (1),(2), and (3). In the second scenario an external edge leading to C is taken, thus having an external reset. The reset (5) is then applied on the abstract model normally. However, the stretch-up applied by the *action* rule restores the zone (6) as an over-approximation. Further internal actions are still approximated.

Application. The application of the abstraction is to be able to deduce safety properties for the concrete model from the abstract model. With the same notation as Definition 6:

$$\frac{M \triangleleft A(M) \quad A(M) \models \forall \square \phi}{M \models \forall \square \phi}$$

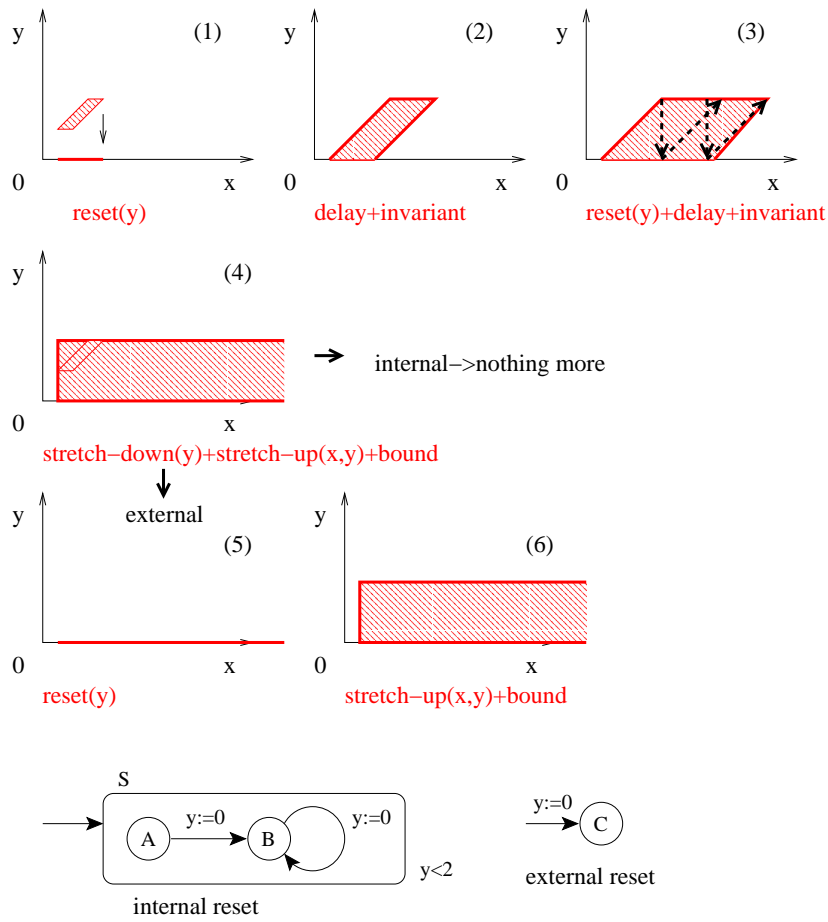


Figure 5.10: Resets and delays for concrete and abstract models.

if a model M is simulated by an abstract model $A(M)$ and $A(M)$ satisfies a property of the form $\forall \square \phi$, then M also satisfies the same property. ϕ is property that characterizes states, it has no TCTL quantifier.

5.2.3 Spurious Traces

A spurious trace is a trace obtained from the abstract model that does not have a corresponding valid trace for the concrete model. The only way to confirm a (counter-example) trace is to try to generate the corresponding concrete trace, which is, to use the reachability algorithm on the concrete model. To reduce the cost of a such check the abstract trace can be used to guide the reachability. For this purpose we propose the notion of guided reachability with respect to an abstract trace:

Definition 13 (Guided Reachability) The transition $S \xrightarrow{\alpha} S'$ may be chosen if $\widehat{S} \xrightarrow{\widehat{\alpha}} \widehat{S}'$, which corresponds to try to simulate the abstract model with the concrete one. \square

Unfortunately this could be as expensive as the original reachability problem on the concrete model if the abstracted part of the model is the main source of states in the statespace. However, this should not happen in practice because the abstraction are localized and models usually follow some logic, which gives order and dependency between different abstracted/non-abstracted parts of the model.

Another approach we used in Chapter 2 is to use another kind of model reduction such that the reduced model is simulated by the concrete one: $R(M) \triangleleft M$ as for the case study (under-approximation in this case), instead of $M \triangleleft A(M)$ for the abstraction we defined here. Such a reduction is able to check for paths in the reduced model that are guaranteed to exist in the concrete model. However, if a path is not found in $R(M)$, the problem of the spurious trace remains because this path may still exist in M . This approach is useful to confirm traces that are “likely” to exist in the concrete model, where “likely” is judged by the modeler.

5.2.4 Implementation

The implementation of our abstraction technique described in Section 5.2 consists of two parts, namely static and dynamic analysis. The static analysis is done when parsing the model to collect static information related to the structure of the model. The dynamic analysis is based on the abstract semantics.

Static Analysis: Collect Information

In this phase superlocations are equipped with a record of their nested actions and synchronizations. This record is made of the lists of assigned variables, reset clocks, weakest invariants, and possible synchronizations. The assigned variables are only those that do not belong to the superlocations since they are completely removed in the actual instance. The same holds for the reset clocks. The weakest invariants are computed for all the non-local clocks for all locations where delay is possible, i.e., non urgent and non committed locations. All the possible non-local channel synchronizations are gathered.

The instance is built with the superlocations as basic locations equipped with this extra information instead of a nested automaton. The problems that do not appear here come from all the parameterized names (variables, channels, clocks) that must be mapped without creating the instance of the nested automaton.

Dynamic Analysis: Reachability

The successor computation uses the pool of possible synchronizations attached to abstracted locations to match for channel synchronization expressions, i.e., $c?$ matches $c!$. The synchronization is recognized although an edge may be missing syntactically, as described in the semantics.

The variables present in the set of possibly modified variables are set to a special *unknown* value when locations that have such sets are entered. The operations on variables are updated to include the abstract semantics to manipulate them. This is done in an extension of the graft/cut filter for simplicity.

The clocks present in the set of possibly reset clocks are subject to the \Downarrow operator when locations having such sets are entered. This is done along with the variable reset to *unknown*. The \Uparrow operation is done along with the delay \uparrow operation, i.e., in the delay filter.

There is no need to do more for the location vector because the nested automata instances simply do not exist.

Experimental Results

We applied the abstraction to the bus coupler model of the case study of Chapter 2. The results show that the abstraction does not work as one could expect in every case. We analyze the limits of the abstraction.

Limits of Abstraction. The abstraction we use is based on the hierarchical structure of a model. A superlocation is abstracted and information concerning internal variable assignments and clock resets/delays is added to the abstracted model. This information allows execution of the abstract semantics.

In the best case, the abstraction simplifies greatly the exploration since potentially many locations, and more importantly cross-products of these with other automata, are replaced by one (abstract) location. Furthermore, some variables are set to an *unknown* value that does not propagate “too far” and does not introduce chaos in the model. Concerning the clocks, the abstract zone always includes the non-abstract zone.

In the worst case, removing locations will introduce chaos and break the order of the automaton. This is done via the side-effects on the variables that are used for further computation. The *unknown* value may propagate and produce many combinations with other known values. Furthermore, much more transitions may be enabled, allowing greater freedom in the abstracted model. This is disastrous if the size of the non-abstracted automata is large, as it is the case in the bus coupler model. If the abstracted locations have synchronizations with the rest of the model that control order, then this is guaranteed to bring chaos to the model since these synchronizations will always be enabled.

Limitation from the Model-Checker. Apart from the reasons coming purely from the model, there is another reason coming from the model-checker that may make the abstraction fail. Let us consider Figure 5.11. It represents four states A1, A2, A3, and A4, with possible values for the variables a, b, c . The state A1 does not contain abstract locations and has a given known variable vector. The states A2, A3, and A4 contain an abstract location and have different unknown sets. After taking a transition to B, we obtain different states where the only difference comes from the values of these variables. Now comes a problem similar to the zone inclusion of the symbolic part of the states: if we have a variable set to *unknown* (U in the figure), then this case contains all other possible cases. We have similarly for zone inclusions, $\langle 1, 2, 3 \rangle \subseteq \langle U, 2, 3 \rangle \subseteq \langle U, U, 3 \rangle \subseteq \langle U, U, U \rangle$ so it is necessary to explore only the case with $\langle U, U, U \rangle$. However, we have a limit in the engine at the present time because the discrete part of a state is used to compute a hash value to quickly find the state and then apply inclusion checking on the zones, see chapter 3. Limiting the hash to the control location part only and then performing an inclusion checking on the

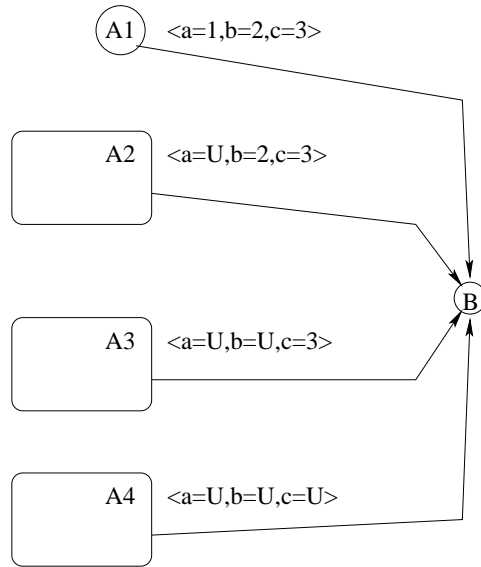


Figure 5.11: How abstraction may break down.

variables requires much more changes and results in a huge performance hit. The performance hit is explained by the sharing results we have from 3.3: we know that given one control location, there are many possible variable combinations.

Generation of the Statespace

Figure 5.12 shows the structure of the bus coupler model. The root is named **system**. We have the tasks for the couplers for the ports 1 and 4, along with the field interface automata. The figure is basically the same as in Figure 2.12 with numbered *XOR* locations. The **FI_{xx}** automata are rather complex (24 locations each), the sublocations of the couplers are simpler (between 3 and 17 locations). The semaphores have 2 or 4 locations. The channel synchronizations follow the communication given in Figure 2.12.

As there are 4096 ways to configure the systems with abstract/non-abstract locations we limit ourselves to a few interesting cases only. Table 5.3 shows the different results obtained. We conduct the experiments on a Celeron 633MHz equipped with 192MB of memory under Linux 2.4.18. We give the abstracted locations in the table. An empty case means no abstraction. We classify the different locations as semaphores, FI, and ports. Furthermore, to classify how the statespace explodes we give for the possible

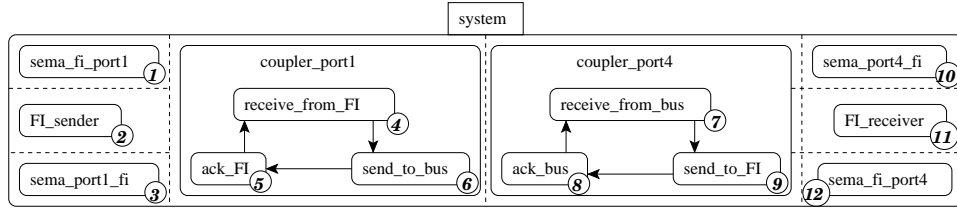


Figure 5.12: Bus coupler model structure and abstractions.

cases the (approximate) maximal size of the waiting list where it stabilized. The property checked is $A[] \text{ true}$ to generate the whole statespace.

Setup	Semaphores	FI	Port 1	Port 4	Consumed	Waiting
1					99s/86M	13000
2	1,3				199s/142M	40000
3	10,12				61s/45M	40000
4	1,3,10,12				155s/70M	40000
5		2,11			300s/66M	70000
6			4,5,6		<i>Stopped</i>	-
7				7,8,9	<i>Stopped</i>	-
8	1,3	2			10s/14M	2700
9	1,3		4,5,6		48s/49M	17000
10	1,3	2	4,5,6		2s/6M	600
11	10,12	11			3s/9M	700
12	10,12			7,8,9	22s/29M	10000
13	10,12	11		7,8,9	<1s/5M	100
14	1,3,10,12	2,11			<1s/4M	100
15	1,3,10,12		4,5,6	7,8,9	7s/9M	5600

Table 5.3: Abstraction configurations and results. *Stopped* means we stopped the model-checker manually.

The model has all the “bad” characteristics for the abstraction: it has only shared variables, the side-effects propagate, the different superlocations synchronize much with each other, and the size of the different locations is such that it is favorable to explosions due to introduced chaos in the behavior of one component. The base reference for the analysis is the setup (1), the full model without any superlocation abstracted.

(2) and (3) abstract the semaphores on the port 1 and the port 4 respectively. The waiting list size shows the explosion of the successors due

to the removal of synchronization. (3) explodes less than (2) because it has no acknowledgment back, which give a lighter synchronization dependency. As the components rely heavily on semaphores, removing them only gives a bad configuration. The case (4) is in between (2) and (3), as expected.

(5) abstracts only the FI parts. As the behavior of the couplers depends on the FI, and because of the shared buffer they use to communicate, this configuration creates more chaos. The successor computation explodes but not the statespace because the propagation of *unknown* stabilizes quickly.

(6) and (7) are the worst cases. These abstract the core of the couplers, so removing them leaves the FI on both sides completely free with their respective semaphores totally free as well. The semaphore counters are free to combine and the *unknown* variables are free to combine with them too. In short the couplers regulate the synchronization and the order of the whole model.

(8) and (11) abstract the FI and the corresponding semaphores. This is a good configuration where a coupler sees the FI with the synchronizations as a black box. (11) is substantially faster as expected from the comparison between (2) and (3).

(9) and (12) abstract the couplers and the corresponding semaphores. This is a good configuration where a FI sees the coupler with the synchronization as a black box. The gain compared to the full model is explained by the results obtained from (6) and (7). We still see with the help of the maximal size of the waiting lists that the couplers are important in controlling the order of the model.

(10) and (13) correspond to the complete abstraction of one port. This is a logical abstraction where one communicating side sees the other as a black box. As expected, this is a good configuration.

(14) abstracts both FIs and combines the gains of (8) and (11). This allows us to focus on the communication between the couplers only and forget the rest of the system.

(15) abstracts both couplers and combines the gains of (9) and (12). This allows us to focus on the FIs alone. The communication from one FI to the other goes through a black box so at this level of abstraction it is loose.

Verification of the Properties. The analysis of the statespace shows how the models explode or are reduced. We study the six main correctness properties checked in Section 2.3.2. It is important to see how the abstraction behaves now with respect to important properties of the model.

Table 5.4 shows the results for these properties on the different configurations. The safety properties ($A \parallel \phi$) marked **yes** are satisfied and those marked **no** are not. Properties marked **X** correspond to those that refer to abstracted locations and therefore can not be checked.

Setup	Safety 1	Safety 2	Safety 3	Safety 4	Safety 5	Safety 6
1	yes	yes	yes	yes	yes	yes
2	yes	no	no	no	no	no
3	yes	yes	yes	yes	yes	no
4	yes	no	no	no	no	no
5	X	yes	yes	yes	yes	X
8	X	yes	yes	yes	yes	yes
9	yes	X	X	yes	yes	yes
10	X	X	X	yes	yes	yes
11	yes	yes	yes	yes	yes	X
12	yes	yes	yes	X	X	yes
13	yes	yes	yes	X	X	X
14	X	yes	yes	yes	yes	X
15	yes	X	X	X	X	yes

Table 5.4: Verification results on the abstract models.

The properties hold except for the configurations (2) and (4). (2) corresponds to the case where the FI and the coupler of port 1 are unsynchronized. As the protocol is supposed to synchronize these components, it is normal that invalid values reach the coupler. The fact that properties 3 to 6 are violated comes from the propagation of this invalid value. (4) is similar. In the configuration (3) the last property does not hold for the same reason as for (2): the coupler and the FI are unsynchronized so invalid values may be sent.

It is interesting to note that if we have good configurations, where the synchronizing semaphores are abstracted with one side (coupler or FI), then the properties hold.

Applying Abstraction. Considering the results of our abstraction on the bus coupler case study, we can now define how to apply successfully this abstraction. This abstraction is suited for models with limited side-effects on variables: a structured model with local variables gives good results. The models should also have limited synchronization between components. For this class of model, a good abstraction configuration is given by the

logical structure of the model. As the bus coupler case study shows, if one component relies heavily on another for communication, it is logical to abstract them both. Abstraction on blocks logically connected together works well because the propagation of the (potential) introduced chaos is contained.

Pacemaker Example

Although the hierarchical model performs already well for the pacemaker, we tried the abstraction on the pacemaker. The verification is faster and consumes slightly less memory. However, for such an example the important property that checks for the heart beat is never satisfied. This comes from the fact that this example is time sensitive and approximating on time incurs a small error enough to trigger bad behaviors. Our experiments included abstractions for the pacemaker superlocations AVI, VVT, and VVI.

5.3 Conclusion

We define and formalize a new abstraction based on our hierarchical timed automaton. This abstraction allows over-approximated analysis of models and is appropriate for models that are too large for direct analysis. Its implementation gives good results for appropriate abstraction configurations.

We discussed some of the issues in implementing an engine supporting hierarchical timed automata. Our current prototype outperforms the non-hierarchical UPPAAL when it checks hierarchical models while UPPAAL checks their corresponding flattened models.

UPPAAL code is changing at a fast pace and we are integrating new algorithms and structures. In particular the tree representation is more compact. This work is in progress for the next major release of UPPAAL since basic structures are changed.

Chapter 6

A New Tool Architecture

Based on the theory of timed automata [AD94] a number of tools have been developed such as UPPAAL [LPY97], Kronos [Yov97], or RED [Wan01]. Other than timed automata, there are other tools in the model-checking arena such as HyTech [HHWT97, AHH96], Murphi [DDHY92], and Spin [Hol91, Hol97]. Every tool has its specific characteristics and focuses on a particular aspect: the timed automata are well suited to handle timed models where clocks are appropriate models time, the hybrid tools are suited for models where real-valued variables that model continuous environment parameters are important, and the discrete tools are best at verifying models written in a formalism close to programming languages or even programs written in C [HS00].

Various techniques and algorithms have been developed to improve model-checking for these tools such as approximate analysis [WT94], compact data structures [LLPY97], clock reductions [DY96], symmetry reduction [ID96, ID93], partial order reductions [BJLY98, Min99], distributed model-checking [BHV00], and many others. Whereas publications are rich in algorithm descriptions, theoretical results, and experimental results, there has been little information on how these optimizations fit together into a common efficient architecture. Implementation of algorithms is as important as the algorithms themselves and this is the reason why we present our architecture.

In this chapter we present the architecture of the UPPAAL tool that can be used more generally in any model-checker. It is a flexible architecture that is appropriate for research experimentation by allowing multiple orthogonal features to be tested. We explain the implementation of UPPAAL and how it can be extended to accommodate new features.

6.1 A Pipeline Architecture for UPPAAL

The first UPPAAL version was released in 1995 [BLL⁺95] and it was then the first verification tool for timed automata where the system could be modeled graphically. In 1999 the architecture was changed to a client-server, see Figure 1.5. Finally, in the past two years the internal structure has been changed around a data flow-centric architecture. We remind that the UPPAAL tool is the result of a team work and many people have been involved in its development.

We present the pipeline architecture and the parser library that accepts the input language. Then we discuss the development of the tool.

6.1.1 Implementation

The main design goals of the engine are flexibility and efficiency. It should be easy to integrate several features for experimenting on different algorithms and it should be fast enough to deal with large models. Most often, fast enough means to consume as little memory as possible because the real bottleneck of computers is the memory system. The older engine was a more or less direct implementation of the reachability algorithm given in Figure 3.1. Such an implementation has several disadvantages:

- The implementation became more and more complicated as new options were added.
- The checks for options were done in the main exploration loop, slowing the verification and cluttering the code.
- Experimental extensions and structures required major changes due to new algorithms.
- Maintenance was difficult because of the mixture of different options.
- Options that are orthogonal in theory, i.e., used in conjunction with each other, could not be combined together orthogonally in practice.

The architecture was then restructured as a pipeline, idea borrowed from computer graphics. The pipeline incarnates the natural data flow of the reachability algorithm and its basic blocks are reused for the liveness algorithm. The architecture is shown in Figure 6.1. Following the terminology for pipeline architecture, the different components are either *filters* or *buffers*, represented as boxes in the figure. These components are detailed in the next section. A filter has a *put* method to receive data. The processed data is then sent to the next component. A buffer is a passive component receiving data with a *put* method and offering data with a *get* method. A

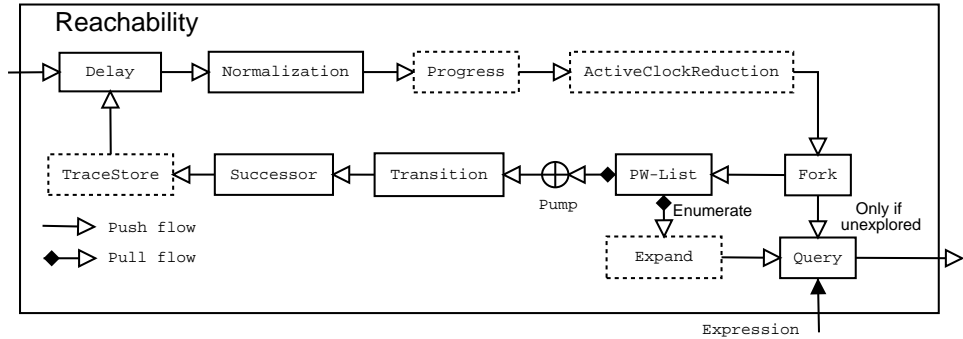


Figure 6.1: The pipeline architecture of UPPAAL. Transitions are generated from the states provided by the `PW-List` buffer. The successor states are then computed and their trace is optionally kept. Delay is applied to these states and the normalization on the zones is applied. Optionally the progress of the verification is displayed before applying the active clock reduction algorithm. If the resulting states are unexplored states then the `Query` filter checks properties on them. The `Expand` filter serves to reuse previously explored states when checking several properties.

pump controls the data flow by getting states from the buffer and injecting them to the transition generator. The pipeline is a data pipeline and involves no concurrency in contrast with common pipeline design seen in processors. The design may accommodate concurrency but we believe it is inefficient due to the amount of data flowing between the different concurrent units and due to load balancing issues.

The benefits of using a common filter and buffer interface are *flexibility*, *code reuse*, and *efficiency*. The flexibility comes from the possibility to exchange a component for another to test different algorithms. Furthermore, the pipeline can be configured at runtime to use a particular component instead of another that implements a given algorithm. Such a configuration allows us to skip completely some stages in the pipeline if they are not necessary. The code reuse comes from the reuse of the components inside other building blocks. For instance the `Successor` filter is used by the reachability checker, the liveness checker, the deadlock checker, the `Expand` filter, and the trace generator.

The programming language used to implement UPPAAL is C++ [Str97]. This object-oriented language fits perfectly to the pipeline design. Figure 6.2 shows the class diagram for a few important classes. The classes are in fact C++ templates and are typed for the different data flowing through the

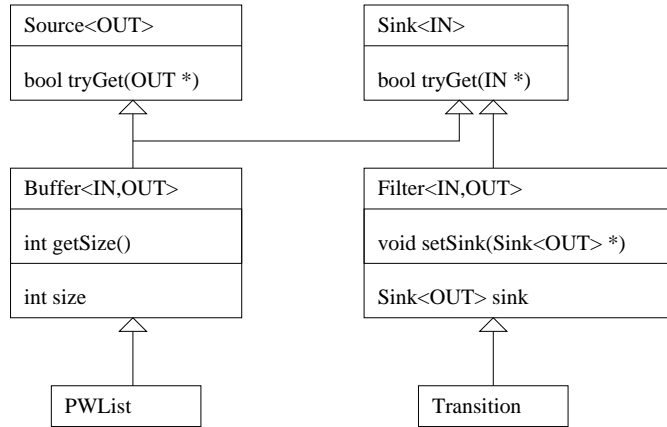


Figure 6.2: Class diagram for the pipeline.

different components. The **Source** is defined to generate a type **OUT** and the **Sink** to receive a type **IN**. The **Buffer** receives and offers data. It is naturally a sink and a source. The **Filter** receives data of type **IN** so it is a sink and it forward the processed data to the next sink (generally a filter too, but may be a buffer). The next sink receives the output type **OUT** from the filter. The filter has a `setSink` method to configure the pipeline. The buffer has a `getSize` method to check for its size, generally it is the length of the waiting list. The **PW-List** component as described in Section 3.1.1 is a (typed) buffer. The **Transition** component is a filter.

Two noticeable features of the sink and the source interface definitions are their respective `tryPut` and `tryGet` methods. These methods return booleans as results, which allows us to use conditional components such as the **Fork** component. A `tryGet` on the buffer may fail if the waiting list is empty, in which case the argument is not written and the reachability terminates. A `tryPut` to the buffer may fail if the state has already been explored, in which case it is discarded.

Technically, these interfaces are virtual, which incurs a performance penalty in C++ because the call of such methods must be resolved dynamically (table lookup). However, it is largely offset by the benefits we named previously. The performance hit is negligible. It even improves performance in the cases where a component is skipped. It also alleviates the need to allocate memory temporary since the stack is used. For example, if we consider the successor generator component **Successor**, the older implementation had to store the different successor states in a vector and then process them.

Now they are sent through the pipeline one by one, as they are computed on-the-fly.

6.1.2 Typed Data Flow

We go now through the reachability pipeline and presents the different data types used. In particular we describe how we limit copying of data.

GlobalStateReference

States are pumped out from the **PW-List** component to compute the successor states. The type of the component is actually a *GlobalStateReference* (GSR). The GSR is an interface whose implementation depends on the kind of PW-List one uses. The GSR is used for two purposes: (1) to reduce copy of data, and (2) to abstract from data representation. The interface has two methods:

```
void writeDiscrete(GlobalState *) and  
void writeSymbolic(GlobalState *).
```

The state reference itself is not a copy of a state. It is a light-weight interface used to copy only the needed data when necessary. When computing transitions, the guards of the edges are evaluated. To do this, the discrete part has to be copied first. If the guard is false then there is no need to copy the symbolic data (zone), and we can try another edge without any further copy. If the guard is true then the symbolic part is copied too. We repeat this for every successor. The internal representation of a GSR depends completely on the PW-List and the low level structures used to store actual data. By using this interface it is easy to try any storage algorithm.

Successor

The **Transition** component generates *Successor* data. Those are pairs made of the global state representation (*GlobalState*) and a list of edges to take for a given transition. This is fed to the **Successor** component that executes the actions on the edge(s) and computes the next successor state. The output type is still a **Successor** to keep track of the taken edge(s). Separating the successor computation in two is more flexible and allows structured code. Furthermore, it is easy to change semantics by changing the **Transition** component only, to adopt for example a maximal progress semantics.

The **TraceStore** is an optional component and is used to record traces of states. The **Delay** filter applies the delay operation on zones if delays

are allowed in the given state. The zone is then normalized [Ben02] by the `Normalization` filter and we obtain a *WaitingState*.

WaitingState

A waiting state is a global state with additional information intended to tune the use of the passed list. For some optimizations, like [LLPY97], it is desirable not to store states in the passed list. Those waiting states go through optional filters: the `Progress` filter displays the current progress of the search by printing the size of the current waiting list. The filter `ActiveClockReduction` applies the active clock reduction technique [Yov97]. The waiting state is then sent to the PW-List via a `Fork` component.

The PW-List receives a state and it stores it in some internal representation, which is unknown at this stage of the pipeline. This is important to allow different optimizations based on on sharing, graph reduction [LLPY97], or approximations like hashing [Hol98, WL93] or convex-hull [Bal96]. The PW-List, with respect to its semantics given in Section 3.1.1, may reject states and not store them because they were already explored or accept them. If a state is accepted, it is copied and the original state is available for modifications. If a state is rejected then the pipeline discards it. The `Fork` conditional filter sends the state to the query checker only if the state was accepted, in which case the query checker `Query` may modify the state for evaluation (zone operations are destructive).

The PW-List offers another capability to enumerate previously explored states. This is used to feed the `Expand` filter when several properties are examined. In this case the engine can skip to generate again the whole statespace and compute only parts of it. The `Expand` filter is actually a compound object made of other internal filters. We do not go into further details.

General Design

The pipeline is adapted for the UPPAAL model-checker and is tuned to handle time. However, its principles are general enough to be reused in other model-checkers. The PW-List structure can be used in any model-checker, along with the global state reference interface for flexibility with respect to memory management. The flexibility of the transition and successor filters can be used to carry out different experiments on semantics.

6.1.3 Parser Library

Along with the pipeline, UPPAAL features a separated parser library, distributed under the LGPL license. This library allows other tool implementors to use the TA language as an input language. The library supports the TA, XTA, and XML formats. The library is built on an abstract interface that is independent of any internal representation of the system. It offers one large interface that is automatically called upon parsing. This library allows greater flexibility for UPPAAL and for other tools that adopt the library.

6.2 Extensions

We discuss the possible extensions of UPPAAL based on its current architecture.

6.2.1 Plugin

Semantics Changes

The flexible pipeline structure of the engine allows semantics changes in the form of plug-in component. We are working on defining an interface to change parts of the pipeline dynamically. This means that it is possible to use a module as a plug-in to replace for example the default `Transition` filter. This allows us to turn the engine into a more UML verification engine if channels are handled like events and maximal progress is implemented.

Engine Reuse

The engine itself can be used as a plug-in within other tools. The Times tool¹ is such an example. The Times tool is a schedulability analysis tool and it is using the engine for its reachability analysis capabilities on timed automata.

6.2.2 Hierarchy

Implementation on hierarchical timed automata has been done with this architecture. The implementation is a prototype and we discuss further extensions of the current version. These extensions all respect the architecture of the tool.

¹www.timestool.com

Data Type

To optimize on the internal hierarchical tree representation, special dynamic data types are in development. These data are dynamic in the sense that the set of defined variables and clocks depends on a given current state configuration. This change has no impact on the architecture. Dynamic insertion/removal of clocks and variables are implemented and will be incorporated to the state representation.

Graft/Cut

The graft/cut operations defined in the semantics are currently implemented as a separate filter that processes states, following this way the architecture philosophy. As a side-effect of this implementation, disabling the filter for non-hierarchical models incurs no overhead at all for such models.

PW-List

The PW-List itself does not need to be modified to accommodate for optimized hierarchical representation. However, the state reference needs to be updated to reflect the data type changes.

6.2.3 New Algorithms

We are investigating on an implementation of PW-List that stores the statespace on disk. This is similar to [SD98] but it is defined as a PW-List implementation. This allows for different caching policies to keep only parts of the statespace in main memory and use sequential synchronizations with the disk. We are developing new experimental normalization algorithms for the symbolic zone representation. In practice this means to swap the normalization filter for a new implementation. We may change the way the query filter handles properties to check for several properties at the same time. Again, this kind of changes involves swapping one filter in the pipeline.

6.3 Conclusion

We have presented a new architecture applicable for model-checkers. The architecture is shown to be flexible and adapts to changes and new algorithms. This is an important research means since it allows us to experiment with different algorithms and combine them to study their influences on each other. The development of new theories and algorithms for model-checking

is important but it must be followed by the development of corresponding implementations.

Chapter 7

Conclusions

The original goal of this work was to develop a modeling language and a model checker for timed systems that may contain hierarchical structures. In order to find the limits of UPPAAL and to evaluate the real needs in terms of modeling capabilities, we started with an industrial case study to check an existing product, a fieldbus protocol. The protocol was too complex to be modeled at once and the model-checker could not cope with the huge size of the statespace. The study was carried out on the core of the protocol implementation, which involved 5541 lines on Modula-2 and 151 pages of documentation. We applied abstraction techniques manually to verify the models step-by-step. Right from the start we realized that we had to face the questions: how faithful the models we developed were with respect to the implementation and what we wanted to verify. We constructed the models from the implementation but we had to approximate the behaviors for some parts of it, e.g., the system calls. Upon validation of the models, the engineers were the judges and it was particularly delicate as we were dealing with a real product. Slight miss-interpretations on semaphore synchronizations for example lead to behaviors in the models that were refuted. Furthermore, statements claiming errors were difficult to be accepted and lead to miss-understandings. Indeed sometimes what we call errors were not real bugs in the software and they were tolerable behaviors that could be improved. Apart from this, the engineers enjoyed the tool and its intuitive graphical language.

From this project we realized that we needed to handle structured models. We designed hierarchical extensions to support such models with the goal to implement them in UPPAAL. To check that the language met our expectations we implemented a flattening algorithm (detailed in [Möl02])

to check hierarchical models. We tested it on a pacemaker example. Then we tried to implement it directly in the model-checker. It turned out that firstly, this version of the language was difficult to implement due to its complex semantics and secondly, that the UPPAAL engine was not suitable for it by far. We had then to optimize it first and bring new structures and algorithms into the engine. Then we got rid of the source of the complexity in the hierarchical extensions while keeping sufficient expressivity. This simplified version was implemented successfully and we could experiment with the bus coupler and the pacemaker.

To utilize the hierarchical structure in the engine we have developed an abstraction technique based on the hierarchical language to handle systems for which direct verification failed. The idea was to achieve an automatic abstraction similar to the one we did manually in the case study. The algorithm implemented works but there is still room for improvement. There are also open issues left concerning the exact verification on how to exploit more information from the hierarchy, for example with local search. We have to find the advantage the hierarchical structures could give over the conventional partial-order reduction, which in our case is particularly difficult when applied to timed systems. Simple solutions have proven to work best until now, for example the notion of committed location. This is an important aspect not to be neglected because of efficiency and correctness issues of the practical implementation problems. We intend to continue to improve the tool with this philosophy.

In this project we succeeded in defining and implementing a hierarchical language to model timed systems. The model-checker is able to handle the model using the hierarchical structure natively. We also improved the performance of the current UPPAAL both in speed and memory usage by means of new structures and implemented an automatic abstraction technique. The remaining goal is how to better exploit hierarchy for exact verification. We currently exploit this information only for approximate verification.

Future Work

The tool UPPAAL is the result of a team work contributed by many people from Uppsala and Aalborg universities mainly. There are occasional cooperation with other universities to try new algorithms. The UPPAAL source code has matured and changed a lot since its early beginning in 1995. The future work from now is to rewrite and re-organize the code to set up a new base for the upcoming 8–10 years. The parser library is in good progress. The PW-List library incorporating all the previous optimizations is almost

complete. Furthermore, as we are developing a tool for formal verification, it is also important to ensure its correctness. We are improving the implementation by using software components and better testing for this purpose. Concerning the future work related to this thesis, we are working on new data structures for a better hierarchical engine. These include an optimized tree structure and special management of dynamic data. These structures require a radical change in the UPPAAL engine. This opens further developments based on hierarchical information exploitation.

The next major UPPAAL version, i.e., UPPAAL 4.xx, will incorporate all the features presented in this thesis in an optimized version, plus all the previous optimizations discussed in this thesis. Furthermore, we are working on new experiments such as distributing statespaces on disk and are addressing the ever growing wish list of features for UPPAAL. The most recent features include improvements of the modeling language (C syntax) and the query language.

Chapter 8

Appendix

8.1 Automata Figures of the Case Study

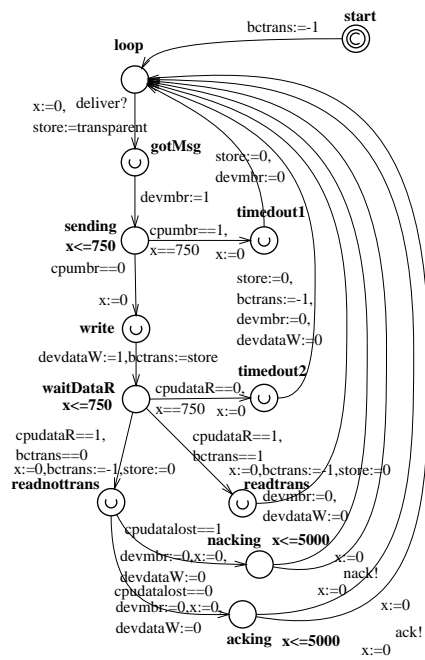


Figure 8.1: The template of the slave coupler.

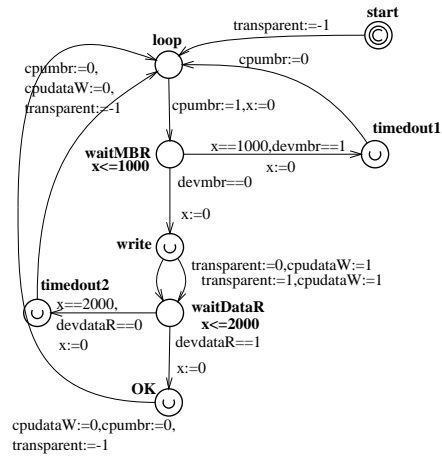


Figure 8.2: The template of the FI master.

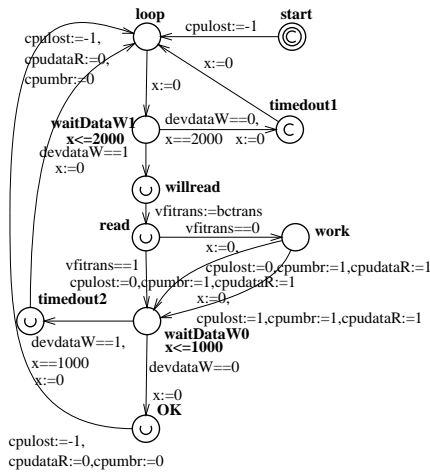


Figure 8.3: The template of the FI slave.

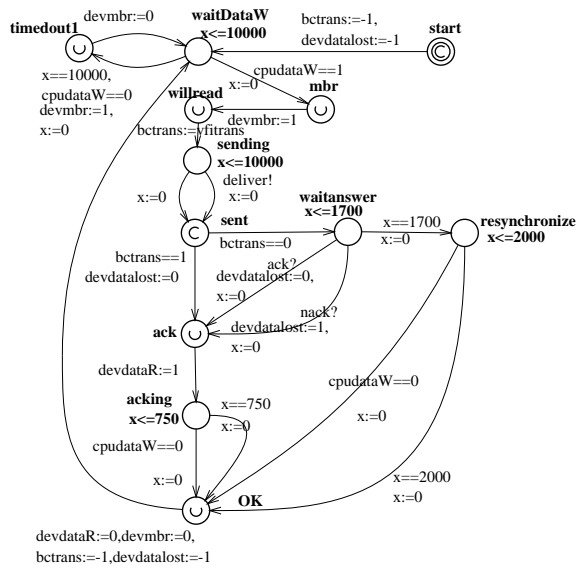


Figure 8.4: The template of the master coupler.

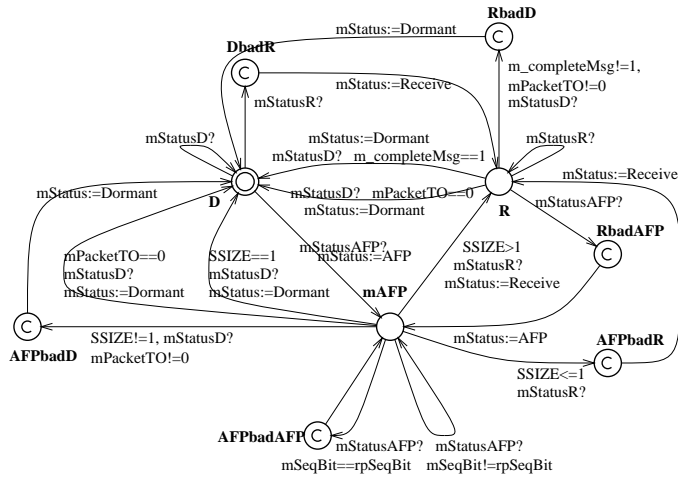


Figure 8.5: Master monitor automaton.

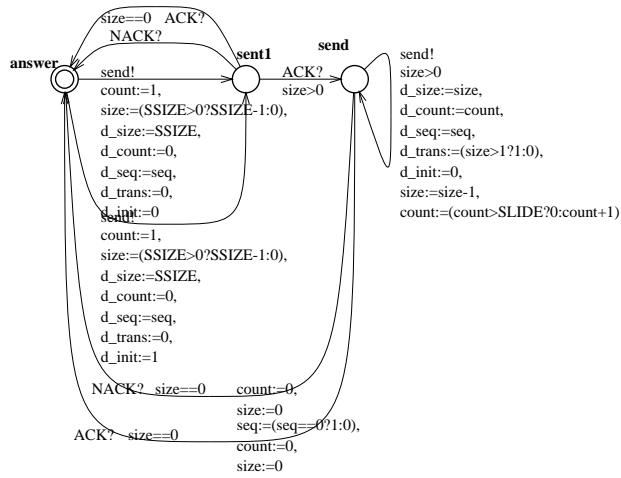


Figure 8.6: Slave test working with the master.

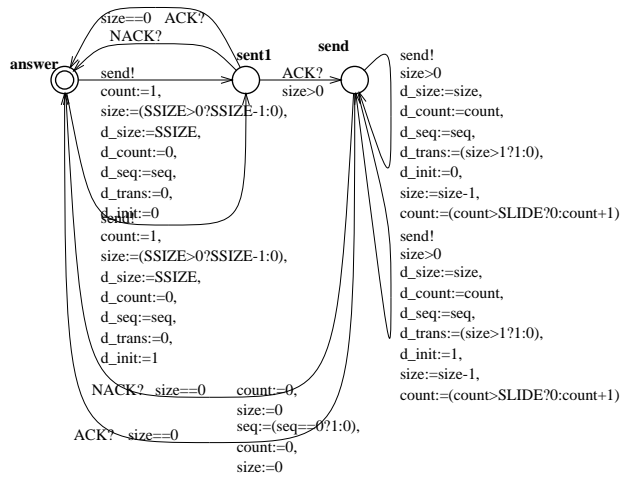


Figure 8.7: Master test working with the slave.

8.2 Grammar of Hierarchical Timed Automata

In the following we present the BNF grammar of the HTA language and the query language as it is used by the verifier UPPAAL. The complete grammar has a compatible alternative set of rules to be backward compatible with the older XTA format. This syntax adopts C style statements. The C implementation is not yet complete: constructs like for-loops are recognized by the parser but not yet implemented by the model-checker. The same applies for the history location.

NAT is a natural number. ID is an identity, a valid name. TYPENAME is a name as well, but used for the type definition. We note them as terminals for simplicity.

<i>HTA</i>	→	<i>Declaration Inst System</i>
<i>Declaration</i>	→	<i>Declaration ADecl</i>
<i>ADecl</i>	→	<i>FunctionDecl VariableDecl</i>
		<i>TypeDecl ProcDecl</i>
<i>Inst</i>	→	<i>Inst ID = ID (ArgList) ;</i>
<i>System</i>	→	<i>"System" ProcessList ;</i>
<i>ProcessList</i>	→	<i>ID</i>
		<i>ProcessList , ID</i>
<i>FunctionDecl</i>	→	<i>Type Id OptionalParameterList Block</i>
<i>OptionalParameterList</i>	→	<i>()</i>
		<i>(ParameterList)</i>
<i>ParameterList</i>	→	<i>Parameter</i>
		<i>ParameterList , Parameter</i>
<i>Parameter</i>	→	<i>Type & ID ArrayDecl</i>
		<i>Type ID ArrayDecl</i>
<i>VariableDecl</i>	→	<i>Type DeclIdList ;</i>
<i>DeclIdList</i>	→	<i>DeclId</i>
		<i>DeclIdList , DeclId</i>
<i>DeclId</i>	→	<i>Id ArrayDecl VarInit</i>
<i>VarInit</i>	→	<i>= Initializer</i>
		<i>Expression</i>
<i>Initializer</i>	→	<i>{ FieldInitList }</i>
<i>FieldInitList</i>	→	<i>FieldInit</i>
		<i>FieldInitList , FieldInit</i>
<i>FieldInit</i>	→	<i>Id : Initializer</i>

<i>ArrayDecl</i>	→	
		<i>ArrayDecl</i> [<i>Expression</i>]
<i>TypeDecl</i>	→	"typedef" <i>Type</i> <i>TypeIdList</i> ;
<i>TypeIdList</i>	→	<i>TypeId</i>
		<i>TypeIdList</i> , <i>TypeId</i>
<i>TypeId</i>	→	<i>Id</i> <i>ArrayDecl</i>
<i>Type</i>	→	<i>TypePrefix</i> TYPENAME <i>Range</i>
		<i>TypePrefix</i> "struct" { <i>FieldDeclList</i> }
<i>Id</i>	→	ID
		TYPENAME
<i>FieldDeclList</i>	→	<i>FieldDecl</i>
		<i>FieldDeclList</i> <i>FieldDecl</i>
<i>FieldDecl</i>	→	<i>Type</i> <i>FieldDeclIdList</i> ;
<i>FieldDeclIdList</i>	→	<i>FieldDeclId</i>
		<i>FieldDeclIdList</i> , <i>FieldDeclId</i>
<i>FieldDeclId</i>	→	<i>Id</i> <i>ArrayDecl</i>
<i>TypePrefix</i>	→	
		"urgent" "broadcast"
		"urgent" "broadcast"
		"const"
<i>Range</i>	→	
		[<i>Expression</i> , <i>Expression</i>]
<i>ProcDecl</i>	→	"orstate" <i>IdOptionalParameterList</i> { <i>ProcBody</i> }
		"andstate" <i>Id</i> <i>OptionalParameterList</i> { <i>ThreadBody</i> }
<i>ProcBody</i>	→	<i>ProcLocalDeclList</i> <i>States</i> <i>ProcInstanceList</i>
		<i>AbstractStates</i> <i>LocFlags</i> <i>History</i> <i>Init</i> <i>Transitions</i>
<i>ThreadBody</i>	→	<i>ProcLocalDeclList</i> <i>States</i> <i>ProcInstanceList</i> <i>LocFlags</i>
<i>ProcInstanceList</i>	→	
		<i>ProcInstanceList</i> "instance" <i>Inst</i>
<i>ProcLocalDeclList</i>	→	
		<i>ProcLocalDeclList</i> <i>LocalDecl</i>
<i>LocalDecl</i>	→	<i>FunctionDecl</i> <i>VariableDecl</i> <i>TypeDecl</i>
<i>AbstractStates</i>	→	"abstract" <i>StateList</i> ;
<i>States</i>	→	"state" <i>StateDeclList</i> ;
<i>StateDeclList</i>	→	<i>StateDecl</i>
		<i>StateDeclList</i> , <i>StateDecl</i>
<i>StateDecl</i>	→	ID
		ID { <i>Expression</i> }
<i>History</i>	→	
		"history" ID ;

<i>Init</i>	→	"init" ID ;
<i>Transitions</i>	→	
		"trans" TransitionList ;
<i>TransitionList</i>	→	Transition
		TransitionList , TransitionOpt
<i>Transition</i>	→	ID - > ID { Guard Sync Assign }
<i>TransitionOpt</i>	→	- > ID { Guard Sync Assign }
<i>Guard</i>	→	
		"guard" Expression ;
<i>Sync</i>	→	
		"sync" SyncExpr ;
<i>Assign</i>	→	
		"assign" ExprList ;
<i>LocFlags</i>	→	
		Commit
		Urgent
		Commit Urgent
<i>Commit</i>	→	"commit" StateList ;
<i>Urgent</i>	→	"urgent" StateList ;
<i>StateList</i>	→	ID
		StateList , ID
<i>Block</i>	→	{ BlockLocalDeclList StatementList }
<i>BlockLocalDeclList</i>	→	
		BlockLocalDeclList VariableDecl
		BlockLocalDeclList TypeDecl
<i>StatementList</i>	→	
		StatementList Statement
<i>Statement</i>	→	Block
		;
		Expression ;
		"for" (ExprList ; ExprList ; ExprList) Statement
		"while" (ExprList) Statement
		"do" Statement "while" (ExprList) ;
		"if" (ExprList) Statement ElsePart
		"break" ;
		"continue" ;
		"switch" (Expression) { SwitchCaseList }
		"return" Expression ;
		"return" ;
<i>ElsePart</i>	→	

		"else" Statement
<i>SwitchCaseList</i>	→	<i>SwitchCase</i>
		<i>SwitchCaseList SwitchCase</i>
<i>SwitchCase</i>	→	"case" Expression : <i>StatementList</i>
<i>ExprList</i>	→	<i>Expression</i>
		<i>ExprList</i> , <i>Expression</i>
<i>Expression</i>	→	"false"
		"true"
		NAT
		ID
		ID (<i>ArgList</i>)
		<i>Expression</i> [<i>Expression</i>]
		(<i>Expression</i>)
		<i>Expression</i> ++
		++ <i>Expression</i>
		<i>Expression</i> --
		-- <i>Expression</i>
		<i>UnaryOp Expression</i>
		<i>Expression BinaryOp Expression</i>
		<i>Expression</i> ? <i>Expression</i> : <i>Expression</i>
		<i>Expression</i> . ID
		"deadlock"
		<i>Expression</i> "imply" <i>Expression</i>
<i>BinaryOp</i>	→	< <= == != > >= + - * / %
		& " " ^ << >> && " " <? >?
<i>Assignment</i>	→	<i>Expression AssignOp Expression</i>
<i>AssignOp</i>	→	= + = - = / = % = & = = ^ =
		<< = >> =
<i>UnaryOp</i>	→	- !
<i>ArgList</i>	→	
		<i>Expression</i>
		<i>ArgList</i> , <i>Expression</i>
<i>PropertyList</i>	→	<i>PropertyList Property</i>
<i>Property</i>	→	
		<i>Quantifier Expression</i>
		<i>Expression</i> -- > <i>Expression</i>
<i>Quantifier</i>	→	<i>E</i> <>
		<i>E</i> []
		<i>A</i> <>
		<i>A</i> []

References

- [Abd01] Parosh Aziz Abdulla. Using (timed) petri nets for verification of parameterized (timed) systems. In *VEPAS'2001, Verification of Parameterized Systems, ICALP'2001 satellite workshop*, 2001.
- [ABH⁺97] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state-space exploration. In *Proceedings of the Ninth International Conference on Computer-Aided Verification*, volume 1254 of *LNCS*, pages 340–351. Springer, 1997.
- [ACD90] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. In *5th Symposium on Logic in Computer Science (LICS'90)*, pages 414–425, 1990.
- [ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.
- [ACH92] Rajeev Alur, Costas Courcoubetis, and Nicolas Halbwachs. Minimization of timed transition systems. In *CONCUR '92, Third International Conference on Concurrency Theory*, volume 630 of *LNCS*, pages 340–354. Springer, 1992.
- [ACH⁺95] Rajeev Alur, Costas Courcoubetis, Nicholas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Journal of Theoretical Computer Science*, 138(1):3–34, 1995.
- [AD90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Proc. of Int. Colloquium on Algorithms, Languages, and Programming*, volume 443 of *LNCS*, pages 322–335, 1990.
- [AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994. Fundamental Study.
- [AHH96] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transaction on Software Engineering*, 22:181–201, 1996.

- [AILKC⁺00] Alcatel, I-Logix, Kennedy-Carter, Kabira Technologies. Inc., Project Technology. Inc., Rational Software Corporation, and Telelogic AB. Action semantics for the UML. Response to OMG RFP ad/98-11-01, September 2000.
- [AK00] Colin Atkinson and Thomas Kühne. Strict profiles: Why and how. In *UML 2000, The Unified Modeling Language*, volume 1939 of *LNCS*, pages 309–322. Springer, 2000.
- [AL92] Martin Abadi and Leslie Lamport. An old-fashioned recipe for real time. In *Proc. of REX Workshop “Real-Time: Theory in Practice”*, number 600 in *LNCS*, pages 1–27. Springer, 1992.
- [AMY02] Rajeev Alur, Michael McDougall, and Zijiang Yang. Exploiting behavioral hierarchy for efficient model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 338–342. Springer, 2002.
- [AN01] Parosh Aziz Abdulla and Aletta Nylén. Timed petri nets and BQQs. In *Proceedings of ICATPN’2001, 22nd International Conference on application and theory of Petri nets*, 2001.
- [AY98] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. In *Proceedings of the Sixth ACM Symposium on Foundations of Software Engineering (FSE’98)*, ACM, pages 175–188, 1998.
- [Bal96] Felice Balarin. Approximate reachability analysis of timed automata. In *17th IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1996.
- [BBD⁺99] Tom Bienmüller, Udo Brockmeyer, Werner Damm, Gert Döhmen, Claus Eßmann, Hans-Jürgen Holberg, Hardi Hungar, Bernhard Josko, Rainer Schlör, Gunnar Wittich, Hartmut Wittke, Geoffrey Clements, John Rowlands, and Eric Sefton. Formal verification of an avionics application using abstraction and symbolic model checking. In Felix Redmill and Tom Anderson, editors, *Towards System Safety - Proceedings of the Seventh Safety-critical Systems Symposium*, pages 150–173. Springer, 1999.
- [BBD⁺02] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL implementation secrets. In *Proc. of 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002.
- [BC95] Robert H. Bourdeau and Betty H.C Cheng. A formal semantics for object model diagrams. *IEEE Transactions on Software Engineering*, 21(10):799–821, October 1995.

- [BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDD. In *Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579. Springer, 1999.
- [BCM⁺92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Journal of Information and Computation*, 98(2):142–170, 1992.
- [BD91] Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Trans. on Software Engineering*, 17(3):259–273, 1991.
- [Ben02] Johan Bengtsson. *Clocks, DBMs and States in Timed Systems*. PhD thesis, Uppsala University, 2002.
- [BF98] Jean-Michel Bruel and Robert B. France. Transforming UML models to formal specifications. In *UML'98 - Beyond the notation*, LNCS. Springer, 1998.
- [BFH⁺01] Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, and Judi Romijn. Efficient guiding towards cost-optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lecture Notes in Computer Science, pages 174–188. Springer, 2001.
- [BFK⁺98] Howard Bowman, Giorgio P. Faconti, Joost-Pieter Katoen, Diego Latella, and Mieke Massink. Automatic verification of a lip synchronisation algorithm using UPPAAL. In Bas Luttik Jan Friso Groote and Jos van Wamel, editors, *In Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems. Amsterdam, The Netherlands*, 1998.
- [BG92] Gerard Berry and Georges Gonthier. The esternel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
- [BHS91] Ferenc Belina, Dieter Hogrefe, and Amardeo Sarma. *SDL with Applications from Protocol Specification*. Prentice-Hall, 1991.
- [BHV00] Gerd Behrmann, Thomas Hune, and Frits Vaandrager. Distributed timed model checking - How the search order matters. In *Proc. of 12th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Chicago, Juli 2000. Springer.
- [BJLY98] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial order reductions for timed systems. In *Proceedings of the 9th International Conference on Concurrency Theory*, September 1998.

- [BJR97] Grady Booch, Ivar Jacobson, and James E. Rumbaugh. *Unified Modeling Language User Guide*. Addison Wesley, 1997.
- [BJR99] Grady Booch, Ivar Jacobson, and James E. Rumbaugh. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.
- [BLA⁺99] Gerd Behrmann, Kim G. Larsen, Henrik Reif Andersen, Henrik Hultgaard, and Jørn Lind-Nielsen. Verification of hierarchical state/event systems using reusability and compositionality. In *TACAS: Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, 1999.
- [BLL⁺95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL — a tool suite for automatic verification of real-time systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer, October 1995.
- [BLP⁺99] Gerd Behrmann, Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. In *Proceedings of the 12th Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
- [Bry86] Randal E. Bryan. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
- [CAB⁺94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, and Paul Jeremaes. *Object-Oriented Development: The Fusion Method*. Object-Oriented Series edition. Prentice Hall, 1994.
- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
- [CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.
- [CC92a] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992. (The editor of *Journal of Logic Programming* has mistakenly published the unreadable galley proof. For a correct version of this paper, see <http://www.di.ens.fr/~cousot>.)

- [CC92b] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92*, Leuven, Belgium, 13-17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer, Berlin, Germany, 1992.
- [CC94] Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to compartment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 95–112, Toulouse, France, 16–19 May 1994. IEEE Computer Society Press, Los Alamitos, California.
- [CC99] Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Automated Software Engineering*, 6(1):69–95, 1999.
- [CC02] Patrick Cousot and Radhia Cousot. Modular static program analysis, invited paper. In R.N. Horspool, editor, *Proceedings of the Eleventh International Conference on Compiler Construction (CC 2002)*, volume 2304, pages 159–178, Grenoble, France, April 2002. Springer.
- [CCK⁺02] Pankaj Chauhan, Edmund M. Clarke, James Kukula, Samir Sapra, Helmut Veith, and Dong Wang. Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In *Formal Methods in Computer Aided Design (FMCAD'02)*, page 18, November 2002.
- [Cer92] Karlis Cerans. Decidability of bisimulation equivalences for parallel timer processes. In *Computer Aided Verification*, volume 663 of *LNCS*. Springer, 1992.
- [CGJ⁺00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, July 2000.
- [CGL93] Karlis Cerans, Jens Chr. Godskesen, and Kim G. Larsen. Time model specification - theory and tools. In *5th International Conference on Computer Aided Verification*, volume 697 of *LNCS*. Springer, 1993.
- [CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
- [CGMP99] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State space reduction using partial order techniques. *STTT*, 2(3):279–287, 1999.

- [CIY95] Rance Cleaveland, Purush Iyer, and Daniel Yankelevich. Optimality in abstractions of model checking. In *Static Analysis, Second International Symposium, SAS'95*, volume 983 of *LNCS*, pages 51–63. Springer, September 1995.
- [CK97] Søren Christensen and Lars Michael Kristensen. State space analysis of hierarchical coloured petri nets. In B. Farwer, D. Moldt, and M-O. Stehr, editors, *Proceedings of Workshop on Petri Nets in System Engineering (PNSE'97) Modelling, Verification, and Validation*, number 205 in *LNCS*, pages 32–43, Hamburg, Germany, 1997.
- [CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press Massachusetts Institute of Technology, 2001.
- [CN97] Gianfranco F. Ciardo and David M. Nicol. Automated parallelization of discrete state-space generation. In *Journal of Parallel and Distributed Computing*, volume 47, pages 153–167. ACM, december 1997.
- [Coo00] Steve Cook. The UML family: Profiles, prefaces and packages. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 255–264. Springer, 2000.
- [Cou00] Patrick Cousot. Interprétation abstraite. *Technique et science informatique*, 19(1–2–3):155–164, January 2000.
- [CP99] Ching-Tsun Chou and Doron Peled. Formal verification of a partial-order reduction technique for model checking. *Journal of Automated Reasoning*, 23(3–4):265–298, 1999.
- [CVWY92] Costas Courcoubetis, Moshe Y. Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, pages 275–288, 1992.
- [Dam96] Dennis René Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University, 1996.
- [dB99] Lydie du Bousquet. Feature interaction detection using testing and model-checking experience report. In *FM'99 Formal Methods*, volume 1708 of *LNCS*, pages 622–641. Springer, 1999.
- [DBL02] Henning Dierks, Gerd Behrmann, and Kim G. Larsen. Solving planning problems using real-time model checking (translating PDDL3 into timed automata). Workshop proceeding of AIPS'02 available at <http://csl.anu.edu.au/~thiebaut/papers/aips02/saipsproc4.ps.gz>, April 2002.

- [DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.
- [DGKK98] Dennis Dams, Rob Gerth, Bart Knaack, and Ruurd Kuiper. Partial-order reduction techniques for real-time model checking. *Formal Aspects of Computing*, 10(5–6):469–482, 1998.
- [Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proceedings of Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer, 1989.
- [DJHP98] Werner Damm, Bernhard Josko, Hardi Hungar, and Amir Pnueli. A compositional real-time semantics of STATEMATE designs. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997*, volume 1536 of *LNCS*, pages 186–238. Springer, 1998.
- [DJVP03] Werner Damm, Bernhard Josko, Angelika Votintseva, and Amir Pnueli. A formal semantics for a UML kernel language. <http://www-omega.imag.fr/queries/dm-downloads.php>, January 2003.
- [DKRT97] Pedro .R. D’Argenio, Joost-Pieter. Katoen, Theo C. Ruys, and Jan Tretmans. The bounded retransmission protocol must be on time! In *In Proceedings of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1217 of *LNCS*, pages 416–431. Springer, April 1997.
- [DM01] Alexandre David and M. Oliver Möller. From HUPPAAL to UPPAAL: A translation from hierarchical timed automata to flat timed automata. Technical Report RS-01-11, BRICS, March 2001.
- [DMY02] Alexandre David, M. Oliver Möller, and Wang Yi. Formal verification of UML statecharts with real-time extensions. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering, 5th International Conference, FASE 2002*, volume 2306 of *LNCS*, pages 218–232. Springer, 2002.
- [DMY03] Alexandre David, M. Oliver Möller, and Wang Yi. Verification of UML statecharts with real-time extensions. Technical Report 2003-009, Uppsala University, 2003.
- [Dou99] Bruce Powel Douglas. *Real-Time UML*. Addison Wesley, 1999.
- [DOY94] Conrado Daws, Alfredo Olivero, and Sergio Yovine. Verifying ET-LOTOS programs with Kronos. In *Proceedings of the 7th IFIP WG G.1 International Conference of Formal Description Techniques*

- FORTE'94*, Formal Description Techniques VII, pages 227–242. Chapman & Hall, October 1994.
- [DU95] David L. Dill and Ulrich Stern. Improved probabilistic verification by hash compaction. In P.E. Camurati and H. Eweking, editors, *Correct Hardware Design and Verification Methods*, volume 987, pages 206–224, Stanford University, USA, 1995. Springer.
- [DY96] Conrado Daws and Sergio Yovine. Reducing the number of clock variables of timed automata. In *Proceedings of the 17th IEEE Real Time Systems Symposium, RTSS'96*. IEEE Computer Society Press, December 1996.
- [DY98] Akash Deshpande and Sergio Yovine. System design using teja and kronos. case study: The FDDI protocol. In “*Educational Case Studies in Protocols*”, *ECASP, FORTE/PSTV'98*, 1998.
- [DY00] Alexandre David and Wang Yi. Modelling and analysis of a commercial field bus protocol. In *Proceedings of the 12th Euromicro Conference on Real Time Systems*, pages 165–172. IEEE Computer Society, 2000.
- [EFLR98] Andy Evans, Robert B. France, Kevin Lano, and Bernhard Rumpe. The UML as a formal modelling notation. In *UML'98 - Beyond the notation*, LNCS. Springer, 1998.
- [ES97] E. Allen Emerson and A. Prasad Sistla. Using symmetry when model checking under fairness assumption: an automata theoretic approach. *ACM Transactions on Programming Languages and Systems*, 19(4), 1997.
- [FELR98] Robert B. France, Andy Evans, Kevin Lano, and Bernhard Rumpe. The UML as a formal modeling notation. *Computer Standards & Interfaces*, 19:325–334, 1998.
- [Flo62] Robert W. Floyd. Acm algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [GLM02] Sefania Gnesi, Diago Latella, and Mieke Massink. Modular semantics for a UML statechart diagrams kernel and its extension to multi-charts and branching time model-checking. *The Journal of Logic and Algebraic Programming*, 51:43–75, 2002.
- [GO03] Susanne Graf and Ileana Ober. A real-time profile for UML and how to adapt it to SDL. <http://www-omega.imag.fr/queries/dm-downloads.php>, March 2003.
- [God90] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Second International Conference on Computer Aided Verification*, volume 531 of *LNCS*, pages 176–185. Springer, 1990.

- [God95] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. PhD thesis, Université de Liège, 1995.
- [GOO] Susanne Graf, Ileana Ober, and Iulian Ober. Timed annotations with UML. to be published at SVERTS Workshop, UML 2003.
- [Hal96] Anthony Hall. Using formal methods to develop an ATC information system. *IEEE Software*, 13(2):66–76, March 1996.
- [Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
- [Hen94] Thomas A. Henzinger. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.
- [HG97] David Harel and Eran Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, 1997.
- [HGdR88] Cornelis Huizing, Rob Gerth, and Willem P. de Roever. Modeling statecharts in a fully abstract way. In *Proceedings of CAAP 88*, volume 299 of *LNCS*, pages 271–294. Springer, 1988.
- [HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A model checker for hybrid systems. *software Tools for Technology Transfer*, 1:110–122, 1997.
- [HJJJ84] Peter Huber, Arne M. Jensen, Leif O. Jespen, and Kurt Jensen. Towards reachability trees for high-level petri nets. In *Advances on Petri Nets'84*, volume 188 of *LNCS*. Springer, 1984.
- [HL02] Martijn Hendriks and Kim G. Larsen. Exact acceleration of real-time model checking. In E. Asarin, O. Maler, and S. Yovine, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, April 2002.
- [HLP00] Thomas Hune, Kim G. Larsen, and Paul Pettersson. Guided synthesis of control programs using UPPAAL. In Ten H. Lai, editor, *Proc. of the IEEE ICDCS International Workshop on Distributed Systems Verification and Validation*, pages E15–E22. IEEE Computer Society Press, April 2000.
- [HLS99] Klaus Havelund, Kim G. Larsen, and Arne Skou. Formal verification of a power controller using the real-time model checker UPPAAL. 5th International AMAST Workshop on Real-Time and Probabilistic Systems, available at <http://www.uppaal.com>, 1999.
- [HLY92] Uno Holmer, Kim G. Larsen, and Wang Yi. Decidability of bisimulation equivalence between regular timed processes. In *Computer Aided Verification*, volume 575 of *LNCS*. Springer, 1992.

- [HM99] Nisse Husberg and Tapio Manner. Emma: Developing an industrial reachability analyser for SDL. In *FM'99 Formal Methods*, volume 1708 of *LNCS*, pages 642–661. Springer, 1999.
- [HN96] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions of Software Engineering and Methodology*, 5(4), October 1996.
- [HNSY92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *Proc. of IEEE Symposium on Logic in Computer Science*, 1992.
- [Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
- [Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
- [Hol98] Gerard J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13:289–307, 1998.
- [HP85] David Harel and Amir Pnueli. On the development of reactive systems. *Logics and Models of Concurrent Systems*, F-13:477–498, 1985.
- [HPSS87] David Harel, Amir Pnueli, Jeanette P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *IEEE Symposium on Logic in Computer Science*, pages 54–64, 1987.
- [HS00] Gerard J. Holzmann and Margaret H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.
- [HSSL97] Klaus Havelund, Arne Skou, Kim G. Larsen, and Kristian Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 2–13, December 1997.
- [HU01] John E. Hopcroft and Jeffrey D. Ullman. *Introduction fo Automata Theory, Languages, and Computation*. Addison Wesley, 2001.
- [Hui91] Cornelis Huizing. *Semantics of Reactive Systems: Comparison and Full Abstraction*. PhD thesis, Technical University Eindhoven, 1991.
- [ID93] C. Norris Ip and David L. Dill. Efficient verification of symmetric concurrent systems. In *International Conference on Computer Design: VLSI in Computers and Processors, IEEE Computer Society*, pages 230–234, 1993.
- [ID96] C. Norris Ip and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, August 1996.

- [IIC⁺02] ARTISAN Software Tools Inc., I-Logix Ind., Rational Software Corp., Telelogic AB, TimeSys Corporation, and Tri-Pacific Software. UML profile for schedulability, performance, and time specification. Available from the OMG web site <http://www.omg.org>, Mars 2002.
- [IKL⁺00] Torsten K. Iversen, Kåre J. Kristoffersen, Kim G. Larsen, Morten Laursen, Rune G. Madsen, Steffen K. Mortensen, Paul Pettersson, and Chris B. Thomasen. Model-checking real-time control programs — Verifying LEGO mindstorms systems using UPPAAL. In *Proc. of 12th Euromicro Conference on Real-Time Systems*, pages 147–155. IEEE Computer Society Press, June 2000.
- [Jen99] Henrik Ejersbo Jensen. *Abstraction-Based Verification of Distributed Systems*. PhD thesis, Aalborg University, 1999.
- [KER99] Stuart Kent, Andy Evans, and Bernhard Rumpe. UML semantics FAQ. In Ana M. D. Moreira and Serge Demeyer, editors, *Object-Oriented Technology, ECOOP'99 Workshop Reader, ECOOP'99 Workshops*, volume 1743 of *LNCS*, pages 33–56, 1999.
- [KLL⁺96] Kåre J. Kristoffersen, François Laroussinie, Kim G. Larsen, Paul Pettersson, and Wang Yi. A compositional proof of a real-time mutual exclusion protocol. Technical Report RS-96-55, BRICS, December 1996.
- [KLPW99] Kåre J. Kristoffersen, Kim G. Larsen, Paul Pettersson, and Carsten Weise. VHS case study 1 - experimental batch plant using UPPAAL. BRICS, University of Aalborg, Denmark, May 1999.
- [KP92] Yonit Kesten and Amir Pnueli. Timed and hybrid statecharts and their textual representation. In J. Vytopil, editor, *Formal Techniques in Real-time and Fault Tolerant Systems, 2nd International Symposium*, volume 571 of *LNCS*, pages 591–620. Springer, 1992.
- [Kwo00] Gihwon Kwon. Rewrite rules and operational semantics for model checking UML statecharts. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference*, volume 1939 of *LNCS*, pages 528–555. Springer, 2000.
- [LBB⁺01] Kim G. Larsen, Gerd Behrmann, Ed Brinksma, Ansgar Fehnker, Thomas Hune, Paul Pettersson, and Judi Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV 2001*, number 2102 in *Lecture Notes in Computer Science*, pages 493–505. Springer, 2001.
- [LE99] Kevin Lano and Andy Evans. Rigorous development in UML. In *Fundamental Approaches to Software Engineering (FASE'99)*. Springer, 1999.

- [Lev97a] Francesca Levi. Compositional verification of timed statecharts. In *Advances in Temporal Logic*, number 16 in Applied Logic, pages 47–70, 1997.
- [Lev97b] Francesca Levi. *Verification of Temporal and Real-time Properties of Statecharts*. PhD thesis, University of Pisa, June 1997.
- [Lev01] Francesca Levi. A syntolic semantics for abstract model checking. *Science of Computer Programming*, 39:93–123, 2001.
- [LLPY97] Fredrik Larsson, Kim G. Larsen, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: Compact data structures and state-space reduction. In *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, December 1997.
- [LMM99a] Diego Latella, István Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.
- [LMM99b] Diego Latella, István Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *3rd International Conference on Formal Methods for Open Object-Oriented Distributed Systems, Boston, 1999 (FMOODS’99)*, pages 331–347. Kluwer Academic Publishers, 1999.
- [LMS97] Yassine Lakhnech, Erich Mikk, and Michael Siegel. Hierarchical automata as model for statecharts. In *Proc. of the Asian Computing Science Conference (ASIAN’97)*, volume 1345 of *LNCS*, pages 181–196. Springer, 1997.
- [LNAB⁺98] Jørn Lind-Nielsen, Henrik Reif Andersen, Gerd Behrmann, Henrik Hulgaard, Kåre J. Kristoffersen, and Kim G. Larsen. Verification of large state/event systems using compositionality and dependency analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, *LNCS*, pages 201–216. Springer, 1998.
- [LP97] Henrik Lönn and Paul Pettersson. Formal verification of a TDMA protocol startup mechanism. In *Proc. of the Pacific Rim Int. Symp. on Fault-Tolerant Systems*, pages 235–242, December 1997.
- [LP99a] Johan Lilius and Ivan Porres. Formalising UML state machines for model-checking. In In Robert B. France and Bernhard Rumpe, editors, *UML’99 - The Unified Modeling Language*, volume 1723 of *LNCS*, pages 430–445. Springer Verlag, 1999.
- [LP99b] Johan Lilius and Ivan Porres. vUML: a tool for verifying UML models. In *Proceedings of the Automatic Software Engineering Conference (ASE’99)*. IEEE Computer Society, October 1999.

- [LPWY99] Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Clock difference diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999.
- [LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-checking for real-time systems. In *Proc. of Fundamentals of Computation Theory*, number 965 in Lecture Notes in Computer Science, pages 62–88, August 1995.
- [LPY97] 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.
- [LPY00] Fredrik Larsson, Paul Pettersson, and Wang Yi. On memory-block traversal problems in model checking timed systems. In Susanne Graf and Michael Schwartzbach, editors, *Proc. of the 6th Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 1785 in Lecture Notes in Computer Science, pages 127–141. Springer, 2000.
- [LPY01] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gearbox controller. *Springer International Journal of Software Tools for Technology Transfer (STTT)*, 3(3):353–368, 2001.
- [LvdBC99] Gerald Luetzgen, Michael von der Beeck, and Rance Cleaveland. Statecharts via process algebra. In J.C.M. Baeten and S. Mauw, editors, *CONCUR’99. Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, 1999 Proceedings, Lecture Notes in Computer Science, vol. 1664*. Springer, 1999.
- [LY93] Kim G. Larsen and Wang Yi. Time abstracted bisimulation: Implicit specification and decidability. In *Proceedings of MFPS93 (the 9th International Conference on Mathematical Foundations of Programming Semantics)*, volume 802 of LNCS, pages 160–176. Springer, 1993.
- [Mar89] Florence Maraninchi. Argonaute: Graphical description, semantics and verification of reactive systems by using a process algebra. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of LNCS, pages 38–53. Springer, 1989.
- [Mar92] Florence Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR’92*, number 630 in LNCS, pages 550–564. Springer, 1992.
- [Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [Min99] Marius Minea. *Partial Order for Verification of Timed Systems*. PhD thesis, Carnegie Mellon University, 1999.

- [MLPS97] Erich Mikk, Yassine Lakhnech, Carsta Petersohn, and Michael Siegel. On formal semantics of statecharts as supported by STATEMATE. In *2nd BCS-FACS Northern Formal Methods Workshop*. Springer, July 1997.
- [MLSH98] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. Implementing statecharts in Promela/SPIN. In *Workshop in Industrial-Strength Formal Specifications Techniques (WIFT'98)*. IEEE Computer Society Press, 1998.
- [Möl02] M. Oliver Möller. *Structure and Hierarchy in Real-Time Systems*. PhD thesis, BRICS, University of Aarhus, February 2002. available from <http://www.verify-it.de/papers.html>.
- [NY01] Peter Niebert and Sergio Yovine. Computing efficient operation schemes for chemical plants in multi-batch mode. *European Journal of Control*, 2001.
- [OMG01] OMG unified modeling language specification. www.omg.org, September 2001. version 1.4.
- [Pag96] Florence Pagani. Partial orders and verification of real-time systems. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 4th International Symposium, FTRTFT'96*, volume 1135 of *LNCS*, pages 327–346. Springer, 1996.
- [Pap95] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1995.
- [Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In *Fifth International Conference on Computer Aided Verification*, volume 697 of *LNCS*, pages 409–423. Springer, 1993.
- [Pel96] Doron Peled. Partial order reduction: Linear and branching temporal logics and process algebras. DIMACS workshop on Partial Order Methods in Verification, 1996.
- [Pet99] Paul Pettersson. *Modelling and Verification of Real-time Systems Using Timed Automata: Theory and Practice*. PhD thesis, Uppsala University, 1999.
- [Por01] Ivan Porres. *Modeling and Analyzing Software Behavior in UML*. PhD thesis, Åbo Akademi University, 2001.
- [PS91] Amir Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Theoretical Aspects of Computer Science*, volume 526 of *LNCS*, pages 244–264. Springer, 1991.
- [RBL⁺95] James E. Rumbaugh, Michael Blaha, William Premer Lani, Frederick Eddy, and William Corengen. *Object Oriented Modeling and Design*. Prentice Hall, 1995.

- [Rok93] Tomas Gerhard Rokicki. *Representing and Modeling Digital Circuits*. PhD thesis, Stanford University, 1993.
- [RP85] Rami R. Razouk and Charles V. Phelps. Performance analysis using timed petri nets. In *Protocol Testing, Specification, and Verification*, pages 561–576, 1985.
- [SB03] Sanjit A. Seshia and Randal E. Bryant. Unbounded, fully symbolic model checking of timed automata using boolean methods. In Warren A. Hunt and Jr. Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725, pages 154–166, LNCS, 2003. Springer.
- [SD98] Ulrich Stern and David L. Dill. Using magnetic disk instead of main memory in the murphi verifier. In *Computer Aided Verification. 10th International Conference*, pages 172–183, 1998.
- [Ski98] Steven S. Skiena. *The Algorithm Design Manual*. Springer, 1998.
- [Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1997.
- [TAKB96] Serdar Taşiran, Rajeev Alur, Robert P. Kurshan, and Robert K. Brayton. Verifying abstractions of timed systems. In *Proceedings of the Seventh Conference on Concurrency Theory*, volume 1119 of LNCS. Springer, 1996.
- [Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.
- [TY98] Stavros Tripakis and Sergio Yovine. Verification of the fast reservation protocol with delayed transmission using the tool Kronos. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium, RTAS'98*. IEEE Computer Society Press, June 1998.
- [TY01] Stavros Tripakis and Sergio Yovine. Analysis of timed systems using time-abtracting bisimulations. *Formal Methods in Systems Design*, 18:25–68, 2001. Kluwers Academic Publishers, Boston.
- [US94] Andrew C. Uselton and Scott A. Smolka. A compositional semantics for statecharts using labeled transition systems. In *International Conference on Concurrency Theory*, volume 836, pages 2–17, 1994.
- [Val90] Antti Valmari. A stubborn attack on state explosion. In *Second International Conference on Computer Aided Verification*, volume 531 of LNCS, pages 156–165. Springer, 1990.
- [vdB94] Michael von der Beeck. A comparison of statecharts variants. In *Formal Techniques in Real-time and Fault-Tolerant Systems*, volume 863 of LNCS, pages 128–148, 1994.
- [vdB01] Michael von der Beeck. Formalization of UML statecharts. In *UML 2001*, volume 2185 of LNCS, pages 406–421, 2001.

- [Wan01] Farn Wang. RED: Model-checker for timed automata with clock-restriction diagram. In Paul Pettersson and Sergio Yovine, editors, *Workshop on Real-Time Tools, Aalborg University Denmark*, number 2001-014 in Technical Report. Uppsala University, 2001.
- [Wan03] Farn Wang. Efficient verification of timed automata with BDD-like data-structures. In *Verification, Model Checking, and Abstract Interpretation: 4th International Conference, VMCAI 2003*, volume 2575 of *LNCS*, pages 189–205. Springer, 2003.
- [WC99] Andre Wong and Marsha Chechik. Formal modeling in a commercial setting: A case study. In *FM'99 Formal Methods*, volume 1708 of *LNCS*, pages 590–605. Springer, 1999.
- [WL93] Pierre Wolper and Denis Leroy. Reliable hashing without collision detection. In C. Courcoubetis, editor, *Computer Aided Verification: Proc. of the 5th International Conference CAV'93*, pages 59–70. Springer, Berlin, Heidelberg, 1993.
- [WT94] Howard Wong-Toi. *Symbolic Approximations for Verifying Real-time Systems*. PhD thesis, Stanford University, 1994.
- [Yan00] Mihalis Yannakakis. Hierarchical state machines. In *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS 2000*, volume 1872 of *LNCS*, pages 315–330. Springer, 2000.
- [Yi91] Wang Yi. *A Calculus of Real Time Systems*. PhD thesis, University of Göteborg, 1991.
- [Yov97] Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal of Software Tools for Technology Transfer*, 1:123–133, October 1997.
- [YPD94] Wang Yi, Paul Pettersson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Seventh International Conference on Formal Description Techniques*, pages 223–238, 1994.
- [Zub80] Wlodzimierz M. Zuberek. Timed petri nets and preliminary performance evaluation. In *Proceedings of the 7th annual symposium on Computer Architecture*, pages 88–96. ACM Press, 1980.
- [Zub85] Wlodzimierz M. Zuberek. Extended d-timed petri nets, timeouts, and analysis of communication protocols. In *Proceedings of the 1985 ACM annual conference on the range of computing : mid-80's perspective*, pages 10–15. ACM Press, 1985.

Recent technical reports from the department of Information Technology:

- 2003-035** Elisabeth Larsson and Bengt Fornberg: *Theoretical and Computational Aspects of Multivariate Interpolation with Increasingly Flat Radial Basis Functions*
- 2003-036** Emad Abd-Elrady, Torsten Söderström, and Torbjörn Wigren: *Periodic Signal Modeling Based on Liénard's Equation*
- 2003-037** Dan Wallin and Erik Hagersten: *Bundling: Reducing the Overhead of Multiprocessor Prefetchers*
- 2003-038** Henrik Björklund, Sven Sandberg, and Sergei Vorobyov: *On Fixed-Parameter Complexity of Infinite Games*
- 2003-039** Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena: *Insights to Angluin's Learning*
- 2003-040** Wendy Kress: *Error Estimates for Deferred Correction Methods in Time*
- 2003-041** Wendy Kress: *A Compact Fourth Order Time Discretization Method for the Wave Equation*
- 2003-042** Gerardo Schneider: *Invariance Kernels of Polygonal Differential Inclusions*
- 2003-043** Kajsa Ljungberg, Sverker Holmgren, and Örjan Carlborg: *Simultaneous Search for Multiple QTL Using the Global Optimization Algorithm DIRECT*
- 2003-044** Dan Wallin, Henrik Johansson, and Sverker Holmgren: *Cache Memory Behavior of Advanced PDE Solvers*
- 2003-045** Sven-Olof Nyström: *A Polyvariant Type Analysis for Erlang*
- 2003-046** Martin Karlsson: *A Power-Efficient Alternative to Highly Associative Caches*
- 2003-047** Jimmy Flink: *Simuleringsmotor för tågtrafik med stöd för experimentell konfiguration*
- 2003-048** Timour Katchaounov and Tore Risch: *Interface Capabilities for Query Processing in Peer Mediator Systems*
- 2003-049** Martin Nilsson: *A Parallel Shared Memory Implementation of the Fast Multipole Method for Electromagnetics*
- 2003-050** Alexandre David: *Hierarchical Modeling and Analysis of Timed Systems*