Structure and Hierarchy in Real-Time Systems

M. Oliver Möller

PhD Dissertation

BRICS

BRICS PhD School
Department of Computer Science
University of Aarhus
Denmark

February 2002 **Supervisor: Kim G. Larsen**

Structure and Hierarchy in Real-Time Systems

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfillment of the Requirements for the
PhD Degree

by
Michael Oliver Möller
20 February 2002

Abstract

The development of digital systems is particularly challenging, if their correctness depends on the right timing of operations. One approach to enhance the reliability of such systems is model-based development. This allows for a formal analysis throughout all stages of design.

Model-based development is hindered mainly by the lack of adequate modeling languages and the high computational cost of the analysis. In this thesis we improve the situation along both axes.

First, we bring the mathematical model closer to the human designer. This we achieve by casting hierarchical structures—as known from statechart-like formalisms—into a formal timed model. This shapes a high-level language, which allows for fully automated verification.

Second, we use sound approximations to achieve more efficient automation in the verification of timed properties. We present two novel state-based techniques that have the potential to reduce time and memory consumption drastically.

The first technique makes use of structural information, in particular loops, to exploit regularities in the reachable state space. Via shortcut-like additions to the model we avoid repetition of similar states during an exhaustive state space exploration.

The second technique applies the abstract interpretation framework to a real-time setting. We preserve the control structure and approximate only the more expensive time component of a state. The verification of infinite behavior, also known as liveness properties, requires an assumption on the progress of time. We incorporate this assumption by limiting the behavior of the model with respect to delay steps. For a next-free temporal logic, this modification does not change the set of valid properties.

We supplement our research with experimental run-time data. This data is gathered via prototype implementations on top of the model checking tools Uppaal and Mocha.

Acknowledgments

Hardly—if ever—a PhD is the sole achievement of one person. It rather is a journey where the traveler is dependent on many an aid on the road, has to learn the language, has to ask for directions, and often needs a helping hand.
In my case, this journey led not only through realms of theory and engineering but also through countries like Denmark, Sweden, Pennsylvania, and California. It is only just to name my debts and thank the people who helped me on my way—here is the place for it.

I want to thank BRICS director Mogens Nielsen for having an open ear, for being open and honest, for believing so much in science that he even likes PhD students.
I want to thank my supervisor Kim Guldstrand Larsen for his kind support, his enlightening lectures, and for many valuable discussions.
I want to thank the golden secretaries Janne Kroun Christensen, Karen Kjær Møller, and Ulla Duus for helping me with tricky organizations and for having answers for my numerous questions. Ingrid Larsen was always willing to suffer my imperfect Danish, and I want to thank her for her patience with my travel reports.

I want to thank professor Rajeev Alur for his kind advice, and for giving me the opportunity to work together with the Mocha group in one of the most historic places in computer science lore.
I want to thank professor Wang Yi for introducing me to the "Upp" part of the Uppaal group. This not only taught me UML and most I know about embedded systems, it also made me alert to how invaluable warm clothing can be.
I want to thank professor John Rushby for once more allowing me to work in the vivacious environment of the Stanford Research Institute (SRI) and to benefit from the people and ideas this institution harbors. It was him who showed me what an exciting place science can be.

Working in the Uppaal team is a great motivation, and I thank its members for sharing this with me for some time. I include special thanks to Paul Pettersson, who introduced me to scheduling, modeling, and AIT-WOODDES. The discussions with him were always an enrichment, and from him I learned a great number of valuable things.

I want to thank my BRICS fellow students for being company, and more often than not friends. Special thanks go to the Danish PhD students, who sought contact with

the internationals. They made Denmark a more welcoming place.

In particular I want to thank Daniele Varacca for the conversations we had over a game of cards. They always reminded me, how colorful the world around us is.
Also, I want to thank Riko Jacob and Paola Quaglia for resisting the temptation to send me to a mental home, after I talked about categories for weeks.

Condensed thanks go to the coffee paradise Altura in Århus Graven for bringing some of the world's best beans into my reach. More than 24 pounds of their ground delights were needed to create this document.

A document of this volume is not composed without any technical problems. I want to thank Uffe Engberg and Kaja P. Christiansen for invaluable help with LaTeX and the `hyperref` package.

Thanks go to Christa Mauer, her colleagues, and Pawel Sobocinski for proof-reading the UML part (Chapter 1) and giving helpful comments. Maria Sorea read through Chapter 7 and helped me to improve the presentation. Thanks go also to Alexandre David for going through the details of Chapters 3, 5, and 9.

I want to thank the family Pedersen, including the she-dog Rudi, for giving me a home in the mind-soothing surroundings of the Lake Brabrand. I consider myself fortunate for every day I was able to spend there.
I want to thank the folks from the Århus Klatreklubben for not letting me down (and sometimes: for letting me down again), after having spent many hours with both hands on the wall.
I want to thank the people from the Jomsborg Vikingeklubben, who helped me getting a clear mind again after many a concentrated working day.
I want to thank my father, my brother, and my sister in law for supporting my work, for paying me visits, and for being a home for me, whenever I showed up in Burgau.
I want to thank my friends in Burgau, Günzburg, AuXburg, Friedberg and many other places inside and outside Bavaria—for not forgetting me, when I was in foreign countries, and for keeping contact with words and visits. They prevented me from feeling lost on this planet and helped me believing that I was doing the right thing.

This list is far from exhaustive; I pray for forgiveness from those I did not mention by name and include them in my heart-felt gratitude.

M. Oliver Möller,
Århus, 20 February 2002.

Overview Table of Contents

Abstract v

Acknowledgments vii

Overview Table of Contents ix

Table of Contents xi

Introduction 1

I Modeling of Real-Time Systems 19

1 UML and Statecharts 23

2 The Timed Automata Model of UPPAAL 41

3 Hierarchical Timed Automata 53

II Algorithmic Verification of Real-Time Systems 73

4 Symbolic Forward Analysis 77

5 Efficiency in Real-Time Model Checking 91

6 The Model Augmentation Technique 107

7 Abstract Interpretation of Dense Real-Time 119

III Making Use of Hierarchical Structure **147**

8 Hierarchical Partitioning **151**

9 Model Checking Hierarchical Timed Automata **173**

Epilogue **197**

Bibliography **199**

Index **219**

Abbreviations **227**

Table of Contents

Abstract v

Acknowledgments vii

Overview Table of Contents ix

Table of Contents xi

Introduction 1

0.1 Doing it Right: Correctness . . . 3
0.2 Formal Methods . . . 5
0.2.1 The Necessity of Being Formal . . . 5
0.2.2 Verification Engineer: A Future Profession? . . . 6
0.2.3 Automation, Automation, Automation . . . 7
0.2.4 What are Reasonable Hopes? . . . 8
0.3 Techniques for Formal Verification . . . 9
0.3.1 Automated Theorem Proving . . . 9
0.3.2 Process Algebraic Methods . . . 9
0.3.3 Stepwise Refinement . . . 10
0.3.4 Abstract Interpretation . . . 10
0.3.5 Model Checking . . . 11
0.3.6 Combining Techniques . . . 11
0.4 The Design of Real-Time Systems . . . 12
0.4.1 Discrete Analysis Techniques for Real-Time Systems . . . 13
0.4.2 Increase in Complexity . . . 14
0.5 The State of the Art . . . 15
0.6 Outline: A Guided Tour Through This Thesis . . . 16

I Modeling of Real-Time Systems **19**

1 UML and Statecharts **23**
1.1 An Outline of UML . . . 24
1.1.1 From Unified Method 0.8 to UML 2.0 . . . 24
1.1.2 Meta-Modeling: The Four Layers of the UML . . . 25
1.1.3 Extensibility: Lightweight and Heavyweight . . . 27
1.1.4 Realizing Technologies: OMG and W3C Standards . . . 27
1.1.5 Learning UML . . . 29
1.1.6 Literature on UML . . . 30
1.2 UML Statecharts . . . 32
1.2.1 The Evolution of Statecharts . . . 32
1.2.2 The Basics of UML Statecharts . . . 33
1.2.3 Semantics: Still under Development . . . 36
1.2.4 CASE Tool Implementations of Statecharts . . . 37
1.3 Reflection: UML and Statecharts . . . 39

2 The Timed Automata Model of UPPAAL **41**
2.1 Timed Automata in UPPAAL . . . 42
2.1.1 Informal Description . . . 42
2.1.2 Formal Syntax . . . 44
2.2 Trace Semantics of the UPPAAL Model . . . 46
2.2.1 Collection of Legal Traces . . . 48
2.3 The Logic Language of UPPAAL . . . 49
2.3.1 Local Properties . . . 49
2.3.2 Temporal Properties . . . 50
2.4 Reflection: What Kind of Tool is UPPAAL? . . . 51

3 Hierarchical Timed Automata **53**
3.1 Syntax of Hierarchical Timed Automata . . . 54
3.1.1 A Restricted Statechart Formalism . . . 54
3.1.2 Data Components . . . 54
3.1.3 Structural Components . . . 55
3.1.4 Well-Formedness Constraints . . . 56
3.2 Operational Semantics of HTAs . . . 58
3.3 Unbounded Event Queues . . . 63
3.3.1 Turing Machines and the Halting Problem . . . 64
3.3.2 Undecidability of Unbounded Queues . . . 65
3.4 Partial Encoding of Events . . . 68
3.4.1 Events in RHAPSODY . . . 68
3.4.2 Respecting Number, Ignoring Order . . . 69
3.5 Reflection: Hierarchical Timed Automata . . . 70

II Algorithmic Verification of Real-Time Systems 73

4 Symbolic Forward Analysis 77
4.1 Symbolic Representation of Traces . . . 78
4.2 Data-Structures for Symbolic Real-Time . . . 80
4.2.1 Regions and Zones . . . 80
4.2.2 Operations on Zones . . . 81
4.2.3 Difference-Bounded Matrices (DBMs) . . . 82
4.3 Forward State-Space Exploration . . . 83
4.3.1 Symbolic Forward Reachability . . . 84
4.3.2 Variations of the Inclusion Test . . . 85
4.3.3 Liveness Checking . . . 86
4.4 Reflection: Symbolic Analysis of Real-Time Systems . . . 88

5 Efficiency in Real-Time Model Checking 91
5.1 Optimizations for Real-Time Model Checking . . . 92
5.1.1 Active Clock Reduction (`-a`) . . . 92
5.1.2 Compact DBM Representation (OFF with `-C`) . . . 92
5.1.3 Space Usage Reduction by Smaller "Passed" List (`-S 1|2`) 93
5.2 Approximation Techniques for Real-Time Systems . . . 93
5.2.1 Convex Hull Over-Approximation (`-A`) . . . 93
5.2.2 Under-Approximation: Bitstate Hashing (`-Z`) . . . 94
5.2.3 Other Approximation Techniques . . . 94
5.3 Other Options of the UPPAAL Engine . . . 95
5.3.1 Depth-First Search (`-d`) . . . 95
5.3.2 Disable Deadlock Checker (`-W`) . . . 95
5.3.3 Display Warnings as Queries (`-Q`) . . . 95
5.3.4 Change Size of Hash Table in "Passed" List (`-H size`) . . 95
5.3.5 Optimize Time Consumptions when Several Prop...(`-T`) . . 96
5.3.6 Unpack Reduced Constraint System Before Inclusion...(`-U`) 96
5.3.7 Do Not Display Copyright Message (`-q`) . . . 96
5.3.8 Run Silently Without Progress Indicator (`-s`) . . . 96
5.3.9 Print Diagnostic Trace to Standard Output (`-t`) . . . 96
5.3.10 Display Traces Symbolically (`-y`) . . . 96
5.4 Run-Time Experiments with UPPAAL . . . 97
5.4.1 Why Run-Time Comparisons are Problematic . . . 97
5.4.2 How to Read the Run-Time Charts (Figures 5.5–5.8) . . . 98
5.4.3 Fischer's Mutual Exclusion Protocol . . . 99
5.4.4 CSMA/CD Protocol . . . 100
5.4.5 FDDI Token Ring Protocol . . . 101
5.5 Reflection: Optimization Techniques for Real-Time Systems . . . 101

6 The Model Augmentation Technique **107**
6.1 Adding Parts to UPPAAL Models . . . 108
6.1.1 Formal Definition . . . 109
6.2 Soundness of Model Augmentation . . . 110
6.2.1 Suitable Augmentations . . . 111
6.3 Model Augmentation for Universal Path Properties . . . 112
6.4 Bricks Sorter Example . . . 114
6.4.1 The Bricks Sorter Model . . . 114
6.4.2 Augmentation of the Bricks Sorter Model . . . 115
6.5 Reflection: Model Augmentation . . . 118

7 Abstract Interpretation of Dense Real-Time **119**
7.1 Outline of this Chapter . . . 120
7.2 Abstract Interpretation . . . 121
7.2.1 Galois Connections . . . 122
7.2.2 Property Preservation over Kripke Structures . . . 122
7.2.3 Strong and Weak Preservation . . . 123
7.3 Predicate Abstraction . . . 124
7.4 Timed Systems with Restricted Delay Steps . . . 125
7.4.1 The Next-Free μ-Calculus . . . 129
7.5 Predicate Abstraction for Real-Time Systems . . . 134
7.6 Sets of Basis Predicates . . . 139
7.7 Refinement of the Abstraction . . . 141
7.8 Reflection: Abstractions of Real-Time Systems . . . 144

III Making Use of Hierarchical Structure **147**

8 Hierarchical Partitioning **151**
8.1 How to Group Together? . . . 152
8.2 The Tree-Indexing Problem . . . 154
8.3 A Greedy Algorithm to Partition Hierarchically . . . 160
8.4 Experimental Results . . . 164
8.4.1 Asynchronous Parity Computer . . . 164
8.4.2 Leader Election in a Ring . . . 167
8.4.3 Opinion Poll Protocol . . . 168
8.5 Reflection: Hierarchical Partitioning . . . 170

9 Model Checking Hierarchical Timed Automata **173**
9.1 Overview on the Flattening Procedure . . . 174
9.2 Flattening in More Detail . . . 175
9.2.1 Translation of Superstates and Entries — Phase I . . . 175
9.2.2 Exit of Superstates via Global Joins — Phase II . . . 178
9.2.3 Post-Processing of Channels — Phase III . . . 180

9.3 Semantic Correspondence of HTAs and TAs 180
9.3.1 Hierarchical and Flat Configurations 181
9.3.2 Correspondence of Steps 182
9.3.3 Correspondence of Traces 184
9.4 Model Checking a Cardiac Pacemaker 185
9.4.1 The Hierarchical Timed Automaton Model 185
9.4.2 Translation to Uppaal Timed Automata 187
9.4.3 Model Checking the Uppaal Model 192
9.5 Reflection: Flattening Hierarchical Timed Automata 194

Epilogue **197**

Bibliography **199**

Index **219**

Abbreviations **227**

Introduction
Formal Methods for Real-Time

I just want to make the point that reliability really is a design issue, in the sense that unless you are conscious of the need for reliability throughout the design, you might as well give up.

— A. G. Fraser, at the NATO Software Engineering Conference 1968

Analyzing a system amounts to exploring its behavior. A complete analysis makes it possible to predict *the behavior under all circumstances. Intuitively a system is correct if it always behaves as intended.*

Even for very simple systems a complete analysis is typically infeasible. Thus the quest for correctness is a story full of high expectations, special cases, and compromises. Hopes are low to include concerns for correctness a posteriori, when the design is done and dependencies are set. The advocated approach is to include these concerns as a part of the construction process. In short, try to make the system analyzable *with respect to interesting properties.*

One classic method to achieve correctness is to break down the problem into smaller sub-problems. A system is understood as a box with an input, an output, and a relation between both. This relation specifies *the intended behavior (e.g., [Bar96]). A box behaves correctly, if its contents guarantee to meet this specification. Now one can break down the box into a number of smaller boxes, each with its own specification. The correctness proof then relies on the assumption that all small boxes satisfy their specification. This assumption is discharged in the next step, where the small boxes are split up into even smaller boxes and so on. The process terminates when every smallest box is supplied with a direct proof to satisfy its specification.*

This approach is appropriate for systems of a purely functional nature. Often this is then visualized via a black box. One does not make assumptions on the inside of it, but only on the description what it does. This point of view is particularly helpful when reasoning about the correctness of algorithms. Algorithms perform a certain finite manipulation on the input. Classically this is deterministic, thus the manipulation can be seen as computation of a discrete mathematical function. The computation is correct if for every input the output is as expected.

However, most software written today does not fit in this picture. Some parts of the code certainly serve to compute functions, but the lion share is concerned with interaction of some sort, e.g., via a user interface. This is especially true for reactive systems, *where the prime functionality is not a general computation service. Rather the performed computations are merely* means *to provide a service of different nature, like coordinating the engine of a car or receiving a phone call. This requires repeated interaction with an environment. As Harel and Pnueli note in [HP85], a reactive system resembles a cactus with multiple inputs and outputs (Figure 1). This nature has to be taken into account during specification and development.*

The specification of interactions requires to reason about evolutions of the system. A simple relation between input and output is not adequate for this: the correct response can depend on what happened just before. One of Pnueli's suggestions is to use dialects of temporal logics *for the specification. These logics basically allow to relate a present state to future and past. By means of a logical formula one can express properties like "whenever* a *happens, then* b *happens some time later." A number of such properties constitutes the specification. Relations of input and output are a special case of temporal logic specifications.*

The process of establishing or refuting a property for a given system is called verification*. Formal verification performs this task in a domain that is defined with mathematical precision. The object of analysis is a* model *of the system, and not the system itself [Bru95]. Depending on the domain of application, the gap between a system and its (mathematical) model can be significant to non-existent. An aero-plane caught in a thunderstorm can experience jolts and pushes that are not present in an idealized model, whereas an assembler program executing on a microcontroller can be expected to behave exactly as modeled.*

Many reactive systems are in fact also real-time systems: their correctness not only relies on the occurrence of an operation, but also on the timing of it. This dictates to include a real-time component in the specification—and also in the model: operations take time. Developers of real-time systems had to experience that this adds an extra dimension of complication. During development the analysis is performed on a model of the system. Thus it is necessarily relative to assumptions one the timing behavior of the real system, and in last consequence of the underlying hardware.

This thesis discusses formal verification techniques for real-time systems. The main problem with automated verification is the high computational complexity. We make use of structural information, like hierarchies and loops, to battle this problem. Our methods of choice to improve efficiency are approximations and abstractions. It is indispensable to assert the soundness of such techniques. In this Chapter we outline scope and range of the treated topics.

0.1 Doing it Right: Correctness

It is fair to say that correct systems are as valuable as gold.[1] *Even more—they are as rare. In this Section we discuss what correctness* means *in first place.*

Correctness is intrinsically relative to some *description* of what the system is supposed to do. In software engineering such a description is also called specification. A formal analysis always requires a formal specification.

For example, it is impossible to assert that the implementation of the UNIX command `ls` (which lists the contents of a directory) is correct. The manual page of the command can be seen as a specification. But this specification is written in natural language and leaves room for ambiguities. For example, the option `-R` is supposed to recursively list subdirectories. Now there is the possibility to have a symbolic link pointing to a directory. Is this a subdirectory? If yes, should the directory also be listed when it is in fact an ancestor of the current directory? The `ls` program would not terminate in this case.

Whether you get what you expect can also depend on factors not under the control of the implementer. Is it correct, if special characters are displayed as escape codes? Is it intended that parts of a directory tree might be listed, even if you do not have read permission for them? Is the report that something cannot be listed already a violation of secrecy? If the number of files in a directory is excessively large, `ls` refuses to list them. Is this a bug or a feature?

It can be argued that recognizing a "bug" is easy, once it occurred: something undesirable happened. Catastrophes like the Challenger disaster[2] or the famous Pentium FDIV bug[3] were certainly not intended behavior of the systems.

It does not always matter how "big" the fault is. In theory, every one bit can change everything. The risk increases enormously with the number of small gears that have to work together in union.

[1] See, e.g., [Bar96].

[2] On 28 January 1986 the world was shocked by the destruction of the space shuttle Challenger and the death of its seven crew members. A decade after this national tragedy, the world wide web hosts a variety of resources. The Space Policy Project of the Federation of American Scientists collected those at a Challenger Accident homepage: http://www.fas.org/spp/51L.html.

[3] The floating point arithmetic of the first Pentium computes the wrong division results for a small set of inputs, see http://support.intel.com/support/processors/pentium/fdiv/wp/.
This is not the only fault of this microprocessor and possibly not even its worst. Intel's list of errata includes a so called Pentium FO bug that can cause the processor to deadlock. For virtually every high-performance microprocessor dozens of failure scenarios are known.

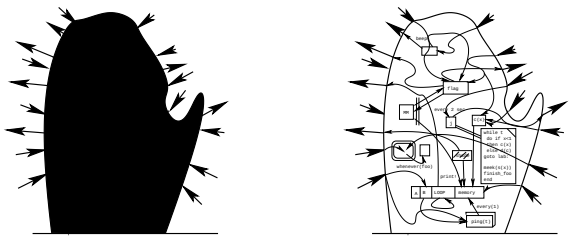


Figure 1: Reactive Systems Resemble a Cactus Rather Than a Box. Various Dependencies can Exist Between Various Inputs and Outputs.

To talk about correctness it is not sufficient to determine what is wrong behavior; more importantly it has to be defined what is *right*. This means that before building a system, one needs to describe more or less exactly what it should do. Natural language is not an adequate vehicle for that, since it can harbor ambiguities. Also it is difficult to assert that all cases are covered.

To exclude all ambiguities it is unavoidable to resort to a language with mathematical precision, *logic*. This does not as much dictate how a statement looks, but more importantly what a statement means. Every logic language prescribes a set of allowed statements (syntax) and how exactly they have to be interpreted (semantics).

The basic unit of observation in a system is commonly known as a *state*. At any point in time one could in principle take a snapshot and extract status information: what bits are up or down, which instruction is currently executed, and so on. A state consists of the combination of all this information.

Propositional and first order logic can be adequately used to express properties of a state. What arises typically is the need to reason about computations over sequences of states. Dialects of *temporal logic* [Pnu77, CES86, Pnu86] can naturally express relations between subsequent states. Temporal operators allow expressions like "all future states" or "some future states". Depending on the dialect, analogous operators may or may not exist for the past.

In practice logics can be tailored for a specific domain of application. Designers are likely to use what suits their needs best. Some aspect may well be described in a different language: the weight of a micro-chip is unlikely to be expressed via logic.

In this thesis we concentrate on the part of a specification that is a set of logical formulas. We refer to a system as "correct" if it fulfills all these logical properties. This does not imply that the right things happen. Specifications can be wrong or incomplete. Even if a property holds, the analysis can be unable to prove it. The benefit of using a logic language is primarily that there are no two ways about what is said and what is true.

Using a logical language solves part of a communication problem that has to be taken serious. In September 1999, NASA lost a Mars Climate Orbiter due to a simple mistake: a piece of ground control software carelessly calculated values in

inch instead of using metric values. Not only is it strange that this error was not spotted. It is amazing that an error like this could *occur* in first place. Obviously some of the exchanged data was incomplete or ambiguous.

The remedy is rigorous and simple: all technical data has to provide all relevant information in an unambiguous way. In the design of complex systems this philosophy has a name—*formal methods*.

0.2 Formal Methods

Formal methods is the collective term for development techniques that apply a purely mathematical model of system and specification. We give some background information and highlight both problems and benefits.

0.2.1 The Necessity of Being Formal

Formal methods were put to practice in the past two decades. The core elements are (1) a description language with well defined semantics and (2) a logic framework to reason about formal statements. The language is meant to grasp the essence of a system in a mathematical manner. Thus desired properties can be expressed as conjectures. With an adequate method of inference, a property can either be proved correct (verified) or refuted—typically via the construction of a counterexample. Ideally the refutation not only uncovers the flaw but also gives an intuition *why* the property does not hold. This makes the task of correcting the flaw much easier. With the modified system a new attempt to establish desired properties can be made. This yields a simple iterative development process.

The mathematical nature of a description comes with several benefits. Statements about the system and its shortcomings can be communicated clearly within a group of designers. The ability to write formulas down leaves no room for ambiguities, though misunderstandings are possible. Via commitment to a mathematical formalism, advanced support by machinery is feasible. Computers are very good at doing simple manipulations on large amounts of data and at being pedantic about every detail. The *understanding* of the formalism, however, has to be added by means of a human factor. At FMCAD'98, Carl-Johann Seger noted that

> If you use formal verification as a black box, it is pretty useless most of the time.

Formal methods do not come for free—to apply them, a number of obligations have to be met. First, a choice of the right formalism has to be made. This has a research component, since the expressive power of several formalisms is not completely understood. Comparisons of proposed approaches with each other are often on the level of argument rather than proof. Second, formal methods are typically hard to learn, since they base on mathematical concepts. They require a high-level understanding of the connection between logical foundations and the actual design. This

requires experts, or at least education for the working engineer. Third, formalisms are tedious and error-prone if handled manually. Making everything explicit entails voluminous descriptions and reasoning on a low level. This requires tool support.

Efforts have been made to develop universal and widely accepted standards to ease these burdens. Organizations like the W3C or the OMG develop industrial standards in information technology. For example, the latter authors the Unified Modeling Language (UML), see Section 1.1. Though not primarily a formal language, it can be used as a starting point to develop formal frameworks.[4]

0.2.2 Verification Engineer: A Future Profession?

The application of formal methods requires from the designer a strong background both in the foundation and in methodology. Expectations are pessimistic that trained designers are available soon in sufficient number to apply formal methods on the large.

Amir Pnueli communicates a vision of a new profession called *verification engineer* in his Turing award lecture [Pnu97]. Pnueli notes that

> There is no purely scientific solution to the system correctness problem.

Instead the way to go could be the less rigorous engineering approach. A verification engineer would be knowledgeable in many tools and methods, among which formal verification should be a major player—but not the only player. A most effective set of tools would also include a number incomplete methods, like the strategic generation of test cases [Nie00]. In place of a deep understanding of any one method the emphasis is on a merely conceptual knowledge about a large number of methods. Elements of the daily work would be the choice of the right method, making compromises between cost and benefit, and following standardized and tried procedures.

More than four years after this lecture, verification engineer is not yet an established profession. There are no specialized educations, no degrees, no job descriptions to be found. However, a considerable number of people in corporations apparently *work* as verification engineers. This is visible through their presence in academic conferences like CAV, CADE, TPHOLs, and CHARME, where major corporations like Intel, NASA, ICASE, IBM or Microsoft Research are listed as affiliations.

It remains to be seen whether verification engineering will crystallize to a profession with standard education programs, techniques, and skills. It seems fair to state that formal methods have their place in the industrial design process and thus necessarily also the people applying them have their place.

One indication for this is the presence of tools like theorem provers and model checkers in industrial design. Applied formal methods move the burden from human designers partially to automated processes and reproducible facts. This is not only beneficial but necessary, as Wolper noted in [Wol98]:

> Manual verification is at least as likely to be wrong as the program itself.

[4] See AIT-WOODDES on page 71.

0.2.3 Automation, Automation, Automation

It might be surprising that some decades after the rise of formal methods tool support is still very basic. Case studies are performed by researchers rather than by engineers. Only few methodologies are off-the-shelf. There are reasonable explanations, why advances are delayed.

One explanation is that finding the "right" formalism is crucial to further tool support. Small variations tend to have vast consequences. It can take years to realize the effects in detail, since the development of tools in an academic setting is tedious and often overshadowed by political decisions. In the search for tool design, an asymmetry is encountered, often referred to as the 80-20 rule: 80% of the functionality can be provided by 20% of the effort, while the remaining fifth is much harder to cover (e.g., in [COR+95]). The *computational complexity* of the problems dictates restrictions, both on automation and on expressiveness of formalism. Understanding the computational complexity can be a guide in making wise choices. To give an example, in the UPPAAL tool, usage of clock values in the transition guards have been rigorously restricted. Allowing more general constraint expressions here would make the task of deciding simple properties of the model computationally much harder.

It has also been pointed out that computational complexity occasionally gives a wrong intuition, since it is classically *worst case* complexity. This can be skewed with respect to the set of inputs, on which the algorithm is actually applied.

An illustrating example of this misalignment comes from operations research. Here the *simplex algorithm* is the most popular and in practice also most efficient algorithm for solving linear programs, though it has a worst-case exponential lower bound and polynomial algorithms are known. A recent paper explains this paradox by redefining the measure of computational complexity according to assignments that are *almost* solutions of the problem [ST01]. According to this measure, which is a hybrid between worst-case and average-case analysis, the shadow-vertex simplex algorithm is polynomial.

In formal verification, an algorithm with high inherent complexity is not necessarily useless. The MONA tool [BK95, HJJ+97, KM01] demonstrates that an algorithm with a stack-of-exponentials[5] lower bound can have reasonable applications in practice. A significant amount of software engineering had to be invested before the tool was up and running.

There is a wide gap between theory and practical application. Industry is reluctant to invest man-power in technology that is not fully established and understood. Thus, tools applied on a daily basis in an industrial setting tend to limp behind the academic state of the art. To give an example, the recent commercial tool TUXEDO-LEC™[6] basically does equivalence checking, advanced properties are

[5] I.e. the value expressed by the term $2^{2^{\cdot^{\cdot^{2}}}}$, with height n; sometimes referred to as *tower*(n).

[6] TUXEDO-LEC™ is released by Verplex Systems, a company founded in 1997. For more information see http://www.verplex.com/lec%5Fbrochure.html.

not addressed. While this gap is not surprising, it is exceptionally painful considered the rapid growth in size and number of systems designed today.

Academy is faced with an additional assessment problem: it is often hard to apply and test new technology on realistic examples. Most modern designs are considered mental property. Therefore companies hesitate to open up their treasure to a scientific community, where every result and insight is considered public.

As yet, algorithms and tools fail to automate verification of the high-end of industrial designs. We cannot expect to scale up to this by merely making the machines faster by a factor; if the input size doubles, the problem gets more difficult by orders of magnitude. Success relies vastly in the detection of optimizations and efficient ways to represent the data. The challenge is to grasp what is *essential* in a design. If we can guide our algorithm to identify the crucial controls, we can expect it to operate them.

0.2.4 What are Reasonable Hopes?

It is doubtful whether automation in formal verification will ever reach a level where the human interaction is reduced to the task of pushing a button. Machinery is not likely to replace understanding of a complex system. Sometimes formal methods cannot possibly guarantee correctness of the system, as it is the case when significant abstractions are applied.

What is reasonable to expect is tool support for experts. This can free the designer from many gory details, allow for the re-use previous work, and act as reliable scribe in long-term and multi-people projects. The ideal tool can also hide information without neglecting it. Not every person working on the project would necessarily have to understand all foundational details. The underlying formal methods could "disappear" into familiar environments like simulation [Rus00].

Partial application of verification technology can also be useful. Formal methods proved to be very successful in uncovering bugs that ran undetected by traditional techniques, like intensive testing.

Design methodologies often start out with a high-level description of a system. This description is likely to be in the scope of algorithmic treatment. Simulation can be used to validate the intuition of the designer. Formal verification can be applied to verify that the design fulfills some high-level constraints. Those may be far from trivial. E.g., one can assert timing constraints under some assumptions on the duration of primitive operations.

In some cases the problem is small enough to be manageable. Not every faulty chip etched today has the dimension of a Pentium. And like many communication protocols or simple software components, they lie within the reach of tool support and could be verified with algorithms out of the box. Thus it is an investment to build experience in this technology, with obvious short-term benefits and a reasonable chance for revenue in future designs. Parts of the industry seem to have adopted this thought. Companies like AT&T, Bell Laboratories, Cadence, IBM, Intel and Motorola started formal verification programs in the recent years, often kept internal

and on a high level of confidence.

For the interested reader we recommend [Hal90] and the followup [BH95]. A tool engineer's point of view on formal verification is recorded in [Ste98].

0.3 Techniques for Formal Verification

Formal verification is the process of establishing or refuting that a system conforms to a specification. We briefly outline some of the most prominent techniques: automated theorem proving, process-algebraic methods, step-wise refinement, abstract interpretation, and model checking.

0.3.1 Automated Theorem Proving

Automated theorem proving spans a wide spectrum of approaches and software systems that can be classified with respect to the *level of automation* they provide.

On the low end there are proof checking formalisms, like LCF [GMW79]. Here the task of establishing soundness of every step in a formal reasoning system is mechanized. Though definitely useful in many settings, the *construction* of the proof is entirely left to the user. Since many proofs are long and complex, this technology alone is rather a building block than a tool itself.

A long-term project aiming for machine checkable proof formalisms is Mizar [Rud92]. The core of it is a language based on set theory formulated in logic. The proof checker, originally implemented by inference rules, uses model checking in the later versions.

Many researchers advocate *semi-automated techniques*. This is incorporated by tools like Ehdm [RvHO91], SDVS [BILT92], PVS [COR+95], HOL [MT93], or its offspring Isabelle [Pau94]. Here the tool helps organizing and documenting the constructed proof in a reproducible way. Typically the *re-usage* of lemmas (or libraries of them) and high-level proof strategies are supported. Moreover, powerful automation can be applied in special cases that occur frequently. For example, the validity of propositional formulas can be established without human interaction. So the user is free to concentrate on the interesting or challenging parts of a proof.

Some tools support a high or complete level of automation. Examples for this are the Boyer-Moore theorem prover (Nqthm) [BM79, BM88]—originally a fully-automatic theorem prover for a logic based on a dialect of Lisp—and the resolution-based theorem prover Otter [McC90, WOLB92]. The price for the automation is either a restricted language or an incomplete analysis. If time or memory of the machinery do not suffice to fulfill the task, it is up to the user to decompose the problem into smaller subgoals.

0.3.2 Process Algebraic Methods

A process algebra is a calculus where the first class citizens is a process. Operations allow to build more complex processes from simple ones [Hoa85, Mil89]. Typically

the simplest processes is the empty (idle) process and the atomic unit of information is the occurrence of one event. Examples for operations are sequential and parallel composition, hiding of events, synchronization and non-deterministic choice. Algebraic laws state the equality of processes. For example, the choice operator can be commutative.

One primary motivation of process algebras is the study of an abstract computing process. For this a large number of calculi have been developed.

0.3.3 Stepwise Refinement

One classic method of software development is the *top-down approach*. From an initial specification, one would derive a design on an abstract level. This would further be more concrete by adding details, until it reaches an implementation level and actual code is generated.

Conceptually this process is a *stepwise refinement*. If a strict discipline is followed, one is able to proof that the design version n is a *refinement* of version $n-1$, in the sense that is introduces intermediate steps, but maintains both data- and control flow.

Systems of functional nature can be broken down into smaller and smaller *"Chinese boxes"* [Bar96]. Referential transparency restricts the mutual dependencies to the caller/callee relation. This is adequate for designs where the level of distribution is low and the problems are rather an algorithmic than an architectural challenge.

For systems that can be described as reactive, the refinements amounts to the traversal of a *"magic square"* [HP85]. Along one dimension the level of detail increases, from high-level sketch to actual implementation. Along the other dimension the description of behavior increases. This includes interactions of distributed parts. Statecharts are advertised as an adequate vehicle to traverse this square, since they allow for successive additions of detail.

0.3.4 Abstract Interpretation

This approach simplifies the analysis of a concrete system by interpreting its operations in another (smaller) universe [CC77]. If a property can be established this way, it also holds in the original system. The conditions for connecting concrete and simplified system are rather general, thus this is a very flexible technique.

Abstract interpretation has a strong tradition in the area of program analysis, where standard abstractions for primitive data types are known. The prominent method is the fixed point computation over lattice structures [NNH99]. In this setting the method is typically applied *a posteriori:* the way the system is described is not influenced by the applied analysis.

A major application area is compiler construction. Information about, e.g., dead code allows for obvious optimizations. If the analysis fails to detect a true property, this is acceptable: the compiler merely misses a possible optimization step. Instead

of focusing on any *one* property, it is more important to detect as many properties as possible in an economic way.

Recently this methodology has also attracted attention in the context of reactive systems [Kel95, Dam96, CC00].

0.3.5 Model Checking

Model checking is a fully automated technique for the verification of finite state systems. It emerged in 1981 from the work of two pairs, Queille & Sifakis and independently Clarke & Emerson. Given a system and a temporal logic formula φ, all states where φ holds are computed in a recursive fashion. For every sub-formula ψ of φ the states where ψ holds are identified by applying this step for ψ and so on, terminating with primitive properties. For more detailed information we refer to [CGP99, Kat99].

Model checking requires a search of the complete reachable state space, which is often excessively large. This phenomenon is known as *state explosion problem*. Approaches to battle this problem are still a busy research topic. Many fall in the categories of *symbolic techniques* and *partial order reductions*. Symbolic techniques represents the state space in an efficient way by combining equivalence classes of states [BCM$^+$90]. Partial order reductions cut down on interleavings of transitions by allowing only one path where several paths are equivalent [JK90, GW91].

The evolution of model checking is strongly connected with one symbolic technique. It started with Bryant's work on *binary decision diagrams*, or BDDs for short [Bry86, Bry95]. A BDD is a directed rooted acyclic graph with two terminal nodes, 0 and 1. The intermediate nodes are labeled. On a traversal of the graph these labels occur always according to a fixed order, and each label occurs at most once. BDDs can be used naturally to encode Boolean functions. The nodes correspond to variables and the next edge taken in the traversal depends on the value of the variable. The terminal node gives the value of the function, 0 or 1.

Interestingly this representation is *canonical*, i.e., for a fixed variable ordering any Boolean function has exactly one BDD representing it. By this virtue, some operations—like checking for equivalence—can be performed very efficiently.

Now a set of states can also be encoded by a Boolean function, i.e., by a BDD. Under certain variable orderings, many such BDDs are in fact small. This phenomenon can be used effectively for model checking [McM93]. Today a large number of BDD variations exist that adopt also to other areas of application.

0.3.6 Combining Techniques

Formalisms strive to preserve as much information of the original design as possible, but some techniques require an abstraction step. There is future potential in the combination of powerful techniques, like (semi-)automated theorem proving and model checking. As an example we list [SS99], where Boolean abstractions of a

bounded retransmission protocol are computed with aid of a theorem prover (PVS). This abstraction is then model checked, while preserving properties from the full μ-calculus.

One step further in this direction is the Symbolic Analysis Laboratory (SAL) [BGL+00]. In SAL an intermediate high-level description language builds the core of an analytic platform that connects to a variety to different tools. Information generated by one tool can serve as input to another one. For example, a model checker could expand the state space of a system symbolically. This state space is also the *strongest invariant* of the system and can be used in a theorem prover to aid an inductive proof.

So far only the surface has been scratched in the combination of techniques. One reason might be that it is difficult to find an expert that is fluent in more than one. We believe that there is a significant potential to be explored.

0.4 The Design of Real-Time Systems

Design and development of real-time systems originated in control theory, where the micro-chip is understood as a machine part. Discrete analysis methods can aid this process to a certain extend, but have to cope with a number of difficulties.

A *real-time system* is one, where the correctness not only depends on the functionality but also on the timeliness of this functionality. A system where a computational device is part of a machinery, whose prime purpose is not a computing service, is called an *embedded system*. Typical examples are wristwatches, mobile phones, or car engines. The digital part of an embedded system tends to have limited computation power and a very restricted amount of memory. Since these devices are manufactured in large number, the cheapest hardware sufficient to fulfill the task is used. Most embedded systems are both real-time and reactive systems, i.e., they interact continuously with their environment in a timely manner.

We note that classically embedded systems do not have a processor at all. They rather solve the computational problem by means of bare wires or simple circuits. These *analog computers* have little in common with their digital cousins. They have no instruction set, no clock cycle, and no memory other than capacity. As an engineering discipline, control theory has long since—and successfully—treated the problem of coordinating, stabilizing, or correcting continuous physical processes by these means

The concept of using *digital* controllers is controversial; traditionally, controllers are synthesized as solutions to differential equations. Those solutions are closely matched by analogous circuit parts like integrator, inverters, or filters. Using digital technology dictates a different infrastructure, different physical layout, and a different realization. Most importantly, it requires other methods for synthesizing an adequate controller.

One of the introduced complications is the need for *scheduling*. Since not all computations are performed in an analogous manner, a number of computation

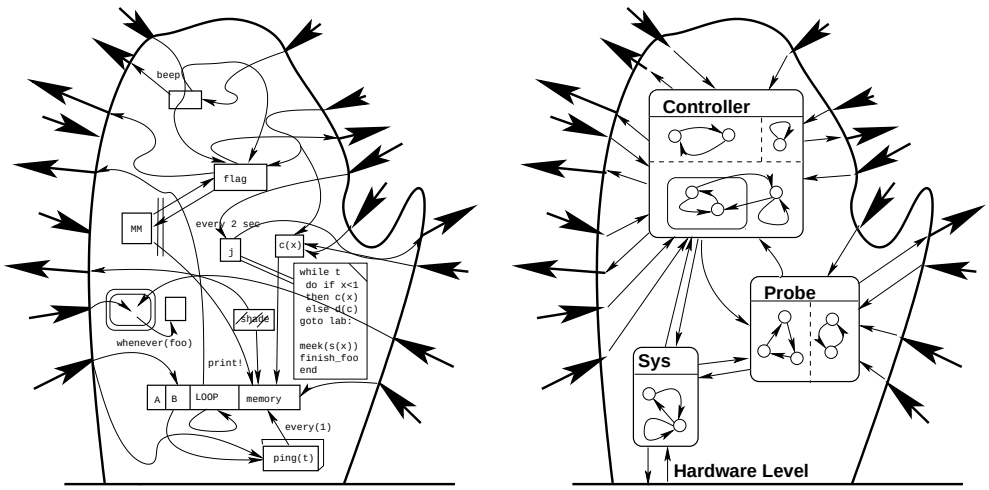


Figure 2: Unstructured and Structured Description of an Embedded System.

tasks have to share the processor. The timeliness of the system is then equivalent to *hard deadlines* that every task has to meet. If a hard deadline is missed, the input data is too old to be useful.

For a *soft deadline*, late data is still good data. It is not catastrophic if a soft deadline is missed. It has merely to be guaranteed that the deadline is met on the average. If both hard and soft deadlines exist for a computation, both together constitute a *firm deadline*. Most systems commonly declared as soft deadline are in fact firm deadline systems.

Such systems needs a *scheduling policy* that guarantees that all hard deadlines are met. Since time is consumed mainly by executing machine instructions, this boils down to counting instructions and computing the longest possible execution path.

How difficult it is to perform this analysis depends on the nature of the tasks. An important factor is whether tasks execute in inseparable blocks or can be put on hold before finishing their computation. An example is a classic paper on rate monotonic scheduling by Liu and Layland [LL73]. Their results give sufficient conditions to guarantee that all task meet the deadlines. Similar results are lacking for dynamic scheduling policies. Numerous technical problems—like context switch, unpredictability, latency, or jitter—render the actual creation of a running system a challenging problem for engineers. The designers who perform the crucial steps tend to be experts on the hardware they control. The analysis can be automated only to a limited extend.

0.4.1 Discrete Analysis Techniques for Real-Time Systems

Most commonly the analysis of a system design is required *a priori*, i.e., before the actual system can be build. This fits well together with model based develop-

ment [Bru95].

Adaption to Time. Many of the verification techniques in Section 0.3 have been adapted to real-time. For *automated theorem proving*, the applicability for timed settings has been explored, e.g., in [Sha93]. For *process algebras*, a number of calculi addressing time as a primitive exist. Examples for this are Timed CSP [RR88], TCCS [Yi90], and ATP [NS94]. For *stepwise refinement*, time settings have been explored, e.g., in [SZJ94]. For *abstract interpretation*, approximation of real-time safety properties can be formulated in the have been formulated in this framework [WT94, DT98]. An approach to approximate also liveness is presented in Chapter 7.

Timed Automata Model. For model checking, the *timed automata model* of Alur and Dill [AD94] is most influential for this thesis. The language is basically an extension of finite state machines with additional real-valued variables that represent clocks. Transitions can be guarded by Boolean expressions over these clocks and clocks can be reset when taking a transition. Properties of timed automata can be expressed in timed dialects of logic. If the guard expressions are chosen from a restricted language, the model checking problem remains decidable [ACD93]. Since we interpret time over a dense domain, the state space is infinite. The analysis has to rely on equivalence classes, i.e., states are treated symbolically.

Duration Calculus. One of the specialized techniques for timed settings is the *duration calculus* [CHR91]. Here a formula can reason about the duration of states without explicit mentioning of the absolute time. E.g., the formula $\int P \leq 5$ holds for the interval $\mathcal{I}$, if during $\mathcal{I}$ the system is at most 5 time units in state P. In addition to the usual Boolean operations the logic features a *chop operation* on intervals. $(D_1; D_2)$ is true for an interval $\mathcal{I}$, if $\mathcal{I}$ can be partitioned into two consecutive intervals $\mathcal{I}_1, \mathcal{I}_2$ such that D_1 holds in $\mathcal{I}_1$ and D_2 holds in $\mathcal{I}_2$.

In general the calculus is undecidable. For certain fragments, however, a formula can be decomposed into an un-timed controller communicating in an asynchronous manner with timers [OD98]. This closes the gap from specification to implementation for a restricted class.

0.4.2 Increase in Complexity

As most designs, real-time systems increased in complexity. Traditional analysis methods often turn out to be inappropriate to meet the new challenges.

One answer to increase of complexity is the introduction of more structure, In particular hierarchies can help to cut down a complex system into manageable parts. In [HP85], Harel and Pnueli suggest to use the statechart formalism to build this hierarchies (Figure 2). This requires tools and analysis techniques to manage this notation.

Another dimension is to introduce abstraction layers. Real-time operating systems, cast in silicon, allow for starting the composition of controller programs on a higher level. The OSes could take care of task management, priority levels, dispatch time, and memory management. Designing a general purpose real-time OS is not easy.

0.5 The State of the Art

It is subtle to determine what problems today are in the scope of formal treatment. First, only a limited number of real-wold design task was addressed with formal methods. Before the first try, it is commonly believed that it is impossible.

Second, new techniques are part of an industrial manufacturing process. If successful, they become immediately a competitive advantage and are regarded as intellectual property. Experiences, tools, and expertise are not necessarily shared freely.

Thus we can only give a vague estimate of the state of the art. We list a number of projects and corporations that apply formal analysis techniques in a larger scope.

One of the largest formal method projects in the last years is the driverless METEOR line 14 metro in Paris. 115˙000 lines of specification compile into a 87˙000 line ADA program. Correctness was established with interactive theorem proving. This required to handle 27˙800 proof obligations. To discharge them, 1˙400 rules were added to the theorem prover and proved correct; for 900 of them this was possible without user interaction. The necessary manpower is listed as 600 person-years. As yet, no errors were claimed to be found in software or specification (see [CC01]).

Practitioner Johnson noted in 2001 [Joh01]:

> Formal analysis is outside the mainstream of system design practice.

The main reasons are lack of educated people and higher development cost. Developing proof carrying code, for example, is regarded roughly six times as expensive as ordinary code.[7] It is arguable, whether initial high cost is over-compensated later by savings in debugging or maintenance. However, there are examples where formal methods are applied on the large scale.

John Hatcliff guides the *Bandera project*[8]. The declared goal is to derive static properties of **Java** programs. In [CDH$^+$00], a combination of state-of-the art methods—like abstract interpretation—is used to establish safety properties.

The hardware corporation Intel uses theorem proving (in addition to testing and other validation disciplines) to develop components of today's microprocessors, e.g., the floating point unit [KK01].

[7] Richard M. Soley, Chairman and Chief Executive Officer of the Object Management Group (OMG), in a panel discussion at ETAPS'2000, Berlin.

[8] http://www.cis.ksu.edu/santos/bandera/

Under the supervision of Thomas Ball and Sriram K. Rajamani Microsoft Research launched the *SLAM project* [BR01].[9] The objective is to analyze device drivers written in **C**. The applied methodologies encompass static analysis, Boolean and Cartesian abstractions, and model checking.

Also other industrial activity reflects the development of formal methods. A number of corporations develop commercial tools in this area. Examples for analysis tools include the Tau SDL[10] Suite from Telelogic[11], VISUALSTATE from IAR Systems[12]. Some examples for other companies concerned with formal analysis technology are AT&T[13], Esterel Technologies[14], and Prover Technology[15].

Challenges. Design and analysis of complex systems is far from being a solved problem. Important lines of work in the field include the enhancement of automation, the cooperation of analysis tools, and the standardization of modeling formalisms. For real-time, we address some of these challenges in our thesis.

0.6 Outline: A Guided Tour Through This Thesis

In the following we give an overview of this thesis, line out the scope of the Chapters and highlight the contribution of the author.

This document reports both on things we discovered and things we built—sometimes alone, sometimes in collaborating with other researchers. It also contains overviews on topics that are relevant for the context and motivation of our work; we summarize, where an appropriate compendium does not seem to exist.

Important keywords are collected in the Index (page 219). We apologize for the multitude of used acronyms, especially in Chapter 1. As a partial remedy, a complete list of acronyms is given on page 227.

Part I

Chapter 1—UML and Statecharts. This Chapter provides background on relevant developments in context of the unified modeling language (UML) and David Harel's statechart formalism. The author of this thesis presents this as a summary without own contributions.

[9] http://www.research.microsoft.com/projects/slam/
[10] Specification and Description Language, see http://www.sdl-forum.org/SDL/index.htm.
[11] http://www.telelogic.com
[12] http://www.iar.com
[13] http://www.att.com
[14] http://www.esterel-technologies.com
[15] http://www.prover.com

Chapter 2—UPPAAL Timed Automata. The author has been part of the UPPAAL development group during the last two years. This included participation in discussions, writing of bug-reports, drafting answers for the mailing list, and maintenance of official web pages. The understanding of the tool reproduced in this Chapter stems from this time.

Published material: The trace semantics of UPPAAL timed automata is to appear in a condensed version in the workshop on "Theory and Practice of Timed Systems" (TPTS 2002) as part of [Möl02].

Chapter 3—Hierarchical Timed Automata. Syntax and semantics of this formalism was discussed and constructed together with Alexandre David, under kind advice of Wang Yi. We designed the `.xml` grammar for this language and revised it after discussions with Emmanuel Fleury.

Published material: A revised draft of syntax and semantics is available in the technical report [DM01]. A condensed version is to appear in "Fundamental Approaches to Software Engineering" (FASE'02) [DMY02].

Part II

Chapter 4—Algorithmic Verification of Real-Time Systems. We provide a summary of background information on real-time model checking as incorporated in the UPPAAL tool. The algorithms were neither developed nor implemented by us. Rather this Chapter presents a high-level description of the implementation. We use the trace semantics from Chapter 2 to formulate a correctness proof. There is no published material associated directly with this part.

Chapter 5—Efficiency in Real-Time Model Checking. This Chapter features experimental data but no technical results. The description of the optimization options is given to make up for the lack of a description in a manual. We conducted the experiments and compiled the data to a readable form.

Published material: The experimental data is made available from the official UPPAAL web pages, http://www.docs.uu.se/docs/rtmv/uppaal/benchmarks/.

Chapter 6—Model Augmentation. Both technical material and case study are sole work of the author.

Published material: A condensed version of this Chapter is to appear in the workshop on "Theory and Practice of Timed Systems" (TPTS 2002) [Möl02].

Chapter 7—Abstract Interpretation of Dense Real-Time. The technical content of this Chapter is joint work with Maria Sorea and Harald Rueß.We extended the presentation to fit the broader context of abstract interpretation.

Published material: A preliminary version of this Chapter is available as technical report [MRS01] and a condensed version (with the proofs cut out) is to appear at the workshop on "Theory and Practice of Timed Systems" (TPTS 2002) [MRS02].

Part III

Chapter 8—Hierarchical Partitioning. This work was advised by Rajeev Alur. Our contributions are the formulation of the algorithm, the experimental implementation in MOCHA, the construction and comparison of benchmark examples, and the *NP*-completeness proof.

Published material: With a minor exception, all of the above work is documented in the technical report [MA00]. A condensed version appeared in "Correct Hardware Design and Verification Methods" (CHARME'01) [MA01].

Chapter 9—Flattening Hierarchical Timed Automata Starting point for this Chapter is the formal semantics of HTAs from Chapter 3. The reference implementation of the flattening procedure[16], the documentation, the description, and the pace maker case study are work of the author.

Published material: The flattening procedure is first outlined in [DM01]. A one page abstract appeared in the "13th Nordic Workshop on Programming Theory" (NWPT'01) [DMY01]. The paper to be published in "Fundamental Approaches to Software Engineering" (FASE'02) [DMY02] contains only a sketch of the flattening procedure. The Nordic Journal of Computation invited to submit an extended version.

Other Records

During the previous two years we enjoyed working as a member of the UPPAAL group. As an effect, we co-authored a number of papers in context of UPPAAL [ABB+01, BDL+01a, BDL+01b] and AIT-WOODDES [ADF+01]. Though reflecting relevant background information, these publications do not share technical content with the material presented in this thesis.

[16] See also http://www.brics.dk/%7Eomoeller/hta/vanilla-1/.

Part I

Modeling of Real-Time Systems

A man with a watch knows what time it is.
A man with two watches is never sure.

— Segal's Law

Real-time systems are systems where timeliness is essential to correctness. Consider the airbag in a car. It does not suffice to establish that after a collision it will expand—it is also crucial that this happens neither too early nor too late.

Such systems play a major role in industrial design and development. In particular embedded systems, whose primary purpose is to offer some service other than computation itself, outnumber full-grown processors by an order of magnitude: most computations today are done on small, weak, and cheap hardware. These systems are found in everyday use and range from simple appliances like wristwatches and remote controls to complex designs, such as mobile phones, cars, and airplanes. Most embedded systems are also real-time systems.

The timeliness of actions adds to the challenge of manufacturing a working system. Typical development breaks down a design into a number of interacting components. This entail the need to communicate the exact description of any part, but nevertheless be able to describe a complex design on a high level of abstraction. Most developments processes involve trail and error. So it is very desirable to validate a design long before the first prototype is built.

One way to address this needs is to use models of the system throughout the design phases. Models can be scaled physical systems, mathematical equations, or functional representations. As design progresses, more and more detail is added to the model or the models.

We are focusing on the analysis of the timed aspect. In the past ten years the formal verification community has developed strong analytic methods for reasonably abstract models of timed systems. One of the most thoroughly studied formalisms is the timed automata model [ACD93]. Some of the analytic methods—and these are our focus—are fully automated. An algorithm can give answers to queries like "will the airbag always come out in time." If this is not true in the (current) model, a counterexample scenario can be generated that shows the designer why it does not hold. Ideally, this counterexample can be used to fix the problem.

However, the mathematical (and usually restricted) models are not the first choice language of a designer. Her modeling language should be shaped according to the crucial design problems and not according to the analysis. We note that there is a gap between the industrial engineering practice and the models used in the formal verification community. This part of the thesis is devoted to bridge this gap in a prominent case: we connect UML statecharts to the timed automata model as used in the Uppaal *model checking tool.*

Our enterprise makes it necessary to explain the UML and the associated statecharts model to some detail (Chapter 1). We also give a formal definition of Uppaal *timed automata model (Chapter 2). As a synthesis we offer a timed model that is halfway between UML statecharts and* Uppaal *(Chapter 3). This can serve as an intermediate step in the formal verification of UML statecharts with respect to timing aspects.*

An important property of our formalism is that verifying specifications expressed in a fragment of timed logic remains decidable. This allows for fully automated analysis of this language. We use this property later to apply model checking on our formalism: in Chapter 9 we report on a flattening procedure that translates our formalism faithfully to Uppaal.

Chapter 1

UML and Statecharts

The multitude of books is making us ignorant.

— Voltaire (1694 - 1778)

It is a frequently repeated fact that systems designed today are far more complex than even the day before. Consequently the development cannot be the task of gifted individuals, but rather of large teams collaborating in different roles.

The need to distribute complicated design tasks over a large and heterogeneous number of designers entails a communication problem. Attempts to agree on standard terms and notations led to the advent of modeling languages *such as Booch, OMT, OOSE, or UML [BE96, RBP+92, JCJÖ93, RTF99].*

The UML is a graphical language that expresses program design in a standard way, allowing design tools to interchange models, provided they comply to some specified standard.

Statecharts are a formalism of hierarchical and parallel state machines that communicate and synchronize over several levels. The UML includes statecharts as part of the behavioral view of the modeled system. Large designs can be naturally decomposed along the architectural hierarchy. Due to the emphasis on the dynamics, statecharts are an attractive language for simulation and even code-generation. Since statecharts allow for a multitude of choices for a precise semantics and have a broad spectrum of possible applications, they have been the center of lively discussions, developments, and research over the last years.

In this Chapter we outline the UML in general and the UML statechart formalism in particular. We do so first to give a brief overview on the basic concepts, second to contrasts the attitude of a modeling language *to the characteristics of a* verification formalism*, and finally to motivate the introduction of hierarchical timed automata in Chapter 3.*

1.1 An Outline of UML

The unified modeling language *(UML) is a large collection of modeling formalisms that have proven to be useful in industrial-sized design projects. In this section, we briefly review its history of its development from version 0.8 to 2.0, point to major current developments, and outline technologies that are inherently connected with the evolution of UML.*

1.1.1 From Unified Method 0.8 to UML 2.0

We briefly outline the development of the modeling language over time. See [BRJ99, Kob01b, Kob01a] for an more detailed exposition.

The first object-oriented modeling languages appeared between the mid-1970s and the late 1980, as methodologists started to experiment with new modeling concepts that were tailored for the new object-oriented design paradigm. In 1994 there were around 50 different object-oriented modeling languages around [BRJ99].

The standardization of modeling languages started out, as Grady Booch and James Rumbaugh collaborated within the company Rational and sought to join their development methods, *Booch* and *OMT* with the *Unified Method v. 0.8* in 1995 [BR95]. One year later, Jacobson—the developer of *OOSE*—also joined Rational and contributed his expertise in UML 0.9 [BJR96] (June 1996). The three outstanding methodologists Booch, Jacobson, and Rumbaugh became known as "Three Amigos".

Rational established the UML consortium, consisting of several organizations[1] willing to contribute to a strong and complete modeling language. In January 1997 the straightened out revision (UML 1.0 [Par97a]) was proposed to the *Object Management Group* (OMG)[2] with the intention to make it their standardized modeling language.

This lead to the expansion of the UML consortium that virtually included all other OMG submitters of standardized modeling languages.[3] The final proposal, UML 1.1 [Par97b], was submitted in September 1997 to the OMG. Two month later the OMG officially adopted UML as its object modeling standard.

From there on, the further development was taken care of by an OMG *Revision Task Force* (RTF), a broad and open society that contains both academic and industrial partners. As a purely editorial revision without significant technical changes,

[1] Digital Equipment Corporation, Hewlett-Packard, I-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational, Texas Instruments, and Unisys—just to name a few.

[2] The OMG is an international organization supported by over 800 members, including system vendors, software developers, and users. Details can be found on the OMG home-page http://-www.omg.org.

[3] Notable new partners included Andersen Consulting, Ericsson, ObjecTime Limited, Platinum Technology, PTech, Reich Technologies, Softeam, Sterling Software, and Taskon.

UML 1.2 [RTF98] followed in June 1998.

The RTF reacted to various feedback and collections of problem lists that pointed out—among other shortcomings—the incomplete semantics of activity graphs and suggested a cleanup of standard elements for relationships. As a result, UML 1.3 [RTF99] was released in fall 1998. This builds the basis of most UML textbooks today and is sometimes referred to as the "first mature release of UML" (e.g., [Kob99]).

UML 1.4 [UML01], completed in September 2001, is the last minor revision of the 1.x series, since work on UML 2.0 is nearing completion as this document is written.

The OMG has organized the construction of UML 2.0 in four parallel *requests for proposals* (RFPs), issued between September 2000 and February 2001: *Infrastructure*, *Superstructure*, *Object Constraint Language*, and *Diagram Interchange*.

The Infrastructure RFP aims to revise the structure of UML, in particular to improve the architectural alignment with the meta object facility (MOF) and XML metadata interchange (XMI)—see page 27f—and provide first class extension mechanisms (meta-classes). The Superstructure RFP aims to improve the UML's applicability for software development practices like component based development, or executable models. This should be achieved by supporting run-time architectures and refining the semantic relationship. The Object Constraint Language RFP aims to define a more powerful OCL meta-model, that is consistent with the UML meta-model. The Diagram Interchange RFP aims to define a meta-model for diagram interchange. As opposed to already existing model exchange technologies like CORBA and XMI, this has to include details like element placement, alignment, fonts, color, etc. See [Kob01b] for an overview and the OMG homepage[4] for all the details.

We point out two recommendation of the revision task force: the semantics of the activity graphs should be defined independent from statecharts, concurrency should be rendered more permissive in both diagram types [RTF01]. From UML 1.1, the description of statecharts underwent a number of changes, most of which resulted in new inconsistencies.

1.1.2 Meta-Modeling: The Four Layers of the UML

The UML modeling language is formally organized in four levels, $\mathbf{M}_0$ to $\mathbf{M}_3$ (see Figure 1.1). Elements on level M_n are instances of element on level $\mathbf{M}_{n+1}$. The lowest level is the actual system, the level above is its model. The level above the model, called meta-model, is a description of entities that can be used for modeling. On this level, UML itself and extensions of UML reside. In particular, profiles that extend UML either in a standardized or in a user-defined way are part of $\mathbf{M}_2$. The top level $\mathbf{M}_3$ consists of rules, how this description of entities may be constructed.

[4] http://www.omg.org

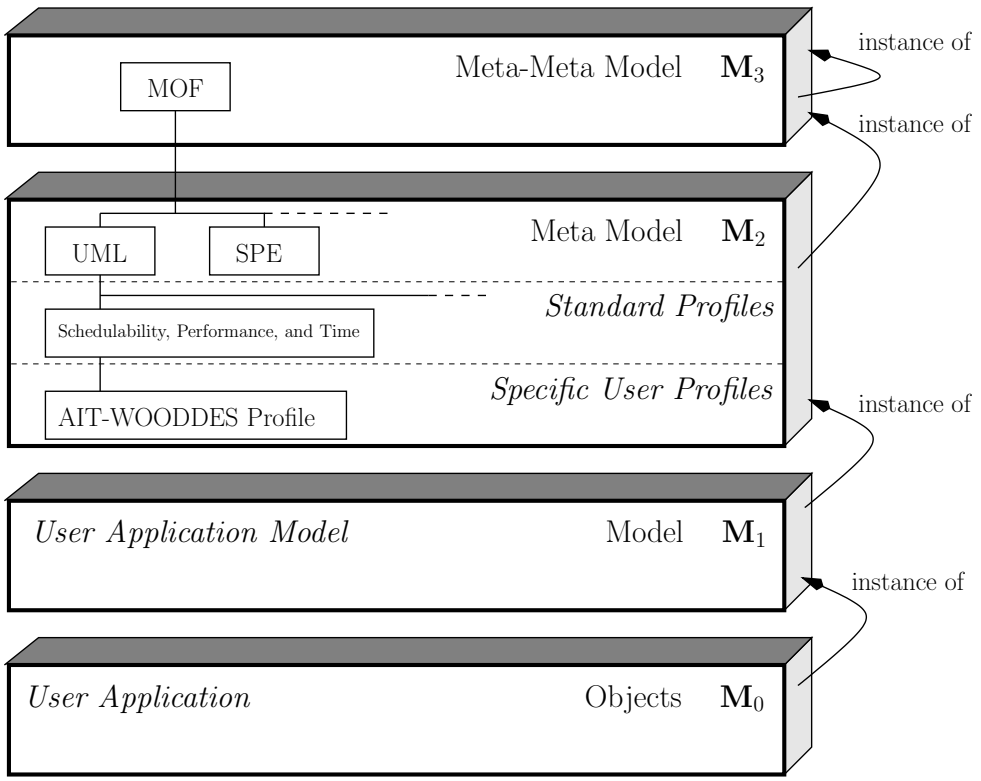


Figure 1.1: The Four-Level UML Structure, Here in the Context of the Standard Profile for Schedulability, Performance, and Time and the AIT-WOODDES Profile.

Example 1.1 *A Java run-time object can exist as a part of the actual system (*$\mathbf{M}_0$*). This object is an instantiation of a more general description—a class—that is formalized in the user application model (*$\mathbf{M}_1$*). The* rules *how to describe a class are given as the UML meta-class (*$\mathbf{M}_2$*) that, e.g., provides a slot for specifying the number of times an object may be instantiated from the derived class. Finally, the language for describing the meta-class, the slot, and what it might contain is the MOF (*$\mathbf{M}_3$*). The meta-class is thus an instantiation of a meta-meta class.*

Some confusion arises, since the description and the described object often look identical. For instance, the UML meta-model is a model of the UML itself - and it is written in UML. The elements of this meta-model are called meta-classes, e.g., *Signal*, *Classifier*, or *Class*.

In a *strict* meta-modeling approach, every element of level M_n is an instance of exactly one element of a level M_{n+1} level element. Up to UML 1.4, the specification follows a *loose* meta-modeling approach, where the "exactly one" requirement is not met.

1.1.3 Extensibility: Lightweight and Heavyweight

Extensions that apply changes on the meta-model level are commonly called *heavyweight extension mechanism*. For example, if you explicitly intend to use a state machine in your model, one way to do so is to introduce meta-model elements "state machine", "state", and"transition" together with proper associations. On the model level, a state-machine is then a class combining these primitives in a proper association (every state machine has one or more states and zero or more transitions; every transition has a start state and a terminal state and so on). An object, finally, is then one specific instance of a state machine model, where the states are named and fixed. The advantage of this approach is that it is possible to tailor modeling elements that exactly correspond to the conventions a group of modelers is already familiar with.

However, introducing new elements on the meta-level comes with new obligations to maintain overall consistency. Moreover, it lets your modeling language diverge from the commonly-used standard. Methodologists warn of what in [Kob99] is called *meta-modeling mania*:

> the tendency to make domain-specific sledge-hammer changes, where claw-hammers would suffice.

No default mechanism for heavyweight model extension existed up to UML 1.4, but it is on the agenda to introduce it in the major revision 2.0 [Kob01b].

Instead of defining a finite state machine from scratch, it could have been expressed as the *restriction* of already existing behavioral elements, like statecharts. An extension that does not apply changes on the meta-model level is also called *lightweight model extension*. A *profile* is the standard lightweight extension mechanism of UML. It comprises a collection of stereotypes, tagged values, and constraints– all expressed in terms of already existing meta-model elements. The intention is to *customize* UML for specific domains of application via expressing key concepts of the domain in terms of already existing model elements.

The advantage of this approach is that syntactic and semantic confusion can potentially be reduced. New definitions and explanations should only be employed, when no appropriate notations and concepts already exist.

Profiles are an integrated concept from UML 1.3 on. Chapter 4 in [UML01] gives sample profiles to demonstrate construction and usage.

1.1.4 Realizing Technologies: OMG and W3C Standards

A number of emerging technologies are closely connected with the evolution of UML. We briefly describe XML, OMA, CORBA, MOF, and XMI as the most relevant ones. The unfamiliar reader of technical texts on the UML is easily confused by the multitude of the employed acronyms. As a reader's aid, we provide a compendium of acronyms on page 227.

The *eXtensible Markup Language* (XML) is a fairly general method to organize textual data in a flexible way. Every XML document is conceptually a rooted tree

with different types of nodes, called *XML elements*, which are textually represented by a pair of opening and closing *tags*. XML itself is defined as an application profile of SGML.[5] It gives a concise way to define a document tree structure via *document type definitions* (dtd).[6]

Today XML (as well as HTML) is standardized by the *World Wide Web Consortium* (W3C). This group develops inter-operable technologies (specifications, guidelines, software, and tools) with the self-declared mission to

> lead the web to its full potential as a forum for information, commerce, communication, and collective understanding.

Details can be found on the W3C homepage.[7]

The *Object Management Architecture* (OMA) embodies the OMG's vision for the component software environment. The architecture provides guidance on how standardization of component interfaces penetrate up—although not into—applications in order to create a plug-and-play component software environment based on object technology.

In connection with OMA, the OMG issued a standard called *Common Object Request Broker Architecture* (CORBA). It is a vendor-independent specification for an architecture and infrastructure that computer applications use to work together over networks.[8]

Inter-operability results from two key parts of the specification: *OMG Interface Definition Language* (OMG IDL), and the standardized protocols *General InterORB Protocol* (GIOP) and *Internet Inter-ORB Protocol* (IIOP). These allow a CORBA-based program to inter-operate with another CORBA-based program, in a way that is largely robust against variations of used operating system, programming language, or network. OMG IDL has been an ISO International Standard for several years. In April 2000, the GIOP and IIOP protocols were almost through the ISO standardization process.

The *Meta-Object Facility* (MOF) is a CORBA Common Facility for the management of meta-information. MOF is a standardized repository for descriptions and definitions of the fundamental concepts that applications work with. It is intended for use in a wide variety of scenarios, from type management to software development, information management and data warehousing - the MOF can be used as a meta-information repository within CORBA distributed systems. MOF is designed general enough, to serve as meta-layer to other constructs than UML, for example for describing the *Software Process Engineering* (SPE) management [SPE99], (as displayed in Figure 1.1).

[5] SGML is the Standard Generalized Markup Language defined by ISO 8879.

[6] The widely used*hypertext markup language* (HTML) can be described as one particular instance of an XML. Document type definitions are found e.g., at http://www.w3.org/TR/REC-html40/-loose.dtd.

[7] http://www.w3.org

[8] One of the best tutorials about CORBA is available online at http://www.omg.org/-gettingstarted/specintro.htm.

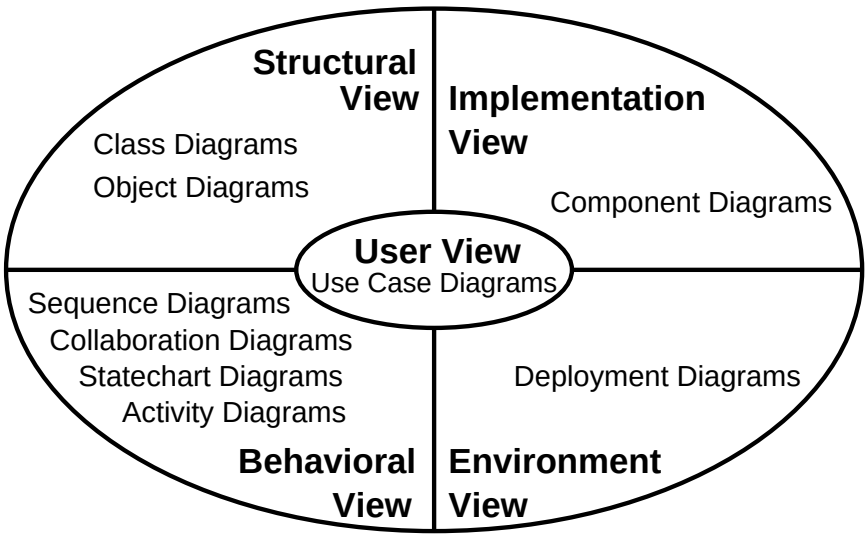


Figure 1.2: UML Offers Five Different Views of a System Under Development. Each View is Substantiated by one or More Diagram Types.

A crucial step in software development is a collaboration of tools that cover different aspects of the development process. Realizing this need, the OMG organized a set of rules on the XML stream format into *XML for Metadata Interchange* (XMI). The main purpose is to enable easy interchange of meta-data between modeling tools, in particular UML and MOF based meta-data repositories. XMI integrates three key industry standards:

1. XML (W3C standard),
2. UML (OMG modeling standard), and
3. MOF (OMG meta-modeling and meta-data repository standard).

1.1.5 Learning UML

It is important to remember that UML is intended to meet the need for *notation* for communicating development teams that start out from various different backgrounds. Therefor preference is given to the *inclusion* of concepts, even at the price of redundancy.

The language is structured into five interlocking *views* (Figure 1.2). Each *view* is a projection of the system under development into its organization and structure.

Since UML does not make assumptions on the area of application, nine kinds of diagrams are present to express specific aspects of the system. They substantiate the five views by providing appropriate descriptive formalisms.

The user view provides a high-level description of the functionality of the system, as it is is perceivable for the user. The structural view defines the connection between different parts of the system and describes their interplay. The behavioral

view describes the dynamics of the system, i.e., possible inputs, internal changes of the system state, and expected outputs. The implementation view describes the system on the level of abstract operational entities (components). The environment view associates the system with its physical representation in hard- and software (mapping).

The description of the diagrams is itself organized in packages and sub-packages. The packages on the top level, are called Foundation, Behavioral Elements, and Model Management. The graphical notation used in the diagrams stems from common industrial usage, ambiguity and redundancy were resolved to a large extent, but some informality remains (October 2001).

Any specific design project is unlikely to make use of the whole language of UML. Much volume of the notation stems from the UML's declared goal, to meet the objectives of different areas of application.

Throughout UML documents, the term "semantics" is used to informally describe intended behavior and rules of usage, as opposed to *formal semantics*, which gives a rigorous and unambiguous definition of all possible behaviors in mathematical terms.

A large part of the description makes use of OMG's *Object Constraint Language* (OCL), which is tailored to express conditions attached to model elements. For example, an OCL expression attached to a class specifies rules that every instantiation of this class has to obey. OCL expressions have a well-defined syntax and instruction in how to read and understand them. We refer the reader to [OCL97] for details. It is, however, also admissible to use natural language to express these conditions, if the designer finds it more appropriate.

For these reasons, UML can be a syntactic, but not a semantic standard. The latter can be approximated, but due to an inherent informality of meaning never be achieved.

1.1.6 Literature on UML

The latest developments and changes in UML are available online from the OMG.[9] These documents of growing volume (the definition of UML 1.4 spans over 548 pages) serve as defining reference of the language and give examples of usage. They are certainly no gentle introduction for learning beginners.

An impressive number of textbooks reached the market to answer the growing need of developers, engineers, and students to learn UML, or rather: to learn about the part of it that is relevant for their work. We briefly review a non-representative collection of them.

The Unified Modeling Language User Guide [BRJ99], authored by the three chief methodologist of Rational, gives a well-organized and detailed description of

[9] http://www.omg.org

all modeling elements of UML. It briefly addresses application issues and outlines the Rational Unified Process as one possible design methodology, where UML can be used. The 1999 edition is on the level of UML 1.3.

UML in a nutshell [Alh98] is published in the popular O'Reilly series and gives a condensed overview of all concepts and diagrams present in UML. Apparently it was put together in a hurry—some descriptions are organized merely as collective lists, technical terms are used inconsistently, and both language and diagrams tend to be ambiguous and confusing. Building on UML 1.1, it is seriously outdated, no upgrades seem to be planned.

Using UML: Software Engineering with Objects and Components [SP00] is unusual in the line of books, for it aims to complement an university course rather than a reference for working engineers. It clearly indicates, which parts are treated in detail and which ones are merely outlined. For the latter ones, references are given. The clean and consequent layout highlights summarizing statements. For issues where experts disagree, discussions are set up and encouraged. The year 2000 is adopted to UML 1.3 and an update to UML 1.4 is already available.

Designing Concurrent, Distributed, and Real-Time Applications with UML [Gom00] focuses on educating engineers for the challenging situation, where the design fits in a problem domain that is characteristically concurrent, distributed, or real-time—or any combination of them.

The book is organized in three parts. The first part gives an overview on UML by describing all diagram types, discusses design concepts, distributed system technology, and provides background in software development methodology. The second part describes the architectural design method COMET in detail. The third part provides five case-studies from different areas.

Appealingly, UML modeling concepts are applied in detailed case studies. The first edition (July 2000) conforms to UML 1.3.

Doing Hard Time: Developing Real-time Systems with UML, Objects, Frameworks, and Patterns [Dou99a] is a domain-specific application of UML to the real-time area, where hard, soft, and firm deadlines are present as the guiding constraints that any implementation has to meet. It addresses mainly professional software developers, focusing on practical development rather than on theoretical introduction. It conceptually distinguishes analysis and design, shaping these phases according to the requirements of real-time development. The ROPES development process is used as a guiding paradigm. The second edition (Sept. 1999) is based on UML 1.3.

Real-Time UML - Developing Efficient Objects for Embedded Systems [Dou99b] is a domain-specific application of the UML. Methodologist Bruce Powel

Douglass focuses on the design of embedded systems that are heavily depended on a *timing component*. Consequently, the emphasis is on the behavioral part. The examples centralize around the RHAPSODY tool [Rha02], which makes use of real-time concepts that are currently in the process of being standardized via UML profiles (like timers and time-out events) [SPT01]. The edition from 1999 refers to the UML 1.3.

1.2 UML Statecharts

Statecharts stem from David Harel's work on visual formalisms for reactive systems. A higraph structure [Har88] is used to represent hierarchical state machines, that can be put in parallel at any level of composition. We summarize this formalism as specified in UML 1.4 [UML01] and relate it with alternative formalizations.

Statechart serve to specify the dynamic behavior of system parts that are capable of receiving and issuing events. Typically this is associated with instances of classes, but statecharts can also be used to describe the behavior of use-cases, actors, subsystems, operations, and other.

We line out the history of state-charts, before we give an exposition of the syntax. Then we discuss the (operational) semantics; we highlight points, where no common agreement exists on several valid choices. Last we discuss briefly tool implementations like STATEMATE, RATIONAL ROSE, or RHAPSODY; the open semantic issues and the crave of costumers for increased functionalities lead to pragmatic extensions of the basic statecharts languages.

1.2.1 The Evolution of Statecharts

The statechart formalism is a description formalism that evolved from well-established diagrams—like flow charts and (communicating) state machines—and offers a remedy for their limitations in the representation of complex reactive systems [HP85, Pnu86].

The original paper [Har84] appeared delayed in [Har87]: various committees considered the work relevant, but not appropriate for their community. David Harel contributed to the implementation of STATEMATE1 in 1986; one of the outcomes of which is a formal operational semantics. Unfortunately, the corresponding publication [HPSS87] is incomplete, in that it only describes *one* approach and misses the full symmetry in the relationship of orthogonal components.

Despite a general agreement on the basic behavior, the hierarchical nature, advanced parallelism, and sophisticated event communication yield a wide number of choices on the details. Various groups contributed to the exploration of variations; the survey article [vdB94] lists 20 different semantics for statecharts and this compendium is far from being exhaustive. The article gives 19 criteria—introduced as "Problem list"—to distinguish the variations. Yet another (very restricted) variant

is introduced in Chapter 3 of this thesis.

The first executable semantics of the STATEMATE tool was supplemented in [HN96]. The paper contains a rigorous, but informal description of the STATEMATE semantics, as it was used by the development team in I-Logix. The choices are classified according to the criteria in [vdB94]. Semantics of statecharts was—and still is—subject of further discussion (e.g. [PU97, MLPS97]).

Today there is a rich volume of literature on statecharts. Because of the apparent practical impact, statecharts are considered an interesting subject in different research disciplines. The flexibility of the formalism offers various challenges. E.g., statecharts can be understood as a structural decomposition that allows the compositional analysis of complex systems; this motivated the formulation of *compositional* axiomatizations, first published in [HRdR92].

Particularly interesting are the timed extensions that have a natural motivation in reactive systems. Already [Har87] suggests to make duration a primitive in the formalism.[10] A structured operational semantics for timed statecharts was first given in [KP92], where a textual representation is preferred over a graphical one. Here, a system is associated with sets of timed traces. Time-bounds attached to transitions serve as to *restrict* the behavior of the system. Time constrains prune the number of possible traces, thus the timed satisfies more properties than the un-timed version.

[HN96] contains the first rigorous (but informal) description of the STATEMATE implementation of statecharts. This is elaborated in [PU97] to a formal semantics in terms of *clocked transition systems*, thus making it possible to benefit from the analysis tools developed for this formalism [KMP96, MP91].

UML adopted statecharts as one of the central behavioral diagrams. This seems natural, since statecharts correspond well with an object-oriented approach (encapsulation) and subsume other useful formalisms, like finite state machines. The development of the UML is governed by large committees, which makes it a subtle issue to make clear choices on a semantic level. As yet, the rigorous and unambiguous description of UML statecharts is still under development.

1.2.2 The Basics of UML Statecharts

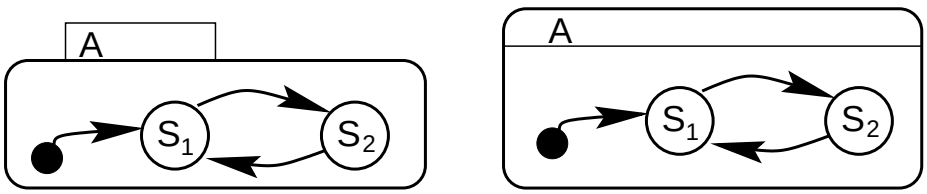
UML statecharts are state machines, where the control locations—usually called *states*—can potentially expand to arbitrarily complex state machines. Thus states are either atomic (*basic states*) or state machines themselves (*superstates*). Superstates can be put in parallel to build a new superstates. Parallel machines communicate via broadcast style operations. [HPSS87] summarizes this as

$$\textbf{statecharts} = \textbf{state-diagrams} + \textbf{depth} + \textbf{orthogonality} + \textbf{broadcast}$$

Basic states are denoted with a round circle, optionally with a name of the state inside: Ⓛ. The initial state is indicated by a smaller bullet ●, transitions are arrows.

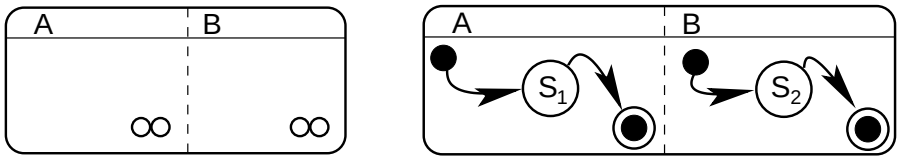
[10] The running example in the paper is a wrist watch. The suggested notation to indicate limited duration of a superstate—a wiggle in the boundary of a state—did not catch on.

Superstates are denoted by rounded boxes, where a name (here: A) is either added in an adjacent square box, or in a field separated by a horizontal line:

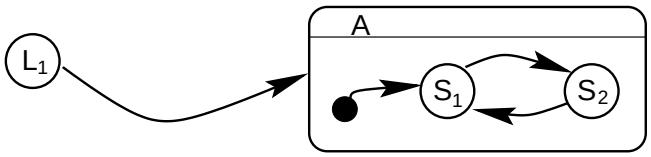


Sometimes it is necessary to indicate that a superstate has terminated and will not perform transitions any more (unless left and re-entered). For this case, a special state called *final state* (also known as *terminal states*) exists, denoted by a bullseye: ◉.

The contents of the rounded box can directly display the associated state machine. However, the definition is often deferred to a separate diagram; in this case, the rounded box contains conventionally the symbol ∞ to indicate the incompleteness. This makes sense, when the superstate is part of a *parallel composition* of superstates, indicated by a dashed separation line (orthogonality). Both of the next pictures describe a (unnamed) superstate that consist of two parallel *sub-machines* A and B:



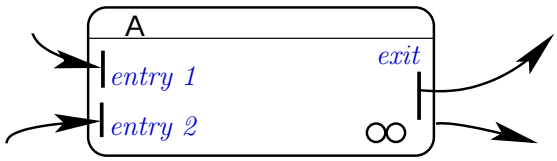
When a transition connects to the border of a superstate, then control starts at the bullet. Strictly speaking, the bullet is not a proper state[11], it has to be left immediately. In the example below, control moves from Location L_1 to S_1 , passing through the bullet, but without staying there:



If the superstate itself is a parallel composition, then all parallel sub-machines start at the bullets. Alternatively, a transition can split up and explicitly point to the target states. This is called a *fork*, whereas the dual construct is a *join*:

The bars in the fork or join can also be used as markers for special entries or exits. This is particularly useful, when the details of a state-machine are omitted in a diagram, but different types of entries and/or exits have to be distinguished. The bars are then reduced to *stubs* as in the following diagram:

[11] In UML nomenclature, it is a *pseudo state*; this means that is merely a notational construct and not part of a proper configuration of the system.

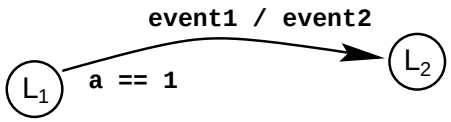


Note that *entry 1*, *entry 2*, and *exit* are the names of the corresponding stubs. The transition starting at the border of the rounded box implies the existence of a *default exit*: the superstate A can be exited at any point in time, provided the transition can be taken.

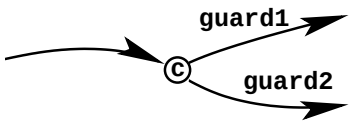
A modeling element called *history connector*, graphically Ⓗ, is a special entry that is useful in presence of default exits. If a transition connects to it, control moves to the last active state in this superstate.[12]

Transitions can be equipped with *guards*, *trigger events*, and *actions*. A guard is a boolean expression that has to evaluate to **true**, in order for the transition to be taken. If no guard is drawn, this corresponds to the constant **true**. The (optional) trigger event has also to be present, in order for the transition to be taken. The (optional) action can be an *assignment* of variables and/or an event that is broadcast if the transition is taken.

In the following diagram, the transition from L1 to S1 carries the guard `a == 1` (requiring that variable `a` has value 1), a trigger event with name `event1`, and—as action—the release of the event with name `event2`:



A transition is not always atomic: it may require other transitions to be taken simultaneously or subsequently. Either might be the case, when an event is broadcast. Subsequent transitions arise, if a transition ends not at a proper state, but at a *choice point*:



The transition cannot stop at Ⓒ, but has to continue by taking one of the subsequent transitions. The intuition behind this construct is rather a cases split than non-determinism, thus `guard1` and `guard2` are often mutually exclusive.

Since taking transitions both consumes and generates events, this gives rise to chain-reaction like effects. A sequence of transitions that terminates in a valid configuration forms a *run-to-completion step*. Since transitions generate events and modify variables that the precise definition of legal run-to-completion steps has been subject to lively discussions.

[12] Strictly speaking, one can distinguish between *deep* and *shallow history*, according to whether the control state of enclosed superstates is also restored (deep) or not (shallow). Deep history can be encoded explicitly via shallow history, if every enclosed superstate has a history connector.

There is a number of details and additional constructions and we are not in a position to treat them exhaustively. Note that UML statecharts are not an isolated formalism, but embedded in a rich description language and the context of other modeling elements. For instance, the guards are in general object constraint language constructs (see page 30) and thus can refer to modeling elements outside the statechart formalism.

Industrial usage motivated the introduction of complex behavioral constructs, like hierarchy of events, actions triggered on entry or exit of components, and activities associated with states. The interested reader is deferred to the expositions in [BRJ99] (Chapter 24), [Dou99a] (Chapter 4), and—of course—the exhaustive definition in [UML01].

1.2.3 Semantics: Still under Development

Up to now, no rigorous description of the semantics of UML statecharts exists. Criticism of the UML addressing the lack of a formal semantics[13] is often targeted at the informal—and occasionally inconsistent—description of statecharts.

Most items on the problem list of [vdB94] constitute the open points today. The discussions centralizes around two issues: events and atomic steps.

Events

Events are the mechanism for synchronization between parallel components. Events are generated either by the environment or by taking a transition. A transition may require the presence of an event, or set of events, in order to be enabled.

If also the absence of an event can be required, this raises the problem of *global consistency*: can an event `e` be generated, if the transition generating `e` requires the absence of `e` in order to be taken? A related issue is *causality* (or *self-triggering*): a transition labeled with `e / e` cannot be taken, unless `e` is generated by some other means than the execution of the transition itself.

In a sequence of steps, events can be understood either synchronous (dealt with in the same step) or asynchronous (deferred until the next step). Both possibilities—and even a mixture of them—make sense under certain circumstances.

In addition to explicitly generated events, *implicit events*—like the entering/exiting of a superstate, or the passage of time—may be useful in specifications.

Finally, instead of taking events as atomic entities, hierarchical (lattice) structures of events may be allowed, where one event can be a generalization of two incomparable ones. Depending on the general or specific occurrence, different behavior can be triggered.

[13] Up to UML1.4, the semantics of diagrams is given merely textually. UML 2.0 is supposed to have an "action semantics", i.e., an unambiguous operational description of all behavioral parts.

Chain Reaction (Run-To-Completion Step)

In the synchronous case, the generation of events entails that one transition cannot be taken in isolation. Rather, a complete *chain reaction* of transitions is required, before new inputs from the environment are accepted. For every transition in this chain reaction the trigger event (if any) has to be present and the guard has to be satisfied in the instance the transition is taken. This requirement is called *local consistency*.

The definition of legal chain reactions is complicated by the presence of variables that might be read and written in the same sequence—a phenomenon sometimes referred to as *race condition*. Since transitions may react to events in parallel, this is a subtle issue.

It is reasonable to require the sequence to be globally consistent and terminate in a legal configuration. This entails decidability questions in the general case: recall that the UML strives to be maximally liberal with respect to what syntactic constructs are allowed.

All these questions have to be answered, to associate an UML statechart with a set of possible executions. For an elaborate discussion on what is in a step see [PS01].

1.2.4 CASE Tool Implementations of Statecharts

Statecharts are implemented as a behavioral description language in various *CASE tools*, e.g., STATEMATE, RATIONAL ROSE, and RHAPSODY. The prime purpose of these tools is to aid industrial design processes as far as possible.

Since statecharts already exhibit a—typically deterministic—execution behavior, it seems natural to *connect* the high-level description to real application code segments that might not require conceptual visualization. The drive from this application lead to extensions in the functionality that are motivated by the demand of customers rather than by reflected considerations.

Bridging the gap from symbolic execution (or: simulation) to executable code is conceptually simple. Superstates can be associated with object classes, where the activation of a superstate corresponds to an instantiation of the corresponding class. User interaction and physical measurements boil down to the release of events and calls to volatile methods and are realized via a special object: the *environment*. A scheduler that governs the release and arrival of events and resolves possible nondeterminism, completes the picture.

Now it is straightforward to add an *target language* that gives syntax (and semantics) of guard expressions and assignments.[14] Such an equipped statechart then compiles down to a set of source-code files in the particular execution language, which can further be compiled to machine executable code. This process is commonly referred to as *code generation*.

[14] E.g., In RHAPSODY, this is realized by a **C**, **C++**, or **Java** method call that is allowed to have side effects.

To maintain the connection to the statechart visualization in the tool, a number of hooks can be carried along during code generation that preserve the association to the original model. The running executable then connects with the CASE tool, This process is sometimes referred to as *animation*. When this animation also accepts user-interaction, e.g., the generation of environmental events, this allows for a high-level form of debugging.

How Real is Real-Time Behavior?

At first it seems straightforward to represent real-time behavior by simply adding *timers*, which correspond very much to hardware elements.[15]

However, the timed semantics make use of the *synchrony hypothesis*—first formulated in [BG92]—, by which the system is infinitely faster than its environment and can always compute its response before the next stimulus arrives.

In an implementation, this clearly does not hold. External events have to be buffered, should they not interfere with the execution of a step in progress. It has to be asked, how serious the deviation of the run-time behavior with respect to the formal semantics is, since crucial parts of the analysis are based on the latter.

Central for the formal description of code-generation is a strong connection of statecharts with a target language. E.g, in RHAPSODY, superstates can be associated with **C++** classes; entering a superstate corresponds to creation of an object of this class. The interaction of objects is by events (of which exactly one can be present at a time[16]) or direct method calls.

The central concept here is the *active object* (sometimes also *executable objects*), which is an entity owning a process of thread and able to initiate control activity. This allows to capture for complex timed behavior on an abstract level. A brief informal description can be found in [HG94, HG97], more details are elaborated in [BRJ99]. As yet, the notion of active objects has not solidified to a robust and widely acknowledged definition.

Importantly, CASE tools *do* give a semantics of the artifacts they deal with via their (deterministic) implementation—but this might not be formalized explicitly. For statecharts this means that the tools make choices wherever the UML lacks prescription. Thus no two tools declared to be "UML tools" need to agree on the behavior of the constructs. E.g., the successor of STATEMATE, RHAPSODY, disagrees in several behavioral aspects, for instance

- simultaneous presence of events (severals vs. maximally one)
- implicit priority of transitions (outermost first vs. innermost first)

It is fair to state that the maturity of the CASE tools that feature statecharts as a behavioral primitive, significantly increased during the last five years, see e.g. [Hil99].

[15] E.g., in RHAPSODY, timers are implicitly attached to states that are sources of time-out transitions. Passage of time can then generate a *time-out event*, which triggers the exit of the state.

[16] In the more functionally oriented predecessor STATEMATE, negation of events is allowed. RHAPSODY makes a simplification here, since the absence of event is rarely a well-motivated condition and typically only used to encode priorities.

However, the level of reliability and stability is not such that they can be claimed to be finished nor "compatible" in any reasonable sense.

1.3 Reflection: UML and Statecharts

The development of the UML dwells on a rich set of experiences and proven technologies. The language provides a complete family of diagrams, each shaped to address a different aspect of the system and each turned out to be relevant in various user experiences. Statecharts are perhaps the most important diagram type, because they address the behavior—i.e., a computation—on a high level of abstraction. For any reasonable degree in size and interaction, construction and analysis of this part is an extremely challenging task.

The maturity of the UML is arguable. Without doubt, the UML evolved very stringent way and the revision mechanisms of the OMG are effective. Since the volume of material is enormous, today no one person is up to date in *all* the details. The current revision to UML 2.0 that is supposed to clarify many semantic issues, is certainly an important step.

One of the declared goals of the UML is the standardization of flexible and useful description elements, both in notation and usage. Standardization of a modeling language comes at a price. It necessarily has to include *all* the needs for a wide range of users. A standard modeling language has to provide extension mechanisms: otherwise, it is sentenced to a priori limited application domain, will be extended ad hoc and encourages non-standard usage.

Though the scope of the language is broad and powerful, it is better suited for some domains than for others. In particular, the object oriented paradigm is central to various diagram types.

It is perceivable that tools have yet to catch up with the fast iterations of revisions. because—if you have customers from industry—you can never take back a feature. The things to come are promising and they have high promises to keep.

The purpose of the UML is to aid the detailed description of complex systems throughout design. Therefore the language is rich, user-centered, and maximally expressive. Quite the contrary is true for formalisms that strive to be minimal, algorithm-centered, and aiming to restrict to decidable cases. An important example, the timed automata model of Uppaal, is given in the following Chapter.

Chapter 2

The Timed Automata Model of UPPAAL

First, a few words about tools. Basically, a tool is an object that enables you to take advantage of the laws of physics and mechanics in such a way that you can seriously injure yourself. Today, people tend to take tools for granted. If you're ever walking down the street and you notice some people who look particularly smug, the odds are that they are taking tools for granted. If I were you, I'd walk right up and smack them in the face.

— Dave Barry, "The Taming of the Screw"

UPPAAL *[LPY97] is a tool box for modeling, verification, and simulation of real-time systems. It has been developed jointly by Uppsala University and Aalborg University throughout the last seven years. It is appropriate for systems that can be described as collection of non-deterministic parallel processes.*

The modeling language used in UPPAAL *is an enriched dialect of the well studied timed automaton formalism [AD94], i.e., it features real-valued clocks over a finite control structure. Additionally the language allows for networks of timed automata that communicate through channels and shared variables. The usability and scalability of this formalism has been demonstrated by successfully application in various case studies, e.g., [LPY98, LP97, HSLL97].*

In this Chapter we formally introduce the modeling language of UPPAAL *and equip it with a trace-based (formal) semantics. We use this semantics to describe the specification language of the tool that allows for (timed) safety, reachability, inevitability, potentially always, and unbounded response.*

2.1 Timed Automata in UPPAAL

First, we give an informal description of the timed automata model as used in UPPAAL*, i.e., networks of timed automata with handshake synchronization and discrete data. Second, we elaborate this and give the formal syntax for a* UPPAAL *model.*

2.1.1 Informal Description

An UPPAAL model consists of a network of timed automata with clocks, invariants, variables over basic data types, guards, handshake synchronization, urgency, and committed locations.

The basic unit is one process that consists of a directed control graph with labels on locations and transitions. One location is marked as initial, indicated by the notation ◎.

Data components. The data part of the model consists of discrete integer variables and (formal) clocks that can take any non-negative real value. In UPPAAL, integers are constrained to have values in the interval `[-32767; 32767]`. Exceeding the limits wraps around to this finite domain. Variables and clocks can be local to one process or global. If they are local, standard scoping rules apply and they cannot be accessed by other processes.

We note that for integer variables, UPPAAL allows for some useful constructs. It is possible to declare integers with limited range, construct arrays of fixed width, and deal with integer expressions containing constants and the operators `+`, `-`, `*`, and `/`. For simplicity, we treat variables here always as integers and do not describe the full range of valid integer expressions. For the details we refer to [LPY97] and the online help.

Control structure. Every location can be equipped with an *invariant*. This is constrained to be a conjunction of expressions $\texttt{x} \leq \texttt{const}$ and $\texttt{x} < \texttt{const}$, where `x` is a clock and `const` is an integer constant.

Locations can be equipped with one of the attributes *urgent* or *committed*. If a location is urgent, no time delay is possible before this location is left. A committed location also has to be left immediately, but leaving this location has precedence over other possible transitions. We use the graphical notations ⓤ and ⓒ for urgent or committed locations respectively.

Transitions are directed arcs between locations called the *source* and the *target*. Transitions can carry guards, assignments, and synchronization signals. We assume that guards and assignments are always given, in case of absence they are considered constant **`true`** or empty respectively.

Attributes for transitions. For a location l, all transitions with source l are called *outgoing transitions* of l.

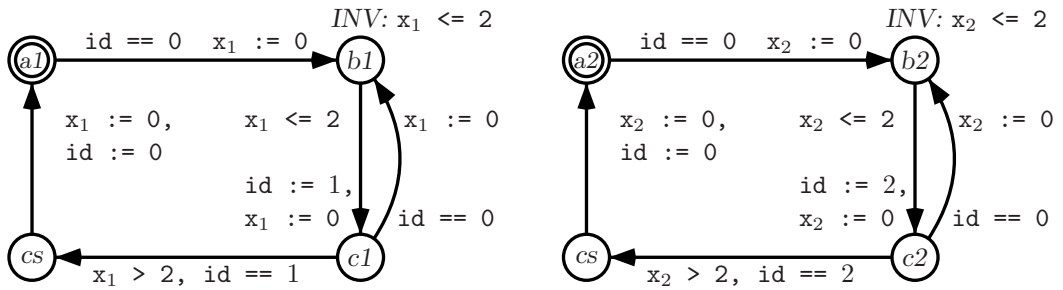


Figure 2.1: Fischer's Protocol for Mutual Exclusion (2 Processes).

A *guard* is a conjunction of boolean expressions over variables and clock constraints of the form `x`$\sim$`const` or `x-y`$\sim$`const`, where `x,y` are clocks, $\sim \in \{<, \leq, >, \geq\}$, and `const` is an integer constant.

Outgoing transitions without synchronization signals are *enabled*, if their guard evaluates to **true** and the invariant of the target location holds after execution of the assignment.

An outgoing transition t_1 with synchronization signal `b!` is enabled, if there exists an outgoing transition t_2 in a parallel process with matching synchronization signal `b?`, and for both t_1 and t_2 the guards evaluate to **true** and the location invariants of the target locations hold after executing the corresponding assignments.

An *assignment* is a sequence of expressions that are either clock resets or of the form `v := expr`, where `v` is an integer variable or element of an array of integers, and `expr` is an arithmetic expression over integers.

Clock resets are of the form `x := 0`, where `x` is a clock.

Example 2.1 (Fischer's Mutex)
Figure 2.1 shows of Fischer's mutual exclusion protocol (see Section 5.4.3) for two UPPAAL *processes. The processes share the integer variable* `id` *(initially set to 0). Each process owns a clock* x_i*, i.e., has exclusive read and reset operations on it. This clock is used to time the progress to the critical section (*cs*). The mutual exclusion property requires that always at most one process in the critical section.*

The processes, call them `P1` *and* `P2`*, start at* a1 *and* a2 *with* `id == 0` *and clocks set to 0. Further progress in action and time delay is non-deterministic, as long as it obeys the restrictions of guards and invariants of the model. For example, an arbitrary amount of time can elapse (delay step) before any of the two processes takes a transition (action step). As a possible first action step, the first process can pass the guard* `id == 0`*, reset its clock* x_i *to 0, and move control to the location* a2*. The invariant INV:* x_i `<= 2` *requires that* a2 *is left again before clock* x_i *exceeds 2, i.e., within 2 time units. The only option to do so is taking the transition to* c1 *that writes the process number (1) to the shared variable* `id` *and resets the clock* x_1*. Now in order to progress to the critical section* cs*, time has to elapse for* more *than 2 time units (guard* x_i `> 2`*). The guard* `id == 1` *makes sure that no other process* i *has taken the transition* b_i *to* c_i *in the meantime. As it turns out, this suffices to establish mutual exclusion.*

Behavior. A *configuration* is a snapshot of the system with one designated control location for every process and values for all variables and clocks. An execution of the model starts in the implicit initial configuration, where every process is in its initial location, all clocks are 0 and all variables (global as local) are set to their initial value (integers are 0, arrays are filled with 0).

A configuration evolves in *action steps* and *delay steps*. Action steps are either isolated of synchronized. A simple action step amounts to taking one enabled transition of one process, execute assignments and clock resets and move control for this process to the new location. A synchronized action step means that two processes with enabled transitions, that carry matching synchronization signals (e.g, `b!` and `b?`) both take these transitions. Both associated assignments and clock resets are executed—the one corresponding to the `!`-transitions first—and control is updated for both processes.

If one of the processes is in a committed location, then all action steps not starting in committed location are blocked. In case of a synchronized action step, at least one of the two participating processes is required to be in a committed location, otherwise the step is blocked.

A delay step increases the value of all clocks by a real value $d > 0$. Delay is only enabled, if several conditions hold true.

1. No process is in an urgent location,
2. No process is in a committed location,
3. No synchronized action on an urgent channel is enabled, and
4. No location invariants are violated after the delay d.

We note that the real-valued nature of the delay steps is not directly observable, since clocks are always compared to integer values (in guards, invariants, and formulas). The possibility of real-valued delays basically allows for any order of the fractional part of clocks, which is not possible if the granularity of time is fixed in advance [Alu91].

A *trace* is a sequence of configurations, starting with the initial configuration. For every two consecutive configurations c_i and c_{i+1} in a trace, there has to exist an action or delay step that transforms c_i into c_{i+1}. For safety properties, it suffices it suffices to consider only finite traces, since every safety property can be violated (if at all) after a finite number of steps. For liveness, we have to consider both infinite and maximally extended finite (deadlocked) traces, since liveness properties can fail in the later case.

2.1.2 Formal Syntax

We define the formal syntax of UPPAAL models as a parallel composition of processes.

For simplicity, we assume a set of labels *Labels* that ranges over syntactically correct invariants, assignments, guards and synchronization labels. As a well-formedness

condition, labels are constrained to occur only in appropriate places, contain only declared variables, and have to respect the variable types.

Definition 2.2 (UPPAAL Process)
An UPPAAL process *A is a tuple $\langle L, T, \mathit{Type}, l^0\rangle$, where*

- *L is a set of locations,*
- *T is a set of transitions $l \xrightarrow{g,s,a} l'$, where $l, l' \in L$, g is a guard, s is a synchronization label (optional), and a is an assignment (possibly empty),*
- *$\mathit{Type} : L \to \{o, u, c\}$ is a type function for locations, and*
- *$l^0 \in L$ is the initial location.*

We use the following access functions to refer to invariants, guards, synchronizations, and assignments.

- *$\mathit{Inv} : L \to \mathit{Labels}$ maps to the invariant of a location (possibly constant* **true***),*
- *$\mathit{Guard} : T \to \mathit{Labels}$ maps to the guard of a transition (possibly constant* **true***),*
- *$\mathit{Sync} : T \to \mathit{Labels} \cup \{\varnothing\}$ maps to the synchronization label of a transition (if any), and*
- *$\mathit{Assign} : T \to \mathit{Labels} \cup \{\varnothing\}$ maps to the assignment associated with a transition (possibly the empty assignment).*

Definition 2.3 (UPPAAL Model)
An UPPAAL model *is a tuple $\langle \vec{A}, \mathit{Vars}, \mathit{Clocks}, \mathit{Channels}, \mathit{Type}\rangle$, where*

- *$\vec{A}$ is a vector of processes $A_1, \ldots, A_n$;*
  We use the index i to refer to A_i-specific parts L_i, T_i, Type_i, and l_i^0,
- *Vars is a set of variables, i.e., (bounded) integers and arrays,*
- *Clocks is a set of clocks, $\mathit{Clocks} \cap \mathit{Vars} = \varnothing$,*
- *$\mathit{Channels}$ is a set of synchronization channels, $\mathit{Channels} \cap \mathit{Vars} = \varnothing$, and $\mathit{Channels} \cap \mathit{Clocks} = \varnothing$,*
- *Type is a polymorphic type function extending the Type_i, i.e., Type maps*
  - *locations to $\{o, u, c\}$ (according to the functions Type_i),*
  - *channels to $\{o, u\}$, and*
  - *variables to $\{\mathsf{int}, \mathsf{array}\}$.*

  We use o, u, c, int, and array as predicates, i.e., for a channel b the expression $u(b)$ evaluates to **true***, if and only if $\mathit{Type}(b) = u$.*

Definition 2.4 (Configuration)
A configuration *of an* UPPAAL *model $\langle \vec{A}, \mathit{Vars}, \mathit{Clocks}, \mathit{Channels}, \mathit{Type}\rangle$ is a triple $(\vec{l}, \mathrm{e}, \nu)$, where $\vec{l}$ is a vector of locations, e is the environment for discrete variables, and ν is the clock evaluation, i.e.:*

- *$\vec{l} = (l_1, \ldots, l_n)$, where $l_i \in L_i$ is a location of process A_i,*
- *$e : \mathit{Vars} \to (\mathbb{Z})^*$ maps every variable v to either a value (if $\mathsf{int}(v)$) or a tuple of values (in case of $\mathsf{array}(v)$), and*

- $\nu : Clocks \rightarrow \mathbb{R}_{\geq 0}$ *maps every clock to a non-negative real number. For* $d > 0$, *the notation* $(\nu + d) : Clocks \rightarrow \mathbb{R}_{\geq 0}$ *describes the function "*ν *shifted by* d*" in the following sense:*
  $\forall x \in Clocks.\ (\nu + d)(x) = \nu(x) + d.$

Sometimes it is necessary to refer to certain parts of a configuration. We call $\vec{l}$ the *control situation* the pair $(\vec{l}, e)$ the *discrete part*, and ν the *continuous part* of a configuration.

2.2 Trace Semantics of the UPPAAL Model

UPPAAL models evolve according to legal steps that are either delays or actions. The compendium of all legal steps defines the behavior of the model.

We start by formulating simple actions, synchronized action, and delay steps. To modify the control situation $\vec{l}$, we use the notation $\vec{l}[l'_i/l_i]$ to indicate that at position i, l_i was replaced by l'_i, and the other positions did not change. We readily use assignments a as transformers on the function e (and ν) and write $a(e)$ (and $a(\nu)$) for the resulting evaluations. Furthermore we use the notation $e, \nu \models_{loc} \varphi$ to indicate that a boolean expression φ holds true under the evaluations e, ν for the contained variables and clocks, and $(\vec{l}, e, \nu) \models_{loc} \varphi$ analogously in the case that φ contains expressions of the form $A_i.l_i$ (denoting that process A_i is in location l_i). We defer a formal definition of $\models_{loc}$ to Section 2.3.1.

Definition 2.5 (Simple Action Step) *For a configuration* $(\vec{l}, e, \nu)$, *a* simple action step *is enabled, if there is a transition* $l_i \xrightarrow{g,a} l'_i \in T_i$, l_i *in* $\vec{l}$, *such that*

1. $e, \nu \models_{loc} g$,
2. $a(e), a(\nu) \models_{loc} Inv(l'_i)$, *and*
3. *if* $\exists l_c$ *in* $\vec{l}$ *with* $c(l_c)$, *then* $c(l_i)$.

We abbreviate this with $(\vec{l}, e, \nu) \xRightarrow{a} (\vec{l}[l'_i/l_i], a(e), a(\nu))$

Definition 2.6 (Synchronized Action Step) *For a configuration* $(\vec{l}, e, \nu)$, *a* synchronized action step *is enabled if and only if for a channel* b *there exist two transitions* $l_i \xrightarrow{g_i, b!, a_i} l'_i \in T$ *and* $l_j \xrightarrow{g_j, b?, a_j} l'_j \in T$, l_i, l_j *in* $\vec{l}$, $i \neq j$, *such that*

1. $e, \nu \models_{loc} g_i \wedge g_j$,
2. $a_j(a_i(e)), a_j(a_i(\nu)) \models_{loc} Inv(l'_i) \wedge Inv(l'_j)$, *and*
3. *if* $\exists l_c$ *in* $\vec{l}$ *with* $c(l_c)$, *then* $c(l_i) \vee c(l_j)$.

We abbreviate this with $(\vec{l}, e, \nu) \xRightarrow{\tau} (\vec{l}[l'_i/l_i][l'_j/l_j], a_j(a_i(e)), a_j(a_i(\nu))$

Definition 2.7 (Delay Step) *For a configuration* $(\vec{l}, e, \nu)$*, a* delay step *with delay* d *is enabled, if and only if all of the following holds.*

1. $\forall l_i$ *in* $\vec{l}$. $\neg \mathrm{u}(l_i)$,
2. $\forall l_i$ *in* $\vec{l}$. $\neg c(l_i)$,
3. $\neg\exists l_i \xrightarrow{g_i,b!,a_i} l_i' \in T_i$, $l_j \xrightarrow{g_j,b?,a_j} l_j' \in T_j$, *with* l_i, l_j *in* $\vec{l}$, $i \neq j$, *such that* $u(b)$, $e, \nu \models_{loc} g_i$, $e, \nu \models_{loc} g_j$, $a_j(a_i(e)) \models_{loc} \mathrm{Inv}(l_i') \wedge \mathrm{Inv}(l_j')$, *and*
4. $e, (\nu + d) \models_{loc} \bigwedge_i \mathrm{Inv}(l_i)$.

We denote this by $(\vec{l}, e, \nu) \stackrel{d}{\Longrightarrow} (\vec{l}, e, (\nu + d))$.

Definition 2.8 (Well-Formed Sequence/Timed Trace)
Let $M = \langle \vec{A}, \mathit{Vars}, \mathit{Clocks}, \mathit{Channels}, \mathit{Type} \rangle$ *be a* UPPAAL *model. A sequence of configurations* $\{(\vec{l}, e, \nu)\}^K = (\vec{l}, e, \nu)^0, (\vec{l}, e, \nu)^1, \ldots$ *of length* $K \in \mathbb{N} \cup \{\infty\}$ *is called a* well-formed sequence for M*, if*

(i) $(\vec{l}, e, \nu)^0 = \big((l_1^0, \ldots, l_n^0), [\mathit{Vars} \mapsto (0)^*], [\mathit{Clocks} \mapsto 0]\big)$,

(ii) (maximally extended finite sequences)
If $K < \infty$*, then for* $(\vec{l}, e, \nu)^K$ *no further step is enabled,*

(iii) (non-zeno)
If $K = \infty$ *and* $\{(\vec{l}, e, \nu)\}^K$ *contains only finitely many* k *such that* $(\vec{l}^{\,k}, e^k) \neq (\vec{l}^{\,k+1}, e^{k+1})$*, then eventually every clock value exceeds every bound* $(\forall x \in \mathit{Clocks} \forall c \in \mathbb{N}\, \exists k.\ \nu^k(x) > c)$.

A well-formed sequence for M *is called a* timed trace for M*, if in addition the following holds.*

(iv) For every $k < K$*, the two subsequent configurations* k *and* $k+1$ *are connected via a simple action step, a synchronized action step, or a delay step, i.e.,*

$$(\vec{l}, e, \nu)^k \stackrel{a}{\Longrightarrow} (\vec{l}, e, \nu)^{k+1} \quad \text{or}$$
$$(\vec{l}, e, \nu)^k \stackrel{\tau}{\Longrightarrow} (\vec{l}, e, \nu)^{k+1} \quad \text{or}$$
$$(\vec{l}, e, \nu)^k \stackrel{d}{\Longrightarrow} (\vec{l}, e, \nu)^{k+1}.$$

Condition *(iii)* weeds out those traces, where time converges towards a finite value in an infinite number of steps. These traces are also called *zeno traces* and correspond to a degenerated behavior of the model, i.e., they have no counterpart in the physical world where time always progresses.

We note that according to this definition, an infinite trace may yield an infinite loop of (synchronized) action steps. This also prevents time from progressing, but is rather a failure of the model than a flaw of the modeling language. These degenerated traces are kept in semantics to make it possible to detect failures of this type.

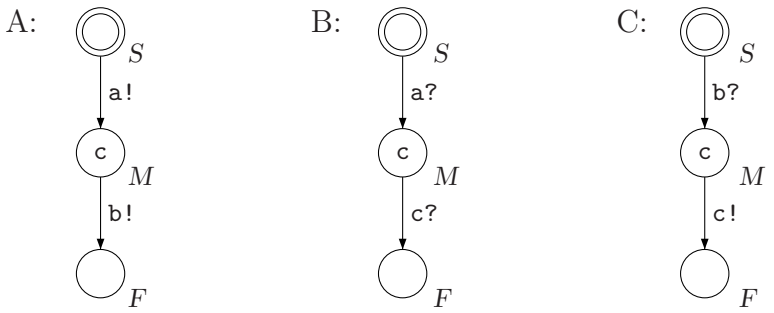


Figure 2.2: The Control Situation `A.F and B.F and C.F` Can be Reached Via the Trace `(A.S B.S C.S)` $\stackrel{\tau}{\Longrightarrow}$ `(A.M B.M C.S)` $\stackrel{\tau}{\Longrightarrow}$ `(A.F B.M C.M)` $\stackrel{\tau}{\Longrightarrow}$ `(A.F B.F C.F)`.

Example 2.9 (Zeno Traces) *Consider a* UPPAAL *model consisting of one* UPPAAL *process A and one clock x. A has only one (initial) location l with the invariant $x \leq 2$. Now one can construct a sequence of delay steps with delay values 1, 1/2, 1/4, 1/8, etc. This sequence can be infinite without ever reaching a configuration with $\nu(x) = 2$.*

According to Definition 2.8 (iii), this sequence is not a valid trace. For this UPPAAL *model every trace is finite and ends, due to (ii), in the configuration where A is at l and $\nu(x) = 2$. There are uncountably many such traces.*

2.2.1 Collection of Legal Traces

We now associate an UPPAAL *model M with an (typically uncountable) set $\mathcal{T}(M)$ of timed traces that are either infinite or maximally extended (deadlocked).*

Definition 2.10 (Trace Semantics) *Let M be an* UPPAAL *model. Then the* trace semantics *of M, written $\mathcal{T}(M)$, is the set of timed traces according to Definition 2.8.*

Note that timed traces are memoryless in the sense that the possible futures do only depend on a configuration and not on the history. If two traces $\sigma_1, \sigma_2 \in \mathcal{T}(M)$ contain the same configuration $\mathbf{s}$, the prefixes leading to $\mathbf{s}$ can be interchanged and the resulting sequences are both again timed traces in $\mathcal{T}(M)$. This property is sometimes called *fusion closure*.

We note that the UPPAAL timed automata model has been equipped with semantics before, in particular in [Pet99]. However, the latter does not correspond to the implementation of committed locations as implemented in UPPAAL 3.0.x, 3.2.x, and later. In Figure 2.2 the control situation `A.F and B.F and C.F` can not be reached according to [Pet99] p. 140 (second bullet point). In the implementation it can be reached, and our semantics reflects this.

2.3 The Logic Language of UPPAAL

The UPPAAL *model checking engine allows to automatically establish or refute properties that are expressed in a specification language. This language is a subset of* timed computation tree logic *(TCTL, [ACD93]), where primitive expressions are location names, variables, and clocks from the modeled system.*

We define validity of formulas in the specification language relative to the semantics given in the previous section.

2.3.1 Local Properties

A local property is a condition that for a specific configuration is either **true** or **false**. The basic building blocks are expressions over locations, variables, and clocks. It is crucial for the efficiency of property verifications that clocks can only be compared to integer values, see Chapter 4.

Definition 2.11 (Local Property)
Given an UPPAAL *model* $\langle \vec{A}, Vars, Clocks, Channels, Type \rangle$. *A formula* φ *is a* local property *iff it is formed according to the following syntactic rules.*

$\varphi ::=$	`deadlock`	
$\mid$	$A.l$	*for* $A \in \vec{A}$ *and* $l \in L_A$
$\mid$	$x \bowtie$ `c`	*for* $x \in Clocks$, $\bowtie \in \{$`<`, `<=`, `==`, `>=`, `>`$\}$, `c` $\in \mathbb{Z}$
$\mid$	$x - y \bowtie$ `c`	*for* $x, y \in Clocks$, $\bowtie \in \{$`<`, `<=`, `==`, `>=`, `>`$\}$, *and* `c` $\in \mathbb{Z}$
$\mid$	$a \bowtie b$	*for* $a, b \in Vars \cup \mathbb{Z}$, $\bowtie \in \{$`<`, `<=`, `!=`, `==`, `>=`, `>`$\}$
$\mid$	`(`φ_1`)`	*for* φ_1 *a local property*
$\mid$	`not` φ_1	*for* φ_1 *a local property*
$\mid$	φ_1 `or` φ_2	*for* φ_1, φ_2 *local properties (logical OR)*
$\mid$	φ_1 `and` φ_2	*for* φ_1, φ_2 *local properties (logical AND)*
$\mid$	φ_1 `imply` φ_2	*for* φ_1, φ_2 *local properties (logical implication)*

The truth value of a local property can effectively be evaluated in a configuration $\mathbf{s}$.

Definition 2.12 (Validity of a Local Property) *A local property* φ *is* valid *in a configuration* $\mathbf{s} = (\vec{l}, e, \nu)$, *in symbols* $\mathbf{s} \models_{loc} \varphi$, *iff it is valid according to the following structural definitions.*

$\mathbf{s} \models_{loc}$ `deadlock`	*iff*	*no delay or action steps are enabled in* $\mathbf{s}$
$\mathbf{s} \models_{loc} A.l$	*iff*	$l = l_i$ *in* $\vec{l}$ *for* $A = A_i$ *in* $\vec{A}$
$\mathbf{s} \models_{loc} x \bowtie$ `c`	*iff*	$\nu(x) \bowtie$ `c`, $\bowtie \in \{$`<`, `<=`, `==`, `>=`, `>`$\}$
$\mathbf{s} \models_{loc} x - y \bowtie$ `c`	*iff*	$\nu(x) - \nu(y) \bowtie$ `c`, $\bowtie \in \{$`<`, `<=`, `==`, `>=`, `>`$\}$
$\mathbf{s} \models_{loc} a \bowtie b$	*iff*	$e(a) \bowtie e(b)$, $\bowtie \in \{$`<`, `<=`, `!=`, `==`, `>=`, `>`$\}$
$\mathbf{s} \models_{loc}$ `(`φ_1`)`	*iff*	$\mathbf{s} \models_{loc} \varphi_1$
$\mathbf{s} \models_{loc}$ `not` φ_1	*iff*	$\neg(\mathbf{s} \models_{loc} \varphi_1)$
$\mathbf{s} \models_{loc} \varphi_1$ `or` φ_2	*iff*	$\mathbf{s} \models_{loc} \varphi_1$ *or* $\mathbf{s} \models_{loc} \varphi_2$
$\mathbf{s} \models_{loc} \varphi_1$ `and` φ_2	*iff*	$\mathbf{s} \models_{loc} \varphi_1$ *and* $\mathbf{s} \models_{loc} \varphi_2$
$\mathbf{s} \models_{loc} \varphi_1$ `imply` φ_2	*iff*	$\neg(\mathbf{s} \models_{loc} \varphi_1)$ *or* $\mathbf{s} \models_{loc} \varphi_2$

`E<>` φ	reachability of φ	
`A[]` φ	safety (invariantly φ)	
`E[]` φ	possibly always φ	
`A<>` φ	inevitably φ	
φ `-->` ψ	unbounded response	
	(corresponds to `A[]` (φ $\Rightarrow$ `A<>` ψ))	φ, ψ: local properties

Figure 2.3: The Classes of TCTL Formulas that UPPAAL can Model Check.

Above, φ_1 and φ_2 stand for local properties.

This notion of locality must not be confused with locality in the sense of "local to one process." The UPPAAL language allows also to declare variables and clocks locally to one process P and uses the syntax `P.var` to identify the `var` that is local to P. Note that every locally declared variable or clock can be equivalently replaced by a global one under appropriate renaming of labels. For simplicity we therefore treat all variables and clocks as global.

2.3.2 Temporal Properties

The five classes of temporal properties that UPPAAL can effectively verify are summarized in Figure 2.3. We define the validity of temporal properties via our trace semantics (Definition 2.10). We chose to give the direct definition of three of the classes and define the remaining two classes as syntactic duals.

Definition 2.13 (Temporal Properties)
Let $M = \langle \vec{A}, Vars, Clocks, Channels, Type \rangle$ be an UPPAAL *model and let φ and ψ be local properties. The* validity of temporal properties *is defined for the classes* `A[]`*,* `A<>`*, and* `-->` *as follows.*

$$M \models \texttt{A[]}\ \varphi \quad \textit{iff} \quad \forall\{(\vec{l}, e, \nu)\}^K \in \mathcal{T}(M).\ \forall k \leq K.\ (\vec{l}, e, \nu)^k \models_{loc} \varphi$$

$$M \models \texttt{A<>}\ \varphi \quad \textit{iff} \quad \forall\{(\vec{l}, e, \nu)\}^K \in \mathcal{T}(M).\ \exists k \leq K.\ (\vec{l}, e, \nu)^k \models_{loc} \varphi$$

$$M \models \varphi\ \texttt{-->}\ \psi \quad \textit{iff} \quad \forall\{(\vec{l}, e, \nu)\}^K \in \mathcal{T}(M).\ \forall k \leq K.\ \textit{if}\ (\vec{l}, e, \nu)^k \models_{loc} \varphi, \textit{ then } \exists k' \geq k \textit{ with } (\vec{l}, e, \nu)^{k'} \models_{loc} \psi$$

The two temporal property classes dual to `A[]` *and* `A<>` *are defined below.*

$$M \models \texttt{E<>}\ \varphi \quad \textit{iff} \quad \neg(M \models \texttt{A[]}\ \texttt{not}(\varphi))$$

$$M \models \texttt{E[]}\ \varphi \quad \textit{iff} \quad \neg(M \models \texttt{A<>}\ \texttt{not}(\varphi))$$

Example 2.14 (Fischer's Mutex, Continued) *The mutual exclusion property of the* UPPAAL *model in Example 2.1 can be expressed as* `not ( P1.cs and P2.cs )`*. This is a local property that has to hold invariantly, i.e., it should be true that* $Fischer_2 \models$ `A[] not ( P1.cs and P2.cs )`*.*

Other temporal properties that should hold include, e.g., that every process can *reach the critical section:* `E<> P1.cs` *and* `E<> P2.cs` .

2.4 Reflection: What Kind of Tool is UPPAAL?

UPPAAL has reached its level of maturity via many years of development by the two academic groups Uppsala and Aalborg. However, the number of contributers is much larger. Some of the best ideas in the field condensed into optimization options that were implemented in the tool. We come back to this in Chapter 5

The input language is basically a timed automata model that is extended with interleaved parallelism and hand-shake synchronization. Syntactic sugar, like variables, eases the modeling process substantially. The tool is freely available for Linux, MS Windows, and Sun-OS[1] and has a graphical front-end (written in **Java**). The download requests indicate that is in fact very popular in teaching.

However, UPPAAL is not primarily an educational tool. It has been the strong ambition of the developers to make it applicable on real-life examples. One symptom for this is the restricted specification language that cuts out a subset of TCTL that has reasonable chances for efficient treatment. An extended reachability analysis suffices, and the generation of the region graph [ACD93] is avoided in general, see Chapter 4.

The success of these efforts is visible along two main axes of large scale application: protocols and embedded systems. We point out that the major actors in the verification of larger examples are all modeling experts with an academic background. It is crucial to focus on the relevant parts of a system and avoid unnecessary explosion of the state space.

UPPAAL is not a modeling tool for design. The timed automata model is much more restricted than a formalism that a system developer would use. One of the important missing features is hierarchical structure.

Most interesting properties in a real-world design language can be expected to be undecidable. Automated analysis then requires an abstraction step. To establish soundness of this step, it has to be clear *what* gets abstracted. In compiler optimization, for example, safe over-approximation by replacing data domains by Boolean values has been very successful (e.g., [NNH99]). Here data is abstracted, but control structure is preserved.

There is a gap between a design tool and a formalism for automated analysis. The former tends to have rich data types, powerful synchronization mechanisms, and hierarchical organization. The latter has the strong obligation to remain in a decidable fragment.

In the following Chapter, we close a part of this gap. We introduce a hierarchical timed automata formalism that is as powerful as UPPAAL's language and can in fact be translated to it.

[1] http://www.uppaal.com

Chapter 3

Hierarchical Timed Automata

It is a great thing in computer science that so many standards exist;
you can always choose the one you like best.
And should you like none at all, you make a new one.

— Prof. Peter Schulthess, lecturing in Ulm

We define a formalism for timed systems that is halfway between UML statecharts and Uppaal *timed automata. Basically we extend timed automata with a statechart-style hierarchy and parallelism on any level. The resulting language is described by a formal syntax and given a operational semantics. Considering the rich set of existing formal statechart-like languages—including several timed variations—, the introduction of yet another formalisms might come as a surprise. It is motivated along two dimensions.*

First, we are primarily concerned with the formal analysis *of models in our language. In particular, we plan to pursue a model checking approach that is powerful enough to capture the complete behavior of a system with respect to a timed logic. To deal with the high computational complexity, we strive to benefit from the intensive research on the timed automata model. This dictates to restrict our formalism to decidable primitives that moreover allow for reasonable efficiency in the exhaustive analysis of a system.*

Second, the multitude of variations in the statechart formalism makes the choice of one formalism not easier. No two variations we know of are comparable. We note a trend to treat statecharts as a high level programming language, e.g., by attaching **C++** *code to states and transitions. It is conceivable that algorithmic treatment of this requires an abstraction step. The anchor of our formalism is the possibility for fully automatic analysis. As a price, the translation of other formalisms into it might have to be an* abstraction function. *This still allows for a faithful analysis with respect to, e.g., safety properties.*

Thus our language is structurally close to full-featured statechart formalisms and conceptually close to timed automata. The former is incorporated, e.g., by the Rhapsody *tool, and the latter by the real-time model checking tool* Uppaal.

We fist give an informal introduction and then define the syntax of our formalism. Next we present the operational semantics. We discuss the encoding of events and give an argument against the introduction of event queues.

3.1 Syntax of Hierarchical Timed Automata

Hierarchical Timed Automata (HTAs) are motivated by the statechart formalism (Section 1.2). As the main syntactic restriction the event communication is replaced by a less expressive hand-shake synchronization. This is necessary to maintain decidability (see Section 3.3).

We introduce the syntax of HTAs first intuitively and then by a formal definition.

3.1.1 A Restricted Statechart Formalism

Since we are primarily interested in formal verification, we restrict the rich and expressive UML statechart formalism. Timed behavior is reflected by (formal) clocks, timed guards, and invariants. Our goal is to tailor a formalism where essential properties remain decidable.

Unlike in UML, where statecharts give rise to the incarnation of objects, we treat a statechart itself as behavioral entity. The notion of thread execution is simplified to the parallel composition of state machines. Relationships to other UML diagrams are dropped.

Our formalism does not support special-purpose modeling constructs, like synchronization states. Some UML tools allow to use C++ as an action language, i.e., C++ code can be arbitrarily added to transitions or states. Formal verification of this is out of scope of this work, we restrict to primitive functions and basic variable assignments. Event communication is simplified to the case where two parts of the system synchronize via handshake.

Some of the restrictions we make can be relaxed as explained in Section 3.5. What we preserve is the essence of the statechart formalism: hierarchical structure, parallel composition at any level, synchronization of remote parts, and history.

3.1.2 Data Components

We introduce the data components of hierarchical timed automata that are used in guards, synchronizations, resets, and assignment expressions. Some of this data is kept local to a superstate S.

Integer variables. Let *Var* be a finite set of integer variables. $Var(S) \subseteq Var$ is the set of integer variables local to a superstate S.

Clocks. Let *Clocks* be a finite set of clock variables. The set $Clocks(S) \subseteq Clocks$ denotes the clocks local to a superstate S. If S has a history entry, $Clocks(S)$ contains only clocks that are explicitly declared as *forgetful*. Other locally declared clocks of S belong to $Clocks(root)$.

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

Synchronizations. *Channels* gives rise to a finite set of channel synchronizations, called *Sync*. For $c \in Channels$, $c?$, $c! \in Sync$.

Guards and invariants. A data constraints is a boolean expressions of the form $E \bowtie E$, where E is an arithmetic expression over *Var* and $\bowtie \in \{<, >, =, \leq, \geq\}$. A clock constraints is an expressions 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 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, *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.

3.1.3 Structural Components

We give now the formal definition of our hierarchical timed automaton.

Definition 3.1 (Hierarchical Timed Automaton (HTA))
A hierarchical timed automaton is a tuple $\langle \mathcal{S}, \mathcal{S}_0, \eta, type, Var, Clocks, Channels, Inv, T \rangle$ *where*

- $\mathcal{S}$ *is a finite set of locations.*
- $\mathcal{S}_0 \subseteq \mathcal{S}$ *is a set of initial locations.*
- $\eta : \mathcal{S} \to 2^{\mathcal{S}}$. η *maps a location* S *to all possible* substates *of* S. η *is required to give rise to a tree structure where a special superstate* root $\in \mathcal{S}$ *is the root. We readily extend* η *to operate on sets of locations in the obvious way.*
- $type : \mathcal{S} \to \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$ *is the type function for locations. Superstates are of type AND or XOR.*
- $Var, Clocks, Channels$ *are sets of variables, clocks, and channels. They give rise to* Guard, Reset, Sync, *and* Invariant *as described in Section 3.1.2.*

- $Inv : \mathcal{S} \to Invariant$ *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 transitions. A transition connects two locations* S *and* S'*, has a guard* g*, an assignment* r *(including clock resets), and an urgency flag* u*.* S *is called the* source *and* S' *is called the* target *of the transition. We use the notation* $S \xrightarrow{g,s,r,u} S'$ *for this and omit* g, s, r, u*, when they are necessarily absent (or* **false***, in the case of* u*).*

Notational conventions. We use the predicate notation $TYPE(S)$ for $TYPE \in \{AND, XOR, BASIC, ENTRY, EXIT, HISTORY\}$, $S \in \mathcal{S}$. E.g., $AND(S)$ 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

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

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

We introduce $\tilde{\eta}$ to refer to the children that are proper locations.

$$\tilde{\eta}(S) := \{b \in \eta(S) \,\big|\, BASIC(b) \vee XOR(b) \vee AND(b)\}$$

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

3.1.4 Well-Formedness Constraints

We give a set of well-formedness constraints to ensure consistency, grouped as for the syntactic categories locations, initial locations, variables, entries, and transitions.

Location constraints. We require a number of sanity properties on locations and structure:

(i) The function η gives rise to a proper tree rooted at $root$, and $\mathcal{S} = \eta^*(root)$.
(ii) Only superstates contain other locations: $AND(S) \vee XOR(S) \Leftrightarrow \eta(S) \neq \varnothing$.
(iii) Substates of AND superstates are not basic: $AND(S) \wedge b \in \eta(S) \Rightarrow \neg BASIC(b)$.
(iv) No invariants on pseudo-locations: $HENTRY(S) \vee EXIT(S) \Rightarrow Inv(S) = \mathbf{true}$.
(v) For every superstate S, at most one exit can be declared to be the *default exit*. If existent, the default exit is reachable from every location in S.

Initial location constraints. $\mathcal{S}_0$ has to correspond to a consistent and proper control situation, i.e., $root \in \mathcal{S}_0$ and for every $S \in \mathcal{S}_0$ the following holds:
(i) $BASIC(S) \vee XOR(S) \vee AND(S)$,
(ii) $S = root \vee \eta^{-1}(S) \in \mathcal{S}_0$,
(iii) $XOR(S) \Rightarrow |\eta(S) \cap \mathcal{S}_0| = 1$, and
(iv) $AND(S) \Rightarrow \eta(S) \cap \mathcal{S}_0 = \tilde{\eta}(S)$.

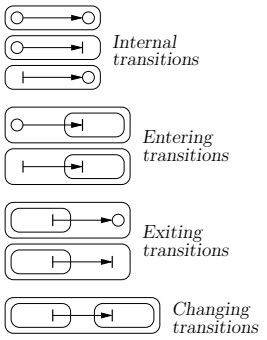
Variable constraints. We explicitly disallow conflict in assignments in synchronizing transitions:
It holds that $S_1 \xrightarrow{g,c!,r,u} S_2$, $S_1' \xrightarrow{g',c?,r',u'} S_2' \in T \Rightarrow vars(r) \cap vars(r') = \varnothing$, where $vars(r)$ is the set of integer variables occurring in r. We require an analogous constraint to hold for the pseudo-transitions originating in the entry of an *AND* superstate.

Static scope: For $S_1 \xrightarrow{g,s,r,u} S_2 \in T$, g, r are defined over $Var^+(\eta^{-1}(S_1)) \cup Clocks^+(\eta^{-1}(S_1))$ and s is defined over $Channels^+(\eta^{-1}(S_1))$.

Entry constraints. 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 substate $B_i \in \tilde{\eta}(\eta^{-1}(S))$, and $e_i \in \eta(B_i)$.
In case of $HISTORY(e)$, outgoing transitions declare the *default history locations*. At most one entry of a superstate can be declared to be the *default entry*. If a superstate S has a history entry, then every substate B of S has to provide a history entry or a default entry.

Transition constraints. Transitions have to respect the structure given in η and cannot cross levels in the hierarchy, except via connecting to entries or exits. The set of legal transitions is given in Table 3.1. Note that transitions cannot lead directly from entries to exits. The internal transitions are those made inside one superstate: from a state to a state, from a state to an exit or from an entry to a state. The constraint expresses that the parent state must be the same. The entering transition is from a state to an entry and the fork transition is from an entry to an entry. The constraints express the transition to a nested state. The exiting and join transitions are symmetric to entering and fork. The changing transition is from the exit of a superstate to the entry of another superstate. The constraint states that both superstates must have a common parent.

Transitions $S \xrightarrow{g,s,r,u} S'$ with $HENTRY(S)$ or $EXIT(S')$ are called *pseudo-transitions*. 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-transition of the form $S \xrightarrow{r} S'$ are allowed. For $EXIT(S')$, only pseudo-transition of the form $S \xrightarrow{g} S'$ are allowed. For $EXIT(S) \wedge EXIT(S')$, this is further restricted to be of the form $S \rightarrow S'$.



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

Table 3.1: Overview on Legal Transitions $S \xrightarrow{g,s,r,u} S'$.

3.2 Operational Semantics of HTAs

We define now the operational semantics of the hierarchical timed automaton formalism. Legal steps between configurations of a HTA give rise to a set of traces.

A *configuration* captures a snapshot of the system, i.e., the active locations, the integer variable values, the clock values, and the history of some superstates. Configurations are of the form (ρ, μ, ν, θ), where

- $\rho : \mathcal{S} \to 2^{\mathcal{S}}$ captures the control situation. ρ can be understood as a partial, dynamic version of η that maps every superstate S to the set of active substates. If a superstate S is not active, $\rho(S) = \varnothing$. We define $\mathit{Active}(S) := S \in \rho^+(\mathit{root})$, where $\rho^+(S)$ is the set of all active substates of S. Notice that $\mathit{Active}(S) \Leftrightarrow S \in \rho(\eta^{-1}(S))$.
- $\mu : \mathcal{S} \to (\mathbb{Z})^*$. μ gives the valuation of the local integer variables of a superstate S as a finite tuple of integer numbers. If $\neg\mathit{Active}(S)$ then $\mu(S) = \lambda$ (the empty tuple). If $\mathit{Active}(S)$ then we require that $|\mu(S)| = |\mathit{Var}(S)|$ and μ is consistent with respect to the value of shared variables (i.e., always maps to the same value). We use $\mu(S)(a)$ to denote the value of $a \in \mathit{Var}(S)$. When entering a non-basic location, local variables are added to μ and set to an initial value (0 by default). We use the shorthand $0^{\mathit{Var}(S)}$ for the tuple $(0, 0 \ldots 0)$ with arity $|\mathit{Var}(S)|$.
- $\nu : \mathcal{S} \to (\mathbb{R}_{\geq 0})^*$. ν gives the real valuation of the clocks $\mathit{Clocks}(S)$ defined locally to the superstate S, thus $|\nu(S)| = |\mathit{Clocks}(S)|$. If $\neg\mathit{Active}(S)$ then $\nu(S) = \lambda$.
- θ reflects the history that might be restored by entering superstates via history entries. It is split up in the two functions θ_{state} and θ_{var}, where $\theta_{state}(S)$ returns the last visited substate of S—or an entry of the substate, in the case where the substate is not basic—(to restore $\rho(S)$), and $\theta_{var}(S)$ returns a vector of values for the local integer variables.
  There is no history for clocks at the semantics level, all non-forgetful clocks belong to $\mathit{Clocks}(\mathit{root})$.

We call a configuration where all S in $\rho^+(root)$ are of type *BASIC*, *XOR*, or *AND* a *proper configuration*.

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

Initially, $\forall S \in \mathcal{S}.\mathit{HasHistory}(S) \Rightarrow \theta_{state}(S) = \mathit{HEntry}(S) \wedge \theta_{var}(S) = 0^{\mathrm{Var}(S)}$.

Reached locations by forks. In order to denote the set of locations reached by following a fork, we define the function $\mathit{Targets}_\theta : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$ relative to θ.

$$\mathit{Targets}_\theta(L) := L \cup \bigcup_{S \in L} \{b \,|\, b \in \theta_{state}(S) \wedge \mathit{HISTORY}(S)\} \cup \{b \,|\, S \xrightarrow{r} b \wedge \mathit{ENTRY}(S)\}$$

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

Configuration vector transformation. Taking a transition $t : S \xrightarrow{g,s,r,u} S'$ entails in general 1. executing a join to exit S, 2. taking the proper transition 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 *proper step*. We represent a proper step formally by a transformation function $\mathcal{T}_t$, which depends on a particular transition t. The three parts of this step are described as follows.

1. *join:*
   (ρ, μ, ν, θ) is transformed to $(\rho^1, \mu^1, \nu^1, \theta^1)$ as follows:
   ρ is updated to $\rho^1 := \rho[\forall b \in \rho^+(S).\ b \mapsto \varnothing]$.
   μ is updated to $\mu^1 := \mu[\forall b \in \rho^+(S).\ b \mapsto \lambda]$.
   ν is updated to $\nu^1 := \nu[\forall b \in \rho^+(S).\ b \mapsto \lambda]$.

   If $\mathit{EXIT}(S)$, the history is recorded. Let H be the set of superstates $h \in \rho^+(\eta^{-1}(S))$, where $\mathit{HasHistory}(h)$ holds. Then
   $\theta^1_{state} := \theta_{state}[\forall h \in H.\ h \mapsto \mathit{HEntry}(\rho(h))]$ and
   $\theta^1_{var} := \theta_{var}[\forall h \in H.\ h \mapsto \mu(h)]$.
   If $\neg\mathit{EXIT}(S)$ or $H = \varnothing$, then $\theta^1 := \theta$.

2. *proper transition 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 $Targets^*_{\theta^2}(S')$. Note that $\rho^2(b) = \varnothing$ for all $b \in \eta^+(S')$. Thus we can compute ρ^3 as follows:

   $\rho^3 := \rho^2$
   FORALL $b \in Targets^*_{\theta^2}(S')$
   IF $ENTRY(b)$
   THEN $\rho^3(\eta^{-2}(b)) := \rho^3(\eta^{-2}(b)) \cup \{\eta^{-1}(b)\}$
   ELSE $\rho^3(\eta^{-1}(b)) := \{b\}$ /⋆ *BASIC* ⋆/

   μ^3 is derived from μ^2 by first initializing all local variables of the superstates B in $Targets^*_{\theta^2}(S')$, i.e., $\mu^3(Var(B)) := 0^{Var(B)}$. If $HasHistory(B)$, $\theta_{var}(B)$ is used instead of $0^{Var(B)}$. Then all variable assignments and clock-resets along the pseudo-transitions belonging to this fork are executed to update μ^3 and ν^3. The history does not change; θ^3 is identical to θ^2.

Note that parts 1. and 3. correspond to the identity transformation, if S and S' are basic locations. We define the configuration vector transformation $\mathcal{T}_t$ for a transition $t : S \xrightarrow{g,s,r,u} S'$:

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

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 configuration corresponding to transition t.

Starting points for joins. A superstate S can only be exited, if all its parallel substates can synchronize on this exit. For an exit $e \in \eta(S)$ we recursively define the family of sets of exits $PreExitSets(e)$. Each element E of $PreExitSets(e)$ is itself a set of exits. If transitions are enabled to all exits in E, then all substates can synchronize.

$$PreExitSets(e) := \begin{cases} \bigcup\limits_{b_1,\ldots,b_k} \boxtimes\limits_{1\le i\le k} PreExitSets(b_i), \text{ where} \\ \qquad k = |\tilde{\eta}(\eta^{-1}(e))|,\ \{b_1,\ldots,b_k\} \subseteq \eta^+(\eta^{-1}(e)), \\ \qquad \forall i.EXIT(b_i) \ \wedge\ b_i \rightarrow e \in T \\ \qquad \eta^{-1}(\{b_1,\ldots,b_k\}) = \tilde{\eta}(e) \end{cases} \text{if } \begin{array}{l} EXIT(e)\wedge \\ AND(\eta^{-1}(e)) \end{array} \\ \\ \begin{cases} \bigcup\limits_{m\in\eta(\eta^{-1}(e))} PreExitSets(m), \text{ where } m \xrightarrow{g,r} e \in T \\ \qquad \cup\, \{\{e\}\} \end{cases} \text{ if } \begin{array}{l} EXIT(e)\wedge \\ XOR(\eta^{-1}(e)) \end{array} \\ \\ \{\,\{\}\,\} \qquad\qquad \text{if } BASIC(e) \end{cases}$$

Here, the operator $\boxtimes : (2^{2^{\mathcal{S}}}) \times (2^{2^{\mathcal{S}}}) \rightarrow 2^{2^{\mathcal{S}}}$ is a product over families of sets, i.e., it maps $(\{A_1,\ldots,A_a\},\{B_1,\ldots,B_b\})$ to $\{A_1\cup B_1, A_1\cup B_2, \ldots, A_a\cup B_b\}$ 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 set of active leaves (in the tree described by ρ), which are the innermost active states in a superstate S:

$$\mathit{Leaves}(\rho, S) \quad := \quad \{b \in \rho^+(S) \,\big|\, \rho(b) = \varnothing\}$$

The predicate expressing that all the substates of a state S can synchronize on a join is:

$$\begin{aligned}\mathit{JoinEnabled}(\rho, \mu, \nu, S) \;:=\; & \mathit{BASIC}(S) \vee \\ & \exists E \in \mathit{PreExitSets}(S).\ \forall b \in \mathit{Leaves}(\rho, S).\ \exists b' \in E. \\ & \qquad b \xrightarrow{g} b' \ \wedge\ g(\mu, \nu)\end{aligned}$$

Note that *JoinEnabled* is trivially true for a basic location S.

For the invariants of a location we use a function $\mathit{Inv}_\nu : \mathcal{S} \to \{\mathbf{true}, \mathbf{false}\}$ that evaluates the invariant of a given location with respect to a clock evaluation ν. We use the predicate $\mathit{Inv}(\rho, \nu)$ to express that for control situation ρ and clock valuation ν all invariants are satisfied.

$$\mathit{Inv}(\rho, \nu) := \bigwedge_{b \in \rho^+(\mathit{root})} \mathit{Inv}_\nu(b)$$

We introduce the predicate *TransitionEnabled* over transitions $t : S \xrightarrow{g,s,r,u} S'$ that evaluates to **true**, if t is enabled.

$$\begin{aligned}&\mathit{TransitionEnabled}(t : S \xrightarrow{g,s,r,u} S', \rho, \mu, \nu) \;:= \\ &\qquad g(\mu, \nu) \wedge \mathit{JoinEnabled}(\rho, \mu, \nu, S) \wedge \mathit{Inv}(\rho^{\mathcal{T}_t}, \nu^{\mathcal{T}_t}) \wedge \neg \mathit{EXIT}(S')\end{aligned}$$

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

$$\begin{aligned}\mathit{UrgentEnabled}(\rho, \mu, \nu) \quad := \quad & \exists t : S \xrightarrow{g,r,u} S'.\ \mathit{TransitionEnabled}(t, \rho, \mu, \nu) \ \wedge\ u \\ \vee \quad & \exists t_1 : S_1 \xrightarrow{g_1,s,r_1,u_1} S_1', t_2 : S_2 \xrightarrow{g_2,\bar{s},r_2,u_2} S_2'. \\ & \qquad \mathit{TransitionEnabled}(t_1, \rho, \mu, \nu) \ \wedge \\ & \qquad \mathit{TransitionEnabled}(t_2, \rho, \mu, \nu) \ \wedge\ (u_1 \vee u_2)\end{aligned}$$

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.

$$\frac{\mathit{TransitionEnabled}(t : S \xrightarrow{g,r,u} S',\ \rho, \mu, \nu)}{(\rho, \mu, \nu, \theta) \xrightarrow{t} \mathcal{T}_t(\rho, \mu, \nu, \theta)}\ \mathit{action}$$

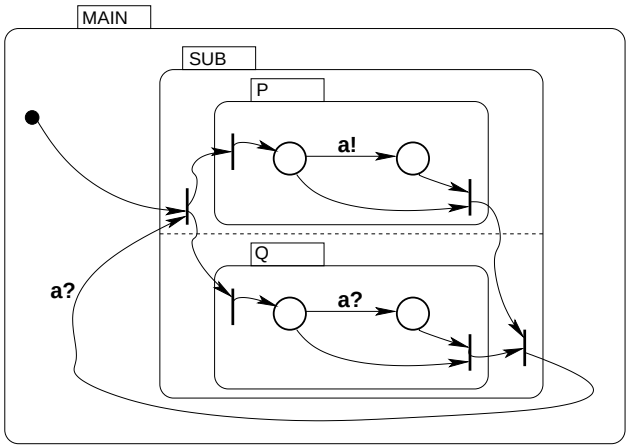


Figure 3.1: The a? Transition Exiting SUB Cannot Synchronize with a! in P.

Here g is the guard of the transition and r the set of resets and assignments. The urgency flag u has no effect here. This rule applies for action transitions between basic locations as well as superstates. In the latter case, this includes the appropriate joins and/or fork operations.

The delay transition rule is:

$$\frac{\mathit{Inv}(\rho, \nu + d) \qquad \neg \mathit{UrgentEnabled}(\rho, \mu, \nu)}{(\rho, \mu, \nu, \theta) \xrightarrow{d} (\rho, \mu, \nu + d, \theta)}\ \mathit{delay}$$

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 configuration only when all invariants are satisfied and there is no urgent transition enabled.

The last transition rule reflects the situation, where two action transitions synchronize via a channel c.

$$\frac{\begin{array}{ll}\mathit{TransitionEnabled}(t_1 : S_1 \xrightarrow{g_1, c!, r_1, u_1} S_1',\ \rho, \mu, \nu) & S_1 \notin \eta^+(S_2) \\ \mathit{TransitionEnabled}(t_2 : S_2 \xrightarrow{g_2, c?, r_2, u_2} S_2',\ \rho, \mu, \nu) & S_2 \notin \eta^+(S_1)\end{array}}{(\rho, \mu, \nu, \theta) \xrightarrow{t_1, t_2} \mathcal{T}_{t_2} \circ \mathcal{T}_{t_1}(\rho, \mu, \nu, \theta)}\ \mathit{sync}$$

We choose the order first t_1, then t_2 here. This could be inverted, since the well-formedness constraints 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 of a superstate with its own descendants. For example, in Figure 3.1 The a? transition exiting SUB cannot synchronize with the a! transition in P.

If no action transition is enabled or becomes enabled when time progresses, we have a *deadlock* configuration, which is typically a bad thing. 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.

Similar to Definition 2.8, we define a set of timed traces for an HTA that capture its behavior. We explicitly exclude sequences that are zeno or not maximally extended.

Definition 3.2 (HTA Timed Trace Semantics)
Let $M = \langle \mathcal{S}, \mathcal{S}_0, \eta, type, Var, Clocks, Channels, Inv, T \rangle$ *be an hierarchical timed automaton. A sequence of configurations* $\{(\rho, \mu, \nu, \theta)\}^K = (\rho, \mu, \nu, \theta)^0, (\rho, \mu, \nu, \theta)^1, \ldots$ *of length* $K \in I\!N \cup \{\infty\}$ *is a* timed trace *of* M*, if*

(i) It starts at the initial configuration, i.e, for $(\rho, \mu, \nu, \theta)^0$*:* $\mathcal{S}_0 = (\rho^0)^*(root)$*,* $\mu = [Var \mapsto (0)^*]$*, and* $\nu = [Clocks \mapsto 0\,]$*,*

(ii) Every step from $(\rho, \mu, \nu, \theta)^k$ *to from* $(\rho, \mu, \nu, \theta)^{k+1}$ *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, and*

(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 (*$\forall x \in Clocks \forall c \in I\!N\, \exists k.\ \nu^k(x) > c$*).*

The set of timed traces, denoted by $\mathcal{T}r(M)$*, is the* timed trace semantics *for* M*.*

3.3 Unbounded Event Queues

Events are not included in the language of hierarchical timed automata, since they—together with unbounded event queue—give rise to an infinite state system. We formalize an undecidability result for this.

In the following we give a minimal definition of a computational structure that uses an unbounded event queue to trigger subsequent steps. We introduce only one process that both writes and reads this queue. In more application-oriented settings the definition would be extended to allow for multiple processes and more then one such queue. However, we intend to show that even for this restricted definition the reachability problem is undecidable.

Definition 3.3 (Event System)
An event system *is a tuple* $\langle \mathbb{L}, \mathbb{E}, \mathbb{T}, l_0 \rangle$*, where*

- $\mathbb{L}$ *is a finite set of locations,*
- $\mathbb{E}$ *is a finite set of events,*
- $\mathbb{T} \subseteq \mathbb{L} \times (\mathbb{E} \cup \{\}) \times (\mathbb{E} \cup \{\}) \times \mathbb{L}$ *is a set of transitions, and*

- $l_0 \in \mathbb{L}$ *is the initial location.*

We write $l \xrightarrow{\mathfrak{e}_1/\mathfrak{e}_2} l'$ for the transition that starts at location l, consumes event $\mathfrak{e}_1$, creates $\mathfrak{e}_2$, and reaches location l'.

A *configuration* of an event system is a pair $(l, Q) \in \mathbb{L} \times \mathbb{E}^*$, where Q is the (finite but unbounded) *event queue*. $\mathbb{E}^*$ is the Kleene-star over $\mathbb{E}$, i.e., the set of finite strings composed from elements in $\mathbb{E}$. We use λ to denote the empty string or queue. The concatenation operator $\cdot$ is used to denote head and tail of a queue, e.g., $\mathfrak{e}_1 \cdot Q \cdot \mathfrak{e}_2$ denotes the queue with $\mathfrak{e}_1$ at the first position, $\mathfrak{e}_2$ at the last position, and $Q \in \mathbb{E}^*$ in between.

$\mathbb{T}$ gives rise to the binary *step relation* $\rightarrow_{\mathbb{T}}$ over configurations of an event system as follows.

$$
\begin{array}{lllll}
(l, Q) \rightarrow_{\mathbb{T}} (l', Q') \text{ iff } & l \xrightarrow{/} l' & \text{and} & Q = Q' & \text{, or} \\
& l \xrightarrow{\mathfrak{e}_1/} l' & \text{and} & Q = \mathfrak{e}_1 \cdot Q' & \text{, or} \\
& l \xrightarrow{/\mathfrak{e}_2} l' & \text{and} & Q \cdot \mathfrak{e}_2 = Q' & \text{, or} \\
& l \xrightarrow{\mathfrak{e}_1/\mathfrak{e}_2} l' & \text{and} & Q \cdot \mathfrak{e}_2 = \mathfrak{e}_1 \cdot Q'. &
\end{array}
$$

A *run* of an event system is a finite sequence $\sigma = \big((l_0, Q_0), (l_1, Q_1), \ldots (l_n, Q_n)\big)$ where $(l_0, \lambda) = (l_0, Q_0)$ and $\forall 0 \leq k < n.\ (l_k, Q_k) \rightarrow_{\mathbb{T}} (l_{k+1}, Q_{k+1})$.

3.3.1 Turing Machines and the Halting Problem

To lead the discussion to an undecidable problem we need to introduce the formal notion of a Turing computation. The knowledgeable reader may skip to Section 3.3.2.

We introduce Turing machines as non-deterministic automata with a finite number of control states. An unbounded tape contains two symbols **0** and **1**, we use the symbol # to mark an unread position of the tape. At one position of this tape is a read/write head. In every step this head reads the symbol under it, writes a symbol, and then either moves to the left, to the right, or stays at the current position. We make the assumption that the head never moves over the left bound of the tape and never writes #. Then the finite string over $\{\mathbf{0}, \mathbf{1}\}$ up to the first # completely describes the tape.

Definition 3.4 (Turing machine)
A Turing machine $\mathbf{M}$ *is a tuple* $\langle Q, \Sigma, \Gamma, \delta, q_0, q_{yes}, q_{no} \rangle$*, where*

- Q *is a finite set of states,*
- $\Sigma = \{\mathbf{0}, \mathbf{1}\}$ *is the input alphabet,*
- $\Gamma = \Sigma \cup \{\#\}$ *is the tape alphabet,*
- $\delta \subseteq Q \times \Gamma \times Q \times \Sigma \times \{\mathbf{L}, \mathbf{N}, \mathbf{R}\}$ *is the transition relation,*
- $q_0 \in Q$ *is the starting state,*
- $q_{yes} \in Q$ *is the accepting state, and*
- $q_{no} \in Q$ *is the rejecting state.*

A *configuration* of a Turing machine is a tuple $\langle q, w, i\rangle$, where $q \in Q$ is the current state, $w \in \{\mathbf{0}, \mathbf{1}\}^* \times \{\#\}$ is the current tape, and $i \in \mathbb{N}$ is the position of the tape head, $i \leq |w|$.

The transition relation δ describes all possible computation steps. E.g., assume $(q, \mathbf{X}, q', \mathbf{X}', \mathbf{L}) \in \delta$. Then if in state q and reading $\mathbf{X}$ from the tape, the Turing machine can write $\mathbf{X}'$, move the head to the left, and change to state q'. If the last part of the tuple is $\mathbf{N}$ or $\mathbf{R}$, this indicates that the head does not move ($\mathbf{N}$) respectively moves to the right ($\mathbf{R}$). There can be multiple elements in δ sharing the same first two positions $(q, \mathbf{X}, \ldots)$, thus the behavior is non-deterministic.

A *computation* of a Turing machine on input $w \in \{\mathbf{0}, \mathbf{1}\}^*$ is a sequence of configurations starting at $\langle q_0, w\#, 0\rangle$. The sequence of two subsequent configurations is required to be related via δ. An *accepting computation* ends in a configuration with state q_{yes} and a *rejecting computation* ends in a configuration with state q_{no}.

The *language* $\mathcal{L}_{\mathbf{M}}$ accepted by a Turing machine $\mathbf{M}$ is the set of strings $\mathcal{L}_{\mathbf{M}} \subseteq \Sigma^*$ such that an accepting computations on input x exists if and only if $x \in \mathcal{L}_{\mathbf{M}}$.

Turing machines are a simple but very general model of computation. According to the *Church-Turing thesis*, any algorithm written in a programming language can be described as the encoding of a Turing machine.

It is an important observation that Turing machines can be *encoded* in a finite string of $\mathbf{0}$ and $\mathbf{1}$. Thus a Turing machine $\mathbf{M}$ can be part of the input to another Turing machine. There exists an *universal Turing machine* $\mathbf{M}_U$ that be behaves like $\mathbf{M}$ when the input of $\mathbf{M}_U$ contains the encoding of $\mathbf{M}$. The interested reader is referred to, e.g., [Sip96].

This ability, sometimes called *self-application*, gives rise to a number of logical impossibilities. One of them is the *undecidability* of a formal language $\mathcal{L}$.

Proposition 3.5 (Halting Problem on Empty Tape is Undecidable)
The halting problem on empty tape is undecidable. I.e., there is no Turing machine that accepts exactly the encodings of Turing machines that can reach the accepting state (i.e., "halt") when started on the empty string as input.

There are several variations of Turing machines and we are choosing our definition for the sake of brevity of the following reduction. The fundamental properties, in particular undecidability, proved to be robust with respect to these variations. For a more detailed exposition we refer to the textbooks [HU80, BDG88, Pap94, Sip96].

3.3.2 Undecidability of Unbounded Queues

We show that basic reachability problems is undecidable for systems with unbounded event queues. This is our main argument against including general events into the hierarchical timed automata language.

Theorem 3.6 (Reachability for Event Systems is Undecidable)
Let $\mathcal{E} = \langle \mathbb{L}, \mathbb{E}, \mathbb{T}, l_0 \rangle$ be an event system and $l \in \mathbb{L}$. It is undecidable whether there exists a run for $\mathcal{E}$ leading to the configuration (l, λ).

Proof: (by reduction from the halting problem on empty tape, Proposition 3.5) For a given Turing machine $\mathbf{M}$ we construct an event system $\mathcal{E}_\mathbf{M}$, such that every computation of $\mathbf{M}$ corresponds to a run in $\mathcal{E}_\mathbf{M}$ and vice versa.

The queue of $\mathcal{E}_\mathbf{M}$ is used to encode the tape of $\mathbf{M}$. One special event # marks the end of the tape and one special event $\mathbf{H}$ marks the position of the read/write head, the current position is right to it. In every step the queue is "wrapped around" and only a finite part around the read/write head is modified. This is possible with finite number of locations in the event system.

Reduction. Given a Turing machine $\mathbf{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{yes}, q_{no} \rangle$. We define the corresponding event system as follows. $\mathcal{E}_\mathbf{M} := \langle \mathbb{L}, \mathbb{E}, \mathbb{T}, l_0 \rangle$, where

- $\mathbb{L} := \bigcup_{q_i \in Q} \{ q_i, q_i^{\mathbf{0}\rightarrow}, q_i^{\mathbf{1}\rightarrow}, q_i^{\mathbf{H}}, q_i^{\mathbf{0H}}, q_i^{\mathbf{1H}}, q_i^{\underline{\mathbf{H}}}, q_i^{\underline{\mathbf{0}}}, q_i^{\underline{\mathbf{1}}}, q_i^{\mathbf{1}\underline{\mathbf{H}}}, q_i^{\mathbf{1}\underline{\mathbf{0}}}, q_i^{\mathbf{1}\underline{\mathbf{1}}}, q_i^{\mathbf{0}\underline{\mathbf{H}}}, q_i^{\underline{\mathbf{00}}}, q_i^{\underline{\mathbf{01}}}, q_i^{!\mathbf{0}}, q_i^{!\mathbf{1}}, q_i^{!\mathbf{00}}, q_i^{!\mathbf{01}}, q_i^{!\mathbf{10}}, q_i^{!\mathbf{11}}, q_i^{!\mathbf{0}\_}, q_i^{!\mathbf{1}\_}, q_i^{!\underline{\mathbf{00}}}, q_i^{!\mathbf{0}\underline{\mathbf{1}}}, q_i^{!\mathbf{1}\underline{\mathbf{0}}}, q_i^{!\mathbf{1}\underline{\mathbf{1}}}, q_i^{\star} \}$
  $\cup \{ l_0, l_0' \}$, and
- $\mathbb{E} := \{ \mathbf{0}, \mathbf{1}, \mathbf{H}, \# \}$.

The transition relation $\mathbb{T}$ is defined to match δ. As an initialization we include $l_0 \xrightarrow{/\mathbf{H}} l_0'$ and $l_0' \xrightarrow{/\#} q_0$. This yields the empty tape.

For every $q_i \in Q$ we include a segment like in Figure 3.2. Intuitively, this part reads the event queue up to the read/write head (represented by event $\mathbf{H}$). Since the head can move to the left, it is necessary to remember the last symbol left of the head position if it exists. The rest of the tape is put back into the event queue. The event # marks both the end of the tape and the start of the tape in the next step.

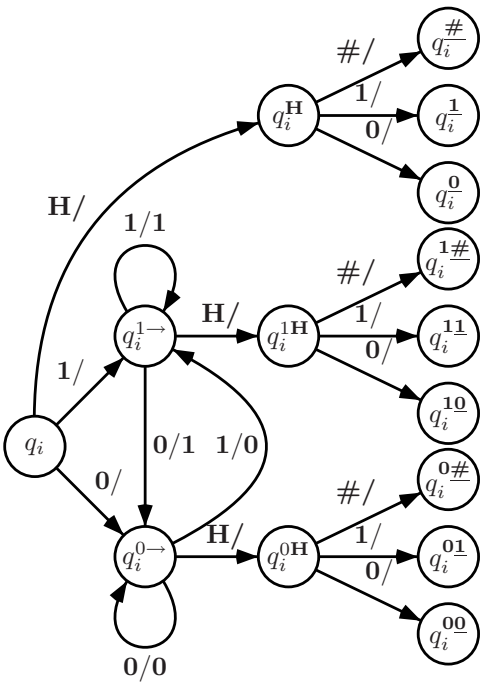
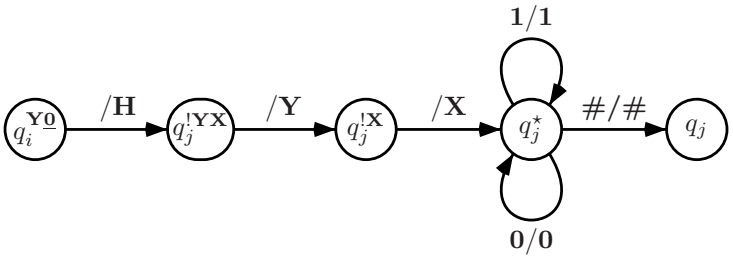
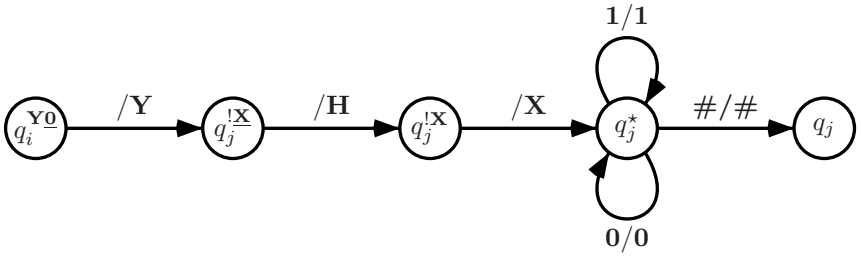


Figure 3.2: Spool Queue to $\mathbf{H}$.

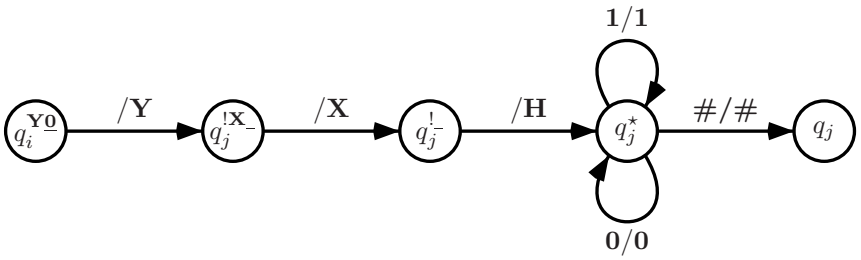
When the read/write head $\mathbf{H}$ is reached, the transition relation δ is mimicked. Every element in δ is encoded in a set of transitions. This not only determines the next q_j, but also moves the head according to the entry $\{\mathbf{L}, \mathbf{N}, \mathbf{R}\}$. We exemplify this for states $q_i^{\mathbf{Y}\underline{\mathbf{0}}}$ with $\mathbf{Y} \in \{\mathbf{0}, \mathbf{1}\}$. The encoding works analogously for states $q_i^{\mathbf{Y}\underline{\mathbf{1}}}$, $q_i^{\underline{\mathbf{0}}}$, and $q_i^{\underline{\mathbf{1}}}$.
For $\mathbf{X}, \mathbf{Y} \in \{\mathbf{0}, \mathbf{1}\}$ and $(q_i, \mathbf{0},\ q_j, \mathbf{X}, \mathbf{L}) \in \delta$ the following transitions are in $\mathbb{T}$.



For $\mathbf{X}, \mathbf{Y} \in \{\mathbf{0}, \mathbf{1}\}$ and $(q_i, \mathbf{0},\ q_j, \mathbf{X}, \mathbf{N}) \in \delta$ the following transitions are in $\mathbb{T}$.



For $\mathbf{X}, \mathbf{Y} \in \{\mathbf{0}, \mathbf{1}\}$ and $(q_i, \mathbf{0},\ q_j, \mathbf{X}, \mathbf{R}) \in \delta$ the following transitions are in $\mathbb{T}$.



Now every step in δ of $\mathbf{M}$ corresponds to a sequence of steps from a configuration (q_i, Q) to a configuration (q_j, Q') in the event system $\mathcal{E}_{\mathbf{M}}$—and vice versa. Thus every computation of $\mathbf{M}$ has a matching run in $\mathcal{E}_{\mathbf{M}}$. Thus $\mathbf{M}$ accepts the empty word if and only if the location q_{yes} is reachable in a run of $\mathcal{E}_{\mathbf{M}}$.

□

We note that our definition of an event system is somewhat artificial, since event queues are typically used as means of communication *between* processes. This is just for the sake of simplicity. The reduction in the previous prove can be modified to two processes that alternate in the computation of the next configuration and send the encoding of the tape to each other.

The presence of clocks is not needed to render the problem undecidable. It is also well-known that for every Turing machine $\mathbf{M}$ there exists an deterministic Turing machine $\mathbf{M}'$ that accepts the same language. For a deterministic Turing machine, the construction from Theorem 3.6 yields a deterministic event system. Thus the root of the undecidability is neither time nor nondeterminism, but the unboundedness of the event queue.

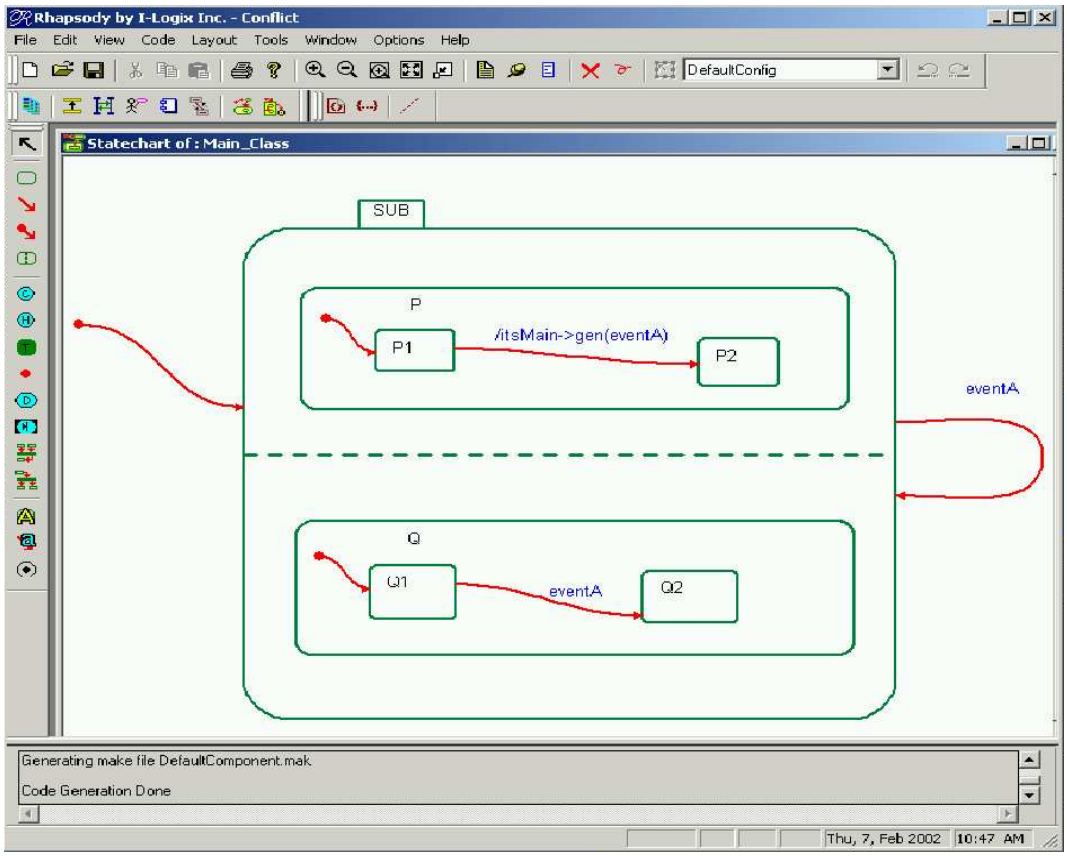


Figure 3.3: In RHAPSODY, UML Statecharts Communicate via Events.

3.4 Partial Encoding of Events

General events are not included in the language of hierarchical timed automata, since this would entail the undecidability of reachability problem (Section 3.3).

As a partial remedy we show how to give an approximative encoding of general events—as they are present in RHAPSODY*—in hierarchical timed automata. This corresponds to situations, where the event queue cannot distinguish the order of arrival.*

3.4.1 Events in RHAPSODY

In the CASE tool RHAPSODY events are a sophisticated means of coordinating parallel components. They are not necessarily atomic, but can be classified according to an event hierarchy. We give a brief summary, the details can be found in [IL00]. There are several ways events can be created, e.g.,

1. along transitions,
2. on entry of superstates,

3. on exit of superstates, or
4. with passage of time.

The generation of events happens by code segments like `itsMain->gen(eventA)`. The syntax explains by the circumstance that RHAPSODY translates parts of the UML model to **C++** source code.[1] The main unit of the translation is a class and its behavior as defined via one statechart. However, class dependencies (like inheritance) and textual annotations also influence the translation.

Events—and also a possible hierarchy on events—are reflected by a collection of classes derived from a ancestor class `OMEvent`. In the *code generation process* the event generation is translated to a **C++** method call on an instance of MAIN, where an object of a class eventA is created.

Once generated, an event goes into one *event queue*. There can be several event queues in one system. As an event reaches the head of a queue, it is *dispatched*, i.e., the event affects all objects in the scope of its queue and possibly entails further transitions. The event is consumed by this. Events are *asynchronous*, i.e., time can pass between the generation of an event and the moment where it is dispatched.

The prime function of events is to *trigger* transitions. Two transitions are in conflict, if they cause the same location to be left. E.g., in Figure 3.3 the two transitions labeled with eventA are in conflict. Some conflicts are resolved by the structure and an (optional) set of *priorities*. If more than two transitions are in conflict and no explicit priorities are defined then only the innermost transition is taken.[2] If a conflict cannot be resolved by this rules, it is unspecified which one of the transitions is taken.

3.4.2 Respecting Number, Ignoring Order

We give one schema to partially encode events via hand-shake synchronization. The idea is to count the occurrences of events. As long as a non-zero number of an event is present, it requires synchronization with all superstates in its scope. This synchronization can be an idle loop or trigger another transition. If another transition is triggered, this can entail the issuing of another event. If this event is already present, the number of occurrences is increased by one.

Figure 3.4 exemplifies this for one event a and process $\mathcal{S}_1$. The event a can be received by all *XOR* superstates $\mathcal{S}_1 \cdots \mathcal{S}_n$. In the *XOR* superstate $\mathcal{S}_1$ only the transition from l_1 to l_2 is triggered by the event a. If $\mathcal{S}_1$ happens to be in other control locations, then then the event a has no effect. The synchronization signal s_S1 is just skipped by taking the self-loops. This includes the situations where $\mathcal{S}_1$ is not active, i.e, where control is at $\mathcal{S}_1$_IDLE.

[1] Other target languages like **C** or **Java** are also possible. The exact nature of the code is dependent on the setup. In particular one can generate code for special realtime environments, e.g., realtime operating systems (RTOSes).

[2] In the predecessor STATEMATE it is the other way round: the outermost transitions have priority by default.

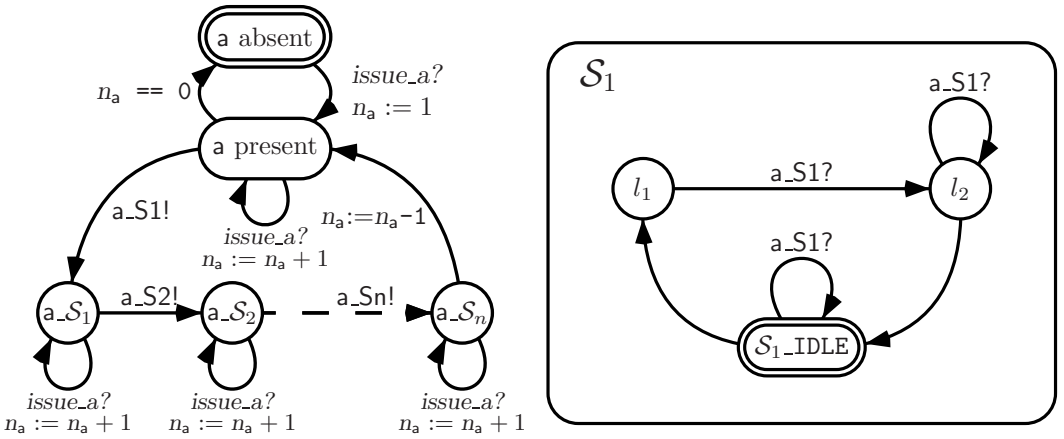


Figure 3.4: Partial Encoding of Event a via a Counter n_a.

Consider situations where transitions triggered by events carry guards. Then it has to be made sure that if the guard evaluates to **true**, those transitions are taken and a self-loop is enabled otherwise. E.g., assume the transition from l_1 to l_2 in Figure 3.4 is guarded by the expression g. Then in the encoding, l_1 has a self-loop synchronizing on a_S1? that is guarded by $\neg g$.

Note that the partial encoding we formulated respects the number, up to arithmetic overflow. If several events are issued at the same point in time, the order of arrival is ignored. The intuition here is that in implementations of distributed systems the precise order of arrival cannot be predicted within one time instance.

Another approach is to maintain finite size queues (also known as bounded buffers). If the queue is full, no further event can be accepted. As a modeling primitive this is common, e.g., in SPIN [Hol97]. For Rhapsody statecharts this approach has been elaborated, e.g., in [Vot02].

3.5 Reflection: Hierarchical Timed Automata

It is perceivable that there is a gap between industrial tools and academic tools. Industrial tools aim to support the design and production activity of their customers. The user interface has to be friendly; employees are going to interact with it for weeks and months. Academic tools aim to support research activity. Implementation is carried out by student programmers or PhD students. The user interface can be anything, even textual, since the typical user is either a researcher or a student.

The hierarchical timed automata formalism is neither the first nor the first timed variation of statecharts. A number of related approaches are compared and classified in [vdB94]. According to this classification, our formalism would be described by the column g/t - (-) + - + - + - + o - + - i c + + + - - - + - - d: graphi-

cal/textual, no negated trigger event, no (implicit) timeout event, timed transitions, no disjunction of trigger events, trigger conditions, no state reference, assignments to variables, no inter-level transition, history mechanism, operational semantics, not compositional, with synchrony hypothesis, not deterministic, interleaved concurrency, continuous time, globally consistent, causal, instantaneous states, no finiteness restriction in number of transitions, no priorities, no non-preemptive interrupt, preemptive interrupt, no distinction of internal and external events, no local events, discrete events.

We substitute "hand-shake synchronization" for "events" in van der Beeck's classification. The main motivation to construct this new formalism is the closeness to the UPPAAL model; a translation to UPPAAL exists, see Chapter 9. We found no existing statechart variant readily appropriate for this purpose. The major omission in HTAs with respect to UML statecharts are events.

There are two main difficulties with events. First, the *precise* notion of events has not (yet) been given in the UML, though version 1.4 is more specific than its predecessors. As a side effect some UML tools (e.g., RHAPSODY) do no longer correspond to this definition. Not all the holes are filled. In particular it is not specified yet if events are instantaneous or are queued and resolve at some later time. An unambiguous definition is a prerequisite for a formal treatment.

Second, if the event queue can grow without bound model checking is undecidable in general. This presents a serious problem, since no complete algorithm can be formulated any more. We argue that this is rather an introduced than an inherent problem. Due to constrained resources in running applications, the event queue usually has a bounded size. The exact bound, however, might not be known a priori. The approach of limiting the size of the event queues is followed in [Vot02].

Another possibility is to reason about event queues that have a certain regular structure. Sets of queue situations can have a finite encoding, though their cardinality is not finite. Here we refer to the work of Abdulla and Jonsson [AJ96, AJ01].

The work on the HTA formalism is continuing. A graphical editor for the language is currently under development at Aalborg University. It uses an XML representation of the described syntax. For practical reasons superstates are not constructed as primitives but generated from parameterized templates. More on this representation can be found in [DM01].

To assert the usability of the HTA formalism bigger examples are needed. However those are tedious to construct without an appropriate editor. We expect that the HTA formalism further evolves once the generation of examples has been made easier.

In the context of the AIT-WOODDES project[3], the HTA formalism is planned to be used as an intermediate format. UML statechart models as constructed by the tool RHAPSODY are to be translated to UPPAAL via the HTA representation. This

[3]AIT-WOODDES: Advanced Information Technology—Workshop on Object-Oriented Design and Development of Embedded Systems. This is a project founded by the European Union, No IST-1999-10069. See http://wooddes.intranet.gr.

requires clearly an abstraction step. For once to safely omit code that is part of the model, and second to approximate events.

Ultimately it would be desirable to make direct use of the structure in terms of a model checking algorithm. It is conjectured that parts of the structure can be exploited directly, either by means of compositional analysis or via reorganization, as performed for a simpler case in Chapter 8. As yet, the maturity of the formalism has to develop further to justify the investment of a direct implementation.

Part II

Algorithmic Verification of Real-Time Systems

There is no time.
There are only clocks.

— A Swedish national diary, town museum Stockholm

The behavior of real-time systems tends to be complex. Some situations occur only under certain interleavings of interactions. For a human designer this makes it difficult and tedious to determine whether his design in fact does what it should do.

In model based development, a digital representation can be analyzed before a system is built. This helps to document and communicate a design. Failure scenarios are reproducible, since the model is completely under control of the developer. The model can also be used to explore the behavior of prototypes, for example by means of simulation.

The approach we follow in this Part is the complete analysis of a system's behavior by means of model checking. Throughout we use a dense model of time, i.e., between any two time instances there is an intermediate one. It has been noted that modeling time in a discrete fashion can miss reachable states, if the granularity of time has to be fixed a priori [Alu91].

Dense real-time models are decidable (for certain syntactic restrictions) and model checking algorithms can be formulated [ACD93]. In the recent years a number of tools have been developed on this background. To name a few of them: Epsilon [CGL93], Polka [HRP94], Rt-Cospan [AK95], Rt-Spin *[TC96],* Treat *[KL96],* Kronos *[BDM+98],* Uppaal *[LPY97],* HyTech *[HHWT97],* CMC *[LL98], and SGM [WH98]. All of them originate from academic environments.*

For dense real-time the state space is generally infinite; the decidability results depend on the fact that it suffices to represent a finite quotient of this space. This is also known as the symbolic representation of infinite sets. In implementations the issue how to represent this quotient turns out to be an important factor.

In Chapter 4 we go into the details of forward analysis as implemented in the tool Uppaal. *We outline a symbolic data-structure and two symbolic algorithms, one for reachability and one for unbounded response. We sketch a correctness proof for these algorithms.*

Model checking algorithms are notoriously consumptive in terms of time and memory. The success of algorithmic treatment often relies on a number of optimizations that tune the efficiency. A number of prominent optimization techniques have been incorporated in the tool Uppaal. *It is difficult to predict the benefit of an optimization. In practice a number of different optimizations have to be tried to find out what works best for a particular problem. In Chapter 5 we strive to give an impression of the impact of optimizations. We apply exhaustive combinations of the optimization options on three classes of benchmark examples and display the obtained run-time data.*

The search for new methods to overcome bottlenecks in real-time model checking is an active field. We contribute two novel techniques to this area. Our first technique (Chapter 6) is an approximation technique. By adding carefully chosen transitions we compute a safe over-approximation of the systems behavior. If we can establish a universal property in the approximation, then it also holds for the original system.

Our second technique (Chapter 7) uses abstractions to compute a coarser representation of the symbolic state space. This is formulated in the framework of abstract interpretation. With the omission of the next-step operator from our logic we are able to incorporate a progress assumption by syntactic means. This solves the problem of spurious loops in the abstracted system and makes our technique suitable for liveness properties.

Chapter 4

Symbolic Forward Analysis

There is no such thing as a correct system.

— Bob Kurshan, Bell Labs, at CAV'97

Formally verify everything!

— Ben Brennan, Intel, at CAV'97

The formal verification of real-time systems is a challenging problem. Serious undecidability results prevent the fully automatic treatment of many interesting classes of timed system, like skewed clocks, and systems with arbitrary large event-queues. In decidable formalisms, a number of hardness-results renders model checking an expensive task.

In practice this is reflected by the notorious inefficiency of model checking engines that even for small models soon exhaust available machine resources and the patience of the user. A number of optimization techniques, data structures, and heuristics was developed that strive to reduce time- and space-consumption wherever possible.

In this Chapter we discuss the real-time model checking algorithm of Uppaal. *First we introduce a formal framework for the symbolic representation of configurations and traces, called symbolic state graphs. It suffices to represent regular sets of clock evaluations that are a generalization of the fundamental clock region construction [ACD93]. As a data structure to represent these sets, called zones, we introduce difference bounded matrices (DBMs). We sketch the model checking algorithms for forward reachability and unbounded response. This spans the whole specification language of* Uppaal. *We use symbolic state graphs to discuss correctness and soundness of the algorithms.*

4.1 Symbolic Representation of Traces

In Chapter 2 the semantics of the timed automata model as used in UPPAAL *is introduced in terms of a set of traces. Since this set is (typically) uncountable, it cannot be used algorithmically.*

In this section we discuss how to represent this set in terms of a symbolic state graph. Later (Section 4.3) we use finite symbolic state graphs to describe the model checking algorithm of UPPAAL.

The excessive number of possible traces has its root in the uncountably many clock evaluations. However, many of theses clock evaluations can be treated uniformly.

The basic idea is to allow for the accumulation of clock evaluations that are in some sense regular. We describe the behavior of an UPPAAL model then by a rooted directed graph. Every node stands for a set of configurations that share the same discrete part. Every edge stands intuitively for one step; delay steps, however, may also occur between two configurations in the same node.

Definition 4.1 (Symbolic State Graph)
Let M be an UPPAAL *model $\langle (A_1, \ldots, A_n), Vars, Clocks, Channels, Type \rangle$. A* symbolic state graph for M *is a rooted directed graph $\mathcal{G} = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G}, \bot_\mathcal{G})$, such that*

- $\mathcal{V}_\mathcal{G} \subseteq \{(\vec{l}, e, \Xi_{Clocks}) \mid \vec{l} \in L_{A_1} \times \cdots \times L_{A_n},$ *e an environment for discrete variables of M, and Ξ_{Clocks} a nonempty set of clock evaluations for Clocks* $\}$
- $\mathcal{E}_\mathcal{G} \subseteq \mathcal{V}_\mathcal{G} \times \mathcal{V}_\mathcal{G}$ *is the edge relation, and*
- $\bot_\mathcal{G} = \big((l_1^0, \ldots, l_n^0), [Vars \mapsto (0)^*], \Xi^0_{Clocks}\big) \in \mathcal{V}_\mathcal{G}$, *where* $[Clocks \mapsto 0] \in \Xi^0_{Clocks}$.

Our definition of symbolic state graph captures only well-formedness constraints and no strong connection to M. Some nodes can be connected without corresponding to a step in the configurations of M. Moreover, $\mathcal{G}$ does not necessarily contain every reachable configuration of M. For example, the graph with no edges and the root as the only node is a symbolic state graph for M. We need additional conditions that have to hold for $\mathcal{G}$ in order to represent M.

Definition 4.2 (Representing Symbolic State Graph)
Let M be an UPPAAL *model $\langle (A_1, \ldots, A_n), Vars, Clocks, Channels, Type \rangle$, and $\mathcal{G}$ be a symbolic state graph for M.* $\mathcal{G}$ represents M *if and only if*
$\forall\, (\vec{l}^{\,1}, e^1, \Xi^1_{Clocks}) \in \mathcal{V}_\mathcal{G}.\ \forall \nu^1 \in \Xi^1_{Clocks}$*:*

- *if* $(\vec{l}^{\,1}, e^1, \nu^1) \stackrel{\alpha}{\Longrightarrow} (\vec{l}^{\,2}, e^2, \nu^2)$, $\alpha \in \{a, \tau\}$,
  then $\exists\, (\vec{l}^{\,2}, e^2, \Xi^2_{Clocks}) \in \mathcal{V}_\mathcal{G}$ *such that*
  $\big((\vec{l}^{\,1}, e^1, \Xi^1_{Clocks}),\ (\vec{l}^{\,2}, e^2, \Xi^2_{Clocks})\big) \in \mathcal{E}_\mathcal{G}$ *and* $\nu^2 \in \Xi^2_{Clocks}$.

- *if* $(\vec{l}^1, e^1, \nu^1) \stackrel{d}{\Longrightarrow} (\vec{l}^1, e^1, \nu^2)$,
  then either $\nu^2 \in \Xi^1_{Clocks}$
  or $\exists\, (\vec{l}^1, e^1, \Xi^2_{Clocks}) \in \mathcal{V}_\mathcal{G}$ *such that*
  $\big((\vec{l}^1, e^1, \Xi^1_{Clocks}),\ (\vec{l}^1, e^1, \Xi^2_{Clocks})\big) \in \mathcal{E}_\mathcal{G}$ *and* $\nu^2 \in \Xi^2_{Clocks}$.

The intuition here is that all possible traces are represented. We make this more precise in the following definition.

Definition 4.3 (Covering)
Let M be an Uppaal *model, $\mathcal{G} = (\mathcal{V}_\mathcal{G}, \mathcal{E}_\mathcal{G}, \perp_\mathcal{G})$ a symbolic state graph for M, and $\mathcal{T}$ be a set of well-formed sequences for M. We say that $\mathcal{G}$* covers *$\mathcal{T}$, if for every $\sigma = \{(\vec{l}, e, \nu)\}^K \in \mathcal{T}$ there exists a sequence of nodes $\{\mathcal{S}\}^K$ in $\mathcal{G}$ mimicking σ, i.e., for every $(\vec{l}^k, e^k, \nu^k)$ with $k < K + 1$*

(i) $\mathcal{S}^k = (\vec{l}^k, e^k, \Xi^k_{Clocks})$ *with* $\nu^k \in \Xi^k_{Clocks}$, *and*

(ii) *either* $(\mathcal{S}^k, \mathcal{S}^{k+1}) \in \mathcal{E}_\mathcal{G}$
or $\mathcal{S}^k = \mathcal{S}^{k+1}$.

Proposition 4.4 (Representing Symbolic State Graphs are Covering)
Let M be an Uppaal *model and $\mathcal{G}$ a symbolic state graph for M. If $\mathcal{G}$ represents M, then $\mathcal{G}$ covers $\mathcal{T}(M)$.*

We note that the covering property is in general a simplification of the behavior of the Uppaal model. There are well-formed sequences that can be covered but do not correspond to a behavior of the system.

Example 4.5 *Assume an* Uppaal *model M with one process, one location l, and two clocks x and y. Location l has the invariant $x \leq 2$. Let $e_\varnothing$ be the empty environment. Then the symbolic state graph $\mathcal{G} = (\{\perp_\mathcal{G}, \mathcal{S}\}, (\perp_\mathcal{G}, \mathcal{S}), \perp_\mathcal{G})$ with $\perp_\mathcal{G} = (l, e_\varnothing, \{[x \mapsto 0, y \mapsto 0]\})$ and $\mathcal{S} = (l, e_\varnothing, \{\nu \,\big|\, 0 \leq \nu(x) \leq 2 \wedge 0 \leq \nu(y) \leq 2\})$ represents M. However, the following well-formed sequence is covered by $\mathcal{G}$ without being a timed trace of M.*

$$(l, e_\varnothing, [x = 0,\ y = 0]),\ (l, e_\varnothing, [x = 1,\ y = 1]),\ (l, e_\varnothing, [x = 2,\ y = 1])$$

It is easy to see that for every Uppaal model M there exist symbolic state graphs that represent M. E.g., consider the (uncountably large) symbolic state graph where every Ξ_{Clocks} is a singleton, every node in $\mathcal{G}$ corresponds to one configuration of M, and ever edge corresponds to one legal step.

The important observation is that there exist *finite* symbolic state graphs. The region graph construction from [ACD93] is such a symbolic state graph that is apt for TCTL model checking. The number of nodes in this graph is exponential in the largest constant and in the number of clocks. For the logical properties we are interested in a smaller symbolic state graph suffices.

The representation of M by a symbolic state graph $\mathcal{G}$ is always *correct* in the sense that every timed trace of M is also covered. As Example 4.5 demonstrates, it is in general not *sound*: since also timed traces are covered that do not belong to M. Later we are going to construct the $\mathcal{G}$ with additional properties that imply soundness (Section 4.3).

First, however, we need to consider the finite representation of sets of clock evaluations with infinite cardinality. Necessarily only a regularly shaped subset of all possible sets can be encoded this way.

4.2 Data-Structures for Symbolic Real-Time

Real-time model checking requires that the uncountably many clock evaluations are represented in a symbolic form. A common way is to use clock constraints to partition clock evaluations into convex sets.

Difference bounded matrices (DBMs) are one important data structure that provides an efficient representation of convex sets of slop one. In this section we introduce DBMs and describe some of their important properties.

4.2.1 Regions and Zones

Assume we have a set of n real-valued clocks. Geometrically, every clock evaluation can be understood as a point in $\mathbb{R}^n_{\geq 0}$. It is clear that only a quotient of $\mathbb{R}^n_{\geq 0}$ can be represented for algorithmic use.

In [ACD93] an answer on how to build this quotient is given. The basis for this are two fundamental observations.

a. If a clock x or a clock difference $x - y$ exceeds the value of the largest constant it ever is compared with, then the actual value does not matter any more.

b. If the constants in the system are integers, then all clock evaluations that agree on the truth values on a finite set of predicates cannot be distinguished by means of the logic TCTL.

Definition 4.6 (Region[1]) *A* region *is a set of points in $\mathbb{R}^n_{\geq 0}$ that agrees in the all predicates of the form $x_i \bowtie \texttt{const}$ and $x_i - x_j \bowtie \texttt{const}$, $\bowtie \in \{\texttt{<}, \texttt{<=}, \texttt{==}, \texttt{>=}, \texttt{>}\}$. Here x_i and x_j are dimensions of $\mathbb{R}^n_{\geq 0}$, $i \neq j$, and $\texttt{const} \in \{-c, \ldots, 0 \ldots, c\}$. The parameter $c \in \mathbb{N}$ is called the* largest constant.

This definition gives rise to a finite set of regions, dependent on the largest constant c. Example regions are shown in Figure 4.1, left.

However, the number of regions is excessive. It can be approximated by the formula $(1+c)^n \cdot \frac{n!}{(log 2)^{n+1}}$, where c stands for the largest constant. However, it is not always necessary to distinguish all regions. For efficiency reasons we are therefore going to allow for larger sets, namely convex unions of regions (zones).

[1] See also Definition 7.8 in Section 7.4.

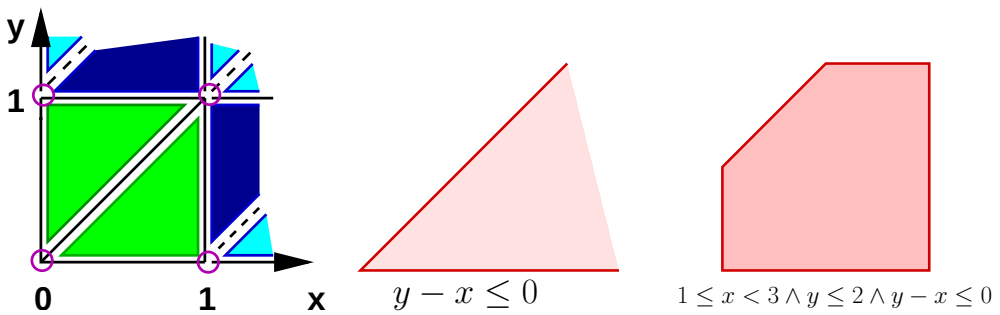


Figure 4.1: Regions and Zones in $\mathbb{R}^2_{\geq 0}$.

Definition 4.7 (Zone) *A* zone *is a convex union of n-dimensional regions. Here, convex means that if two points $a, b \in \mathbb{R}^n_{\geq 0}$ are in the zone, then every point on the line $\overline{ab}$ are also in the zone.*

We note that zones can represent infinite or finite areas of $\mathbb{R}^n_{\geq 0}$ (Figure 4.1, right). We require a number of operations on them to manipulate these areas in an algorithm.

4.2.2 Operations on Zones

Let $\mathcal{D}_c$ denote the set of zones for largest constant c and $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$ be the Boolean domain. In our context, the following operations are needed.

- **Intersection** $. \cap . : \mathcal{D}_c \times \mathcal{D}_c \to \mathcal{D}_c$
  This operation computes the set of points that is contained in both argument zones. By the convexity condition, the result is again a zone.

- **Reset** $r(.) : 2^{\{1,\ldots,n\}} \times \mathcal{D}_c \to \mathcal{D}_c$
  This operation "sets" some of the values to 0. Geometrically this corresponds to a projection of the zone into a lower dimensional space. E.g., $\{y\}(D)$ in $\mathbb{R}^2_{\geq 0}$ projects the area D down to the x-axis.

- **(possibly restricted) Delay** $.^{\uparrow .} : \mathcal{D}_c \times 2^{\bigcup L_i} \to \mathcal{D}_c$
  The delay operation is a specific need of our application in symbolic model checking. It can be understood as two steps. First all the upper bounds on constraints of the form $x_i \leq k$ or $x_i < k$ are removed in zone D to yield D'. I.e., whenever a point a is in D, then for every $\delta > 0$, $a + \delta \cdot (1, \ldots, 1)$ is in D'. Second, the invariants of the location vector $\vec{l} \in 2^{\bigcup L_i}$ are enforced on D'. Since invariants are conjunctions of the form $x_i \leq k$ or $x_i < k$, the result is again a zone D''. We use $D^{\uparrow \vec{l}}$ to denote this D''.

- **Emptiness Test** $. = \varnothing : \mathcal{D}_c \to \mathbb{B}$
  This test returns **true** whenever there is no point represented by the zone.

- **Inclusion Test** $. \subseteq . : \mathcal{D}_c \times \mathcal{D}_c \to \mathbb{B}$
  This test returns **true** whenever every point of the first zone is also in the second zone.

- **Equality Test** $. = . : \mathcal{D}_c \times \mathcal{D}_c \to \mathbb{B}$
  This test returns **true** if both arguments represent the same set of points.

This is not a minimal set of operations. E.g., the equality test can be reduced to two inclusion tests. It is, however, the set of operations we use in the symbolic model checking algorithm in the next Section and we list them here as an overview.

As it turns out, all these operations can be implemented efficiently by means of the following data structure.

4.2.3 Difference-Bounded Matrices (DBMs)

Difference bounded matrices are data-structure to represent n-dimensional convex geometric areas of slope one. They have been discovered and re-discovered in different contexts, we refer to constraint graphs in [Bel58] and square matrices of bounds in [Dil89].

Definition 4.8 (DBM) *A* difference bounded matrix *(or* DBM*) is a data-structure over n variables x_i that gives for the pair (i, j) with $1 \leq i, j \leq n$, $i \neq j$:*

- *an upper bound on the difference $x_i - x_j$ (from $\mathbb{Z} \cup \{\infty\}$) and*
- *an extra bit that indicates, whether this bound is strict.*

If the bound is ∞, the extra bit is not needed. DBMs can naturally represent convex regions with slope one, i.e., zones. The bounds on clocks of the form $x \bowtie$ `const`, $\bowtie \in \{$`<`, `<=`, `==`, `>=`, `>`$\}$, can be encoded via adding an dummy clock that stands always for value zero.

Canonicity. A given DBM is not necessarily tight in the sense that all bounds can be reached. For instance, if the bounds $x - y \leq 2$, $x \leq 1$, and $y \leq 1$ are given, then $x - y$ can reach at most the value 1. For a given DBM, all bounds can be tightened by an all-pair shortest path computation (e.g., with the Floyd-Warshall algorithm [CLR92] in $\mathcal{O}(n^3)$). The resulting DBM is a *canonical* representation of the zone, in [LLPY97] called the *shortest-path closure*. Two zones are identical if and only if their shortest-path closures are identical. Canonical forms allow for cheap equality checks.

Minimal Representation. An important alternative representation of DBMs is the one described in [LLPY97, Pet99]. Instead of storing *all* the constraints, the smallest system of constrains is memorized that *implies* the full DBM. It turns out that this representation is also canonical.

Algorithm: *symbolic_reach*
input: UPPAAL model: $M = \langle \vec{A}, Vars, Clocks, Channels, Type \rangle$
Local Property: φ

1 PASSED := {}; WAITING := $\{(\vec{l_0}, e_0, D_0^{\uparrow \vec{l_0}})\}$
2 REPEAT
3 $(\vec{l}, e, D) := pop(\text{WAITING})$
4 IF $\forall (\vec{l}, e, D') \in \text{PASSED}. D \not\subseteq D'$ THEN
5 PASSED := PASSED $\cup \{(\vec{l}, e, D)\}$
6 FORALL enabled $\vec{l} \xrightarrow{g,a,r} \vec{l'}$
7 $D' := r(D \cap D_g) \cap D_{Inv(\vec{l'})}$; $e' := a(e)$
8 IF $D' \neq \varnothing$ THEN
9 $push\big((\vec{l'}, e', D'^{\uparrow \vec{l'}}), \text{WAITING}\big)$

10 UNTIL $\exists (\vec{l}, e, D) \in \text{PASSED}. \exists \nu \in D. (\vec{l}, e, \nu) \models_{loc} \varphi$ /⋆ YES ⋆/
11 $\vee$ WAITING $= \varnothing$ /⋆ NO ⋆/

Figure 4.2: Symbolic Reachability Algorithm.

Convexity. One limitation of DBMs is that they always represent convex sets of points. Therefore the *union of DBMs* is problematic. DBMs are not closed under union, i.e., the result of this operation is not necessarily again representable as DBM (e.g., see Figure 4.3). If the definition of zones is relaxed to allow for non-convex unions of regions, disjunctions of DBMs are required to represent these sets.

Other Data Structures for Zones. There are other symbolic data structures that do not feature this problem. The possibly most notable ones are BDD-like data-structures [Bry86], where tests correspond to traversals of a directed acyclic graph. The interested reader is referred to the literature on Difference Decision Diagrams (DDDs) [MLAH99], Clock Difference Diagrams (CDDs) [LWYP99], Region Encoding Diagrams (REDs) [Wan00], and Clock Restriction Diagrams (CRDs) [Wan01].

4.3 Forward State-Space Exploration

Though the state-space of dense real-time interpretations is typically infinite, it can be be quotiented into a finite number of equivalence classes. A symbolic state graph (Section 4.1) can be used to keep track of all possible timed traces.

Symbolic forward reachability is an inexpensive algorithm that suffices to establish the important class of real-time safety properties. An extension of it can be used to verify unbounded response and thus builds the core of UPPAAL*'s model checking engine.*

4.3.1 Symbolic Forward Reachability

An algorithm for symbolic reachability is given in Figure 4.2.[2] It operates with *symbolic states* of the form $(\vec{l}, e, D)$, where D is a zone. Every $(\vec{l}, e, D)$ represents the set of configurations $\{(\vec{l}, e, \nu) \mid \nu \in D\}$.

The central data-structures are the lists WAITING (the symbolic states to be explored) and PASSED (the symbolic states that have already been explored). We use the operations *pop* and *push* on WAITING, where pop removes the first element (and returns it as result of the operation) and push inserts a new element at the end of the list.[3]

Notation. We use some shorthands here. D_g stands for the clock constraints of guard g, $D_{Inv(\vec{l}')}$ stand for the conjunction of clock constraints of the invariants of the locations in the location vector $\vec{l}'$. By our syntactic definition (Section 2.1.1) it is guaranteed that those clock constraints can indeed be represented by zones.

In line (6) we use the vector notation $\vec{l} \xrightarrow{g,a,r} \vec{l}'$ to indicate that this test covers both action steps and synchronized action steps. Up to two locations in $\vec{l}$ can be changed in one such step to reach $\vec{l}'$.

We note that the expression $\exists(\vec{l}, e, D) \in$ PASSED. $\exists \nu \in D.\ (\vec{l}, e, \nu) \models_{loc} \varphi$ can be evaluated effectively. First one builds the disjunctive normal form $\varphi \equiv \varphi^1 \vee \ldots \vee \varphi^m$, where φ^i do not contain any `or` connectors. For each φ^i a zone D_{φ^i} is constructed that represent the clock constraints in φ^i (if φ^i has no constraints on clocks, this is the full $\mathbb{R}^n_{\geq 0}$). Testing for the the remaining constraints in φ is then done for every φ^i, if $D \cap D_{\varphi^i} \neq \varnothing$.

To assert soundness, we need to observe that the successors are always constructed in a special way.

Lemma 4.9 (Soundness)
In the algorithm symbolic_reach all configurations $(\vec{l}, e, \nu)$ represented by symbolic states $(\vec{l}, e, \Xi_{Clocks})$ inserted in PASSED *are reachable in M.*

Proof: This property holds by induction. Observe that all symbolic states inserted in PASSED (line 5) have been in WAITING before. They are inserted in lines (1,9). In line 1, the initial configuration $(\vec{l_0}, e_0, D_0)\}$ and all the configurations reachable via delaying, $(\vec{l_0}, e_0, D_0^{\uparrow \vec{l_0}})$, are certainly reachable. The symbolic state $(\vec{l}', e', D')\}$ is constructed by (symbolically) taking a transition $\vec{l} \xrightarrow{g,a,r} \vec{l}'$, where every configuration represented by $(\vec{l}', e', D')\}$ has by induction a reachable predecessor. Thus also every configuration in $(\vec{l}', e', D'^{\uparrow \vec{l}'})$ is reachable. ///

[2] This formulation is for the sake of explanation; the implementation in UPPAAL applies some natural optimizations, e.g., if a symbolic state it is detected where φ holds, the search is aborted.

[3] This description corresponds to the implementation of WAITING as a queue, which yields a breadth-first search of the (symbolic) state space. Implementing it as a stack yields depth-first search.

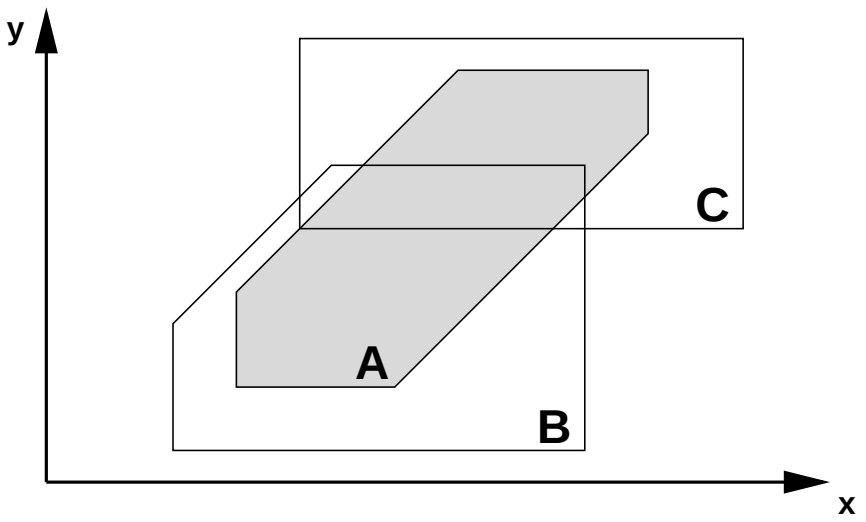


Figure 4.3: The Zone A is Subsumed by the Union of Zones B and C.

Proposition 4.10 (*symbolic_reach* is Sound and Correct)
The algorithm symbolic_reach terminates with the correct answer for the query "$M \models$ E<> φ".

Proof: The algorithm terminates, since there are only finitely many different symbolic states $(\vec{l}, e, D)$. In every execution of REPEAT the element inserted in PASSED is new, since the inclusion test (line 4) failed for every entry. Since there are only finitely many different potential elements in PASSED, there are only finally many executions of the REPEAT loop.

The algorithm can be understood as the construction of a symbolic state graph for M, where the nodes are stored in PASSED and the edge relation is not stored explicitly. Since for a node *all* possible steps are performed (action and synchronized action in line 6, delay $.^{\uparrow\vec{l}}$ in line 9), the conditions of Definition 4.2 are met. By Proposition 4.4 every timed trace in $\mathcal{T}(M)$ is covered. Thus if the answer is **"No"**, this is correct.

Soundness (answer **"Yes"**) holds by Lemma 4.9. □

4.3.2 Variations of the Inclusion Test

We note that some parts of *symbolic_reach* can be modified without losing soundness and correctness. In particular the inclusion test in line 4 allows for variations.

Observes that the elements in PASSED can represent the same configuration a multiple number of times. For reachability it suffices to retain from exploring the successors from a symbolic state $(\vec{l}, e, D)$, if every configuration it represents is already represented by some other symbolic state in PASSED. This makes it necessary to reason about the *union* of zones, as Figure 4.1 demonstrates: zone A is subsumed by $\mathsf{B} \cup \mathsf{C}$, but all three zones are pairwise incomparable.

In Figure 4.2, line (4) can be replaced by

4′ IF $D \not\subseteq \bigcup\{D' \mid (\vec{l}, e, D') \in \text{PASSED}\}$ THEN

However, this test is more expensive, since the DBM data structure used to implement zones does not support union.

Another variation is to replace the inclusion test by the (cheaper) equality test. In Figure 4.2, line (4) is replaced by

4″ IF $(\vec{l}, e, D) \notin \text{PASSED}$ THEN

We can still guarantee termination, since there are only finitely many different symbolic states. However, the size of PASSED in a complete state space exploration can grow considerably.

4.3.3 Liveness Checking

The liveness properties expressible in UPPAAL's logic are `A<>` φ, `E[]` φ, and φ `-->` ψ (unbounded response), where φ, ψ are local properties (see Section 2.3). We describe only response here (Figure 4.4), since the other two classes can be reduced to it.[4]

The validity of the local properties φ and ψ is central for this algorithm. Therefore all symbolic states $\mathcal{S}$ are split up such that each configuration represented by them is consistent in the evaluation of φ and ψ. We use the notation $\mathcal{S}^{[\xi]}$ to attach this information to the symbolic states. $\xi \in \{\neg\psi,\ \varphi \wedge \psi,\ \varphi \wedge \neg\psi,\ \neg\varphi \wedge \psi,\ \neg\varphi \wedge \neg\psi\}$.

In lines (4,5,15,16) we use the notation $D \cap D_\xi$ to restrict D to the part where ξ holds; this is a simplification, since in general this part needs to be represented by a *disjunction* of zones. We assume that the operations (*push*, *store*) are applied for every element of this disjunction.

The notation $\nearrow(\mathcal{S})$ is used to indicate that the zone of $\mathcal{S}$ is open to infinity. Intuitively this means that some configurations represented by $\mathcal{S}$ can engage in an infinite, unbounded number of delay steps. By convexity, in fact all configurations represented by $\mathcal{S}$ then have this property.

The notation $\downarrow(\mathcal{S})$ is used to indicate that for some of the configurations represented by $\mathcal{S}$, there is no further step enabled, i.e., they are deadlocked.

The strategy of the algorithm is as follows. First, construct a symbolic state graph $\mathcal{G} = \big(\text{PASSED}, \rightrightarrows, (\vec{l_0}, e_0, D_0)\big)$. This is done such that the configurations represented by a symbolic state agree on the validity of φ and ψ. It is easy to see that $\mathcal{G}$ represents M.

Assume there are reachable configurations, such that φ holds. There are precisely three cases such that φ `-->` ψ does not hold:

1. there exists an infinite timed trace with infinitely many action/synchronized action steps, such that after reaching φ, ψ never holds,
2. there exists an infinite timed trace with finitely many action/synchronized action steps, such that after reaching φ, ψ never holds, or

[4] `A<>` φ is equivalent to $\bigwedge_{1 \le i \le n} A_i.l_i^0 \ \wedge\ \bigwedge_{x \in Clocks} x \le 0$ `-->` φ. `E[]` φ is true if and only if `A<>` $\neg\varphi$ is false.

Algorithm: *symbolic_response*

input: UPPAAL model : $M = \langle \vec{A}, Vars, Clocks, Channels, Type \rangle$
Query : φ --> ψ

1 PASSED := {}; WAITING := {}; *relation* $\Rightarrow$:= {}
2 FOREACH $\xi \in \{\varphi \wedge \psi,\ \varphi \wedge \neg\psi,\ \neg\varphi \wedge \psi,\ \neg\varphi \wedge \neg\psi\}$
3 IF $\exists \nu \in D_0^{\uparrow \vec{l_0}}.\ (\vec{l_0}, e_0, \nu) \models_{loc} \xi$ THEN
4 *store* $(\vec{l_0}, e_0, D_0) \Rightarrow (\vec{l_0}, e_0, D_0^{\uparrow \vec{l_0}} \cap D_\xi)^{[\xi]}$
5 *push*$\big((\vec{l_0}, e_0, D_0^{\uparrow \vec{l_0}} \cap D_\xi)^{[\xi]}$, WAITING$\big)$
6 REPEAT
7 $(\vec{l}, e, D)^{[\chi]} := pop$(WAITING)
8 IF $(\vec{l}, e, D) \notin$ PASSED THEN
9 PASSED := PASSED $\cup \{(\vec{l}, e, D)\}$
10 FORALL enabled $\vec{l} \xrightarrow{g,a,r} \vec{l'}$
11 $D' := r(D \cap D_g) \cap D_{Inv(\vec{l'})}$; $e' := a(e)$
12 $D'' := D'^{\uparrow \vec{l'}}$
13 FOREACH $\xi \in \{\varphi \wedge \psi,\ \varphi \wedge \neg\psi,\ \neg\varphi \wedge \psi,\ \neg\varphi \wedge \neg\psi\}$
14 IF $\exists \nu \in D''.\ (\vec{l'}, e', \nu) \models_{loc} \xi$ THEN
15 *store* $(\vec{l}, e, D)^{[\chi]} \Rightarrow (\vec{l'}, e', D'' \cap D_\xi)^{[\xi]}$
16 *push*$\big((\vec{l'}, e', D'' \cap D_\xi)^{[\xi]}$, WAITING$\big)$
17 UNTIL WAITING $= \varnothing$
18 FORALL $\mathcal{S}_1^{[\varphi \wedge \neg\psi]} \in$ PASSED
19 IF $\exists$ a loop $(\mathcal{S}_1^{[\neg\psi]} \Rightarrow \mathcal{S}_2^{[\neg\psi]} \Rightarrow \cdots \Rightarrow \mathcal{S}_1^{[\neg\psi]})$
20 or $\exists\big(\mathcal{S}_1^{[\neg\psi]} \Rightarrow \mathcal{S}_2^{[\neg\psi]} \Rightarrow \cdots \Rightarrow \mathcal{S}_n^{[\neg\psi]}\big)$ with $\nearrow(\mathcal{S}_n)$
21 or $\exists\big(\mathcal{S}_1^{[\neg\psi]} \Rightarrow \mathcal{S}_2^{[\neg\psi]} \Rightarrow \cdots \Rightarrow \mathcal{S}_n^{[\neg\psi]}\big)$ with $\downarrow(\mathcal{S}_n)$
22 THEN RETURN **No**
23 RETURN **Yes**

Figure 4.4: Symbolic Response Checking Algorithm.

3. there exists a maximally extended finite trace, such that after reaching φ, ψ never holds.

It is clear by Proposition 4.4 that all these counterexample traces are covered by $\mathcal{G}$. However, we need a special property of $\Rightarrow$ to assert that they are also discovered by the algorithm.

Lemma 4.11 ($\rightrightarrows$ is a Predecessor Relation)
In algorithm symbolic_response, for $(\vec{l}^{\,1}, e^1, D^1) \rightrightarrows (\vec{l}^{\,2}, e^2, D^2)$ *the following holds. For all* $\nu^2 \in D^2$ *then there exists a* $\nu^1 \in D^1$ *such that*

either $(\vec{l}^{\,1}, e^1, \nu^1) \stackrel{\alpha}{\Longrightarrow} (\vec{l}\,, e, \nu)$

or $(\vec{l}^{\,1}, e^1, \nu^1) \stackrel{\alpha}{\Longrightarrow} (\vec{l}\,, e, \nu^2) \stackrel{d}{\Longrightarrow} (\vec{l}\,, e, \nu), \quad \alpha \in \{a, \tau\}.$

Proof: By construction only those configurations are represented via $(\vec{l'}, e', D'' \cap D_\xi)^{[\xi]}$ (line 15) that are can be reached from *some* configuration represented by $(\vec{l}, e, D)^{[\chi]}$ via action/synchronized action step (line 10) and possibly a subsequent delay (line 12).

////

An immediate consequence of Lemma 4.11 is that all configurations represented by entries in PASSED are reachable.

Proposition 4.12 (*symbolic_response* Sound and Correct)
The algorithm symbolic_response in Figure 4.4 terminates with the correct answer for the query "$M \models \varphi$ `-->` ψ*".*

Proof: We noted before that the constructed symbolic state graph $\mathcal{G} = \big($PASSED, $\rightrightarrows, (\vec{l_0}, e_0, D_0)\big)$ represents M and thus covers $\mathcal{T}(M)$.

For soundness and correctness it suffices to assert that lines 19-21 identify all traces $\sigma \in \mathcal{T}(M)$ that are counterexamples for φ `-->` ψ.

By Lemma 4.11, every path $(\,(\vec{l_0}, e_0, D_0) \rightrightarrows \cdots \rightrightarrows \mathcal{S}_1^{[\varphi \wedge \neg\psi]} \rightrightarrows \mathcal{S}_2^{[\neg\psi]} \rightrightarrows \cdots \rightrightarrows \mathcal{S}_1^{[\neg\psi]})$ respectively $(\,(\vec{l_0}, e_0, D_0) \rightrightarrows \cdots \rightrightarrows \mathcal{S}_1^{[\neg\psi]} \rightrightarrows \mathcal{S}_2^{[\neg\psi]} \rightrightarrows \cdots \rightrightarrows \mathcal{S}_n^{[\neg\psi]})$ indeed corresponds to a counterexample trace. Due to the covering property and the allowing for maximal delay in every symbolic state, all counter-example traces are of one of these three shapes.

□

4.4 Reflection: Symbolic Analysis of Real-Time Systems

Due to fundamental (un-)decidability results, modeling languages for real-time model checking are typically restricted to a decidable fragment. Symbolic techniques allow for the treatment of infinite (even uncountable) state spaces. Still, algorithmic analysis suffers from the high computational complexity. In practice this requires specialized and well-engineered implementations.

The algorithms we described in Section 4.3 are only conceptual. Their actual implementation in UPPAAL applies various optimization steps, e.g., for the unbounded response the symbolic states are not split according to the local properties φ, ψ.

Though the supported logic is rather limited, the classes up to unbounded response seem—from a specification point of view—to be the most relevant ones. This justifies the development of specialized algorithms that do not depend on the capability of treating the full TCTL.

It has to be noted that this does not eliminate the need for modeling expertise. Building models with surplus clocks, surplus states, and excessive amount of nondeterminism is easy, and they are typically out of scope for algorithmic verification. In courses taught using UPPAAL, the above seem to be typical beginner mistakes.

The need for abstraction and approximation is reflected in a number of automated techniques that take this burden from the user to the algorithm. The overall efficiency of an algorithm depends on many factors. With various options, datastructures, and strategies available in a tool, it is often difficult (or even impossible) to find the best choices. For UPPAAL we treat this phenomenon in more detail in the following Chapter.

Chapter 5

Efficiency in Real-Time Model Checking

It's exponential all over the place.

— Amir Pnueli, at FMCAD'98

The algorithmic verification of real-time systems suffers from the high computational complexity of the problem. Model checking the full class of TCTL properties has been shown to be PSPACE-complete in [ACD93]. Surprisingly, the complexity does not depend on the alternation of quantifiers in TCTL formulas. Timed reachability is already PSPACE-complete [CY91]. The necessity for discrete data parts contributes potentially further to excessive run-times.

The consequence is twofold. First, it is perceivable that the models indented for algorithmic verification need to reside on a certain level of abstraction. *Details that are irrelevant to the correctness of the system have to be omitted. System parts that apparently do not influence the timing behavior should be captured in an inexpensive way that reflects only the essence of their nature. In practice this means that a verifiable model cannot be constructed without understanding of the algorithms that are used to verify it.*

Second, verification algorithms have to be tuned to be efficient in many cases. Often this means that specialized routines exploit structure and regularities of the model and avoid expensive computations whenever possible. The worst-case complexity dictates that this cannot succeed for all input models. Nevertheless, a number of techniques have been developed during the last decade that perform very well in practice.

Some of these techniques have been implemented in the Uppaal *model checking tool and are available either isolated or in combination. For the command-line version,* `verifyta`*, they correspond to switches that can be set for individual runs. It is hard to predict which combination yields the lowest consumption of time or memory.*

In this Chapter we give an overview on the options as present in `verifyta` *version 3.1.64 (September 2001). The intention is to detect beneficial combinations. For this we consider all combinations of five most influential options in terms of time and memory consumption. We use three classes of scalable benchmark examples to give experimental data.*

5.1 Optimizations for Real-Time Model Checking

Optimizations aim to render time- or space-consuming steps in the model checking algorithms more efficient, while not altering the result. In some cases, in particular with small examples, the application of optimizations can yield longer run-times or higher memory consumption due to some algorithmic overhead. We briefly review the techniques that are currently implemented in the Uppaal *tool.*

5.1.1 Active Clock Reduction (`-a`)

Active clock reduction was introduced in [DY96] and has been extended to networks of timed automata in [DT98]. It builds on the observation that in certain control situations the value of some clocks do not influence future behavior. A sufficient condition is that a clock is *always* reset to 0 before its value appears in a guard, invariant, or part of a local property to be checked.

A pre-computation phase determines for each location vector a set of in-active clocks. In the reachability analysis the DBMs are only built over the active clocks. This does not hurt the inclusion check $D \subseteq D'$ (line 4 in Figure 4.2 on page 83), since it requires identical discrete parts of the configurations $(\vec{l}, D)$ and $(\vec{l}, D')$.

By default, this option is disabled.

5.1.2 Compact DBM Representation (OFF with `-C`)

Zones of clock evaluations can be represented by difference bounded matrices (DBMs). The straightforward representation of a DBM for n clocks stores $(n+1)^2 - n$ integer values that represent the upper bounds on the differences of every pair of clocks, including one additional zero clock.

Some of these clock constraints can be redundant, i.e., are implied by other constraints. In the compact DBM representation, originally introduced as *local reduction* [LLPY97], only a minimal equivalent set of constraints is stored. This minimal set is computed with a $\mathcal{O}(n^3)$ algorithm.

The experiments in [LLPY97] yield space savings between 68 and 85%. In nine out of ten cases the optimized algorithm also runs faster, time savings are between -8 and 78%. This explains by a cheaper inclusion check that has fewer values to consider when browsing the Passed list.

By default the compact DBM representation is used, the switch `-C` disables it explicitly. The `-C` option is incompatible with convex hull approximation. If `-A`

is set, this compact DBM representation is automatically set; a possibly set `-C` is ignored.

5.1.3 Space Usage Reduction by Smaller "Passed" List (`-S 1|2`)

The memory consumption of UPPAAL depends primarily on the size of the PASSED list. As it has been observed in [LLPY97, Pet99], not all encountered symbolic states have to be stored in order to guarantee sound termination of the state-space exploration. E.g., whenever control reaches committed locations, these configurations have to be left immediately. Thus no configuration can contain committed locations after a time delay. By default (option `-S 1`) no configurations containing committed locations are stored in PASSED.

The number of stored symbolic states can be reduced even further without sacrificing sound termination. It suffices to store certain prominent control structure elements in the PASSED, namely the entry nodes of "dynamic loops". In [LLPY97] this is introduced as *global reduction*. The price for this is a potential increase in run-time, for the same symbolic states are possibly explored more than once. This reduction is chosen with `-S 2`.

The setting `-S 1` is the default, option `-S 0` disables the space reduction.

5.2 Approximation Techniques for Real-Time Systems

Approximation translates the original problem into a computationally cheaper problem. In case of a successful run, the property established for the translation also holds for the original problem. We briefly describe the approximation techniques implemented in UPPAAL*, i.e., convex hull and bitstate hashing.*

5.2.1 Convex Hull Over-Approximation (`-A`)

The convex hull technique yields a conservative over-approximation of the reachable state-space and has first been applied by Howard Wong-Toi [WT94].

In the symbolic state space exploration, one control configuration $\vec{l}$ can be associated with a large number of clock zones. Since the union of zones is not necessarily a zone again, many of them have to be stored separately.

The approximation idea now is to subsume *all* zones associated with one $\vec{l}$ under the smallest surrounding zone. This surrounding zone can contain clock evaluations that are not reachable in the original system. In forward reachability analysis, this yields an over-approximation of the reachable state-space. Thus, if a safety property (invariant) can be established in the over-approximation, it also holds in the original system.

If the safety property does not hold in the over-approximation, UPPAAL's model checking engine reports that the property is `MAYBE satisfied`. Though this technique is rather coarse, in practice it surprisingly often suffices to establish safety

properties. The granularity could be refined by allowing only the subsumption of zones that are "close" according to some heuristic.

Convex hull approximation is incompatible with bitstate hashing and enforces compact data structures (i.e., absence of `-C`). If both `-A` and `-C` are set, the latter one is ignored.

By default, the convex hull approximation is disabled.

5.2.2 Under-Approximation: Bitstate Hashing (`-Z`)

This under-approximation technique was introduced by Holzmann [Hol98] in context of the LTL model checker SPIN [Hol91, Hol97].[1] Instead of the full state-space only a hashed version of it is stored in the PASSED list, i.e., one bit per encountered symbolic state.

With the application of bitstate hashing it is possible to establish the reachability of a state, but not to refute it in general. If the hash values of two encountered states collide, the one encountered later is not explored any further, since there is already a matching entry for it in PASSED. This implies that potentially not the whole state space is expanded, but only a part of it.

The percentage of symbolic states reached by this under-approximation is known as *coverage*. There are refinements of bitstate hashing, most notably double hashing [Hol98] that yield a high coverage for economical cost.

In UPPAAL a single hash of the complete symbolic state is used for bitstate hashing. This implies that the subsumption check is in fact an equality check for the occurrence of the same the same zone for matching discrete parts. This does not destroy termination or soundness (see Section 4.3.2), but means that before hashing a potentially much larger number of zones have to be explored than for the exact analysis.

For most real-time systems, the key to efficient model checking is the specific treatment of the symbolic part. Therefor this technique got less attention here than it is in LTL model checking.

Bitstate hashing is incompatible with the convex hull over-approximation. By default, bitstate hashing is disabled.

5.2.3 Other Approximation Techniques

There is a large number of techniques that can be classified as approximations and we are not in a position to cover them here. Approximations are expecially useful, if (a) the computational complexity of a problem is high and (b) many details of the problem can be safely ignored. In model checking, all details of a system boil down to a yes/no answer. The "intuitive complexity" of the state-space is often much lower than the worst case bound that complexity theoretic results yield [BCM+90].

For real-time systems this has been studied, e.g., in [WT94, HNSY94, TAKB96, DT98]. Two novel approximation techniques are subject of Chapter 7 and Chapter 6.

[1] See also the SPIN homepage at http://netlib.bell-labs.com/netlib/spin/whatispin.html.

5.3 Other Options of the UPPAAL Engine

There are some additional options of the UPPAAL *engine that are neither optimizations nor approximations. We briefly describe them in the following.*

5.3.1 Depth-First Search (`-d`)

Though the state space of a system is determined by the definition, the number of symbolic states that represent it can depend on the search order.

In breadth-first search, the enumeration of the symbolic state space proceeds in layers around the initial state. First all symbolic states reachable in one (symbolic) step are explored, then all symbolic states reachable in two (symbolic) steps, and so on. In depth-first search, the last symbolic state encountered is also the next one to be explored.

The default is to use breadth-first search. Intuitively, this way the zones have a better chance to remain as large as possible.

5.3.2 Disable Deadlock Checker (`-W`)

The deadlock checker issues a warning, if an encountered (symbolic) state does not have a successor for some clock evaluation in the associated zone. This includes the case, where all possible steps enabled in the future are delay steps (unbounded delay).

The deadlock check was activated by default until version 3.0.39 (March 2000) and de-activated in later versions.

5.3.3 Display Warnings as Queries (`-Q`)

If the deadlock checker detects a deadlock situation, a warning is issued. If this option is set, this warning contains a logical query. For example, if a deadlock occurs whenever process P is in location T and the clock x exceeds value 1, the query might state `E<> ( P.T and x>1 )`. This query can later be used to generate a trace leading to the deadlock.

We note that the semantics of deadlocks changed from version 3.2.1 on (29 Oct 2001).

By default, this option is not set.

5.3.4 Change Size of Hash Table in "Passed" List (`-H size`)

The PASSED list is implemented as a hash table of fixed size. Collisions are resolved via re-hashing. The size of the hash-table determines the minimal amount of allocated memory. If the number of encountered symbolic states is significantly larger than the hash-table size, hash-table lookups become expectedly more expensive by a factor proportional to the quotient encountered states/hash table size.

The default size of the hash table is 17˙609.

5.3.5 Optimize Time Consumptions when Several Properties are Examined (-T)

With this option, it is possible to verify several properties in the same model checking run in parallel instead of one after the other. However, this makes it necessary to store extra information during the computation of the next step.

This option should be switched off if only one property is verified. Is is disabled by default.

5.3.6 Unpack Reduced Constraint System Before Inclusion Check (-U)

If this option is enabled, the inclusion check (line 4 in Figure 4.2) is performed in an alternative way: The DBM of the currently explored symbolic state is not transformed to the minimal corresponding constraint system. Instead, the constraint system of the PASSED list entry is expanded to the full DBM before it is compared to the current symbolic state.

This option is disabled by default and only possible, if `-C` is not set.

5.3.7 Do Not Display Copyright Message (-q)

At the beginning of a model checking run, `verifyta` prints a copyright message to the standard output. This is suppressed, if the `-q` option is set.

5.3.8 Run Silently Without Progress Indicator (-s)

If `verifyta` is called by command line, a rotating ASCII bar is displayed to visualize progress of the model checking run.

This is suppressed, if this option is set.

5.3.9 Print Diagnostic Trace to Standard Output (-t)

Successful verification of existential properties (`E<>` and `E[]`) yields traces that can be understood as witnesses for the truth-hood of the formula. In case of a `E<>`φ property, the trace leads to a configurations satisfying φ. In case of a `E[]`φ property, φ will hold in every configuration and the trace will terminate in a loop, guaranteeing that φ potentially always holds.

Dually, for universal properties (`A[]`, `A<>`, and response), traces can serve as counter-examples that witness the false-hood of the property at hand.

This option yields the generation of such witness traces whenever appropriate. By default, this is disabled and merely **true** or **false** is reported.

5.3.10 Display Traces Symbolically (-y)

Counter-example traces are detected on symbolic runs, i.e., every symbolic state corresponds to a—in general uncountable—set of concrete states, since the associ-

ated zone encodes a set of clock evaluations. Analogously, every step in the trace can correspond to an uncountable set of steps that are treated uniformly.

When displaying a counter-example trace, by default these symbolic states and steps are instantiated with concrete clock values. If this option is set, however, the counter-example traces are displayed with symbolic states and symbolic delay steps.

This option can only be enabled in combination with `-t`.

5.4 Run-Time Experiments with UPPAAL

As it is the case for many tools, UPPAAL *is equipped with a number of verification options. Typically one has to run experiments in order to find out, which combination of options works well. Time and space consumption are often tradeoffs.*

In this section we aim at contrasting different options and their combination. We use Fischer's protocol, the CSMA/CD protocol, and a FDDI token ring protocol as scalable benchmark examples.

5.4.1 Why Run-Time Comparisons are Problematic

In the search for efficient verification algorithms it is necessary to *compare* the performance of different approaches with each other. For real-time reachability all known algorithms are worst-case exponential in the number of reachable control situations, due to the *PSPACE*-hardness of the problem. Therefore experimental data is used for the comparison of algorithms. This is problematic for several reasons.

(a) Run-times are machine-dependent. Not only on processor speed matters, but also main memory, operating system, context switch delay, system load, and often also on the compiler used.

Comparisons between tools should always use the same machine and configuration.

(b) Performance is heavily dependent on the choice of optimization options and sometimes subtleties like the order in which processes are declared.

(c) Performance is heavily dependent on the sample problem.

(d) The implementation of the sample problems in various tools is often not "identical", despite best intentions. Some constructs in one modeling language might not have an equivalent counterpart in the other one—e.g., UPPAAL is the only tool that allows for committed locations.

What can be considered a meaningful comparison is "best options that a tool has to offer". Arguably it matters most, which size of examples you can treat before your machinery breaks down. As yet, there are no standard benchmark suites for real-time model checking tools. One first step towards comparable run-time data is making

the input files electronically available. We do this for all run-time experiments in this Chapter at http://www.docs.uu.se/docs/rtmv/uppaal/benchmarks/.

We use only UPPAAL for our benchmarks and focus on the effects of different optimization options. We use `verifyta` version 3.1.64. It executes on Spark 450 MHz with 4 GB main memory under Solaris OS. Time and space consumption for each run were measured simultaneously with the aid of Johan Bengtsson's `memtime` utility, version 1.2.[2]

5.4.2 How to Read the Run-Time Charts (Figures 5.5–5.8)

We compiled experimental data for a number of runs of one input problem in a run-time chart as follows. For every input problem, all 36 combinations of the `verifyta` command line options `A` (yes/no), `C` (yes/no), `S` (0/1/2), `a` (yes/no), and `d` (yes/no) are used, see Figure 5.1—the options `C` and `A` cannot be set at the same time. Every run is labeled with the corresponding combination of these letters—read `S0` and `S2` as `-S 0` and `-S 2` (`-S 1` is the default and thus omitted). The empty label stands for `A`, `C`, `a`, `d` "no" and `S` "1". In every run the command line options `-s` (run silently, do not display the progress indicator) and `-W` (disable deadlock checker) were also used and are not listed explicitly. For all runs in this Chapter, the convex hull over-approximation (`A`) turns out to be sufficient to infer the respective safety properties.

`-A`	use convex-hull approximation
`-C`	disable compact data structures
`-S`	optimize space consumption
`0|1|2`	0: none, 1: default, 2: most
`-a`	detect active/inactive clocks
`-d`	use depth-first search

Figure 5.1: Command Line Options Used.

One run corresponds to one plotted point (center of a circle) in the run-time chart. The runs are sorted according to run-time (x-axis, in seconds). The y-axis records the allocated memory in Megabytes. A bold line connects the points for better readability. Runs exceeding 4 hours time or 800 MB memory usage were aborted; this is indicated by labels of the form "`(switches)*`".

[2] Available at http://www.update.uu.se/%7Ejohanb/memtime.

5.4.3 Fischer's Mutual Exclusion Protocol

One of the more popular benchmarks for real-time is Fischer's protocol for mutual exclusion. We note that this example is *not* very revealing, mainly due to the fact that in every modeling language the algorithm is implemented slightly differently.

For example, Lamport describes one process of Fischer's protocol in [Lam87] with a shared variable `x` and the following pseudo-code.

```
repeat
       await x = 0 ;
       x := i ;
       delay
until  x = i ;
critical section;
x := 0
```

In [AL92] Fischer's protocol is quoted in a similar—but not identical—fashion.

```
a: await x = 0;
b: x := i;
c: await x = i;
cs : critical section
```

An upper bound on the delay between `a` and `b` and a lower bound on the delay between `b` and `c` is required. There are many combinations, such that mutex holds.

We note that the differences are minor, but of potential relevance. The "delay" is in both cases a constrained but not clearly specified value. For some implementations (e.g, DBMs), the concrete value does not matter, while for others (e.g., BDD based ones) it does. Thus Fischer's protocol stands in the literature for a whole class of algorithms and run-time comparisons of tools using this example have to be regarded with care.

For our benchmarks we use a concretization of the second formalism. The system consists of N symmetric processes and one shared variable `id`. Every process has one clock $\mathtt{x}_i$ to measure the delay. The execution of the assignment `id := i` may take up to 2 time units. The critical section is entered, if after a delay of *more* than 2 time units `id = i` holds. We verify the (true) safety property that no pair of processes is in the critical section at the same point in time. The run-time charts for $N = 2, \ldots, 9$ are compiled in Figure 5.5.

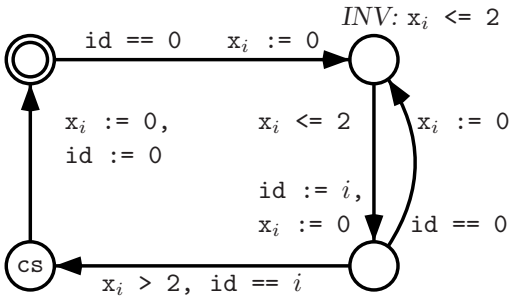


Figure 5.2: Fischer's Mutex: Process i.

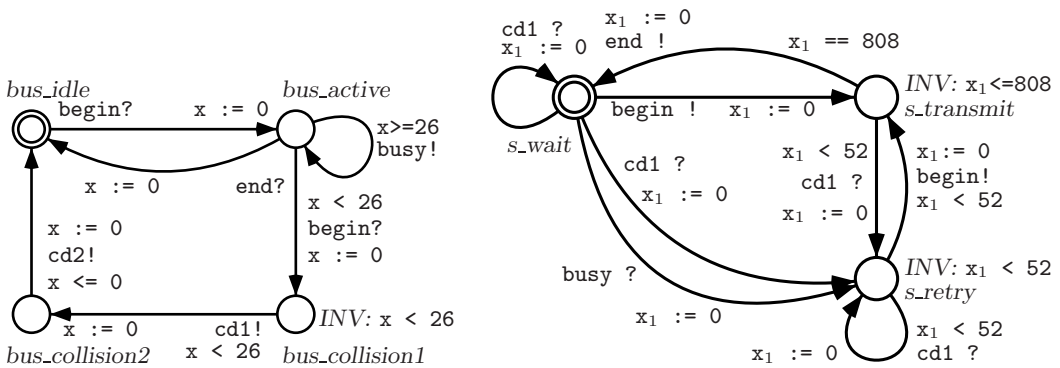


Figure 5.3: CSMA/CD Protocol: Bus and Sender #1 for $N = 2$.

5.4.4 CSMA/CD Protocol

In a broadcast network with a multi-access bus, the problem of assigning the bus to only one of many competing stations arises. The CSMA/CD protocol (Carrier Sense, Multiple-Access with Collision Detection) describes one solution. Roughly, whenever a station has data to send, if first listens to the bus. If the bus is idle (i.e., no other station is transmitting), the station begins to send a message. If it detects a busy bus in this process, it waits a random amount of time and then repeats the operation. A detailed description of the (simplified) model we use is found, e.g., in [BDM+98].

In our model, there is one process for the ring (with one local clock x) and one process for every of the N stations (with one local clock x_i), see Figure 5.3. For our experiments we use the (true) safety property stating that station 1 and station 2 are at longest for 52 time units simultaneously in transmission mode. The run-time charts for $N = 2 - 9$ are compiled in Figure 5.6.

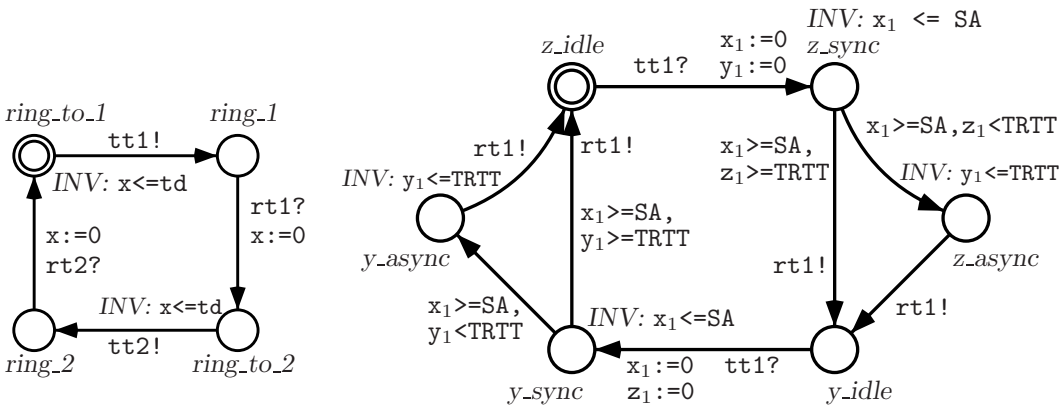


Figure 5.4: FDDI Token Ring Protocol: Ring and Station #1 for $N = 2$.

5.4.5 FDDI Token Ring Protocol

The FDDI (Fiber Distributed Data Interface) is a fiber-optic token ring local area network, e.g., described in [Jai94, DT98]. FDDI networks are composed from N symmetric stations that are organized in a ring.

We use a simplified model of N-ary networks for our benchmarks. One process models the ring that hands the token in one direction to N symmetric processes that may hand back the token in a synchronous (high-speed) or asynchronous (low priority) fashion. The ring process owns a local clock and every station owns three local clocks (Figure 5.4). The timing constants for this model are 0, 20, and $50*N+20$, i.e., no delay in the token passing of the ring (`td`), 20 time units for high-speed communication (`SA`), and maximally $50*N+20$ time for the asynchronous token passing (`TRTT`).

We verify a (true) safety property stating that the token is not at two places at the same time. The run-time charts for $N = 2 - 9$ are compiled in Figure 5.7.

5.5 Reflection: Optimization Techniques for Real-Time Systems

Figure 5.8 lists the switch setting sorted according to the *sum* of time (respectively memory) over all runs. We note that the behavior for every single switch is heavily influenced by presence or absence of other options.

This experimental data comes with two surprises:

1. There are large gaps between best and worst options.
2. Instead of being wildly distributed, the options form "clusters" of nearly identical run-time behavior.

These clusters are not stable—they are present for one input problem but change in constellation over different inputs. The one option that consistently yields improved performance (both time- and memory-wise) is the convex hull over-approximation (`A`). In our benchmark examples, the approximation suffices to establish validity of the safety properties. In the absence of `-A`, active clock reduction (`-a`) is in our experiments always an advantage.

Other than expected, the option `-S 2` occasionally yields a larger memory consumption than the weaker optimization `-S 1`. This can be explained by the need for (temporary) memory allocation, when parts of the state-space are explored repeatedly.

Also other then sometimes predicted depth first search performs sometimes measurably better than breadth-first search. E.g., in Token Ring FDDI `dC` is faster than `C` for $N \geq 4$. In the same examples with `d` vs. the empty switch setting, additionally the memory consumption of the first is lower.

We also note that the scalable benchmark examples do not seem behave "uniform", in particular Fischer's mutex protocol surprises with irregularities when

scaled up. Note that the clusters of switch settings that give a nearly identical run-time performance *change* in Figure 5.5, when the problem is scaled up.

The somewhat bitter conclusion of this run-time comparisons is that it is hard to predict what will work best. We recommend to try first some of the best option settings we found, since they behaved well on all three benchmark classes. However, apart from convex hull approximation, there is no clear pattern indicating what options to use. Neither can certain combinations be regarded as inefficient a priori. It remains a burden to the user to find good choices here.

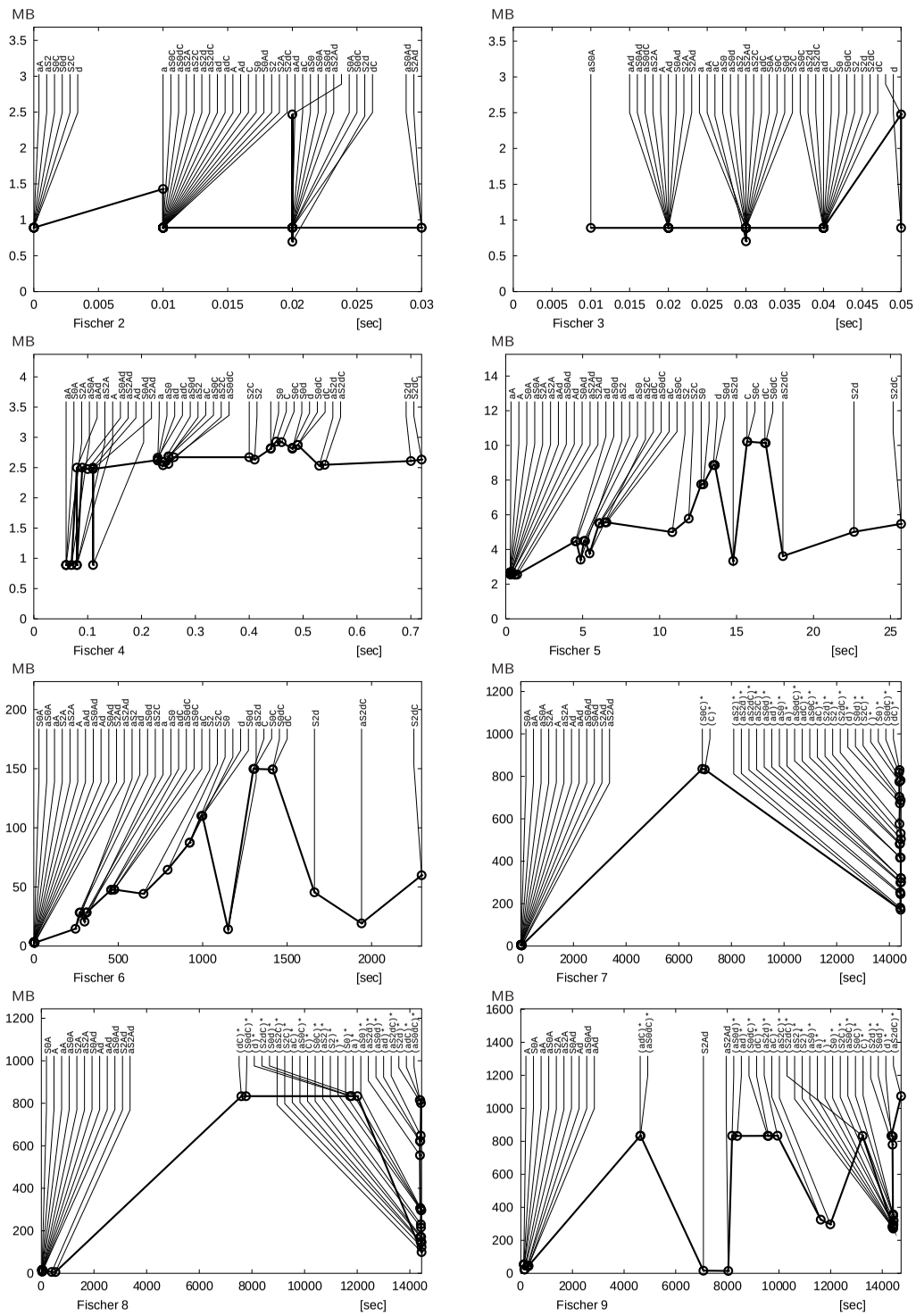


Figure 5.5: Run-Time Charts for Fischer's Mutex, $N = 2, \ldots, 9$.

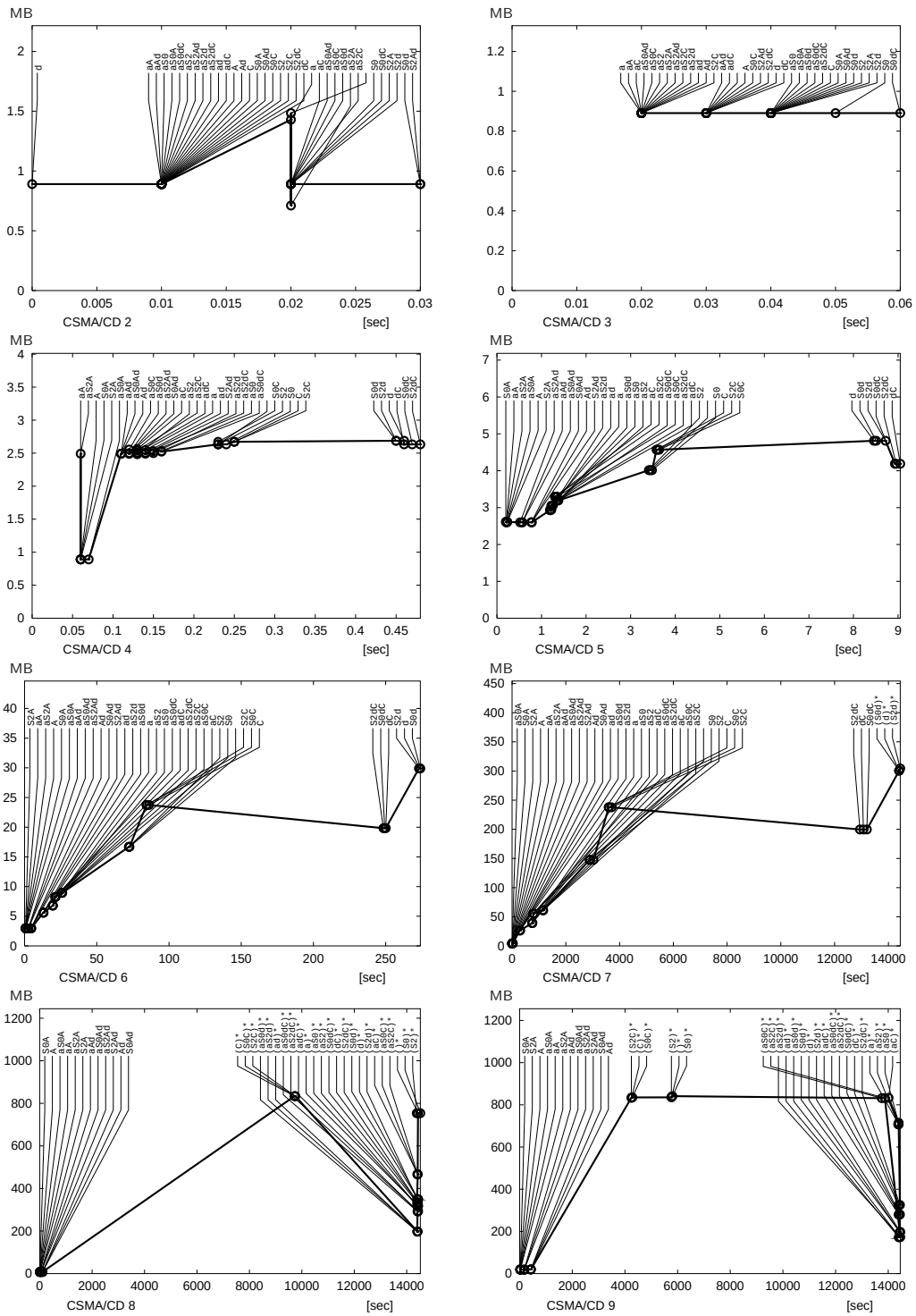


Figure 5.6: Run-Time Charts for CSMA/CD, $N = 2, \ldots, 9$.

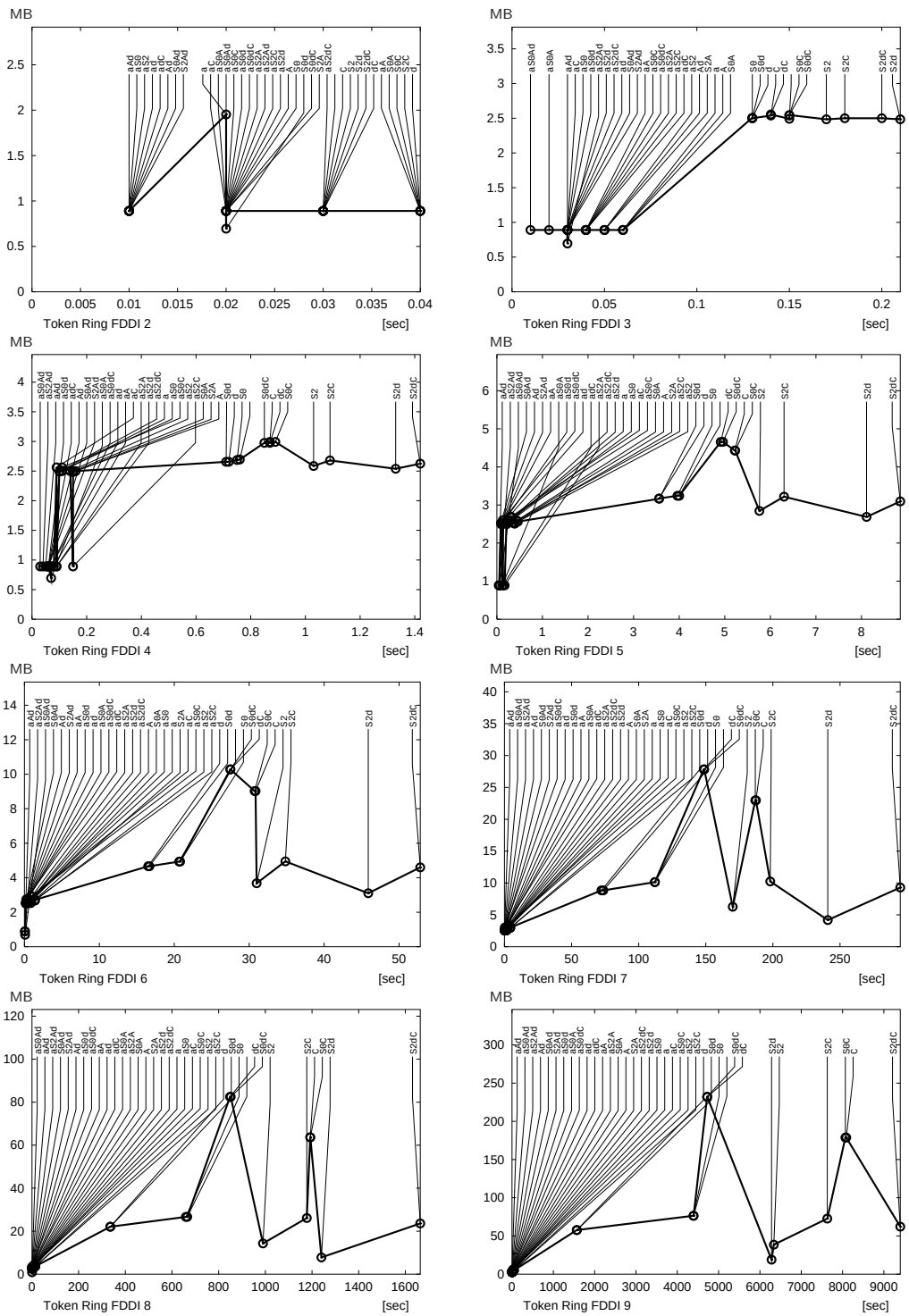


Figure 5.7: Run-Time Charts for Token Ring FDDI, $N = 2, \ldots, 9$.

options	Σ sec	Σ MB	#abort
S0A	195.28	142.49	0
A	196.47	142.49	0
aA	207.56	139.80	0
aS0A	207.70	138.05	0
S2A	225.28	101.05	0
aS2A	237.05	98.30	0
aAd	598.72	125.95	0
aS0Ad	598.95	124.05	0
S0Ad	879.53	129.25	0
Ad	881.96	130.84	0
S2Ad	8070.87	93.05	0
aS2Ad	8860.21	86.01	0
adC	63526.28	2780.84	5
aS0dC	63571.98	2786.47	5
S0C	64260.64	3752.76	5
C	64287.07	3750.15	5
aS0d	66415.87	2218.74	5
ad	66610.91	2217.88	5
aS2d	69106.56	2017.67	5
aC	70575.83	2414.62	5
aS2C	70694.50	2075.55	5
S2C	70905.50	3282.80	5
	72544.45	3363.05	5
S0	72701.28	3367.95	5
aS0C	73133.52	2435.20	5
aS2	73157.34	1781.25	5
aS0	73159.30	2055.19	5
a	73194.60	2048.08	5
S2	74687.03	2926.78	5
aS2dC	75250.68	2656.30	5
dC	80940.67	3910.24	5
S0dC	81138.23	3898.49	5
d	87042.08	3450.65	6
S0d	87422.95	3461.20	6
S2dC	95200.66	3309.90	5
S2d	96338.41	3074.81	6

options	Σ MB	Σ sec	#abort
aS2Ad	86.01	8860.21	0
S2Ad	93.05	8070.87	0
aS2A	98.30	237.05	0
S2A	101.05	225.28	0
aS0Ad	124.05	598.95	0
aAd	125.95	598.72	0
S0Ad	129.25	879.53	0
Ad	130.84	881.96	0
aS0A	138.05	207.70	0
aA	139.80	207.56	0
S0A	142.49	195.28	0
A	142.49	196.47	0
aS2	1781.25	73157.34	5
aS2d	2017.67	69106.56	5
a	2048.08	73194.60	5
aS0	2055.19	73159.30	5
aS2C	2075.55	70694.50	5
ad	2217.88	66610.91	5
aS0d	2218.74	66415.87	5
aC	2414.62	70575.83	5
aS0C	2435.20	73133.52	5
aS2dC	2656.30	75250.68	5
adC	2780.84	63526.28	5
aS0dC	2786.47	63571.98	5
S2	2926.78	74687.03	5
S2d	3074.81	96338.41	6
S2C	3282.80	70905.50	5
S2dC	3309.90	95200.66	5
	3363.05	72544.45	5
S0	3367.95	72701.28	5
d	3450.65	87042.08	6
S0d	3461.20	87422.95	6
C	3750.15	64287.07	5
S0C	3752.76	64260.64	5
S0dC	3898.49	81138.23	5
dC	3910.24	80940.67	5

Figure 5.8: Different Option Settings, Sorted According to Accumulated Time and Memory Consumption Over the Benchmark Examples Fischer 2-9, CSMA/CD 2-9, and Token Ring HDDI 2-9. `#abort` is the Number of Runs That Were Aborted; Those Runs Would Consume More Than 4 Hours Time or 800 MB of Memory to Complete.

Chapter 6

The Model Augmentation Technique

Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius—and a lot of courage—to move in the opposite direction.

— Ernst F. Schumacher

Model augmentation is an approximation technique with the potential to drastically improve the symbolic analysis of real-time systems. It is beneficial, if the system exhibits repetition of control situation over time under regularities that can be derived from the control structure of the isolated processes.

The state-space exploration of real-time systems yields a number of sets of clock evaluations. Often the granularity of these sets is unnecessarily fine and exhibits regularities. With model augmentation certain regularities can be approximated effectively. If two configurations match modulo time delay, the later one (with respect to time) might require a large number of transitions to be taken, before it is finally reached. It can be reached in a smaller number of steps *via a shortcut-like addition in the model.*

The problematic part is to detect regularities in the symbolic state space exploration and find the appropriate shortcuts. In general this cannot be easier than exploring the state space itself. However, in certain scenarios the syntactic structure of the model suffices to find beneficial additions. For example, this is the case when a scheduler dictates repetition of similar configurations over time.

Model augmentation always entails an increase of the state space. Nevertheless, the number of symbolic states *that have to be explored can be smaller, because many small sets of clock evaluations can be subsumed by one large set. We illustrate this phenomenon by applying of our technique to a bricks sorter model.*

Since the model itself is changed, not every property established in the modified model also holds in the original one. Safety properties that hold for the modified model carry over to the original one. With a slight modification in the execution semantics, the technique is even conservative with respect to those liveness properties that rely on the universal quantification over timed traces.

6.1 Adding Parts to UPPAAL Models

Model augmentation relies on a phenomenon that is peculiar for symbolic real-time model checking. Repetition of control situations in the exploration of the model can lead to a large number of small zones that are merely shifted by small time values. Adding carefully selected transitions to the model can construct a "shortcut" that allows to subsume these small step by one large time delay.

In this Section we elaborate this idea to a formal definition.

Recall the definition of UPPAAL models from Section 2.1 and the description of the forward state space exploration in Section 4.3. The algorithmic treatment relies on the storage of symbolic states of the form $(\vec{l}, e, D)$, where $\vec{l}, e$ is the discrete part of the state and the zone D encodes a set of clock evaluations. Two zones are only comparable in the reachability algorithm (Figure 4.2), if one is contained in the other. If the set of zones associated with the same discrete part shows other regularities, these are not exploited. For some models a particular regularity exist: the zones are *shifted* by a certain time delay. For example, for one clock x the zones representing $0 \leq x \leq 1$, $2 \leq x \leq 3$, $4 \leq x \leq 5$, etc, could occur. In the symbolic exploration these yield separate entries in the PASSED list.

The reason for these regularities could be that the model in fact *waits* for a certain time delay k to be exceeded. In one step, however, only a small delay is allowed; after this the clock value is compared to k. If k is large with respect to the delay in one step, a large number of steps is necessary.

In certain scenarios this value k is *known*. For example, if a scheduler is present that assigns blocks of execution time to a set of tasks, the next execution frame starts after time k. Now taking a number of small steps to reach k corresponds to one *actual behavior* of the system that we intend to approximate. Under certain conditions the sequence of delay steps might not be completed, e.g., if an interrupt occurs. Then it is dangerous to simplify the behavior to "delay until time k", since this potentially introduces an error.

Our suggestion is to allow for the *option* of skipping to the value k without enforcing it. In practice this means that extra transitions are added to the model in a careful manner. Every step that has been possible before should also be possible after the modification.

How can *strictly adding* to the behavior of the model be of help? To understand this, one has to recall that the symbolic state space can be expanded in a breadth-first manner. If the next possible steps from the zone $0 \leq x \leq 1$ yield both $2 \leq x \leq 3$ and

$2 \leq x \leq k$, then the former successor is *subsumed* by the latter one. Algorithmically this means that $2 \leq x \leq 3$ no longer needs to be explored, since it is already treated by the exploration of $2 \leq x \leq k$. If there is a time between 3 and k such that different behavior—say, an interrupt—occurs, this interrupt is explored as a successor of the symbolic state with zone $2 \leq x \leq k$.

6.1.1 Formal Definition

We define the transformation of an original UPPAAL model as a tuple that contains additional transitions and locations.

Definition 6.1 (Model Augmentation)
Let $M = \langle \vec{A}, Vars, Clocks, Channels, Type \rangle$ *be an* UPPAAL *model. The tuple* $\mathfrak{A} = \langle l_i \xrightarrow{g,\vec{a}} l', L_{\mathfrak{A}}, T_{\mathfrak{A}}, Type_{\mathfrak{A}} \rangle$ *is a* model augmentation*, if the following holds.*

- $l_i \in L_i$ *for some process* A_i *that is part of* $\vec{A}$*, where we require* $o(l_i)$*; we call* l_i *the* augmentation point *of* $\mathfrak{A}$*,*
- g *a guard,*
- $\vec{a}$ *a list of assignments,*
- $l' \in L_{\mathfrak{A}}$*,* $L_{\mathfrak{A}}$ *a set of fresh locations,* $L_{\mathfrak{A}} \cap (\bigcup L_i) = \varnothing$*,*
- $T_{\mathfrak{A}}$ *a set of transitions, such that all sources are in* $L_{\mathfrak{A}}$ *and all targets are in* $L_{\mathfrak{A}} \cup \bigcup L_i$*, and*
- $Type_{\mathfrak{A}} : L_{\mathfrak{A}} \to \{o, u, c\}$ *the type function for the fresh locations.*

Then $Aug_{\mathfrak{A}}(M)$ *is the* UPPAAL *timed automata model* $\langle \vec{A}', Vars, Clocks, Channels, Type' \rangle$ *that enriches* M *in the following sense:*

1. $\vec{A}' = A_1, \ldots, A_{i-1}, A_i', A_{i+1}, \ldots, A_n$*, with*
   $A_i' = \langle L_i \uplus L_{\mathfrak{A}}, T_i \uplus T_{\mathfrak{A}} \uplus \{l_i \xrightarrow{g,\vec{a}} l'\}, Type_i \uplus Type_{\mathfrak{A}}, l_i^0 \rangle$,[1]

2. $Type'$ *extends* $Type$ *by mapping locations* $l_{\mathfrak{A}} \in L_{\mathfrak{A}}$ *to* $Type_{\mathfrak{A}}(l_{\mathfrak{A}})$*.*

We call $Aug_{\mathfrak{A}}(M)$ *the* augmented model*.*

It is almost obvious that this modification is sound for safety. The key observation is that for augmentations returning to the original control location after some carefully selected time delay, this modification *improves* model checking time.

Introducing additional loops can be understood as "parking" a process until something relevant happens or, more precisely, until some condition depending on a timing constraint is met. We illustrate this with a somewhat artificial example.

Example 6.2 (Delay Loop) *Consider an* UPPAAL *model* M *with a single process* P *(Figure 6.1).* P *performs a number of delay loops of duration* `SMALL`*, and leaves the loop when a total delay* `LARGE` *was reached. When the property* `E<> P.QUICK` *is*

[1] The symbol $\uplus$ denotes disjoint union.

	M			$Aug_{\mathfrak{A}}(M)$		
LARGE	#explored	time [sec]	memory [KB]	#explored	time [sec]	memory [KB]
10	8	0.01	376	9	0.01	448
100	35	0.01	440	9	0.01	376
1000	305	0.04	424	9	0.01	440
10˙000	3˙005	1.51	1˙704	9	0.01	440
100˙000	30˙005	175.21	5˙440	9	0.02	416
1˙000˙000	300˙005	22˙449.94	42˙792	9	0.02	400

Table 6.1: Time and memory consumption for parameters `SMALL := 10` and `LARGE` variating. All measurements are made using the UPPAAL model checking engine version 3.1.57 executing on a 300 MHz UltraSPARC-II processor, with breadth-first search and active clock reduction.

verified in forward state space exploration, a large number of these delay loops are explored, before it is established that the location `QUICK` *indeed cannot be reached. Figure 6.1 shows the augmented model* $Aug_{\mathfrak{A}}(M)$ *on the right, where*

$$\mathfrak{A} = \left\langle T \xrightarrow{\texttt{x <= LARGE}} \texttt{AUGMENT}, \{\texttt{AUGMENT}[\mathit{Inv}\text{:}\ \texttt{x} <= \texttt{LARGE}]\}\,,\ \ \{\texttt{AUGMENT} \to \texttt{S}\} \right\rangle .$$

The augmented process can be understood to "park" in location `AUGMENT` *until time has progressed enough to pass the guard* $x >$ `LARGE`.

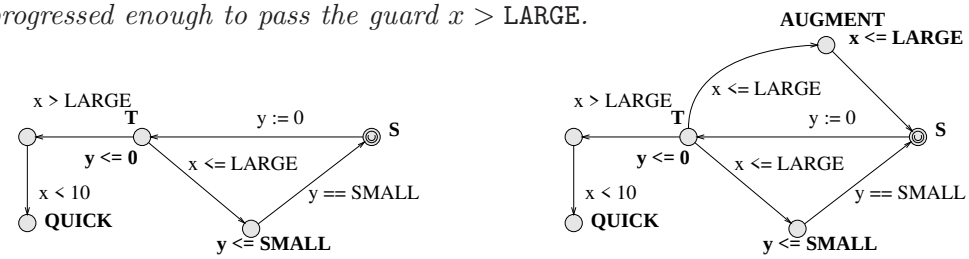


Figure 6.1: The original process P (left), and P with model augmentation $\mathfrak{A}$ (right). x and y are clocks, the initial location S is urgent. In both cases location `QUICK` cannot be reached for constants `LARGE`≥ 10.

Table 6.1 shows data for forward reachability analysis with UPPAAL. *We model check the safety property* `A[] not P.QUICK`. *Since this property holds, the compete state-space is explored symbolically. The number of explored symbolic states in the original model with process P (left) increases linearly in the parameter* `LARGE`, *whereas this number stays constant when* $Aug_{\mathfrak{A}}(P)$ *(right) is used.*

6.2 Soundness of Model Augmentation

We are now ready to state and prove the soundness of the model augmentation technique.

Theorem 6.3 (Soundness) *Let* $M = \langle \vec{A}, Vars, Clocks, Channels, Type \rangle$ *be an* UPPAAL *timed automaton model and* φ *be a local property for* M*. Then for any model augmentation* $\mathfrak{A} = \langle l_{\mathfrak{A}} \xrightarrow{g,\vec{a}} l'_{\mathfrak{A}}, L_{\mathfrak{A}}, T_{\mathfrak{A}} \rangle$ *with* $l_{\mathfrak{A}} \in L_i$ *for some* i*, the following holds:*

$$M \models \texttt{E<>}\ \varphi \qquad \Rightarrow \qquad Aug_{\mathfrak{A}}(M) \models \texttt{E<>}\ \varphi\,.$$

Proof: We show that $\mathcal{T}(M) \subseteq \mathcal{T}(Aug_{\mathfrak{A}}(M))$. For this it suffices to show that for any configuration $\mathbf{s} = (\vec{l}, e, \nu)$ reached in M, every enabled step is also enabled in the corresponding configuration $\mathbf{s}_{\mathfrak{A}} = (\vec{l}, e, \nu)$ in $Aug_{\mathfrak{A}}(M)$.

Assume a simple or synchronized action step is enabled in $\mathbf{s}$. For every transition of M, there is an equivalent transition in $Aug_{\mathfrak{A}}(M)$. By Definition 6.1, $o(l_{\mathfrak{A}})$, thus $\neg c(l_{\mathfrak{A}})$ and the transition $l_{\mathfrak{A}} \xrightarrow{g,\vec{a}} l'_{\mathfrak{A}}$ has no precedence over other action steps. Since $\vec{l}$, e, and ν of $\mathbf{s}$ and $\mathbf{s}_{\mathfrak{A}}$ are identical, the same simple or synchronized action step is then enabled in $\mathbf{s}_{\mathfrak{A}}$.

Assume a delay step of duration d is enabled in $\mathbf{s}$. Then conditions 1. through 4. from Definition 2.7 are met. For $\mathbf{s}_{\mathfrak{A}}$, the conditions 2 and 4 then also hold true, because $\vec{l}$, e, and ν are identical for $\mathbf{s}$ and $\mathbf{s}_{\mathfrak{A}}$. Condition 1 is met, because no action step leaving an urgent location is enabled in $\mathbf{s}$. $o(l_{\mathfrak{A}})$ entails $\neg u(l_{\mathfrak{A}})$, thus $l_{\mathfrak{A}} \xrightarrow{g,\vec{a}} l'_{\mathfrak{A}}$ does not introduce an additional one for $\mathbf{s}_{\mathfrak{A}}$.

As for condition 3, $l_{\mathfrak{A}} \xrightarrow{g,\vec{a}} l'_{\mathfrak{A}}$ does not carry a synchronization by definition. Thus no synchronization on an urgent channel can be enabled in $\mathbf{s}_{\mathfrak{A}}$, unless it was also enabled in $\mathbf{s}$, which is not the case.

Thus a delay step of duration d is also enabled in configuration $\mathbf{s}_{\mathfrak{A}}$, completing the proof. □

Corollary 6.4 (Conservative for Safety) *Let* M *be an* UPPAAL *timed automaton model and* ψ *be a local property for* M*. For a model augmentation* $\mathfrak{A}$ *for* M*:*

$$Aug_{\mathfrak{A}}(M) \models \texttt{A[]}\ \psi \qquad \Rightarrow \qquad M \models \texttt{A[]}\ \psi\,.$$

6.2.1 Suitable Augmentations

Though not formally required, model augmentations have to return to the original control structure. Otherwise they never yield an improvement.

Model augmentation adds both to the state space and to the level of nondeterminism. In general this is a bad thing. The modification is only beneficial, if the additional loop cuts out long and tedious repetitions of control sequences that are only distinguishable by the passage of time. I.e., repetitions modulo a certain clock shift must exist. It is necessarily to apply augmentation in all processes of the model before this phenomenon can be exploited.

It is crucial that the newly introduced loop is taken early in the state space exploration. In forward reachability analysis this can be achieved by using a breadth-first search order. Then one augmented loop is explored before the concrete control returns to the augmentation point. A more rigorous possibility is to modify the model

checking algorithm in such a way that the transitions starting model augmentations are explored first. E.g., in the algorithm in Figure 4.2, PASSED can be implemented as a priority queue. Symbolic states where the discrete part contains locations from the augmentation can be moved to the front of the queue and thus are explored earlier.

The challenges for successful model augmentation are

1. To find promising augmentation points,
2. To identify suitable delays, and
3. To construct conditions that should trigger a return to the original control structure.

In Section 6.4 we exemplify this on a medium sized example with two parallel tasks, where a Round-Robin scheduler dictates repetitions over time.

6.3 Model Augmentation for Universal Path Properties

Universal path properties are the fragment of TCTL [HNSY94], where a property can be refuted by a single counter-example trace. We extend our model augmentation technique to be conservative with respect to this richer set of properties. In order to preserve deadlocks, we modify the transition relation relative to the model-augmentation.

Definition 6.5 (Universal Path Property)
Universal path properties ζ *are formulas of the following syntax.*

$$\zeta ::= \texttt{A[]}\zeta \;\mid\; \texttt{A<>}\zeta \;\mid\; \zeta \vee \zeta \;\mid\; \zeta \wedge \zeta \;\mid\; (\zeta) \;\mid\; \varphi$$

Here, φ is a local property (see Definition 2.11).

Note that negation is only allowed at the local property level. In particular the definition of unbounded response, $\texttt{A[]}\left(\varphi \Rightarrow \texttt{A<>}\psi\right)$, is equivalent to $\texttt{A[]}\left(\neg\varphi \vee \texttt{A<>}\psi\right)$ and thus is a universal path formula.

The operator A<> expresses *inevitability:* at some point in the future, some property ζ will necessarily hold. A<> ζ is violated, if there exists either an infinite trace not containing a configuration, where ζ holds, or some maximally extended finite trace that does not contain a configuration where ζ holds.

Definition 6.6 (Semantics of Universal Path Properties)
A trace $\sigma = (\mathbf{s}_0, \mathbf{s}_1, \ldots) \in \mathcal{T}(M)$ satisfies an universal path formula ζ at position i,

in short $(\sigma, i) \models \zeta$*, according to the following rules.*

$$\begin{array}{lcll}
(\sigma,i) & \models & \texttt{A[]}\zeta & \textit{iff} \quad \forall j \geq i.\ (\sigma,j) \models \zeta \\
(\sigma,i) & \models & \texttt{A<>}\zeta & \textit{iff} \quad \exists j \geq i.\ (\sigma,j) \models \zeta \\
(\sigma,i) & \models & \zeta_1 \vee \zeta_2 & \textit{iff} \quad (\sigma,i) \models \zeta_1 \ \textit{or} \ (\sigma,i) \models \zeta_2 \\
(\sigma,i) & \models & \zeta_1 \wedge \zeta_2 & \textit{iff} \quad (\sigma,i) \models \zeta_1 \ \textit{and} \ (\sigma,i) \models \zeta_2 \\
(\sigma,i) & \models & (\zeta) & \textit{iff} \quad (\sigma,i) \models \zeta \\
(\sigma,i) & \models & \varphi & \textit{iff} \quad s_i \models_{loc} \varphi
\end{array}$$

Again, φ *is a local property and thus does not contain path quantifiers.*

An Uppaal *model* M *satisfies a universal path formula* ζ*, if for all traces* $\sigma = (\mathbf{s}_0, \mathbf{s}_1, \ldots) \in \mathcal{T}(M)$*,* $(\sigma, 0) \models \zeta$*.*

Applying model augmentation with universal path properties raises a technical problem. It could be the case that the new transition at the augmentation point allows to escape from a deadlock situation, where no further action transitions are possible. In this situation, `A<>` properties in the augmented model could hold, though they do not for the original system.

The solution is conceptually simple. We require that in the augmentation point, the added transition can only be taken, if another action transition can be taken as well. We formalize this as follows.

Definition 6.7 (Augmented Path Semantics) *Let* M *be an* Uppaal *model and* $\mathfrak{A} = \langle l_i \xrightarrow{g,\vec{a}} l', L_{\mathfrak{A}}, T_{\mathfrak{A}}, Type_{\mathfrak{A}} \rangle$ *a model augmentation of* M*. We define the set of* weak traces *of* $Aug_{\mathfrak{A}}(M)$ *as the subset of* $\mathcal{T}(Aug_{\mathfrak{A}}(M))$ *(see Definition 2.8) that is generated by the following side condition:*

in a configuration $(\vec{l}, \mathrm{e}, \nu)$ *with* $l_i \in \vec{l}$*, the transition* $l_i \xrightarrow{g,\vec{a}} l'$ *is only enabled, if another action or synchronized action transition is enabled.*

All traces containing steps that violate this condition are removed from $\mathcal{T}(Aug_{\mathfrak{A}}(M))$ *to yield the set* $\mathcal{T}^{\mathfrak{A}}(Aug_{\mathfrak{A}}(M))$*. We refer call* $\mathcal{T}^{\mathfrak{A}}(Aug_{\mathfrak{A}}(M))$ *the* augmented path semantics *of* M *relative to* $\mathfrak{A}$*. We write* $Aug_{\mathfrak{A}}(M) \models^{\mathfrak{A}} \zeta$*, if and only if for all traces* $\sigma = (\mathbf{s}_0, \mathbf{s}_1, \ldots) \in \mathcal{T}^{\mathfrak{A}}(Aug_{\mathfrak{A}}(M))$*,* $(\sigma, 0) \models \zeta$*.*

For example, in the symbolic response algorithm in Figure 4.4 this side condition would change line 10 to

$$10'\quad \textsc{Forall}\ \text{enabled}\ \vec{l} \xrightarrow{g,a,r} \vec{l}'\ \text{where}\quad \exists \vec{l}''\ \text{from}\ \vec{L} \setminus L_{\mathfrak{A}}.\ \text{enabled}\ \vec{l} \xrightarrow{g'',a'',r''} \vec{l}'' \\ \vee\ \exists l_{\mathfrak{A}} \in \vec{l}.\ l_{\mathfrak{A}} \in L_{\mathfrak{A}}$$

Theorem 6.8 (Approximation of Universal Path Properties) *Let* M *be an* Uppaal *timed automaton model and* ζ *an universal path property. For arbitrary model augmentations* $\mathfrak{A} = \langle l_i \xrightarrow{g,\vec{a}} l', L_{\mathfrak{A}}, T_{\mathfrak{A}}, Type_{\mathfrak{A}} \rangle$*:*

$$Aug_{\mathfrak{A}}(M) \models^{\mathfrak{A}} \zeta \qquad \Rightarrow \qquad M \models \zeta\,.$$

Proof: It suffices to show that $\mathcal{T}(M) \subseteq \mathcal{T}^{\mathfrak{A}}(Aug_{\mathfrak{A}}(M))$.

Note that $o(l_i)$ and the added transition does not carry synchronization labels. Thus the of an additional transition from l_i does not prevent any originally possible action or delay step.

If $\sigma \in \mathcal{T}(M)$ is an infinite trace, then it is also present in $\mathcal{T}^{\mathfrak{A}}(Aug_{\mathfrak{A}}(M))$. If $\sigma \in \mathcal{T}(M)$ is finite—and thus maximally extended, see Definition 2.8 (ii)—, then it is no prefix of a trace in $\mathcal{T}^{\mathfrak{A}}(Aug_{\mathfrak{A}}(M))$ by Definition 6.7. Thus $\sigma \in \mathcal{T}^{\mathfrak{A}}(Aug_{\mathfrak{A}}(M))$.

□

Example 6.9 (Liveness in Delay Loop)
For the Uppaal *model M with the single process P in Figure 6.1:*

$Aug_{\mathfrak{A}}(M) \models^{\mathfrak{A}}$ `A[]` ((`not` $P.T$) `or A<>` $P.S$),

and thus $M \models$ `A[]` ((`not` $P.T$) `or A<>` $P.S$).

Note that whenever transition $T \rightarrow$ `AUGMENT` *is enabled then one of the original transitions in M is also enabled (due to the invariant $y \leq 0$ at T). Thus the side condition in Definition 6.7 is always fulfilled for $T \rightarrow$* `AUGMENT` *and $\mathcal{T}(Aug_{\mathfrak{A}}(M)) = \mathcal{T}^{\mathfrak{A}}(Aug_{\mathfrak{A}}(M))$.*

6.4 Bricks Sorter Example

We demonstrate how to apply our model augmentation technique on a special class of examples, namely Uppaal *models of task-based LEGO® RCX™ programs. These programs can be automatically translated to* Uppaal *models. We use the bricks sorter example from [IKL+00] as a case study.*

The bricks sorter model is augmented in all places, where control loops in the structure were detected. For safety-properties, this yields a speed-up in terms of model checking time.

6.4.1 The Bricks Sorter Model

The bricks sorter (Figure 6.2) is a machine consisting of a conveyor belt, a light sensor, and a kick-off arm. Red and black bricks are transported on the conveyor belt past the sensor which is sensitive enough to distinguish the two colors. Some time later, the kick-off arm can push a brick off the belt. A controller coordinates sensor and kick-off arm and tries to ensure that every black brick is pushed off, while every red brick is allowed to pass.

In a physical implementation, this system was built in LEGO® with a RCX™ Mindstorm micro-controller as the control unit. This controller executes up to ten tasks that are organized by a deterministic scheduler in Round-Robin fashion. Two tasks `main` and `kick-off` are used in the RCX™ program, which are translated to three Uppaal processes `Scheduler`, `RCX0_main`, and `RCX0_kick_off` (see [IKL+00]). Three processes `black_brick`, `black_brick2`, and `kick_off_arm` are added by hand to model the environment. The process `black_brick` models a black brick passing

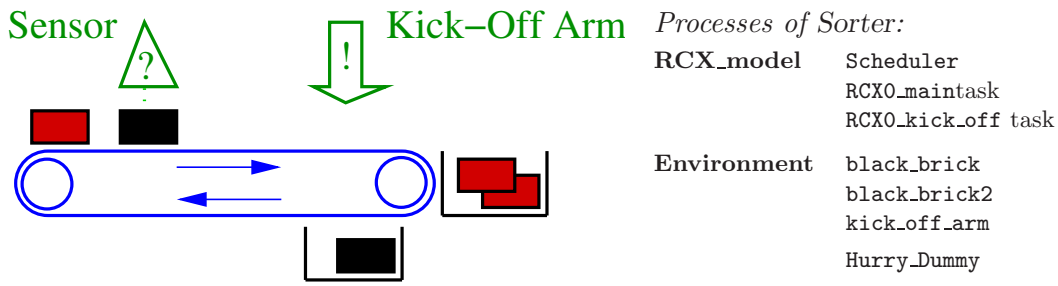


Figure 6.2: Schematic Description of the Bricks Sorter.

the sensor after a small fixed time delay. Process `black_brick2` models a brick that is released with an additional non-deterministic delay between 25˙500 and 250˙000 time units. A process `Hurry_Dummy` is added as an auxiliary one-location process that always offers synchronization on the urgent channel `Hurry`. This combines to the UPPAAL model *Sorter*. We want to establish a safety property stating that the second brick is kicked off, regardless of the actual time delay value in `black_brick2`:

$$Sorter \models \texttt{A[]} \quad \texttt{not black\_brick2.PASSED}.$$

6.4.2 Augmentation of the Bricks Sorter Model

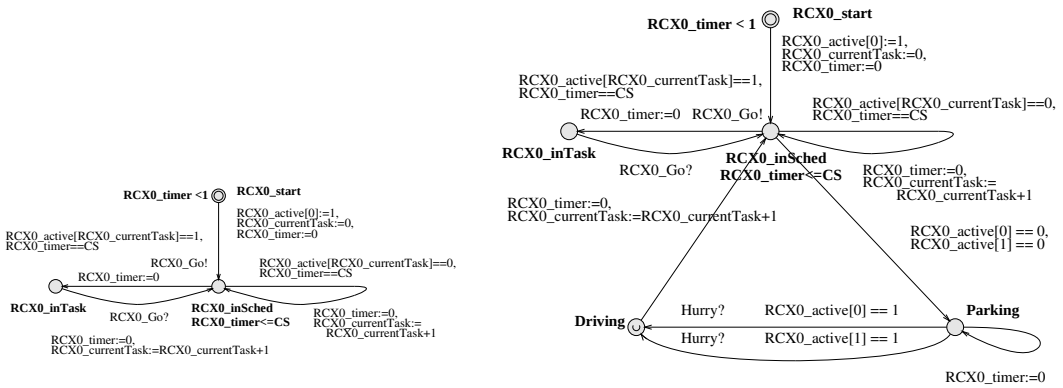
Starting with the UPPAAL model *Sorter*, we define a sequence of model augmentations. The processes `Scheduler`, `RCX0_main`, and `RCX0_kick_off` are augmented. The augmentation points are chosen according to the nature of the particular process.

Scheduler. Figure 6.3 *(i)* displays the UPPAAL process `Scheduler`. It uses the array `RCX0_active` and the integer variable `RCX0_current_task` to keep track of the next task to release. It does so by taking the transition to `RCX0_inTask`, if possible, and otherwise idles via the self loop at `RCX0_inSched`.

The augmented model *(ii)* is shown on the right. If all tasks are inactive, the location `Parking` can be reached. As soon as one of the tasks becomes active again. Synchronization on the urgent channel `Hurry` and declaring the location `Driving` urgent ensures that the augmented Scheduler does not remain parking unnecessarily long.

Tasks. In the processes `RCX0_main` and `RCX0_kick_off`, the conditional tests cause loops in the control structure. Model augmentations are applied in eight places. Six of them are wait conditions for conditions to hold true, like the one shown in Figure 6.4. The remaining two are allowing optional time delay, whenever progress depends on timing conditions to be met.

This amounts to nine model augmentations that add 16 locations and 34 transitions in total. We refer to the obtained model as $Aug^*_{21}(Sorter)$.



(i) Original `Scheduler` Process *(ii)* Augmented Scheduler Process

Figure 6.3: The Round-Robin scheduler *(i)* repeatedly toggles through the list of tasks. If the respective task is active, execution of this task is released (`RCX0_inTask`). The task executes an instruction and then hands back control to the scheduler by synchronizing on channel `RCX0_go`. If the respective task is inactive, it is skipped (self-loop to the right). Hereafter, the scheduler moves on to the next task. The variable `RCX0_current_task` wraps around to 0, when the number of existing tasks is exceeded. When augmented *(ii)*, the scheduler is additionally allowed to move to a location `Parking` if all tasks are inactive. This location has to be left immediately if one of the tasks become active again (synchronization on urgent channel `Hurry`).

```
*** Task 0 = main
...
031 InType      2, Switch
034 InMode      2, Boolean
037 OutDir      A, Fwd
039 OutMode     A, On
041 OutPwr      A, 1
045 OutDir      B, Fwd
047 OutMode     B, On
049 OutPwr      B, 6
053 Display     1
057 StartTask   1
059 Test        Input(0) <= var[4], 70
067 Jump        59
070 ...
```

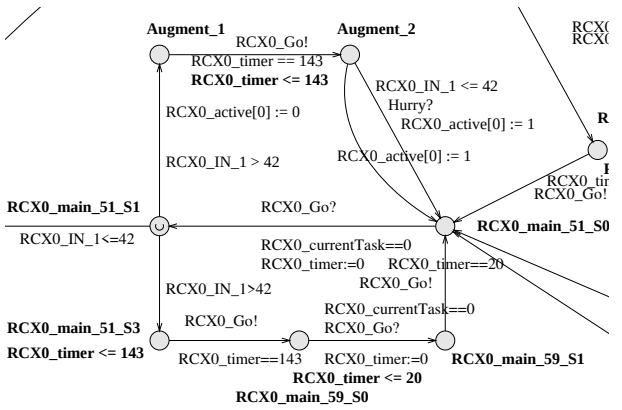


Figure 6.4: Part of a LEGO® RCX™ task program (left). When at position `059`, the task continues no sooner with position 70 than the sensor input 0 equals or falls below the value of `var[4]`. In the UPPAAL process (right) this corresponds to the counter-clockwise loop from `RCX0_main_S59_S0` to `RCX0_main_S59_S1`. The process is augmented with locations `Augment_1` and `Augment_2`. This adds the possibility of a clock-wise loop, where the process parks at `Augment_2`. Note that the augmentation hat to "fit" into the model. In the augmentation, `RCX0_go!` is offered and the variable `RCX0_active[0]` is set properly.

	#explored states	average number of successors	#dead-locks	time [sec]	space [KB]
Sorter	151˙103	1.28	0	86.84	1˙840
$Aug^*_{\mathfrak{A}}(Sorter)$	22˙966	2.09	20	21.15	2˙512

Table 6.2: Run-Time comparison of the bricks sorter model and its augmented version. Both measurements used `verifyta` from UPPAAL 3.1.58 with switch settings `-sWabA -S 2` (active clock reduction, breadth-first search, convex hull approximation enabled—see Chapter 5—, running on a 300 MHz UltraSPARC-II processor). The number of explored states and average number of successors was determined in an extra run, using a version of `verifyta` that was modified to output these diagnostics.

The run-time comparison of *Sorter* and $Aug^*_{\mathfrak{A}}(Sorter)$ is given in Table 6.2. On $Aug^*_{\mathfrak{A}}(Sorter)$, forward state space exploration runs four times faster, but consumes slightly more memory. The larger memory consumption indicates that in the augmented version a larger number of incomparable symbolic states are encountered.

This is a consequence both of the additional control structure and the application of the convex hull over-approximation (see Section 5.2.1). With each repetition of the control loop in *Sorter*, the entry in PASSED is updated to a *larger* convex hull over the encountered sets clock regions. The total size of PASSED does not increase by then.

$Aug^*_{\mathfrak{A}}(Sorter)$ exhibits a considerably higher average number of successors, which can be understood as additional non-determinism. $Aug^*_{\mathfrak{A}}(Sorter)$ yields additional deadlock states, apparently due to unfortunate timing clashes. In general this is undesirable, for it remains unclear whether the original model is deadlock-free. In this class of examples, however, the original model is *by construction* deadlock-free and thus the deadlocks are necessarily spurious.

Note that $Aug^*_{\mathfrak{A}}(Sorter)$ is also good for proving other safety properties. If they hold in $Aug^*_{\mathfrak{A}}(Sorter)$, the whole state-space is expanded symbolically in one run; thus the run-time data would be identical modulo small differences caused by the evaluation of the invariant ψ for each symbolic configuration.

In the model checking run we made use of the convex-hull approximation technique. Here, the clock evaluations—represented by zones—of symbolic configurations with coinciding discrete part are replaced by a single configuration, where the clock evaluation is computed as the smallest convex zone encapsulating the original two zones. This increases the number of reachable states, but is conservative with safety properties in one direction. Without this option, the large number of created symbolic states in this example exceeded the available 1 GB of memory, both for *Sorter* and for $Aug^*_{\mathfrak{A}}(Sorter)$.

6.5 Reflection: Model Augmentation

Model augmentation is a very specialized approximation technique. We restrain from *pruning* processes, i.e., existing behavior of the system is never prohibited. This allows for a general soundness proof and enables combination with other over-approximations like convex hull.

We applied our technique on UPPAAL models of LEGO® RCX™ programs. Though the savings are not drastic, the run-time improvements in a brick-sorter example demonstrate the benefits of our technique. The time savings were roughly 75%, but slightly more memory was consumed with the augmented model. More experiments are needed to determine, whether this is specific only to this example.

The augmentation points of the tasks—and thus the associated augmentation—can be either derived from the control structure of the UPPAAL model, or even directly from the LEGO® RCX™ program. There exists a translation from RCX programs to corresponding UPPAAL models [IKL+00]. It is possible to modify this translation to directly compute an augmented version of the UPPAAL models, providing full automation for this optimization technique in this class of application.

Model Augmentation suffers from two major drawbacks. First, it can only be applied if repetition of control situations exists and can be detected effectively. Both hold only for rather special models. Second, the introduction of new traces ultimately sentences the technique to be at best conservative with respect to safety and universal path properties. The subsequent Chapter 7 introduces an approximation technique that allows approximation for a larger set of properties.

Chapter 7

Abstract Interpretation of Dense Real-Time

I don't have any solution, but I certainly admire the problem.

— Ashleigh Brilliant

Every system denotes some sort of computation in an universe. Abstract interpretation of a system uses this denotation to perform computations in another *universe in such a way that the result allows us to draw conclusions on the behavior in the concrete universe.*[1]

The formal framework of abstract interpretation originated in the work of Cousot and Cousot [CC77]. It relies on a non-standard (abstract) interpretation of a given system description that unfolds to a much smaller structure than the standard interpretation. This can effectively be applied in an a posteriori analysis of a system. It does not hurt that the analysis is essentially incomplete and may fail to either establish or refute one specific property φ. The attitude here is to detect as many properties as possible in an economic way.

In verification, it is typically required to establish (or refute) one particular property φ. Since one abstract interpretation may fail to do either, one can try to refine *the abstraction, until either φ or $\neg\varphi$ holds. In the worst case this refinement may lead back to the concrete (non-abstracted) system. One systematic way to perform this refinement is* predicate abstraction *(sometimes also called* Boolean abstraction*). Here the abstraction is formed compositionally by means of Boolean predicates. Adding one predicate corresponds to refining the abstraction.*

The abstract interpretation framework has been used before to formalize approximations of safety in real-time systems [WT94, DT98]. Ours is—to the best of our knowledge—the first work that also allows for liveness in this context. Liveness in dense real-time is complicated by the possible sequences of infinitesimally decreasing delay steps; they constitute a degenerated behavior of a system which has to be weeded out. We get rid of this by a non-convergence assumption that is weaker than non-zenoness. The operative point is that this assumption can be incorporated syntactically by restricting delay steps.

7.1 Outline of this Chapter

This section give a brief description of the problems connected with applying abstract interpretation on real-time systems, sketches our proposed solution and lists the contents of this Chapter in detail.

In the following we adopt the framework of predicate abstraction to real-time systems. Instead of carrying out the analysis on the concrete system, we attempt to establish them in an abstracted version. This abstract version has to be constructed in such a way that certain properties of the abstract version also hold in the concrete one.

Since both safety and liveness properties shall be verified, this approach dictates the construction of *two* abstract transition relations, sometimes called *MUST* (under-approximation) and *MAY* (over-approximation). Existential operators have to be established via MUST, while for universal operators MAY is appropriate (see, e.g., [GHJ01]).

In dense real-time this entails a technical complication. Since the state space is infinite, only equivalence classes of states can be treated. This a priori approximation hinders the establishing of liveness properties—considering all possible clock evaluations, the MUST part is in general never satisfied. The root of this problem is the quantitative treatment of time: If one can distinguish a delay of 1/2 and 1/4 of a time unit, then the configurations of a system before and after taking *any* small delay are not bisimilar—and the analysis cannot restrict to a finite quotient of the system.

We solve this problem by removing the quantitative reasoning over time delay steps in the logic. This yields a next-free version of the (un-timed) μ-calculus, which we call next-free μ-calculus. This restricts to properties where certain future situations are reached either potentially or inevitably. The number of steps or the duration between steps cannot explicitly be observed. Under a non-convergence assumption on the timed behavior of the system, we can syntactically restrict the delay-steps to cross *some* boundaries. With respect to next-free μ-calculus, the standard semantics and the restricted semantics are equivalent.

[1] A slight adaptation of the frequently quoted description in [CC77].

We show how to apply predicate abstraction to real-time systems, where the properties are taken from next-free μ-calculus. Here the control structure is preserved; the abstraction merely addresses the regions of clock evaluations, which are the expensive part in real-time verification. Note that the abstracted systems no longer refer to the real-time nature of a trace. Any standard model checking tool can be used to establish safety and liveness properties in the abstracted system. Those expressible in next-free μ-calculus then also hold for the concrete system. Thus the reasoning about real-time equivalence classes is moved to the abstraction function that is in turn build via predicates over clock constraints.

We propose a novel algorithm for the stepwise refinement of finite state abstractions for a timed automaton. This sequence of abstractions converges toward the region graph of the real-time system, thus the method is complete with respect to our property language. This technique has the potential to be significantly cheaper than the region graph construction. Other than safe over-approximation, it can also be used to establish liveness properties.

Organization. The rest of this Chapter is structured as follows. Section 7.2 gives a brief introduction to the machinery of abstract interpretation. Section 7.3 introduces predicate abstraction as a special technique to construct the abstraction function. This set the stage for application on dense real-time systems. In Section 7.4 we review the basic notions of timed automata including a natural semantics based on a non-convergence assumption of time. The language to express properties of this model is a next-free version of the propositional μ-calculus, called next-free μ-calculus. We then define the notion of restricted delay steps and show that this restricted semantics of a timed automata is observationally equivalent to the natural semantics. In Section 7.5 the restricted semantics is used to define finite over- and under-approximations of timed systems. In Section 7.6 we introduce the concept of a basis as a set of abstraction predicates expressive enough to distinguish between any two different clock regions. We show that for predicate abstraction with a basis as abstraction predicates the approximation is exact with respect to the next-free μ-calculus. Then, in Section 7.7, we define a terminating algorithm for iteratively refining abstractions until the given property is either proved or refuted. Section 7.8 concludes and lists related work.

7.2 Abstract Interpretation

We briefly introduce the basics of abstract interpretation, as connecting two Kripke structures via a Galois connection. The properties that might be preserved are expressed in the propositional μ-calculus. As a incremental way to construct the Galois connection, we use predicate abstraction.

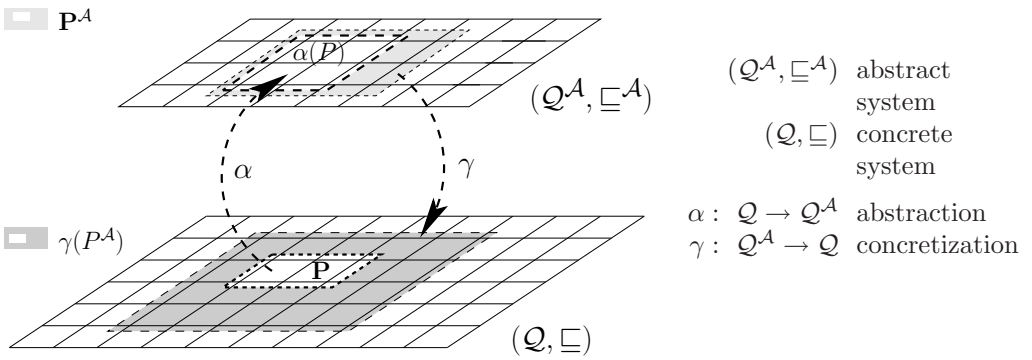


Figure 7.1: Galois connection: $\forall P \in \mathcal{Q},\ P^{\mathcal{A}} \in \mathcal{Q}^{\mathcal{A}}.\ \alpha(P) \sqsubseteq^{\mathcal{A}} P^{\mathcal{A}} \ \Leftrightarrow\ P \sqsubseteq \gamma(P^{\mathcal{A}})$

7.2.1 Galois Connections

Definition 7.1 (Galois Connection) *Let $(\mathcal{Q}, \sqsubseteq)$ and $(\mathcal{Q}^{\mathcal{A}}, \sqsubseteq^{\mathcal{A}})$ be lattice structures. We call a pair (α, γ) of functions $\alpha :\ \mathcal{Q} \to \mathcal{Q}^{\mathcal{A}}$ and $\gamma :\ \mathcal{Q}^{\mathcal{A}} \to \mathcal{Q}$ a* Galois connection*, if and only if*

$$\forall P \in \mathcal{Q},\ P^{\mathcal{A}} \in \mathcal{Q}^{\mathcal{A}}.\ \alpha(P) \sqsubseteq^{\mathcal{A}} P^{\mathcal{A}} \ \Leftrightarrow\ P \sqsubseteq \gamma(P^{\mathcal{A}})$$

We call α the abstraction *and γ the* concretization*, see Figure 7.1.*

Galois connections have many interesting properties that are of no further consequence in this context. We refer to [CC77, Dam96, Kel95] for a compendium.

As for which lattice structures to use in the Galois connection, there is a wide variety of choices. Throughout this Chapter, we treat model checking *state-based*, i.e., a formula in our temporal logic is given a semantics in terms of the set of states in the system that satisfy this formula. Consequently, the lattices of our Galois connection will be sets of states, with set-inclusion as partial order.

In contrast to the more general *trace-based* approach, state-based model checking may loose properties that are dependent on past operators (as demonstrated in [CC00]). Since we restrict to a forward-analysis, this is of no further consequence.

7.2.2 Property Preservation over Kripke Structures

We define Kripke structures as models for systems that we aim to analyze. The states (sometimes called worlds or nodes) are labeled with elements from a set of atomic properties A.

Definition 7.2 (Kripke Structure)
A Kripke structure *is a tuple $\langle \Sigma, P, R, I \rangle$, where*

- *Σ is a set of states,*
- *$P : \Sigma \to 2^A$ is a labeling function,*
- *$R \subseteq \Sigma \times \Sigma$ is the transition relation, and*
- *$I \subseteq \Sigma$ is the set of initial states.*

The properties of a state are expressed in the logic of the μ-calculus [Koz83]. We define an adoption for the case, where only the states are labeled and not the transitions.

Definition 7.3 (Propositional μ-Calculus)
Given a set A of atomic predicates and a set of formal variables Var, $p \in A$, $Z \in Var$. Formulas φ of the propositional μ-calculus *are constructed according to the following grammar.*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \mu Z.\varphi(Z) \mid Z$$

Here, $\bigcirc$ is the existential next operator and μZ denotes the least fix-point. As a well-formedness requirement, the formal variables Z can appear in $\varphi(Z)$ only under an even number of negations.

We formalize the relationship between Kripke structures and formulas by associating every (well-formed) formula with a set of states, where it is valid.

Definition 7.4 (Semantics of Propositional μ-Calculus)
Given a set of atomic predicates A, a Kripke structure $\langle \Sigma, P, R, I \rangle$, and an assignment $\vartheta : Var \to 2^\Sigma$. The set of states validating a formula φ of the propositional μ-calculus with respect to ϑ is defined inductively on the structure of φ (where $p \in A$ and $Z \in Var$).

$$\begin{aligned}
[\![p]\!]_\vartheta &:= \{ s \in \Sigma \mid p \in P(s) \} \\
[\![\neg\varphi]\!]_\vartheta &:= \Sigma \setminus [\![\varphi]\!]_\vartheta \\
[\![\varphi_1 \wedge \varphi_2]\!]_\vartheta &:= [\![\varphi_1]\!]_\vartheta \cap [\![\varphi_2]\!]_\vartheta \\
[\![\bigcirc\varphi]\!]_\vartheta &:= \{ s \in \Sigma \mid \exists s' \in \Sigma.\ s\,R\,s' \ \wedge\ s' \in [\![\varphi]\!]_\vartheta \ \} \\
[\![\mu Z.\varphi]\!]_\vartheta &:= \bigcap \left\{ \mathcal{E} \subseteq \Sigma \mid [\![\varphi]\!]_{\vartheta[Z:=\mathcal{E}]} \subseteq \mathcal{E} \right\}
\end{aligned}$$

7.2.3 Strong and Weak Preservation

If we describe our concrete system by a Kripke structure $\langle \Sigma, P, R, I \rangle$, we can relate it to an abstract system described by a Kripke structure $\langle \Sigma^\mathcal{A}, P^\mathcal{A}, R^\mathcal{A}, I^\mathcal{A} \rangle$. We do this by finding a Galois connection (α, γ) between Σ and $\Sigma^\mathcal{A}$ that fulfills some additional requirements. Before we go into details with this, let us consider what preservations we are interested in. We can distinguish strong and weak preservation.

Strong preservation requirement: A property holds in some state of the abstract structure, if and only if it holds in the corresponding state(s) of the concrete structure, i.e., $\gamma([\![\varphi]\!]^\mathcal{A}) = [\![\varphi]\!]$.

As it turns out, this dictates $\langle \Sigma^\mathcal{A}, P^\mathcal{A}, R^\mathcal{A}, I^\mathcal{A} \rangle$ to be bisimilar to the concrete system, and thus at least its bisimulation quotient. The additional requirements are exactly such that R and $R^\mathcal{A}$ are in a *bisimulation* relation [Dam96].

Computing a bisimulation quotient gives a lot of information about the system and is in general an expensive operation. Thus it is often desired to construct abstractions that are smaller and less expressive. We are satisfied, if all properties we can establish in the abstracted version carry over to the concrete system.

The alternation between existential and universal quantifiers causes complications. If the formula φ only contains universal quantification, (i.e., if it is expressible in $\forall CTL^\star$), it suffices to compute an abstraction that is a *simulation* of the concrete system. If both existential and universal quantifiers are present, the concrete system can be approximated using *two* abstracted transition relations: one over- and one under-approximation [Dam96, GHJ01]. Consequently, the interpretation of every formula has an over- and an under-approximation ($[\![\cdot]\!]^{\mathcal{A}}_{+}$ respectively $[\![\cdot]\!]^{\mathcal{A}}_{-}$).

Weak preservation requirement: If a property holds in some state of the abstract system, it also holds in the corresponding state(s) of the concrete system, i.e., $\gamma([\![\varphi]\!]^{\mathcal{A}}_{-}) \subseteq [\![\varphi]\!]$.

7.3 Predicate Abstraction

Predicate abstraction [GS97, BLO98, SS99] is a technique to build the Galois connection (α, γ) from a set of Boolean predicates over the concrete system. It can be used, e.g., to compute a finite approximation of a given infinite state transition system.

We use the power-set of concrete system states with ordinary set inclusion as the lower lattice in the Galois connection. Now the idea is to build equivalence classes on sets of states via Boolean predicates that evaluate over the concrete state space.

Assume n predicates. As for the abstracted system, this gives rise to three different natural lattice structures, each of which is a refinement of the one before.

(1) The set of monomials[2] of size n over the predicates, together with constants $\bot$ and $\top$ ($2^n + 2$ elements),
(2) The monomials over the predicates, together with constants $\bot$ ($3^n + 1$ elements), or
(3) The complete Boolean algebra over n Boolean variables (2^{2^n} elements).

In real-time systems, the bottle-neck in model checking is the multitude of distinguishable clock regions that can be associated with the same control location. Consequently we use an extension of (1), where the control structure of the concrete system is preserved, similar to the approach taken in [BLO98]. Our predicates solely range over clock constraints; this entails that our technique is *incremental* in the sense that adding one more predicate to an abstraction yields always a refinement.

[2]Monomials are expressions $\bigwedge_{i \in I \subseteq \{1,\ldots,n\}} l_i$, where each l_i is a literal, i.e., either B_i or $\neg B_i$ for a Boolean predicate B_i.

Since we are concerned with both safety and liveness properties, both over- and under-approximation of the transition relation is required. This has been exemplified in [SS99], where two lattices over Boolean predicates are constructed.

The main problem with applying predicate abstraction in general is to come up with an appropriate set of predicates. For timed systems a set of abstraction predicates expressive enough to distinguish between any two clock regions determines a strongly preserving abstraction. More precisely, a timed system satisfies the property under consideration if and only if the predicate-abstracted system satisfies this property.

The set of abstraction predicates required to compute a strongly preserving abstraction, a so-called *basis*, can still be excessively large. Starting with a trivial over-approximation, we successively select predicates from the finite basis. Counterexamples from failed model checking attempts are used in guiding the selection. The idea of counterexample-guided refinement has been used before by many researchers, and recent work includes [CGJ+00, DD01, LBBO01]). In contrast to these approaches, we use the counterexample only as a heuristic for selecting good pivot predicates from a fixed, predetermined pool of abstraction predicates in order to speed-up convergence of the approximation processes.

7.4 Timed Systems with Restricted Delay Steps

We review some basic notions of transition systems and timed systems. Furthermore, we introduce the notion of time-progressing systems by syntactically restricting the delay steps. These restrictions, however, are not observable in a version of the propositional μ-calculus without a next-step operator. This sets the stage for proving completeness of our abstraction techniques in Section 7.7.

The model of timed system as defined below is motivated by the timed automata model as introduced by Alur, Courcoubetis, and Dill [ACD93].[3] Clocks for measuring time are encoded as variables, which are interpreted over the nonnegative reals $\mathbb{R}_{\geq 0}$. Transitions of timed systems are usually constrained by timing constraints.

Definition 7.5 (Timing Constraints) *Given a set of clocks C, the set of* timing (or clock) constraints *Constr contains* **true**, $x \bowtie m$, *and* $x - y \bowtie m$, *where* $x, y \in C$, $m \in \mathbb{N}$, $\bowtie \in \{\leq, <, =, >, \geq\}$. *The set Inv is the subset of Constr, where $\bowtie$ is chosen from $\{\leq, <\}$. For a positive integer c, Constr(c) is the finite subset of all timing constraints $x \bowtie m$, $x - y \bowtie m$, where $x, y \in C$, $\bowtie \in \{<, \leq, =, \geq, >\}$ and $m \in \{0, \ldots, c\}$.*

[3] For simplicity, we do not treat (synchronized) networks of timed automata, as introduced earlier in Chapter 2. The techniques used in the following can be extended for such networks. The ommission of communication and the representation of variables as control locations simplifies the description.

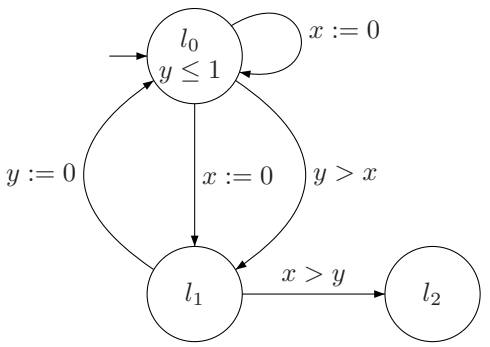


Figure 7.2: Example of a Timed System.

Definition 7.6 (Timed System) *Given a finite set of propositional symbols A, a* timed system $\mathcal{S}$ *is a tuple $\langle L, P, C, T, l_0, I\rangle$, where*

- *L is a nonempty finite set of locations,*
- *$P: L \to 2^A$ maps each location to a set of propositional symbols,*
- *C is a finite set of clocks,*
- *$T \subseteq L \times 2^{Constr} \times 2^C \times L$ is a transition relation,*
- *$l_0 \subseteq L$ is the initial location, and*
- *$I : L \to 2^{Inv}$ assigns a set of downward closed clock constraints to each location l; the elements of $I(l)$ are the* invariants *for location l.*

We write $l \xrightarrow{g,r} l'$ for $\langle l, g, r, l'\rangle \in T$. Firing a transition does not only change the current location but also resets the clocks in r to 0. A transition may only be fired if the timing constraint (guard of the transition) g holds with respect to the current value of the clocks, and if the invariant of the target location is satisfied with respect to the modified value of the clocks.

Example 7.7 *A timed system with three locations l_0, l_1, l_2 and two clocks x, y is displayed in Figure 7.2. The initial location is l_0, transitions are decorated with both timing constraints and clock resets such as $x := 0$. The invariant for location l_0 is $y \leq 1$. Timing constraints that are* **true** *are omitted.*

A function $\nu : C \to \mathbb{R}_{\geq 0}$ is a *clock evaluation*, and the set of clock evaluations is collected in $\mathcal{V}_C$. The clock evaluation $(\nu + d)$ is obtained by adding d to the value of each clock in ν. For $X \subseteq C$, $\nu[X := 0]$ denotes the clock evaluation that updates every clock $x \in X$ to zero, and leaves all the other clock values unchanged. The value $g\nu$ of a clock constraint g with respect to the clock evaluation ν is obtained by substituting the clocks x in g with the corresponding value $\nu(x)$. If $g\nu$ simplifies to the true value, ν satisfies g and we write $\nu \models g$. A set $\mathcal{X} \subseteq \mathcal{V}_C$ of clock evaluations satisfies $g \in Constr$, written as $\mathcal{X} \models g$, if and only if $\nu \models g$ for all $\nu \in \mathcal{X}$. A pair $(l, \nu) \in L \times \mathcal{V}_C$ is called a *timed configuration*, if it satisfies the invariants $I(l)$; formally, $\nu \models I(l)$ if $\nu \models g$ for every invariant $g \in I(l)$. Alur, Courcoubetis, and Dill [ACD93] introduce the fundamental notion of clock regions, which partition the space of possible clock evaluation for a timed automaton into finitely many regions.

Definition 7.8 (Clock Region) *Let $\mathcal{S}$ be a timed system with clocks C and largest constant c, occurring in any timing constraint of $\mathcal{S}$. A* clock region *is a set $\mathcal{X} \subseteq \mathcal{V}_C$ of clock evaluations, such that for all timing constraints $g \in Constr(c)$ and for any two $\nu_1, \nu_2 \in \mathcal{X}$ it is the case that $\nu_1 \models g$ if and only if $\nu_2 \models g$. In this case we write $\nu_1 \equiv_{\mathcal{S}} \nu_2$.*

A *timed step* is either a *delay step*, where time advances by some positive real-valued d, or an instantaneous *state transition step*.

Definition 7.9 (Timed Step) *Let $\mathcal{S}$ be a timed system with clock set C and transition relation T. For $d > 0$, we say that the timed configuration $(l, \nu + d)$ is obtained from (l, ν) by a* delay step *$(l, \nu) \xrightarrow{d} (l, \nu + d)$, if the invariant constraint $\nu + d \models I(l)$ holds. A* state transition step *$(l, \nu) \xrightarrow{g,r} (l', \nu')$ occurs if there exists a $l \xrightarrow{g,r} l' \in T$, and $\nu \models g$, $\nu' = \nu[r := 0]$, and $\nu' \models I(l')$. The union of delay and state transition steps defines the timed transition relation $\Rightarrow$ of a timed system $\mathcal{S}$. Now, a* path *is an infinite or maximally extended finite sequence of configurations $\mathbf{s}_0 \Rightarrow \mathbf{s}_1 \Rightarrow \ldots$.*

Timed systems, as defined above, allow for infinite sequences of delay steps without ever exceeding some given bound. The sequence

$$(l, x = 0) \overset{1/2}{\Longrightarrow} (l, x = 1/2) \overset{1/4}{\Longrightarrow} (l, x = 3/4) \overset{1/8}{\Longrightarrow} (l, x = 7/8) \qquad \cdots \qquad (*)$$

for example, never reaches point in time where $x = 1$. Systems with paths such that an infinite number of steps may happen in a bounded time frame are said to be *zeno*. This kind of behavior is usually ruled out by restricting possible behaviors to non-zeno only. In order to preserve faulty behavior that is caused by an infinite sequence of state transition steps, we use a slightly weaker assumption than non-zenoness. We consider paths which satisfy the following assumption.

Assumption 7.10 (Non-Convergence of Time)
In every infinite sequence of delay steps, the evaluation of every clock eventually exceeds every bound.

In the sequel we build time-abstractions which do not distinguish between state transition steps and delay steps. The main difficulty in defining such abstractions is to prevent delay steps to be abstracted into self-loops on the abstract system.

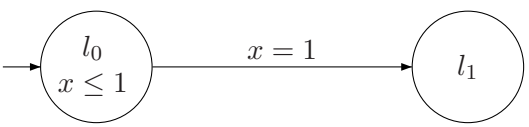
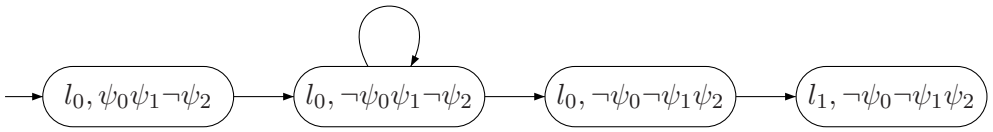


Figure 7.3: Timed System for Example 7.11.

Example 7.11 *Consider the timed system in Figure 7.3. Under the non-convergence assumption this system satisfies the property that location l_1 is always reached. For example, the following sequence is the prefix of a possible path of this system.*

$$(l_0, x = 0) \overset{1/2}{\Longrightarrow} (l_0, x = 1/2) \overset{1/4}{\Longrightarrow} (l_0, x = 3/4) \overset{1/4}{\Longrightarrow} (l_0, x = 1) \overset{x=1,\emptyset}{\Longrightarrow} (l_1, x = 1)$$

We abstract the timed system from Figure 7.3 using the three abstraction predicates $\psi_0 \equiv x = 0$, $\psi_1 \equiv x < 1$, and $\psi_2 \equiv x = 1$. On the abstract system the single state transition step of the timed system is split according to whether or not these predicates hold. For example, in the initial abstract configuration only ψ_0 and ψ_1 hold, since the value of the clock in the initial concrete state is zero. Corresponding to delay steps with delay less than one, there is an abstract transition to a state where only ψ_1 holds. Using small enough delay steps one remains in this state or one reaches a state in which only ψ_2 holds, that is, the clock value is exactly one. A fragment of the resulting abstract transition system is given below.



Notice the self-loop at configuration $(l_0, \neg\psi_0\psi_1\neg\psi_2)$, which has not been present in the concrete system. Due to the loop, it no longer holds for the abstracted system that on every possible path a configuration with location l_1 is reached eventually.

In order to avoid such extraneous self-loops, the non-convergence assumption must somehow be incorporated into the abstract system. Such a restriction, however, can not be defined by means of time delays in the abstract system for the simple reason that there is no notion of time or time delay on this level. In our approach, we enforce the non-convergence assumption explicitly by restricting the model of timed system to delay steps that force a clock to step beyond integer bounds when all fractional clock values are not zero. In this way, the second and third delay step of the path $(*)$ above, for example, are explicitly ruled out.

Definition 7.12 (Restricted Delay Step) *For a timed system $\mathcal{S}$ with clock set C and largest constant c, a* restricted delay step *is a delay step $(l, \nu) \xrightarrow{d} (l, \nu + d)$ for all positive, real-valued d, such that*

$$\exists x \in C.\, \exists k \in \{0, \ldots, c\}.\, \nu(x) = k \ \vee\ (\nu(x) < k \wedge \nu(x) + d \geq k) \tag{7.1}$$

The union of state transition steps and restricted delay steps gives rise to a relation $\Rightarrow_R \subseteq (L, \mathcal{V}_C) \times (L, \mathcal{V}_C)$. Now, a restricted path *is an infinite sequence of configurations $\mathbf{s}_0 \Rightarrow_R \mathbf{s}_1 \Rightarrow_R \ldots$.*

Obviously, it is the case that $\Rightarrow_R$ is a sub-relation of $\Rightarrow$. The restriction of delay steps above does not necessarily enforce time to progress, as is demonstrated by the following restricted path for the system in Example 7.7.

$$(l_0, x = y = 0) \overset{\mathbf{true},\varnothing}{\Longrightarrow} (l_1, x = y = 0) \overset{\mathbf{true},\varnothing}{\Longrightarrow} (l_0, x = y = 0) \overset{\mathbf{true},\varnothing}{\Longrightarrow} (l_1, x = y = 0) \cdots$$

Note that a loop of state transition steps is required in order to prevent the clocks x and y from exceeding the clock value 0.

Corresponding to the non-convergence assumption on timed paths and the restricted delay steps we associate two semantics for timed systems in terms of transition systems. The natural semantics $\mathcal{M}$ includes arbitrary delay steps under the non-convergence of time assumption, while the restricted semantics $\mathcal{M}_R$ includes only restricted delay steps as in Definition 7.12.

Definition 7.13 (Semantics of a Timed System) *Let $\mathcal{S} = \langle L, P, C, T, l_0, I\rangle$ be a timed system. We associate two transition systems $\mathcal{M}$ and $\mathcal{M}_R$ with $\mathcal{S}$ as follows.*

$$\begin{aligned} \mathcal{M} &:= \langle L \times \mathcal{V}_C, P,\ (\Rightarrow),\ (l_0,\ \nu_0)\,\rangle \\ \mathcal{M}_R &:= \langle L \times \mathcal{V}_C, P,\ (\Rightarrow_R),\ (l_0,\ \nu_0)\,\rangle \end{aligned}$$

The symbol ν_0 denotes the special clock evaluation, that maps every clock to 0. $\mathcal{M}$ is called the natural semantics *of $\mathcal{S}$, and $\mathcal{M}_R$ is referred to as the* restricted semantics *of $\mathcal{S}$.*

We demonstrate that the restriction of delay steps does not change the possible observations of the model with respect to μ-calculus formulas without next-step operators.

7.4.1 The Next-Free μ-Calculus

The μ-calculus [Koz83] is a branching-time temporal logic. where formulas are built from atomic propositions, boolean connectives, the least-fixpoint operator, and the next-step operator $\bigcirc$. $\bigcirc\varphi$ expresses that there is a successor satisfying φ. We remove the next-step operator, since we do not want to distinguish between one delay step of duration, say, 1 and two subsequent delay steps of durations 2/5 and 3/5. Both alternatives should be considered to be observationally equivalent. Logics without explicit next-step operator have also been considered, for example, by Dams [Dam96] and Tripakis and Yovine [TY01].

Definition 7.14 (Next-Free μ-Calculus) *Let A be a set of atomic predicates, and Var be a set of variables; then, for $p \in A$ and $Z \in$ Var, the set $\mathcal{L}_\mu$ of next-free μ-calculus formulas is described by the following grammar.*

$$\varphi ::= tt \mid p \mid \varphi \wedge \varphi \mid \neg\varphi \mid \varphi \forall U \varphi \mid \varphi \exists U \varphi \mid Z \mid \mu Z.\varphi(Z)$$

In addition, every variable is assumed to appear under an even number of negations. A sentence *is a formula without free variables.*

Intuitively, an *existential until* formula $\varphi_1 \exists U \varphi_2$ holds in some configuration $\mathbf{s}$ iff φ_1 holds until φ_2 holds on some path starting from $\mathbf{s}$. Similarly, a *universal until*

formula $\varphi_1 \forall U \varphi_2$ holds in $\mathbf{s}$ if this conditions holds for all paths from $\mathbf{s}$. We use the abbreviations:

ff	$:= \neg \mathit{tt}$	false
$\nu Z.\varphi(Z)$	$:= \neg \mu Z.\neg \varphi(\neg Z)$	greatest fixpoint
$\Diamond^* \varphi$	$:= \mathit{tt} \exists U \varphi$	on some path, φ holds eventually
$\Box^* \varphi$	$:= \mathit{tt} \forall U \varphi$	for all paths, φ holds eventually

Given a transition system $\mathcal{M} = \langle S, P, \Rightarrow_R, \mathbf{s}_0 \rangle$, the semantics of a next-free μ-calculus sentence is given by the set of timed configurations $\mathbf{s} = (l, \nu)$ for which the formula holds. Sub-formulas containing free variables $Z \in \mathit{Var}$ are dealt with by a *valuation function* $\vartheta : \mathit{Var} \to 2^S$. The updating notation $\vartheta[Z := S']$ denotes the valuation ϑ' that agrees with ϑ on all variables except Z, where $\vartheta'(Z) = S' \subseteq S$.

Definition 7.15 (Semantics of the Next-Free μ-Calculus) *Given a transition system $\mathcal{M} = \langle S, P, \Rightarrow_R, \mathbf{s}_0 \rangle$ over the set $S = L \times \mathcal{V}_C$ of timed configurations and an assignment $\vartheta : \mathit{Var} \to 2^S$, the set of configurations $[\![\varphi]\!]^{\mathcal{M}}_{\vartheta}$ validating a formula $\varphi \in \mathcal{L}_\mu$ with respect to ϑ is defined inductively on the structure of φ.*

$$
\begin{aligned}
[\![\mathit{tt}]\!]^{\mathcal{M}}_{\vartheta} &:= S \\
[\![p]\!]^{\mathcal{M}}_{\vartheta} &:= \big\{ (l,\nu) \in S \,\big|\, p \in P(l) \big\} \\
[\![\varphi_1 \wedge \varphi_2]\!]^{\mathcal{M}}_{\vartheta} &:= [\![\varphi_1]\!]^{\mathcal{M}}_{\vartheta} \cap [\![\varphi_2]\!]^{\mathcal{M}}_{\vartheta} \\
[\![\neg \varphi]\!]^{\mathcal{M}}_{\vartheta} &:= S \setminus [\![\varphi]\!]^{\mathcal{M}}_{\vartheta} \\
[\![\varphi_1 \exists U \varphi_2]\!]^{\mathcal{M}}_{\vartheta} &:= \{ \mathbf{s}_0 \in S \,\big|\, \textit{there exists a path } \sigma = (\mathbf{s}_0 \Rightarrow \mathbf{s}_1 \Rightarrow \ldots),\ \textit{s.t.} \\
&\qquad\qquad \mathbf{s}_i \in [\![\varphi_2]\!]^{\mathcal{M}}_{\vartheta} \ \textit{for some } i \geq 0, \\
&\qquad\qquad \textit{and for all } 0 \leq j < i,\ \mathbf{s}_j \in [\![\varphi_1]\!]^{\mathcal{M}}_{\vartheta} \} \\
[\![\varphi_1 \forall U \varphi_2]\!]^{\mathcal{M}}_{\vartheta} &:= \{ \mathbf{s}_0 \in S \,\big|\, \textit{for every path } \sigma = (\mathbf{s}_0 \Rightarrow \mathbf{s}_1 \Rightarrow \ldots), \\
&\qquad\qquad \textit{there exists } i \geq 0,\ \textit{s.t. } \mathbf{s}_i \in [\![\varphi_2]\!]^{\mathcal{M}}_{\vartheta}, \\
&\qquad\qquad \textit{and for all } 0 \leq j < i,\ \mathbf{s}_j \in [\![\varphi_1]\!]^{\mathcal{M}}_{\vartheta} \} \\
[\![Z]\!]^{\mathcal{M}}_{\vartheta} &:= \vartheta(Z) \\
[\![\mu Z.\varphi]\!]^{\mathcal{M}}_{\vartheta} &:= \bigcap \Big\{ \mathcal{E} \subseteq S \,\Big|\, [\![\varphi]\!]^{\mathcal{M}}_{\vartheta[Z:=\mathcal{E}]} \subseteq \mathcal{E} \Big\}
\end{aligned}
$$

We write $\mathcal{M}, \mathbf{s}, \vartheta \models \varphi$ to denote that $\mathbf{s} \in [\![\varphi]\!]^{\mathcal{M}}_{\vartheta}$. The subscript ϑ is omitted whenever φ is a sentence.

Two configurations are said to be indistinguishable if they satisfy the same set of $\mathcal{L}_\mu$ sentences.

Definition 7.16 (μ-Equivalence) *For a transition system $\mathcal{M}$, two configurations $\mathbf{s}, \mathbf{s}'$ are* μ-equivalent*, denoted by $\mathbf{s} \equiv_{\mathcal{M}} \mathbf{s}'$, if for every sentence $\varphi \in \mathcal{L}_\mu$: $\mathbf{s} \in [\![\varphi]\!]^{\mathcal{M}}$ if and only if $\mathbf{s}' \in [\![\varphi]\!]^{\mathcal{M}}$.*

The binary relation $\equiv_{\mathcal{M}}$ is indeed an equivalence relation on clock evaluations. Moreover, μ-equivalence characterizes clock regions in the sense that two clock valuations are in the same clock region if and only if they are μ-equivalent. Consequently, μ-equivalence is of finite index.

Lemma 7.17 (Region Equivalence)
Let $\mathcal{S}$ be a timed system with clock set C and largest constant c, and let $\mathcal{M}$ be the corresponding natural transition system. Then for all $l \in L$ and clock evaluations $\nu, \nu' \in \mathcal{V}_C$ with $\nu \equiv_{\mathcal{S}} \nu'$ the time configurations (l, ν) and (l, ν') are μ-equivalent, that is, $(l, \nu) \equiv_{\mathcal{M}} (l, \nu')$.

Proof: Following Definition 7.16 two time configurations (l, ν) and (l, ν') are μ-equivalent if and only if

$$\forall\, \varphi \in \mathcal{L}_\mu.\, (l, \nu) \in [\![\, \varphi \,]\!]^{\mathcal{M}} \Leftrightarrow (l, \nu') \in [\![\, \varphi \,]\!]^{\mathcal{M}}$$

For arbitrary sentences $\varphi \in \mathcal{L}_\mu$ we show $(l, \nu) \in [\![\, \varphi \,]\!]^{\mathcal{M}}$ if and only if $(l, \nu') \in [\![\, \varphi \,]\!]^{\mathcal{M}}$. The proof works by a straightforward structural induction on φ.

$\boxed{\varphi = tt}$ From Definition 7.15 we have $[\![\, tt \,]\!]^{\mathcal{M}} = S$. Therefore $(l, \nu) \in [\![\, tt \,]\!]^{\mathcal{M}}$ and $(l, \nu') \in [\![\, tt \,]\!]^{\mathcal{M}}$.

$\boxed{\varphi = p}$ Also following Definition 7.15 we obtain $[\![\, p \,]\!]^{\mathcal{M}} = \{\tilde{l}, \tilde{\nu} \mid p \in \mathcal{P}(l)\}$. And since the configurations (l, ν) and (l, ν') have the same locations, both are contained in $[\![\, p \,]\!]^{\mathcal{M}}$.

$\boxed{\varphi = \varphi_1 \ \&\ \varphi_2}$ By Definition 7.15 we have that $[\![\, \varphi_1 \ \&\ \varphi_2 \,]\!]^{\mathcal{M}} = [\![\, \varphi_1 \,]\!]^{\mathcal{M}} \cap [\![\, \varphi_2 \,]\!]^{\mathcal{M}}$, and by Induction Hypothesis it follows

$$\begin{array}{rcl} (l, \nu) \in [\![\, \varphi_1 \,]\!]^{\mathcal{M}} & \Leftrightarrow & (l, \nu') \in [\![\, \varphi_1 \,]\!]^{\mathcal{M}} \ \text{ and} \\ (l, \nu) \in [\![\, \varphi_2 \,]\!]^{\mathcal{M}} & \Leftrightarrow & (l, \nu') \in [\![\, \varphi_2 \,]\!]^{\mathcal{M}} \end{array}$$

Thus, $(l, \nu) \in [\![\, \varphi_1 \,]\!]^{\mathcal{M}} \cap [\![\, \varphi_2 \,]\!]^{\mathcal{M}} \Leftrightarrow (l, \nu') \in [\![\, \varphi_1 \,]\!]^{\mathcal{M}} \cap [\![\, \varphi_2 \,]\!]^{\mathcal{M}}$.

$\boxed{\varphi = \neg\varphi_1}$ By Definition 7.15 we have that $[\![\, \neg\varphi_1 \,]\!]^{\mathcal{M}} = S \setminus [\![\, \varphi_1 \,]\!]^{\mathcal{M}}$. By Induction Hypothesis we obtain $(l, \nu) \notin [\![\, \varphi_1 \,]\!]^{\mathcal{M}} \Leftrightarrow (l, \nu') \notin [\![\, \varphi_1 \,]\!]^{\mathcal{M}}$, and therefore $(l, \nu) \in [\![\, \neg\varphi_1 \,]\!]^{\mathcal{M}} \Leftrightarrow (l, \nu') \in [\![\, \neg\varphi_1 \,]\!]^{\mathcal{M}}$.

$\boxed{\varphi = \varphi_1\, \exists U\, \varphi_2}$ By Definition 7.15
$[\![\, \varphi_1\, \exists U\, \varphi_2 \,]\!]^{\mathcal{M}} =$
$\{(l_0, \nu_0) \in S \mid$ there exists a path $\sigma = ((l_0, \nu_0) \Rightarrow (l_1, \nu_1) \Rightarrow \ldots),$ s.t.
$(l_i, \nu_i) \in [\![\, \varphi_2 \,]\!]^{\mathcal{M}}$ for some $i \geq 0,$ and for all $0 \leq j < i,\ (l_j, \nu_j) \in [\![\, \varphi_1 \,]\!]^{\mathcal{M}}\}$
By Induction Hypothesis we have that $(l, \nu'_i) \in [\![\, \varphi_2 \,]\!]^{\mathcal{M}}$ and $(l, \nu'_j) \in [\![\, \varphi_1 \,]\!]^{\mathcal{M}}$, for all

$0 \leq j < i$.

From the assumption $\nu \equiv_{\mathcal{S}} \nu'$ it follows by Definitions 7.8 and 7.9 that there exists a path $\tau' = ((l_0, \nu'_0) \Rightarrow (l_1, \nu'_1) \Rightarrow \ldots)$ such that $(l_i, \nu'_i) \in [\![\varphi_2]\!]^{\mathcal{M}}$ for some $i \geq 0$, and $(l_j, \nu'_j) \in [\![\varphi_2]\!]^{\mathcal{M}}$ for all $0 \leq j < i$. Thus, $(l, \nu') \in [\![\varphi_1 \, \exists U \varphi_2]\!]^{\mathcal{M}}$.

$\boxed{\varphi = \varphi_1 \, \forall U \varphi_2}$ By Definition 7.15

$[\![\varphi_1 \, \forall U \varphi_2]\!]^{\mathcal{M}} \;=$

$\{(l, \nu) \in S \mid$ for every path $\sigma = ((l_0, \nu_0) \Rightarrow (l_1, \nu_1) \Rightarrow \ldots)$ with $(l_0, \nu_0) = (l, \nu)$, there exists $i \geq 0$ s.t. $(l_i, \nu_i) \in [\![\varphi_2]\!]^{\mathcal{M}}$, and for all $0 \leq j < i$, $(l_j, \nu_j) \in [\![\varphi_1]\!]^{\mathcal{M}}\}$

By Induction Hypothesis we have that $(l, \nu'_i) \in [\![\varphi_2]\!]^{\mathcal{M}}$ and $(l, \nu'_j) \in [\![\varphi_1]\!]^{\mathcal{M}}$, for all $0 \leq j < i$.

From the assumption $\nu \equiv_{\mathcal{S}} \nu'$ it follows by Definitions 7.8 and 7.9 that for all paths $\tau' = ((l_0, \nu'_0) \Rightarrow (l_1, \nu'_1) \Rightarrow \ldots)$ there exists $i \geq 0$ such that $(l_i, \nu'_i) \in [\![\varphi_2]\!]^{\mathcal{M}}$, and $(l_j, \nu'_j) \in [\![\varphi_2]\!]^{\mathcal{M}}$ for all $0 \leq j < i$. Thus, $(l, \nu') \in [\![\varphi_1 \, \forall U \varphi_2]\!]^{\mathcal{M}}$.

$\boxed{\varphi = \mu Z.\varphi_1}$ Assume $(l, \nu) \in [\![\mu Z.\varphi_1]\!]^{\mathcal{M}}_{\vartheta}$, that is, by Definition 7.15

$$(l, \nu) \in \bigcap \Big\{ \mathcal{E} \subseteq S \,\Big|\, [\![\varphi_1]\!]^{\mathcal{M}}_{\vartheta[Z:=\mathcal{E}]} \subseteq \mathcal{E} \Big\}$$

By Induction Hypothesis it follows that $(l, \nu') \in [\![\mu Z.\varphi_1]\!]^{\mathcal{M}}_{\vartheta}$. ///

We now show that the natural semantics and the restricted semantics of a timed system as introduced in Definition 7.13 are indistinguishable in the next-free μ-calculus. Intuitively, sentences in $\mathcal{L}_\mu$ can not distinguish quantitative values of clocks, and therefore all configurations with identical control locations and μ-equivalent clock evaluations satisfy the same set of $\mathcal{L}_\mu$ sentences.

Theorem 7.18 *Let $\mathcal{S}$ be a timed system with clocks C, largest constant c, natural semantics $\mathcal{M}$, and restricted semantics $\mathcal{M}_R$. Under the non-convergence assumption for $\mathcal{M}$, for every sentence $\varphi \in \mathcal{L}_\mu$:*

$$[\![\varphi]\!]^{\mathcal{M}} \quad = \quad [\![\varphi]\!]^{\mathcal{M}_R}$$

Proof: The proof works by structural induction on the formula φ, where we strengthen the claim to $[\![\varphi]\!]^{\mathcal{M}}_{\vartheta} = [\![\varphi]\!]^{\mathcal{M}_R}_{\vartheta}$ for arbitrary valuation functions ϑ.

$\boxed{\varphi = tt}$ By Definition 7.15, $[\![tt]\!]^{\mathcal{M}}_{\vartheta} \stackrel{def}{=} S \stackrel{def}{=} [\![tt]\!]^{\mathcal{M}_R}_{\vartheta}$

$\boxed{\varphi = p}$ By Definition 7.15, $[\![p]\!]^{\mathcal{M}}_{\vartheta} \stackrel{def}{=} \{(l, \nu) \mid p \in P(l)\} \stackrel{def}{=} [\![p]\!]^{\mathcal{M}_R}_{\vartheta}$

Induction Hypothesis: assume we already established $[\![\varphi']\!]^{\mathcal{M}}_{\vartheta} = [\![\varphi']\!]^{\mathcal{M}_R}_{\vartheta}$ for all subformulas φ' of φ and all valuation functions ϑ.

$\boxed{\varphi = \varphi_1 \wedge \varphi_2}$ By Definition 7.15 and by Induction Hypothesis we have that

$$[\![\, \varphi_1 \wedge \varphi_2 \,]\!]_\vartheta^{\mathcal{M}} \stackrel{def}{=} [\![\, \varphi_1 \,]\!]_\vartheta^{\mathcal{M}} \cap [\![\, \varphi_2 \,]\!]_\vartheta^{\mathcal{M}} \stackrel{I.H.}{=} [\![\, \varphi_1 \,]\!]_\vartheta^{\mathcal{M}_R} \cap [\![\, \varphi_2 \,]\!]_\vartheta^{\mathcal{M}_R} \stackrel{def}{=} [\![\, \varphi_1 \wedge \varphi_2 \,]\!]_\vartheta^{\mathcal{M}_R}$$

$\boxed{\varphi = \neg\varphi_1}$ By Definition 7.15 and by Induction Hypothesis we have that

$$[\![\, \neg\varphi_1 \,]\!]_\vartheta^{\mathcal{M}} \stackrel{def}{=} S \setminus [\![\, \varphi_1 \,]\!]_\vartheta^{\mathcal{M}} \stackrel{I.H}{=} S \setminus [\![\, \varphi_1 \,]\!]_\vartheta^{\mathcal{M}_R} \stackrel{def}{=} [\![\, \neg\varphi_1 \,]\!]_\vartheta^{\mathcal{M}_R}$$

$\boxed{\varphi = \varphi_1\, \exists U\, \varphi_2}$ According to Definition 7.15, $\mathbf{s} \in [\![\, \varphi_1\, \exists U\, \varphi_2 \,]\!]_\vartheta^{\mathcal{M}}$ iff there exists an path starting at $\mathbf{s}$ such that

$$\mathbf{s}_i \in [\![\, \varphi_2 \,]\!]_\vartheta^{\mathcal{M}} \text{ for some } i \geq 0, \text{and for all } 0 \leq j < i,\ \mathbf{s}_j \in [\![\, \varphi_1 \,]\!]_\vartheta^{\mathcal{M}} \qquad (**)$$

Since every path in the restricted semantics is also a path in the natural semantics, it suffices to show that for every path in the natural semantics which validates $(**)$, there exists a path in the restricted semantics which also validates $(**)$. First, we show that a delay step in $\Rightarrow \setminus \Rightarrow_R$ does not step across the border of any region. Let $(l, \nu) \xrightarrow{d} (l, \nu + d)$ be a delay step in $\mathcal{M}$ but not in $\mathcal{M}_R$. Then by Definition 7.12:

$$\begin{array}{ll} & \neg\,(\exists x \in C.\ \exists k \in \{0, \ldots, c\}.\ \nu(x) = k \ \vee\ (\nu(x) < k \wedge \nu(x) + d \geq k)) \\ \Leftrightarrow & \forall x \in C.\ \forall k \in \{0, \ldots, c\}.\ (\nu(x) \neq k \ \wedge\ (\nu(x) < k \ \Rightarrow\ \nu(x) + d < k)) \\ \Leftrightarrow & \forall x \in C.\ \lfloor \nu(x) \rfloor \ <\ \nu(x),\ \nu(x) + d \ <\ \lfloor \nu(x) + 1 \rfloor \end{array}$$

Consequently, it is the case that $\nu \equiv_{\mathcal{S}} (\nu + d)$. Using Lemma 7.17, for $(\mathbf{s}, \mathbf{s}') \in \Rightarrow \setminus \Rightarrow_R$ it holds that $\mathbf{s} \equiv_{\mathcal{M}} \mathbf{s}'$. Now, consider a finite path $\sigma = (\mathbf{s}_1 \Rightarrow \ldots \Rightarrow \mathbf{s}_i)$ with $\mathbf{s}_i \in [\![\, \varphi_2 \,]\!]_\vartheta^{\mathcal{M}}$ and $\forall\, 1 \leq j < i.\ \mathbf{s}_j \in [\![\, \varphi_1 \,]\!]_\vartheta^{\mathcal{M}}$. We transform this path σ to a restricted path σ_R by removing the steps not contained in $\Rightarrow_R$ and by merging adjacent delays. Using Lemma 7.17, all $\mathbf{s}_{e+f} = (l_{e+f}, \nu_{e+f})$ with $l_{e+f} = l_e$ and $\nu_{e+f} \equiv_{\mathcal{S}} \nu_e$, are μ-equivalent, that is, $(l_{e+f}, \nu_{e+f}) \equiv_{\mathcal{M}} (l_e, \nu_e)$. Removing all $\mathbf{s}_{e+f}$ with $f \geq 1$ from σ yields the sub-path

$$\tau_R = (\mathbf{s}_1 = \mathbf{s}_{k_1} \Rightarrow_R \mathbf{s}_{k_2} \cdots \Rightarrow_R \mathbf{s}_{k_m} = \mathbf{s}_i),\ k_h \in \{1, \ldots, i\},\ k_h < k_{h+1}$$

such that $\mathbf{s}_{k_m} \in [\![\, \varphi_2 \,]\!]_\vartheta^{\mathcal{M}}$ and for all $h < m$, $\mathbf{s}_{k_h} \in [\![\, \varphi_1 \,]\!]_\vartheta^{\mathcal{M}}$. By induction hypothesis, $\mathbf{s}_{k_m} \in [\![\, \varphi_2 \,]\!]_\vartheta^{\mathcal{M}_R}$ and for all $h < m$, $\mathbf{s}_{k_h} \in [\![\, \varphi_1 \,]\!]_\vartheta^{\mathcal{M}_R}$. Since both guards and invariants are timing constraints in *Constr*, they have identical truth values for the clock evaluations of $\mathbf{s}_e$ and $\mathbf{s}_{e+f}$. Thus every step $\mathbf{s}_{k_1} \Rightarrow_R \mathbf{s}_{k_2}$ is indeed possible according to the restricted semantics, and σ_R is a restricted path. Thus $[\![\, \varphi_1\, \exists U\, \varphi_2 \,]\!]_\vartheta^{\mathcal{M}} = [\![\, \varphi_1\, \exists U\, \varphi_2 \,]\!]_\vartheta^{\mathcal{M}_R}$.

$\boxed{\varphi = \varphi_1\, \forall U\, \varphi_2}$ According to Definition 7.15, $\mathbf{s} \in [\![\, \varphi_1\, \forall U\, \varphi_2 \,]\!]_\vartheta^{\mathcal{M}}$ iff for all paths starting at $\mathbf{s}$ the following holds:

$$\mathbf{s}_i \in [\![\, \varphi_2 \,]\!]_\vartheta^{\mathcal{M}} \text{ for some } i \geq 0, \text{and for all } 0 \leq j < i,\ \mathbf{s}_j \in [\![\, \varphi_1 \,]\!]_\vartheta^{\mathcal{M}} \qquad (***)$$

Every path in the restricted semantics is also a path in the natural semantics. We have to establish that if a path in the natural semantics violates the condition (∗∗∗), then also a path in the restricted semantics does.

Assume a path $\sigma = \mathbf{s}_1 \Rightarrow \mathbf{s}_2 \Rightarrow \cdots$ in the natural semantics, that *violates* the condition

$$(\star) \quad := \quad \exists i.\, \mathbf{s}_i \in [\![\, \varphi_2 \,]\!]^{\mathcal{M}}_{\vartheta} \text{ and for all } 1 \leq j < i,\, \mathbf{s}_j \in [\![\, \varphi_1 \,]\!]^{\mathcal{M}}_{\vartheta}\,.$$

Now we show that there exists also a path $\sigma_R = (\mathbf{s} = \mathbf{s}_{k_1} \Rightarrow_R \mathbf{s}_{k_2} \Rightarrow_R \cdots)$ in the restricted semantics that violates $(\star)$.

If σ is either finite or contains infinitely many state transition steps, then—by the same argument as in the previous case—there exists also a sub-path σ_R of σ, where no two subsequent configurations have clock evaluations in the same region and σ_R violates $(\star)$.

Suppose σ is infinite and contains only finitely many state transition steps. By the non-convergence assumption it cannot contain an infinite suffix of delay steps, without exceeding the largest constant c for every clock at some point $\mathbf{s}_k$. By Lemma 7.17, $\mathbf{s}_k \equiv_{\mathcal{M}} \mathbf{s}_{k'}$ for all $k' \geq k$. Therefore, if σ violates $(\star)$, then already the finite prefix $\mathbf{s}_1 \Rightarrow \cdots \Rightarrow \mathbf{s}_k$ does. For this finite prefix we can construct a sub-path σ_R according to the restricted semantics as before.

Thus $[\![\, \varphi_1 \,\forall U\, \varphi_2 \,]\!]^{\mathcal{M}}_{\vartheta} = [\![\, \varphi_1 \,\forall U\, \varphi_2 \,]\!]^{\mathcal{M}_R}_{\vartheta}$.

$\boxed{\varphi = Z}$ By Definition 7.15 it follows $[\![\, Z \,]\!]^{\mathcal{M}}_{\vartheta} \stackrel{def}{=} \vartheta(Z) \stackrel{def}{=} [\![\, Z \,]\!]^{\mathcal{M}_R}_{\vartheta}$.

$\boxed{\varphi = \mu Z.\varphi_1}$ By Definition 7.15 and Induction Hypothesis it follows

$$\begin{aligned} [\![\, \mu Z.\varphi_1 \,]\!]^{\mathcal{M}}_{\vartheta} &\stackrel{def}{=} \bigcap \left\{ \mathcal{E} \subseteq S \,\middle|\, [\![\, \varphi_1 \,]\!]^{\mathcal{M}}_{\vartheta[Z:=\mathcal{E}]} \subseteq \mathcal{E} \right\} \stackrel{I.H}{=} \\ &\stackrel{I.H}{=} \bigcap \left\{ \mathcal{E} \subseteq S \,\middle|\, [\![\, \varphi_1 \,]\!]^{\mathcal{M}_R}_{\vartheta[Z:=\mathcal{E}]} \subseteq \mathcal{E} \right\} \stackrel{def}{=} [\![\, \mu Z.\varphi_1 \,]\!]^{\mathcal{M}_R}_{\vartheta}\,. \end{aligned}$$

□

This result allows us to focus on the restricted semantics of timed systems only, since any result expressible in $\mathcal{L}_\mu$ for the restricted semantics $\mathcal{M}_R$ also holds for the natural semantics $\mathcal{M}$. In the sequel we omit the indices R; thus the system $\mathcal{M}$ and the transition relation $\Rightarrow$ denote a restricted system and a restricted transition relation, respectively.

7.5 Predicate Abstraction for Real-Time Systems

We adopt predicate abstraction for dense real-time systems, while focusing on the approximation of clock regions.

Definition 7.19 (Abstraction Predicates) *Given a set of clocks C, an* abstraction predicate *with respect to C is any formula with the set of free variables*

in C. Similarly to timing constraints, the value of an abstraction predicate ψ with respect to a clock evaluation ν, where both free and bound variables are interpreted in the domain C, is denoted by the juxtaposition $\psi\nu$. Whenever $\psi\nu$ evaluates to tt, we write $\nu \models\!\!\approx \psi$.

A set of abstraction predicates $\Psi = \{\psi_0, \cdots, \psi_{n-1}\}$ determines an abstraction function α, which maps clock valuations ν to a *bit-vector* b of length n, such that the i-th component of b is set if and only if ψ_i holds for ν. Here, we assume that bit-vectors of length n are elements of the set $\mathbb{B}_n$, which are functions of domain $\{0, \cdots, n-1\}$ and codomain $\{0, 1\}$. The inverse image of α, that is, the concretization function γ, maps a bit-vector to the set of clock valuations which satisfy all ψ_i whenever the i-th component of the bit-vector is set. Thus, a set of concrete states (l, ν) is transformed by the abstraction function α into the abstract state $(l, \alpha(\nu))$, and an abstract state (l, b) is mapped by γ to a set of concrete states $(l, \gamma(b))$.

Definition 7.20 (Abstraction/Concretization) *Let C be a set of clocks and $\mathcal{V}_C$ the corresponding set of clock valuations. Given a finite set of predicates $\Psi = \{\psi_0, \cdots, \psi_{n-1}\}$, the* abstraction function $\alpha : L \times \mathcal{V}_C \to L \times \mathbb{B}_n$ *is defined by*

$$\alpha(l, \nu)(i) := (l, \psi_i\nu)$$

and the concretization function $\gamma : L \times \mathbb{B}_n \to L \times 2^{\mathcal{V}_C}$ *is defined by*

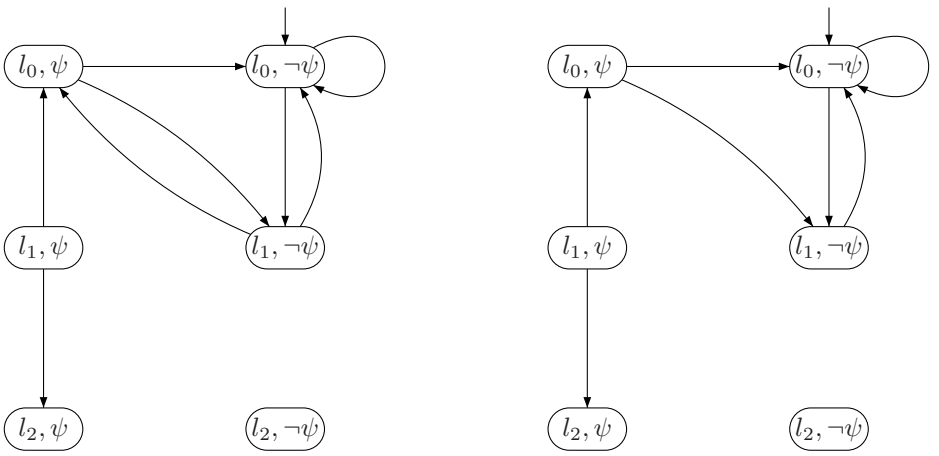
$$\gamma(l, b) := \{(l, \nu) \in L \times \mathcal{V}_C \mid I(l) \wedge \bigwedge_{i=0}^{n-1} \psi_i\nu \equiv b(i)\}.$$

We use the notations $\alpha(S) := \{\alpha(l, \nu) \mid (l, \nu) \in S\}$ and $\gamma(S^{\mathcal{A}}) := \{\gamma(l, b) \mid (l, b) \in S^{\mathcal{A}}\}$. Now, the abstraction/concretization pair (α, γ) forms a Galois connection. If l is clear from the context, we use the notation $\gamma(b)$ to denote the set of clock evaluations in $\gamma(l, b)$.

Definition 7.21 (Over-/Under-Approximation) *Given a (concrete) transition system $\mathcal{M} = \langle S^C, P, \Rightarrow, s_0^C \rangle$, where $S^C = L \times \mathcal{V}_C$ and $s_0^C = (l_0, \nu_0)$, and a set Ψ of abstraction predicates, we construct two (abstract) transition systems $\mathcal{M}_\Psi^+ = \langle S^{\mathcal{A}}, P, \Rightarrow^+, s_0^{\mathcal{A}} \rangle$, and $\mathcal{M}_\Psi^- = \langle S^{\mathcal{A}}, P, \Rightarrow^-, s_0^{\mathcal{A}} \rangle$.*

- $S^{\mathcal{A}} := L \times \mathbb{B}_n$,
- $(l, b) \Rightarrow^+ (l', b')$ *iff* $\exists \nu \in \gamma(b).\ \exists \nu' \in \gamma(b').\ (l, \nu) \Rightarrow (l', \nu')$,
- $(l, b) \Rightarrow^- (l', b')$ *iff* $\forall \nu \in \gamma(b).\ \exists \nu' \in \gamma(b').\ (l, \nu) \Rightarrow (l', \nu')$, *and*
- $\mathbf{s}_0^{\mathcal{A}} := (l_0, b_0)$, *where* $b_0(i) = 1$ *iff* $\nu_0 \models \psi_i$.

$\mathcal{M}_\Psi^+$ is called an over-approximation *of $\mathcal{M}$, $\mathcal{M}_\Psi^-$ is an* under-approximation *of $\mathcal{M}$.*



(i) Over-approximation *(ii)* Under-approximation

Figure 7.4: Approximation of the Timed System from Figure 7.2 With $\psi \equiv x > y$.

Since $\gamma(b) \neq \emptyset$, we have that $\Rightarrow^- \subseteq \Rightarrow^+$.

Example 7.22 *Figure 7.4 shows the over- and under-approximation of the (concrete) system from Figure 7.2 with respect to the predicate set* $\Psi = \{x > y\}$.

For the transition relations $\Rightarrow^-$ and $\Rightarrow^+$ we define $\gamma(\Rightarrow^-)$, respectively $\gamma(\Rightarrow^+)$ as follows:

$$\gamma(\Rightarrow^-) := \{((l,\nu),(l',\nu')) \in S^C \mid \exists b, b'.\ (l,b) \Rightarrow^- (l',b') \wedge \nu \in \gamma(b) \wedge \nu' \in \gamma(b')\}$$
$$\gamma(\Rightarrow^+) := \{((l,\nu),(l',\nu')) \in S^C \mid \exists b, b'.\ (l,b) \Rightarrow^+ (l',b') \wedge \nu \in \gamma(b) \wedge \nu' \in \gamma(b')\}$$

Lemma 7.23 *For a (concrete) transition system* $\mathcal{M}$ *with the transition relation* $\Rightarrow$ *and the corresponding over- and under-approximations* $\mathcal{M}^+_\Psi$, $\mathcal{M}^-_\Psi$ *with respective transition relations* $\Rightarrow^+$, $\Rightarrow^-$ *it is the case that*

1. $\gamma(\Rightarrow^-) \subseteq \Rightarrow \subseteq \gamma(\Rightarrow^+)$, *and*
2. $\Rightarrow^- \subseteq \alpha(\Rightarrow) \subseteq \Rightarrow^+$.

Proof: Follows from Definition 7.21. ///

Definition 7.24 (Predicate Abstraction) *Let* $\mathcal{M} = \langle S^C, P, \Rightarrow, s_0^C\rangle$ *be a transition system with corresponding over-approximation* $\mathcal{M}^+_\Psi = \langle S^{\mathcal{A}}, P, \Rightarrow^+, \mathbf{s}_0^{\mathcal{A}}\rangle$, *and under-approximation* $\mathcal{M}^-_\Psi = \langle S^{\mathcal{A}}, P, \Rightarrow^-, \mathbf{s}_0^{\mathcal{A}}\rangle$, *as given in Definition 7.21. Then, the* predicate abstracted *semantics* $[\![\,\varphi\,]\!]^{\mathcal{M}^\varsigma_\Psi}_\vartheta$, *where* ς *is either* $+$ *or* $-$, *of a formula* $\varphi \in \mathcal{L}_\mu$ *with respect to a valuation function* ϑ *and the finite transition systems* $\mathcal{M}^\varsigma_\Psi$ *is defined in a mutually inductive way. The notation* $\overline{\varsigma}$ *is used to toggle the sign* ς.

$$
\begin{aligned}
[\![\mathit{tt}]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma} &:= S^{\mathcal{A}} \\
[\![p]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma} &:= \{(l,b) \in S^{\mathcal{A}} \mid p \in P(l)\} \\
[\![\varphi_1 \wedge \varphi_2]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma} &:= [\![\varphi_1]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma} \cap [\![\varphi_2]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma} \\
[\![\neg\varphi]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma} &:= S^{\mathcal{A}} \setminus [\![\varphi]\!]_{\vartheta}^{\mathcal{M}_\Psi^{\overline{\varsigma}}} \\
[\![\varphi_1 \,\exists U\, \varphi_2]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma} &:= \{s_0 \in S^{\mathcal{A}} \mid \textit{there exists a path } \sigma = (s_0 \Rightarrow^{\varsigma} s_1 \Rightarrow^{\varsigma} \ldots), \\
&\qquad \textit{s.t. } s_i \in [\![\varphi_2]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma} \textit{ for some } i \geq 0, \textit{ and} \\
&\qquad \textit{for all } 0 \leq j < i,\ s_j \in [\![\varphi_1]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma}\} \\
[\![\varphi_1 \,\forall U\, \varphi_2]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma} &:= \{s_0 \in S^{\mathcal{A}} \mid \textit{for every path } \sigma = (s_0 \Rightarrow^{\overline{\varsigma}} s_1 \Rightarrow^{\overline{\varsigma}} \ldots), \\
&\qquad \textit{there exists } i \geq 0, \textit{s.t. } s_i \in [\![\varphi_2]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma}, \\
&\qquad \textit{and for all } 0 \leq j < i,\ s_j \in [\![\varphi_1]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma}\} \\
[\![Z]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma} &:= \vartheta(Z) \\
[\![\mu Z.\varphi]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma} &:= \bigcap \{\mathcal{E} \subseteq S^{\mathcal{A}} \mid [\![\varphi]\!]_{\vartheta[Z:=\mathcal{E}]}^{\mathcal{M}_\Psi^\varsigma} \subseteq \mathcal{E}\}
\end{aligned}
$$

We also write $\mathcal{M}_\Psi^\varsigma, (l,b), \vartheta \models^{\mathcal{A}} \varphi$, *to denote that* $(l,b) \in [\![\varphi]\!]_{\vartheta}^{\mathcal{M}_\Psi^\varsigma}$.

Theorem 7.25 (Soundness of Abstraction)
Let $\mathcal{M} = \langle S^C, P, \Rightarrow, \mathbf{s}_0^C \rangle$ *be a transition system,* Ψ *a set of abstraction predicates, and* $\mathcal{M}_\Psi^+$, $\mathcal{M}_\Psi^-$ *the over-approximation and under-approximation of* $\mathcal{M}$ *with respect to* Ψ. *Let* γ *be the concretization function with respect to* Ψ. *Then for any sentence* $\varphi \in \mathcal{L}_\mu$ *the following holds:*

$$\gamma([\![\varphi]\!]^{\mathcal{M}_\Psi^-}) \subseteq [\![\varphi]\!]^{\mathcal{M}} \subseteq \gamma([\![\varphi]\!]^{\mathcal{M}_\Psi^+})$$

Proof: The proof is by induction on the structure of φ. We show here only the case $\varphi = \varphi_1 \,\exists U\, \varphi_2$. By induction hypothesis we have that

$$\gamma([\![\varphi_1]\!]^{\mathcal{M}_\Psi^-}) \subseteq [\![\varphi_1]\!]^{\mathcal{M}} \subseteq \gamma([\![\varphi_1]\!]^{\mathcal{M}_\Psi^+})$$

and

$$\gamma([\![\varphi_2]\!]^{\mathcal{M}_\Psi^-}) \subseteq [\![\varphi_2]\!]^{\mathcal{M}} \subseteq \gamma([\![\varphi_2]\!]^{\mathcal{M}_\Psi^+}).$$

Let $\rightsquigarrow_\gamma^\varsigma$, $\rightsquigarrow_\alpha$ denote the transition relations $\gamma(\Rightarrow^\varsigma)$, $\alpha(\Rightarrow)$ respectively.

$$
\begin{aligned}
&\gamma([\![\varphi_1 \,\exists U\, \varphi_2]\!]^{\mathcal{M}_\Psi^-}) = \\
&\quad = \ /\star \text{ by Definition 7.24 } \star/ \\
&\qquad \gamma(\{s_0 \in S^{\mathcal{A}} \mid \text{there exists a path } \sigma = (s_0 \Rightarrow^- s_1 \Rightarrow^- \ldots), \text{ s.t.} \\
&\qquad\quad s_i \in [\![\varphi_2]\!]^{\mathcal{M}_\Psi^-} \text{ for some } i \geq 0, \text{ and for all } 0 \leq j < i,\ s_j \in [\![\varphi_1]\!]^{\mathcal{M}_\Psi^-}\})
\end{aligned}
$$

$=$ /⋆ by Definition 7.24 ⋆/
$\{s \in \gamma(S^{\mathcal{A}}) \mid$ there exists a path $\sigma = (\gamma(s_0) \rightsquigarrow^-_\gamma \gamma(s_1) \rightsquigarrow^-_\gamma \ldots)$
with $s = \gamma(s_0)$,
and $\rightsquigarrow^-_\gamma = \gamma(\Rightarrow^-)$, s.t. $\gamma(s_i) \in \gamma([\![\varphi_2]\!]^{\mathcal{M}^-_\Psi})$ for some $i \geq 0$,
and for all $0 \leq j < i$, $\gamma(s_j) \in \gamma([\![\varphi_1]\!]^{\mathcal{M}^-_\Psi})\}$

$=$ /⋆ by induction hypothesis ⋆/
$\{s \in \gamma(S^{\mathcal{A}}) \mid$ there exists a path $\sigma = (\gamma(s_0) \rightsquigarrow^-_\gamma \gamma(s_1) \rightsquigarrow^-_\gamma \ldots)$
with $s = \gamma(s_0)$,
and $\rightsquigarrow^-_\gamma = \gamma(\Rightarrow^-)$, s.t. $\gamma(s_i) \in [\![\varphi_2]\!]^{\mathcal{M}}$ for some $i \geq 0$,
and for all $0 \leq j < i$, $\gamma(s_j) \in [\![\varphi_1]\!]^{\mathcal{M}}\}$

$\subseteq$ /⋆ by Definition 7.21 and Lemma 7.23 ⋆/
$\{s^c \in S^C \mid$ there exists a path $\sigma = (s^c_0 \Rightarrow s^c_1 \Rightarrow \ldots)$ with $s^c = s^c_0$,
and $s^c_i \in \gamma(s_i)$, for all $i \geq 1$, s.t. $s^c_i \in [\![\varphi_2]\!]^{\mathcal{M}}$ for some $i \geq 0$,
and for all $0 \leq j < i$, $s^c_j \in [\![\varphi_1]\!]^{\mathcal{M}}\}$

$\subseteq$ /⋆ by Definition 7.15 ⋆/
$[\![\varphi_1 \exists U \varphi_2]\!]^{\mathcal{M}}$

$[\![\varphi_1 \exists U \varphi_2]\!]^{\mathcal{M}} =$

$=$ /⋆ by Definition 7.15 ⋆/
$\{s^c_0 \in S^C \mid$ there exists a path $\sigma = (s^c_0 \Rightarrow s^c_1 \Rightarrow \ldots)$, s.t.
$s^c_i \in [\![\varphi_2]\!]^{\mathcal{M}}$ for some $i \geq 0$, and
for all $0 \leq j < i$, $s^c_j \in [\![\varphi_1]\!]^{\mathcal{M}}$

$\subseteq$ /⋆ by Definitions 7.20 and 7.21 ⋆/
$\gamma(\{s^a \in \alpha(S^C) \mid$ there exists a path $\sigma = (s^a_0 \rightsquigarrow_\alpha s^a_1 \rightsquigarrow_\alpha \ldots)$
with $s^a = s^a_0$,
and $\rightsquigarrow_\alpha = \alpha(\Rightarrow)$, and $s^a_i = \alpha(s_i)$, for all $i \geq 1$, s.t.
$s^a_i \in [\![\varphi_2]\!]^{\mathcal{M}^+_\Psi}$ for some $i \geq 0$, and
for all $0 \leq j < i$, $s^a_j \in [\![\varphi_1]\!]^{\mathcal{M}^+_\Psi}\})$

$$
\begin{aligned}
\subseteq\ & /\star \text{ by Lemma 7.23-2 and since } \gamma \text{ is monotone } \star/ \\
& \gamma(\{s^a \in \alpha(S^C) \mid \text{there exists a path } \sigma = (s_0^a \Rightarrow^+ s_1^a \Rightarrow^+ \ldots) \\
& \qquad \text{with } s^a = s_0^a, \\
& \qquad \text{and } s_i^a = \alpha(s_i), \text{ for all } i \geq 1, \text{ s.t. } s_i^a \in [\![\, \varphi_2 \,]\!]^{\mathcal{M}_\Psi^+} \text{ for some } i \geq 0, \\
& \qquad \text{and for all } 0 \leq j < i,\ s_j^a \in [\![\, \varphi_1 \,]\!]^{\mathcal{M}_\Psi^+}\}) \\
=\ & /\star \text{ by Definition 7.24 } \star/ \\
& \gamma([\![\, \varphi_1 \,\exists U\, \varphi_2 \,]\!]^{\mathcal{M}_\Psi^+})
\end{aligned}
$$

□

Example 7.26 *Consider our running example in Figure 7.2, for which we want to verify that location l_2 is never reached. This property is expressed by the μ-calculus formula*

$$\varphi := \neg tt\, \exists U\, at\_l_2$$

where $at\_l_2 \in A$ is a (Boolean) proposition that is true iff the system is in location l_2. The over-approximation of $\mathcal{M}$ with respect to the abstraction predicate $\psi \equiv (x > y)$ is shown in Figure 7.4. According to Definition 7.24, the set of abstract states of $\mathcal{M}_{\{\psi\}}^+$ which validate φ is given by

$$[\![\, \neg tt\, \exists U\, at\_l_2 \,]\!]^{\mathcal{M}_{\{\psi\}}^+} = S^{\mathcal{A}} \setminus [\![\, tt\, \exists U\, at\_l_2 \,]\!]^{\mathcal{M}_{\{\psi\}}^-} = \{(l_0, \psi), (l_0, \neg\psi), (l_1, \neg\psi)\}\ .$$

Since the initial state $(l_0, \neg\psi)$ of $\mathcal{M}_{\{\psi\}}^+$ is contained in this set, the formula φ holds on the abstract transition system. Thus, $\mathcal{M}_{\{\psi\}}^+, (l_0, b_0) \models^{\mathcal{A}} \varphi$ holds. By Theorem 7.25, property φ also holds on the concrete transition system, $\mathcal{M}, (l_0, \nu_0) \models \varphi$.

An interesting aspect of this example is that the over- and under-approximations with respect to only a single abstraction predicate ψ already coincide. We now give a criterion, based on the notion of regions, for a set of abstraction predicates, which is sufficient for guaranteeing convergence in general.

7.6 Sets of Basis Predicates

A basis is a set of abstraction predicates that is expressive enough to distinguish between two clock regions. If a basis is used for predicate abstraction, then the approximation is exact with respect to the next-free μ-calculus.

Definition 7.27 (Basis) *Let $\mathcal{S}$ be a timed system with clock set C and let Ψ be a set of abstraction predicates. Then Ψ is a* basis *with respect to $\mathcal{S}$ iff for all clock evaluations $\nu_1, \nu_2 \in \mathcal{V}_C$*

$$(\forall \psi \in \Psi.\ \nu_1 \mathrel{|\!\approx} \psi \Leftrightarrow \nu_2 \mathrel{|\!\approx} \psi) \quad \textit{implies} \quad \nu_1 \equiv_{\mathcal{S}} \nu_2 \quad .$$

For example, for a timed system $\mathcal{S}$ with clock set C and largest constant c, the (infinite) set of clock constraints *Constr*, the (infinite) set of invariant constraints *Inv*, the (finite) set of clock constraints $Constr(c)$ with largest constant c as the largest constant of $\mathcal{S}$, and the (finite) set of membership predicates for the quotient $\mathcal{V}_C$ modulo $\equiv_{\mathcal{S}}$ are all basis sets. Since the set of predicates $Constr(c)$ is finite, there is a finite basis for every timed automaton. Notice, however, that this basis is not necessarily minimal.

Example 7.28 *The set* $\Psi := \{x = 0, y = 0, x \leq 1, x \geq 1, y \leq 1, y \geq 1, x > y, x < y\}$ *is a basis for the timed system in Figure 7.2.*

Theorem 7.29 *Let* $\mathcal{S}$ *be a timed system with clock set* C *and largest constant* c*, and* $\mathcal{M}$ *the corresponding transition system. Let* Ψ *be a basis with respect to* $\mathcal{S}$*, and* $\mathcal{M}^-_\Psi$*,* $\mathcal{M}^+_\Psi$ *the under- and over-approximation of* $\mathcal{M}$ *with respect to* Ψ*. Then, for any sentence* $\varphi \in \mathcal{L}_\mu$*,*

$$[\![\varphi]\!]^{\mathcal{M}^-_\Psi} \quad = \quad [\![\varphi]\!]^{\mathcal{M}^+_\Psi}.$$

Proof: Since it suffices to show that $\Rightarrow^- \supseteq \Rightarrow^+$, we assume two configurations (l, b) and (l', b') such that $(l, b) \Rightarrow^+ (l', b')$. According to Definition 7.21, there exist $\nu \in \gamma(b)$ and $\nu' \in \gamma(b')$ such that $(l, \nu) \Rightarrow (l', \nu')$.

In case $(l, \nu) \Rightarrow (l', \nu')$ holds due to a state transition step, all the guards g of some transition $l \xrightarrow{g,r} l'$ evaluate to the true value for $\nu \in \gamma(b)$. Since Ψ is a basis, the guards then evaluate to the true value for all clock evaluations $\tilde{\nu} \in \gamma(b)$ (Definition 7.27). The clock values at ν' are either identical to ν, or are reset to 0 (for clocks $x \in r$). Thus for all clock evaluations $\tilde{\nu} \in \gamma(b)$, the state transition step $l \xrightarrow{g,r} l'$ can be applied and leads to a clock evaluation $\tilde{\nu}'$, such that $\tilde{\nu}' \equiv_{\mathcal{S}} \nu' \wedge (l, \tilde{\nu}) \Rightarrow (l', \tilde{\nu}')$. Then, by Definition 7.21, $(l, b) \Rightarrow^- (l', b')$.

In case $(l, \nu) \Rightarrow (l', \nu')$ holds due to a delay step, then $l = l'$ and the invariants $I(l)$ evaluate to the true value for $\nu' \in \gamma(b')$. Since invariant expressions are taken from *Inv*, by Definition 7.27, the invariants evaluate to the true value for all $\tilde{\nu}' \in \gamma(b')$. Moreover, Definition 7.12 requires that ν and ν' are not in the same region, and thus a delay step according to the restricted semantics is possible. Consequently, at location l, for all $\tilde{\nu} \in \gamma(b)$, a delay step to some $\tilde{\nu}' \in \gamma(b')$ is possible. Again, by Definition 7.21, $(l, b) \Rightarrow^- (l', b')$. □

Corollary 7.30 (Basis Completeness) *Let* $\mathcal{S} = \langle L, P, C, T, l_0, I \rangle$ *be a timed system,* $\mathcal{M} = \langle L \times \mathcal{V}_C, P, \Rightarrow, (l_0, \nu_0) \rangle$ *the corresponding transition system, let* Ψ *be a basis for* $\mathcal{S}$*, and let* $\gamma(b_0) = \nu_0$*. Then for any sentence* $\varphi \in \mathcal{L}_\mu$*:*

$$(l_0, b_0) \in [\![\varphi]\!]^{\mathcal{M}^-_\Psi} \quad \Leftrightarrow \quad (l_0, \nu_0) \in [\![\varphi]\!]^{\mathcal{M}} \quad \Leftrightarrow \quad (l_0, b_0) \in [\![\varphi]\!]^{\mathcal{M}^+_\Psi} \ .$$

Proof: By Theorem 7.25, $\gamma([\![\varphi]\!]^{\mathcal{M}^-_\Psi}) \subseteq [\![\varphi]\!]^{\mathcal{M}} \subseteq \gamma([\![\varphi]\!]^{\mathcal{M}^+_\Psi})$. By Theorem 7.29, $[\![\varphi]\!]^{\mathcal{M}^-_\Psi} = [\![\varphi]\!]^{\mathcal{M}^+_\Psi}$, and thus $\gamma([\![\varphi]\!]^{\mathcal{M}^-_\Psi}) = \gamma([\![\varphi]\!]^{\mathcal{M}^+_\Psi})$. □

7.7 Refinement of the Abstraction

Given a concrete model $\mathcal{M}$ of a timed system, a finite basis Ψ of abstraction predicates, and a formula φ. We present an algorithm for computing an over-approximation of $\mathcal{M}$ that is sufficient to prove or refute the model checking problem $\mathcal{M} \models \varphi$. The algorithm starts with a rough approximation of $\mathcal{M}$ based on a subset of basis predicates. This approximation converges towards an exact representation via stepwise refinement.

The abstraction-refinement algorithm is displayed in Figure 7.5. The variables Ψ_{new} and Ψ_{act} store the currently unused and used abstraction predicates, respectively. Initially Ψ_{act} contains a subset Ψ' of predicates from the basis, and Ψ_{new} contains the remaining predicates (lines (2)-(4) in Figure 7.5).

First over- and under-approximation of $\mathcal{M}$ with respect to the set Ψ_{act} is computed. Now it is checked whether the approximation suffices to establish validity of the formula φ. This is done by calling a finite-state μ-calculus model checker. If the approximation can guarantee that initial state $\mathbf{s}_0$ is in $[\![\varphi]\!]^{\varphi}_{\mathcal{M}}$, then our algorithm returns **true** (line (6)). Otherwise $\mathbf{s}_0 \notin \gamma\big([\![\varphi]\!]^{\mathcal{M}^{-}_{\Psi_{act}}}\big)$ and the μ-calculus model checker returns a counterexample in form of an abstract path (see [Kic96]). If for the abstract path there exists a corresponding path in the concrete transition system, then we get a counterexample for the concrete model checking problem (lines (9)-(12)). In this case the algorithm returns **false**.

This check requires an off-the-shelf satisfiability-checker for the boolean combination of linear arithmetic constraints such as ICS [FORS01]. In case the abstract counterexample is spurious, there exists a smallest index k and a concrete path $y_0 \Rightarrow \cdots \Rightarrow y_k$, where y_0 is the initial location of $\mathcal{M}$, and for all $i \in \{0, \cdots, k\}$, $y_i \in \gamma(\mathbf{s}_i)$, such that there is no (concrete) transition from y_k to y_{k+1}, where $y_{k+1} \in \gamma(\mathbf{s}_{k+1})$ (lines (13)-(16)). We choose a minimal set of new abstraction predicates from Ψ_{new} such that the transition from $\mathbf{s}_k$ to $\mathbf{s}_{k+1}$ is eliminated (lines (17)-(20)). This new set of abstraction predicates is chosen in such a way that the formula

$$\exists\, y_1, y_2 \in S^C.\, y_1 \in \gamma(\mathbf{s}_k) \;\wedge\; y_2 \in \gamma(\mathbf{s}_{k+1}) \;\wedge\; y_1 \not\Rightarrow y_2$$

holds. Notice that the concretization function γ depends on the current set Ψ_{act} of abstraction predicates.

Theorem 7.31 (Termination, Soundness, and Completeness)
Let $\mathcal{M}$ be a transition system with a corresponding finite basis Ψ, and φ a sentence in $\mathcal{L}_\mu$. Then the algorithm in Figure 7.5 always terminates. Moreover, if it terminates with **true**, *then $\mathcal{M} \models \varphi$, and if the result is* **false**, *then $\mathcal{M} \not\models \varphi$.*

Proof: By Theorem 7.25 it follows that a given answer is correct. Let n be the cardinality of the basis Ψ. Every execution of the loop (line 4) adds at least one new predicate from the basis (line 17). After at most n iterations we have $[\![\varphi]\!]^{\mathcal{M}^{-}_{\Psi_{act}}} = [\![\varphi]\!]^{\mathcal{M}^{+}_{\Psi_{act}}}$ by Theorem 7.29. Then the algorithm necessarily terminates, since either φ can be established or a concrete counter-example can be derived. □

Algorithm: *abstract_and_refine*
input: $\mathcal{M}$, φ, basis Ψ
output: answer to model checking query $\mathcal{M} \stackrel{?}{\models} \varphi$

```
1   CHOOSE Ψ' = {ψ_1, ..., ψ_i} from Ψ
2   Ψ_new := Ψ \ Ψ'
3   Ψ_act := Ψ'
4   LOOP
5     IF s_0 ∈ γ( [[ φ ]]^{M⁻_{Ψ_act}} )
6       THEN RETURN true
7       ELSE LET (s_0 ⇒⁺ s_1 ··· ⇒⁺ s_n) be a counterexample
8            IF ∃ path σ = (y_0 ⇒ y_1 ··· ⇒ y_n)
9               such that y_0 = s_0^C and y_i ∈ γ(s_i) for all 0 ≤ i ≤ n
10              THEN RETURN false
11              ELSE LET k be s.t. ∃ path σ = (y_0 ⇒ y_1 ··· ⇒ y_k)
12                   where y_0 = s_0^C and
13                         y_i ∈ γ(s_i) for all 0 ≤ i ≤ k and
14                         ∀ y_{k+1} ∈ γ(s_{k+1}). y_k ↛ y_{k+1}
15                   CHOOSE minimal Ψ' = {ψ_1, ..., ψ_i} from Ψ_new with
16                         ∀ y_1 ∈ γ(s_k), y_2 ∈ γ(s_{k+1}). y_1 ↛ y_2
17                   Ψ_act := Ψ_act ∪ Ψ'
18                   Ψ_new := Ψ_new \ Ψ'
19  ENDLOOP
```

Figure 7.5: Iterative Abstraction-Refinement Algorithm.

Example 7.32 *Consider again the timed system from Figure 7.2, and the formula* $\varphi := \neg tt\, \exists U\, at\_l_2$ *which describes the property that location* l_2 *is never reached. A given basis for this system is* $\Psi := \{x = 0, y = 0, x \leq 1, x \geq 1, y \leq 1, y \geq 1, x > y, x < y\}$. *The transition system of the initial over-approximation with the single abstraction predicate* $\{x = 0\}$ *is shown in Figure 7.6, where* ψ_0 *denotes the abstraction predicate.*

Here $\mathbf{s}_0 = (l_0, x = y = 0) \notin \gamma\big([\![\varphi]\!]^{\mathcal{M}^-_{\{x=0\}}} \big)$. *Instead a counterexample can be derived via* $\mathbf{s}_0 \in \gamma\big([\![\neg\varphi]\!]^{\mathcal{M}^+_{\{x=0\}}} \big) = \gamma\big([\![tt\, \exists U\, at\_l_2]\!]^{\mathcal{M}^+_{\{x=0\}}} \big)$.

$$(l_0, \psi_0) \Rightarrow^+ (l_1, \psi_0) \Rightarrow^+ (l_0, \neg\psi_0) \Rightarrow^+ (l_1, \neg\psi_0) \Rightarrow^+ (l_2, \neg\psi_0)$$

which is emphasized in Figure 7.6 using lines in bold face. The concretizations of the states on this abstract path are as follows. To simplify the notation we denote sets of

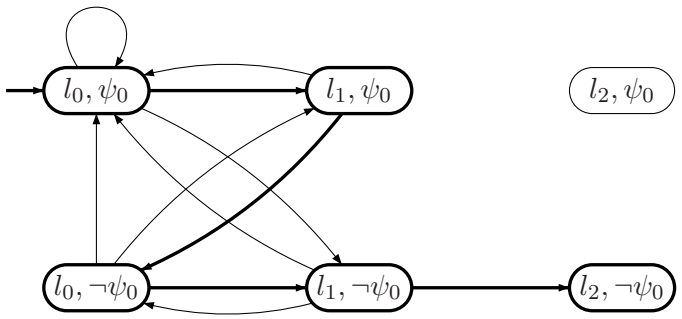


Figure 7.6: Over-Approximation of the System From Figure 7.2 With $\psi_0 \equiv x = 0$.

configurations such as $\{(l, \nu) \mid l = l_1 \wedge \nu(x) = 0 \wedge \nu(y) \geq 0\}$ *by* $(l_1,\ x = 0 \wedge y \geq 0)$.

$$\begin{array}{rcl}
\gamma(s_0) & = & \gamma((l_0, \psi_0)) \ = \ (l_0, \gamma(\psi_0)) \ = \ (l_0,\ x = 0 \ \wedge \ y \geq 0) \\
\gamma(s_1) & = & \gamma((l_1, \psi_0)) \ = \ (l_1, \gamma(\psi_0)) \ = \ (l_1,\ x = 0 \ \wedge \ y \geq 0) \\
\gamma(s_2) & = & \gamma((l_0, \neg\psi_0)) \ = \ (l_0, \gamma(\neg\psi_0)) \ = \ (l_0,\ x > 0 \ \wedge \ y \geq 0) \\
\gamma(s_3) & = & \gamma((l_1, \neg\psi_0)) \ = \ (l_1, \gamma(\neg\psi_0)) \ = \ (l_1,\ x > 0 \ \wedge \ y \geq 0) \\
\gamma(s_4) & = & \gamma((l_2, \neg\psi_0)) \ = \ (l_2, \gamma(\neg\psi_0)) \ = \ (l_2,\ x > 0 \ \wedge \ y \geq 0)
\end{array}$$

Now we have to check if there is a corresponding counterexample on the concrete transition system. If there exists a path $y_0 \Rightarrow y_1 \Rightarrow y_2 \Rightarrow y_3 \Rightarrow y_4$, *where* $y_0,\ y_1,\ y_2,\ y_3,\ y_4 \in S^C$, *such that* $y_0 \in \gamma(\mathbf{s}_0)$, $y_1 \in \gamma(\mathbf{s}_1)$, $y_2 \in \gamma(\mathbf{s}_2)$, $y_3 \in \gamma(\mathbf{s}_3)$, $y_4 \in \gamma(\mathbf{s}_4)$, *and* $y_0 = \mathbf{s}_0^C$. *This is the case if the formula*

$$\begin{array}{rl}
F_1 := \exists\, y_0, y_1, y_2, y_3, y_4 \in S^C. & \\
& y_0 \in \gamma(s_0) \ \wedge \ y_1 \in \gamma(s_1) \ \wedge \ y_2 \in \gamma(s_2) \ \wedge \ y_3 \in \gamma(s_3) \ \wedge \ y_4 \in \gamma(s_4) \ \wedge \\
& y_1 \Rightarrow y_2 \ \wedge \ y_2 \Rightarrow y_3 \ \wedge \ y_3 \Rightarrow y_4 \ \wedge \\
& y_0 = s_0^C
\end{array}$$

is valid. In our example, F_1 *is unsatisfiable, since on the concrete transition system there is no transition between* y_3 *and* y_4, *as it is illustrated by the following path.*

$$\underbrace{(l_0, x = y = 0)}_{\exists y_0} \Rightarrow \underbrace{(l_1, x = 0 \wedge 0 \leq y \leq 1)}_{\exists y_1} \Rightarrow \underbrace{(l_0, x > 0 \wedge y \leq 1 \wedge x \geq y)}_{\exists y_2} \Rightarrow \underbrace{(l_1, x > 0 \wedge y > x)}_{\exists y_3}$$

Thus, $k = 3$ *in our algorithm, and we choose a new set of abstraction predicates such that there exist concrete configurations* $y_1,\ y_2 \in S^C$ *with* $y_1 \in \gamma(\mathbf{s}_3)$ *and* $y_2 \in \gamma(\mathbf{s}_4)$ *such that there is no transition from* y_1 *to* y_2. *For example, by choosing the new abstraction predicate* $\psi_1 \equiv x > y$ *the formula* $\exists\, y_1, y_2 \in S^C. y_1 \in \gamma(\mathbf{s}_3) \ \wedge \ y_2 \in \gamma(\mathbf{s}_4) \ \wedge \ y_1 \not\Rightarrow y_2$ *can be shown to hold using a verification procedure for this decidable fragment of arithmetic. Figure 7.7 shows the reachable fragment of the resulting*

over-approximation $\mathcal{M}^{+}_{\{\psi_0,\psi_1\}}$. *Testing for* $\mathbf{s}_0 \stackrel{?}{\in} [\![\neg tt \exists U \, at\_l_2]\!]_{\mathcal{M}^{-}_{\Psi_{act}}}$ *succeeds, since no state* $(l_2, \_)$ *is reachable in* $\mathcal{M}^{+}_{\{\psi_0,\psi_1\}}$.

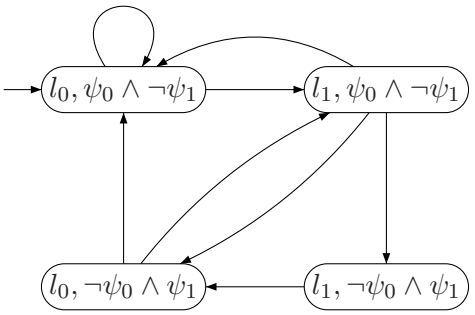


Figure 7.7: Over-Approximation (Reachable Part) of the Timed System From Figure 7.2 With $\Psi = \{x = 0, x > y\}$.

7.8 Reflection: Abstractions of Real-Time Systems

We have presented a verification algorithm for timed automata based on predicate abstraction, un-timed model checking, and decision procedures for the Boolean combination of linear arithmetic constraints. The main advantage of this approach is that bisimilar time-abstractions are computed lazily. This can result in substantial savings in computation, whenever coarse abstractions are sufficient to prove the property at hand. Initial investigations are encouraging in that standard benchmark examples for timed systems such as the train-gate controller and a version of the Fischer mutual exclusion protocol can generally be proved using only a few abstraction predicates. Such an observation has already been made by Alur, Itai, Kurshan, and Yannakakis [AIKY95] in a similar context. However, more experimentation is needed to corroborate the conjecture that many real-life timed systems can already be verified with rather coarse-grain abstractions.

Related Work

Our introduction of a non-convergence assumption can be seen as a syntactic way of enforcing fairness in the system. The main purpose is to get rid of self-loops in the abstracted system that are caused by infinitesimal delay steps in the concrete one. Uribe [Uri98] distinguishes between three different approaches in the literature for building fairness into the abstraction: first, by adding new fairness constraints to the abstract system, second, by incorporating fairness into the logic, and third, by modifying the finite-state model checker. Thus our restriction of the delay steps constitutes a fourth approach. Our restriction is a manifestation of a

weaker assumption than non-zenoness. With respect to next-free μ-calculus, natural semantics *plus* the non-convergence assumption and restricted semantics without additional assumptions are equivalent.

The algorithm as described in this paper is restricted to deal with real-time systems with finite control only. The predicate abstraction of timed systems, however, can readily be extended to also apply to richer models such as parameterized timed automata and even to timed automata with other infinite data types such as counters or stacks. The price to pay, of course, is that such extensions are necessarily incomplete.

Dill and Wong-Toi [DWT95] also use an iteration of both over- and under-approximations of the reachable state set of timed automata, but their techniques are limited to proving invariants. Based on techniques of predicate abstraction, Namjoshi and Kurshan's algorithm [NK00] computes a finite bisimulation whenever it exists. Thus, in principle, their algorithm could be applied to compute finite bisimulations of timed automata. Currently it is unclear, however, if their approach is applicable in practice, since there is no explicitly stated upper bound on the number of abstraction rounds and abstraction predicates needed for convergence. In contrast, for the special case of timed automata, we are able to predetermine a finite set of abstraction predicates. Tripakis and Yovine [TY01] show how to abstract dense real-time in order to obtain time-abstracting, finite bisimulations. Whenever it suffices to compute rather coarse abstractions, we expect to obtain much smaller transition systems by means of predicate abstraction and refinement of predicate abstractions.

The main challenge in successful real-time abstraction seems to be the wise choice of good predicates. While for data abstraction a rich set of experiences condensed into default mechanisms, the choices for real-time systems seem to be wide open.

Part III

Making Use of Hierarchical Structure

Remember that there are two alleyways to formal verification: The one is foundation, the other is application.

— Anders Peter Ravn

There is a limit on the number of concepts any one person can keep in mind at any given time. Hierarchical structures allow a human designer to organize her understanding of a system. Conceptually lower levels in a hierarchy correspond to a higher level of detail. Parts that are currently in her focus can be zoomed in, while others remain on an abstract level.

It is a justified hope that not only humans, but also algorithms can benefit from the presence of structural information. We address this in Chapter 8, where we incrementally build a hierarchy over an initial set of unstructured atoms. The dependencies between atoms guide us in this construction. We compare hierarchies with each other by means of a heuristic cost function. We use the model checking tool Mocha *to perform a state space exploration on different hierarchies over the same atoms. Experiments on three benchmark examples suggest that hierarchies with low cost also yield better performance of the model checking engine.*

The second line of work goes the opposite way: from hierarchical structures to flat ones. For the implementer of an analytic algorithm, hierarchies add to complication. It is easier to focus on a small number of core functionalities with a limited number of dependencies. Often one can rely on the fact that more sophisticated—or more usable—concepts can be encoded in terms of these core functionalities. Together with an automated translation, the higher-level concepts can then be added as syntactic sugar.

In Chapter 9 we follow this approach for the formal verification of the hierarchical timed automata language (introduced in Chapter 3). We report on construction and implementation of an algorithm that flattens a hierarchical model to a Uppaal *timed automata model. This allows us to make use of the model checking engine of* Uppaal*. We sketch a proof of semantic correspondence between hierarchical and flattened model.*

As case study we use the model of a cardiac pacemaker, known as a standard UML example [Dou99a]. The run-time data we get for model checking suggests that the overhead introduced by the flattening is tolerable.

Chapter 8

Hierarchical Partitioning

All parts should go together without forcing.
You must remember that the parts you are reassembling were disassembled by you.
Therefore, if you can't get them together again, there must be a reason.
By all means, do not use a hammer.

— IBM maintenance manual, 1925

Hierarchical partitioning is the successive grouping of a set of components, starting with a set of atoms, and ending with one single compound. This can be seen as the construction of a rooted tree, where the leaf-nodes are known before and every intermediate node has two or more successors.

The number of possible choices suffers a considerable combinatorial explosion, but many of them can be refuted a priori as unreasonable. In this Chapter, we strive to "discover" good hierarchies. We give a cost measure that allows us to compare hierarchical partitions, whenever the means of connection can be adequately described by a hypergraph. Determining the best structure for this measure is NP-complete.

We present a greedy polynomial-time algorithm that approximates good hierarchical partitions by local evaluation of a heuristic function. We corroborate applicability and usefulness via three case studies with our implementation of this algorithm in the model checker Mocha. *When applied to a tree-shaped topology, this results in significant time- and memory-savings. For leader election in a ring and a opinion poll protocol, the run-time performance is not drastically improved.*

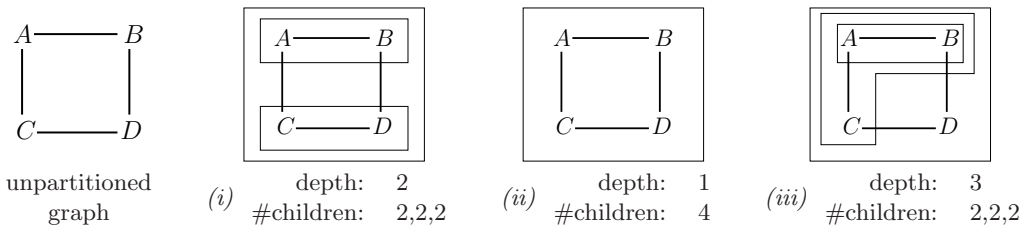


Figure 8.1: Different Ways to Hierarchically Partition a Square.

8.1 How to Group Together?

Imposing a hierarchical structure on a collection of components is helpful in many contexts for different reasons, such as better understanding and better analysis. The process can be understood as grouping together a set of items to a new item that hides the element of the group. This step is repeated, until a single item remains.

Consider four items, call them A, B, C, and D. They may be connected in some way, say by a mutual dependency. Let us assume that this gives rise to a ring structure, like in Figure 8.1. Then, instead of viewing the system as a *set* with 4 elements, we can understand it structured as $\{\{A, B\}, \{C, D\}\}$, like in *(i)*. Here, only two connections A-C and B-D need to be visible (or understood) at the top level. With the new compounds $\{A, B\}$ and $\{C, D\}$ we found a more abstract description of the same data that can be refined on demand. Another possible partition is $\{\{A, D\}, \{B, C\}\}$, but this requires all connections to be visible at the top-level, and thus should be rejected in favor of the first. We can partition in a hierarchic fashion: for example, $\{\{\{A, B\}, C\}, D\}$. In general, given a set, we partition it, and apply the process recursively to each set in the partition.

The set of distinguishable *hierarchical partitions* is adequately described as the set of rooted trees over leaf nodes $\{A, B, C, D\}$. In Figure 8.1, we draw these trees as cascading polygons that may contain other polygons. Every polygon corresponds to an intermediate node and the outermost polygon to the root. As a rule, we favor trees that have a low degree of branching and are nevertheless shallow. The diagrams *(ii)* and *(iii)* depict both not very good trees, since they are either too broad or too deep. Moreover, it is desirable to minimize dependencies among remote tree parts, i.e., the number of links crossing polygon boundaries should be low.

We can regard this as a general design problem, where trees form an architectural hierarchy over atomic units. This *modular* description helps to see the same system on different levels of abstraction or detail. The emphasis on modularity and hierarchy is a central theme in software engineering, particularly in software design notations such as Statecharts [Har87] and UML [BRJ99]. While the most appropriate hierarchical structure can best be chosen by the designer, automatically constructing a hierarchical partition is required if no manually chosen structure is available, or if the original structure is lost during translations between models (e.g., during the process of abstraction).

Formal verification is a field where structure is particularly useful, since it is

generally considered infeasible to deal with unorganized descriptions. Structure helps to spot design flaws, but it can also be exploited to make algorithmic treatment more efficient, or even possible at all. Well-known examples for this are model checking problems. *Model checking* [CE82, CK96, Hol97] is a powerful technique for discovering inconsistencies in high-level designs in hardware and communication protocols. Since it typically requires search in the global state-space, much research aims at providing heuristics to make this step less time- and space-consuming.

Consider once more the example with the four components. Let us interpret each atom as a process and each connection as the ability to synchronize on some action. We view the system hierarchically decomposed as in Figure 8.1 *(i)*. Tools such as the concurrency workbench [CPS93] can analyze it in the following way. First take the product of processes A and B. Now their synchronization can be viewed as internal to this composite process, and we can apply a reduction based on weak bisimulation minimizing the size. Analogously, compose C with D and minimize. The obtained description is still adequate, since it shows *the same behavior* (modulo the internal synchronization), but questions about this behavior are algorithmically easier to answer.

An alternative method that benefits from a hierarchical structure is implemented in a recent version of the model checking tool MOCHA, and will form the basis of the experiments in this paper. The technique, called *"Next" heuristic*, is a heuristic for on-the-fly search based on compressing unobservable transitions to a single meta-transition [AW99]. The basic idea is to describe the implementation P in a hierarchical manner, so that P is a tree whose leaves are atomic processes, and internal nodes compose their children and hide as many variables as possible. The basic reduction strategy, proposed by many researchers, is simple: while computing the successors of a state of a process, apply the transition relation repeatedly until a shared variable is accessed. This is applicable since changes to a private state are treated as stuttering steps. The benefit is greatly amplified by applying the reduction in a *recursive* manner exploiting the hierarchical structure, and has been shown to give significant reductions in space and time requirements, particularly for well-structured systems such as rings and trees.

As a rule, we want to hide variables as soon as possible. At the same time it is an advantage, if the structure reflects areas of strong interaction, i.e. if many variables can be hidden in sub-components at the same time. It is to be expected that even in good hierarchical partitions some variables cannot be hidden early. The challenge is to make advantageous choices. In last consequence, the quality of our structure is determined by the savings with respect to a model checking algorithm. The gain depends heavily on the hierarchical partition we impose. In practice, we need a measure to to *compare* different choices that is cheaper to evaluate.[1]From

[1]Estimating the run-time of a model checking algorithm cannot be significantly cheaper than solving the model checking problem itself: suppose a desired safety property does not to hold. Then the best structure uncovers a short violating path, which is much faster than an exhaustive state space exploration. We cannot know that this path exists without solving the original problem, thus we cannot have a tight estimate on the run-time without also knowing the answer.

an academic point of view, run-time comparisons are too dependent on low-level implementation details to give clear analytic data. Thus we strive for a more abstract notion of comparison by means of a cost measure. For typical measures, the problem of finding the *best* hierarchical decomposition is likely to be *NP*-hard, and hence we must look for heuristics that are to be validated by experimentation.

In the following we consider a system to be given as a hypergraph, where processes are vertices and shared variables among them are represented by hyperedges. We rearrange the vertices as leaves of a rooted tree $\mathcal{T}$. Hyperedges that range over large portions of the tree are punished in terms of cost. The problem of structuring the system reduces then to find a tree of low cost.

8.2 The Tree-Indexing Problem

In the following, we describe systems as hypergraphs, where the atomic units correspond to vertices and their connections are represented by hyperedges. E.g., in a reactive module description every hyperedge would correspond to a variable shared by the modules it connects to. Hierarchical partitions introduce an additional tree structure on top of this hypergraph and are augmented with a cost value. We briefly treat combinatorial and computational complexity of finding a tree of minimal cost.

A *hypergraph* $\mathcal{H} = (\mathcal{C}, \mathcal{E})$ is a finite set of vertices $\mathcal{C}$ together with a multi set $\mathcal{E}$, where every *hyperedge* $e \in \mathcal{E}$ is a subset of $\mathcal{C}$. We assume that every e corresponds to a unique label ℓ_e. Hyperedges of size 0 or 1 are disallowed. We draw hyperedges as branching lines. This coincides with common graph representation for the special case that every hyperedge is of size 2.

A *tree-indexing* $\mathcal{T}$ of a hypergraph $\mathcal{H} = (\mathcal{C}, \mathcal{E})$ is a rooted tree over leaf nodes $\mathcal{C}$, where every internal node has at least two children. We draw internal nodes as polygons, all contained polygons and vertices $v \in \mathcal{C}$ are children of this node. The outermost polygon corresponds to the root. Every tree-indexing is qualified by a cost value dependent on $\mathcal{E}$. For instance, in Figure 8.1, the tree-indexing $\{\{A, B\}, \{C, D\}\}$ is better than $\{A, B, C, D\}$, and thus should be of lower cost.

Combinatorial complexity. Given a hypergraph with n labeled vertices, we want to determine the number $T(n)$ of distinguishable tree-indexings. This is in an equivalent formulation recorded as Schröder's fourth problem [Sch70]. It can be solved (for every fixed n) via a generating function method. Let $\varphi(z)$ be the ordinary generating function, where the n^{TH} coefficient corresponds to $T(n)$. Let $\hat{\varphi}(z)$ be its exponential transform. We can construct an equation that $\hat{\varphi}(z)$ has to satisfy according to the theory of admissible constructions [Fla88]. Every tree-indexing is either atomic, i.e. represented as z, or a set of at least two other tree-indexings, namely its children. This can be expressed using the admissible constructions $\mathfrak{Union}$ and $\mathfrak{Set}$.

$$\hat{\varphi}(z) \;=\; \mathfrak{Union}\,(z,\ \mathfrak{Set}\,(\hat{\varphi}(z),\ \text{cardinality} \geq 2)) \tag{8.1}$$

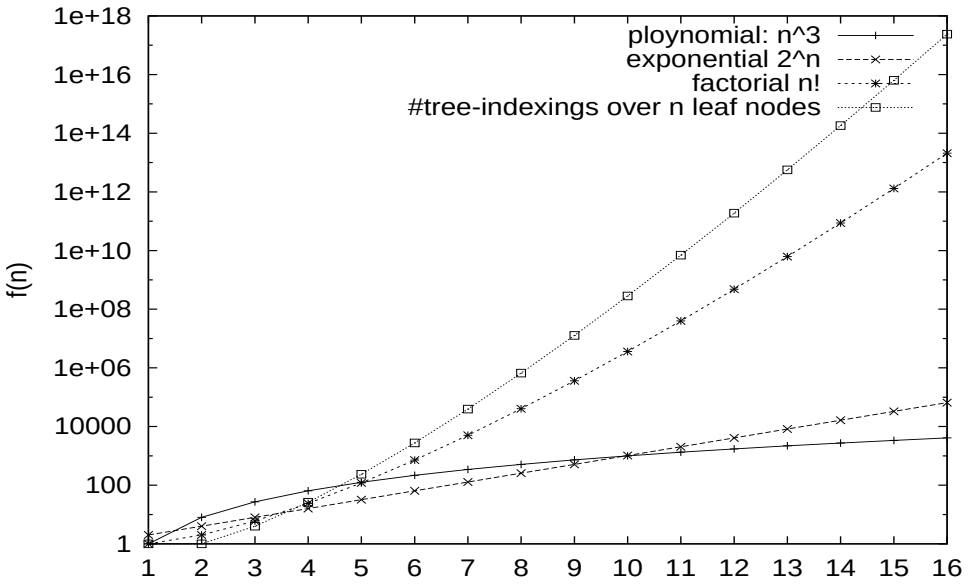


Figure 8.2: Combinatorial Explosion in the Number of Tree-Indexings.

Equation (8.1) can be transcribed as follows.

$$\begin{array}{rrcl} & \hat{\varphi}(z) & = & z \;+\; \sum_{k\geq 2} \frac{1}{k!} \cdot (\hat{\varphi}(z))^k \\ \Longleftrightarrow & \hat{\varphi}(z) & = & z \;+\; \exp(\hat{\varphi}(z)) \;-\; \hat{\varphi}(z) \;-\; 1 \\ \Longleftrightarrow & \exp(\hat{\varphi}(z)) & = & 2\,\hat{\varphi}(z) - z + 1 \end{array} \tag{8.2}$$

There is no closed form known for $\hat{\varphi}(z)$, $\varphi(z)$, or $T(n)$. However, for every fixed n we can extract the n^{TH} coefficient of $\varphi(z)$ with algebraic methods and thus approximate $T(n)$, as done in [Fla97]. Figure 8.2 gives an impression how fast this series grows. Thus we have only little hope to perform an exhaustive search on the domain of possible tree-indexings.

Computational Complexity. We can formulate the problem of finding a good tree-indexing as an optimization problem relative to a fixed *cost* function. This function should punish both deep structures and hyperedges that span over big subtrees. For every $e \in \mathcal{E}$ let $\mathcal{T}_e$ denote the smallest complete subtree of $\mathcal{T}$, such that every vertex $v \in e$ is a leaf of $\mathcal{T}_e$. With *leaves*($\mathcal{T}$) we denote the set of leaf nodes in a tree $\mathcal{T}$. The *depth* of $\mathcal{T}$ is the length of the longest descending path from its root. The *depth cost* of a tree $\mathcal{T}$ is defined as a function

$$depth\_cost(\mathcal{T}) \;:=\; \begin{cases} 2 & \text{if } depth(\mathcal{T}) = 1 \\ depth(\mathcal{T}) & \text{otherwise}. \end{cases} \tag{8.3}$$

The cost of a tree-indexing $\mathcal{T}$ is then defined relative to $\mathcal{H} = (\mathcal{C}, \mathcal{E})$.

$$cost(\mathcal{T}) \;:=\; \sum_{e \in \mathcal{E}} depth\_cost(\mathcal{T}_e) \cdot |leaves(\mathcal{T}_e)| \tag{8.4}$$

For example, the tree-indexing *(i)* in Figure 8.1 has cost $2 \cdot 2 \cdot 2 + 2 \cdot 2 \cdot 4 = 24$, which is preferable to tree-indexings *(ii)* and *(iii)* with costs $4 \cdot 2 \cdot 4 = 32$ and $2 \cdot 2 + 2 \cdot 3 + 2 \cdot 3 \cdot 4 = 34$ respectively. We can transform the problem of finding a tree-indexing with minimal cost into a decision problem.

EDGE-GUIDED TREE-INDEXING: Given a hypergraph $\mathcal{H} = (\mathcal{C}, \mathcal{E})$ and a number $K \in I\!N$. Decide whether there exists a tree-indexing of cost at most K.

The decision problem EDGE-GUIDED TREE-INDEXING is *NP*-complete, even if we restrict to the special case where $\mathcal{H}$ is a multi graph. This precludes the possibility to determine an *optimal* tree-indexing in polynomial time[2] and suggests the application of heuristics in order to find a *reasonably good* tree-indexing efficiently.

Theorem 8.1 *For a multi graph $G = (V, E)$ and an integer K, deciding whether there exists a tree-indexing $\mathcal{T}$ for G with cost at most K is NP-complete.*

Proof: Containment in *NP* is holds, since we can guess any possible tree-indexing $\mathcal{T}$ and compute $cost(\mathcal{T})$ in polynomial time. We show *NP*-hardness by reduction from the following *NP*-complete problem.

MINIMUM CUT INTO EQUAL-SIZED SUBSETS [GJS76][3] Given a graph $G = (V, E)$ with specified vertices $s, t \in V$ and a positive integer K. It is *NP*-complete to answer the following question. Is there a partition of V into disjoint sets V_1, V_2 such that $s \in V_1$, $t \in V_2$, $|V_1| = |V_2|$, such that the number of edges with one endpoint in V_1 and the other endpoint in V_2 is no more than K?

Note that $n := |V|$ is restricted to be even. For $n = 2$ there exists only one solution, thus we consider $n \geq 4$. Furthermore, we assume that that $m := |E| > n/4$. For smaller m the problem is trivial, since we have at least $n/2$ isolated vertices.

Reduction. For every instance (G, s, t, K) we construct—in polynomial time and logarithmic space—an instance (G', K'), such that there exists a partition V_1, V_2 with cost $\leq K$ if and only if there exists a tree-indexing $\mathcal{T}$ of G' with $cost(\mathcal{T}) \leq K'$.

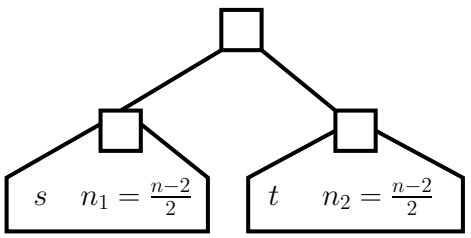


Figure 8.3: Binary, Shallow, and Balanced Tree-Indexing $\mathcal{T}^*$.

[2] Unless *NP* turns out to be equal to *P*.

[3] See also: [GJ79], comment to problem MINIMUM CUT INTO BOUNDED SETS (ND17).

The idea is to augment G in a way, such that a tree-indexing with lowest cost has the shape of a balanced tree of depth 2 (Figure 8.3). To achieve this, we add edges leading to the nodes s and t. We call s and t *attractors*. The number of edges between V_1 and V_2 then corresponds to the number of edges between the subtrees of $\mathcal{T}$ plus some offset.

Given $G = (V, E), s, t, K$, $|V| = n$, $|E| = m$. Let $V' := V \setminus \{s, t\}$. Then we define G' to be the graph with vertices V and $4\,m^2$ additional edges between each $v \in V'$ and every attractor. We call these edges β-edges and use $\beta := 4\,m^2$ as a parameter.

$$\begin{array}{lcl} cost^+ & := & \beta\ (2\,(n/2-1)\ \cdot\ 2\,n/2\ +\ (n-2)\ \cdot\ 2\,n) \\ K' & := & cost^+\ +\ (m-K)\ \cdot\ 2\,n/2\ +\ K\ \cdot\ 2\,n \end{array}$$

It suffices to show that an optimal tree-indexing has the shape of a balanced tree of depth 2. Then the cost for the newly introduced edges is fixed ($cost^+$). We still have to pay for the edges that originated from E. If they live inside a subtree, they are cheap ($2\,n/2$), else they are punished with factor $2\,n$. It is clear that cost K' can be achieved if and only if a balanced partition of V with at most K edges between V_1 and V_2 is possible.

Let $\mathcal{T}^*$ be a tree-indexing like in Figure 8.3. In the worst case, all m original edges in E are in between the subtrees. Thus we can give an upper bound $cost^*$ on $cost(\mathcal{T}^*)$.

$$\begin{array}{lcl} cost^* & := & \beta\ (2\,(n/2-1)\ \cdot\ 2\,n/2\ +\ (n-2)\ \cdot\ 2\,n)\ \ +\ \ m\ \cdot\ 2\,n \\ & = & 3\,\beta\,n(n-2) + 2\,mn \end{array}$$

Lemma 8.2 *Let $\mathcal{T}$ be a tree-indexing of depth 1. Then $cost(\mathcal{T}) > cost^*$.*

Proof: The cost of $\mathcal{T}$ is precisely

$$\begin{array}{ll} (2\,\beta\,(n-2)\ +\ m)\ \cdot\ 2\,n & = \\ 4\beta\,n(n-2) + 2\,mn & > \ \ 3\beta\,n(n-2) + 2\,mn. \end{array}$$

Lemma 8.3 *Let $\mathcal{T}$ be a tree-indexing, where attractors s and t live in the same subtree. Then $cost(\mathcal{T}) > cost^*$.*

Proof: Suppose a tree-indexing $\mathcal{T}$ where t and s live in the same subtree. To estimate the cost of $\mathcal{T}$, we can assume that the tree containing s, t has no siblings within subtree 1 (there is additional punishment by increased depth and no gain). The same holds true for vertices located in a subtree starting at the level of s and t. Thus we can assume that the two attractors share a subtree of depth 1 with n_1 additional vertices and that the remaining $(n - n_1 - 2)$ graph nodes are collected in a second subtree, for in alternative structures the relevant costs are the same or worse. Thus the only scenario to consider is displayed in Figure 8.4.

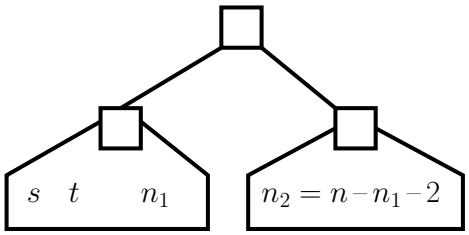


Figure 8.4: Best Case of a Tree-Indexing $\mathcal{T}$ With Attractors in the Same Subtree.

A lower bound on the cost of structure $\mathcal{T}$ is obtained by looking only at β edges. The costs for subtree 1 is $\beta\,(2\,n_1)\,\cdot\,2\,(2+n_1)$ and the graph violations amount to cost $2\,\beta\,(n-n_1-2)\,\cdot\,2\,n$.
Comparing the costs yields

$$\begin{aligned}
cost(\mathcal{T}) - cost^* &= \\
\beta\,(2\,n_1)\,\cdot\,2\,(2+n_1) + 2\,\beta\,(n-n_1-2)\,\cdot\,2\,n - (3\,\beta\,n(n-2) + 2\,m\,n) &= \\
8\,\beta\,n_1 + 4\,\beta\,{n_1}^2 + \beta\,n^2 - 4\,\beta\,n n_1 - 2\,\beta\,n - 2\,mn &= \\
\beta\,\left(n^2 - 4\,n + 4 - 4\,n n_1 + 8\,n_1 + 4\,n_1^2\right) + \beta\,(2\,n-4) - 2\,mn &= \\
\beta\,((n-2) - 2\,n_1)^2 + \beta\,(2\,n-4) - 2\,mn &\geq \\
4\beta - 2\,mn &\geq \\
4\,mn - 2\,mn &> 0.
\end{aligned}$$

///

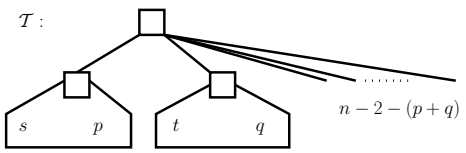
We can assume now that the optimal solution (i.e. this that allows the largest K' and thus the largest K) is non-monolithic and that attractors s and t live in different subtrees. Before we argue that more than two subtrees are too expensive, we need to state a lower bound on the cost of subtrees with one attractor.

Lemma 8.4 *Let $\mathcal{T}$ be a tree-indexing where the root has ≥ 3 children, attractor s lives in subtree 1 and attractor t in subtree 2. Then $cost(\mathcal{T}) > cost^*$.*

Proof: Assume that subtrees 1 and 2 contain, in addition to the attractors, p respectively q nodes, $p+q < n-2$. Make a case split on the depth of the tree-indexing.

(i) The depth of the tree-indexing is 2.
Then the cost can be estimated as in Figure 8.5. It suffices to show the following inequality:

$$\begin{aligned}
& \beta\,p\,\cdot\,2\,(p+1) + \beta\,q\,\cdot\,2\,(q+1) + \beta\,n\,\cdot\,2\,n + \beta\,(n\text{-}2\text{-}p\text{-}q)\,\cdot\,2\,n > cost^* \\
\Leftrightarrow\quad & \beta\,\left(n^2 - 2\,n - 2\,np - 2\,nq + 2\,p + 2\,p^2 + 2\,q + 2\,q^2\right) - 2\,mn > 0 \\
\Leftrightarrow\quad & \beta\left(\begin{array}{cc} n^2{-}4\,n{+}4{-}2\,np - 2\,nq{+}p^2 + q^2{+}2\,pq{+}4\,p{+}4\,q \\ +2\,n{-}4 \qquad\qquad +p^2 + q^2{-}2\,pq{-}2\,p{-}2\,q \end{array}\right) - 2\,mn > 0 \\
\Leftrightarrow\quad & \beta\big(\underbrace{((n-2)-(p+q))^2}_{\geq 0} + \underbrace{(p-q)^2}_{\geq 0} + 2\,\underbrace{(n-p-q-2)}_{\geq 1}\big) - 2\,mn > 0 \\
\Leftarrow\quad & 4\,m^2\cdot 2 > 2\,mn
\end{aligned}$$



$cost(\mathcal{T})\geq$

$\beta\, p \cdot 2\,(p+1)$	inside subtree 1
$\beta\, q \cdot 2\,(q+1)$	inside subtree 2
$\beta\, n \cdot 2\, n$	to attractor in other subtree
$\beta\,(n\text{-}2\text{-}p\text{-}q) \cdot 2\, n$	new rightmost violations

Figure 8.5: Lower Bound on Cost of a Non-Binary Tree-Indexing $\mathcal{T}$.

(ii) The depth of the tree-indexing is at least 3.
Then there are $(n-2-p-q)+(n-2) \geq n-1$ edges that have to be payed for with factor $\beta \cdot 3\, n$.

$$\begin{array}{rrcl} 3\,\beta\, n(n-1) & -\, cost^* & = & \\ 3\,\beta\, n & -\, 2\, mn & > & 0. \end{array}$$

///

Assuming that the root has exactly two children, we can now exclude tree-indexings of depth greater than 2.

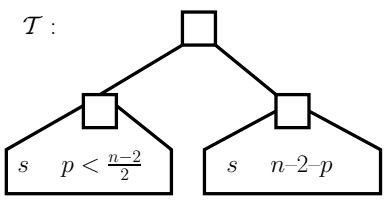
Lemma 8.5 *Let $\mathcal{T}$ be a tree-indexing where the root has 2 children, in each subtree lives one attractor and $depth(\mathcal{T}) \geq 3$. Then $cost(\mathcal{T}) > cost^*$.*

Proof: Every of the $n-2$ leaves in $V \setminus \{s,t\}$ has β edges to the attractor in the other subtree. It suffices to estimate the internal costs of the two subtrees with $\beta\, 2 \cdot 2$ (we have at least two edges somewhere inside).

$$\begin{array}{rrcl} & \beta\,(4 \;+\; (n-2)\,3\,n) & > & cost^* \\ \Leftrightarrow & 4 \cdot 4\, m^2 & > & 2\, mn. \end{array}$$

///

Lemma 8.6 *Let $\mathcal{T}$ be a tree-indexing where the root has 2 children, in each subtree lives one attractor, $depth(\mathcal{T}) = 2$ and subtree 1 contains $p < (n-2)/2$ nodes from $V \setminus \{s,t\}$. Then $cost(\mathcal{T}) > cost^*$.*



$cost(\mathcal{T})$:

$\beta\, p \cdot 2\,(p+1)$	inside subtree 1
$\beta\,(n\text{–}2\text{–}p) \cdot 2\,(n\text{–}2\text{–}p)$	inside subtree 2
$\beta\,(n\text{–}2) \cdot 2\, n$	to attractor in other subtree

Figure 8.6: Lower Bound on Cost of an Unbalanced Tree-Indexing $\mathcal{T}$.

Proof: We can estimate $cost(\mathcal{T})$ as in Figure 8.6.

$$\begin{array}{lll} & \beta\,(p\cdot 2\,(p+1)+(n-2-p)\cdot 2\,(n-2-p)+(n-2)\cdot 2\,n) & > \quad cost^* \\ \Leftrightarrow & \beta\big(\underbrace{4\,p^2+8\,p+n^2-4\,n-4\,np+4}_{f_n(p)}\big) & > \quad 2\,mn \end{array}$$

The term in $f_n(p)$ is minimal for $\frac{d}{dp}\,f_n(p) = 0$, i.e. for $p = n/2-1$. Since $p \leq n/2-2$ and f_n is quadratic in p, it suffices to check the value $p_0 = n/2-2$.

$$\begin{array}{rcl} f_n(n/2-2) & = & 2\,(1/2\,n-2)\,(-1+1/2\,n)+n\,(1/2\,n+1)-(n-2)\,n \\ & = & 4 \end{array}$$

Since $4\,\beta = 16\,m^2 > 4\,mn > 2\,mn$, this proves the lemma.

///

Thus we can be sure that only binary, shallow and balanced tree-indexings like in Figure 8.3 are candidates for optimal solutions. The soundness of the reduction follows.

□

8.3 A Greedy Algorithm to Partition Hierarchically

In this section we develop a greedy-style algorithm that constructs a tree-indexing by successively grouping together sets with strong correspondence. The choice of these candidates relies on heuristics, which make use of a key observation: strong correspondences are likely to represented by a large number of connections.

A schematic description of our proposed algorithm is given in Figure 8.7. The variable $\mathcal{F}$ is used to maintain a partial tree-indexing, i.e., a forest $\mathcal{F}$ with leaves $\mathcal{C}$. It is initialized as the forest with $|\mathcal{C}|$ trees, each consisting of a single node. The priority queue Q is ordered according to a rating function $\mathbf{r} : \wp^{\mathcal{D}} \times 2^{\mathcal{C}}_{\star} \to I\!R$. $\wp^{\mathcal{D}}$ is the set of forests over leaves $\mathcal{D} \subseteq \mathcal{C}$ and thus contains all possible sub-forests of $\mathcal{F}$. $2^{\mathcal{C}}_{\star}$ denotes a multi set of hyperedges and initially corresponds to $\mathcal{E}$. The top element of the queue is a subset of $\mathcal{F}$ with maximal $\mathbf{r}$-value.

The algorithm proceeds as follows. An initial set of candidates proposed for grouping together is inserted in the priority queue. Then a small number of executions of the while-loop follow. In each execution, the most promising candidate $\mathcal{A}$ is dequeued and the data is updated: in the forest, the trees in $\mathcal{A}$ are replaced by a tree with the fresh root $\mathcal{A}'$ and children $t \in \mathcal{A}$. Every set containing trees $t \in \mathcal{A}$ is removed from the priority queue and new candidates containing $\mathcal{A}'$ are inserted. Hyperedges $e \subseteq \mathit{leaves}(\mathcal{A}')$ are deleted, since they should not influence later selections.

This description leaves open the questions, what should be used as a rating function and which candidates should be considered. We explain these aspects of the algorithm in the following.

Algorithm: *partition_incrementally*
input: hypergraph $\mathcal{H} = (\mathcal{C}, \mathcal{E})$
output: tree-indexing over leaves $\mathcal{C}$

PriorityQueue Q:=*emptyQueue*
Forest $\mathcal{F} := \mathcal{C}$
FORALL considered candidates $\mathcal{A} \subseteq \mathcal{F}$
  insert$(\mathcal{A}, Q)$
WHILE *notempty*(Q)
  $\mathcal{A} := top(Q)$ /⋆ pick the best candidate ⋆/
  let $\mathcal{A}'$:=*fresh root node with children* $t \in \mathcal{A}$
  $\mathcal{F} := (\mathcal{F} \setminus \mathcal{A}) \cup \{\mathcal{A}'\}$
  $\mathcal{E} := \mathcal{E} \setminus \{e \mid e \subseteq leaves(\mathcal{A}')\}$ /⋆ remove covered hyperedges ⋆/
  update$(\mathcal{E}, \mathcal{A}, \mathcal{A}')$ /⋆ replace all $t \in \mathcal{A}$ by $\mathcal{A}'$ ⋆/
  FORALL $\mathcal{B} \in Q$ with $\mathcal{B} \cap \mathcal{A} \neq \emptyset$
    remove$(\mathcal{B}, Q)$
  FORALL new candidates $\mathcal{D}$ containing $\mathcal{A}'$
    insert$(\mathcal{D}, \mathcal{Q})$
RETURN $\mathcal{F}$

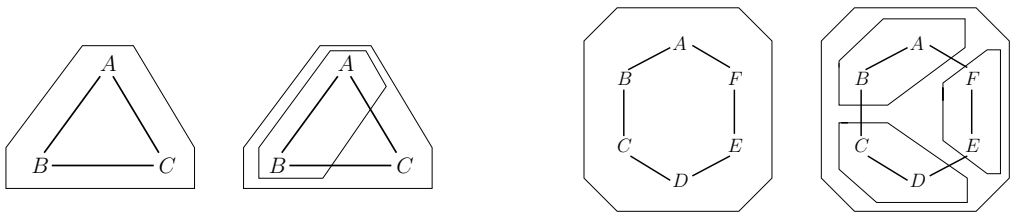
Figure 8.7: Incremental Algorithm for Constructing a Tree-Indexing.

Developing a good rating function. The local choice of the best candidate could be performed by means of the *cost* function defined in (8.4), i.e., by picking the candidate with lowest cost after clustering. We chose not to do so for two reasons. First, the specific cost function was derived such that an *NP*-complete problem could be encoded into it; for small variations of this definition, the proof failed - thus, this particular definition is somehow artificial. Second, we would like to tune the choice by means of parameters in the rating function. Doing this with the cost function would almost certainly destroy the provable *NP*-hardness, and thus the justification for the choice.

Instead, we develop a *rating function* subsequently by taking into account the—supposedly—crucial factors concerning the structure of the proposed candidate. Most importantly, we want to know the number of additional hyperedges that are completely covered by this set, and thus can be hidden from the outside without losing information.

Definition 8.7 (Cover Number) *Let* $\mathcal{H} = (\mathcal{C}, \mathcal{E})$ *be a hypergraph,* $\mathcal{F}$ *a forest over leaves* $\mathcal{C}$*,* $\mathcal{A} = \{\mathcal{T}_1, \ldots, \mathcal{T}_k\} \subseteq \mathcal{F}$*. The* cover number *of* $\mathcal{A}$*, in symbols* $\langle\!\langle \mathcal{A} \rangle\!\rangle$*, is defined as the number of hyperedges covered by the trees in* $\mathcal{A}$*.*

$$\langle\!\langle \{\mathcal{T}_1, \ldots, \mathcal{T}_k\} \rangle\!\rangle \;:=\; \left|\left\{ \ell_e \,\middle|\, e \in \mathcal{E},\ e \subseteq leaves(\mathcal{A}),\ \forall i.\ e \not\subseteq leaves(\mathcal{T}_i) \right\}\right|$$



(i) triangle *(ii)* hexagon

Figure 8.8: Alternative Hierarchical Partitionings: How Strong is a Connection?

Though this value tells a lot about a candidate, it is isolated not a good guideline. Recall that the set $\mathcal{C}$ has naturally always the highest possible cover number $|\{\ell_e \,|\, e \in \mathcal{E}\}|$.

Strength of connections. Figure 8.8 illustrates, how the strength of connection is not proportional to the edge/node ratio. In the right triangle in *(i)*, it is not intuitive why one pair $\{A, B\}$ should be on a level below in the hierarchy. Rather we would like the left option to be taken. When considering the hexagon instead, the six hyperedges seem too weak an argument to group this big structure monolithically[4]; we would favor the right alternative. So, three components with three links should be stronger than two with just one. But at the same time, two components with one link should be rated higher than six components with six links. This suggests that size is not to be taken as a linear factor.

We relate the cover number to the *size* n of a candidate, where size matters in terms of *possible* connections, which is $\binom{n}{2} = \mathcal{O}(n^2)$. We propose the following rating function.

$$\mathbf{r}_{pref}(\mathcal{A}) \;:=\; \frac{\langle\!\langle \mathcal{A} \rangle\!\rangle}{|\mathcal{A}|^2} \tag{8.5}$$

Comparing this to the examples in Figure 8.8, we can verify that $\mathbf{r}_{pref}$ precisely favors the options we argued for. In the following we refine $\mathbf{r}_{pref}$ by adding more structural information.

Definition 8.8 (Touch of a Candidate) *Let $\mathcal{H} = (\mathcal{C}, \mathcal{E})$ be a hypergraph and $\mathcal{F}$ be a forest over leaves $\mathcal{C}$. Then the* touch *of $\mathcal{A} \subseteq \mathcal{F}$ is defined as the labels from hyperedges that connect $\mathcal{A}$ with the rest of $\mathcal{H}$.*

$$\mathit{touch}(\mathcal{A}) \;:=\; \left\{\, \ell_e \,\middle|\, e \cap \mathit{leaves}(\mathcal{A}) \neq \emptyset \wedge e \not\subseteq \mathit{leaves}(\mathcal{A}) \,\right\}$$

Definition 8.9 (Depth of a Candidate) *The* depth *of a tree with only one node equals 0. Let $\mathcal{F}$ be a forest, $\mathcal{A} = \{\mathcal{T}_1, \ldots, \mathcal{T}_k\} \subseteq \mathcal{F}$. The* depth *of $\mathcal{A}$ is defined as*

$$\mathit{depth}\,(\{\mathcal{T}_1, \ldots, \mathcal{T}_k\}) \;:=\; 1 + \max_{1 \leq i \leq k} \mathit{depth}(\mathcal{T}_i)$$

[4]We note that a greedy algorithm can run into a steric trap here: choosing first $\{A, B\}$ and then $\{D, E\}$ yields a sub-optimal hierarchical partitioning.

We do not want to cut out subsystems that are multiply connected to the rest, i.e., those who share many hyperedges with their complement. This is reflected by the number of labels in the touch: if it is small, the candidate is more attractive. Also, it is perceivable that preference should be given to candidates with small depth. Hence we propose the following improved rating function.

$$\mathbf{r}^{+}_{pref}(\mathcal{A}) := \frac{\langle\!\langle \mathcal{A} \rangle\!\rangle}{|\mathcal{A}|^2} \;+\; \frac{\varepsilon_1}{|touch(\mathcal{A})|} \;+\; \frac{\varepsilon_2}{depth(\mathcal{A})} \tag{8.6}$$

The parameters ε_1 and ε_2 are supposed to be chosen small and positive. For the experiments in Section 8.4, the assignments $\varepsilon_1 := 1/1000$, $\varepsilon_2 := 1/100000$ were used.

Restricting the set of considered candidates. In our formulation of the algorithm *partition_incrementally* we remained unclear on what the considered candidates are. We want to weed out hopeless candidates, e.g., those not sharing any labels, before adding them to our priority queue. In a positive formulation, consider only candidates that are extensions of interesting pairs.

Definition 8.10 (Interesting Pair) *Given a hypergraph $\mathcal{H}(\mathcal{C}, \mathcal{E})$ and a forest $\mathcal{F}$ over leaves $\mathcal{C}$. An* interesting pair $\{\mathcal{T}_1, \mathcal{T}_2\}$ *is a subset of $\mathcal{F}$, such that $touch(\mathcal{T}_1) \cap touch(\mathcal{T}_2) \neq \emptyset$.*

Clearly, every candidate that is *not* a superset of an interesting pair has cover number 0 and thus can be neglected. As it turns out in our implementation, the expensive part of the algorithm is the computation of the cover numbers. First computing interesting pairs and then extending them to candidates is an advantage.

The number of candidates can still be excessive. Consider a hyperedge connecting all vertices, then all pairs are interesting pairs. Since the number of subsets of $\mathcal{C}$ is exponential in $|\mathcal{C}|$, an exhaustive enumeration is not feasible for large systems. If conservative techniques (like considering just extensions of interesting pairs) do not suffice, we have to apply a more rigorous pruning, even for the price of thereby ignoring good candidates. An obvious suggestion is to consider only candidates up to a certain size k, thus establishing an upper bound of $n^{k+1} - n - 1$ candidates. This k can be adjusted according to n, which provides a simple and reasonable method to prune the search.

In the algorithm, the number of forests—initially n—decreases by one with each execution of the while loop. Operations like evaluating the rating function, testing $\mathcal{B} \cap \mathcal{A} \neq \emptyset$, and constructing new candidates containing $\mathcal{A}'$ can be assumed to be $\mathcal{O}(n)$, thus one execution of the while loop has the run-time bound $\mathcal{O}(n \cdot n^{k+1})$. The whole algorithm *partition_incrementally* has n executions of the while loop, which yields the polynomial bound $\mathcal{O}(n^{k+3})$ on its run-time.

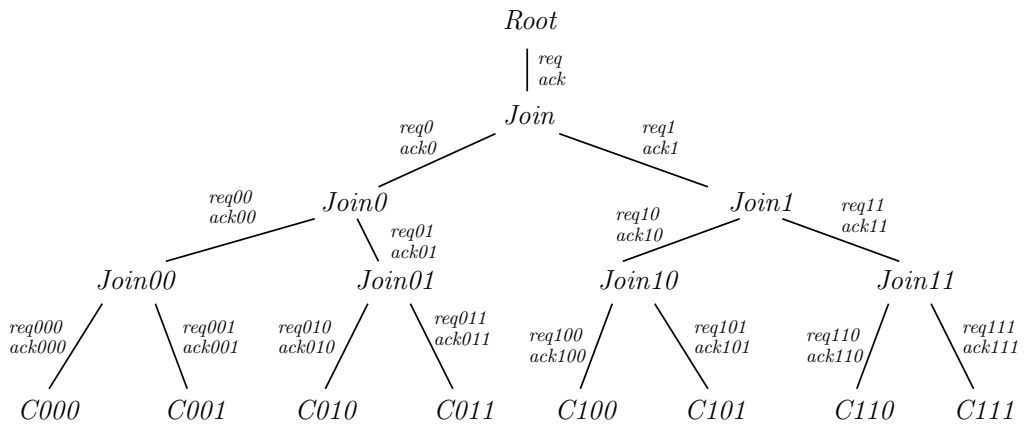


Figure 8.9: Layout of an Asynchronous Parity Computer With Eight Clients.

8.4 Experimental Results

We implemented the algorithm from Section 8.3 in an experimental version of the Mocha *model checking tool [AdAG+01] and report run-time data in the following.*

For symbolic (BDD-based) model checking, the Java implementation makes use of native libraries. However, our experiments do not make use of this option and perform the check in a purely enumerative manner. Therefore, given run-times (in milli-seconds) and memory requirements are those of the Java Virtual Machine, executing on a Sun Enterprise 450 with UltraSPARC-II processors, 300 MHz. A contingent of 128 MB of memory was allocated, run-times are in milliseconds. Together with an optimization in the enumerative check called *"Next" heuristic* [AW99], we are able to corroborate effectiveness and usability of our algorithm in some simple examples. We consider an asynchronous parity computer, leader election in a ring, and an opinion poll protocol. The Mocha specifications are given in [MA00].

Note that in these experiments the checked property influences the obtained structure. In Mocha every property relies on *variables* of the system. These variables can not be hidden and therefore are neglected in the partition algorithm, i.e., they are ignored in the evaluation of the rating function.

8.4.1 Asynchronous Parity Computer

This example models a parity computer, designed as a binary tree (Figure 8.9). The leaf nodes are *Client* modules (abbreviated with C), communicating a binary value to the next higher *Join*. A simple hand-shake protocol is devised by the two variables *req* and *ack*. All components are supposed to move asynchronously. Thus the join nodes have to wait for both values to be present, before reporting their exclusive-or upwards. The *Root* component, after receiving the result of the computation, hands down an acknowledgment. When a client receives an acknowledgment, it is able to devise a fresh value.

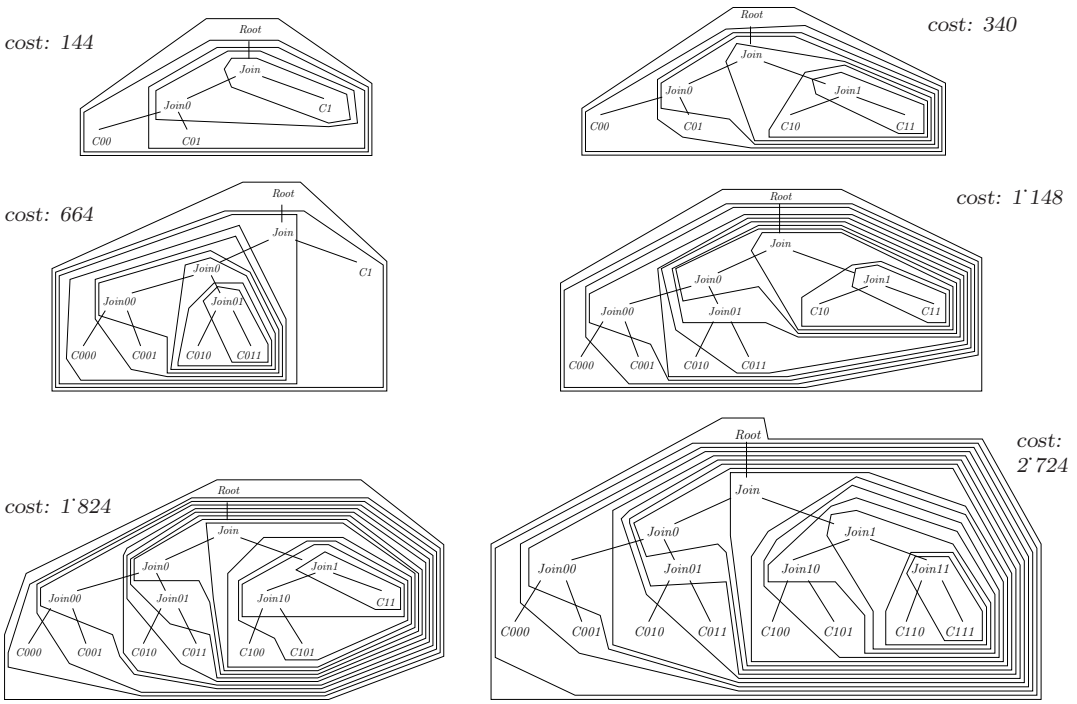


Figure 8.10: Parity Computers $N = 3, \ldots, 8$, Partitioned via Rating Function $\mathbf{r}_{pref}$.

We consider binary trees with N client nodes, where N varies from 3 to 8. The number of variables increases linearly with N, whereas the state-space grows exponentially. The sample question we pose is whether the module *Root* will ever output a value *zero* or *one*. We expect our model checking algorithm to falsify the claim that it never will.

Reachability involves computing the successors of every state encountered, starting with the initial states. Consider the set S of all the processes. Then, successors of a state are computed by executing one step of one of the processes in S. Now suppose, we cluster the processes *Join00*, *C000*, and *C001* into one composite process called P, and replace these three processes in S with P. It is clear that the communication between *J00* with its children clients can be hidden from the rest of the system. Consequently, in reachability analysis of S, when we compute the successors due to execution of P, we can let the subprocesses in P repeatedly execute until *Join00* communicates with its parent *Join0*. This is formalized in MOCHA by substituting P by a construct *next* Θ *for* P, where atomic transitions correspond to sequences of transitions of P until a variable shared with the remaining system is accessed. The modified search yields an improved performance as it cuts down on unnecessary interleavings.[5] This scheme can be applied repeatedly. It should be clear that the effectiveness of the scheme depends on the hierarchical partition.

[5] A well-known method for reducing state-space in asynchronous systems is based on partial-order reductions [GPS96]. The "Next" heuristic is incomparable to this method, see [AW99].

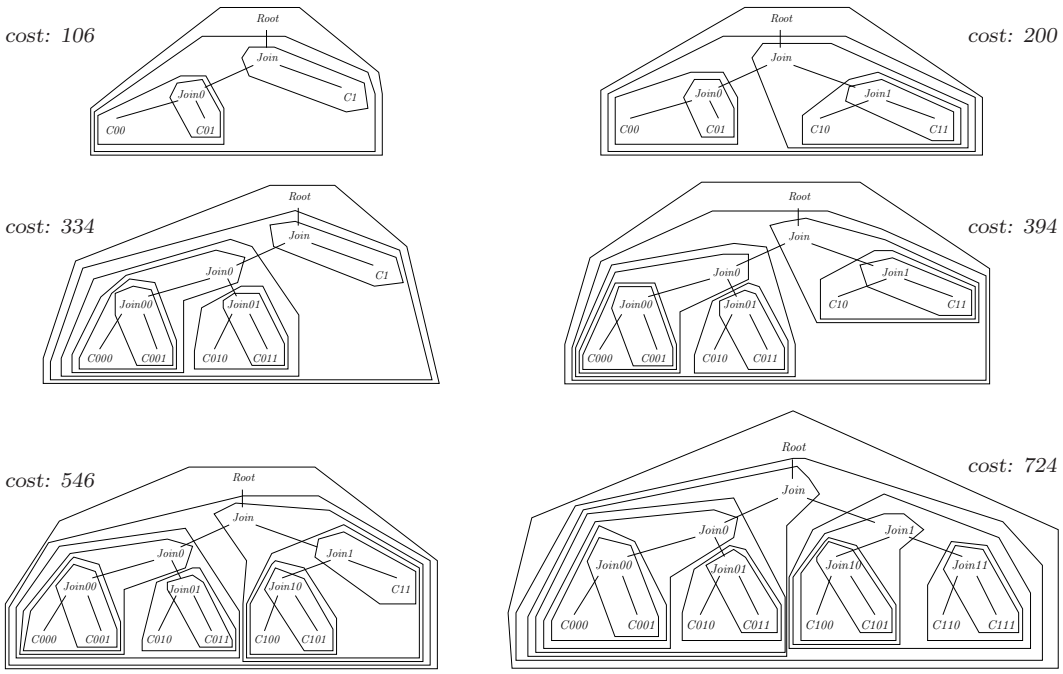


Figure 8.11: $\mathbf{r}^{+}_{pref}$ Yields Shallower Partitions With Lower Cost Values Than $\mathbf{r}_{pref}$.

N	partition	|table|	check
3	162	95	1˙121
4	853	645	4˙921
5	740	1˙943	17˙086
6	2˙811	16˙045	161˙394
7	9˙928	58˙351	694˙834
8	47˙239	410˙901	5˙442˙315

Using $\mathbf{r}_{pref}$ as Rating Function

N	partition	|table|	check
3	105	51	973
4	148	117	1˙707
5	627	139	1˙780
6	2˙097	205	3˙000
7	10˙592	271	4˙395
8	50˙664	469	8˙322

Using $\mathbf{r}^{+}_{pref}$, $\varepsilon_1 := \frac{1}{1000}$, $\varepsilon_2 := \frac{1}{100000}$

Table 8.1: Parity Computer: Comparison of two Heuristic Functions.

An intuitively good choice for this hierarchical partition is grouping together bottom up. Detecting this algorithmically is subtle. E.g., the difference between {*Join0*, *Join00*} and {*Client000*, *Join00*} is only minor, since both pairs cover exactly two variables. An incautious technique easily runs into errands, as to be seen in Figure 8.10. Using $\mathbf{r}_{pref}$ as rating function in the algorithm *partition_incrementally* leads to uncomfortably deep hierarchies.

N	partition	|table|	check
3	57	51	404
4	75	117	1˙097
5	127	139	1˙726
6	516	205	2˙929
7	247	271	4˙364
8	342	469	8˙184

Table 8.2: $\mathbf{r}^{+}_{pref}$ with $|\mathcal{A}| \leq 2$

The more sophisticated rating function $\mathbf{r}^+_{pref}$ performs far better, as seen in Figure 8.11. The parameters ε_1 and ε_2 were calibrated to $\varepsilon_1 := 1/1000$ and $\varepsilon_2 := 1/100000$, giving shallow structures a smaller bonus than those touching only few variables.

The deep hierarchical structure obtained by using $\mathbf{r}_{pref}$ lead to excessive number of explored states, whereas with $\mathbf{r}^+_{pref}$ the growth of the explored state space with increasing N is only moderate. This gap is also reflected by the significantly higher cost values. Table 8.1 shows the run-time data in detail for unrestricted candidate size, Table 8.2 restricts to candidates of size 2. With "partition" we denote the preprocessing time used by *partition_incrementally* and "check" corresponds to the run-time of the model checking algorithm. The number of explored states is recorded under "|table|", MOCHA keeps the states in a hash table. Note that the property we check does not hold, thus the model checking algorithm is able to abort without exploring all reachable states.

In the left and middle table, the time consumed for computing the hierarchical partition exceeds the model checking time for bigger examples. This is because we chose not to restrict the candidate size here, which yields a number of candidates increasing exponentially in N. The obtained hierarchical partitions right in Table 8.1 and in Table 8.2 are identical—due to the tree structure, the best rated candidate here is always of size two.

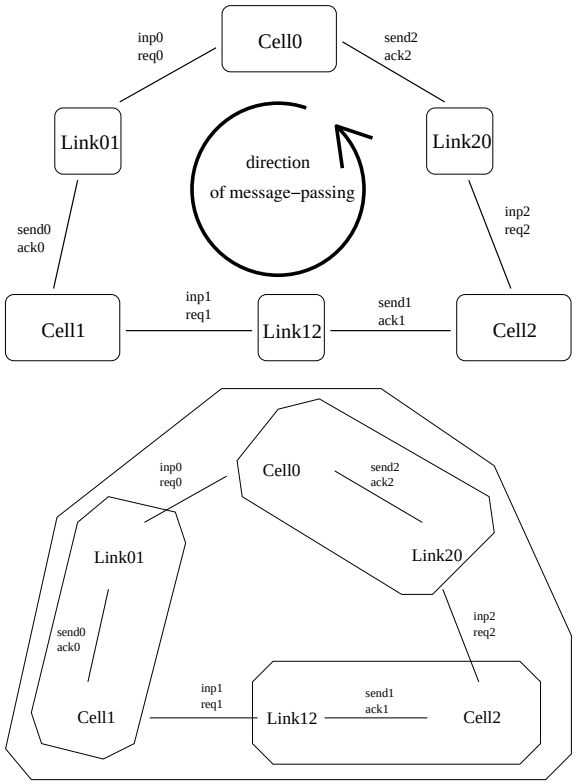
In fact, for the shape of our rating functions and for tree-shaped graphs, the best rated candidate is *always* of size two, since we severely punish increase in size. E.g., for $\mathbf{r}_{pref}$, assume a candidate $\mathcal{A}$ of size $n > 2$. There are $n-1$ connections of multiplicity $\leq m$ for some m. Then the best sub-candidate of $\mathcal{A}$ with size two has rating $m/4$, while $\mathbf{r}_{pref}(\mathcal{A}) \leq (n-1) \cdot m/n^2$, and $m/4 - (n-1) \cdot m/n^2 > 0 \Leftrightarrow mn^2 - 4m(n-1) > 0 \Leftrightarrow (n-2)^2 > 0$, thus $\mathbf{r}_{pref}(\mathcal{A}) < m/4$. The additive terms in $\mathbf{r}^+_{pref}$ give the candidate of size two always a equal or higher bonus than to $\mathcal{A}$.

8.4.2 Leader Election in a Ring

We consider a leader election protocol as a second sample problem. The modules are arranged in a ring topology, where each buffered channel is modeled by a separate module. The modules use a standard leader election protocol to elect the one with the highest (unique) id number: every cell proceeds by sending the highest id number it has seen so far, starting with its own.

If a cell receives its own id number, it declares itself to be the leader (see [Lyn96] for an exhaustive treatment of this problem). Figure 8.12 shows how a ring with 3 cells is partitioned incrementally with application of rating function $\mathbf{r}^+_{pref}$. The checked property is a valid invariant: at no time there is more than one process denoted leader (a safety property).

The gap in time performance between unstructured and clustered system was not extreme, but noticeable (tables in Figure 8.12). Unfortunately, we are not able to perform checks with larger rings, since the modeling of the links as finite state machines turns out to be very consumptive with respect to the state space.



size	hash table	model checking
2	563	6˙270
3	70˙797	1˙327˙756

Unpartitioned

size	partition	hash table	model checking
2	217	563	4˙615
3	279	61˙455	661˙275

Hierarchically partitioned via $\mathbf{r}^{+}_{pref}$

Figure 8.12: Leader Election Protocol With 2 and 3 Cells.

8.4.3 Opinion Poll Protocol

The third sample problem is meant to demonstrate the behavior of our heuristic in a setting, where there is no obviously preferable choice.

Consider a poll for a public opinion. There is a line of N pollers P_i and two non-connected lines of citizen A_i and B_i, plus two special citizen C and D. Poller P_1 starts raising an issue with a `Yes/No` question. Let us assume that the way one asks influences the answer. Poller P_1 starts of with an opinion he got from a random source (called *Master*). Poller P_{i+1} is influenced by P_i. The citizen are influenced by one other citizen and the poller who interviews them. For instance, A_{i+1} is influenced by A_i's and P_i's opinions, and A_1 is influenced by a random source N_A and P_1. Figure 8.13 illustrates this for the cases $N = 1$ and $N = 2$, where the communication pattern is indicated by arrows,

For $N = 1, 2, 3, 4$ we considered three invariants: *(i)* a false property that is easy to falsify, *(ii)* a false property that requires a special scenario (called *bad property* in the following), and *(iii)* a true property. The experiments compare plain enumerative model checking and application of the "Next" heuristic, where the preprocessing follows one of the following strategies. *a. 2-merge*: Group any pair with a connection without further preference, i.e., use $sig(\langle\!\langle . \rangle\!\rangle)$ as rating function and consider only

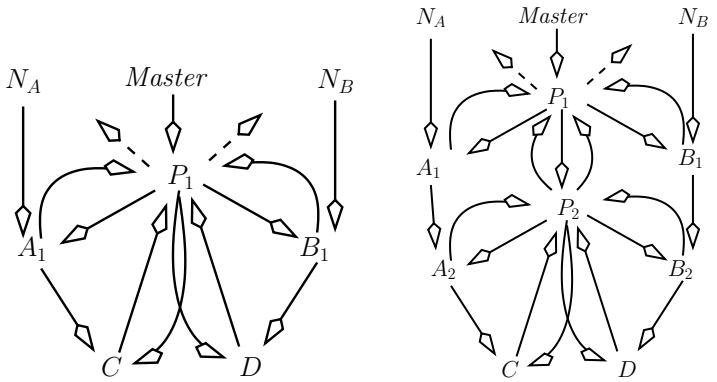


Figure 8.13: Opinion Poll Communication Pattern for $N = 1$ and $N = 2$.

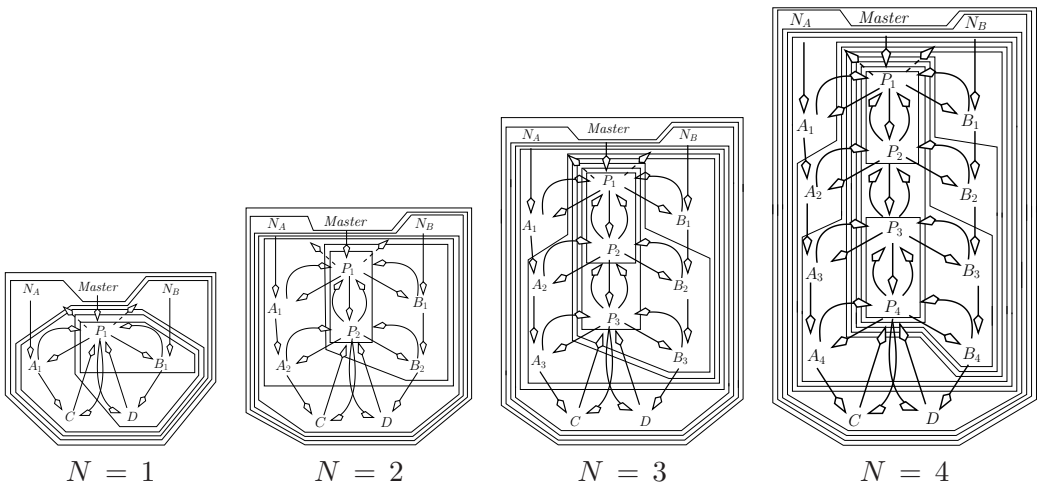


Figure 8.14: Opinion Poll: Hierarchically Partitioned With Rating Function $\mathbf{r}^{+}_{pref}$.

candidates of size 2, *b. pref*: Partition incrementally according to rating function $\mathbf{r}_{pref}$, and *c. pref+*: Partition incrementally according to rating function $\mathbf{r}^{+}_{pref}$. For the latter, we included the results of the preprocessing in Figure 8.14. It is interesting to note that sometimes triples were preferred to pairs.

The quantitative comparison is listed in Table 8.3. For the false properties *(i)* and *(ii)* the enumerative check is up to five times *slower*, if sophisticated heuristics are applied. Apparently it is more tedious to reach a counter-example scenario here, if more structure is given. For the true property *(iii)*, the enumerative check speeds up by a third, when the "Next" heuristic is applied. For larger N the more sophisticated clustering techniques *pref* and *pref+* perform slightly better than *2-merge*.

(i) **false Judgment:** `System |= (result = DontKnow)`							
N\Method	*plain*	*2-merge*		*pref*		*pref+*	
1	742	2 854	(partition) (check)	280 754	(partition) (check)	274 861	(partition) (check)
2	3'713	32 4'313	(partition) (check)	1'808 6'862	(partition) (check)	1'886 6'850	(partition) (check)
3	32'181	7 26'330	(partition) (check)	1'790 87'708	(partition) (check)	2'047 88'879	(partition) (check)
4	345'071	22 435'529	(partition) (check)	5'256 1'390'739	(partition) (check)	4'828 1'351'527	(partition) (check)

(ii) **bad Judgment:** `System |= NoNegativeResult`							
N\Method	*plain*	*2-merge*		*pref*		*pref+*	
1	1'113	2 916	(partition) (check)	238 766	(partition) (check)	203 886	(partition) (check)
2	4'846	5 3'930	(partition) (check)	1'625 7'130	(partition) (check)	1'667 6'561	(partition) (check)
3	32'580	7 29'324	(partition) (check)	1'788 87'827	(partition) (check)	1'920 73'350	(partition) (check)
4	385'951	20 375'977	(partition) (check)	5'476 1'665'765	(partition) (check)	6'458 1'306'961	(partition) (check)

(iii) **true Judgment:** `System |= ~((result = DontKnow) & (result = Yes))`							
N\Met.	*plain*	*2-merge*		*pref*		*pref+*	
1	30'565	2 23'689	(partition) (check)	290 24'369	(partition) (check)	292 24'423	(partition) (check)
2	610'131	5 454'089	(partition) (check)	1'787 482'600	(partition) (check)	2'148 482'214	(partition) (check)
3	8'488'532	17 6'392'536	(partition) (check)	2'301 5'920'255	(partition) (check)	2'357 5'865'170	(partition) (check)
4	93'557'192	23 60'934'073	(partition) (check)	5'733 57'762'294	(partition) (check)	5'068 57'165'981	(partition) (check)

Table 8.3: Opinion Poll Protocol: Run-Time Comparison of Three Rating Functions.

8.5 Reflection: Hierarchical Partitioning

We developed a notion of hierarchical partitions and introduced a method to compare different structures by means of a cost function. This is applicable, whenever the relationship of entities can be adequately described via hyperedges. For the presented cost function, the problem of determining an optimal hierarchical partition is *NP*-complete.

We presented a scalable greedy method to compute approximately good hierarchical partitions based on a heuristic rating function. We argued that—in order to achieve a good result—this function should be based on four criteria: number of covered hyperedges, size, number of occurring hyperedges, and structural depth. This is corroborated by qualitative and quantitative data. We implemented our algorithm in an experimental version of the Mocha model checking tool and measured its performance on small and medium sized examples.

It should be noted that our proposed method gives no guarantee on how the obtained result compares to an optimal solution. Since we apply a variation of

local search, it is to be expected that our algorithm can get caught in local optima. Moreover, optimality in the sense of least cost does not necessarily imply minimal time- or space-consumption when running a model checking algorithm. In general, we cannot expect to express such subtle behavioral properties of a system via a simple function, i.e., a function that is fast to evaluate.

Though our case studies suggest that in the rating function both touch and depth of the candidates should be taken into account, it remains open, *how* this should be reflected. The values for the parameters ε_1 and ε_2 in (8.6) were chosen according to fit the parity computer example. It would be desirable to investigate the impact of parameter changes in general, but we seem to lack apt mathematical means to do so.

Related problems. Hierarchical structures find a wide range of application in design, description and physical organization of both software and hardware. In particular, the decomposition of large circuits in VLSI layout turns out to be a crucial problem and has received a respectable amount of attention [She95]. Here the partitions are typically shallow (i.e., of depth two) and mainly motivated by size constraints that single components have to meet. Optimality is typically described as the least number of components with as few as possibly connections.

Similar structures, called classification trees (e.g. [GRD91]), are used as expressive decision trees over large sets of data. The internal nodes are labeled by distinguishing criteria and all leaf nodes are distinguishable. Finding expressive classification trees is computationally hard.

Though various advanced techniques have been developed for these problems, to the best of our knowledge none of them is applicable in the considered case. In our setting, *every* tree-indexing is a feasible solution, there is no constraint satisfaction component and there might well be two leaves that are alike.

Open problems. We noted that finding an optimal solution with respect to our cost function is *NP*-hard, but this does not preclude the existence of a polynomial approximation scheme. Also, it remains unknown, how the computational complexity compares with respect to other cost functions, like $\mathit{depth\_cost}(\mathcal{T}) := \mathit{depth}(\mathcal{T})$ or $\mathit{cost}(\mathcal{T}) := \sum_e |\mathit{leaves}(\mathcal{T}_e)|$. It is conjectured that the tree-indexing problem remains *NP*-complete in both cases.

Our proposed method is not limited to Mocha, but can be applied in other settings where connected entities have to be structured or re-structured. The parameters can be adjusted accordingly. An interesting means to make use of the obtained partitions could be to construct property preserving abstractions based on this particular hierarchy.

In our considered application, the difficulty of the model checking problem relies on the size of the state space, which is typically exponential in the number of modules and—more often than not—turns out to be the bottleneck. Though our automation

seems to cater well for the "Next" heuristic in order to overcome this, comparison with other approaches remain to be made.

The author has considered applying this technique in a timed setting, i.e., where the model contains a real-time component. timed version of the "Next" heuristic. The bricks sorter model from Chapter 6 served here as a motivating example. These attempts, however, remained fruitless—it is the author's (philosophical) conclusion that you just cannot hide time.

Chapter 9

Model Checking Hierarchical Timed Automata

I took a course in speed reading and was able to read "War and Peace" in twenty minutes. It's about Russia.

— Woody Allen

In this Chapter we address the algorithmic verification of the hierarchical timed automata (HTA) model from Chapter 3. Our claim is that presence of the hierarchies does merely complicate the verification part, but not hinder it. In particular we consider the specification language of Uppaal *suitable for specifying properties.*

The foundation for establishing properties of HTAs is the trace-based formal semantics. We do not have a model checking engine for HTAs. Instead we flatten *a HTA model to a* Uppaal *model and make use of the well-engineered implementation of that tool. This translation is complicated mainly by the implicit synchronization on exit. We give first a high-level description and subsequently elaborate to the relevant details. Based on this we sketch a proof for the correspondence of the semantics given to hierarchical and flattened model.*

The flattening procedure has been implemented in **Java** *and translates an XML file format of HTAs to* Uppaal *input files (which are also XML). As a case study we use the model of a cardiac pacemaker, known as a standard UML design example (e.g., [Dou99a]). We model check a safety and a liveness property; the run-times assert that this example is well in the scope of algorithmic treatment.*

9.1 Overview on the Flattening Procedure

Flattening of statechart-like languages is complicated mainly by the presence of transitions that result in a cascade of entries and exits. In particular the synchronization on exit gives rise to complex auxiliary constructs.

In this Section we give an overview description of our flattening procedure. It is subsequently elaborated in Section 9.2.

Flattening a hierarchical timed automaton. On the topmost level of an HTA we find a parallel composition of superstates, conceptually under an implicit root. Each can be of type *AND* or *XOR* and can itself contain superstates. The complete collection of superstates is called the *instantiation tree*.[1] At any point in time the behavior of a HTA depends on the sub-tree of this instantiation tree that is currently active.

Every superstate S in the instantiation tree is translated to one UPPAAL process $\widehat{S}$. All those processes are put in parallel. An auxiliary location in $\widehat{S}$ is added for the configurations where S is not active (i.e., is idle). The translation proceeds in three main phases.

I. *Collection of instantiations:* The instantiation tree is traversed and for every superstate S the skeleton of a (flat) process $\widehat{S}$ is constructed. This contains basic locations, transitions, and the auxiliary initial location $\widehat{S}$_IDLE. Entries to S are translated to guarded transitions from $\widehat{S}$_IDLE.

II. *Computation of global joins:* Transitions originating from superstates can require a cascade of substate exits, called *global join*. All configurations that can synchronize to such a global join are computed. This yields a guard condition that evaluates to **true** if an only if one such cascade can be taken to completion.

III. *Post-processing channel communication:* If a transition in the HTA starts at a superstate S and carries a synchronization, it cannot synchronize with a transition *inside* S. Since the sub-state/superstate relation is lost in the translation, we resolve this conflict explicitly by duplicating channels and transitions.

Correspondence of hierarchical and flattened model. A configuration in the HTA model M corresponds to one configuration in the flattened version $\widehat{M}$. All other configurations of $\widehat{M}$ are either intermediate to this or unreachable. This correspondence allows us to associate every trace of M with one in $\widehat{M}$.

This association dictates the property language for hierarchical timed automata. We sketch this only conceptually. Of main interest are the classes of properties that can be model checked with UPPAAL, see Section 2.3. Consequently, the syntax of properties for hierarchical timed automata is like in Figure 2.3. The difference

[1] In Section 3.1 this corresponds to η.

is that the local properties are required to identify (super)locations, variables, and clocks uniquely. It is necessary to trace back every identifier to the point in the instantiation tree where it is declared. Note that scoping rules allow to override a declarations of x in an ancestor superstate in the instantiation tree. Thus the identifier x can be associated with a different variable, and even a different type, depending on where it occurs.

These scoping problems can be solved via *renaming*. All ambiguities introduced by name duplications can be consistently resolved by prefixing a path of instantiation names to identifiers, starting at the implicit *root*. For simplicity we omit this renaming in our description and treat all variables, clocks, and channels as global. This way for every property φ in the HTA we can compute a corresponding property $\widehat{\varphi}$ for the flattened model, where the identifiers and names of superstates are replaced accordingly.

The subsequent Section 9.2 contains a more detailed description of the flattening procedure. In Section 9.4 we use a cardiac pacemaker as a case study.

9.2 Flattening in More Detail

We now give a detailed description of our flattening procedure. This is organized in three phases: Translation of superstates and their entries, translation of exits, and post-processing of channels.

In their syntactic representation via XML files, both the hierarchical timed automata model and then Uppaal model rely on a *template mechanism*. Templates for superstates (processes) are instantiated to create the concrete superstates (processes) that constitute the actual model. This works very much like instantiation of classes to objects, and the motivation is also similar. It should be easy to make small consistent modifications, e.g., via setting parameters. Parts that are (nearly) identical should not be described twice but derived as two instantiations of the same template. The implementation of our flattening procedure therefore in fact translates a set of HTA templates plus an instantiation at root level to a set of flat timed automata templates where each is instantiated exactly once.

Conceptually, however, the translation works on instantiation level. If a superstate template is instantiated twice, the two instantiations are translated separately. This makes it easier to take the context into account. At template level, e.g., no parent superstate can be attributed to a template. To construct translations of entries or exits, knowledge about this context is crucial. For simplicity we therefore describe the translation as if all superstates and processes were primitives.

9.2.1 Translation of Superstates and Entries — Phase I

We sketch now the translation of a superstate S to a process $\widehat{S}$, the pseudo-code is given in Figure 9.1.

For every location l in S, $\widehat{l}$ is created in $\widehat{S}$. Additional $\widehat{S}$ contains the location

Algorithm: *PHASE I: instantiateTemplates*

input: Stack $\mathfrak{S}$ of superstates to translate
output: Set **P** of (flat) timed automata
Set **G** of global join starting points

$\mathbf{P} := \{$*Global_Kickoff* automaton for $s \in \mathfrak{S}\}$
$\mathbf{G} := \varnothing$
WHILE *notempty*($\mathfrak{S}$)
  $S := pop(\mathfrak{S})$
  $\mathcal{C} := \{$non-basic locations B in $S\}$
  FORALL $B \in \mathcal{C}$
    $push([B \text{ in } S], \mathfrak{S})$
    /⋆ $[B$ in $S]$ inherits all invariants attached to S ⋆/
    create a location $\widehat{B}$ in $\widehat{S}$
    $E_B := \{$set of entries of B in $S\}$
    FORALL $e \in E_B$
      create a committed location $\widehat{B_e}$ in $\widehat{S}$
      create a transition from $\widehat{B_e}$ to $\widehat{B}$ in $\widehat{S}$
      /⋆ this transition carries a synchronization `enter_`B`_in_`S`_via_`e! ⋆/
    IF *type*(S) = *XOR* THEN
      $\mathbf{G} := \mathbf{G} \cup \{B \text{ in } S\}$
  $\mathbf{P} := \mathbf{P} \cup \{$translation $\widehat{S}$ of superstate S, depending on *type*(S)$\}$

Figure 9.1: Algorithm for Translation of the Instantiation Tree.

S`_IDLE`, which is the initial location. Every entry of S corresponds to a transition in $\widehat{S}$ originating from S`_IDLE`. Some auxiliary constructions are necessary to mimic the behavior of hierarchical machines adequately. They depend on the type (*XOR* or *AND*) of S.

Translation of *XOR* superstates. In a hierarchical *XOR* superstate X, at most one location is active at a given time. For every substate B of X we introduce a location B`_ACTIVE_IN_`X in $\widehat{X}$. Moreover, for every entry e of B we introduce an auxiliary location in $\widehat{X}$, called X`_AUX_`B`_`e. These are declared committed and are connected to B`_ACTIVE_IN_`X with a transition that synchronizes on a channel `enter_`B`_in_`X`_via_`e. Transitions leading originally to a B-entry e in X are represented in the translation by leading to X`_AUX_`B`_`e and trigger—without interleaving with other processes—the activation of the substate B.

Exits of substates B are translated similarly by transitions from B`_ACTIVE_IN_`X. They give rise to additional complications since leaving an *AND* substate B is only possible if all descendants of B can exit. So in fact a chain of exit transitions starting at B`_ACTIVE_IN_`X can be necessary, see Section 9.2.2.

If the *XOR* superstate X is inactivated (exited), this corresponds in the trans-

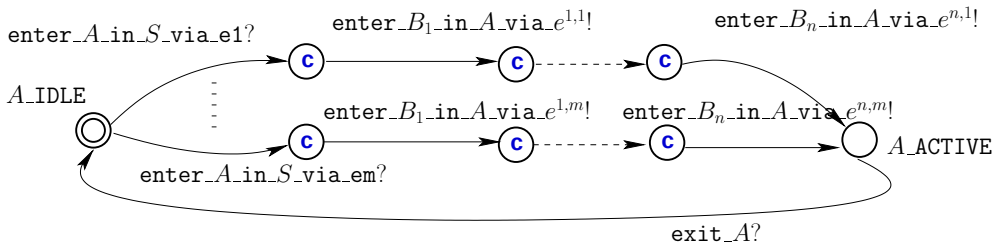


Figure 9.2: Translation of Entries and Exits an *AND* Superstate.

lation $\widehat{X}$ to transitions to X_IDLE. This transition carries the synchronization exit_X?. If the superstate X has a default exit, every non-auxiliary location in $\widehat{X}$ has such a transition to B_IDLE.

Translation of *AND* superstates. An *AND* superstate A is a parallel composition of superstates. Either non of them is active or all of them are. In the translation $\widehat{A}$ (Figure 9.2), this corresponds to locations A_IDLE and A_ACTIVE. If A is activated, this is specific to an entry e_i of A. The substates B_j of A are entered one after another. Which entry is used for each B_j is dependent on e_i. Thus for every entry e_i of A there is a separate chain of transitions leading from A_IDLE to A_ACTIVE. The choice of entries of B_j is reflected by appropriate signals enter_B_j_in_A_via_$e^{j,i}$. The auxiliary locations in the chain are declared committed, so no time can elapse before A_ACTIVE is reached and interleaving with other processes is blocked.

Kickoff. Since the root of the instantiation tree is implicit, one special process is needed to trigger the entry of the topmost superstates. This process is called Global_Kickoff and also initializes all variables.

We note that the topmost superstates S_i are considered special, since they do not synchronize on exit. Instead, they can be enabled to become in-active via following a special exit transition. Once one of these S_i becomes inactive, this status can never be revoked in our hierarchical timed automaton formalism, since there is no machine that could accommodate a transition *to* some S_i. If a superstate S is intended to be able to be both inactivated and activated again, it cannot nest at the root level but must be itself contained in a superstate.

History. History amounts to record the status of an *XOR* superstate X when it is exited. Since we assume all variables and clocks to be global, this amounts to storing the last control location. This can be encoded via an auxiliary integer variable *hist* that is updated along each transition in $\widehat{X}$. Each value corresponds exactly to one location $\widehat{l_i}$ in $\widehat{X}$. The history entry then has a transition to each location $\widehat{l_i}$ guarded by the expression $hist == i$. If *hist* has its initial value 0, then then the only guard evaluating to **true** leads to the default history location.

The clocks local to superstates with history entry are not frozen on exit but kept

running.[2] If local clocks are declared to be forgetful, then they are reset along every entry. Otherwise they resume with the accumulated value.

For simplicity we do not treat history in our flattening procedure.

Urgent transitions. In the HTA formalism transitions can be declared urgent. The corresponding concept in the UPPAAL model is to declare channels urgent, i.e., channels where synchronization has preference over time delay. An urgent transition t can be encoded by this as follows.

a) If t does not carry synchronization:
Add a dummy synchronization `Hurry?` on the transition and add one parallel process `HurryDummy` that constantly offers synchronization on this channel.

b) If t synchronizes on channel c:
Declare c urgent. If there are situations where two non-urgent transitions can synchronize on c, then it is necessary to introduce a urgent and non-urgent copy of c and duplicate all transitions where both urgent and non-urgent synchronizations are possible.

For simplicity we do not treat urgency in our flattening procedure.

9.2.2 Exit of Superstates via Global Joins — Phase II

The exit of a superstate S is represented in $\widehat{S}$ by a transition to S`_IDLE` which carries the synchronization signal `exit_`S?. These exits do not necessarily happen in isolation, but might happen as part of a cascade of exits from superstates and non-basic substates. Thus it is necessary

(1) to derive conditions that allow a set of superstates to exit, and
(2) to make sure that always the complete set of exits is performed.

We call the process of performing a legal set of exits a *global join*.

Example 9.1 (Global Join)
Consider Figure 9.4 (i) with control at (`L2`,`L3`*). Then the superstates* `S3`, `S2`, *and* `S1` *have to be left, in order to reach* l. *The same holds for control situation (*`L2`$'$, `L3`*). This cascade of exits is encoded the sequence of in Figure 9.4 (ii). However, if control is at (*`L2`,`L4`*), then* `S4` *must be left as well, this would correspond to a* different *sequence of substate exits than displayed in (ii), i.e., a different global join.*

One transition leaving a superstate B can give rise to a number of global joins, possibly exponential in the depth of hierarchical structure.

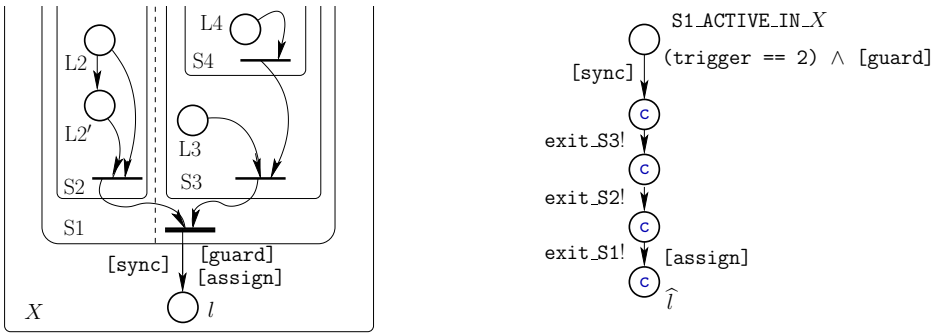
For every global join there is exactly one proper transition that does not lead to an exit. In Example 9.1 this is the transition to l. An auxiliary variable `trigger` keeps track of the number of active basic locations that can participate in this join.

[2] Reachability for automata with stopwatches is undecidable [CL00].

Algorithm: *PHASE II: expandGlobalJoins*
input: Set **G** of global join starting points
output: auxiliary constructions: counters and guarded transitions

JoinTrees := $\varnothing$
FORALL $\mathbf{g} \in \mathbf{G}$
collect all trees **T** of control locations that can synchronize to **g**;
the leaves of **T** are sets of basic locations that share transitions to the same exit e.
/⋆ These sets are singletons, if e is an ordinary exit
and span over all basic locations in the superstate otherwise ⋆/
JoinTrees := *JoinTrees* $\cup$ $\{\mathbf{T}\}$
FORALL $\mathbf{T} \in$ *JoinTrees*
let $\widehat{L} := \{\widehat{l} \,|\, l$ is element in a basic location set of $\mathbf{T}\}$
declare the counter $trigger_{\mathbf{T}}$
FORALL $\widehat{l} \in \widehat{L}$
FORALL transitions $\widehat{k} \to \widehat{l}$
add the assignment $trigger_{\mathbf{T}} := trigger_{\mathbf{T}} + 1$ to $\widehat{k} \to \widehat{l}$
FORALL transitions $\widehat{l} \to \widehat{m}$
add the assignment $trigger_{\mathbf{T}} := trigger_{\mathbf{T}} - 1$ to $\widehat{l} \to \widehat{m}$
let $N :=$ number of leaves of **T**
let $\mathcal{S}_{\mathbf{T}} :=$ superstates S occurring in **T**
FORALL transitions t starting at $root(\mathbf{T})$
create a chain of transitions, starting with $\widehat{t}$,
corresponding to exiting every $S \in \mathcal{S}_{\mathbf{T}}$
/⋆ see Figure 9.4 (ii); note the additional guard $trigger_{\mathbf{T}}$==N ⋆/

Figure 9.3: Pseudo-code for the Encoding of All Global Joins.



(i) Part of an XOR Superstate X *(ii) Exits in $\widehat{X}$, starting at (L2,L3)*

Figure 9.4: Translation of a Global Join That is Rooted at *XOR* Superstate X.

In a transition from `L2` to `L2′`, for example, the value of `trigger` does not change. `trigger` has to reach the threshold value—here: 2—to enable the global join. It is

Algorithm: *PHASE III: postprocessChannels*
input: *Queue* Q over (*syncSignal*, *transition*, S)

WHILE *notempty*(Q)
($syncSignal$, $transition$, S) :=*pop*(Q)
IF $\exists$ transition t with *match*(*syncSignal*) in S:
create a new channel c
replace *channel*(*syncSignal*) on *transition* by c
FORALL transitions t with *match*(*syncSignal*) outside S
create a copy t' of t, where *channel*(*syncSignal*) is replaced by c
if $\exists(s', t, S') \in Q$ then $push\big((s', t', S'), Q\big)$

Figure 9.5: Pseudo-code for Post-processing Synchronization Channels.

crucial that the proper transition terminating the global join—here: `S1` to l—can be taken, i.e., that the guard (if any) evaluates to **true**. Likewise the synchronization with other transitions (if any) has to be possible at this point in time.

Thus, in the sequence of substate exits in Figure 9.4 (ii), `[guard]` and `[sync]` are attached to the *first* transition, while `[assign]` is executed along the last transition.

9.2.3 Post-Processing of Channels — Phase III

Transitions that cause the same location to be exited are in conflict, i.e., they cannot be executed simultaneously. The only case where two transitions in the HTA model are taken truly simultaneous (and not interleaved) is the synchronization along channels. E.g., in Figure 3.1, the a? transition exiting SUB cannot synchronize with the a! transition in P.

In the flattening the structural relation of ancestor/descendant is lost. Therefor we have to prevent synchronization between the processes $\widehat{\mathsf{SUB}}$ and $\widehat{\mathsf{P}}$ explicitly. We achieve that by introducing *duplications* of channels such that synchronization is guaranteed to happen between processes that correspond to parallel superstates. This can make it necessary to also introduce duplications of transitions.

For example, the HTA in Figure 3.1 is flattened such that channel a is replaced by two copies a_parallel_P and a_parallel_MAIN. One can synchronize with superstates parallel to P and one with superstates parallel to MAIN. The signals a! and a? along channel a have to be replaced accordingly.

Parts of the flattened model are drawn in Figure 9.6. If a superstate is both parallel to P and to MAIN, a transition originally carrying a! is replaced by two transitions, one carrying a_parallel_P! and one carrying a_parallel_MAIN!. The pseudo-code for this post-processing is given in Figure 9.5.

9.3 Semantic Correspondence of HTAs and TAs

Hierarchical and flattened model are related in that with every hierarchical configu-

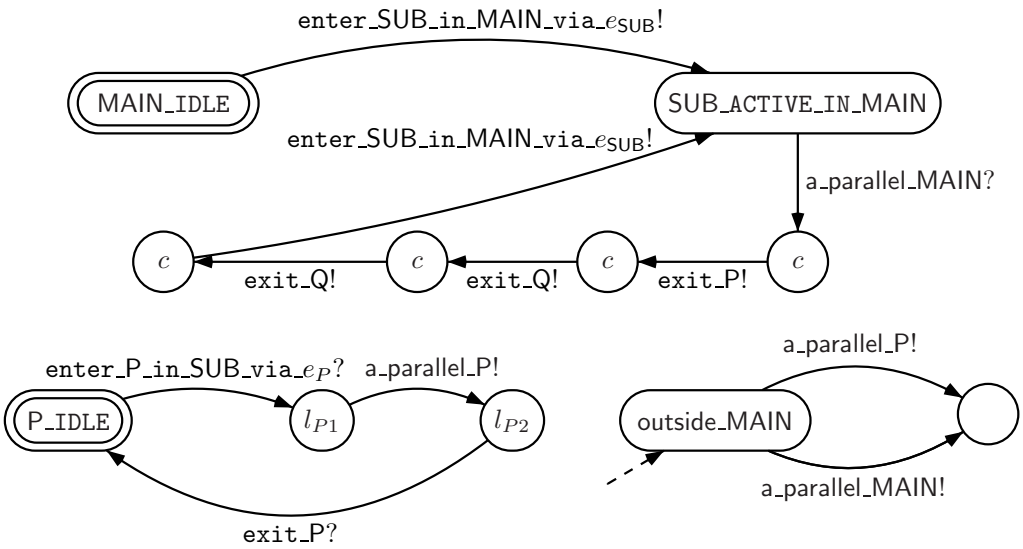


Figure 9.6: Part of the Flattened Model of the HTA in Figure 3.1 After Phase III.

ration we can associate a flat one. We show that every hierarchical trace corresponds to a projection of a trace in the flattened version. A similar connection holds in the other direction. It follows that both models are equivalent with respect to the TCTL properties checkable with Uppaal.

9.3.1 Hierarchical and Flat Configurations

Conceptually we can relate a configuration of a HTA M to a configuration of the flattened Uppaal model $\widehat{M}$. The reverse direction is not possible in general; some configurations of the Uppaal model do not make sense from the HTA point of view, e.g., if a process corresponding to a substate is active but not the process corresponding to its superstate. Our construction guarantees that those configurations are not reachable. Other configurations in the Uppaal model are intermediate steps in the encoding of an exit or entry. We call those configurations of the Uppaal model that have a counterpart in the hierarchical model *stable*.

Definition 9.2 (Stable/Unstable Configuration)
Given a HTA M and a corresponding Uppaal *model $\widehat{M}$, where every superstate S in M corresponds to the process $\widehat{S}$ in $\widehat{M}$. A* stable configuration *of $\widehat{M}$ then is a configuration $(\vec{l}, e, \nu)$, where*

- *No $l \in \vec{l}$ is committed, i.e., $\forall i.\ \neg c(l_i)$,*
- *If X is a XOR superstate and for some S X_ACTIVE_IN_$S \in \vec{l}$, then X_IDLE $\notin \vec{l}$, and*
- *If A is a AND superstate and for some S A_ACTIVE_IN_$\widehat{S} \in \vec{l}$, then for every substate B_i of A: B_i_IDLE $\notin \vec{l}$.*

Every consistent Uppaal *model configuration that is not stable is called* unstable.

We can define a relation of configurations of a HTA M to stable configuration of $\widehat{M}$.

Definition 9.3 (Matching Configuration)
Given a HTA M and a proper configuration $\mathbf{c} := (\rho, \mu, \nu, \theta)$ of it. A configuration $\mathbf{s} := (\vec{l}, e, \nu)$ of $\widehat{M}$ is a matching configuration*, in symbols $\mathbf{c} \sim^M \mathbf{s}$ if the following holds.*

(i) $\forall S \in \rho^+(root).\ BASIC(S) \ \Rightarrow\ \widehat{S} \in \vec{l}$,

(ii) $\forall S \in \rho^+(root).\ XOR(S) \vee AND(S) \ \Rightarrow\ S\_\texttt{ACTIVE\_IN\_}(\eta^{-1}(S)) \in \vec{l}$, *and*

(iii) $\forall v \in Var(root).\ \mu(v) = e(v)$

It is easy to see that the flat configuration in the above definition is necessarily stable. The relation $\sim^M$ ignores history and the values of auxiliary variables. In general $\sim^M$ is an injection. By construction of the steps, however, for every reachable hierarchical configuration $\mathbf{c}$ only one flat configuration $\mathbf{s}$ is reachable.

9.3.2 Correspondence of Steps

The flattened version $\widehat{M}$ of a HTA M is a *refinement* in the sense that every step in M corresponds to a finite sequence of steps in $\widehat{M}$. If an ordinary transition or a delay is mimicked this sequence is of length 1. The exit and entry of superstates require a larger number of steps to be taken in the flattened version.

Delay. A delay step of duration d is possible if no urgent transition is enabled and all invariants remain **true** throughout this delay. In phase I, all invariants of superstates are inherited, i.e., every location in the flattened model carries a conjunction of the invariants of all ancestor superstates it is derived from. Thus, a duration step from a HTA configuration $\mathbf{c}$ is possible if and only if it is possible in a corresponding flat $\mathbf{s}$ with $\mathbf{c} \sim^M \mathbf{s}$.

Join. The computation of *PreExitSets*(e) in Section 3.2 corresponds to the sets of locations that are computed in *expandGlobalJoins*. Recall that *PreExitSets*(e) is a family of sets of basic locations. The global join can be taken if control is such that one location in each set is active. These sets are locations in the same *XOR* superstate, thus not more than one can be active. For the global join $\mathbf{g}_i$ the auxiliary variables ($\texttt{trigger}_i$) reflects the number of locations that are in the sets of $\mathbf{g}_i$, i.e. *PreExitSets*(e). If this number reaches the threshold $|$*PreExitSets*$(e)|$, the global join can be taken.

Every such performed global join relies on one proper transition t that does not lead to an exit. t is necessarily part of a *XOR* superstate X. The encoding of the global join is a chain of transitions (like in Figure 9.4 (b)). The first transition carries

guard and synchronization of t. The subsequent transitions signal the substates B_i of X to become idle, i.e., the processes $\widehat{B_i}$ corresponding to these substates take a transition to B_i`_IDLE`. Since the intermediate locations of the chain are declared committed, this sequence cannot be disturbed by ordinary transitions or time delays.

If t synchronizes (with a transition parallel to t) this can entail two simultaneous executions of global joins and, possibly, also entries of substates. Since the transitions are necessarily parallel (or: *independent*), this does not cause problems. There might be several legal sequences of transitions that lead to the same next stable configuration.

Transition. A simple action step that does not exit or enter any superstates corresponds naturally to taking one transition in a (flat) process. In the flattened model, auxiliary variables (`trigger`) are updated along this transition. This is merely housekeeping and does not enable or block transitions. The invariants of locations are inherited. Thus the transition part of the HTA is directly mimicked in the translation.

The analogous argument holds for the synchronization of two transitions along a channel. The renaming in phase III guarantees that synchronizations are only possible between transitions that correspond to parallel transitions in the HTA.

Fork. Entries of *XOR* superstates activate one location that can be basic or a superstate. Entries of *AND* superstates activate all substates; those are necessarily superstates again. Thus every entry can result in the activation of a set of superstates. This set is given by the (static) structure.

In the flattened version this set of superstates is activated by adding auxiliary locations and synchronizing via `enter_`B`_in_`S`_via_`e!. There are no guards allowed and the auxiliary locations are declared committed. Thus this sequence of synchronizations takes place without interleaving with ordinary transitions and without time delay.

It is important that all parts, once started, can execute to completion. Thus we can relate one step in a HTA M to a sequence of steps in $\widehat{M}$, where only the first and the last configurations are stable.

Lemma 9.4 (Step Encoding)
For a HTA M there exist a step between two configurations (ρ, μ, ν, θ) and $(\rho', \mu', \nu', \theta')$ according to rules action *and* sync *(see Section 3.2) if and only if for the* Uppaal *model $\widehat{M}$ there exists a corresponding sequence*

$$(\vec{l}, \mathrm{e}, \nu) \stackrel{\alpha}{\Longrightarrow} (\vec{l}_1, \mathrm{e}_1, \nu_1) \stackrel{\tau}{\Longrightarrow} \cdots \stackrel{\tau}{\Longrightarrow} (\vec{l}_k, \mathrm{e}_k, \nu_k) \stackrel{\tau}{\Longrightarrow} (\vec{l}', \mathrm{e}', \nu')$$

where $(\rho, \mu, \nu, \theta) \sim^M (\vec{l}, \mathrm{e}, \nu)$, $(\rho', \mu', \nu', \theta') \sim^M (\vec{l}', \mathrm{e}', \nu')$, all $(\vec{l}_i, \mathrm{e}_i, \nu_i)$ are unstable configurations, $\alpha \in \{a, \tau\}$ and the remaining synchronizations τ are along channels `exit_`B *and* `enter_`B`_in_`S`_via_`e.

Other modeling elements. We do not address history or urgency in our argumentation. This is for the sake of clarity; they are not causing complications.

History amounts to the assignment of special variables that direct control on reentry. In the flattened version this yields a mutual exclusive choice of the transitions from the history entry to exactly one location (which can be in fact a superstate; then either the history entry or default entry is used). Along this transitions only those clocks declared as forgetful are reset to 0 and all others remain untouched.

Urgency can be completely replaced by UPPAAL's mechanism for synchronization on urgent channels as explained earlier.

9.3.3 Correspondence of Traces

After asserting that the step relation of a HTA M is indeed refined to the step relation of the flattened $\widehat{M}$, we can relate the sets of traces. The key observation is that for every timed trace in M there exits at least one corresponding timed traces for $\widehat{M}$. For every timed trace for $\widehat{M}$ there exists exactly one timed trace for M.

The trace relation is not a bijection, since in $\widehat{M}$ interleavings between the intermediate transitions are possible. This is only the case for synchronized action steps, which are guaranteed to connect only independent transitions. Thus all such interleavings lead to the same stable configuration.

Proposition 9.5 (Correspondence of Hierarchical and Flattened Model) *Given a HTA M and the flattened* UPPAAL *model $\widehat{M}$ of it. For every timed trace $\sigma = \{(\rho, \mu, \nu, \theta)_i\}_{i\geq 0}$ of M there exists a corresponding timed trace $\widehat{\sigma} = \{(\vec{l}, \mathrm{e}, \nu)_j\}_{j\geq 0}$ of $\widehat{M}$ such that*

$$\begin{aligned} \forall i.\ \exists k, k',\ k < k'.\quad & (\rho, \mu, \nu, \theta)_i \sim^M (\vec{l}, \mathrm{e}, \nu)_k && \wedge \\ & (\rho, \mu, \nu, \theta)_{i+1} \sim^M (\vec{l}, \mathrm{e}, \nu)_{k'} && \wedge \\ & \forall k < j < k'.\ (\vec{l}, \mathrm{e}, \nu)_j \text{ is unstable.} \end{aligned}$$

Conversely, for every timed trace $\widehat{\sigma} = \{(\vec{l}, \mathrm{e}, \nu)_j\}_{j\geq 0}$ of $\widehat{M}$ there exists a corresponding timed trace $\sigma = \{(\rho, \mu, \nu, \theta)_i\}_{i\geq 0}$ of M such that

$$\begin{aligned} \forall k, k',\ k < k'.\quad & \text{if } (\vec{l}, \mathrm{e}, \nu)_k \text{ and } (\vec{l}, \mathrm{e}, \nu)_{k'} \text{ are stable} \\ & \text{and all } (\vec{l}, \mathrm{e}, \nu)_j \text{ with } k < j < k' \text{ are unstable, then} \\ & \exists i.\ (\rho, \mu, \nu, \theta)_i \sim^M (\vec{l}, \mathrm{e}, \nu)_k \quad \wedge \\ & \quad (\rho, \mu, \nu, \theta)_{i+1} \sim^M (\vec{l}, \mathrm{e}, \nu)_{k'}. \end{aligned}$$

Observe also that by construction the entries and exits cannot get "stuck" in the middle of the transition. Thus $\widehat{M}$ does not yield maximally extended finite traces that terminate in unstable configurations. This entails that all trace properties that UPPAAL can establish for $\widehat{M}$, also hold for M.

Corollary 9.6 (Flattening Sound and Complete) *A timed property φ from the TCTL fragment in Section 2.3 holds in an hierarchical model M if and only if the the corresponding property $\widehat{\varphi}$ holds in $\widehat{M}$.*

Proof: (Sketch)
By Proposition 9.5 the sets of traces match modulo the unstable configurations contained in the traces of $\widehat{M}$. Local properties of M cannot refer to the auxiliary variables in the unstable configurations and by our well-formedness conditions the values of variables in $Var(root)$ change at most once along a sequence of unstable configurations.

For the TCTL fragment in Section 2.3 it suffices to quantify over traces. The hierarchical and the flat traces are only distinguishable by the names of identifiers. Those we assume to be translated properly in $\widehat{\varphi}$.

□

9.4 Model Checking a Cardiac Pacemaker

We exemplify our flattening procedure on the model of a cardiac pacemaker. The flattened version is model checked with UPPAAL *for a safety and a liveness property. The code of the procedure and the example are available online.*[3]

The pacemaker is put in parallel with a model of a human heart and a programmer. We translate the hierarchical timed automaton model of this composition to an equivalent (flat) UPPAAL *timed automata model and explain the obtained automata in detail. Then we report on run-time data of the formal verification of this translation with respect to safety and response properties.*

9.4.1 The Hierarchical Timed Automaton Model

The hierarchical model is a parallel composition of three *XOR* superstates: the human heart, the cardiac pacemaker itself, and a programmer setting up the pacemaker.

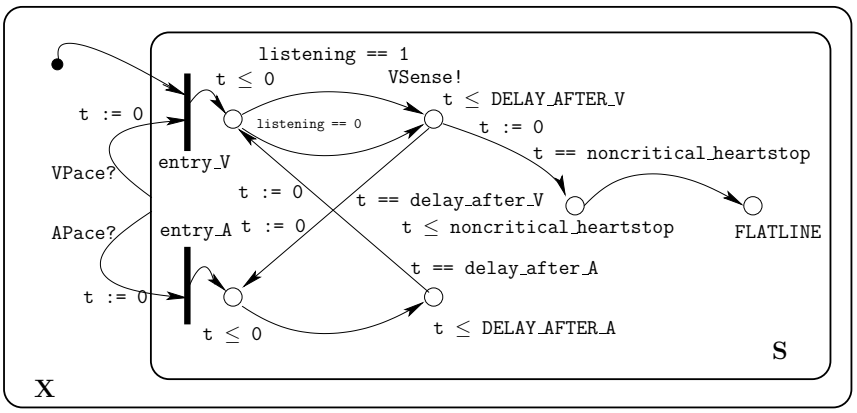


Figure 9.7: Model of a Human Heart That Might Require Pacing.

[3] http://www.brics.dk/%7Eomoeller/hta/vanilla-1/

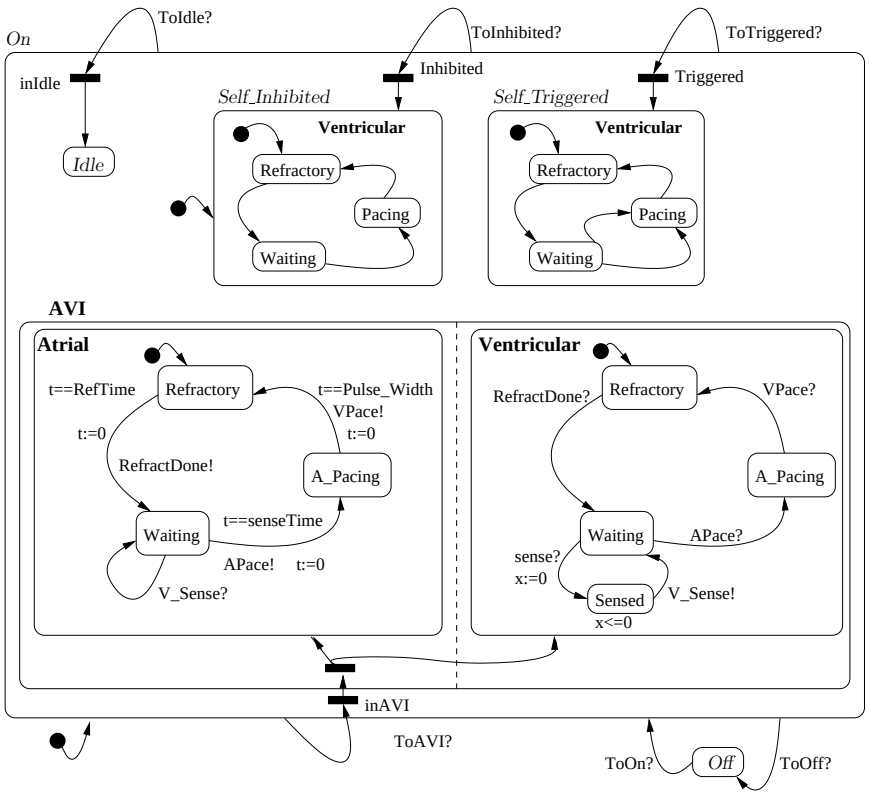


Figure 9.8: Model of the Pacemaker. Initially *Self_Inhibited* is Entered.

Heart model. The human heartbeat is in fact 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 might require pacing (Figure 9.7). We consider only two chambers, namely the (left) atrial and ventricular ones. A healthy heart contracts those in a steady rhythm. We mimic this by the time delays `DELAY_AFTER_V` and `DELAY_AFTER_A` and the local clock `t`. 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 might non-deterministically stop beating on own account. 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 channels `APace` or `VPace`. The particular heart chamber then is scheduled for contraction in the very 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.

We use the local clock `t` to model this rhythm. Since in our example we only mon-

itor 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 might non-deterministically stop beating on own account. 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 very 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 an *XOR* superstate with the two sub-states *Off* and *On*. If the pacemaker is on, it can 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* superstate with two parallel substates. In our example only the ventricular chamber is observed, but a pace signal may be sent either 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. We include two additional locations, *Random* and *Idle*, to encode alternative behavior of the programmer. They are not relevant here.

9.4.2 Translation to UPPAAL Timed Automata

The three superstates `Heart`, `Pacemaker`, and `Programmer` are flattened to a network of UPPAAL processes. In particular this translation yields

- two processes for the `Heart`: a top-level, where exit and re-entry happens and one for the substate where the heart is beating (Figure 9.9),

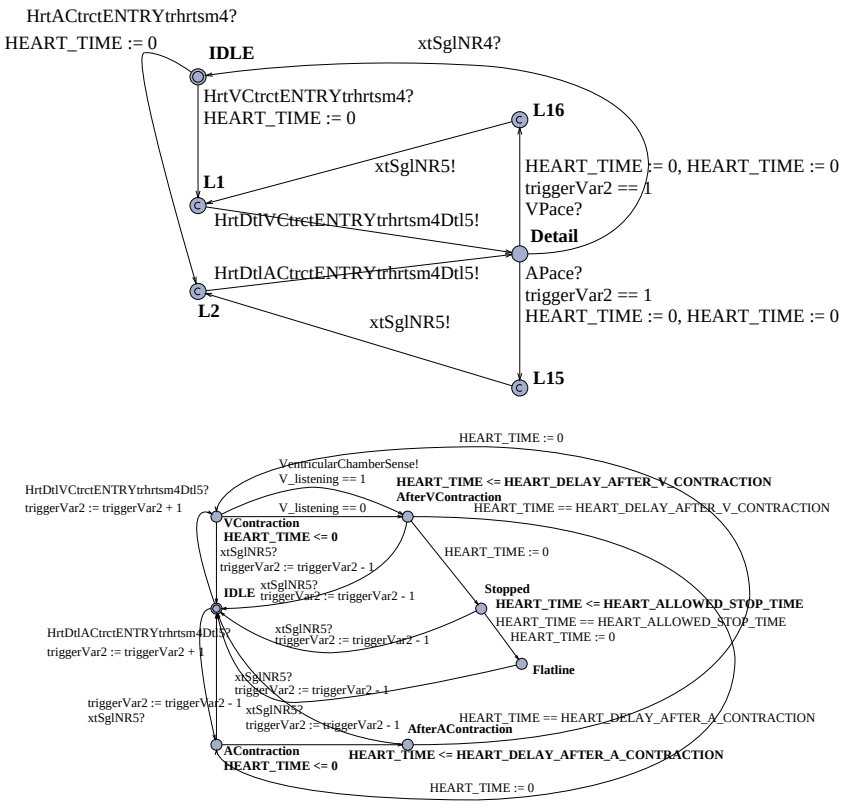


Figure 9.9: Flattened Version of the Heart Model.

- seven processes for the `Pacemaker`, put together as
  - one process for the top-level where the pacemaker is either *On* or *Off* (Figure 9.10),
  - one process for superstate where the pacemaker is on (Figure 9.11),
  - one process for the VVI operation mode (Figure 9.12),
  - one process for the VVT operation mode (Figure 9.13),
  - three processes for the AVI operation mode, one for the *AND* superstate (Figure 9.14) and two for the substates listening to the ventricular chamber (Figure 9.15) and pacing the articular chamber (Figure 9.16),
- one process for the `Programmer` (Figure 9.17), and
- one process to start the three parts (Figure 9.18).

Translation of heart (Figure 9.9). The *XOR* superstate **X** and the *XOR* substate **S** are translated to the two processes. The translation of **X** (upper part of Figure) is responsible for selecting the entry `VContraction` or `AContraction`. The translation of **S** (lower part of Figure) encodes the behavior. Note that from every location there is a transition to `IDLE`; this corresponds to the default exit of **S**.

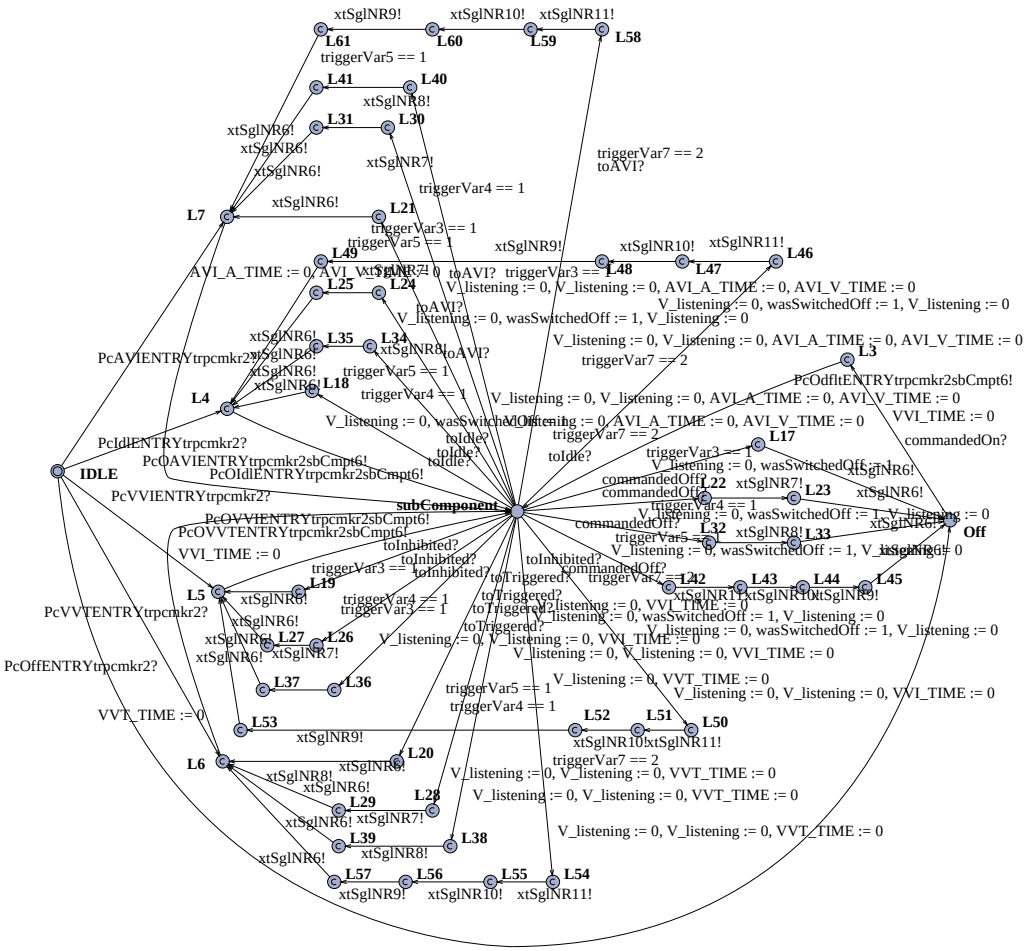


Figure 9.10: Translation of the Topmost *XOR* Superstate of the Pacemaker.

Flattened pacemaker (Figures 9.10, 9.11, 9.12, 9.13, 9.14, 9.15, 9.16). The most complicated process is the translation of the topmost *XOR* superstate. The basic locations are `IDLE` (far left), `subComponent` (center), and `Off` (far right). The pacemaker is *on*, when it control resides in `subComponent` and *off*, when the control is at `Off`.

The committed locations serve to encode the entry of the single substate and the global joins originating from it. For example, the four locations on the left `L4`, `L5`, `L6`, and `L7` correspond to entering the modes `Idle`, `VVIMode`, `VVTMode`, and `AVIMode`. Control of the pacemaker can reside in the locations `Idle`, `VVIMode`, `VVTMode`, and `AVIMode`. There are no direct transitions between these modes, the superstate has to be exited to change in between them.

The AVI mode is modeled by a *AND* superstate with two parallel *XOR* substates. In the translation this is reflected by a process with two non-committed locations `IDLE` and `ACTIVE` (Figure 9.14) that synchronizes with two other processes AVI-A

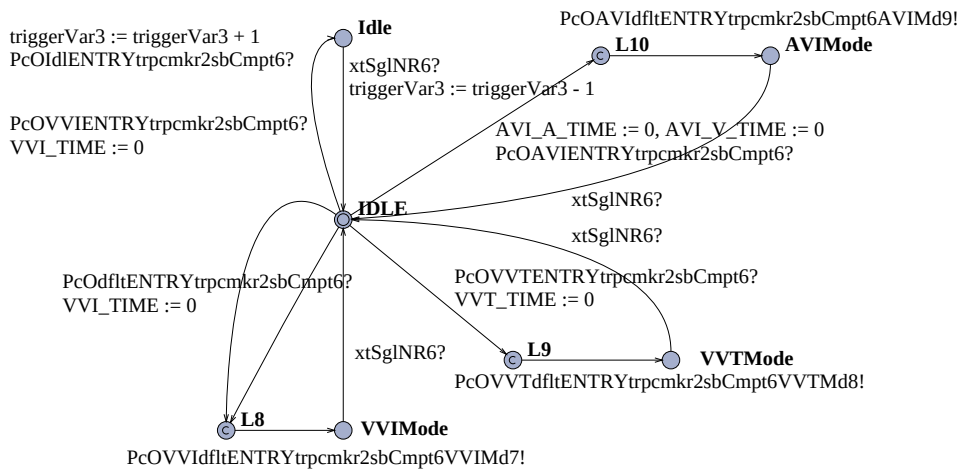


Figure 9.11: Translation of the *XOR* Superstate *On*.

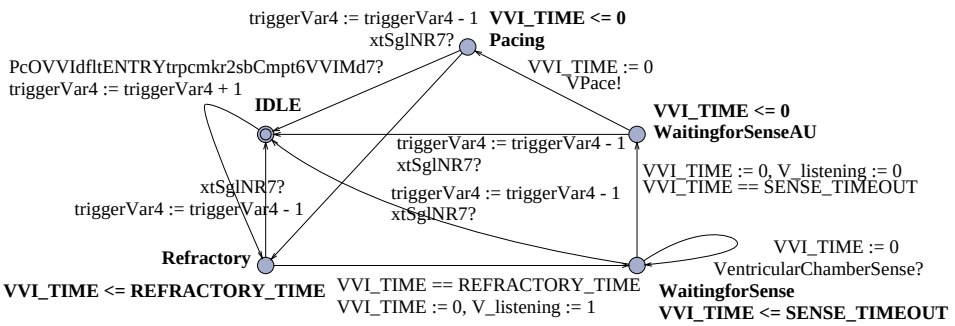


Figure 9.12: Translation of the *XOR* Superstate Corresponding to the VVI Mode.

and AVI-V (Figures 9.16,9.15).

Translation of programmer (Figure 9.17). Since the programmer is a *XOR* superstate with only basic locations, the translation is very similar. It contains the additional location `IDLE`.

Kickoff (Figure 9.18). This process starts the three superstates `Heart`, `Pacemaker`, and `Programmer`. In the only process of the UPPAAL model where in the initial configuration a transition is enabled.

Increase in Model Size

Both data formats, HTA and UPPAAL timed automata, are described in terms of XML grammars. The flattening of the HTA yields an moderate increase in terms of model size. Table 9.1 lists this data in detail. A large number of committed locations

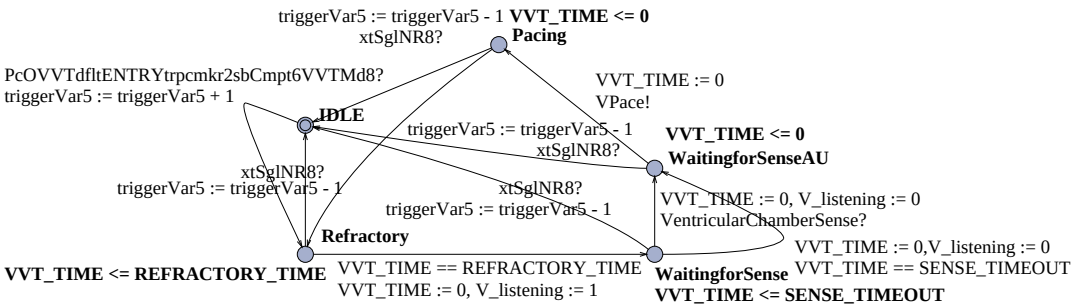


Figure 9.13: Translation of the *XOR* Superstate Corresponding to the VVT Mode.

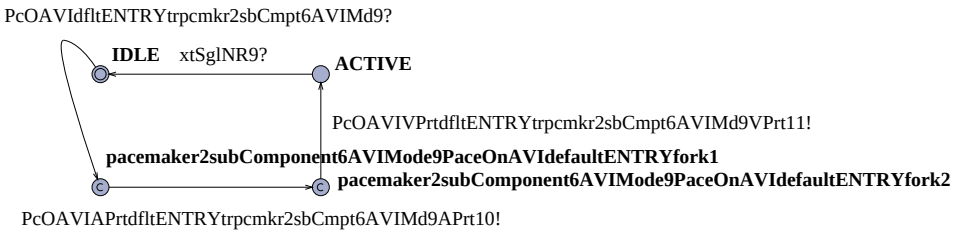


Figure 9.14: Translation of the *AND* Superstate Corresponding to the AVI Mode.

	HTA model	Uppaal model
# XML tags	564	1191
# proper control locations	35	45
# pseudo-states / committed locations	33	63
# transitions	47	177
# variables and constants	33	72
# formal clocks	6	6

Table 9.1: Size of the HTA Model and the Corresponding Uppaal Model.

were introduced to encode entry and global joins. However, these forks and joins are triggering a deterministic sequence of actions and thus do not significantly increase the state space. A similar observation holds for the introduced auxiliary variables:

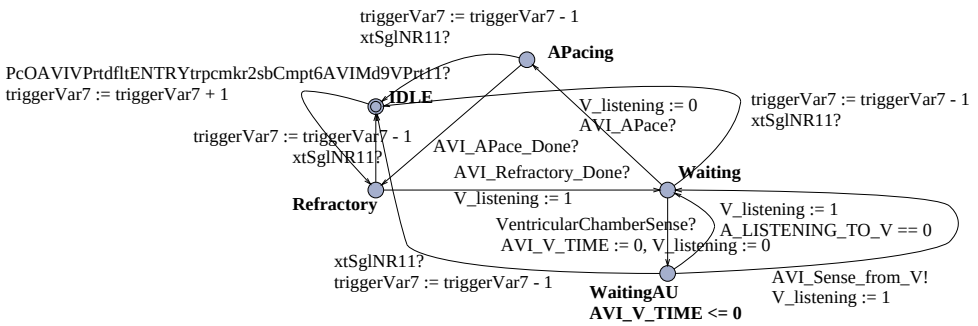


Figure 9.15: Translation of the *XOR* Superstate AVI-V.

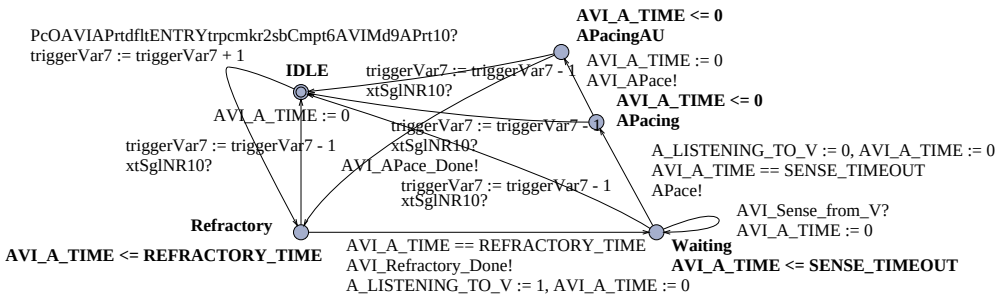


Figure 9.16: Translation of the *XOR* Superstate AVI-A.

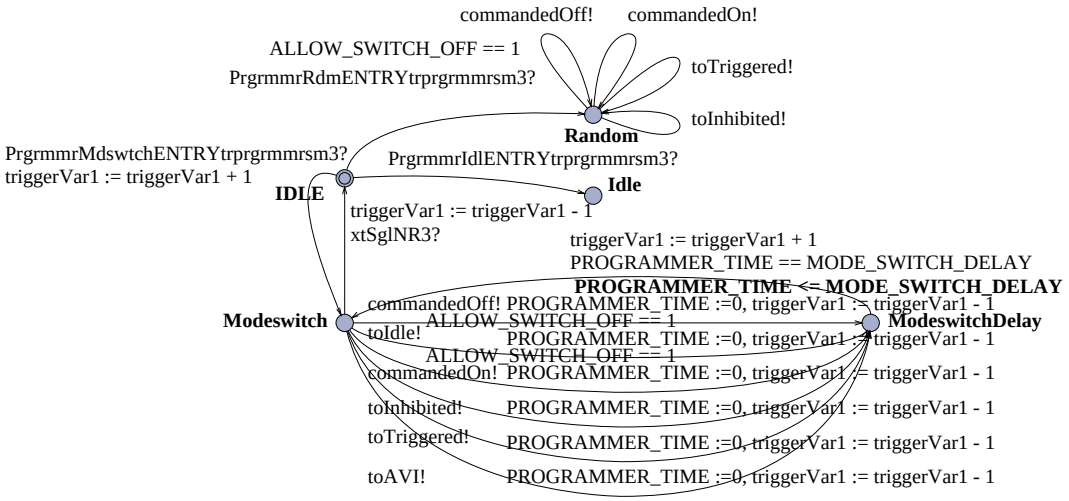


Figure 9.17: Translation of the *XOR* Superstate `Programmer`.

The values of variables triggering global joins are completely determined by the current control state. The auxiliary channels introduced to switch components from `IDLE` to `ACTIVE` and vice versa does not increase the complexity significantly.

9.4.3 Model Checking the UPPAAL Model

The translation of the HTA model can serve as input to the UPPAAL tool. The system is not deadlock free. When the programmer switches off the pacemaker and the heart stops beating, a configuration is reached where unbounded delay is possible. In one variation, the programmer was explicitly disallowed to exit. In a second variation, the pacemaker could not be switched off. In both variations, deadlock freedom was established via a run of the model checking engine on a true invariant with switch settings `-Aa` (convex hull approximation and active clock reduction

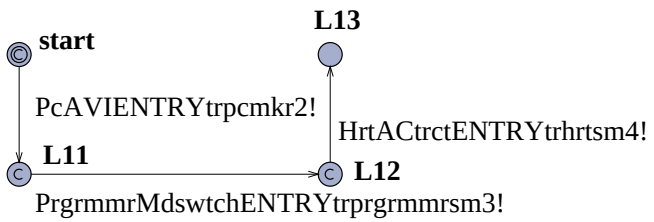


Figure 9.18: The Additional `KickOff` Process.

switched on), and took 3.50 respectively 1.75 seconds.

We verified two desirable properties in the (non-variated) obtained hierarchical timed automaton model.

(i) `A[] ( heart_sub.FLATLINE => (wasSwitchedOff == 1) )`
(ii) `A[] ( heart_Sub.AfterAContraction => A<> heart_Sub.AfterVContraction )`

Property *(i)* is a safety property and states that the heart never stops for too long, unless the pacemaker was switched off by the programmer (in which case we cannot give any guarantees). Property *(ii)* is a response property and states that after an articular contraction, there will *inevitably* follow a ventricular contraction. In particular this guarantees that no deadlocks are possible between these control situations.

```
REFRACTORY_TIME  = 50
SENSE_TIMEOUT    = 15

DELAY_AFTER_V = 50
DELAY_AFTER_A =  5

HEART_ALLOWED_STOP_TIME = 135

MODE_SWITCH_DELAY  = 66
```

Figure 9.19: Parameters That Yield Property *(i)*.

Version 3.1.57 of the UPPAAL tool[4] is able to perform the model checking of both properties successfully in 11.83 respectively 4.26 seconds. The verification of the typically more expensive property *(ii)* is faster, since here it is possible to apply a property preserving convex hull over-approximation that is not preservative with respect to property *(i)*. We use a Sun Enterprise 450 with UltraSPARC-II processors, 300 MHz, and made use of UPPAAL's rich set of optimization options.[5] In particular the active clock reduction gives drastic improvements in model checking time in this example.

[4]The first release that includes the possibility to model check response properties was available in April 2001.

[5]See also Chapter 5.

It is worthwhile to mention that validity of property *(i)* is strongly dependent on the parameter setting of the model. We use the constants from Figure 9.19. If the programmer is allowed to switch between modes very fast, it is possible that she prevents the pacemaker from doing its job. E.g., for `MODE_SWITCH_DELAY = 65` the property *(i)* does not hold any more. In practice it is often a problem to find parameter settings, that entail a safe or correct operation of the system. In related work, an extended version of UPPAAL is used to derive parameters yielding property satisfaction automatically, see [HRSV01].

9.5 Reflection: Flattening Hierarchical Timed Automata

Hierarchical structures are powerful formalisms; one indication for this is that there are many options on how to fill the details. This has been subject to intensive research [vdB94, Har97]. As we see it, the crucial choice in our semantics for HTAs is to treat cascades of entries and exits of superstates monolithically. This is somewhat clumsy, but allows for a conceptually simple correspondence between configurations of the hierarchical model and the flattened version.

Partially due to this decision, the reference implementation turned out to be surprisingly complicated. The source consists of more than 9000 lines of documented **Java** code.[6] The high-level description given in this Chapter is a way to increase trust in our procedure and to allow for future maintenance.

The global join construction is a side effect of treating exit steps monolithically. We point out that entries and exits do *not* behave fully symmetric here. This is not an introduced problem; exiting more than one superstate implicitly requires synchronization. Giving conditions under which parts of a system to be entered is simpler than specifying at what point in time they can be left or interrupted. To the best of our knowledge this hat not been addressed before in the literature and we believe there is room for further elaboration on this topic.

In the pacemaker case-study, the increase in size of the generated model seems acceptable. Mainly entries and exits complicate Since we use committed locations to encode this it probably does not contribute significantly to the model checking time. The medium-sized model is sufficiently complicated to render the properties we model check non-trivial. The parameters that yield the safety property, e.g., were found experimentally. As for the usability of the flattened model, a lay-outer is desirable. The processes of the pacemaker case study are layouted by hand.

An alternative approach for model checking HTAs is to implement a model checking engine that operates directly on the hierarchical model. The configuration vector is more complicated to encode, but the sets of clock evaluations is not different from other dense-time formalisms. The algorithmic challenge is the implementation of superstate exits; basically the same computations as used in the global joins have to be performed. We consider it interesting to compare the run-times of model checking HTA models directly with those obtained after a flattening step. This would

[6] http://www.brics.dk/%7Eomoeller/hta/vanilla-1/

give an impression on how much overhead is really introduced by the flattening. There are plans in the DoCS group at Uppsala to address this, and we refer to their web-pages[7] for further information.

[7] http://www.docs.uu.se/docs/index.eng.shtml

Epilogue

After all is said and done, a lot more will be said than done.

— Unknown

Tony Hoare started to do program verification as early as 1969. It was to supplement the intuition and understanding of the designer by solid mathematical proof. The development of analytic methods for digital creations has been a busy field, decidability results for real-time formalisms are just one example for this. Others are security protocols, formal hardware design, and proof carrying code.

So why do things go wrong?

We name three reasons. First, there is market pressure. Humans are willing to take risks to win, rather than they are willing to do so to avoid losses. As for products this means: better ship today and hope for the best. Second, correctness is not enough an issue. During the last decade the world has witnessed that badly written faulty software sells. *Apparently other criteria, like feature, are more important. Third, people developing in digital tend to be optimists. In the tradition of mathematics, everything is wrong by default—until you proof it to be right. For computers, this attitude is sometimes reversed: everything is fine, until you find a bug.*

As for doing things right, there cannot be a silver bullet. Digital design is intrinsically complex. No single method can be expected to be sufficient to deal with all of today's complications, even less with those to come. But correctness can be created. In small amounts.

It was our ambition to aid this quest for correctness with our work. Looking back, we regret to leave many threads unfinished. The solutions to bring to problems that deep and tasks that complex will take more than one man and more than one life to finish.

It will be the challenge for industry to hold up virtues of design and engineering in an age of market pressure and deadlines. Not only must it be worth doing, it must be understood as worth doing it right. It is our conviction that—above all other criteria of quality—correctness requires understanding.

It will be the challenge for academics to make the fruits of their research not only useful, but also used. The threshold to overcome is to build methodologies and tools that are guided to gold by other hands than those of their creators.

Bibliography

[ABB+01] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Möller, Paul Pettersson, Carsten Weise, and Wang Yi. UPPAAL - Now, Next, and Future. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Modelling and Verification of Parallel Processes*, number 2067 in Lecture Notes in Computer Science Tutorial, pages 100–125. Springer–Verlag, 2001. [18]

[ACD93] Rajeev Alur, Costas Courcoubetis, and David Dill. Model Checking in Dense Real-Time. *Information and Computation*, 104(1):2–34, 1993. A preliminary version appeared in the Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS 1990). [14, 22, 49, 51, 76, 77, 79, 80, 91, 125, 126]

[AD94] Rajeev Alur and David L. Dill. Automata for Modelling Real-Time Systems. *Theoretical Computer Science*, 126(2):183–236, April 1994. [14, 41]

[AdAG+01] Rajeev Alur, Luca de Alfaro, Radu Grosu, Thomas A. Henzinger, Minsu Kang, Rupak Majumdar, Freddy Y.C. Mang, Christoph Meyer-Kirsch, and Bow-Yaw Wang. MOCHA: A Model Checking Tool that Exploits Design Structure. *Proceedings of 23rd International Conference on Software Engineering*, 2001. See http://www.cis.upenn.edu/%7Emocha/ . [164]

[ADF+01] Tobias Amnell, Alexandre David, Elena Fersman, M. Oliver Möller, Paul Petterson, and Wang Yi. Tools for Real-Time UML: Formal Verification and Code Synthesis, June 2001. in *Implementation and Validation of Object-oriented Embedded Systems (SIVOES'2001)*, Budapest, Hungary. [18]

[AIKY95] Rajeev Alur, Alon Itai, Robert P. Kurshan, and Mihalis Yannakakis. Timing Verification by Successive Approximation. *Information and Computation*, 118(1):142–157, April 1995. [144]

[AJ96] Parosh Aziz Abdulla and Bengt Jonsson. Verifying Programs with Unreliable Channels. *Information and Computation*, 127(2):91–101, June 1996. [71]

[AJ01] Parosh Aziz Abdulla and Bengt Jonsson. Ensuring Completeness of Symbolic Verification Methods for Infinite-State Systems. *Theoretical Computer Science*, 256(1–2), 2001. [71]

[AK95] Rajeev Alur and Robert P. Kurshan. Timing Analysis in COSPAN. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Proc. of Workshop on Verification and Control of Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science (LNCS)*, pages 220–231. Springer–Verlag, October 1995. [76]

[AL92] Martin Abadi and Leslie Lamport. An Old-Fashioned Recipe for Real Time. In *Proc. of REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science (LNCS)*, pages 1–27. Springer–Verlag, 1992. [99]

[Alh98] Sinan Si Alhir. *UML in a Nutshell*. O'Reilly, Sebastapol, CA, 1998. [31]

[Alu91] Rajeev Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Stanford University, 1991. [44, 76]

[AW99] Rajeev Alur and Bow-Yaw Wang. "Next" Heuristic for On-the-fly Model Checking. In *Proc. of CONCUR '99: Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science (LNCS)*, pages 98–113. Springer–Verlag, 1999. [153, 164, 165]

[Bar96] Henk Barendregt. The Quest for Correctness. In *Images of SMC research*, pages 39–58. Stichting Mathematisch Centrum, P.O. Box 94079, 1090 GB Amsterdam, 1996. [1, 3, 10]

[BCM+90] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proc. of IEEE Symp. on Logic in Computer Science*, 1990. [11, 94]

[BDG88] José Luis Balcázar, Josep Díaz, and Joaquim Gabarró. *Structural Complexity I*. Springer–Verlag, New York, NY, 1988. [65]

[BDL+01a] Gerd Behrman, Alexandre David, Kim G. Larsen, M. Oliver Möller, Paul Petterson, and Wang Yi. UPPAAL - Present and Future. In P. Pettersson and S. Yovine, editors, *Workshop on Real-Time Tools*, August

2001. Proceedings appeared as technical report 2001-014 Uppsala University, Sweden. [18]

[BDL+01b] Gerd Behrman, Alexandre David, Kim G. Larsen, M. Oliver Möller, Paul Petterson, and Wang Yi. UPPAAL - Present and Future. In *Proc. of the 40th IEEE Conference on Decision and Control*, pages 2281–2286, Orlando, Florida, December 2001. IEEE Service Center. [18]

[BDM+98] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A Model Checking Tool for Real-Time Systems. In *Proc. of the 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science (LNCS)*, pages 546–550. Springer–Verlag, 1998. [76, 100]

[BE96] Grady Booch and Ed Eykholt. *The Best of Booch: Designing Strategies for Object Technology.* SIGS Books & Multimedia, 1996. [23]

[Bel58] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958. [82]

[BG92] Gerard Berry and Georges Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, November 1992. [38]

[BGL+00] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, Natarajan Shankar, Eli Singerman, and Ashish Tiwari. An Overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center. [12]

[BH95] Jonathan P. Bowen and Michael G. Hinchey. Seven More Myths of Formal Methods. *IEEE Software*, 12(4):34–41, July 1995. [9]

[BILT92] Beth H. Levy, Ivan V. Filippenko, Leo Marcus, and Telis Menas. Using the State Delta Verification System (SDVS) for Hardware Verification. In *Proc. of the International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 337–360, Nijmegen, June 1992. IFIP TC10/WG 10.2, North-Holland. [9]

[BJR96] Grady Booch, Ivar Jacobson, and Jim Rumbaugh. Unified Method for Object-Oriented Development v. 0.9, 1996. Rational Software Corp. [24]

[BK95] David A. Basin and Nils Klarlund. Hardware Verification Using Monadic Second-Order Logic. In P. Wolper, editor, *Proc. of the 7th Int. Conf. on Computer Aided Verification*, volume 939 of *Lecture Notes in*

Computer Science (LNCS), pages 31–41, Berlin;Heidelberg;New York, January 1995. Springer–Verlag. [7]

[BLO98] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In Alan J. Hu and Moshe Y. Vardi, editors, *Proc. of the 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science (LNCS)*, pages 319–331, Vancouver, Canada, June 1998. Springer–Verlag. [124]

[BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, New York, 1979. [9]

[BM88] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988. [9]

[BR95] Grady Booch and Jim Rumbaugh. Unified Method for Object-Oriented Development v. 0.8, 1995. Rational Software Corp. [24]

[BR01] Thomas Ball and Sriram K. Rajamani. The SLAM Toolkit. In G. Berry and A. Finkel, editors, *Proc. of the 13th Int. Conf. on Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science (LNCS)*, pages 260–264, 2001. [16]

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, Massachusetts, USA, 1st edition, 1999. [24, 30, 36, 38, 152]

[Bru95] Giorgio Bruno. *Model-based Software Engineering*. Chapman & Hall, London, 1995. [2, 14]

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean-Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986. [11, 83]

[Bry95] Randal E. Bryant. Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification. In *International Conference on Computer Aided Design*, pages 236–245. IEEE Computer Society Press, 1995. [11]

[CC77] Patrik Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Proc. of the 4*th *ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977. [10, 119, 120, 122]

[CC00] Patrick Cousot and Radhia Cousot. Temporal Abstract Interpretation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)*, pages 12–25, N.Y., January 19–21 2000. ACM Press. [11, 122]

[CC01] Patrick Cousot and Radhia Cousot. Static Analysis of Embedded Software: Problems and Perspectives, invited paper. In T.A. Henzinger and C.M. Kirsch, editors, *Proc.* First Int. Workshop on Embedded Software, EMSOFT 2001, volume 2211 of *Lecture Notes in Computer Science (LNCS)*, pages 97–113. Springer–Verlag, 2001. [15]

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Corina S. Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proc. of the 22nd International Conference on Software Engineering*, pages 439–448, June 2000. [15]

[CE82] Edmund M. Clarke and E. Allen Emerson. Synthesis of Synchronization Skeletons from Branching Time Temporal Logic. In *Proc. Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science (LNCS)*, pages 52–71. Springer–Verlag, 1982. [153]

[CES86] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite State Concurrent System Using Temporal Logic. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, 1986. [4]

[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, *Proc. of the 12th Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science (LNCS)*, pages 154–169. Springer–Verlag, 2000. [125]

[CGL93] Karlis Cerans, Jens Chr. Godskesen, and Kim G. Larsen. Time Modal Specification – Theory and Tools. In *Proc. of the 5th Int. Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science (LNCS)*, pages 253–267. Springer–Verlag, 1993. [76]

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999. [11]

[CHR91] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40(5):269–276, December 1991. [14]

[CK96] Edmund M. Clarke and Robert P. Kurshan. Computer-Aided Verification. *IEEE Spectrum*, 6(33):61–67, June 1996. [153]

[CL00] Franck Cassez and Kim G. Larsen. The Impressive Power of Stopwatches. In *Proc. of CONCUR 2000: Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science (LNCS)*, pages 138–152. Springer–Verlag, 2000. [178]

[CLR92] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, 6th edition, 1992. [82]

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A Tutorial Introduction to PVS. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida*, April 1995. Revised version available from http://-www.csl.sri.com/fm-papers.html. [7, 9]

[CPS93] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993. [153]

[CY91] Costas Courcoubetis and Mihalis Yannakakis. Minimum and Maximum Delay Problems in Real Time Systems. In *Proc. of 3rd Workshop on Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science (LNCS)*, pages 399–409. Springer–Verlag, 1991. Full version in *Formal Methods in System Design* (special issue for 3rd CAV), 1(4), pp. 385-415, 1992. [91]

[Dam96] Dennis René Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, July 1996. [11, 122, 123, 124, 129]

[DD01] Satyaki Das and David L. Dill. Successive Approximation of Abstract Transition Relations. In *Proc. of Logic in Computer Science (LICS2001)*, pages 51–60, 2001. [125]

[Dil89] David L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science (LNCS)*, pages 197–212, Berlin, June 1989. Springer–Verlag. [82]

[DM01] Alexandre David and M. Oliver Möller. From HUPPAAL to UPPAAL: A Translation from Hierarchical Timed Automata to Flat Timed Automata. Research Series RS-01-11, BRICS, Department of Computer Science, University of Aarhus, March 2001. [17, 18, 71]

[DMY01] Alexandre David, M. Oliver Möller, and Wang Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In M. R. Hansen, editor, *The 13th Nordic Workshop on Programming Theory (NWPT'01)*, appeared as technical report of the Technical University of Denmark, *IMM-TR-2001-12*, October 2001. [18]

[DMY02] Alexandre David, M. Oliver Möller, and Wang Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE'2002)*, volume 2306 of *Lecture Notes in Computer Science (LNCS)*, pages 218–232. Springer–Verlag, 2002. [17, 18]

[Dou99a] Bruce Powel Douglass. *Doing Hard Time: Developing Real-time Systems with UML, Objects, Frameworks, and Patterns*. Object Technology Series. Addison-Wesley, 1999. [31, 36, 149, 173]

[Dou99b] Bruce Powel Douglass. *Real-Time UML, Second Edition - Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1999. [31]

[DT98] Conrado Daws and Stavros Tripakis. Model Checking of Real-Time Reachability Properties Using Abstractions. In B. Steffen, editor, *Proc. of the 4*th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science (LNCS)*, pages 313–329. Springer–Verlag, 1998. [14, 92, 94, 101, 120]

[DWT95] David L. Dill and Howard Wong-Toi. Verification of Real-Time Systems by Successive Over and Under Approximation. In P. Wolper, editor, *Proc. of the 7th Conference on Computer-Aided Verification, CAV'95*, volume 939 of *Lecture Notes in Computer Science (LNCS)*, pages 409–422. Springer–Verlag, 1995. [145]

[DY96] Conrado Daws and Sergio Yovine. Reducing the Number of Clock Variables of Timed Automata. In *Proc. of the 17th IEEE Real-Time Systems Symposium*, pages 73–81. IEEE Computer Society Press, 1996. [92]

[Fla88] Phillippe Flajolet. Mathematical Methods in the Analysis of Algorithms and Data Structures. Lecture Notes for *A Graduate Course on Computation Theory*, Udine (Italy), Fall 1984. In E. Börger, editor, *Trends in Theoretical Computer Science*, pages 225–304. Computer Science Press, 1988. [154]

[Fla97] Philippe Flajolet. A Problem in Statistical Classification Theory, 1997. http://pauillac.inria.fr/algo/libraries/autocomb/schroeder-html/schroeder.html. [155]

[FORS01] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and Natarajan Shankar. ICS: Integrated Canonizer and Solver. In A. Finkel G. Berry,

H. Comon, editor, *CAV 01: Computer-Aided Verification*, volume 2101 of *Lecture Notes in Computer Science (LNCS)*, pages 246–249. Springer–Verlag, 2001. [141]

[GHJ01] Patrice Godefroid, Michael Huth, and Radha Jagadeesan. Abstraction-Based Model Checking Using Modal Transition Systems. In K. G. Larsen and M. Nielsen, editors, *Proc. of CONCUR 2001: Concurrency Theory*, Lecture Notes in Computer Science (LNCS), pages 426–440. Springer–Verlag, August 2001. [120, 124]

[GJ79] Michael R Garey and David S Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., San Francisco, 1979. [156]

[GJS76] Michael R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some Simplified NP-Complete Graph Problems. *Theoretical Computer Science*, 1(3):237–267, February 1976. [156]

[GMW79] Michael J. Gordon, Arthur J. Miller, and C. P. W. Wadsworth. *Edinburgh LCF*. Springer–Verlag, Berlin, 1 edition, 1979. [9]

[Gom00] Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Object Technology Series. Addison-Wesley, 2000. [31]

[GPS96] Patrice Godefroid, Doron Peled, and Mark Staskauskas. Using Partial Order Methods in the Formal Validation of Industrial Concurrent Programs. *IEEE, Transactions on Software Engineerings*, 22:496–507, 1996. [165]

[GRD91] Saul B. Gelfand, C. S. Ravishankar, and Edward J. Delp. An Iterative Growing and Pruning Algorithm for Classification Tree Design. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-13(2):163–174, February 1991. [171]

[GS97] Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification. 9th International Conference (CAV97)*, volume 1254 of *Lecture Notes in Computer Science (LNCS)*, pages 72–83. Springer–Verlag, 1997. [124]

[GW91] Patrice Godefroid and Pierre Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In *Proc. of 3rd Workshop on Computer Aided Verification*, Lecture Notes in Computer Science (LNCS). Springer–Verlag, 1991. [11]

[Hal90] Anthony Hall. Seven Myths of Formal Methods. *IEEE Software*, 7(5):11–19, September 1990. [9]

[Har84] David Harel. Statecharts: A Visual Approach to Complex Systems. Technical Report CS84-05, The Weizmann Institute of Science, February 1984. (revised December 1984). [32]

[Har87] David Harel. Statecharts: A Visual Formalism for Complex System. *Science of Computer Programming*, 8(3):231–274, 1987. [32, 33, 152]

[Har88] David Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988. [32]

[Har97] David Harel. Some Thoughts on Statecharts, 13 Years Later. In O. Grumberg, editor, *Proc. of the 9th Int. Conf. on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science (LNCS)*, pages 226–231. Springer–Verlag, 1997. [194]

[HG94] David Harel and Eran Gery. Executable Object Modeling with Statecharts. Technical Report CS94-20, Weizmann Institute of Science, Faculty of Mathematics and Computer Science, January 1, 1994. [38]

[HG97] David Harel and Eran Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7), 1997. [38]

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A Model Checker for Hybrid Systems. In O. Grumberg, editor, *Proc. of the 9th Int. Conf. on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science (LNCS)*, pages 460–463. Springer–Verlag, 1997. [76]

[Hig01] Uwe Higgen. Automatic Test Case Generation out of UML Models, 2001. master thesis, Universität Oldenburg, 2001, see http://www.uwe-higgen.de/. [216]

[Hil99] Martin Hiller. Objektorientierte Systemanalyse mit Rhapsody, 1999. Diplomarbeit (in German), University of Ulm, Germany, corresponds to a master thesis. [38]

[HJJ+97] Jesper G. Henriksen, Jakob Jensen, Michael Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. *Mona: Monadic Second-Order Logic in Practice*. BRICS, Centre of the Danish National Research Foundation for Basic Research in Computer Science, Department of Computer Science, University of Aarhus, 1.1st edition, 1997. [7]

[HN96] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. on Software Engineering and Methodology*, 5(4):293–333, October 1996. [33]

[HNSY94] Thomas. A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2):193–244, 1994. [94, 112]

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice–Hall, 1985. [9]

[Hol91] Gerard J. Holzmann. *The Design and Validation of Computer Protocols*. Prentice Hall, 1991. [94]

[Hol97] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. [70, 94, 153]

[Hol98] Gerard J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13:289–307, 1998. [94]

[HP85] David Harel and Amir Pnueli. On the Development of Reactive Systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO ASI*, pages 477–498, New York, 1985. Springer–Verlag. [2, 10, 14, 32]

[HPSS87] David Harel, Amir Pnueli, J. P. Schmidt, and Rivi Sherman. On the Formal Semantic of Statecharts. In *Proc. of IEEE Symposium on Logic in Computer Science (LICS 1987)*, pages 54–64, 1987. [32, 33]

[HRdR92] Jozef J. M. Hooman, S. Ramesh, and Willem-Paul de Roever. A Compositional Axiomatization of Statecharts. *Theoretical Computer Science*, 101(2):289–335, July 1992. [33]

[HRP94] Nicolas Halbwachs, Pascal Raymond, and Yann-Erick Proy. Verification of Linear Hybrid Systems by Means of Convex Approximations. In *Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science (LNCS)*, pages 223–237, 1994. [76]

[HRSV01] Thomas S. Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. Linear Parametric Model Checking of Timed Automata. Research Series RS-01-5, BRICS, Department of Computer Science, University of Aarhus, January 2001. 44 pp. [194]

[HSLL97] 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 *Proc. of the 18th IEEE Real-Time Systems Symposium*, pages 2–13. IEEE Computer Society Press, December 1997. [41]

[HU80] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, N. Reading, MA, 1980. [65]

[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 12*th *Euromicro Conference on Real-Time Systems*, pages 147–155. IEEE Computer Society Press, June 2000. [114, 118]

[IL00] I-Logix. Rhapsody User Guide, 2000. Release 3.0, available as part of the free trial distribution http://www.ilogix.com/quick%5Flinks/-downloads.cfm. [68]

[Jai94] Raj Jain. *FDDI Handbook: High-Speed Networking with Fiber and Other Media*. Addison-Wesley, Reading, MA, April 1994. [101]

[JCJÖ93] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, revised printing edition, 1993. [23]

[JK90] Ryszard Janicki and Maciej Koutny. Using Optimal Simulations to Reduce Reachability Graphs. In *Proc. of 2nd International Conference on Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science (LNCS)*, pages 166–175. Springer–Verlag, 1990. [11]

[Joh01] Steven D. Johnson. View from the Fringe of the Fringe. In *Proc. in Theorem Proving and Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science (LNCS)*, page 4. Springer–Verlag, 2001. [15]

[Kat99] Joost-Pieter Katoen. Concepts, Algorithms, and Tools for Model Checking, 1999. Lecture Notes of the Course "Mechanized Validation of Parallel Systems", course number 10359, Friedrich-Alexander Universität Erlangen-Nürnberg. [11]

[Kel95] Peter Kelb. *Abstraktionstechniken für automatische Verifikationsmethoden*. PhD thesis, Universität Oldenburg, Carl v. Ossietzky Universität, Dekanat Informatik, Postfach, 26111 Oldenburg, Germany, December 1995. [11, 122]

[Kic96] Alexander Kick. *Generation of Counterexamples and Witnesses for the Mu-Calculus*. PhD thesis, University of Karlsruhe, Germany, 1996. [141]

[KK01] Roope Kaivola and Katherine Kohatsu. Proof Engineering in the Large: Formal Verification of Pentium4 Floating-Point Divider. In T. Margaria and T. Melham, editors, *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001, Livingston, Scotland, UK*, volume 2144 of *Lecture Notes in Computer Science (LNCS)*, pages 196–211, New York, NY, USA, September 2001. Springer–Verlag. [15]

[KL96] Inhye Kang and Insup Lee. An Efficient State Space Generation for Analysis of Real-Time Systems. In *International Symposium on Software Testing and Analysis*, January 1996. TREAT is available at http://www.cis.upenn.edu/%7Elee/inhye/treat.html. [76]

[KM01] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1, Department of Computer Science, University of Aarhus, January 2001. Available from http://www.brics.dk/mona/. Revision of BRICS Notes Series NS-98-3. [7]

[KMP96] Yonit Kesten, Zohar Manna, and Amir Pnueli. Verifying Clocked Transition Systems. In *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science (LNCS)*, pages 13–40. Springer–Verlag, 1996. [33]

[Kob99] Cris Kobryn. UML 2001: A Standardization Odyssey. *Communications of the ACM*, 42(10):29–37, October 1999. [25, 27]

[Kob01a] Cris Kobryn. UML 2.0 Roadmap, 2001. slide show corresponding to article [Kob01b], online available at http://www.celigent.com/outgoing/presentations/Kobryn%5FUML2%5FRoadmap%5FR4.zip. [24]

[Kob01b] Cris Kobryn. UML 2.0 Roadmap: Fast Track or Detours, April 2001. magazine article in "Software Development", online available at http://www.sdmagazine.com/articles/2001/0104/. [24, 25, 27, 210]

[Koz83] Dexter C. Kozen. Results on the Propositional mu-Calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983. [123, 129]

[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*, volume 571 of *Lecture Notes in Computer Science (LNCS)*, pages 591–619. Springer–Verlag, 1992. [33]

[Lam87] Leslie Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transations on Computer Systems*, 5(1):1–11, February 1987. [99]

[LBBO01] Yassine Lachnech, Saddek Bensalem, Sergey Berezin, and Sam Owre. Incremental verification by abstraction. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems: 7th International Conference, TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science (LNCS)*, pages 98–112, Genova, Italy, April 2001. Springer–Verlag. [125]

[LL73] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973. [13]

[LL98] Francois Laroussinie and Kim G. Larsen. CMC: A Tool for Compositional Model Checking of Real-Time Systems. In *Proc. of Joint International Conference on Formal Description Techniques and Protocol Specification, Testing, and Verification*, 1998. [76]

[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. [82, 92, 93]

[LP97] Henrik Lönn and Paul Pettersson. Formal Verification of a TDMA Protocol Start-Up Mechanism. In *Proc. of IEEE Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 235–242, 1997. [41]

[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. [41, 42, 76]

[LPY98] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear Controller. In *Proc. of the 4th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems.*, volume 1384 of *Lecture Notes in Computer Science (LNCS)*, pages 281–297. Springer–Verlag, 1998. [41]

[LWYP99] Kim G. Larsen, Carsten Weise, Wang Yi, and Justin Pearson. Clock Difference Diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999. [83]

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, San Mateo, CA, 1996. [167]

[MA00] M. Oliver Möller and Rajeev Alur. Heuristics for Hierarchical Partitioning with Application to Model Checking. Research Series RS-00-21, BRICS, Department of Computer Science, University of Aarhus, August 2000. 30 pp, available online at http://www.brics.dk/RS/00/21/. [18, 164]

[MA01] M. Oliver Möller and Rajeev Alur. Heuristics for Hierarchical Partitioning with Application to Model Checking. In T. Margaria and T. Melham, editors, *Correct Hardware Design and Verification Methods, 11th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2001, Livingston, Scotland, UK*, volume 2144 of *Lecture Notes in Computer Science (LNCS)*, pages 71–85, New York, NY, USA, September 2001. Springer–Verlag. [18]

[McC90] William W. McCune. *OTTER Users' Guide, Version 2.0.* Argonne National Laboratory, 1990. [9]

[McM93] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993. [11]

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice–Hall, 1989. [9]

[MLAH99] Jesper Møller, Jakob Lichtenberg, Henrik R. Andersen, and Henrik Hulgaard. Difference Decision Diagrams. In *Computer Science Logic*, The IT University of Copenhagen, Denmark, September 1999. [83]

[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–Verlag, July 1997. [33]

[Möl02] M. Oliver Möller. Parking Can Get You There Faster - Model Augmentation to Speed up Real-Time Model-Checking. in Theory and Practice of Timed Systems (TPTS'2002), 2002. [17]

[MP91] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification.* Springer–Verlag, 1991. [33]

[MRS01] M. Oliver Möller, Harald Rueß, and Maria Sorea. Predicate Abstraction for Dense Real-Time Systems. Research Series RS-01-44, BRICS, Department of Computer Science, University of Aarhus, November 2001. [18]

[MRS02] M. Oliver Möller, Harald Rueß, and Maria Sorea. Predicate Abstraction for Dense Real-Time Systems. in Theory and Practice of Timed Systems (TPTS'2002), 2002. [18]

[MT93] Michael J.C. Gordon and Thomas F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993. [9]

[Nie00] Brian Nielsen. *Specification and Test of Real-Time Systems.* PhD thesis, Department of Computer Science, Aalborg University, Denmark, April 2000. [6]

[NK00] Kedar S. Namjoshi and Robert P. Kurshan. Syntactic Program Transformations for Automatic Abstraction. In E.A. Emerson and A.P. Sistla, editors, *Proc. of the 12th Conference on Computer-Aided Verification, CAV'2000*, volume 1855 of *Lecture Notes in Computer Science (LNCS)*, pages 435–449, Chicago, IL, 2000. Springer–Verlag. [145]

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer–Verlag, 1999. [10, 51]

[NS94] Xavier Nicollin and Joseph Sifakis. The Algebra of Timed Processes ATP: Theory and Application. *Information and Computation*, 114(1):131–178, 1994. [14]

[OCL97] Object Constraint Language Specification, 1997. part of the UML standardization, see http://www.klasse.nl/ocl/ocl-status-text.html. [30]

[OD98] Ernst-Rüdiger Olderog and Henning Dierks. Decomposing Real-Time Specifications. In *COMPOS: International Symposium on Compositionality: The Significant Difference*, volume 1536 of *Lecture Notes in Computer Science (LNCS)*, pages 465–489. Springer–Verlag, 1998. [14]

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts - Menlo Park, California - New York, 1994. [65]

[Par97a] UML Partners. Unified Modeling Language v. 1.0, January 1997. OMG document ad/97-01-14. [24]

[Par97b] UML Partners. Unified Modeling Language v. 1.1, August 1997. OMG document ad/97-08-11. [24]

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science (LNCS)*. Springer–Verlag, New York, NY, USA, 1994. [9]

[Pet99] Paul Pettersson. *Modelling and Analysis of Real-Time Systems Using Timed Automata: Theory and Practice*. PhD thesis, Department of Computer Systems, Uppsala University, February 1999. [48, 82, 93]

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, October 31–November 2 1977. IEEE Computer Society Press. [4]

[Pnu86] Amir Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: a Survey of Current Trends. In J.W. de Bakker, W.P de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency: Overviews and Tutorials*, volume 224 of *Lecture Notes in Computer Science (LNCS)*, pages 510–584. Springer–Verlag, 1986. [4, 32]

[Pnu97] Amir Pnueli. Verification Engineering: A Future Profession, August 1997. A. M. Turing Award Lecture, Sixteenth Annual ACM Symposium on Principles of Distributed Computing (PODC 1990),

San Diego, slides available at http://www.wisdom.weizmann.ac.il/-%7Eamir/invited-talks.html. [6]

[PS01] Amir Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. In *Proc. Symp. on Theoret. Aspects of Comput. Soft.*, volume 525 of *Lecture Notes in Computer Science (LNCS)*, pages 244–464, Berlin, 1001. Springer–Verlag. [37]

[PU97] Carsta Petersohn and Luis Urbina. A Timed Semantics for the STATEMATE Implementation of Statecharts. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science (LNCS)*, pages 553–572. Springer–Verlag, September 1997. [33]

[RBP+92] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1992. [23]

[Rha02] Rhapsody, 2002. a commercial UML modeling tool by I-Logix, see http://www.ilogix.com/products/rhapsody/rhap%5Finc.cfm. [32]

[RR88] George M. Reed and A. W. Roscoe. A Timed Model for Communicating Sequential Processes. *Theoretical Computer Science*, 58(1-3):249–261, June 1988. [14]

[RTF98] Revision Task Force. OMG Unified Modeling Language Specification, v. 1.2, December 1998. document ad/98-12-02, Object Management Group. [25]

[RTF99] UML Revision Task Force. Unified Modeling Language Specification v. 1.3, June 1999. document ad/99-06-08, Object Management Group. [23, 25]

[RTF01] UML Revision Task Force. Requests for Proposals: Unified Modeling Language Specification v. 2.0, 2000/2001. documents ad/00-09-01, ad/00-09-03, and Group, issued between March and September. [25]

[Rud92] Piotr Rudnicki. An Overview of the Mizar Project. Notes to a talk at the workshop on *Types for Proofs and Programs*, available through anonymous ftp: `pub/cs-reports/baastad.92/proc.ps.Z` on ftp://-ftp.cs.chalmers.se, June 1992. [9]

[Rus00] John Rushby. Disappearing Formal Methods. In *High-Assurance Systems Engineering Symposium*, pages 95–96, Albuquerque, NM, November 2000. Association for Computing Machinery. [8]

[RvHO91] John Rushby, Friedrich von Henke, and Sam Owre. An Introduction to Formal Specification and Verification Using EHDM. Technical Report CSL-91-2, SRI International, Menlo Park, California, February 1991. online available at http://www.csl.sri.com/reports/html/csl-91-2.html. [9]

[Sch70] Ernst Schröder. Vier combinatorische Probleme. *Zentralblatt. f. Math. Phys.*, 15:361–376, 1870. [154]

[Sha93] Natarajan Shankar. Verification of Real-Time Systems Using PVS. In Costas Courcoubetis, editor, *Proc. of the 5th Int. Conf. on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science (LNCS)*, pages 280–291, Elounda, Greece, June/July 1993. Springer–Verlag. [14]

[She95] Naveed Sherwani. *Algorithms for VLSI Physical Design Automation - 2nd Edition*. Kluwer Academic Publishers, Norwell, USA, 1995. [171]

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing company, Boston, MA, 1996. [65]

[SP00] Perdita Stevens and Rob Pooley. *Using UML Software Engineering with Objects and Components*. Object Techology Series. Addison-Wesley, updated edition, 2000. [31]

[SPE99] Request for Proposal: Software Process Engineering (SPE) Management, 1999. documents ad/99-11-04, Object Management Group, issued between March and September. [28]

[SPT01] Response to the OMG RFP for Schedulability, Performance, and Time, 2001. Revised Submission, 18 June 2001, OMG document ad/2001-06-14 available from http://www.omg.org. [32]

[SS99] Hassen Saïdi and Natarajan Shankar. Abstract and Model Check While you Prove. In *Proc. of the 11th Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science (LNCS)*, pages 443–454, 1999. [11, 124, 125]

[ST01] Daniel A. Spielman and Shang-Hua Teng. Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time. In *Proceedings of the 33rd Annual ACM Symposium on the Theory of Computing*, pages 296–305, 2001. [7]

[Ste98] Perdita Stevens. A Verification Tool Developer's Vade Mecum. *Int. Journal on Software Tools for Technology Transfer*, 2:89–94, 1998. [9]

[SZJ94] David Scholefield, Hussein Zedan, and He Jifeng. A Specification Oriented Semantics for the Refinement of Real-Time Systems. *Theoretical Computer Science*, 131(1):219–241, 1994. [14]

[TAKB96] Serdar Tasiran, Rajeev Alur, Robert P. Kurshan, and Robert K. Brayton. Verifying Abstractions of Timed Systems. In *Proc. of CONCUR '96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science (LNCS)*, pages 546–562. Springer–Verlag, 1996. [94]

[TC96] Stavros Tripakis and Costas Courcoubetis. Extending Promela and Spin for Real-Time. In *Proc. of the 2*nd *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, pages 329–348. Springer–Verlag, March 1996. [76]

[TY01] Stavors Tripakis and Sergio Yovine. Analysis of Timed Systems Using Time-abstracting Bisimulations. *Formal Methods in System Design*, 18(1):25–68, January 2001. [129, 145]

[UML01] Unified Modeling Language v. 1.4, 2001. online available from the Object Management Group (OMG) at http://www.omg.org. [25, 27, 32, 36]

[Uri98] Tomás E. Uribe. *Abstraction-Based Deductive-Algorithmic Verification of Reactive Systems*. PhD thesis, Computer Science Department, Stanford University, December 1998. Technical report STAN-CS-TR-99-1618. [144]

[vdB94] Michael von der Beeck. A Comparison of Statechart Variants. In H. Langmaack, W. de Roever, and J. Vytopil, editors, *Formal Techniques in RealTime and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science (LNCS)*, pages 128–148. Springer–Verlag, 1994. [32, 33, 36, 70, 194]

[Vot02] Angelika Votintseva. Specification-Based Test Generation for UML. to appear: Technical report, Universität Oldenburg (Abteilung Technische Informatik) see also Chapter 4 in [Hig01], 2002. [70, 71]

[Wan00] Farn Wang. Efficient Data-Structure for Fully Symbolic Verification of Real-Time Systems. In *Proc. of the 6th Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science (LNCS)*, pages 157–171. Springer–Verlag, 2000. [83]

[Wan01] Farn Wang. Clock Restiction Diagram: Yet Another Data-Structure for Fully Symbolic Verification of Timed Automata. Technical Report TR-IIS-01-002, Institute of Information Science, Academia Sinica, Tapei, Taiwan, 2001. [83]

[WH98] Farn Wang and Pao-Ann Hsiung. Automatic Verification on the Large. In *Proc. of the 3rd IEEE High-Assurance Systems Engineering Symposium*, November 1998. [76]

[Wol98] Pierre Wolper. Verification: Dreams and Reality, 1998. Inaugural lecture of the course *"The algorithmic verification of reactive systems"*, online available at http://www.montefiore.ulg.ac.be/%7Epw/-cours/francqui.html. [6]

[WOLB92] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications.* McGraw-Hill, New York, second edition, 1992. [9]

[WT94] Howard Wong-Toi. *Symbolic Approximations for Verifying Real-Time Systems.* PhD thesis, Stanford University, November 1994. [14, 93, 94, 120]

[Yi90] Wang Yi. Real-Time Behaviour of Asynchronous Agents. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90: Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science (LNCS)*, pages 502–520. Springer–Verlag, 1990. [14]

Index

$\eta^*(.)$, 56
$Inv(.,.)$, 61
$Inv_\nu(.)$, 61
$.\forall U.$, 130
$.\exists U.$, 129
$2^{\cdot}$: power set of a set, 55, 81, 122, 126
$\mathcal{T}_t(.,.,.,.)$, 60
μ-calculus
    semantics of, 123
$\eta^{-1}(.)$, 56
$\tilde{\eta}(.)$, 56
$\eta^{-*}(.)$, 56
$Var^+(.)$, 56
`-->` property, 50
`-A`: convex hull approximation, **93***
`-C`: compact DBM representation *OFF*, **92***
`-H size`: change size of hash table in passed list, **95***
`-Q`: display warnings as queries, **95***
`-S 1|2|3`: space usage reduction, **93***
`-T`: optimize time consumptions when several properties are examined, **95***
`-U`: unpack reduced constraint system before inclusion check, **96***
`-W`: disable deadlock checker, **95***
`-Z`: bitstate hashing, **94***
`-a`: active clock reduction, **92***
`-d`: depth-first search, **95***
`-q`: do not display copyright message, **96***
`-s`: run silently without progress indicator, **96***
`-t`: print diagnostic trace to stdout, **96***
`-y`: display traces symbolically, **96***
2-merge, 168
80-20 rule, 7

a posteriori, 1*, 10, **119***
a priori, 13, 76, 102
`A<>` property, 50, 112
`A[]` property, 50, 112
abstract_and_refine, **142***
abstract interpretation, 10, 119
abstraction, 122
abstraction function, 135
abstraction predicate, 134
accepting computation
    of a Turing machine, **65***
action, 35
action rule, 61
delay transition rule, 62
action step, 44
    simple, 46
    synchronized, 46
Active, **58***
active clock reduction (`-a`), **92***
active object, 38
airbag, 21
AIT-WOODDES project, 71
analog computers, 12
analysis, 1
AND predicate, 56
animation, 38
approximation of universal path properties, **113***
assignment, 35, 43
assumption
    non-convergence of time, 127
asynchronous, 69
asynchronous event, 36
asynchronous parity computer, **164***
atrial chamber, 186
AT&T, 16
attractor, 157
augmentation point, **109***
augmented model, **109***
augmented path semantics, **113***
automated theorem proving, 9

bad property, **168***
Bandera project, 15

BASIC predicate, 56
basis, 125
    set of predicates, **139**⋆
BDD: Binary Decision Diagram, 11, 83
Binary Decision Diagram (BDD), 11, 83
bisimulation, 123
bitstate hashing (`-Z`), **94**⋆
Booch, 24
Boolean abstraction, 119
Boyer-Moore theorem prover, 9
Bricks Sorter Model, 114–117
bricks sorter model, 115

CADE: Correct Hardware Design and Verification Methods, 6
canonical, 11
canonical representation of zones, 82
CASE tool, 37
CASE: Computer-Aided Software Engineering, 37
causality, 36
CAV: Computer Aided Verification, 6, 77
CDD: Clock Difference Diagram, 83
chain reaction, **36**⋆
change size of hash table in passed list (`-H size`), **95**⋆
CHARME: Correct Hardware Design and Verification Methods, 6, 18
Chinese boxes, 10
choice point, 35
chop operation, 14
Church-Turing thesis, 65
clock
    (formal), **42**⋆
    constraints, 125
    evaluation, **45**⋆, 126
    region, 127
    reset, 43
    set, 126
Clock Difference Diagram (CDD), 83
Clock Restriction Diagram (CRD), 83
clocked transition systems, 33
clocks
    set of, 126
code generation, 37
code generation process, 69
COMET, 31
committed location, 42
Common Object Request Broker Architecture (CORBA), 28
compact DBM representation (*OFF* with `-C`), **92**⋆
completeness
    of flattening, **184**⋆
    theorem for interactive refinement, 141
composition
    parallel, 34
computation
    of a Turing machine, **65**⋆
computational complexity, 7
Computer Aided Verification (CAV), 6, 77
Computer-Aided Software Engineering (CASE), 37
concretization, 122
concretization function, 135
configuration, 44
    of a event system, **64**⋆
    of a HTA, **58**⋆
    of a Turing machine, **65**⋆
    of an UPPAAL model, **45**⋆
configuration of a UPPAAL model, **45**⋆
configuration vector transformation $\mathcal{T}_t$, 60
connector
    history, 35
constraint graphs, 82
continuous part, 46
control situation, 46
convex hull approximation (`-A`), **93**⋆
CORBA: Common Object Request Broker Architecture, 28
Correct Hardware Design and Verification Methods (CADE), 6
Correct Hardware Design and Verification Methods (CHARME), 6, 18
cost of a hierarchical partitioning, 155
cover number (of a candidate), **161**⋆
coverage, 94
covers a set of well-formed sequence, **79**⋆
CRD: Clock Restriction Diagram, 83

DBM: Difference Bounded Matrix, 82
DDD: Difference Decision Diagram, 83
deadline, 13
deadlock, 62
deep history, 35
default
    entry, 57
    exit, 35, 56
default history locations, 57

delay loop example, 109
  continued, 114
delay step, 44, 47
  restricted, 128
dense real-time, 76
depth, 155
depth (of a candidate), **162**⋆
depth cost, 155
depth-first search (`-d`), **95**⋆
development process
  ROPES, 31
Difference Bounded Matrix (DBM), 82
Difference Decision Diagram (DDD), 83
disable deadlock checker (`-W`), **95**⋆
discrete part, 46
discrete real-time, 76
dispatched, 69
display traces symbolically (`-y`), **96**⋆
display warnings as queries (`-Q`), **95**⋆
do not display copyright message (`-q`), **96**⋆
document type definition (dtd), 28
dtd: document type definition, 28
duplications of channels, 180
duration calculus, 14

`E<>` property, 50
`E[]` property, 50
Edge-Guided Tree-Indexing, **156**⋆
Ehdm, 9
80-20 rule, 7
embedded system, 12
embedded systems, 21
enabled, 43
entry
  default, 57
ENTRY predicate, 56
environment, 37
  of an Uppaal model, 45
equivalence of semantics
  logical, 132
Esterel Technologies, 16
ETAPS: European joint conference on Theory and Practice of Software, ETAPS: European15
European joint conference on Theory and Practice of Software (ETAPS), European joint15
event, **36**⋆
  time-out, 38
  trigger, 35
event queue, **64**⋆
event queues, 69
event system, **63**⋆
event system configuration, **64**⋆
events
  in Rhapsody, 68
executable object, 38
existential until, 129
exit
  default, 35, 56
EXIT predicate, 56
expandGlobalJoins, **179**⋆
eXtensible Markup Language (XML), 27

FASE: Fundamental Approaches to Software Engineering, 17, 18
FDDI Token Ring Protocol, **100**⋆
FDDI: Fiber Distributed Data Interface, 100
Fiber Distributed Data Interface (FDDI), 100
final state, 34
firm deadline, 13
Fischer's protocol for mutual exclusion, 43
FMCAD: Formal Methods in Computer Aided Design, 5, 91
fork, 34
fork, 59
formal
  methods, 5
  semantics, 30
  verification, 9
Formal Methods in Computer Aided Design (FMCAD), 5, 91
formal verification, 2
Fundamental Approaches to Software Engineering (FASE), 17, 18
fusion closure, 48

Galois connection, 122
General InterORB Protocol (GIOP), 28
GIOP: General InterORB Protocol, 28
global consistency, 36
global join
  example, 178
global reduction, 93
greedy, 160
guard, 35, 43

halting problem, **65**⋆
HasHistory(.), 59
heavyweight model extension, 27
HENTRY predicate, 56
heuristic
"Next", 153, **164**⋆
hierarchical partition, 152
hierarchical partitioning, 151, **154**⋆
cost of a, 155
hierarchical timed automata, 53
semantics, **58**⋆
syntax, **54**⋆
hierarchical timed automaton (HTA), 55
Higher Order Logic (HOL), 9
history
connector, 35
deep, 35
shallow, 35
HISTORY predicate, 56
HOL: Higher Order Logic, 9
HTA configuration, 58
HTA: hierarchical timed automaton, 55
HTML: hypertext markup language, 28
hyperedge, 154
hypergraph, 154
hypertext markup language, **28**⋆
hypertext markup language (HTML), 28

I Can Solve (ICS), 141
ICS: I Can Solve, 141
IIOP: Internet Inter-ORB Protocol, 28
implicit events, 36
incremental, 124
independent, 183
inevitability property, 50
initial location, 126
initial states, 122
instantiateTemplates, **176**⋆
instantiation tree, 174
interesting pair, 163
interface definition language, 28
Internet Inter-ORB Protocol (IIOP), 28
invariant, 42, 126
Isabelle, 9

join, 34
join, 59

Kleene-star, 64
Kripke structure, 122

labeling function, 122
language
modeling, 23
language accepted by a Turing machine, **65**⋆
largest constant, 80, 125
lattice structure, 122
LCF: Logic of Computable Functions, 9
leader election in a ring, **167**⋆
Leaves, 61
levels, UML, 25
lightweight model extension, 27
linear programming, 7
Linear Temporal Logic (LTL), 94
literal, 124
local consistency, 37
local property, 49
validity of, 49
local reduction, **92**⋆
locations
set of, 126
vector of, 45
logic, 4
Logic of Computable Functions (LCF), 9
logical equivalence of semantics, 132
LTL: Linear Temporal Logic, 94

magic square, 10
mania
meta-modeling, 27
matching configuration, 182
MAY, 120
meta-model, 25
meta-modeling
loose, 26
strict, 26
meta-modeling mania, 27
Meta-Object Facility (MOF), 28
methods
formal, 5
Minimum Cut into Bounded Sets, 156
Minimum Cut Into Equal-Sized Subsets, 156
Mizar, 9
model
timed automata, 22
model augmentation, **109**⋆
model checking, 11, 153
state-based, 122

trace-based, 122
model extension
heavyweight, 27
lightweight, 27
modeling language, 23
modular, 152
MOF: Meta-Object Facility, 28
monomial, 124
μ-calculus
propositional, 123
μ-equivalence, 130
MUST, 120

"Next" heuristic, 153, **164***
next-free μ-calculus, 129
semantics of the, 130
non-convergence of time assumption, 127
Nordic Workshop on Programming Theory (NWPT), 18
NWPT: Nordic Workshop on Programming Theory, 18

Object Constraint Language (OCL), 30
Object Management Architecture (OMA), 28
Object Management Group (OMG), 24
Object Modeling Technique (OMT), 24
Object Request Broker (ORB), 28
object-oriented software engineering (OOSE), 24
OCL: Object Constraint Language, 30
OMA: Object Management Architecture, 28
OMG IDL: OMG interface definition language, 28
OMG interface definition language, 28
OMG interface definition language (OMG IDL), 28
OMG: Object Management Group, 24
OMT: Object Modeling Technique, 24
OOSE: object-oriented software engineering, 24
Operating System (OS), 15
operations research, 7
opinion poll protocol, **168***
optimize time consumptions when several properties are examined (`-T`), **95***
ORB: Object Request Broker, 28
OS: Operating System, 15
OTTER, 9
outgoing transitions, 42
over-approximation, 135

parallel composition, 34
partial order reductions, 11
partition_incrementally, **161***
partitioning
hierarchical, 151
postprocessChannels, **180***
potentially always property, 50
predicate abstracted semantics, 136
predicate abstraction, 119, **124***
pref, 169
pref+, 169
preservation requirement
strong, 123
weak, 124
print diagnostic trace to stdout (`-t`), **96***
priorities, 69
profile
UML, 27
proper configuration of a HTA, 59
proper step, 59
proper transition part, 59
propositional μ-calculus, **123***
propositional symbols, 126
protocol
opinion poll, **168***
Prover Technology, 16
pseudo state, 34
pseudo-transitions, 57
PVS, 9

race condition, 37
Rapid Object-Oriented Process for Embedded Systems (ROPES), 31
rating function, **161***
RATIONAL ROSE, 32
reachability property, 50
reactive systems, 2*, 32
real-time
dense, 76
discrete, 76
real-time system, 12, 21
RealTime Operating System (RTOS), 69
RED: Region Encoding Diagram, 83
referential transparency, 10
refinement, 182
refinement checking, 10
refinement of abstraction, 141

refining, 119
region, **80**⋆
  clock, 127
Region Encoding Diagram (RED), 83
region equivalence, 131
rejecting computation
  of a Turing machine, **65**⋆
renaming, 175
request for proposal (RFP), 25
response property, 50
restricted delay step, 128
restricted path, 128
Revision Task Force (RTF), 24
RFP: request for proposal, 25
Rhapsody, 68
Rhapsody, 32
ROPES development process, 31
ROPES: Rapid Object-Oriented Process for Embedded Systems, 31
Round-Robin scheduler, 116
RTF: Revision Task Force, 24
RTOS: RealTime Operating System, 69
run of an event system, **64**⋆
run silently without progress indicator (`-s`), **96**⋆
run-to-completion step, 35, **36**⋆

safety property, 50
SAL: Symbolic Analysis Laboratory, 12
scheduling, 12
scheduling policy, 13
SDL: Specification and Description Language, 16
SDVS, 9
self-application, 65
self-triggering, 36
semantics
  augmented path, **113**⋆
  for hierarchical timed automata, **58**⋆
  formal, 30
  of HTA, **63**⋆
  of universal path properties, **112**⋆
  predicate abstracted, 136
  UML, 30
semantics of μ-calculus, 123
semantics of a timed system, 129
μ-calculus
  semantics of, 123
semantics of the next-free μ-calculus, 130
semi-automated theorem proving, 9
sentence, 129
set of clocks, 126
set of locations, 126
set of states, 122
SGML: Standard Generalized Markup Language, 28
shadow-vertex simplex algorithm, 7
shallow history, 35
shortest-path closure, 82
simple action step, 46
simplex algorithm, 7
simulation, 124
size (of a candidate), **162**⋆
SLAM project, 16
smoothed analysis, 7
soft deadline, 13
Software Process Engineering (SPE), 28
soundness
  of flattening, **184**⋆
  theorem for interactive refinement, 141
source, 42
source of a transition, 56
space usage reduction (`-S 1|2|3`), **93**⋆
SPE: Software Process Engineering, 28
Specification and Description Language (SDL), 16
SPIN, 94
square matrices of bounds, 82
stable configuration, 181
Standard Generalized Markup Language (SGML), 28
state, 4, 33
  basic, 33
  pseudo, 34
  sub-, 55
  super-, 33
state explosion problem, 11
state-based model checking, 122
Statemate, 32
step
  delay, 47
  proper, 59
  run-to-completion, 35
  simple action, 46
  synchronized action, 46
step encoding, 183
step relation of an event system, **64**⋆
step wise refinement, 10

strong preservation requirement, 123
strongest invariant, 12
stub, 34
sub-machines, 34
substate, 55
superstate, 33
Symbolic Analysis Laboratory (SAL), 12
symbolic representation, 76
symbolic state graph, **78***
symbolic states, 84
symbolic techniques, 11
symoblic reachability, 83
symoblic response, 87
symbolic_reach, **83***
symbolic_response, **87***
sync rule, 62
synchronized action step, 46
synchronous event, 36
synchrony hypothesis, 38
syntactic sugar, 149
syntax
    of hierarchical timed automata, **54***

target, 42
target language, 37
target of a transition, 56
TCTL: timed computation tree logic, 49
template mechanism, 175
temporal logic, 4
temporal logics, 2
temporal property
    validity of, **50***
terminal state, 34
termination
    theorem for interactive refinement, 141
Theorem Proving in Higher Order Logics (TPHOLs), 6
Theory and Practice of Timed Systems (TPTS), 17, 18
Three Amigos, 24
time stopping deadlock, 63
time-out event, 38
timed automata, 22
    hierarchical, 53
timed automata model, 14, 22
timed automaton, 126
timed computation tree logic (TCTL), 49
timed configuration, 126
timed step, **127***
timed system, 126
    semantics of a, 129
timed trace of an HTA, 63
timed trace semantics, 63
timed trace semantics of HTA, **63***
timer, 38
timing constraints, 125
top-down approach, 10
touch (of a candidate), **162***
TPHOLs: Theorem Proving in Higher Order Logics, 6
TPTS: Theory and Practice of Timed Systems, 17, 18
trace, 44
trace of a UPPAAL model, **47***
trace semantics of UPPAAL, **48***
trace-based model checking, 122
transition, 42
    of a transition, 56
    target of a, 56
transition relation, 122, 126
TransitionEnabled, 61
transitions
    outgoing, 42
trigger, 69
trigger event, 35
Turing machine, **64***
    language accepted by a, **65***
2-merge, 168

UML
    0.9, **24***
    1.0, **24***
    1.1, **24***
    1.2, **25***
    1.3, **25***
    1.4, **25***
    levels, 25
    profile, 27
    semantics, 30
UML profile, 25
UML: Unified Modeling Language, 24
undecidability, 65
Unified
    Method 0.8, **24***
    Modeling Language, 24–32
Unified Modeling Language (UML), 24
union of DBMs, 83
universal path properties

approximation of, **113**⋆
semantics of, **112**⋆
universal path property, **112**⋆
universal Turing machine, 65
universal until, 129
unpack reduced constraint system before inclusion check (`-U`), **96**⋆
unstable configuration, **182**⋆
UPPAAL model, **45**⋆
UPPAAL process, **45**⋆
urgent location, 42
UrgentEnabled, 61

validity
of a local property, 49
of a temporal property, **50**⋆
vector of locations, 45
ventricular chamber, 186
verification, 2
formal, 9
verification engineer, 6
verification of real-time systems, **77**⋆
view, 29
VISUALSTATE, 16

W3C: World Wide Web Consortium, 28
weak preservation requirement, 124
weak trace, **113**⋆
well-formed sequence, 47
WOODDES project, 71
Workshop on Object-Oriented Design and Development of Embedded Systems (WOODDES), 71
World Wide Web Consortium (W3C), 28

XMI: XML Metadata Interchange, 29
XML Metadata Interchange (XMI), 29
XML: eXtensible Markup Language, 27
XOR predicate, 56

zeno, 127
zeno traces, 47
zone, **81**⋆

Abbreviations

AIT-WOODDES: Advanced Information Technology—Workshop on Object-Oriented Design and Development of Embedded Systems, 71

BDD: Binary Decision Diagram, 11, 83

CADE: Correct Hardware Design and Verification Methods, 6
CASE: Computer-Aided Software Engineering, 37
CAV: Computer Aided Verification, 6, 77
CDD: Clock Difference Diagram, 83
CHARME: Correct Hardware Design and Verification Methods, 6, 18
CORBA: Common Object Request Broker Architecture, 28
CRD: Clock Restriction Diagram, 83

DBM: Difference Bounded Matrix, 82
DDD: Difference Decision Diagram, 83
dtd: document type definition, 28

ETAPS: European joint conference on Theory and Practice of Software, 15

FASE: Fundamental Approaches to Software Engineering, 17, 18
FDDI: Fiber Distributed Data Interface, 100
FMCAD: Formal Methods in Computer Aided Design, 5, 91

GIOP: General InterORB Protocol, 28

HOL: Higher Order Logic, 9
HTA: hierarchical timed automaton, 55
HTML: hypertext markup language, 28

ICS: I Can Solve, 141
IIOP: Internet Inter-ORB Protocol, 28

LCF: Logic of Computable Functions, 9
LTL: Linear Temporal Logic, 94

MOF: Meta-Object Facility, 28

NQTHM: aka Boyer-Moore theorem prover, 9*
NWPT: Nordic Workshop on Programming Theory, 18

OCL: Object Constraint Language, 30
OMA: Object Management Architecture, 28
OMG IDL: OMG interface definition language, 28
OMG: Object Management Group, 24
OMT: Object Modeling Technique, 24
OOSE: object-oriented software engineering, 24
ORB: Object Request Broker, 28
OS: Operating System, 15

RED: Region Encoding Diagram, 83
RFP: request for proposal, 25
ROPES: Rapid Object-Oriented Process for Embedded Systems, 31
RTF: Revision Task Force, 24
RTOS: RealTime Operating System, 69

SAL: Symbolic Analysis Laboratory, 12
SDL: Specification and Description Language, 16
SGML: Standard Generalized Markup Language, 28
SPE: Software Process Engineering, 28

TCTL: timed computation tree logic, 49
TPHOLs: Theorem Proving in Higher Order Logics, 6
TPTS: Theory and Practice of Timed Systems, 17, 18

UML: Unified Modeling Language, 24

W3C: World Wide Web Consortium, 28
WOODDES: Workshop on Object-Oriented Design and Development of Embedded Systems, 71

XMI: XML Metadata Interchange, 29
XML: eXtensible Markup Language, 27