Analyzing Real-Time Systems: Theory and Tools

Thomas S. Hune

BRICS Dissertation Series DS-01-3

ISSN 1396-7002 March 2001

Analyzing Real-Time Systems: Theory and Tools

Thomas Seidelin Hune

Ph.D. Dissertation

Department of Computer Science
University of Aarhus
Denmark

Analyzing Real-Time Systems: Theory and Tools

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

by
Thomas Seidelin Hune
January 31, 2001

Abstract

The main topic of this dissertation is the development and use of methods for formal reasoning about the correctness of real-time systems, in particular methods and tools to handle new classes of problems. In real-time systems the correctness of the system does not only depend on the order in which actions take place, but also the timing of the actions. The formal reasoning presented here is based on (extensions of) the model of timed automata and tools supporting this model, mainly UPPAAL. Real-time systems are often part of safety critical systems e.g. control systems for planes, trains, or factories, though also everyday electronics as audio/video equipment and (mobile) phones are considered real-time systems. Often these systems are concurrent systems with a number of components interacting, and reasoning about such systems is notoriously difficult. However, since most of the systems are either safety critical or errors in the systems are costly, ensuring correctness is very important, and hence formal reasoning can play a role in increasing the reliability of real-time systems.

We present two classes of cost extended timed automata, where a cost is associated to an execution of the automaton. We show that calculating the minimum cost of reaching a location in the automaton, the minimum-cost reachability problem, is decidable for both classes. Since a number of optimization problems, e.g. scheduling problems in a natural way, can be modeled using cost extended timed automata, we can now solve these problems using extensions of timed model checkers. The state-space of the simpler class, *uniformly priced timed automata* (UPTA), which is a subclass of *linearly priced timed automata* (LPTA), can effectively be represented using a slightly modified version of the well known difference bounded matrix (DBM) data-structure for representing zones, used in most timed model checkers. Using an extension of the region construction, the minimum-cost reachability problem can also be solved for LPTAs. However, the standard way of using zones for representing the state-space cannot be used for LPTAs, since there is no way of capturing the cost of states. Based on the notion of facets, zones can be split into smaller zones which can be represented by extended DBMs in an effective way. Minimum-cost reachability for both UPTAs and LPTAs have been implemented in extensions of UPPAAL, and successfully tested on a number of case studies. In particular, part of the Sidmar steel production plant, which is a case study of the Esprit VHS project, has been studied. Schedulability, without considering cost and optimality, has also been addressed using standard timed automata and UPPAAL. In order to solve the schedulability problem in UPPAAL it proved crucial to add a number

of *guides* to the model, in order to limit the search space. In the cost extended versions of UPPAAL, guiding in terms of changing the order in which states are searched has also been used, and shown to be effective both for finding solutions to optimization problems and in ordinary timed model checking.

The second extension of timed automata is parametric timed automata, where parameters can be used in expressions for guards and invariants. We consider the problem of synthesizing values for the parameters ensuring satisfiability of reachability properties. Since there are in most cases infinitely many values ensuring that a property is satisfied, the result is presented in terms of constraints for the parameters. We present a semi-decision procedure synthesizing the constraints. The problem of synthesizing constraints for the parameters has been show to be undecidable. To represent the state-space we extend the DBM data-structure to parametric DBMs, capable of representing zones were the bounds are given by expressions including parameters. The semi-decision procedure is implemented in UPPAAL and constraints ensuring correctness of a number of industrial protocols is synthesized.

Since (timed) reachability checking requires large amounts of resources in terms of memory and CPU time, we have studied the possibility of distributing the reachability checking to a network of computers. We have implemented a distributed version of UPPAAL and tested it on a number of the largest known case studies for UPPAAL. Not only did we achieve effective usage of all the connected computers (close to linear speedup in the number of computers) we also discovered that the breadth-first search order, which previously has been considered to be the best known, is not optimal.

We apply the general categorical framework of *open maps* to timed automata by presenting a category where the elements are timed automata, and a subcategory suitably for representing observations, timed words. Following the framework, maps in the category can be seen as simulations, and two timed automata $\mathcal{A}$ and $\mathcal{B}$ are timed bisimilar if and only if there exists a timed automaton $\mathcal{C}$ and open maps $\mathcal{C} \rightarrow \mathcal{A}$ and $\mathcal{C} \rightarrow \mathcal{B}$. We show that this notion of timed bisimulation coincides with the know notion of timed bisimulation, and using the region construction show that the bisimulation is decidable.

Building timed automata models of systems can be an error prone and time consuming task. We address this problem by presenting a translation from a low level programming language used in the programmable LEGO® RCX™ brick to timed automata. Programs for the RCX™ brick can consist of several tasks running concurrently. Therefore an important part of the model of the program is a model of the scheduler. The translation has been implemented and tested on a control program for a car.

Finally, we consider a kind of partial program synthesis for untimed systems. Given a safety specification written in monadic second order logic, we use the Mona tool to derive an automaton accepting the language of the specification. The automaton is used to restrict the executions of a handwritten control program, ensuring that the safety requirements are met. To demonstrate the approach we consider a control program for a crane, written for the RCX™ brick. We also discuss more generally what should happen when there is a conflict between the actions of the control program and the specification.

Acknowledgments

First and foremost I would like to thank my supervisor Mogens Nielsen for skillfully guiding me through my time as a PhD student. During this time he has not only inspired me to investigate various interesting topics, but also encouraged me to diverge into areas of my interest, still being able to guide me in the right directions.

I would also like to thank Prof. Kim G. Larsen for directing me into the world of timed model checking and including me in many of his research activities. He has always found the time to discuss ideas and thoughts on timed model checking and its applications. Also, he introduced me to the Esprit Project *Verification of Hybrid Systems* (VHS) and has invited me to spend much time at the BRICS group at University of Aalborg. The importance of both of these opportunities cannot be underestimated. Also thanks to everyone at the Aalborg part of BRICS for receiving me well. A special thank goes to Paul Pettersson and Gerd Behrmann for many hours of interesting discussions, co-authoring a number of papers, and enjoyable travels abroad. I will also like to take the opportunity to thank everyone involved in the VHS project for some very interesting meetings.

A special thanks also goes to Prof. Frits Vaandrager for letting me visit his group, ITT, at the University of Nijmegen. I would like to thank everyone there for receiving me very warmly, giving me insight in Dutch traditions such as Sinter Klass, and teaching about the life (and looting) of the Danish Vikings. A special thanks to Marieke Huismann for helping me with various practical problems and finding the time to show me Nijmegen and its surroundings. Thanks to Judi Romijn, Ansgar Fhenker and Mariëlle Stoelinga for many interesting discussions, including discussions on computer science, and for co-authoring a number of papers.

During my time as a PhD student I have shared offices with a number of different people: Carsten Butz, Marcin Jurdziński, Bernd Grobauer, Anders Møller and Flemming Rodler. With these I have enjoyed many interesting and fruitful discussions ranging from various topics within computer science to the world far beyond, for which I am grateful.

I would also like to thank everyone at BRICS in Århus for creating a very dynamic and pleasant research environment, enriched by its many visitors. Also thanks to the secretaries for helping out with all kinds of practical problems keeping the administrative overhead for a PhD student at a minimum.

A special thanks also goes to my study group from my time as an undergraduate student, Claus Brabrand, Flemming Rodler, and Tom Sørensen, for sharing many hours of frustration with me. Without them I would probably never have made it through my first years at university.

Last, but not least, I would like to thank my parents, Elna and Jes Hune, for supporting me throughout all of my studies in all possible ways. I would also like to thank my girlfriend, Lone Juul Hansen, for all her love and understanding.

Thomas Seidelin Hune,
Århus, January 31, 2001.

Contents

Chapter 1

Introduction

The main topic of this dissertation is formal reasoning about the correctness of real-time systems, in particular development of methods and tools for handling new classes of problems. As a part of this two more fundamental decidability results are also presented. A minor part of the dissertation is concerned with support for programming real-time systems.

In this chapter some of the basic concepts used throughout the dissertation are presented. Especially, the model of timed automata used to represent real-time systems is defined formally.

1.1 Real-Time Systems

A *real-time system* is a system where the correctness of the system depends on the time at which actions take place, and not only on their order. The theory and methods presented in this dissertation are developed with systems controlled by computers in mind and, unless stated explicitly, the term real-time system will in the following refer to a system controlled by computers. However, it may as well be applied to systems where computers play no role.

Many of the computer controlled systems we use in our everyday life, often called *embedded systems*, fall in the category of real-time systems. Some examples are washing machines, mobile phones, and audio/video equipment. These are examples of so called *soft* real-time systems, since a single failure to meet the specified timing requirements can be accepted at times. For soft real-time systems one often talks about *quality of service* instead of correctness. Another class of real-time systems is known as *hard* real-time systems. For hard real-time systems a single failure to meet a timing constraint cannot be tolerated. Hard real-time systems are found in many *safety critical* systems such as planes, trains, and factories, where a failure can be very costly or even cost lives. Therefore the correctness of hard real-time systems is very important. A number of communication protocols also fall into the category of hard real-time systems since a single error can stop communication completely. The methods presented in this dissertation focus on checking the existence of errors and not on how often they occur. Therefore we are only considering hard real-time systems.

Most real-time systems consists of several components which operate in parallel and communicate with each other. Such systems are known as *concurrent systems*. The real-time systems we are focusing on in this dissertation are, like most real-time systems, built to control an environment through different kinds of interaction, not compute a result based on some input. Such systems, called *reactive systems*, are therefore not supposed to stop but work continuously.

We will assume that time can be measured on a common time *scale* for all components in the system, including the environment. However, we do not require, that time is global in the sense of the existence of one global clock. Each part of the system may have its own local clock which can be started and stopped independently. Having a common time scale allows for comparing the times measured in different components.

One way of getting experience with a physical real-time system is to build one e.g. using LEGO® Mindstorms™. Being able to work with a physical real-time system can be very useful in understanding how it works and how it relates to what is modeled. The RCX™ brick which is part of Mindstorms™ is a large brick with a built-in computer. It can be used for controlling LEGO® models through three output ports which can be connected to LEGO® motors. The RCX™ brick is also equipped with three input ports each of which can be connected to one of four different kinds of sensors (heat, light, rotation, and touch), supplied by LEGO®. This enables one to build a reactive system interacting with the environment (the LEGO® model) through the input and output ports. For communication the RCX™ brick has a built-in I/R sender and receiver. This is intended for downloading programs from a PC. However, this can also be used for communication between different RCX™ bricks enabling one to build concurrent systems controlled by several independent RCX™ bricks communicating via I/R signals. As part of this dissertation, we have developed control software for LEGO® models, since access to 'real' real-time systems like a steel plant, has not been possible.

1.2 Formal Reasoning

The problem of ensuring that real-time systems are correct is very important, since the consequences of failures can be serious. Reasoning about concurrent systems is known to be difficult, since their behavior is complex due to the possible interaction between the different components. Including real-time behaviors does not make this task easier. Defining the properties a system has to satisfy to behave correctly can also be difficult because many different possible behaviors have to be taken into consideration. For these reasons descriptions of systems and requirements written in plain language are often very long, difficult to interpret correctly, and reasoning about them by hand error prone. One way of removing the ambiguities and making the reasoning about the system less error prone, is to base this on a mathematical foundation.

When reasoning formally about a system one needs two components; a *formal model* of the system and a *formal specification* of the properties the system should satisfy. Both are mathematical objects described in a formal language which need not be the same for both objects. Using formal reasoning one can then attempt to prove whether the formal model satisfies the formal specification or not. The possible behaviors a formal model can exhibit is defined via a *formal semantics* of the model.

Logic based formalism are often used for describing both the model and the specification. One can then use logical implication for checking whether the model satisfies the specification. Describing both model and specification in an automata based formalism is also used often; language inclusion is then typically used for checking whether the specification is satisfied. Mixing these two approaches and representing the model in an automaton based language and the specification as a logical formula is also common. In this case the satisfiability of the formula is defined over the semantics of the automaton. Describing both model and specification in a process algebra and using equivalence checking to decide whether the property is satisfied is yet another approach. These should only be seen as some examples of classes of formal models, formal specifications and how they are checked, many others exist. In this dissertation the focus will be on describing the models in an automaton based formalism (timed automata) and the specifications in a real-time logic.

Though the use for formal methods is based on a mathematical framework with unambiguous foundations, formal reasoning by hand is error prone. Therefore, tool support for doing the formal reasoning can be very useful. Many tools exist for supporting the checking of whether a model satisfies a specification. Theses range from *theorem provers* where the user in many cases has to assist the construction of the proof, to *model checkers* where a 'witness' of whether the model satisfies the specification is found automatically (if possible). Combining these two approaches has lately been a topic for much research.

Another way of applying formal methods is to *synthesize* a system based on a formal specification. However, the synthesis problem is often harder to solve than the problem of checking whether a given model satisfies its specification. In the last chapter we address the problem of mixing synthesized code with existing code. This is the only part of this dissertation which does not consider real-time problems.

1.3 Timed Automata

The formal model which will be used throughout this dissertation is (networks of) *timed automata*. In this section the model of networks of timed automata used in the model checking tool UPPAAL is formally defined. First we will present timed automata and then extend these to networks of timed automata. Several variations of timed automata exist, however, all of these are based on the definition of timed automata presented in [12] and the similar timed graphs defined in [8]. The model presented in [12] is an extension of Büchi automata, adding a set of real-valued variables called *clocks* or *clock variables* to the automata. On transitions the value of one or more clocks can be reset to zero. During an execution the value of all the clocks increase with the same rate, measuring the time since they were last reset. By adding constraints, called *guards*, over the clocks on the edges of a Büchi automata the possible executions can be restricted based on the values of the clocks.

In [75] *timed safety automata* were introduced, very similar to the timed automata of [12] and the timed graphs of [8] but extended with an *invariant* in locations. An invariant is a guard associated to a location, which must be satisfied while the automaton is in the location. Using invariants is a way of ensuring local progress, since an invariant can ensure that it is not possible to stay in the same location for ever.

The timed automata used in UPPAAL are automata over finite strings (where all locations are implicitly considered accepting locations). In all other respects they are very similar to the timed safety automata of [75]. Since the timed automata of UPPAAL is used as a modeling language for a number of case studies, a number of features has been added to ease modeling. These will be presented in the last part of this section.

We start by defining the clock guards which can be used to restrict the possible executions.

Definition 1.1 (Clock Guard) *Let $\mathbb{C}$ be a set of clocks. A* guard *is a conjunction of simple guards of the form $x - y \bowtie n$ or $x \bowtie n$ where $x, y \in \mathbb{C}$, $\bowtie \in \{<, \leq, \geq >\}$ and n is a natural number. The set of guards is denoted $\mathcal{B}(\mathbb{C})$.*

Let *Act* be a finite set of action, and let $\mathcal{P}(\mathbb{C})$ denote the power set of $\mathbb{C}$.

Definition 1.2 (Timed Automata) *A* timed automaton *A over a set Act of actions and a set $\mathbb{C}$ of clocks is a tuple (L, l_0, E, I) where*

- *L is a finite set of locations,*
- *$l_0 \in L$ is the initial location,*
- *$E \subseteq L \times \mathcal{B}(\mathbb{C}) \times Act \times \mathcal{P}(\mathbb{C}) \times L$ the set of edges, consisting of a source location, a guard, an action, a set of clocks to be reset, and a target location, and*
- *$I : L \rightarrow \mathcal{B}(\mathbb{C})$ a function assigning invariants to locations.*

An edge $(l, g, a, r, l') \in E$ is written $l \xrightarrow{g,a,r} l'$.

Example 1.1 *The timed automaton in Figure 1.1 is a model of a telephone. It has seven locations and one clock x. The initial location is the location Idle with a circle inside. When moving from Idle to Lifted the clock x is reset ($x := 0$). It is not possible to stay in location Lifted for more than 20 time units because of the invariant $x \leq 20$ in the location. The transition from Lifted to Error is not enabled until at least 20 time units has passed because of the guard $x \geq 20$, whereas the transition from Lifted to One_digit is always enabled.*

A state of an execution consists of the location of the automaton and the value of all the clocks. To represent the values of the clocks we define clock valuations.

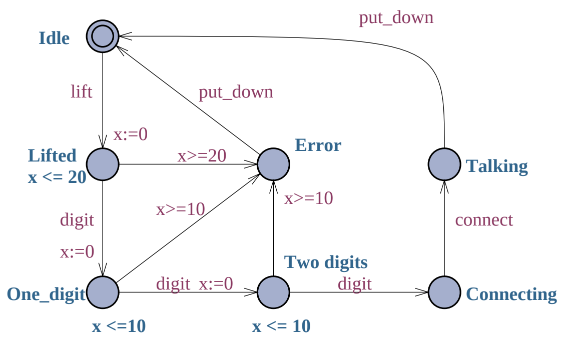


Figure 1.1: The timed automaton representing a phone of Example 1.1.

Definition 1.3 (Clock Valuation) *A* clock valuation *over a set of clocks* $\mathbb{C}$ *is a function* $v : \mathbb{C} \to \mathsf{R}_{\geq 0}$. *Two operations are defined on a clock valuation;* adding *a constant* $d \in \mathsf{R}$ *to a clock valuation* $(v + d)(x) = v(x) + d$ *for* $x \in \mathbb{C}$, *and* resetting *the clocks in a set* r, $v[r](x) = 0$ *iff* $x \in r$ *and* $v[r](x) = v(x)$ *otherwise.*

A clock valuation v satisfies a guard g if, when substituting the values in v for the clocks in the expression for g, the expression evaluates to true. We will write this as $v \in g$ and similarly for invariants.

The semantics of a timed automaton is defined in terms of the behavior of a labelled transition system where the set of states is states of the execution of the automaton.

Definition 1.4 (Semantics of Timed Automata) *We define the semantics of the timed automaton* (L, l_0, E, I) *as a labelled transition system with states* $L \times \mathsf{R}^{\mathbb{C}}$ *with initial state* (l_0, v_0) *where* v_0 *is the clock valuation assigning zero to all clocks. The transitions are given by the following transition relation*

- $(l, v) \xrightarrow{d} (l, v + d)$ *if* $\forall 0 \leq e \leq d : v + e \in I(l)$,
- $(l, v) \xrightarrow{a} (l', v[r])$ *if* $l \xrightarrow{g,a,r} l'$, $v \in g$, *and* $v[r] \in I(l')$.

The first type of transition is called a delay *transition and the second type an* action *transition.*

In Section 1.1 it was mentioned that most real-time systems are concurrent systems consisting of several components. Therefore it is natural to be able to describe a system by several timed automata executing in parallel and being able to communicate. For this purpose *networks* of timed automata is introduced. Communication in the network takes place between two automata and is synchronous.

Definition 1.5 (Parallel Composition) *Let* $A_i = (L_i, l_{i,0}, E_i, I_i)$ *for* $i = 1, 2$ *be two timed automata over the same set Act of actions, where the set of edges*

E_i has been extended with a transition $l \xrightarrow{tt,0,\emptyset} l$ for each location in L_i. This is a special null transition which is used to enable a transition in only one of the automata. Let $f : Act \cup \{0\} \times Act \cup \{0\} \to Act$ be a function which we will call a synchronization function. The parallel composition *of A_1 and A_2 is $A_1|^f A_2 = (L_1 \times L_2, (l_{1,0}, l_{2,0}), E, I)$ where $I(l_1, l_2) = I_1(l_1) \wedge I_2(l_2)$ and $(l_1, l_2) \xrightarrow{g,a,r} (l_1', l_2') \in E$ iff there exists transition $l_i \xrightarrow{g_i,a_i,r_i} l_i' \in E_i$ such that $g = g_1 \wedge g_2$, $f(a_1, a_2) = a$ and $r = r_1 \cup r_2$.*

The systems analyzed in UPPAAL are all considered *closed* systems, so the properties of the system only depends on the automata described in the network.

Instead of defining the synchronization explicitly when presenting networks of timed automata in UPPAAL or in figures we make actions complementary as in CCS [121] by adding either a '!' or a '?' as a postfix to the action. Synchronization between two automata then takes place over a *channel* which has the same name as the action used for the communication. In UPPAAL all edges with an action must synchronize with another edge with the complementary action over a channel, therefore internal edges cannot have a label.

The networks of timed automata used in UPPAAL has been extended with bounded integer variables and arrays of bounded integer variables which can be very helpful when modeling. Guards over integer variables are possible as are updates of integer variables. The state of an execution also contains the values of the integer variables. In [32] the notion of *committed location* was introduced. A committed location is a special kind of location in which the timed automaton is not allowed to stay. This means that when a committed location is entered in one of the automata in a network, delay transitions are not allowed and the next action transition *must* involve the automaton which entered the committed location. Committed location are useful for modeling different things such as the sending of a broadcast message only using handshake communication, or control actions being much faster than the rest of the system modeled. Another kind of location is *urgent locations*, which are somewhat like committed locations. When a timed automaton is in an urgent location, delay transitions are not allowed, however, all action transitions are allowed also the ones not including the automaton in an urgent location. Channels can also be defined to be urgent. When synchronization over an urgent channel is possible no delay transitions are allowed, but all action transitions are. Only guards over integer variables are allowed on transitions which is part of an urgent channel (edges labeled with the name of an urgent channel).

Committed and urgent locations and urgent channels do not add to the expressiveness of timed automata. They can all be 'simulated' using standard timed automata. However, simulating the constructions would make the models more complex and therefore more difficult to understand.

Example 1.2 *If all the action labels in Figure 1.1 gets a '?' added as a postfix, we can use the timed automaton in a network with the timed automaton in Figure 1.2 which represents the user of the phone. The timed automaton in Figure 1.2 has two integer variables* d *and* connected*. The variable* d *counts the number of digits dialed by the user, and* connected *represents whether the* *network has established a connection or not. Therefore we also need a timed automaton representing the network to complete the model.*

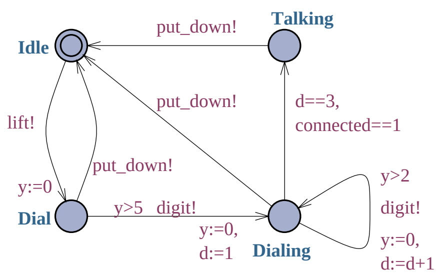


Figure 1.2: The timed automaton of a person using a phone.

The models presented in Examples 1.1 and 1.2 are representing an abstract view of a person using a phone. Making models which are abstract versions of what is modeled is important for several reasons. First of all the tools built to support formal reasoning cannot handle very large systems mainly because of the state-space explosion problem. Constructing very detailed models of systems also takes a very long time and often results in errors in the model which can be difficult to discover. If an error is found in a design including lots of details which are not important for the problem at hand, this also makes correcting the design harder. Leaving out too many details is even worse, since the results obtained based on the model might be meaningless. Finding the right level of abstraction when modeling is therefore one of the key points when reasoning about real systems based on formal models like timed automata.

1.4 Outline of Dissertation

The rest of this dissertation falls in four main parts. The first part contains an overview of the presented material in the remaining three parts, which each consist of a number of papers presenting the research conducted during my Ph.D. program. The first part consists of three chapters. Each chapter introduces one of the remaining parts by giving a short survey of the results presented in the papers and relating this to other results from the literature.
Region Based Methods The second part consists of the two papers presenting dedicability results based on the use of regions.

[28] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn and F. Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. In *Proceedings of Hybrid Systems: Computation and Control*, pages 147–161, 2001.

[128] M. Nielsen and T. Hune. Bisimulation and Open Maps for Timed Transition Systems. In *Fundamenta Informatica*, 38, special issue dedicated to Professor Arto Salomaa, pages 61–77, 1999.

Timed Reachability Analysis The third part consists of five papers all presenting results based on (forward) timed reachability analysis. In this part all the papers are based on the tool Uppaal.

[30] G. Behrmann, T. Hune, and F. Vaandrager. Distributing Timed Model Checking – How the Search Order Matters. In *Proceedings Computer Aided Verification, CAV 2000*, pages 216–231, 2000.

[82] T. Hune, K. G. Larsen, and P. Pettersson. Guided Synthesis of Control Programs Using Uppaal. In *Proceedings of the IEEE ICDCS International Workshop on Distributed Systems Verification and Validation*, pages E15–E22, 1998.

[26] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson and J. Romijn. Efficient Guiding Towards Cost-Optimality in Uppaal. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001*, pages 174–188, 2001.

[105] K. G. Larsen, G. Behrmann, E. Briksma, A. Fehnker, T. Hune, P. Pettersson and J. Romijn. As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata. To appear in *Proceedings of Computer Aided Verification*, 2001.

[86] T. Hune, J. Romijn, M. Stoelinga and F. Vaandrager. Linear Parametric Model Checking of Timed Automata. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, TACAS2001*, pages 189–203, 2001.

Development Methods The fourth part consists of two papers presenting methods and tools for helping in the construction of real-time systems.

[78] T. Hune. Modeling a real-time language. In *Proceedings of Workshop on Formal Methods for Industrial Critical Systems, FMICS'99*, pages 259–282, 1999.

[89] T. Hune, and A. Sandholm. A Case Study on using Automata in Control Synthesis. In *Proceedings of Fundamental Approaches to Software Engineering, FASE 2000*, pages 349–362, 2000.

Part I

Overview

Chapter 2

Region Based Methods

In (almost) all cases the *concrete* semantics of a timed automaton in Definition 1.4 gives rise to a labelled transition system with uncountably many states and transitions. Therefore representing the set of reachable states of a timed automaton, the *state-space*, in this way is not suited as a basis for algorithms. Instead of using the concrete semantics as a representation of the state-space various *symbolic* representations have been suggested. Common to all these is that each symbolic state represents a set of concrete states, and that only a finite number of symbolic states is needed to represent all the concrete states. This make symbolic representations suitable as a basis for algorithms exploring the state-space. The first symbolic representation of the state-space of a timed automaton was based on the notion of *regions* [12, 13]. In this chapter we will focus on results based on regions, and Chapters 5 and 6 present two fundamental decidability results based on regions.

2.1 Regions

Since a state consists of a location and a clock valuation, and there are finitely many locations, we are interested in a finite partition of the clock valuations. Not only should a partition be finite, it must also reflect the properties of the concrete semantics. The properties can vary depending on the use of the symbolic model. However, as a minimum it seems reasonable that reachability properties should be reflected.

Regions offer such a finite partitioning of the clock valuations reflecting reachability in the concrete semantics. The partitioning is based on the observation that some clock valuations behave in very much the same way. Two clock valuations can be considered to behave in a similar way if they satisfy the same set of guards and if they, when time passes, can reach new clock valuations which also can be considered to behave similarly. For two clock valuations to satisfy the same set of guards of the type $x \bowtie n$, the integer value of each clock must be the same, and to satisfy the same set of guards of the type $x - y \bowtie n$, the ordering among the values of the clocks must also be the same. The next clock in a clock valuation to change integer value when time passes, is the clock with the largest fractional part. If the ordering among the clocks is the same in two clock valuations and they agree on the integer part of the value of the clocks, then it will also be the same clock which changes integer value next when time passes. This ensures that the two clock valuations also behave in the same way whenever some time has passed and different guards may be enabled. The partitioning arising from these requirements is not finite since the clocks can grow unbounded, but it is countable. The final observation needed in order to reach a finite partitioning is that when the value of a clock is larger than any constant used in a guard, it is not important how much larger it is. Based on these observations we arrive at the following definition of a region.

Definition 2.1 (Region [12, 13]) *Let A be a timed automaton over a set of clocks $\mathbb{C}$. For each $x \in \mathbb{C}$, let c_x be the largest integer n that is compared to x in any guard in A. Let $\lfloor v(x) \rfloor$ denote the largest integer smaller than or equal to $v(x)$ and $\mathit{fract}(v(x)) = v(x) - \lfloor v(x) \rfloor$. A* region *is an equivalence class of the equivalence relation $\cong$ over clock valuations, where $v \cong v'$ iff*

- *for each $x \in \mathbb{C}$: $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ or both $v(x) > c_x$ and $v'(x) > c_x$,*
- *for every pair of clocks $x, y \in \mathbb{C}$ where both $v(x) \leq c_x$ and $v(y) \leq c_y$, $\mathit{fract}(v(x)) \leq \mathit{fract}(v(y))$* iff *$\mathit{fract}(v'(x)) \leq \mathit{fract}(v'(y))$,*
- *for every clock $x \in \mathbb{C}$: $\mathit{fract}(v(x)) = 0$ iff $\mathit{fract}(v'(x)) = 0$ or both $v(x) > c_x$ and $v'(x) > c_x$.*

Example 2.1 *Consider a timed automaton with two clocks x and y where $c_x = c_y = 2$. There are 44 possible regions for this automaton which are shown in Figure 2.1. The regions in the figure are: 9 vertices, 22 open line segments, and*

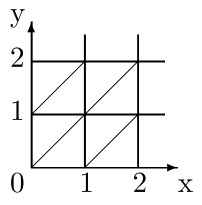


Figure 2.1: The regions of the timed automaton in Example 2.1.

13 open areas. The open line segments are both lines such as the one between the points $(0, 1)$ and $(1, 2)$ (the points not included), and unbounded line segments such as the one from $(1, 2)$ (not included) and parallel with the y-axis. The open areas are either 'triangles' such as the one with vertices $(0, 0), (1, 1)$ and $(1, 0)$ (again not including the points or the lines between them), or unbounded areas such as the one where all points have y value larger than 2 and lies between the y-axis and the line $x = 1$.

Letting symbolic states be pairs of a location and a region, one can define a symbolic state-space of a timed automaton called the *region graph* in [12, 13]. To be able to build the region graph of a timed automaton, some operations on regions are needed. For deciding whether an edge is enabled or not we need to be able to decide whether a region satisfies a guard. Given a region we need to be able to calculate the next reachable region when time is passing, and finally we need to be able to reset a number of clocks in a region. With a data-structure for representing regions and these three operations on the data-structure, one can generate the region graph of a timed automaton.

The number of regions for a timed automaton is exponential in the number of clocks and it is very much influenced by the size of the constants used in guards and invariants. Given a set of clocks $\mathbb{C}$ and a constant c_x for each clock in $\mathbb{C}$, the number of regions is bounded by $|\mathbb{C}|! \cdot 2^{|\mathbb{C}|} \cdot \Pi_{x \in \mathbb{C}}(2c_x + 2)$ [12, 13].

Because of the large number of possible regions, representing a state-space using regions is not very efficient. However, some (mostly early) tools use regions as part of the symbolic states used to represent the symbolic state-space. Some of these are Epsilon [51], RT-Cospan [16] (versions using other representations exist), and more recent RED [142] which uses a structure very similar to regions. In Chapter 3 we will discuss other more efficient ways of representing the state-space.

Due to the regular and intuitive structure of regions they have proven very useful as a basis for a number of fundamental results concerning timed automata. In particular a number of decidability results have been proven based on (algorithms using) regions, e.g. [12, 13, 49, 103, 68, 84, 40, 28, 101]. Many of these prove decidability by explicitly stating an algorithm using the operations defined on regions, and have since been extended to more efficient ways of representing sets of states.

2.2 Timed Automata Extended with Linear Cost

Timed verification tools like Uppaal [109] and Kronos [41] have been used to solve a number of optimization problems, especially as part of the VHS [1] project where scheduling problems for different batch plants [64, 82, 127, 100] have been addressed. The problems are modeled using timed automata such that the optimization problem can be reformulated as a (number of) reachability questions which can be solved by the tools. The examples from the VHS project are scheduling of tasks in a production.

Probably one of the first and most well known of these problems solved by timed verification tools is the Bridge problem presented in [134]. Four soldiers have to cross a bridge in the dark with different constraints and physical abilities. The question is how fast can they all cross the bridge. However, when using tools like Uppaal and Kronos, questions like how fast can a state be reached where all soldiers have crossed the bridge, cannot be asked. Instead it is possible to ask whether a state is reachable, where all the soldiers have crossed the bridge and no more than e.g. 60 time units have passed. Asking several questions like that with different time bounds, the minimum time for reaching a set of states can be determined.

One might not only be interested in optimizing the consumption of time but in optimizing a more general notion of *cost*. In Chapter 5 we present the model of *linearly priced timed automata* which is an extension of timed automata. The

[1]Verification of Hybrid Systems, European Community Esprit-LTR Project 26270.

edges in a linearly priced timed automaton are labelled with a price, so taking an action transition has a cost. Locations are also labelled with a price, called a rate, which is the cost per time unit of delaying in that location, so taking a delay transition also has a cost. The question asked is, what is the minimum cost of reaching a given location.

In Chapter 5 we show that the *minimum-cost reachability* problem is decidable by presenting an algorithm solving the problem. The algorithm is based on an extension of regions to *priced regions* where each vertex of a region has an associated cost. The operations on regions are extended to priced regions and three new operations are defined which are needed for the algorithm. Termination of the algorithm is shown using Higman's lemma [76].

The operations on priced regions are given in a way suitable for implementation. However, because of the inherent inefficiency of regions based algorithms we have chosen not to implement the algorithm.

The model of linearly priced timed automata can be seen as a special case of multitriangular hybrid automata [72] or linear hybrid automata [72], where one real-valued variable is used to represent the cost. Reachability is known to be undecidable for multitriangular hybrid automata [72]. In a linearly priced timed automaton the price of an execution cannot be used in guards and therefore does not influence the possible executions. Intuitively, this is the reason reachability can be decidable for linearly priced timed automata, despite the undecidability result for multitriangular hybrid automata.

Linearly priced timed automata have also been studied in [17], where they are called weighted timed automata. Here the complexity of different variations of the minimum-cost reachability problem for linearly priced timed automata is addressed. The minimum-cost reachability problem is shown to be decidable in exponential time in the size of the automaton.

The simpler problem of minimum-time reachability was solved in [126] where three different algorithms were presented. All the algorithms first generate the complete state-space (called simulation graph in [126]). One algorithm is using a backwards fixed point algorithm, while keeping track of the elapsed time. The two other algorithms are variations of depth-first search on the simulation graph with an extra clock. The algorithm presented in Chapter 5 will in the worst case also search (and therefore generate) the complete state-space. However, because of the relation check between priced regions, it might be discovered that a part of the state-space cannot contain the optimal solution, before exploring this part of the state-space. In this case the algorithm will not search that part of the state-space. In Chapter 9 this idea is extended further for a simpler model than linearly priced time automata, which also captures the problem of minimum-time reachability. Since neither of the approaches in [126] nor the algorithm presented in Chapter 5 has been implemented we cannot compare the efficiency. The algorithms in [126] are based on a more efficient representation of the state-space than regions, and therefore we expect them to be the most efficient ones even though they always generate the complete state-space.

A related but more general problem is the synthesis of a controller for a timed automaton assuming that the actions are divided into a set of controllable actions and a set of uncontrollable actions. The controller should ensure that a property is satisfied, e.g. reachability of a state, by choosing which of the controllable actions should be taken next. In [23] the problem of synthesizing a controller using minimal time to satisfy a property is solved using a backwards fixed point computation. An interesting question is whether the techniques described in Chapter 5 can be used to solve the problem of synthesizing minimum-cost controllers for linearly priced timed automata.

In Chapter 9 we solve the minimum-cost reachability problem for a simpler model using a more efficient representation of the state-space than regions, and in Chapter 10 an efficient zone based representation of linearly priced timed automata will be given. We will return to this in Chapter 3.3 and Chapter 3.4.

2.3 Open Maps and Timed Bisimulation

The notion of bisimulation equivalence was introduced for CCS in [130, 121] and has since been studied for a range of other modeling languages, especially process algebras [67, 104, 122]. The notion of *timed bisimulation* was introduced in [145] and has since been studied in a number of papers e.g. [19, 125, 49, 103, 143]. The decidability of timed bisimulation was first shown in [49] by giving an algorithm based on regions.

Since the introduction of the bisimulation equivalence for CCS a large number of other equivalences for CCS and other languages have been introduced. In [95] a general framework for defining bisimulation based on category theory is offered and a general way of defining a logic which is characteristic for the notion of bisimulation is presented. This gives a canonical way of defining a notion of bisimulation. The framework is based on the idea of defining a category with objects representing the models, and a subcategory of this representing the observations. A morphism between two objects $m : X \to Y$ specifies how the behavior of X can be observed or be simulated in Y. Intuitively, an *open* map is a morphism which also requires that the behavior of Y can be simulated in X. Two models X and Y are bisimilar if and only if there exists a model Z and two open maps $m : Z \to X$ and $m' : Z \to Y$; this is called a *span of open maps*.

In Chapter 6 we apply the framework to timed automata by presenting a category of timed automata with a subcategory of timed words. We show that the notion of timed bisimulation presented in [145, 49] coincides with the notion of timed bisimulation arising from using open maps. Deciding whether there exists an open map between two objects is shown to be decidable using regions. Decidability of timed bisimulation is shown by giving an upper bound on the size of the vertex (the 'Z' object) in terms of the size of the two systems checked. One of the advantages of the bisimulation presented in [121] is that one can present an explicit bisimulation. By defining the timed bisimulation in terms of open maps, one can exhibit a bisimulation directly in terms of the models.

The presentation of a timed bisimulation in [145] is given in terms of the unfolding of the two systems which have uncountably many states. This presentation has the disadvantages of being infinite and, moreover, not given in terms of the models. Using the approach of [49] a timed bisimulation is presented in terms of the state-space of the two timed automata (based on regions). This is a finite representation but again the bisimulation is presented in terms of different models (the region graphs).

The method for deciding timed bisimulation presented in Chapter 6 can be used as the basis of an implementation. However, the algorithm presented in [49] is much better suited for implementing. A more efficient algorithm was presented in [143], where the state-space is represented using stable zones. Stable zones are collections of regions with the same properties as a region. One advantage of stable zones is that each stable zone in most cases captures a larger area than a region (and never a smaller), therefore fewer symbolic states are needed to represent the reachable state-space. Another advantage is that the number of stable zones does not depend on the size of the constants used in the timed automaton, which is the case for regions.

Deciding the bisimulation in [121] can also be done via checking for equivalence between characteristic formulae for the two systems. A characteristic logic for timed bisimulation between timed automata was presented in [103] yielding yet another way of deciding timed bisimulation.

The open maps framework has been used to define a large range of different equivalences between different models ranging from different kinds of transition systems to Petri nets [95, 52].

Chapter 3

Timed Reachability Analysis

Due to the fine granularity of regions defined in the previous chapter, they are not suited as a representation of the state-space in tools. Therefore tools like Kronos [41] and Uppaal [109] are based on the notion of *zones* which are collections of regions. The two main ways of representing zones are Difference Bounded Matrices (DBMs) [60] and Clock Difference Diagrams (CDDs) [31] (also presented as Difference Decision Diagrams in [124]). Both Kronos and Uppaal represent zones using DBMs, which have proven to be well suited for representing the state-space of large systems. In Uppaal, zones can also be represented using CDDs but this has not been tested to the same extent as DBMs yet.

Developing and using the timed model checker Uppaal is the topic of this chapter and Chapters 7–11. The properties that can be specified in the logic used in Uppaal can all be reformulated as reachability questions. Therefore, given a (symbolic) state it is possible to check whether this particular state satisfies the property, by checking whether it is in a set S of goal states. This allows the checking of properties to be performed more efficiently than checking for properties of a more expressive timed logic such as TCTL. Using a more powerful logic like TCTL will on the other hand allow checking for more properties.

The reachability algorithm used in Uppaal (see Figure 3.1) is based on two data-structures Passed and Waiting. The states which are known to be reachable but have not been explored yet are kept in Waiting, and the states which have been explored are kept in Passed. As long as a goal state has not been encountered and there are unexplored states in Waiting the algorithm continues by selecting a new state from Waiting and exploring that state. This proceeds by checking whether the state is a goal state or not. If it is a goal state the algorithm terminates, otherwise it is checked whether the state is in Passed. Not only is it checked whether the exact same state is in Passed but whether it is included in a state or includes a state in Passed (recall that a zone consists of a collection of regions). If the state has not been encountered before, the successors of the state are generated and placed in Waiting. Here there is also a check for whether the state is included in or includes a state already in Waiting. If there are no more states in Waiting and a goal state has not been found, the algorithm terminates with a negative answer, since a

```
PASSED := $\emptyset$
WAITING := $\{(l_0, D_0)\}$
**repeat**
    get $(l, D)$ from WAITING
    **if** $(l, D) \in S$ **then**
        **return** Yes
    **if for all** $(l, D') \in$ PASSED : $D \not\subseteq D'$ **then**
        add $(l, D)$ to PASSED
        SUCC := $\{(l', D') : (l, D) \rightarrow (l', D') \wedge D' \neq \emptyset\}$
        **for all** $(l', D') \in$ SUCC **do**
            add $(l', D')$ to WAITING
        **end for**
    **end if**
**until** WAITING = $\emptyset$
**return** No
```

Figure 3.1: The reachability algorithm of UPPAAL.

goal state was not reachable.

3.1 Distributed State-Space Generation

One of the main problems when using UPPAAL and other model checkers is storing the state-space. Even for seemingly small models the reachable state-space can be too large to be stored in the memory of a workstation or PC. Searching large state-spaces is also very time consuming. We present a way of dealing with this problem by distributing the state-space search over several computers (nodes) connected in a network.

In Chapter 7 we present an approach to distributing reachability checking. The approach has been implemented in UPPAAL and tested on a number of large examples. To make the search more efficient a heuristic for the order in which the states are searched is presented. The efficiency of the heuristic is demonstrated through the examples. We also present a way of finding the shortest trace to a state in a distributed search.

The basic idea is to partition the state-space into disjoint classes and letting one node be responsible for exploring the states in each class. So each symbolic state belongs to one particular node which can be calculated using a distribution function. When a node generates the successors of a state each successor is sent to the node it belongs to. Each node runs the same algorithm which is very similar to the algorithm in Figure 3.1. The data-structures WAITING and PASSED are local which allow the algorithm to be run on both shared and distributed memory models. The main difference between the standard algorithm and the distributed algorithm is in the way newly generated successors of a state is treated. In the distributed algorithm the successors of a state are not put into WAITING on the node they were generated, but sent to the node they belong to and placed in WAITING on that node. Communication between the nodes is asynchronous. Before a node gets a state from WAITING it receives all the nodes which have been sent to it and places them in WAITING. One node is responsible for generating the initial state and sending it to the node it belongs to. If a node finds an error state it sends a termination message to the rest of the nodes and reports about the error state. Termination of the algorithm can be decided by any of the nodes when a special *termination token* has passed through all the nodes while they were idle without any change in the number of states sent or received.

One of the properties of a breadth-first search on a single node is that if a goal state is found, the trace leading to this goal state will be the shortest trace leading to any goal state. This is because all states with shorter traces have been explored. However, the search order arising when doing a distributed search does not have this property due to nondeterministic communication delays and different workloads on different nodes. When using a trace to understand an error, it is desirable that the trace is as short as possible. We have therefore added the possibility of finding the shortest trace to a goal state by including the *depth* of a state (the number of transitions used to reach the state from the initial state) to a state. When a goal state is found, all the states with smaller depth (and only states with smaller depth) are searched such that the goal state with the shortest trace can be found.

During the initial experiments with the distributed version of UPPAAL we did not obtain as good speedups as we had hoped. This was caused by an increase in the number of symbolic states explored. Generally, the number of explored symbolic states increased as the number of nodes increased, and always the fewest symbolic states were searched using breadth-first order on one node. The number of states explored depends on the order in which state are searched due to the symbolic states being compared by inclusion checks. Section 7.3.1 explains this in detail. Using the depth of a state we tried to make the distributed search order closer to breadth-first order by letting each node first explore the state with the least depth, the state closest to the initial state. The speedups gained using this heuristic were very close to linear as we had hoped. Using this heuristic, the number of explored symbolic states remained almost constant as the number of nodes increased and in some cases even decreased below the number of states explored on one node. The fact that the number of states decreased when using more than one node shows that using breadth-first search order is not optimal in the sense of generating as few symbolic states as possible when exploring the complete reachable state-space. An interesting problem is how to define an optimal search order in this sense and what the complexity of calculating it is. Such an optimal search order might in general make tools like KRONOS and UPPAAL more space efficient.

The close to linear speedups obtained were not very surprising since similar results have been obtained in [138] when constructing a distributed version of the model checker Murφ. Our approach to distributing UPPAAL was inspired by the approach in [138] and the abstract algorithms of the two tools are basically the same, so we were hoping for as good results as in [138]. Inclusion between states is not used in Murφ, therefore the number of states searched is not influenced by the search order. The only difference between the two approaches is the detection of termination. In [138] the node responsible for generating the initial state is also responsible for deciding termination by from time to time collecting from all nodes how many nodes they have send and received. We let detection of termination be something all nodes can do. Since the detection of termination takes a short time compared to the exploration of the state-space this does not influence the running time much. However, we believe that having as little central control as possible is desirable.

Similar ideas for distributing state-space generation have previously been used for generalized stochastic Petri nets in [47] and later in [71]. There, the state-space (the reachable markings) is also split between the nodes using a distribution function. Each node explores the markings sent to it and sends the newly generated markings to the node they belong to.

Before that, another approach was suggested in [7] where the state-space of the parallel composition of finite state machines is being searched. Here some nodes are responsible for generating the successors and some nodes are responsible for checking whether the newly generated states have been searched before. Using this approach one has to address the problem of how to allocate nodes to the two different tasks.

3.2 Guided Synthesis of Control Programs

One of the case studies of the VHS project is dealing with schedulability of the SIDMAR steel plant[1] in Gent in Belgium. This case study considers the part of the plant between the blast furnace and the hot rolling mill where raw iron is converted into steel by treatments in different machines [37, 129]. The raw iron is poured into ladles when leaving the blast furnace and stays in the ladle until the finished steel enters the hot rolling mill. Different qualities of steel can be produced depending on which machines are used for the treatment of the raw iron. The different treatments and the durations of these are defined in a *recipe*. The problem to be solved is to decide given an ordered list of recipes, whether it is possible to schedule the production such that the right qualities of steel are produced entering the hot rolling mill in the right order. When scheduling the production, the topology of the plant (see Figure 8.2 in Section 8.2) has to be taken into account since movement between the different machines is restricted. The time elapsed from the raw iron leaves the blast furnace until it enters the hot rolling mill is also limited because the steel must sustain a minimum temperature during the process. The treatment of a ladle takes a fixed time in the hot rolling mill and the hot rolling mill must be kept busy at all times. A correct schedule must satisfy all these constraints.

In Chapter 8 we present a timed automata model of the SIDMAR plant and reformulate the schedulability problem as a reachability question. The size of the reachable state-space is reduced by adding guiding to the model. This considerably increases the size of the problems which can be handled. Based on the traces generated from the model, controls program are automatically synthesized and executed in a LEGO plant resembling the SIDMAR plant.

The timed automaton model of the SIDMAR plant consists of a number of different timed automata representing the behavior of the different physical components of the system like the ladles and the cranes needed for moving

[1]http://www.sidmar.be/

the ladles in some parts of the plant. Each recipe is also represented by a timed automaton, and each ladle is connected to one recipe. If it is possible to schedule the production, the final state of a timed automaton monitoring the production is reachable, otherwise it will not be. If the final state is reachable, UPPAAL also produces a trace leading to the final state. The trace leading to the final state defines a schedule for the production, specifying where the ladles should be treated, for how long, and how they should move between the different machines.

Unfortunately, UPPAAL is not able to handle models of this kind with more than two ladles. This is because the reachable state-space is very large due to the high degree of freedom in the model. The model describes *all* the possible behaviors of the plant without considering what will benefit the final goal of producing the steel. Therefore 'strange' behaviors like a ladle moving back and forth between two machines are also possible, making the reachable state-space very large. By adding extra variables to the model and extra guards over these variables it is possible to decrease the size of the reachable state-space drastically. In this way it is possible to rule out a number of possible behaviors which does not seem to help in achieving the goal, thereby guiding the search towards a goal state. This will most likely also rule out some possible solutions but as long as a solution is found this does not constitute a problem. An important property of adding guides in this way is that any trace obtained in the guided model is also a trace in the unguided models, because no new behaviors are made possible, only existing ones are ruled out. By adding different guides we were able to find a schedule for a model with 60 ladles, compared to only two ladles without guides.

As mentioned earlier, the trace generated by UPPAAL defines what actions take place and what delays are taken. Therefore, one can generate a control program for the plant based on the trace. Since the program is based on one trace it will be a program without branching and without feedback. We were not able to run programs in the SIDMAR plant, therefore we decided to build a LEGO plant inspired by the SIDMAR plant. In particular, the LEGO plant had the same topology as the SIDMAR plant. The timed automaton model was modified to model the LEGO plant. Based on the traces obtained from the modified model we generated control programs running in the RCX™ bricks controlling the plant. Running the programs in the LEGO plant allowed us to find three mistakes in the timed automata model of the plant. After these had been corrected we were able to generate new traces and programs which worked as intended. Not only did we find some errors in the timed automata model using the LEGO plant, our confidence in the model increased by doing the experiments. Understanding the generated traces was much easier when seeing them 'executed' in the LEGO plant.

The guiding techniques described can be applied in general to models where part of the state-space is considered uninteresting. This could be because some properties are only interesting for part of the state-space. Even if it is possible to generate the complete state-space, it might be a good idea initially to search only the parts of the state-space which are considered critical for the property checked. The next section will discuss the use of guides in general model checking further.

Adding guides to the model might rule out the optimal schedule, but we are only interested in whether it is possible to schedule the production or not. In the next two sections and in Chapters 9 and 10 we will consider the problem of finding optimal schedules.

The SIDMAR plant has been studied by a number of other people. Our timed automata model is based on the timed automata model in [64]. The model in [64] is a little more abstract than ours with respect to timing, since moving a ladle between machines is considered to take zero time (which was the case in the original description of the plant in [37]). For a model with three ladles a schedule is generated.

A timed Petri net model of the SIDMAR plant has been presented in [38]. However, this model has not been analyzed. In [35] some ideas are given for how to analyze the model by splitting the events into foreground and background events, but these have not been implemented yet.

The problem of finding feasible schedules for the SIDMAR plant is solved using constraint programming in [139]. The modeling is based on a general scheme for modeling of batch productions for solving scheduling problems using constraint programming. On top of the model obtained following the general scheme, some guides are added to the model. With the guides added to the model, it is possible to generate schedules for models with 30 batches which are very close to the optimal schedule.

As part of the VHS project a smaller chemical batch plant [97] has been studied. This has also been analyzed using timed automata in [100] and here optimal schedules have been found. In this model guiding was not needed because the behavior of the plant has a higher degree of determinism than in SIDMAR so the state-space easily could be stored in the memory of a workstation.

3.3 Efficient Implementation of Uniform Cost

In the study of the SIDMAR plant in Chapter 8 we did not consider optimal schedules because searching for an optimal schedule could not be expressed in Uppaal. In Chapter 5 we proved that the minimum-cost reachability problem is decidable for linearly priced timed automata. This was based on an extension of regions which means that it is not suited for an efficient implementation, at least not if one wants to handle large systems. In Chapter 9 we present the model of *uniformly priced timed automata* which is a restricted class of linearly priced automata. In a uniformly priced timed automaton all the locations have the same rate which is either zero or one, and edges can have a cost like in linearly priced timed automata. The restriction is in effect that all locations have the same (integer) rate, since this can be simulated by only having rate zero or one. Though uniformly priced timed automata are less expressive that linearly priced timed automata they still capture a number of interesting problems such as minimum-time reachability. Therefore we can model a number of optimization problems e.g. the SIDMAR plant and job shop scheduling problems using uniformly priced timed automata.

In Chapter 9 we show how to represent the state-space of a uniformly priced timed automaton using DBMs which have been implemented in UPPAAL. We present a number of techniques for limiting the part of the state-space which needs to be searched to find the optimal cost of reaching a goal state. These have also been implemented in UPPAAL and through a number of examples we show the effectiveness of these techniques.

Uniformly priced timed automata enjoy the important property that it is possible to represent the state-space of such using DBMs with very few changes. This enables us to use the existing UPPAAL implementation with some modifications. We will consider separately the case when the rate is zero and when it is one. When the rate is zero only edges have cost. Therefore, all concrete states represented by a symbolic state have the same cost. This can be represented by adding an integer to a symbolic state representing the cost. The operations on symbolic states are easily extended to handle this case. When the rate is one we need to 'measure' how much time has passed since the execution started. Therefore we extend the DBMs with an extra clock measuring the elapsed time. It is only necessary to handle the new clock in a special way when comparing DBMs. We want to measure the minimum time, therefore we can forget the upper bound on the new clock. This also solves the problem of the inclusion check. A new operation is needed when the cost of an edge should be added, which basically increases the lower bound on the new clock by the cost of the edge. These two cases have been implemented in UPPAAL and the algorithm has been changed such that it does not stop when the first goal state has been found (since we are interested in the optimal cost).

Naively, we need to search the complete state-space to find the optimal cost of reaching a goal state. We present a new search strategy which always searches the state with the smallest cost first and show that in some sense this is optimal. However, this does not give a strategy for choosing between several states with the same cost. Our experiments show that the number of states searched can change drastically, depending on the order in which states with the same cost are searched. We also present a number of methods inspired by branch-and-bound algorithms [21] for reducing the part of the state-space which needs to be searched.

Since the order in which states are searched can drastically influence the number of states searched, we have added the possibility for the user to guide the search. This can be done by adding priorities to the model which are used when states are selected from WAITING. Priorities can be used in models both with and without cost. Analyzing the Biphase Mark Protocol we show that guiding the search towards an error can speedup the finding of errors. The optimal search order defined for finding minimum cost is a kind of breadth-first search order always choosing the state with the smallest cost first. For models with very large state-spaces, like the model of the SIDMAR plant, this approach cannot be used. Therefore the possibility of using guides enables the user to get results without ruling out the optimal solutions, as it is the case with the approach in Chapter 8. Also, when a solution has been found, it is possible to improve the schedule found, by letting the search continue. In Chapter 9 other case studies are presented to demonstrate the usefulness of the model of uniformly priced timed automata and the new features added to UPPAAL.

As mentioned the minimum-time reachability problem is solved [126] by generating the complete state-space and doing analysis on it. In [126] an extra clock is added when searching the already generated state-space for the minimum time of reaching a state, whereas we let this be part of the state-space. Otherwise the state-space is represented in the same way. Therefore we would expect our approach to be faster since the complete state-space is not always generated.

We have applied UPPAAL to a number of known job shop problems. Since UPPAAL is designed for doing timed model checking, we cannot hope that it will perform quite as good as tools dedicated for solving job shop problems, and indeed this is so. However, it does find optimal solutions for a number of job shop problems very fast. The advantage of UPPAAL in this context is mostly that modeling the problems using timed automata is straightforward and intuitive. In many cases the timed automata models have clear similarities with the real problem as in the case of SIDMAR, where the topology of the plant is reflected directly in the timed automata model. Tools like UPPAAL are also suited for checking whether the model is correct or not, by checking various properties and by doing simulations. Therefore, for the problems which can be handled by UPPAAL sufficiently fast, it provides a nice and flexible interface in which it is easy to model the problems and maintain the models.

The difference between the guiding presented in Chapter 8 and 9 is that in Chapter 8 the reachable state-space is changed, while in Chapter 9 the order in which the state-space is searched is changed. This means that in Chapter 8 an optimal solution might be removed from the reachable state-space while in Chapter 9 it will always be searched (if we let the search continue sufficiently long and enough memory is available). When defining the priority of a state in Chapter 9 one can use the cost of the state, which can be very useful. The cost cannot be used in guards since this will make the reachability question undecidable as mentioned earlier. Applying the techniques in Chapter 8 did not require modifications to UPPAAL and can easily be used in other model checkers. Using priorities as in Chapter 9 requires a tool supporting this. We implemented this in a prototype of UPPAAL but it is not included in any released versions yet.

As shown in the case of the Biphase Mark Protocol, guiding can also be useful when searching for an error state without any notion of cost. This is not the first case where using different search orders than breadth-first or depth-first normally offered by tools, have proven useful. In [132] an automatic way of guiding was presented for untimed systems. A general heuristic for the number of transitions to an error state was presented and a BDD based state-space exploration algorithm was presented, searching the states with the shortest path to an error state first.

3.4 Efficient Implementation of Linear Cost

The general cost model of linearly priced timed automata was presented in Chapter 5 and using an extension of regions we showed that for this model the minimum-cost reachability problem is decidable. However, since an implementation based on regions cannot be very efficient, we try to represent the state-space using zones in the standard way represented by DBMs. For the simpler case of UPTAs we have seen that this is possible, however, the same approach cannot be used for LPTAs. This is because the cost of delaying can be different in different locations. As a result of this the cost of the states represented by a zone arising from a delay action, is not necessarily linear (see Section 10.4).

In Chapter 10 we introduce the notion of *facets* and use these to split a zone into smaller zones such that in each zone the, cost of the states represented by the zone can be calculated by a linear expression. The *priced zones* arising when splitting with respect to facets can be represented by triples including a standard DBM, the cost of one given clock valuation, called Δ, in the zone, and a cost rate for each clock. A version of UPPAAL handling LPTAs have been implemented and tested on a number of optimization problems. We also compare this version of UPPAAL to the version restricted to UPTAs.

Intuitively, a facet is the border line or border plane of a zone. When making a delay or a reset of a zone, the zone is split based on different facets of the zone and what can be reached from the facets. This ensures that the cost of the states represented by the zone can be calculated based on a linear expression. The clock valuation Δ is the the valuation with the point wise smallest clock values, which can easily be computed given a DBM representation of a zone. The cost of reaching a state with clock valuation u in a priced zone can be calculated as $c + \sum_{x \in \mathbb{C}} r_x(u(x) - \Delta(x))$ where r_x is the rate of the clock x for the priced zone.

A version of UPPAAL splitting zones with respect to facets has been implemented and tested. When delaying and resetting zones may need to be split based on the facets. The inclusion check between two priced zones can be reduced to an LP problem. For a given priced zone the minimum cost of reaching that zone can also be calculated by solving an LP problem. We have used a know LP solver for this. The techniques presented in Chapter 9 for optimizing the search order and limiting the state-space have also been implemented in this verison of UPPAAL.

We have tried the new prototype of UPPAAL on different instances of the aircraft landing problem from [25] with promising results. For comparing this version of UPPAAL with the version handling UPTAs we look at the bridge problem also addressed in Chapter 9. The version LPTA version is 15% slower than the UPTA. Since we split zones with respect to facets, the zones we end up representing by DBMs are smaller and therefore more zones needs to be represented. We can therefore not hope that this method will be quite as efficient as when standard zones are used.

The work in Chapter 10 relates to other works in the same way as the work in Chapter 9.

3.5 Parametric Analysis

The main purpose of timed model checking is to check whether the timing constants of a given model are correct. What we are really interested in, is in most cases to find some timing parameters which makes the model behave correctly. This can be achieved if we, given a timed automaton with some *parameters*, are able to synthesize some or all values for the parameters making the model behave correctly. We will call this parametric model checking. This problem is addressed in [15] where it is shown to be undecidable for systems with three clocks or more. A semi-decision procedure is also presented in [15].

The model of timed automata is extended to the model of *parametric timed automata* by adding a set of parameters. Guards in parametric timed automata can be on the form $x \bowtie e$ or $x - y \bowtie e$ where e is a linear expression over the set of parameters. The parameters cannot be used in resets.

In Chapter 11 we present an extension of DBMs, parametric DBMs (PDBMs), used for symbolic representation of the state-space of parametric timed automata. We extend UPPAAL with PDBMs and prove that this approach to parametric model checking is correct. Since the problem is undecidable, obviously, termination is not guaranteed. For deciding which of two linear expressions is the smallest one we have borrowed an LP solver from the PMC tool [24]. Parametric versions of the rootcontention protocol and the bounded retransmission protocol have been analyzed using the implementation and minor errors in a published paper on the bounded retransmission protocol has been discovered. For a special class of parametric timed automata called *lower bound/upper bound* automata the parametric model checking problem is shown to be decidable. For this class of automata the number of parameters can in many cases be reduced to one or zero, making the analysis much faster.

The extension of DBMs to parametric DBMs is natural since parametric timed automata have linear expressions in the guards and not only integers. Normally, the entry (i, j) of a DBM representing a zone is a pair, $(n, \bowtie)$, where n is an integer or infinity and $\bowtie \in \{<, \leq\}$. This represents the inequality $x_i - x_j \bowtie n$ where $x_i, x_j \in \mathbb{C}$. All clock valuations in the zone satisfies $v(x_i) - v(x_j) \bowtie n$. In PDBMs the integer is replaced with a linear expression over the parameters.

All the operations on DBMs are based on adding entries and comparing two entries to find the smallest one. Without knowing anything about the values of the parameters, we can in general not compare linear expressions over the parameters to each other or to integers. Comparing a parameter p to the constant 3 has two possible outcomes depending on the value of p. When such comparisons arise we will have to distinguish the two possibilities. We will do this by adding a *constraint set* to a PDBM, consisting of constraints of the form $e \bowtie e'$ where e and e' are linear expressions and $\bowtie \in \{<, \leq, >, \geq\}$. In the example earlier we will then split into two cases; one where the constraint $p < 3$ is added to the constraint set and one where $p \geq 3$ is added to the constraint set. We can now compare entries of PDBMs based on their constraint sets, and add to the constraint sets if needed.

This is the only change needed to the existing operations on DBMs and we can use the standard algorithm for state-space exploration. Symbolic states consist of a location, a PDBM, and a constraint set. The constraints on the parameters needed for a goal (or error) state to be reachable, are the constraints in the constraint set of the goal (or error) state when it is reached. If we want to find all the possible values for the parameters, we need to search the complete state-space to find all the different constraint sets making a goal state reachable.

Other tools for parametric model checking exist. Parametric DBMs were independently presented in [20] but without proof of the correctness of the representation. A tool using PDBMs is presented in [20] which also has a method for guessing and verifying the effect of loops in the automaton. This leads to non-linear expressions which can also be used in the PDBMs. The Omega [46] tool is used for deciding the non-linear relations. The use of non-linear constraints makes the tool much slower than our extension of Uppaal, but also enables it to terminate more often. How the use of guessing the effect of loops and the use of non linear constraints effect the class of parametric timed automata which can be analyzed successfully, is not discussed in the paper.

The PMC tool [24] is another tool capable of doing parametric model checking. This is not based on a forward reachability algorithm but on partition refinement [116] (also called minimization in [11]). It has been used to analyze the rootcontention protocol [24] but with fewer parameters that the analysis we have made, so comparing performance is not possible in this case. However, the PMC tool seems to be very efficient also for doing non parametric timed model checking. The HyTech [74] model checker for hybrid systems can also be used for parametric model checking by letting variables have slope zero always.

Parametric analysis has also been performed by hand sometimes supported by timed model checkers in a number of cases, especially for communication protocols. A few such examples are the bounded retransmission protocol [57], the rootcontention protocol [136, 140], and the Biphase Mark protocol [141, 90].

Chapter 4

Development Methods

As mentioned earlier there are two main ways of applying formal methods. Either one makes a formal model of the system and verifies that the model satisfies a given specification, or one derives the system from a specification. In the following we present one example of each of these two approaches.

4.1 Automatic Model Generation

Applying model checkers to real systems can be hard for a number of reasons, e.g. making a correct model of the system, specifying the properties correctly, and understanding the output of the model checker correctly. In Chapter 12 we will address parts of the problem of making correct models of the system, by presenting an automatic translation from LEGO RCX™ programs to networks of timed automata. This provides an automatic way of obtaining part of the model of a system. Still, the environment controlled by the RCX™ program must be modeled to obtain a full model of the system. However, it is in many cases possible to check properties of the program with very basic models of the environment.

In Chapter 12 we present a compositional translation from a subset of the RCX™ language to networks of timed automata. The part of the language which is considered contains the most important constructions which are needed during our experiments. The model also includes a model of the scheduler, taking care of scheduling the different tasks. The translation has been implemented in a prototype tool and is tested on a control program for a car. The control program is analyzed with respect to some response properties using test automata.

There are several reasons for choosing the RCX™ language as the language to model. First of all, it is quite simple and yet powerful enough for implementing complex control programs. The RCX™ language does not support indirect addressing which makes the translation and the model considerably simpler. A program consists of a number of *tasks* which are executing concurrently. This can naturally be modeled by one automaton for each task. The processor in the RCX™ brick does not pipeline instructions or guess the next type of instruction, which makes the execution time of each instruction independent of the context, making timing analysis much simpler.

A number of other people has been looking at modeling real-time programs. In [92] LEGO RCX™ programs are also modeled in networks of timed automata. Here the model of the scheduler is distributed to a number of smaller automata which makes it simpler. The model in [92] also models timers which are used in an example of a LEGO brick sorter. This increases the size of the state-space because time is split into small portions, making the symbolic states small.

Other languages have also been analyzed, especially Ada. A translation from Ada tasks to timed automata is given in [34] where also suggestions for a translation from timed automata to Ada tasks is presented. A special kind of hybrid automata is used for modeling Ada programs in [54]. For each control path leading to an accepting state, a linear programming problem is solved to find the minimum time of reaching that state. Often, models obtained from automatic modeling of program are very detailed and therefore intractable to analyze. Since the models generated automatically tend to be large, applying abstraction by hand can be very hard and time consuming. The problem of generating models which are sufficiently abstract for analysis is addressed in [55].

4.2 Synthesis using Mona

The main topic of this dissertation is verifying existing code and designs of real-time system. In Chapter 13 we address the problem of synthesizing programs from a specification which does not consider real-time aspects. We have again used the LEGO RCX™ language which enabled us to build a model for testing the generated program.

In Chapter 13 we study how to mix handwritten code and synthesized code through an example. We consider the control program of a crane which can move and turn. Different constraints are imposed on the behavior of the crane. A simple control program which at all times allow any input and responds with the related action of either turning on or off a motor is written by hand. The constraints are imposed by presenting a specification in Monadic Second Order Logic (M2L) [45]. Using Mona [96] the finite automaton accepting the language of the specification is generated. The automaton is translated into RCX™ code by hand, however, the translation is standard and could easily be implemented. With the automaton running together with the handwritten code, the possible executions of the handwritten code can be restricted to the ones satisfying the specification. Finally, it is discussed what to do in general when an action is not allowed and suggestions for extending the approach with time is given.

We use M2L as our specification language, and use the tool Mona to generate automata from the specifications. Mona implements a translation from M2L to minimal finite automata. Instead of specifying the complete control program in M2L we have taken another approach where part of the control program is supplied by the user (written by hand) and another part is synthesized from the M2L specification. The actions in the specification are input and output actions of the program which allows the specification to determine which sequences of

I/O actions should be allowed. Since the M2L formula is translated into an automaton with the I/O actions as alphabet, running alongside the handwritten control program, it can only restrict the behavior of the control program, not add new behaviors. This allows the user to write complex control programs and just use the specification to ensure that certain safety properties are met by the program. One can also write a very simple control program basically allowing any kind of behavior and then write a complex specification. In general some dependencies will be very natural to build into the control program, while more complex constraints are imposed by the specification. The desired response to the control program trying to generate an illegal sequence of I/O actions can vary from application to application. Basically, one can choose to delay the illegal action until other actions have made it legal, or one can choose simply to skip the action.

In the example in Chapter 13 the constraints imposed a priority between different actions. In our case, the control program did not implement any kind of priority between the different actions, so the priorities in the specification did not cause any problems. However, if the control program implements a priority scheme it is up to the user to ensure that deadlock cannot occur when mixing the two priority schemes.

The major drawback of using Mona is that the automata generated do not have acceptance conditions for infinite executions. This makes specifying liveness conditions difficult.

Traditionally, synthesis was intended for generating complete systems from a specification [118] and not a component as presented in Chapter 13. Since simple parts of the code are easier to write by hand than to specify, we think it makes sense to combine handwritten code with synthesized code. Synthesis based on automata has been studied widely, and [131] gives an overview of the area. In [113] a game-theoretic approach to controller synthesis is presented for infinite games. This has been studied in the case of real-time in [23]. Also controllers for hybrid systems have been studied by a number of people. In [114] control software is synthesized from hybrid automata. The synthesized code can also be mixed with handwritten code, but here the handwritten code is included as modules, implementing certain tasks which are used by the synthesized code.

Part II

Region Based Methods

Chapter 5

Decidability of Minimum-Cost Reachability

The paper *Minimum-Cost Reachability for Priced Timed Automata* presented in this chapter has been published as a technical report [29] and a conference paper [28].

[28] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn and F. Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. In *Proceedings of Hybrid Systems: Computation and Control*, pages 147–161, 2001.

[29] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn and F. Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. Technical Report RS-01-03, BRICS, January 2001.

The technical report extends the conference version by adding complete proofs and some more examples. Except for minor typographical changes the content of this chapter is equal to the technical report [29].

Minimum-Cost Reachability for Priced Timed Automata[¶]

Gerd Behrmann[*] Ansgar Fehnker[‡][‖] Thomas Hune[†]
Kim Larsen[*] Judi Romijn[‡] Frits Vaandrager[‡]
Paul Pettersson[§][**]

Abstract

This paper introduces the model of *linearly priced timed automata* as an extension of timed automata, with prices on both transitions and locations. For this model we consider the minimum-cost reachability problem: i.e. given a linearly priced timed automaton and a target state, determine the minimum cost of executions from the initial state to the target state. This problem generalizes the minimum-time reachability problem for ordinary timed automata. We prove decidability of this problem by offering an algorithmic solution, which is based on a combination of branch-and-bound techniques and a new notion of priced regions. The latter allows symbolic representation and manipulation of reachable states together with the cost of reaching them.
Keywords: Timed Automata, Verification, Data Structures, Algorithms, Optimization.

5.1 Introduction

Recently, real-time verification tools such as Uppaal [109], Kronos [41] and HyTech [74], have been applied to synthesize feasible solutions to static job-shop scheduling problems [64, 82, 127]. The basic common idea of these works is to reformulate the static scheduling problem as a reachability problem that can be solved by the verification tools. In this approach, the timed automata [13]

[*]**B**asic **R**esearch **I**n **C**omputer **S**cience, BRICS, (www.brics.dk), funded by the Danish National Research Foundation, University of Aalborg, Denmark, Email: `[behrmann,kgl]@cs.auc.dk`

[†]**B**asic **R**esearch **I**n **C**omputer **S**cience, BRICS, (www.brics.dk), funded by the Danish National Research Foundation, University of Århus, Denmark, Email: `baris@brics.dk`

[‡]Computing Science Institute, University of Nijmegen, Email: `[ansgar,judi,fvaan]@cs.kun.nl`

[§]Department of Information Technology, Uppsala University, Email: `paupet@docs.uu.se`.

[¶]This work is partially supported by the European Community Esprit-LTR Project 26270 VHS (Verification of Hybrid Systems)

[‖]Research supported by Netherlands Organization for Scientific Research (NWO) under contract SION 612-14-004.

[**]Research partly sponsored by the AIT-WOODDES Project No IST-1999-10069.

based modeling languages of the verification tools serve as the basic input language in which the scheduling problem is described. These modeling languages have proven particularly well-suited in this respect, as they allow for easy and flexible modeling of systems, consisting of several parallel components that interact in a time-critical manner and constrain each other's behavior in a multitude of ways.

In this paper we introduce the model of *linearly priced timed automata* and offer an algorithmic solution to the problem of determining the minimum cost of reaching a designated set of target states. This result generalizes previous results on computation of minimum-time reachability and accumulated delays in timed automata, and should be viewed as laying a theoretical foundation for algorithmic treatments of more general optimization problems as encountered in static scheduling problems.

As an example consider the very simple static scheduling problem represented by the timed automaton in Fig. 5.1 from [126], which contains 5 'tasks' $\{A, B, C, D, E\}$. All tasks are to be performed precisely once, except task C, which should be performed *at least* once. The order of the tasks is given by the timed automaton, e.g. task B must not commence before task A has finished. In addition, the timed automaton specifies three timing requirements to be satisfied: the delay between the start of the first execution of task C and the start of the execution of E should be at least 3 time units; the delay between the start of the last execution of C and the start of D should be no more than 1 time unit; and, the delay between the start of B and the start of D should be at least 2 time units, each of these requirements are represented by a clock in the model. Using a standard timed model checker we are able to verify that

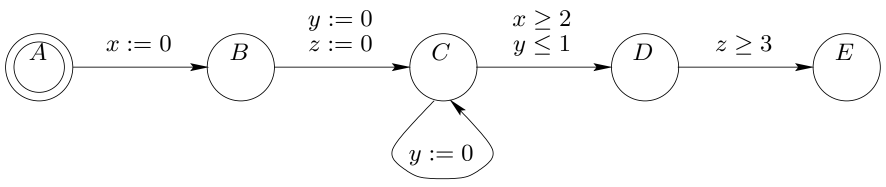


Figure 5.1: Timed automata model of scheduling example.

location E of the timed automaton is reachable. This can be demonstrated by a trace leading to the location[1]:

$$(A, 0, 0, 0) \xrightarrow{\tau} \xrightarrow{\epsilon(1)} (B, 1, 1, 1) \xrightarrow{\tau} \xrightarrow{\epsilon(1)} (C, 2, 1, 1) \xrightarrow{\tau} \xrightarrow{\epsilon(2)} (D, 4, 3, 3) \xrightarrow{\tau} (E, 4, 3, 3) \tag{5.1}$$

The above trace may be viewed as a feasible solution to the original static scheduling problem. However, given an optimization problem, one is often not satisfied with an arbitrary feasible solution but insist on solutions which are *optimal* in some sense. When modeling a problem like this one using timed automata an obvious notion of optimality is that of minimum accumulated time. For the timed automaton of Fig. 5.1 the trace of (5.1) has an accumulated

[1]Here a quadruple (X, v_x, v_y, v_z) denotes the state of the automaton in which the control location is X and where v_x, v_y and v_z give the values of the three clocks x, y and z. The transitions labelled τ are actual transitions in the model, and the transitions labelled $\epsilon(d)$ represents a delay of d time units.

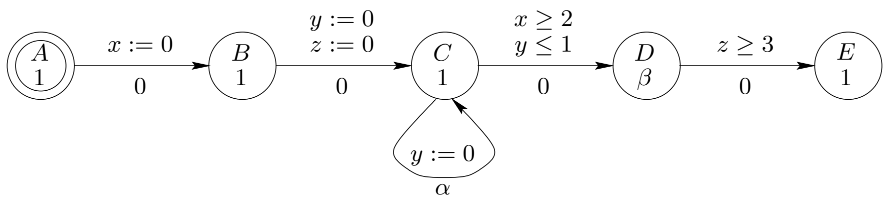


Figure 5.2: A linearly priced timed automaton.

time-duration of 4. This, however, is not optimal as witnessed by the following alternative trace, which by exploiting the looping transition on C reaches E within a total of 3 time-units[2]:

$$(A,0,0,0) \xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\epsilon(2)} (C,2,2,2) \xrightarrow{\tau} (C,2,0,2) \xrightarrow{\tau}\xrightarrow{\epsilon(1)} (D,3,1,3) \xrightarrow{\tau} (E,3,1,3) \quad (5.2)$$

In [23] algorithmic solutions to the minimum-time reachability problem and the more general problem of controller synthesis has been given using a backward fix-point computation. In [126] an alternative solution based on forward reachability analysis is given, and in [26] an algorithmic solution is offered, which applies branch-and-bound techniques to prune parts of the symbolic state-space which are guaranteed not to contain optimal solutions. In particular, by introducing an additional clock for accumulating time-elapses, the minimum-time reachability problem may be dealt with using the existing efficient data structures (e.g. DBMs [60], CDDs [110] and DDDs [123]) already used in the real-time verification tools Uppaal and Kronos for reachability. The results of the present paper also extends the work in [10] which provides an algorithm for computing the accumulated delay in a timed automata.

In this paper, we provide the basis for dealing with more general optimization problems. In particular, we introduce the model of *linearly priced timed automata*, as an extension of timed automata with *prices* on both transitions and locations: the price of a transition gives the cost for taking it and the price on a location specifies the cost *per time-unit* for staying in that location. This model can capture not only the passage of time, but also the way that e.g. tasks with different prices for use per time unit, contributes to the total cost. Figure 5.2 gives a linearly priced extension of the timed automaton from Fig. 5.1. Here, the price of location D is set to β and the price on all other locations is set to 1 (thus simply accumulating time). The price of the looping transition on C is set to α, whereas all other transitions are free of cost (price 0). Now for $(\alpha, \beta) = (1, 3)$ the costs of the traces (1) and (2) are 8 and 6, respectively (thus it is cheaper to actually exploit the looping transition). For $(\alpha, \beta) = (2, 2)$ the costs of the two traces are both 6, thus in this case it is immaterial whether the looping transition is taken or not. In fact, the optimal cost of reaching E will in general be the minimum of $2 + 2 * \beta$ and $3 + \alpha$, and the optimal trace will include the looping transition on C depending on the particular values of α and β.

[2]In fact, 3 is the minimum time for reaching E.

In this paper we deal with the problem of determining the minimum cost of reaching a given location for linearly priced timed automata. In particular, we offer an algorithmic solution to this problem[3]. In contrast to minimum-time reachability for timed automata, the minimum-cost reachability problem for linearly priced timed automata requires the development of new data structures for symbolic representation and the manipulation of reachable *sets* of states *together with* the cost of reaching them. In this paper we put forward one such data structure, namely a priced extension of the fundamental notion of *clock regions* for timed automata [13].

The remainder of the paper is structured as follows: Section 5.2 formally introduces the model of linearly priced timed automata together with its semantics. Section 5.3 develops the notion of priced clock regions, together with a number of useful operations on these. The priced clock regions are then used in Section 5.4 to give a symbolic semantics capturing (sufficiently) precisely the cost of executions and used as a basis for an algorithm solution to the minimum-cost problem. Finally, in Section 5.5 we give some concluding remarks.

5.2 Linearly Priced Timed Automata

In this section, we introduce the model of linearly priced timed automata, which is an extension of timed automata [13] with prices on both locations and transitions. Dually, linearly priced timed automata may be seen as a special type of linear hybrid automata [72] or multirectangular automata [72] in which the accumulation of prices (i.e. the cost) is represented by a single continuous variable. However, in contrast to known undecidability results for these classes, minimum-cost reachability is computable for linearly priced timed automata. An intuitive explanation for this is that the additional (cost) variable does not influence the behavior of the automata.

Let C be a finite set of clocks. Then $\mathcal{B}(C)$ is the set of formulas obtained as conjunctions of atomic constraints of the form $x \bowtie n$ where $x \in C$, n is natural number, and $\bowtie \in \{<, \leq, =, \geq, >\}$. Elements of $\mathcal{B}(C)$ are called *clock constraints* over C. Note that for each timed automaton that has constraints of the form $x - y \bowtie c$, there exists a strongly bisimilar timed automaton with only constraints of the form $x \bowtie c$. Therefore, the results in this paper are applicable to automata having constraints of the type $x - y \bowtie c$ as well.

Definition 5.1 (Linearly Priced Timed Automaton) *A Linearly Priced Timed Automaton (LPTA) over clock set C and action set Act is a tuple (L, l_0, E, I, P) where L is a finite set of locations, l_0 is the initial location, $E \subseteq L \times \mathcal{B}(C) \times Act \times \mathcal{P}(C) \times L$ is the set of edges, $I : L \to \mathcal{B}(C)$ assigns invariants to locations, and $P : (L \cup E) \to \mathsf{N}$ assigns prices to both locations and edges. In the case of $(l, g, a, r, l') \in E$, we write $l \xrightarrow{g,a,r} l'$.*

Formally, clock values are represented as functions called *clock assignments* from C to the non-negative reals $\mathsf{R}_{\geq 0}$. We denote by R^C the set of clock assign-

[3]Thus settling an open problem given in [23].

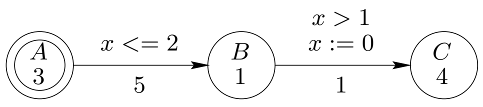


Figure 5.3: An example LPTA.

ments for C ranged over by u, u' etc. We define the operation $u' = [r \mapsto 0]u$ to be the assignment such that $u'(x) = 0$ if $x \in r$ and $u(x)$ otherwise, and the operation $u' = u + d$ to be the assignment such that $u'(x) = u(x) + d$. Also, a clock valuation u satisfies a clock constraint g, $u \in g$, if $u(x) \bowtie n$ for all atomic constraints $x \bowtie n$ in g. Notice that the set of clock valuations satisfying a guard is always a convex set.

The semantics of an LPTA A is defined as a transition system with states $L \times \mathsf{R}^C$, with initial state (l_0, u_0) (where u_0 assigns zero to all clocks in C), and with the following transition relation:

- $(l, u) \xrightarrow{\epsilon(d),p} (l, u + d)$ if $u + d \in I(l)$, and $p = P(l) * d$.
- $(l, u) \xrightarrow{a,p} (l', u')$ if there exists g, r such that $l \xrightarrow{g,a,r} l'$, $u \in g$, $u' = [r \mapsto 0]u$, $u' \in I(l')$ and $p = P((l, g, a, r, l'))$.

Note that the transitions are decorated with two labels: a delay-quantity or an action, together with the cost of the particular transition. For determining the cost, the price of a location gives the cost rate of staying in that location (per time unit), and the price of a transition gives the cost of taking that transition. In the remainder, states and executions of the transition system for LPTA A will be referred to as states and executions of A.

Definition 5.2 (Cost) *Let $\alpha = (l_0, u_0) \xrightarrow{a_1,p_1} (l_1, u_1) \dots \xrightarrow{a_n,p_n} (l_n, u_n)$ be a finite execution of LPTA A. The* cost *of α, $\mathsf{cost}(\alpha)$, is the sum $\Sigma_{i\in\{1,\dots,n\}} p_i$.*

For a given state (l, u), the minimal cost of reaching (l, u), $\mathsf{mincost}((l, u))$, is the infimum of the costs of finite executions ending in (l, u). Similarly, the minimal cost of reaching a location l is the infimum of the costs of finite executions ending in a state of the form (l, u).

$$\mathsf{mincost}(l) = \inf\{\mathsf{cost}(\alpha) \mid \alpha \text{ a run ending in location } l\}$$

Example 5.1 *Consider the LPTA of Fig. 5.3. The LPTA has a single clock x, and the locations and transitions are decorated with guards and prices. A sample execution of this LPTA is for instance:*

$$(A, 0) \xrightarrow{\epsilon(1.5),4.5} (A, 1.5) \xrightarrow{\tau,5} (B, 1.5) \xrightarrow{\tau,1} (C, 1.5)$$

The cost of this execution is 10.5. In fact, there are executions with cost arbitrarily close to the value 7, obtainable by avoiding delaying in location A, and delaying just long enough in location B. Due to the infimum definition of $\mathsf{mincost}$*, it follows that* $\mathsf{mincost}(C) = 7$*. However, note that because of the strict comparison in the guard of the second transition, no execution actually achieves this cost.* □

5.3 Priced Clock Regions

For ordinary timed automata, the key to decidability results has been the valuable notion of *region* [13]. In particular, regions provide a finite partitioning of the uncountable set of clock valuations, which is also stable with respect to the various operations needed for exploration of the behavior of timed automata (in particular the operations of delay and reset).

In the setting of linearly priced timed automata, we put forward a new extended notion of *priced region.* Besides providing a finite partitioning of the set of clock-valuations (as in the case of ordinary regions), priced regions also associate costs to each individual clock-valuation within the region. However, as we shall see in the following, priced regions may be presented and manipulated in a symbolic manner and are thus suitable as an algorithmic basis.

Definition 5.3 (Priced Regions) *Given set S, let $Seq(S)$ be the set of finite sequences of elements of S. A* priced clock region *over a finite set of clocks C*

$$R = (h, [r_0, \ldots, r_k], [c_0, \ldots, c_l])$$

is an element of $(C \to \mathsf{N}) \times Seq(2^C) \times Seq(\mathsf{N})$, with $k = l$, $C = \cup_{i \in \{0,\ldots,k\}} r_i$, $r_i \cap r_j = \emptyset$ when $i \neq j$, and $i > 0$ implies that $r_i \neq \emptyset$.

Given a clock valuation $u \in \mathsf{R}^C$, and region $R = (h, [r_0, \ldots, r_k], [c_0, \ldots, c_k])$, $u \in R$ iff

1. *h and u agree on the integer part of each clock in C,*
2. *$x \in r_0$ iff $\mathsf{frac}(u(x)) = 0$,*
3. *$x, y \in r_i \Rightarrow \mathsf{frac}(u(x)) = \mathsf{frac}(u(y))$, and*
4. *$x \in r_i$, $y \in r_j$ and $i < j \Rightarrow \mathsf{frac}(u(x)) < \mathsf{frac}(u(y))$.*

For a priced region $R = (h, [r_0, \ldots, r_k], [c_0, \ldots, c_k])$ the first two components of the triple constitute an ordinary (unpriced) region $\hat{R} = (h, [r_0, \ldots, r_k])$. The naturals $c_0, \ldots, c_k$ are the costs, which are associated with the vertices of the closure of the (unpriced) region, as follows. We start in the left-most lower vertex of the exterior of the region and associate cost c_0 with it, then move one time unit in the direction of set r_k to the next vertex of the exterior, and associate cost c_1 with that vertex, then move one unit in the direction of r_{k-1}, etc. In this way, the costs $c_0, \ldots, c_k$, span a linear cost plane on the k-dimensional unpriced region.

The closure of the unpriced region R is the convex hull of the vertices. Each clock valuation $u \in R$ is a (unique) convex combination[4] of the vertices. Therefore the cost of u can be defined as the same convex combination of the cost in the vertices. This gives the following definition:

[4]A linear expression $\sum a_i v_i$ where $\sum a_i = 1$, and $a_i \geq 0$.

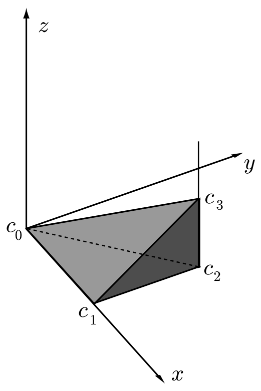


Figure 5.4: A three dimensional priced region.

Definition 5.4 (Cost inside Regions) *Let $R = (h, [r_0, \ldots, r_k], [c_0, \ldots, c_k])$ be a priced region and clock valuation $u \in R$, the* cost *of u in R is defined as:*

$$\mathit{cost}(u, R) = c_0 + \sum_{i=0}^{k-1} \mathit{frac}(u(x_{k-i})) * (c_{i+1} - c_i)$$

where x_j is some clock in r_j. The minimal cost associated with R is $\mathit{mincost}(R) = \min(\{c_0, \ldots, c_k\})$.

In the symbolic state-space, constructed with the priced regions, the costs will be computed such that for each concrete state in a symbolic state, the cost associated with it is the minimal cost for reaching that state by the symbolic path that was followed. In this way, we always have the minimal cost of all concrete paths represented by that symbolic path, but there may be more symbolic paths leading to a symbolic state in which the costs are different. Note that the cost of a clock valuation in the region is computed by adding fractions of costs for equivalence sets of clocks, rather than for each clock.

To prepare for the symbolic semantics, we define in the following a number of operations on priced regions. These operations are also the ones used in the algorithm for finding the optimal cost of reaching a location.

The delay operation computes the time successor, which works exactly as in the classical (unpriced) regions. The changing dimensions of the regions cause the addition or deletion of vertices and thus of the associated cost. The price argument will be instantiated to the price of the location in which time is passing; this is needed only when a vertex is added. The two cases in the operation are illustrated in Fig. 5.5 to the left (5.1) and (5.2).

Definition 5.5 (Delay) *Given a priced region $R = (h, [r_0, \ldots, r_k], [c_0, \ldots, c_k])$ and a price p, the function* delay *is defined as follows:*

1. *If r_0 is not empty, then*

$$\mathit{delay}(R, p) \quad = \quad (h, [\emptyset, r_0, \ldots, r_k], [c_0, \ldots, c_k, c_0 + p])$$

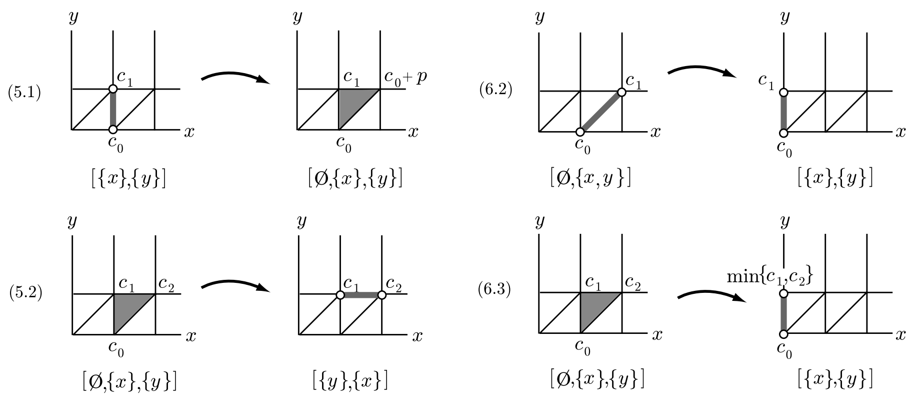


Figure 5.5: Delay and reset operations for two-dimensional priced regions.

2. *If r_0 is empty, then*

$$\begin{aligned} \textsf{delay}(R,p) &= (h',[r_k,r_1,\dots,r_{k-1}],[c_1,\dots,c_k]) \\ \textit{where } h' &= h \textit{ incremented for all clocks in } r_k \end{aligned}$$

When resetting a clock, a priced region may lose a dimension. If so, the two costs, associated with the vertices that are collapsed, are compared and the minimum is taken for the new vertex. Two of the three cases in the operation is illustrated in Fig. 5.5 to the right (6.2) and (6.3).

Definition 5.6 (Reset) *Given a priced region $R = (h, [r_0,\dots,r_k],[c_0,\dots,c_k])$ and a clock $x \in r_i$, the function* reset *is defined as follows:*

1. *If $i = 0$ then $\textsf{reset}(x,R) = (h',[r_0,\dots,r_k],[c_0,\dots,c_k])$, where $h' = h$ with x set to zero*

2. *If $i > 0$ and $r_i \neq \{x\}$, then*

$$\begin{aligned} \textsf{reset}(x,R) &= (h',[r_0\cup\{x\},\dots,r_i\setminus\{x\},\dots,r_k],[c_0,\dots,c_k]) \\ \textit{where } h' &= h \textit{ with } x \textit{ set to zero} \end{aligned}$$

3. *If $i > 0$ and $r_i = \{x\}$, then*

$$\begin{aligned} \textsf{reset}(x,R) &= (h',[r_0\cup\{x\},\dots,r_{i-1},r_{i+1},\dots,r_k], \\ &\qquad [c_0,\dots,c_{k-i-1},c',c_{k-i+2},\dots,c_k]) \\ \textit{where } c' &= \min(c_{k-i},c_{k-i+1}) \\ h' &= h \textit{ with } x \textit{ set to zero} \end{aligned}$$

The reset operation on a set of clocks: $\textsf{reset}(C\cup\{x\},R) = \textsf{reset}(C,\textsf{reset}(x,R))$, and $\textsf{reset}(\emptyset,R) = R$.

The price argument in the increment operation will be instantiated to the price of the particular transition taken; all costs are updated accordingly.

Definition 5.7 (Increment ($\oplus$)) *Given a priced region $R = (h, [r_0, \dots, r_k], [c_0, \dots, c_k])$ and a price p, the increment of R with respect to p is the priced region $R \oplus p = (h, [r_0, \dots, r_k], [c'_0, \dots, c'_k])$ where $c'_i = c_i + p$.*

If in region R, no clock has fractional part 0, then time may pass in R, that is, each clock valuation in R has a time successor and predecessor in R. When changing location with R, we must choose for each clock valuation u in R between delaying in the previous location until u is reached, followed by the change of location, or changing location immediately and delaying to u in the new location. This depends on the price of either location. For this the following operation self is useful.

Definition 5.8 (Self) *Given a priced region $R = (h, [r_0, \dots, r_k], [c_0, \dots, c_k])$ and a price p, the function self is defined as follows:*

1. *If r_0 is not empty, then $\mathsf{self}(R, p) = R$.*
2. *If r_0 is empty, then*

$$\begin{aligned} \mathsf{self}(R, p) &= (h, [r_0, \dots, r_k], [c_0, \dots, c_{k-1}, c']) \\ \textit{where } c' &= \min(c_k, c_0 + p) \end{aligned}$$

Definition 5.9 (Comparison) *Two priced regions may be compared only if their unpriced versions are equal: $(h, [r_0, \dots, r_k], [c_0, \dots, c_k]) \leq (h', [r'_0, \dots, r'_{k'}], [c'_0, \dots, c'_{k'}])$ iff $h = h'$, $k = k'$, and for $0 \leq i \leq k$: $r_i = r'_i$ and $c_i \leq c'_i$.*

The operations delay and self satisfy the following useful properties:

Proposition 5.1 (Interaction Properties)

1. $\mathsf{self}(R, p) \leq R$,
2. $\mathsf{self}(\mathsf{self}(R, p), p) = \mathsf{self}(R, p)$,
3. $\mathsf{delay}(\mathsf{self}(R, p), p) \leq \mathsf{delay}(R, p)$,
4. $\mathsf{self}(\mathsf{delay}(R, p), p) = \mathsf{delay}(R, p)$,
5. $\mathsf{self}(R \oplus q, p) = \mathsf{self}(R, p) \oplus q$,
6. $\mathsf{delay}(R \oplus q, p) = \mathsf{delay}(R, p) \oplus q$,
7. *For $g \in \mathcal{B}(C)$, whenever $R \in g$ then $\mathsf{self}(R, p) \in g$.*

Proof. Directly from the definitions of the operators and $\leq$. □

Stated in terms of the cost, $\mathsf{cost}(u, R)$, of an individual clock valuation, u, of a priced region, R, the symbolic operations behave as follows:

Proposition 5.2 (Cost Relations)

1. *Let $R = (h, [r_0, \dots, r_k], [c_0, \dots, c_k])$. If $u \in R$ and $u + d \in R$ then $\mathsf{cost}(u + d, R) = \mathsf{cost}(u, R) + d * (c_k - c_0)$.*

2. *If* $R = \mathsf{self}(R, p)$*,* $u \in R$ *and* $u + d \in \mathsf{delay}(R, p)$ *then* $\mathsf{cost}(u + d, \mathsf{delay}(R, p)) = \mathsf{cost}(u, R) + d * p$*.*

3. $\mathsf{cost}(u, \mathsf{reset}(x, R)) = \mathsf{inf}\{\, \mathsf{cost}(v, R) \mid [x \mapsto 0]v = u \,\}$*.*

Proof. Directly from the definitions of the operators and cost. □

5.4 Symbolic Semantics and Algorithm

In this section, we provide a symbolic semantics for linearly priced timed automata based on the notion of priced regions and the associated operations presented in the previous section. As a main result we shown that the cost of an execution of the underlying automaton is captured sufficiently accurately. Finally, we present an algorithm based on priced regions.

Definition 5.10 (Symbolic Semantics) *The symbolic semantics of an LPTA* A *is defined as a transition system with the states* $L \times ((C \to \mathsf{N}) \times \mathit{Seq}(2^C) \times \mathit{Seq}(\mathsf{N}))$*, with initial state* $(l_0, (h_0, [C], [0]))$ *(where* h_0 *assigns zero to the integer part of all clocks in* C*), and with the following transition relation:*

- $(l, R) \to (l, \mathsf{delay}(R, P(l)))$ *if* $\mathsf{delay}(R, P(l)) \in I(l)$*.*
- $(l, R) \to (l', R')$ *if there exists* g*,* r *such that* $l \xrightarrow{g,a,r} l'$*,* $R \in g$*,* $R' = \mathsf{reset}(R, r) \oplus P((l, g, a, r, l'))$ *and* $R' \in I(l')$*.*
- $(l, R) \to (l, \mathsf{self}(R, P(l)))$

In the remainder, states and executions of the symbolic transition system for LPTA A will be referred to as the symbolic states and executions of A.

Lemma 5.1 *Given LPTA* A*, for each execution* α *of* A *that ends in state* (l, u)*, there is a symbolic execution* β *of* A*, that ends in symbolic state* (l, R)*, such that* $u \in R$*, and* $\mathsf{cost}(u, R) \leq \mathsf{cost}(\alpha)$*.*

Proof. For this proof we first observe that, given $g \in \mathcal{B}(C)$, if $u \in R$ and $u \in g$, then $R \in g$.

We prove this by induction on the length of α. Suppose α ends in state (l, u). The base step concerns α with length 0, consisting of only the initial state (l_0, u_0) where u_0 is the valuation assigning zero to all clocks. Clearly, $\mathsf{cost}(\alpha) = 0$. Since the initial state of the symbolic semantics is the state $(l_0, (h_0, [C], [0]))$, in which h_0 assigns zero to the integer part of all clocks, and the fractional part of all clocks is zero, we can take β to be $(l_0, (h_0, [C], [0]))$. Clearly, there is only one valuation $u \in (h_0, [C], [0])$, namely the valuation u that assigns zero to all clocks, which is exactly u_0, and by definition, $\mathsf{cost}(u_0, (h_0, [C], [0])) = 0$ and trivially $0 \leq 0$.

For the induction step, assume the following. We have an execution α' in the concrete semantics, ending in (l', u'), a corresponding execution β' in the symbolic semantics, ending in (l', R'), such that $u' \in R'$, and $\mathsf{cost}(u', R') \leq \mathsf{cost}(\alpha')$.

Suppose $(l', u') \xrightarrow{a,p} (l, u)$. Then there is a transition $l' \xrightarrow{a,g,r} l$ in the automaton A such that $u \in g$, $u = [r \mapsto 0]u'$, $u \in I(l)$ and $p = P((l', a, g, r, l))$. Now $u' \in g$ implies that $R' \in g$. Let $R = \mathsf{reset}(R', r) \oplus p$. It is easy to show that $u = [r \mapsto 0]u' \in R$ and as $u \in R$ we then have that $R \in I(l)$. So $(l', R') \to (l, R)$ and

$$\begin{aligned} \mathsf{cost}(u, R) &= \mathsf{inf}\{\, \mathsf{cost}(v, R') \mid [r \mapsto 0]v = u \,\} + p \\ &\leq \mathsf{cost}(u', R') + p \\ &\leq^{\text{IH}} \mathsf{cost}(\alpha') + p \\ &= \mathsf{cost}(\alpha) \end{aligned}$$

Suppose $(l', u') \xrightarrow{\epsilon(d), p*d} (l, u)$, where $p = P(l)$, i.e. $l = l'$, $u = u' + d$, and $u \in I(l)$. Now there exist sequences $R_o, R_1, \ldots, R_m$ and $d_1, \ldots, d_m$ of price regions and delays such that $d = d_1 + \cdots + d_m$, $R_0 = R'$ and for $i \in \{1, \ldots, m\}$, $R_i = \mathsf{delay}(R_{i-1}, p)$ with $u' + \sum_{k=1}^{i} d_k \in R_i$. This defines the sequence of regions wisited without considering cost. To obtain the priced regions with the optimal cost we apply the self operation. Let $R'_0 = \mathsf{self}(R_0, p)$ and for $i \in \{1, \ldots, m\}$ let $R'_i = \mathsf{delay}(R'_{i-1}, p)$ (in fact, for $i \in \{1, \ldots, m\}$, $R'_i = \mathsf{self}(R'_i, p)$ due to Proposition 5.1.4 and $R'_i \leq R_i$). Clearly we have the following symbolic extension of β':

$$\beta' \to (l', R'_0) \to \cdots \to (l', R'_m)$$

Now by Proposition 5.2.2 (the condition of Proposition 5.2.2 is satisfied for all $R'_i (i \geq 0)$ because of Proposition 5.1.4:

$$\begin{aligned} \mathsf{cost}(u' + d, R'_m) &= \mathsf{cost}(u', R'_0) + d * p \\ &\leq \mathsf{cost}(u', R') + d * p \\ &\leq^{\text{IH}} \mathsf{cost}(\alpha') + d * p \\ &= \mathsf{cost}(\alpha) \end{aligned}$$

□

Lemma 5.2 *Whenever (l, R) is a reachable symbolic state and $u \in R$, then $\mathit{mincost}((l, u)) \leq \mathit{cost}(u, R)$.*

Proof. The proof is by induction on the length of the symbolic trace β reaching (l, R). In the base case, the length of β is 0 and $(l, R) = (l_0, R_0)$, where R_0 is the initial price region $(h_0, [C], [0])$ in which h_0 associates 0 with all clocks. Clearly, there is only one valuation $u \in R_0$, namely the valuation which assigns 0 to all clocks. Obviously, $\mathsf{mincost}((l_0, u_0)) = 0 \leq \mathsf{cost}(u_0, R_0) = 0$.

For the induction step, assume that (l, R) is reached by a trace β with length greater than 0. In particular, let (l', R') be the immediate predecessor of (l, R) in β. Let $u \in R$. We consider three cases depending on the type of symbolic transition from (l', R') to (l, R).

Case 1: Suppose $(l', R') \rightarrow (l, R)$ is a symbolic delay transition. That is, $l = l'$, $R = \mathsf{delay}(R', p)$ with $p = P(l)$ and $R \in I(l)$. We consider two sub-cases depending on whether R' is self-delayable or not[5].
Assume that R' is not self-delayable, i.e. $R' = (h', [r'_0, \dots, r'_k], [c'_0, \dots, c'_k])$ with $r'_0 \neq \emptyset$. Then $R = (h', [\emptyset, r'_0, \dots, r'_k], [c'_0, \dots, c'_k, c'_0 + p])$. Let $x \in r'_0$ and let $u' = u - d$ where $d = \mathsf{frac}(u(x))$. Then $u' \in R'$ and $(l', u') \xrightarrow{\epsilon(d), q} (l, u)$ where $q = d{*}p$. Thus $\mathsf{mincost}((l, u)) \leq \mathsf{mincost}((l', u')) + d{*}p$. By induction hypothesis, $\mathsf{mincost}((l', u')) \leq \mathsf{cost}(u', R')$, and as $\mathsf{cost}(u, R) = \mathsf{cost}(u', R) + d * p$, we obtain, as desired, $\mathsf{mincost}((l, u)) \leq \mathsf{cost}(u, R)$.
Assume that R' is self-delayable. That is, $R' = (h', [r'_0, r'_1, \dots, r'_k], [c'_0, \dots, c'_k])$ with $r'_0 = \emptyset$ and $R = (h'', [r'_k, r'_1, \dots, r'_{k-1}], [c'_1, \dots, c'_k])$. Now, let $x \in r'_1$. Then for any $d < \mathsf{frac}(u(x))$ we let $u_d = u - d$. Clearly, $u_d \in R'$ and $(l, u_d) \xrightarrow{\epsilon(d), p*d} (l, u)$. Now,

$$\begin{array}{rcl} \mathsf{mincost}((l,u)) & \leq & \mathsf{mincost}((l,u_d)) + p * d \\ & \leq^{\text{IH}} & \mathsf{cost}(u_d, R') + p * d \end{array}$$

Now $\mathsf{cost}(u, R) = \mathsf{cost}(u_d, R') + (c'_k - c'_0) * d$ so it is clear that $\mathsf{cost}(u_d, R') + k * d \rightarrow \mathsf{cost}(u, R)$ when $d \rightarrow 0$ for any k. In particular, $\mathsf{cost}(u_d, R') + p * d \rightarrow \mathsf{cost}(u, R)$ when $d \rightarrow 0$. Thus $\mathsf{mincost}((l, u)) \leq \mathsf{cost}(u, R)$ as desired.

Case 2: Suppose $(l', R') \rightarrow (l, R)$ is a symbolic action transition. That is $R = \mathsf{reset}(R', r) \oplus p$ for some transition $l' \xrightarrow{g,a,r} l$ of the automaton with $R' \in g$ and $p = P((l', g, a, r, l))$. Now let $v \in R'$ such that $[r \mapsto 0]v = u$. Then clearly $(l', v) \xrightarrow{a,p} (l, u)$. Thus:

$$\begin{array}{rcl} \mathsf{mincost}((l,u)) & \leq & \mathsf{inf}\{\, \mathsf{mincost}((l,v)) \mid v \in R', [r \mapsto 0]v = u \,\} \\ & \leq^{\text{IH}} & \mathsf{inf}\{\, \mathsf{cost}(v, R') \mid [r \mapsto 0]v = u \,\} \\ & = & \mathsf{cost}(u, R) \;\; \text{by Proposition 5.2.3} \end{array}$$

Case 3: Suppose $(l', R') \rightarrow (l, R)$ is a symbolic self-delay transition. Thus, in particular $l = l'$. If $R = R'$ the lemma follows immediately by applying the induction hypothesis to (l', R'). Otherwise, R' is self-delayable and R' and R are identical except for the cost of the 'last' vertex which has been changed by the self operation; i.e. $R' = (h, [r_0, \dots, r_k], [c_0, \dots, c_{k-1}, c_k])$ and $R = (h, [r_0, \dots, r_k], [c_0, \dots, c_{k-1}, c_0 + p])$ with $r_0 = \emptyset$, $c_0 + p < c_k$ and $p = P(l)$. Now let $x \in r_1$. Then for any $d > u(x)$ we let $u_d = u - d$. Clearly, $u_d \in R$ (and $u_d \in R'$) and $(l, u_d) \xrightarrow{\epsilon(d), p*d} (l, u)$. Now:

$$\begin{array}{rcl} \mathsf{mincost}((l,u)) & \leq & \mathsf{mincost}((l,u_d)) + p * d \\ & \leq^{\text{IH}} & \mathsf{cost}(u_d, R') + p * d \end{array}$$

Now let $R'' = (h, [r_1, \dots, r_k], [c_0, \dots, c_{k-1}])$. Then $R = \mathsf{delay}(R'', p)$ and $R' = \mathsf{delay}(R'', c_k - c_0)$. Now $\mathsf{cost}(u_d, R') = \mathsf{cost}(u_{u(x)}, R'') + (c_k - c_0) * (d - u(x))$ which converges to $\mathsf{cost}(u_{u(x)}, R'')$ when $d \rightarrow u(x)$. Thus $\mathsf{cost}(u_d, R') + p *$

[5]A priced region, $R = (h, [r_0, \dots, r_k], [c_0, \dots, c_k])$, is self-delayable whenever $r_0 = \emptyset$.

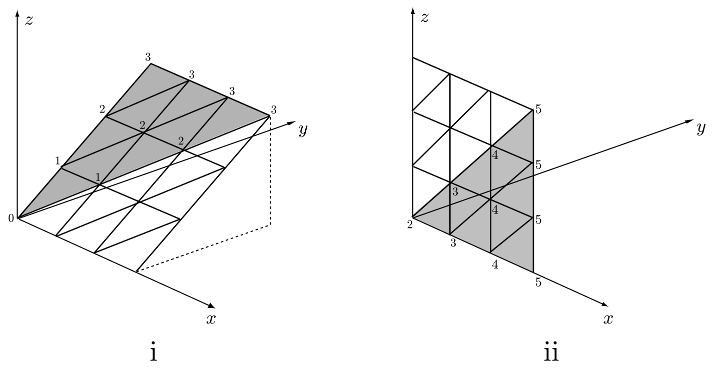


Figure 5.6: Two reachable sets of priced regions.

$d \rightarrow \mathsf{cost}(u_{u(x)}, R'') + p * d = \mathsf{cost}(u, R)$ for $d \rightarrow u(x)$. Hence, as desired, $\mathsf{mincost}((l, u)) \leq \mathsf{cost}(u, R)$.

□

Combining the two lemmas we obtain as a main theorem that the symbolic semantics captures (sufficiently) accurately the cost of reaching states and locations:

Theorem 5.1 *Let l be a location of an LPTA A. Then*

$$\mathsf{mincost}(l) = \mathsf{min}(\{\ \mathsf{mincost}(R) \mid (l, R) \text{ is reachable}\})$$

Example 5.2 *We now return to the linearly priced timed automaton in Fig. 5.2 where the value of both α and β is two, and look at its symbolic state-space. The shaded area in Fig. 5.6(i) including the lines in and around the shaded area represents some of the reachable priced regions in location B after time has passed (a number of delay actions have been taken). Only priced regions with integer values up to 3 are shown. The numbers are the cost of the vertices. The shaded area in Fig. 5.6(ii) represents in a similar way some of the reachable priced regions in location C after time has passed. For a more elaborate explanation of the reachable state-space we refer to the appendix.* □

The introduction of priced regions provides a first step towards an algorithmic solution for the minimum-cost reachability problem. However, in the present form both the integral part as well as the cost of vertices of priced regions may grow beyond any given bound during symbolic exploration. In the unpriced case, the growth of integral parts is often dealt with by suitable abstractions of (unpriced) regions, taking the maximal constant of the given timed automaton into account. Here we have chosen a very similar approach exploiting the fact, that any LPTA A may be transformed into an equivalent "*bounded*" LPTA $\tilde{A}$ in the sense that A and $\tilde{A}$ reaches the same locations with the exact same cost.

Theorem 5.2 *Let $A = (L, l_0, E, I, P)$ be an LPTA with maximal constant* max. *Then there exists* a bounded time equivalent *of A, $\tilde{A} = (L, l_0, E', I', P')$, satisfying the following:*

1. *Whenever (l, u) is reachable in $\tilde{A}$, then for all $x \in C$, $u(x) \leq \max + 2$.*

2. *For any location $l \in L$, l is reachable with cost c in A if and only if l is reachable with cost c in $\tilde{A}$*

Proof. We construct $\tilde{A} = (L, l_0, E \cup E', I', P')$, as follows. $E' = \{(l, x == \mathsf{max}_A(x) + 2, \tau, x := \mathsf{max}_A(x) + 1, l) \mid x \in C, l \in L\}$. For $l \in L$, $I'(l) = I(l) \bigwedge_{x \in C} x \leq \mathsf{max}_A(x) + 2$, $P'(l) = P(l)$. For $e \in (E \cup E')$, if $e \in E$ then $P'(e) = P(e)$ else $P'(e) = 0$.

By definition, $\tilde{A}$ satisfies the first requirement.

As to the second requirement. Let R be a relation between states from A and $\tilde{A}$ such that for $((l_1, u_1), (l_2, u_2)) \in R$ iff $l_2 = l_1$, and for each $x \in C$, if $u_1(x) \leq \mathsf{max}_A(x)$ then $u_2(x) = u_1(x)$, else $u_2(x) > \mathsf{max}_A(x)$. We show that for each state (l_1, u_1) of A which is reached with cost c, there is a state (l_2, u_2) of $\tilde{A}$, such that $((l_1, u_1), (l_2, u_2)) \in R$ and (l_2, u_2) is reached with cost c, and vice versa.

Let (l_1, u_1), (l_2, u_2) be states of A and $\tilde{A}$, respectively, We use induction on the length of some execution leading to (l_1, u_1) or (l_2, u_2).

For the base step, the length of such an execution is 0. Trivially, the cost of such an execution is 0, and if (l_1, u_1) and (l_2, u_2) are initial states, clearly $((l_1, u_1), (l_2, u_2)) \in R$.

For the transition steps, we first observe that for each clock $x \in C$, $u_1(x) \sim c$ iff $u_2(x) \sim c$ with $\sim \in \{<, \leq, >, \geq\}$ and $c \leq \mathsf{max}_A(x)$ $(*)$. Assume that $((l_1, u_1), (l_2, u_2)) \in R$, and (l_1, u_1) and (l_2, u_2) can both be reached with cost c. We make the following case distinction.

Case 1: Suppose $(l_1, u_1) \xrightarrow{\epsilon(d), p}_A (l_1, u_1 + d)$. In order to let d time pass in (l_2, u_2), we have to alternatingly perform the added τ transition to reset those clocks that have reached the $\mathsf{max}_A(x) + 2$ bound as many times as needed, and then let a bit of the time pass. Let $d_1 \dots d_m$ be a sequence of delays, such that $d = d_1 + \dots + d_m$, and for $x \in C$ and $i \in \{1, \dots, m\}$, if $\mathsf{max}_A(x) + 2 - (u_1(x) + d_1 + \dots + d_{i-1}) \geq 0$ then $d_i \leq \mathsf{max}_A(x) + 2 - (u_1(x) + d_1 + \dots + d_{i-1})$, else $d_i \leq 1 - \mathsf{frac}(u_1(x) + d_1 + \dots + d_{i-1})$. It is easy to see that for some u_2',

$$(l_2, u_2)(\xrightarrow{\tau, 0})^* \xrightarrow{\epsilon(d_1), p_1} \dots (\xrightarrow{\tau, 0})^* \xrightarrow{\epsilon(d_m), p_m} (l_2, u_2')$$

where $p_i = d_i * P(l_2)$. The cost for reaching $(l_1, u_1 + d)$ is $c + d * P_A(l_1) = c + d * P_{\tilde{A}}(l_2) = c + (d_1 + \dots + d_m) * P_{\tilde{A}}(l_2)$, which is the cost for reaching (l_2, u_2'). Now, $((l_1, u_1 + d), (l_2, u_2')) \in R$, because of the following. For each $x \in C$, If $u_1(x) > \mathsf{max}_A(x)$, then $u_2(x) > \mathsf{max}_A(x)$, and either x is not reset to $\mathsf{max}_A(x) + 1$ by any of the τ transitions, in which case still $u_2'(x) > \mathsf{max}_A(x)$, or x is reset by some of the τ transitions, and then $\mathsf{max}_A(x) + 1 \leq u_2'(x) \leq \mathsf{max}_A(x) + 2$, so $u_2'(x) > \mathsf{max}_A(x)$. If $u_1(x) \leq \mathsf{max}_A(x)$, then by $u_1(x) = u_2(x)$, $u_2(x) \leq \mathsf{max}_A(x)$. If $(u_1 + d)(x) \leq \mathsf{max}_A(x)$, then x is not touched by any of the τ transitions leading to (l_2, u_2'), hence $u_2'(x) = u_2(x) + d_1 + \dots + d_m = u_2(x) + d = (u_1 + d)(x)$. If $(u_1 + d)(x) > \mathsf{max}_A(x)$, then x may be reset by some of the τ transitions leading to (l_2, u_2'). If so, then $\mathsf{max}_A(x) + 1 \leq u_2'(x) \leq \mathsf{max}_A(x) + 2$, so $u_2'(x) > \mathsf{max}_A(x)$. If not, then $u_2'(x) = u_2(x) + d_1 + \dots + d_m = u_2(x) + d = (u_1 + d)(x) > \mathsf{max}_A(x)$.

Case 2: Suppose $(l_2, u_2) \xrightarrow{\epsilon(d),p}_A (l_2, u_2 + d)$. Then trivially $((l_1, u_1 + d), (l_2, u_2+d)) \in R$. Now we show $(l_1, u_1) \xrightarrow{\epsilon(d),p}_A (l_1, u_1+d)$. Since $(l_2, u_2+d) \in I_{\tilde{A}}$, since $I_{\tilde{A}}$ implies I_A and since $((l_1, u_1+d), (l_2, u_2+d)) \in R$, from observation $(*)$ it follows that $(l_1, u_1 + d) \in I_A$. So $(l_1, u_1) \xrightarrow{\epsilon(d),p}_A (l_1, u_1 + d)$, and trivially, the cost of reaching $(l_2, u_2 + d)$ is $c + d * P_{\tilde{A}}(l_2) = c + d * P_A(l_1)$, which is the cost of reaching $(l_1, u_1 + d)$.

Case 3: Suppose $(l_1, u_1) \xrightarrow{a,p}_A (l'_1, u'_1)$. Let (l, g, a, r, l') be a corresponding edge. Then $p = P_A((l, g, a, r, l'))$. By definition, $(l, g, a, r, l') \in E_{\tilde{A}}$ and $P_{\tilde{A}}((l, g, a, r, l')) = P_A((l, g, a, r, l'))$. From observation $(*)$ it follows that $u_1 \in g$ implies $u_2 \in g$. It is easy to see that for $x \in r$, $u'_1(x) = 0 = u_2[r \mapsto 0](x)$, and for $x \notin r$, $u'_1(x) = u_1(x)$ and $u_2(x) = u_2[r \mapsto 0](x)$, so $((l'_1, u'_1), (l', u_2[r \mapsto 0])) \in R$. Combining this with observation $(*)$ it follows that $u_1[r \mapsto 0] \in I_A(l')$ implies $u_2[r \mapsto 0] \in I_{\tilde{A}}(l')$, hence $(l_2, u_2) \xrightarrow{a,p}_{\tilde{A}} (l', u_2[r \mapsto 0])$. Clearly, the cost of reaching (l_1, u'_1) is $c + d * P_{\tilde{A}}((l, g, a, r, l')) = c + d * P_A((l, g, a, r, l'))$, which is the cost of reaching $(l_2, u_2[r \mapsto 0])$.

Case 4: Suppose $(l_2, u_2) \xrightarrow{a,p}_{\tilde{A}} (l'_2, u'_2)$. Let (l, g, a, r, l') be a corresponding edge. If $(l, g, a, r, l') \in E_A$, then the argument goes exactly like in the previous case. If $(l, g, a, r, l') \notin E_A$, then $a = \tau$, $p = 0$, $l'_2 = l' = l = l_2$, and $x \in r$ implies $u'_2(x) = \mathsf{max}_A(x) + 1$ and $u_2(x) = \mathsf{max}_A(x) + 2$. Since the cost of reaching (l'_2, u'_2) is $c + 0 = c$, it suffices to show $((l_1, u_1), (l_2, u'_2)) \in R$. For $x \notin r$, this follows trivially. For $x \in r$, $u_2(x) = \mathsf{max}_A(x) + 2$, so $u_1(x) > \mathsf{max}_A(x)$ and by $u'_2(x) = \mathsf{max}_A(x) + 1$ we have $u'_2(x) > \mathsf{max}_A(x)$. □

Now, we suggest in Fig. 5.7 a branch-and-bound algorithm for determining the minimum-cost of reaching a given target location l_g from the initial state of an LPTA. All encountered states are stored in the two data structures PASSED and WAITING, divided into explored and unexplored states, respectively. The global variable COST stores the lowest cost for reaching the target location found so far. In each iteration, a state is taken from WAITING. If it matches the target location l_g and has a lower cost than the previously lowest cost COST, then COST is updated. Then, only if the state has not been previously explored with a lower cost do we add it to PASSED and add the successors to WAITING. This bounding of the search in line 8 of Fig. 5.7 may be optimized even further by adding the constraint $\mathsf{mincost}(R) <$ COST; i.e. we only need to continue exploration if the minimum cost of the current region is below the optimal cost computed so far. Due to Theorem 5.1, the algorithm of Fig. 5.7 does indeed yield the correct minimum-cost value.

Theorem 5.3 *When the algorithm in Fig. 5.7 terminates, the value of* COST *equals* $\mathsf{mincost}(l_g)$.

Proof. First, notice that if (l_1, R_1) can reach (l_2, R_2), then a state (l_1, R'_1), where $R'_1 \leq R_1$, can reach a state (l_2, R'_2), such that $R'_2 \leq R_2$. We prove that COST equals $\min\{\mathsf{mincost}(R) \mid (l_g, R) \text{ is reachable}\}$. Assume that this does not hold. Then there exists a reachable state (l_g, R) where $\mathsf{mincost}(R) <$ COST. Thus the algorithm must at some point have discarded a state (l', R') on the

```
COST := $\infty$
PASSED := $\emptyset$
WAITING := $\{(l_0, R_0)\}$
while WAITING $\neq \emptyset$ do
    select $(l, R)$ from WAITING
    if $l = l_g$ and mincost(R) < COST then
        COST := mincost($R$)
    if for all $(l, R') \in$ PASSED : $R' \not\leq R$ then
        add $(l, R)$ to PASSED
        for all $(l', R')$ such that $(l, R) \rightarrow (l', R')$: add $(l', R')$ to WAITING
return COST
```

Figure 5.7: Branch-and-bound state-space exploration algorithm.

path to (l_g, R). This can only happen in line 8, but then there must exist a state $(l', R'') \in$ PASSED, where $R'' \leq R'$, encountered in a prior iteration of the loop. Then, there must be a state (l_g, R''') reachable from (l', R''), and COST $\leq$ $\mathsf{mincost}(R''') \leq \mathsf{mincost}(R)$, contradicting the assumption. The theorem now follows from Theorem 5.1. □

For bounded LPTA, application of Higman's Lemma [76] ensures termination. In short, Higman's Lemma says that under certain conditions the embedding order on strings is a well quasi-order.

Theorem 5.4 *The algorithm in Fig. 5.7 terminates for any bounded LPTA.*

Proof. Even if A is bounded (and hence yields only finitely many unpriced regions), there are still infinitely many priced regions, due to the unboundedness of cost of vertices. However, since all costs are positive application of Higman's lemma ensures that one cannot have an infinite sequence $\langle (c_1^i, \ldots, c_m^i) : 0 \leq i < \infty \rangle$ of cost-vectors (for any fixed length m) without $c_l^j \leq c_l^k$ for all $l = 1, \ldots, m$ for some $j < k$. Consequently, due to the finiteness of the sets of locations and unpriced regions, it follows that one cannot have an infinite sequence $\langle (l_i, R_i) : 0 \leq i < \infty \rangle$ of symbolic states without $l_j = l_k$ and $R_j \leq R_k$ for some $j < k$, thus ensuring termination of the algorithm. □

Finally, combining Theorem 5.3 and 5.4, it follows, due to Theorem 5.2, that the minimum-cost reachability problem is decidable.

Theorem 5.5 *The minimum-cost problem for LPTA is decidable.*

5.5 Conclusion

In this paper, we have successfully extended the work on regions and their operations to a setting of timed automata with linear prices on both transitions and locations. We have given the principle basis of a branch-and-bound algorithm for the minimum-cost reachability problem, which is based on an accurate symbolic semantics of timed automata with linear prices, and thus showing the minimum-cost reachability problem to be decidable.

The algorithm is guaranteed to be rather inefficient and highly sensitive to the size of constants used in the guards of the automata — a characteristic inherited from the time regions used in the basic data-structure of the algorithm. An obvious continuation of this work is therefore to investigate if other more (in practice) efficient data structures can be found. Possible candidates include data structures used in reachability algorithms of timed automata, such as DBMs, extended with costs on the vertices of the represented zones (i.e. convex sets of clock assignments). In contrast to the priced extension of regions, operations on such a notion of priced zones[6] can not be obtained as direct extensions of the corresponding operations on zones with suitable manipulation of cost of vertices.

The need for infimum in the definition of minimum cost executions arises from linearly priced timed automata with strict bounds in the guards, such as the one shown in Fig. 5.3 and discussed in Example 5.1. Due to the use of infimum, a linearly priced timed automaton is not always able to realize an execution with the exact minimum cost of the automata, but will be able to realize one with a cost (infinitesimally) close to the minimum value. If all guards include only non-strict bounds, the minimum cost trace can always be realized by the automaton. This fact can be shown by defining the minimum-cost problem for executions covered by a given symbolic trace as a linear programming problem.

In this paper we have presented an algorithm for computing minimum-costs for reachability of linearly priced timed automata, where prices are given as constants (natural numbers). However, a slight modification of our algorithm provides an extension to a parameterized setting, in which (some) prices may be parameters. In this setting, costs within priced regions will be finite collections, C, of linear expressions over the given parameters rather than simple natural numbers. Intuitively, C denotes for any given instantiation of the parameters the minimum of the concrete values denoted by the linear expressions within C. Now, two cost-expressions may be compared simply by comparing the sizes of corresponding parameters, and two collections C and D (both denoting minimums) are related if for any element of D there is a smaller element in C. In the modified version of algorithm Fig. 5.7, COST will similarly be a collection of (linear) cost-expressions with which the goal-location has been reached (so far). From recent results in [6] (generalizing Higman's lemma) it follows that the ordering on (parameterized) symbolic states is again a well-quasi ordering, hence guaranteeing termination of our algorithm. Also, we are currently working on extending the algorithmic solution offered here to synthesis of minimum-cost controllers in the sense of [23]. In this extension, a priced region will be given by a conventional unpriced region together with a min-max expression over cost vectors for the vertices of the region. Also for this extension it follows from recent results in [6] (generalizing Higman's lemma) that the orderings on symbolic states are again well-quasi orderings, hence guaranteeing termination of our algorithms.

[6]In particular, the reset-operation and the delay-operation.

Acknowledgements

The authors would like to thank Lone Juul Hansen for her great, creative effort in making the figures of this paper. Also, the authors would like to thank Parosh Abdulla for sharing with us some of his expertise and knowledge on the world beyond well-quasi orderings.

Appendix: Example of Symbolic State-Space

In this appendix, we present part of the symbolic state-space of the linearly priced timed automaton in Fig. 5.2 where the value of both α and β is two. Figures 5.8(i)-(viii) show some of the priced regions reachable in a symbolic representation of the states space. We only show the priced regions with integer value less than or equal to three.

Initially all three clocks have value zero and when delaying the clocks keep on all having the same value. Therefore the priced regions reachable from the initial state are the ones on the line from $(0, 0, 0)$ through $(3, 3, 3)$ shown in Fig. 5.8(i). The numbers on the line are the costs of the vertices of the priced regions represented by the line. Since the cost of staying in location A is one, the price of delay one time unit is one. Therefore the cost of reaching the point $(3, 3, 3)$ is three.

The priced regions presented in Fig. 5.8(ii) are the ones reachable after taking the transition to location B, resetting the x clock. Performing the reset does not change any of the costs, since the new priced regions are still one-dimensional and no vertices are collapsed. In Fig. 5.8(iii) the reachable priced regions are marked by a shaded area, including the lines inside the area and on the boundary. These priced regions are reachable from the priced regions in Fig. 5.8(ii).

Taking the transition from location B to location C causes clocks y and z to be reset. After resetting the priced regions in Fig. 5.8(iii), the priced regions in Fig. 5.8(iv) are reachable. Finding the cost of a state s after the reset (projection) is done by taking the minimum of the cost of the states projecting to s. When delaying from these priced regions, the priced regions in Fig. 5.8(v) are reached (again represented by the shaded area and the lines in and surrounding it).

Now we are left with a choice; Either we can take the transition to location D, or take the loop transition back to location C. Taking the transition to location D is only possible if the guard $x \geq 2 \wedge y \leq 1$ is satisfied. Some of the vertices in Fig. 5.8(vi) are marked: only priced regions where all vertices are marked satisfy the guard. Before reaching location E from D with these priced regions, we must delay at least two time units to satisfy the guard $z \geq 3$ on the transition from location D to location E (this part of the symbolic state-space is not shown in the figure). The minimum cost of reaching location E in this way is six.

The other possibility from location C is to take the loop transition which resets the y clock. After resetting y in the priced regions in Fig. 5.8(v), the priced regions in Fig. 5.8(vii) are reachable. From these priced regions we again can let time pass. However, a two dimensional picture of the three dimensional priced regions, reachable from the priced regions in Fig. 5.8(vii), is very hard to understand. Therefore, we have chosen to focus on the priced regions which satisfy the guard on the transition to location D. These priced regions are displayed by stating the cost of their vertices in Fig. 5.8(viii). The reachable priced regions satisfying the guard are the ones for which all vertices are marked with a cost in Fig. 5.8(viii). Three of the priced regions satisfying the guard on the transition from location C to location D, also satisfies the guard on the transition to location E. This is the two vertices where z has the value three and the line between these two points. The cost of reaching these points is five, so it is also possible to reach location E with this cost.

After taking the loop transition in location C once we also had the choice of taking it again. Doing this would yield the same priced regions as displayed in Fig. 5.8(vii) but now with two added to the cost. Therefore the new priced regions would be more costly than the priced regions already found and hence not explored by our algorithm.

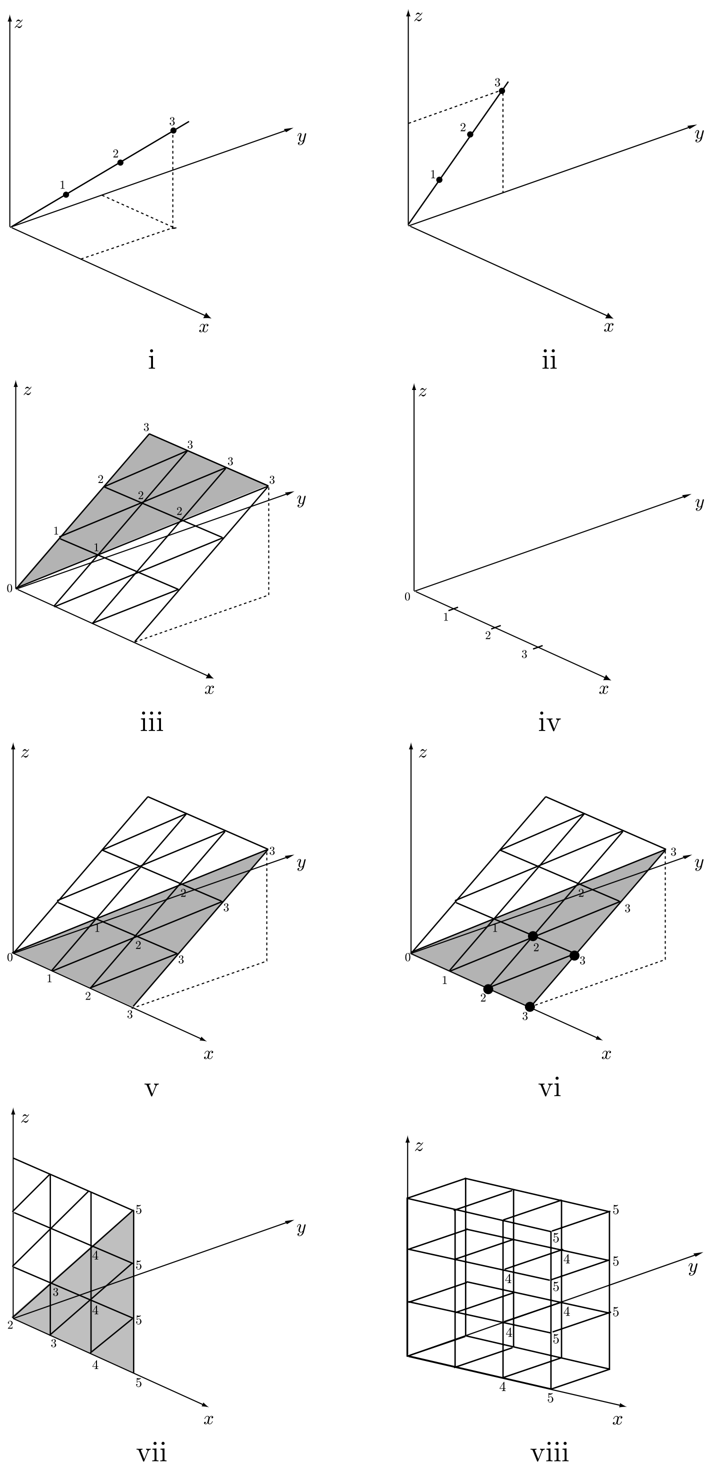


Figure 5.8: Sets of reachable priced regions.

Chapter 6

Open Maps for Timed Transition Systems

The paper *Bisimulation and Open Maps for Timed Transition Systems* presented in this chapter has been published as a technical report [85], in a journal version [128] and a conference paper [84].

[84] T. Hune, and M. Nielsen. Timed Bisimulation and Open Maps . In *Proceedings of Mathematical Foundations of Computer Science (MFCS'98)*, pages 378–387, 1998.

[85] T. Hune, and M. Nielsen. Timed Bisimulation and Open Maps. Technical Report RS-98-04, BRICS, February 1998.

[128] M. Nielsen and T. Hune. Bisimulation and Open Maps for Timed Transition Systems. In *Fundamenta Informatica, special issue dedicated to Professor Arto Salomaa*, pages 61–77, 1999.

The journal version extends the conference paper by adding some proofs and a section about handling invariants in locations. The technical report extends the journal version by adding complete proofs. Except for minor typographical changes the content of this chapter is equal to the technical report [85].

Bisimulation and Open Maps for Timed Transition Systems

Thomas Hune[*] and Mogens Nielsen[*]

Abstract

Formal models for real-time systems have been studied intensively over the past decade. Much of the theory of untimed systems have been lifted to real-time settings. One example is the notion of bisimulation applied to timed transition systems, which is studied here within the general categorical framework of open maps. We define a category of timed transition systems, and show how to characterize standard timed bisimulation in terms of spans of open maps with a natural choice of a path category. This allows us to apply general results from the theory of open maps, e.g. the existence of canonical models and characteristic logics. Also, we obtain here an alternative proof of decidability of bisimulation for finite transition systems, and illustrate the use of open maps in finite presentations of bisimulations

6.1 Introduction

When specifying and reasoning about a computing system, it is often sufficient to view its behavior from a classical point of view in terms of computations defined as sequences of atomic discrete actions of the system. For some systems, however, it is essential to include more detailed information. In the specification of a controller of a railway crossing it is not sufficient to state that the gate is closed when the train is at the crossing. It is equally important to specify timing constraints on the actions of gate closing and train crossing. Formal models for such so-called real-time systems have been studied intensively over the past decade, e.g. the timed automata [12], timed process algebras [145], timed nets [108], and timed Petri Nets [119].

Much of the theory of untimed systems has been lifted successfully to these models of real-time behavior of systems. As examples, many results from automata theory apply also to timed automata, [12, 13, 22], and a number of timed versions of classical specification logics have been studied, [14, 103].

In this paper we focus on the classical notion of bisimulation [121] which has already been introduced and studied for real-time models by many researchers, e.g. in [145, 19, 125, 18]. A large part of the elegant theory of bisimulation

[*]Basic Research in Computer Science, **BRICS**, Centre of the Danish National Research Foundation. Department of Computer Science, University of Aarhus, Denmark, E-mail: `[baris,mn]@brics.dk`

for transition systems and reactive languages has been lifted to the real-time setting. As an example, bisimulation was shown decidable for finite timed transition systems by Čerāns [49], and efficient algorithms checking for bisimilarity have been discovered [103, 143] and implemented in tools for automatic verification [98].

Our aim here is to apply the general categorical framework of open maps [95] to timed transition systems. The open map approach provides a general concept of bisimulation for any categorical model of computation, i.e. models consisting of objects (systems) and morphisms (to be thought of as simulations between two systems). The general definition is in terms of spans of so-called open maps, which are those morphisms which, roughly speaking, reflect as well as preserve behavior. Formally, the definition of open maps is parameterized not just on a categorical presentation of a model (i.e. on the choice of morphisms), but also on a notion of computation path and what it means to extend a computation path by another.

For the standard model of transition systems, computation paths are naturally chosen as sequences of consecutive transitions, formally picked out by morphisms from strings of actions, extended by standard composition of strings. With this choice, it was shown in [95] that the open map bisimulation simply specializes to Milner's notion of bisimulation. However, many other behavioral equivalences are captured by the open morphism approach, e.g. Hoare's trace equivalence and Milner's weak bisimulation, both of which may be obtained by slightly changing the notion of path extension from the one indicated above [52]. Also, the open morphism approach has been applied successfully to different categories of models, e.g. probabilistic [52], higher-order models [48], and models with independence [95].

Rather than having bisimulations defined in terms of two parameters, a model and a path category, it was suggested in [95] to study presheaves as models derived directly from path categories. Intuitively, a presheaf represents the effect of gluing together a set of computation paths to form a nondeterministic computation, and hence can be looked upon as labelled transition systems, in which the labels are morphisms of path extension. Following [144] this yields logical and game-theoretic characterizations of open morphisms and their bisimulations on presheaves. Furthermore, models and their notion of bisimulation can be understood in a uniform way via their representation as presheaves, and via this representation, the characterizations can be specialized to concrete models. The characteristic logics take the form of Hennessy-Milner like modal logics, with modalities indexed by path morphisms (path extensions, future modalities) and their inverses (path projections, past modalities).

Here we define a category of timed transition systems, where the morphisms are to be thought of as simulations, with computation paths which are equivalent to the standard notion of runs of timed words. We show the derived notion of bisimulation in terms of open maps to coincide with the standard timed bisimulation from e.g. [49]. Hence, we may apply the general results from [95], e.g. obtaining canonical models and characteristic games and logics.

Furthermore, we show within the framework of open maps that bisimilarity is decidable for finite timed transition systems. As for many existing results for timed models, including results concerning verification of real-time systems, our proof relies heavily on the idea behind the regional construction of [12, 13], which essentially provides a finite description of the uncountable behavior of a finite real-time system.

One of the main advantages of Milners notion of bisimulation for untimed transition systems, is the fact that for two transition systems, the property of being bisimilar may be expressed in terms of presenting an explicit bisimulation between the two systems, i.e. a relation on the states of the two systems. Unfortunately, this property does not generalize to the setting of timed transition systems, where bisimulations are defined in terms of the uncountable unfolded version of given timed transition systems, and where the decision procedures from e.g. [49] produce relations over nontrivial regional constructions. Here, we obtain as a corollary, a way of presenting bisimilarity between two finite timed transition systems in terms of a span with a finite vertex.

In Section 2 we define formally our category of timed transition systems and computation paths, and the set-up is shown to have a number of useful properties following the approach of [95]. Next, in Section 3 the resulting notion of bisimulation is studied, and it is shown to coincide with the standard notion of timed bisimulation. A new proof of the decidability of timed bisimulation is provided in Section 4, and the use of open maps to express bisimulations is illustrated. We briefly address the issue of robustness of our approach in Section 5 by extending our results to models with state time-invariants. Section 6 contains some conclusions and ideas for future work.

6.2 A Category of Timed Transition Systems

In the following we define the categorical set-up for our use of the open map approach.

The objects of our model category will be timed transition systems, i.e. timed automata in the sense of Alur and Dill [13] without accepting states and acceptance conditions (called timed transition tables in [13]).

Definition 6.1 (Timed Transition Systems) *A timed transition system is a quintuple* $(S, \Sigma, s^{in}, X, T)$ *where*

- S *is a set of states and* s^{in} *is the initial state.*
- Σ *is a finite alphabet of actions.*
- X *is a set of clock variables.*
- T *is the set of transitions such that* $T \subseteq S \times \Sigma \times \Delta \times 2^X \times S$ *where* Δ *is a clock constraint generated by the grammar* $\Delta ::= c \sharp x \mid x + c \sharp y \mid \Delta \wedge \Delta$ *in which* $\sharp \in \{\leq, <, \geq, >\}$*, c is a real valued constant and x, y are clock variables. A transition* $(s, \sigma, \delta, \lambda, s')$ *is written* $s \xrightarrow[\delta,\lambda]{\sigma} s'$*.*

Timed transition systems are to be thought of as generalizations of standard transition systems, having runs over timed words as obvious generalizations of words over an alphabet.

Definition 6.2 (Timed Words) *A timed word over an alphabet Σ is a finite sequence of pairs $\alpha = (\sigma_1, \tau_1)(\sigma_2, \tau_2)(\sigma_3, \tau_3)\cdots(\sigma_n, \tau_n)$, where for all $1 \leq i \leq n$, $\sigma_i \in \Sigma, \tau_i \in \mathsf{R}_+$ and furthermore $\tau_i < \tau_{i+1}$.*

A pair (σ, τ) represents an occurrence of action σ at time τ relative to the starting time (0) of the execution.

Example 6.1 *The timed transition system in Figure 6.1 has two clocks x and y, and three actions* a,b,c. *The state s_0 is the initial state.*

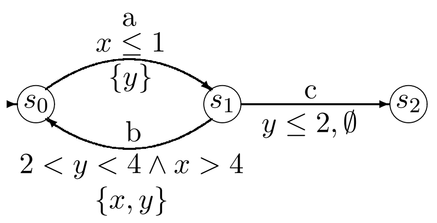


Figure 6.1: A timed transition system .

Before introducing formally computations of timed transition systems, we need the notion of a clock evaluation.

Definition 6.3 (Clock Evaluation) *A clock evaluation ν is a function $\nu : X \rightarrow \mathsf{R}_+$ which assigns times to the clock variables of a system. We define $(\nu + c)(x) := \nu(x) + c$ for all clock variables x. If λ is a set of clock variables then $\nu[\lambda \mapsto 0](x) := 0$ if $x \in \lambda$, and $\nu(x)$ otherwise.*

A constraint δ is satisfied by clock evaluation ν iff the expression $\delta[\nu(x)/x]$[1] evaluates to true. A constraint δ defines a subset of R^n where n is the number of clocks in X. We will speak of this subset as the meaning of δ and write it $[\![\delta]\!]_X$. As an example the meaning of the constraint on the transition from s_0 to s_1 in Figure 6.1 is the hatched area in Figure 6.2. A clock evaluation defines

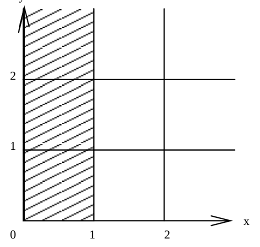


Figure 6.2: Interpretation of constraint $[\![x \leq 1]\!]_{\{x,y\}}$.

a point in R^n which we shall denote by $[\![\nu]\!]_X$, so the constraint δ is satisfied for the clock evaluation ν if and only if $[\![\nu]\!]_X \in [\![\delta]\!]_X$.

[1]$\delta[y/x]$ is syntactic substitution of y for x in δ.

Definition 6.4 *Let $\mathcal{T}$ be a timed transition system. A configuration is a pair $\langle s, \nu \rangle$, where s is a state and ν is a clock evaluation. A run of $\mathcal{T}$ is a sequence $\langle s_0, \nu_0 \rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, \nu_1 \rangle \xrightarrow[\tau_2]{\sigma_2} \cdots \xrightarrow[\tau_n]{\sigma_n} \langle s_n, \nu_n \rangle$ such that for all $i > 0$ there is a transition $s_{i-1} \xrightarrow[\delta_i, \lambda_i]{\sigma_i} s_i$ such that $[\![\nu_{i-1} + (\tau_i - \tau_{i-1})]\!]_X \in [\![\delta_i]\!]_X$ and $\nu_i = (\nu_{i-1} + (\tau_i - \tau_{i-1}))[\lambda_i \mapsto 0]$. The state s_0 is the initial state of $\mathcal{T}$, ν_0 is the constant 0 function, and τ_0 is defined to be 0. A run as above is said to generate the timed word $(\sigma_1, \tau_1)\,(\sigma_2, \tau_2)\,(\sigma_3, \tau_3) \cdots (\sigma_n, \tau_n)$.*

Example 6.2 *A run in the timed transition system in Figure 6.1 generating the timed word $(a, 0.9)(c, 2.3)$ is*

$$\langle s_0, [0,0] \rangle \xrightarrow[0.9]{a} \langle s_1, [0.9, 0] \rangle \xrightarrow[2.3]{c} \langle s_2, [2.3, 1.4] \rangle$$

where $[2.3, 1.4]$ denotes the clock assignment assigning 2.3 to the clock variable x and 1.4 to y.

Another run in the timed transition system could be

$$\langle s_0, [0,0] \rangle \xrightarrow[0.7]{a} \langle s_1, [0.7, 0] \rangle \xrightarrow[4.2]{b} \langle s_0, [0,0] \rangle \xrightarrow[4.4]{a} \langle s_1, [0.2, 0] \rangle \xrightarrow[8.3]{b} \langle s_0, [0,0] \rangle$$

which generates the timed word $(a, 0.7)(b, 4.2)(a, 4.4)(a, 8.3)$.

The morphisms of our model category will be simulation morphisms following the approach of [95]. This leads to the following definition of a morphism, consisting of two functions, one mapping states of the simulated system to simulating states of the other, and one mapping clocks of the simulating system to simulated clocks of the other.

Definition 6.5 *A morphism (m, η) between timed transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$ consists of two components; a map $m : S_1 \to S_2$ between the states and a map $\eta : X_2 \to X_1$ between the clocks. These maps must satisfy that $m(s_1^{in}) = s_2^{in}$ and whenever there is a transition in $\mathcal{T}_1$ of the form $s_1 \xrightarrow[\delta_1, \lambda_1]{\sigma} s_1'$ then there is a transition $m(s_1) \xrightarrow[\delta_2, \lambda_2]{\sigma} m(s_1')$ in $\mathcal{T}_2$ satisfying the following two constraints:*

- $\lambda_2 = \eta^{-1}(\lambda_1)$ *where* $\eta^{-1}(\lambda_1) = \{x \in X_2 \mid \eta(x) \in \lambda_1\}$
- $[\![\delta_1]\!]_{X_1} \subseteq [\![\delta_2[\eta(x)/x]]\!]_{X_1}$

Example 6.3 *Consider the map m from the states of the timed transition system in Figure 6.3 to the states of the one in Figure 6.1, mapping states with index i to s_i, paired with the map η sending the clock variable x to z and y to u. We leave it to the reader to check to check that m, η constitute a morphism.*

Definition 6.6 *For a function $\eta : X' \to X$ and a clock evaluation $\nu : X \to \mathsf{R}_+$ we define $\eta^{-1}(\nu) : X' \to \mathsf{R}_+$, the inverse image of ν under η, as*

$$\eta^{-1}(\nu)(x) := \nu(\eta(x))$$

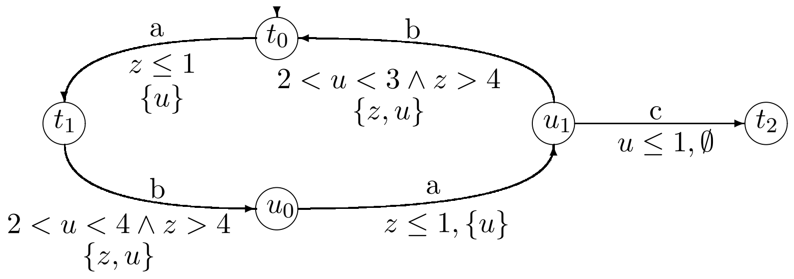


Figure 6.3: A timed transition system.

Theorem 6.1 *Given two timed transition systems $\mathcal{T}$ and $\mathcal{T}'$ and a morphism (m, η) from $\mathcal{T}$ to $\mathcal{T}'$. If*

$$\langle s_0, \nu_0 \rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, \nu_1 \rangle \xrightarrow[\tau_2]{\sigma_2} \cdots \xrightarrow[\tau_n]{\sigma_n} \langle s_n, \nu_n \rangle$$

is a run of $\mathcal{T}$ generating the timed word $(\sigma_1, \tau_1)(\sigma_2, \tau_2)(\sigma_3, \tau_3)\cdots(\sigma_n, \tau_n)$, then

$$\langle m(s_0), \eta^{-1}(\nu_0) \rangle \xrightarrow[\tau_1]{\sigma_1} \langle m(s_1), \eta^{-1}(\nu_1) \rangle \xrightarrow[\tau_2]{\sigma_2} \cdots \xrightarrow[\tau_n]{\sigma_n} \langle m(s_n), \eta^{-1}(\nu_n) \rangle$$

is a run of $\mathcal{T}'$ generating the same timed word.

Proof We will prove this by induction on the length of the run.

As base case, we have the empty run with just one configuration. Since the initial state of $\mathcal{T}$ is mapped to the initial state of $\mathcal{T}'$ and all clock values initially are set to 0, we also have $\forall x' \in X' : \eta^{-1}(\nu_0)(x') = 0$ which is the initial clock evaluation for a run in $\mathcal{T}'$.

For the induction step, assume $\mathcal{T}$ is in the configuration $\langle s_i, \nu_i \rangle$, $\mathcal{T}'$ is in the configuration $\langle m(s_i), \eta^{-1}(\nu_i) \rangle$, and $\mathcal{T}$ can extend its run by

$$\langle s_i, \nu_i \rangle \xrightarrow[\tau_{i+1}]{\sigma_{i+1}} \langle s_{i+1}, \nu_{i+1} \rangle$$

extending the generated timed word with the element $(\sigma_{i+1}, \tau_{i+1})$.

The extension uses some transition

$$s_i \xrightarrow[\delta_{i+1}, \lambda_{i+1}]{\sigma_{i+1}} s_{i+1}$$

in $\mathcal{T}$ satisfying $[\![\nu_i + (\tau_{i+1} - \tau_i)]\!]_X \in [\![\delta_{i+1}]\!]_X$. From the definition of a morphism we must have some transition

$$m(s_i) \xrightarrow[\delta'_{i+1}, \lambda'_{i+1}]{\sigma_{i+1}} m(s_{i+1})$$

in $\mathcal{T}'$ such that $\lambda'_{i+1} = \eta^{-1}(\lambda_{i+1})$ and $[\![\delta_{i+1}]\!]_X \subseteq [\![\delta'_{i+1}[\eta(x)/x]]\!]_X$. From the latter property we get that $[\![\eta^{-1}(\nu_i) + (\tau_{i+1} - \tau_i)]\!]_{X'} \in [\![\delta'_{i+1}]\!]_{X'}$ so the transition can be used to extend the run in $\mathcal{T}'$, obtaining

$$(m(s_i), \eta^{-1}(\nu_i)) \xrightarrow[\tau_{i+1}]{\sigma_{i+1}} (m(s_{i+1}), \nu'_{i+1})$$

where

$$\begin{array}{rcl}\nu'_{i+1} & = & (\eta^{-1}(\nu_i) + (\tau_{i+1} - \tau_i))[\lambda'_{i+1} \mapsto 0] \\ & = & (\eta^{-1}(\nu_i) + (\tau_{i+1} - \tau_i))[\eta^{-1}(\lambda_{i+1}) \mapsto 0] \\ & = & (\eta^{-1}(\nu_i + (\tau_{i+1} - \tau_i)))[\eta^{-1}(\lambda_{i+1}) \mapsto 0] \\ & = & \eta^{-1}((\nu_i + (\tau_{i+1} - \tau_i))[\lambda_{i+1} \mapsto 0]) \\ & = & \eta^{-1}(\nu_{i+1})\end{array}$$

□

Example 6.4 *Using the morphism from Example 6.3 the run*

$$\langle t_0, [0,0]\rangle \xrightarrow[0.7]{a} \langle t_1, [0.7,0]\rangle \xrightarrow[4.2]{b} \langle t_2, [0,0]\rangle \xrightarrow[4.4]{a} \langle t_3, [0.2,0]\rangle \xrightarrow[8.3]{b} \langle t_0, [0,0]\rangle$$

in the timed transition system in Figure 6.3 is simulated by the second run in Example 6.2. Here $[0.7, 0]$ *is notation for* ν *assigning the value* 0.7 *to the clock* z *and the value* 0 *to* u.

So, in the formal sense of Theorem 6.1 we have shown that the morphisms from Definition 6.5 do represent a notion of simulation. Our category of timed transition systems is defined as follows.

Definition 6.7 *The category* CTTS_Σ *has timed transition systems with alphabet* Σ *as objects, and the morphisms from Definition 6.5 as arrows. For morphisms* $\mathcal{T} \xrightarrow{(m,\eta)} \mathcal{T}'$ *and* $\mathcal{T}' \xrightarrow{(m',\eta')} \mathcal{T}''$ *composition is defined as* $(m', \eta') \circ (m, \eta) := (m' \circ m, \eta \circ \eta')$. *The identity morphism is the morphism where both* m *and* η *are the identity function.*

Proposition 6.1 CTTS_Σ *is a category.*

Proof The only non-trivial part of the proof is to see that composition is well-defined. Assume we have morphisms $\mathcal{T} \xrightarrow{(m,\eta)} \mathcal{T}'$ and $\mathcal{T}' \xrightarrow{(m',\eta')} \mathcal{T}''$. A transition $s_1 \xrightarrow[\delta,\lambda]{\sigma} s_2$ in $\mathcal{T}$ implies the existence of a transition $m(s_1) \xrightarrow[\delta',\lambda']{\sigma} m(s_2)$ in $\mathcal{T}'$ where $\lambda' = \eta^{-1}(\lambda)$ and $[\![\delta]\!]_X \subseteq [\![\delta'[\eta(x)/x]]\!]_X$. This transition implies the existence of a transition $m'(m(s_1)) \xrightarrow[\delta'',\lambda'']{\sigma} m'(m(s_2))$ in $\mathcal{T}''$ where $\lambda'' = \eta'^{-1}(\lambda') = \eta'^{-1}(\eta^{-1}(\lambda))$ and $[\![\delta']\!]_X \subseteq [\![\delta''[\eta'(x)/x]]\!]_X$. Combining these facts we get $[\![\delta]\!]_X \subseteq [\![\delta''[(\eta \circ \eta')(x)/x]]\!]_X$ from which we conclude that composition is well-defined.

□

CTTS_Σ has a number of useful properties. For our purpose here we only need the following.

Theorem 6.2 CTTS_Σ *has (binary) products.*

Proof Given two timed transition systems $\mathcal{T}_1 = (S_1, \Sigma, s_1^{in}, X_1, T_1)$ and $\mathcal{T}_2 = (S_2, \Sigma, s_2^{in}, X_2, T_2)$, we define the product of the two systems as $\mathcal{T}_1 \times \mathcal{T}_2 = (S_1 \times S_2, \Sigma, (s_1^{in}, s_2^{in}), X_1 \uplus X_2, \mathcal{T})$, where $X_1 \uplus X_2$ denotes the disjoint union of X_1 and X_2, and the set of transitions $\mathcal{T}$ consists of all transitions of the form $(s_1, s_2) \xrightarrow[\delta_1 \wedge \delta_2, \lambda_1 \cup \lambda_2]{\sigma} (s_1', s_2')$ such that $s_i \xrightarrow[\delta_i, \lambda_i]{\sigma} s_i'$ belongs to $\mathcal{T}_i$ for $i = 1, 2$.

The projections $(m_i, \eta_i) : \mathcal{T}_1 \times \mathcal{T}_2 \to \mathcal{T}_i$ for $i = 1, 2$ are defined as expected, with m_i as the projection on states and η_i is the embedding of X_i into $X_1 \uplus X_2$. It follows easily that this defines products in CTTS$_\Sigma$.

□

Theorem 6.3 CTTS$_\Sigma$ *has pullbacks.*

Proof Given two morphisms $(m_1, \eta_1) : \mathcal{T}_1 \to \mathcal{T}$ and $(m_2, \eta_2) : \mathcal{T}_2 \to \mathcal{T}$, we construct $\mathcal{T}_1 \times_\mathcal{T} \mathcal{T}_2$ and two morphisms $(m_i', \eta_i') : \mathcal{T}_1 \times_\mathcal{T} \mathcal{T}_2 \to \mathcal{T}_i$ such that

$$(m_1, \eta_1) \circ (m_1', \eta_1') = (m_2, \eta_2) \circ (m_2', \eta_2') \tag{6.1}$$

The construction of m_i' and the states of $\mathcal{T}_1 \times_\mathcal{T} \mathcal{T}_2$ is based on pullbacks in the category of sets with functions. Similarly the construction of η_i' and the clocks of $\mathcal{T}_1 \times_\mathcal{T} \mathcal{T}_2$ is based on pushouts in the category of sets with functions, i.e. the clocks of $\mathcal{T}_1 \times_\mathcal{T} \mathcal{T}_2$ are the equivalence classes of the equivalence relation $\mathcal{R}$ over $X_1 \uplus X_2$ generated by $\mathcal{R}_0$ where

$$\mathcal{R}_0 = \{(x_1, x_2) \mid \exists x \in X. \eta_1(x) = x_1 \text{ and } \eta_2(x) = x_2\},$$

and η_i' sends a clock variable of $\mathcal{T}_i$ to the equivalence class to which it belongs. More specifically we define $\mathcal{T}_1 \times_\mathcal{T} \mathcal{T}_2$ as follows.

- S$\times_\mathcal{T}$: $\{(s_1, s_2) \in S_1 \times S_2 \mid m_1(s_1) = m_2(s_2)\}$
- s$^{\text{in}}\times_\mathcal{T}$: (s_1^{in}, s_2^{in})
- X$\times_\mathcal{T}$: the equivalence classes of $\mathcal{R}$ defined above
- T$\times_\mathcal{T}$: $(s_1, s_2) \xrightarrow[\delta_1[\eta_1'(x)/x] \wedge \delta_2[\eta_2'(x)/x], \eta_1'(\lambda_1) \cup \eta_2'(\lambda_2)]{\sigma} (s_1', s_2')$, whenever $s_i \xrightarrow[\delta_i, \lambda_i]{\sigma} s_i'$ and $(s_1, s_2), (s_1', s_2')$ belongs to S$\times_\mathcal{T}$.

With $m_i'(s_1, s_2) = s_i$ we leave it for the reader to check that (m_i', η_i') is indeed a morphism, and it follows immediately from the underlying conditions from the category of sets with functions that the required commutativity of (6.1) is satisfied.

Now consider $\mathcal{T}'$ with morphisms $(m_i'', \eta_i'') : \mathcal{T}' \to \mathcal{T}_i$ for $i = 1, 2$, such that the following diagram commutes.

$$\begin{tikzcd}
\mathcal{T}' \arrow[drr, bend left, "(m_1'', \eta_1'')"] \arrow[ddr, bend right, "(m_2'', \eta_2'')"'] & & \\
& \mathcal{T}_1 \times_\mathcal{T} \mathcal{T}_2 \arrow[r, "(m_1', \eta_1')"'] \arrow[d, "(m_2', \eta_2')"] & \mathcal{T}_1 \arrow[d, "(m_1, \eta_1)"] \\
& \mathcal{T}_2 \arrow[r, "(m_2, \eta_2)"'] & \mathcal{T}
\end{tikzcd}$$

The required morphism (m, η) from $\mathcal{T}'$ to $\mathcal{T}_1 \times_{\mathcal{T}} \mathcal{T}_2$ is defined as expected, i.e. $m(s') = (m''_1(s'), m''_2(s'))$ and $\eta(x) = \eta''_1(x) \cup \eta''_2(x)$. We leave it for the reader to check that (m, η) indeed is a morphism. Finally, from the underlying constructions in the category of sets with functions we get that the required commutativities $(m''_i, \eta''_i) = (m'_i, \eta'_i) \circ (m, \eta)$ hold for $i = 1, 2$.

□

6.2.1 A Path Category

Following the standards of timed transition systems and [95], we would like to choose timed words over Σ with word extension as our category of computation paths. However, it is not immediately clear how to see formally this choice as a subcategory of CTTS$_\Sigma$, as required in the approach of [95].

Definition 6.8 *Given a timed word $\alpha = (\sigma_1, \tau_1)\,(\sigma_2, \tau_2)\,(\sigma_3, \tau_3) \cdots (\sigma_n, \tau_n)$, we define a timed transition system $\mathcal{T}_\alpha$: $0 \xrightarrow[\delta_1,\lambda_1]{\sigma_1} 1 \xrightarrow[\delta_2,\lambda_2]{\sigma_2} \cdots \xrightarrow[\delta_n,\lambda_n]{\sigma_n} n$ as follows. The states are the integers $0..n$, with 0 as the initial state, and the set of clock variables, X, consists of the 2^n subsets of states $\{1, 2, \ldots, n\}$. We define λ_i and δ_i as*

$$\lambda_i = \{x |\, i \in x\} \text{ and } \delta_i = \bigwedge_{x \in X} (x = \tau_i - \tau_{I(i,x)})$$

where $I(i, x) := \max\{k \in x \cup \{0\} \mid k < i\}$, and $\tau_0 := 0$. The index returned by $I(i, x)$ is the index of the last state at which x was reset. We write $\mathcal{T}_\alpha$ for the transition system in CTTS$_\Sigma$ *representing α.*

The only purpose of this seemingly ad hoc construction is that it allows us to represent the category of timed words with extension inside CTTS$_\Sigma$, and to identify runs of α in $\mathcal{T}$ with morphisms from $\mathcal{T}_\alpha$ to $\mathcal{T}$, as expressed formally in the following two results.

Proposition 6.2 *The construction of $\mathcal{T}_\alpha$ from α above, extends to a full and faithful functor from the category of timed words (as objects) and word extension (as morphisms) into* CTTS$_\Sigma$

Proof The main observation is that for all timed words α, α', there is at most one morphism between $\mathcal{T}_\alpha$ and $\mathcal{T}_{\alpha'}$.

□

Theorem 6.4 *Given a timed transition system $\mathcal{T}$, and a timed word $\alpha = (\sigma_1, \tau_1)(\sigma_2, \tau_2) \ldots (\sigma_n, \tau_n)$. For each run of α in $\mathcal{T}$,*

$$\langle s_0, \nu_0 \rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, \nu_1 \rangle \xrightarrow[\tau_2]{\sigma_2} \ldots \xrightarrow[\tau_n]{\sigma_n} \langle s_n, \nu_n \rangle \tag{6.2}$$

we may associate a morphism $(m, \eta) : \mathcal{T}_\alpha \to \mathcal{T}$ where:

$m(i) = s_i$

$\eta(x) = \{i \mid 1 \leq i \leq n \text{ and } \nu_i(x) = 0\}$.

Furthermore, this association is a bijection between the runs of α in $\mathcal{T}$ and the morphisms $\mathcal{T}_\alpha \xrightarrow{(m,\eta)} \mathcal{T}$.

Proof It follows from the definition of runs and the definition of $\mathcal{T}_\alpha$ that (m, η) as defined is indeed a morphism.

Now, let (m, η) be a morphism from $\mathcal{T}_\alpha$ to $\mathcal{T}$. With (m, η) we associate the run of the form (6.2) where

$$s_i = m(i)$$

$$\nu_i(x) = \begin{cases} 0 & \text{if } i = 0 \text{ or } i \in \eta(x) \\ \nu_{i-1}(x) + (\tau_i - \tau_{i-1}) & \text{otherwise} \end{cases}$$

Again, it follows from the definition of morphisms that this indeed defines a run of α in $\mathcal{T}$.

It is easily shown that the correspondence given above is one to one.

□

6.3 Timed Bisimulation

Given our categories of timed transition systems and paths, we can now apply the general framework from [95], defining notions of open maps and bisimulation.

Definition 6.9 (Open Map [95]) *A morphism $\mathcal{T} \xrightarrow{(m,\eta)} \mathcal{T}'$ in* CTTS_Σ *is $\mathcal{TW}$-open iff for all timed words α and α', and morphisms such that the following diagram commutes:*

$$\begin{tikzcd} \mathcal{T}_\alpha \arrow[r, "(p,\eta_p)"] \arrow[d, "(f,\eta_f)"'] & \mathcal{T} \arrow[d, "(m,\eta)"] \\ \mathcal{T}_{\alpha'} \arrow[r, "(q,\eta_q)"] & \mathcal{T}' \end{tikzcd}$$

there exists a morphism $(p', \eta_{p'}) : \mathcal{T}_{\alpha'} \to \mathcal{T}$ such that the in the diagram

$$\begin{tikzcd} \mathcal{T}_\alpha \arrow[r, "(p,\eta_p)"] \arrow[d, "(f,\eta_f)"'] & \mathcal{T} \arrow[d, "(m,\eta)"] \\ \mathcal{T}_{\alpha'} \arrow[r, "(q,\eta_q)"] \arrow[ur, "(p',\eta_{p'})"] & \mathcal{T}' \end{tikzcd}$$

the two triangles commute.

Definition 6.10 *Two timed transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$ are $\mathcal{TW}$-bisimilar iff there exists a span $\mathcal{T}_1 \xleftarrow{(m,\eta)} \mathcal{T} \xrightarrow{(m',\eta')} \mathcal{T}_2$ with vertex $\mathcal{T}$ of $\mathcal{TW}$-open morphisms.*

Example 6.5 *In Figure 6.4 the (only) two morphisms from $\mathcal{T}$ to $\mathcal{T}'$ are open. We leave it for the reader to check that this is indeed the case.*

Notice that it follows from [95] and Theorem 6.3 that $\mathcal{TW}$-bisimulation is exactly the equivalence generated by $\mathcal{TW}$-open maps. Our next aim is to characterize $\mathcal{TW}$-open morphisms.

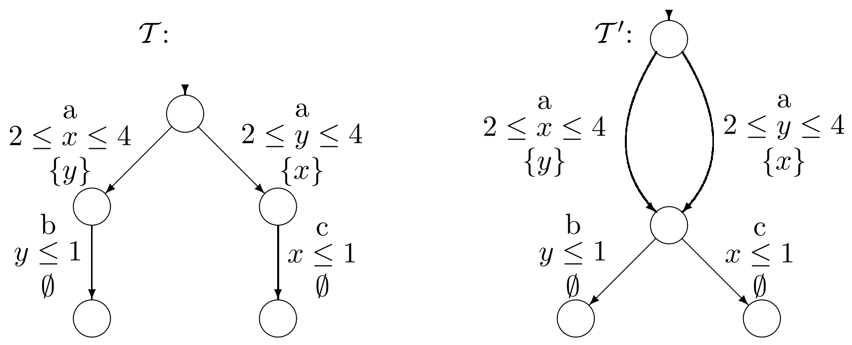


Figure 6.4: Two bisimilar timed transition systems.

Definition 6.11 *Given a timed transition system $\mathcal{T}$, a configuration $\langle s, \nu\rangle$ of $\mathcal{T}$ is reachable iff $\mathcal{T}$ has a run with an occurrence of $\langle s, \nu\rangle$.*

Theorem 6.5 *A morphism $\mathcal{T}_1 \xrightarrow{(m,\eta)} \mathcal{T}_2$ is open iff for all reachable configurations $\langle s_1, \nu\rangle$ in $\mathcal{T}_1$, and for all $\nu' = \nu + \tau$ whenever there is a transition $m(s_1) \xrightarrow[\delta_2,\lambda_2]{\sigma} s_2'$ such that $[\![\eta^{-1}(\nu')]\!]_{X_2} \in [\![\delta_2]\!]_{X_2}$, then there exists a transition $s_1 \xrightarrow[\delta_1,\lambda_1]{\sigma} s_1'$ such that $m(s_1') = s_2'$, $[\![\nu']\!]_{X_1} \in [\![\delta_1]\!]_{X_1}$, and $\lambda_2 = \eta^{-1}(\lambda_1)$.*

Proof

Assume $\mathcal{T}_1 \xrightarrow{(m,\eta)} \mathcal{T}_2$ is open, and that the configuration $\langle s_1, \nu\rangle$ is reachable in $\mathcal{T}_1$, i.e. we have a run of some timed word α ending in $\langle s_1, \nu\rangle$. From the assumptions of the theorem the (m, η)-image of this run in $\mathcal{T}_2$ may be extended by some σ-timed transition $\langle m(s_1), \eta^{-1}(\nu)\rangle \xrightarrow[\tau']{\sigma} \langle s_2', \eta^{-1}[\lambda_2 \mapsto 0]\rangle$. Hence we have a commuting diagram with $\alpha' = \alpha(\sigma, \tau')$

$$\begin{tikzcd}
\mathcal{T}_\alpha \arrow[r, "(q,\eta_q)"] \arrow[d] & \mathcal{T}_1 \arrow[d, "(m,\eta)"] \\
\mathcal{T}_{\alpha'} \arrow[r, "(q',\eta_{q'})"] & \mathcal{T}_2
\end{tikzcd}$$

From the definition of openness we get a mediating morphism

$$\begin{tikzcd}
\mathcal{T}_\alpha \arrow[r, "(q,\eta_q)"] \arrow[d] & \mathcal{T}_1 \arrow[d, "(m,\eta)"] \\
\mathcal{T}_{\alpha'} \arrow[r, "(q',\eta_{q'})"'] \arrow[ur, dashed, "(p,\eta_p)"] & \mathcal{T}_2
\end{tikzcd}$$

From this diagram, it follows from Theorem 6.1 and Theorem 6.4 that there exists a transition $s_1 \xrightarrow[\delta_1,\lambda_1]{\sigma} s_1'$ such that $m(s_1') = s_2'$, $[\![\nu']\!]_{X_1} \in [\![\delta_1]\!]_{X_1}$, and $\lambda_2 = \eta^{-1}(\lambda_1)$.

For the if part of the theorem, assume the we have a commuting square

$$\begin{tikzcd}
\mathcal{T}_\alpha \arrow[r, "(q,\eta_q)"] \arrow[d] & \mathcal{T}_1 \arrow[d, "(m,\eta)"] \\
\mathcal{T}_{\alpha'} \arrow[r, "(q',\eta_{q'})"] & \mathcal{T}_2
\end{tikzcd}$$

In the following we assume that $\alpha' = \alpha(\sigma, \tau')$, i.e. that α' is an extension of of

α by a single timed action. The general case follows from repeated applications of the arguments in the following.

From Theorem 6.4, the morphism (q, η_q) defines a run of α in $\mathcal{T}_1$ ending in some configuration $\langle s_1, \nu \rangle$, mapped by (m, η) to $\langle m(s_1), \eta^{-1}(\nu) \rangle$. Now, $(q', \eta_{\ q'})$ implies that there is some transition $m(s_1) \xrightarrow[\delta_2, \lambda_2]{\sigma} s_2'$ in $\mathcal{T}_2$, such that $[\![\eta^{-1}(\nu')]\!]_{X_2} \in [\![\delta_2]\!]_{X_2}$, where $\nu' = \nu + \tau$ for some τ determined by α'. From the assumptions of the theorem, we now get that there exists a transition $s_1 \xrightarrow[\delta_1, \lambda_1]{\sigma} s_1'$, such that $m(s_1') = s_2'$, $[\![\nu']\!]_{X_1} \in [\![\delta_1]\!]_{X_1}$, and $\lambda_2 = \eta^{-1}(\lambda_1)$. Using Theorem 6.4 this implies the existence of a morphism from $\mathcal{T}_{\alpha'}$ to $\mathcal{T}_1$, for which the commutativity properties required by openness follows by the properties listed above.

□

The standard notion of timed bisimulation is defined in terms of configurations as follows.

Definition 6.12 (Timed Bisimulation [49, 18]) *Two timed transition systems are bisimilar iff there exists a relation R over configurations $(\langle s, \nu_s \rangle, \langle t, \nu_t \rangle)$ of the two systems satisfying $(\langle s^{in}, \nu_s^0 \rangle, \langle t^{in}, \nu_t^0 \rangle) \in R$ and for all $(\langle s, \nu_s \rangle, \langle t, \nu_t \rangle) \in R$*

- *whenever $\langle s, \nu_s \rangle \xrightarrow[\tau]{\sigma} \langle s', \nu_s' \rangle$ then $\langle t, \nu_t \rangle \xrightarrow[\tau]{\sigma} \langle t', \nu_t' \rangle$ with $(\langle s', \nu_s' \rangle, \langle t', \nu_t' \rangle) \in R$ for some $\langle t', \nu_t' \rangle$.*
- *whenever $\langle t, \nu_t \rangle \xrightarrow[\tau]{\sigma} \langle t', \nu_t' \rangle$ then $\langle s, \nu_s \rangle \xrightarrow[\tau]{\sigma} \langle s', \nu_s' \rangle$ with $(\langle s', \nu_s' \rangle, \langle t', \nu_t' \rangle) \in R$ for some $\langle s', \nu_s' \rangle$.*

Theorem 6.6 *Two timed transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$ are $\mathcal{TW}$-bisimilar iff they are bisimilar according to Definition 6.12.*

Proof Assume $\mathcal{T}_1$ and $\mathcal{T}_2$ to be $\mathcal{TW}$-bisimilar with span of open maps

$$\begin{tikzcd} & \mathcal{T} \arrow[dl, "(m_1,\eta_1)"'] \arrow[dr, "(m_2,\eta_2)"] & \\ \mathcal{T}_1 & & \mathcal{T}_2 \end{tikzcd}$$

Define $\mathcal{R}$ to be the following relation of configurations of $\mathcal{T}_1$ and $\mathcal{T}_2$:

$\langle s_1, \nu_1 \rangle \mathcal{R} \langle s_2, \nu_2 \rangle$ iff
there exists a reachable configuration $\langle s, \nu \rangle$ of $\mathcal{T}$ such that $s_i = m_i(s)$ and $\nu_i = \eta_i^{-1}(\nu)$ for $i = 1, 2$.

It follows easily from Theorem 6.5 that $\mathcal{R}$ satisfies the required properties of Definition 6.12.

Assume $\mathcal{T}_1$ and $\mathcal{T}_2$ to be bisimilar with relation $\mathcal{R}$ as defined in Definition 6.12. We construct a span of open maps with vertex $\mathcal{T}$ defined as follows.

The states of $\mathcal{T}$ will be pairs of "$\mathcal{R}$-related runs" of $\mathcal{T}_1$ and $\mathcal{T}_2$ - formally defined as follows.

Two runs of a timed word $\alpha = (\sigma_1, \tau_1)(\sigma_2, \tau_2) \dots (\sigma_n, \tau_n), n \geq 0$ in $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively

$$\langle s_i^0, \nu_i^0 \rangle \underset{\tau_1}{\overset{\sigma_1}{\longrightarrow}} \langle s_i^1, \nu_i^1 \rangle \underset{\tau_2}{\overset{\sigma_2}{\longrightarrow}} \dots \underset{\tau_n}{\overset{\sigma_n}{\longrightarrow}} \langle s_i^n, \nu_i^n \rangle, i = 1, 2 \tag{6.3}$$

are said to be $\mathcal{R}$-related iff

$$\langle s_1^j, \nu_1^j \rangle \mathcal{R} \langle s_2^j, \nu_2^j \rangle \text{for } 0 \leq j \leq n$$

The initial state of $\mathcal{T}$ is the pair of initial configurations of $\mathcal{T}_1$ and $\mathcal{T}_2$.

The clock variables of $\mathcal{T}$ will be the disjoint union of the clock variables of $\mathcal{T}_1$ and $\mathcal{T}_2$, $X_1 \uplus X_2$.

Finally, for each pair of $\mathcal{R}$-related runs of the form (6.3), there will be an incoming transition in $\mathcal{T}$ from the pair of $\mathcal{R}$-related runs of ending in $(\langle s_1^{n-1}, \nu_1^{n-1} \rangle, \langle s_2^{n-1}, \nu_2^{n-1} \rangle)$ of the form $\xrightarrow[\delta,\lambda]{\sigma_n}$, where

$$\delta = \bigwedge_{x \in X_i, i=1,2} (x = \nu_i^{n-1}(x) + (\tau_n - \tau_{n-1}))$$

$$\lambda = \{x_i \in X_i \mid i = 1, 2 \text{ and } \nu_i^n(x_i) = 0\}$$

The open morphisms from $\mathcal{T}$ to $\mathcal{T}_i$ is $(m_i, \eta_i) : \mathcal{T} \to \mathcal{T}_i,\ i = 1, 2$ where the m_i-value on a pair of $\mathcal{R}$-related runs as in (6.3) is taken to be s_i^n, and η_i is the injection function from X_i to $X_1 \uplus X_2$. It follows from the construction that (m_i, η_i) are morphisms, and openness follows from Theorem 6.5.

□

Example 6.6 *Consider the timed transition systems in Figure 6.5. It is easy to see that there is exactly one morphism from $\mathcal{T}$ to $\mathcal{T}_i$, for $i = 1, 2$, and that this morphism is open. Hence, we have a span of open maps between $\mathcal{T}_1$ and $\mathcal{T}_2$ (with $\mathcal{T}$ as vertex), and bisimilarity between $\mathcal{T}_1$ and $\mathcal{T}_2$ follows from Theorem 6.6.*

Notice that there are simple arguments following Theorem 6.5 for openness of the morphisms in the example above. Hence we suggest spans of open maps as a convenient framework for presentations of bisimilarity of finite timed transition systems. In the next section this will be supported by two decidability results: openness of morphisms and bisimilarity for finite timed transition systems.

6.4 Decidability

In this section we restrict ourselves to finite timed transition systems, i.e. systems with a finite number of states, clocks and transitions, and for which all constants referred to in constraints have rational values. By scaling the rational constants we assume without loss of generality in the following that all constants are integer valued [13].

To get a decidable characterization of openness we introduce the notion of regions, [13].

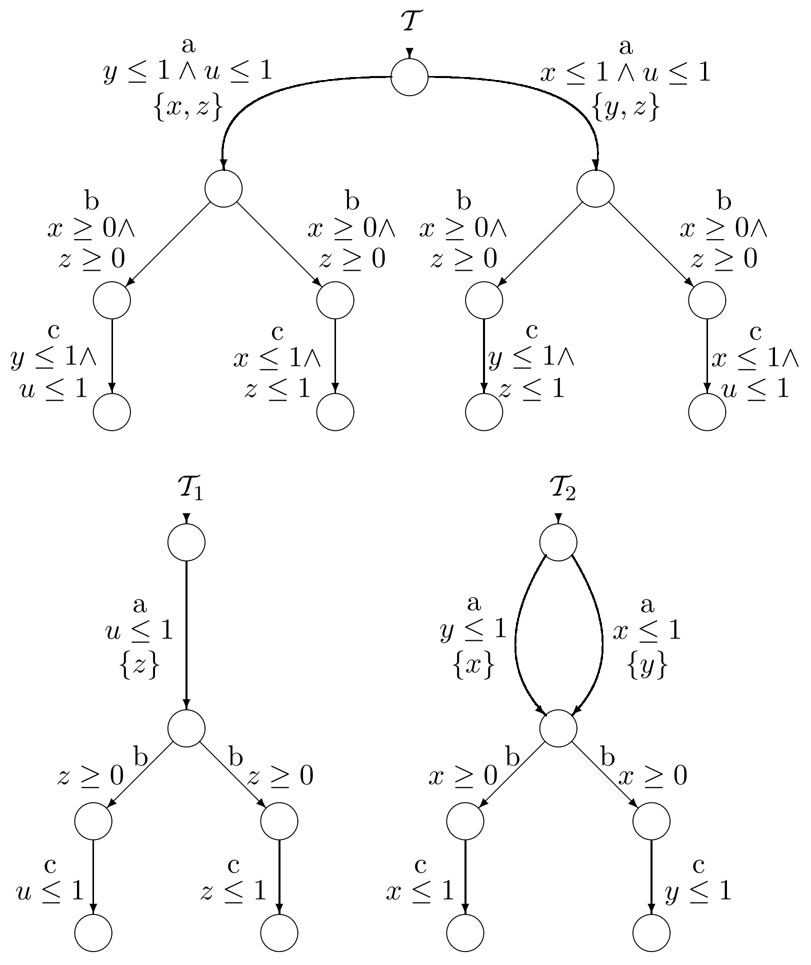


Figure 6.5: Three systems with a span.

Definition 6.13 (Region [13]) *Given a finite set of clock variables X and an integer constant c, a region is an equivalence class of the equivalence relation $\cong$ over clock valuations, where $\nu \cong \nu'$ iff*

- *For each $x \in X$: $\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor$*[2] *or both $\nu(x) > c$ and $\nu'(x) > c$.*
- *For every pair of clock variables $x, y \in X$ where both $\nu(x) \leq c$ and $\nu(y) \leq c$ we have that $fract(\nu(x)) \leq fract(\nu(y))$ iff $fract(\nu'(x)) \leq fract(\nu'(y))$.*
- *For every clock variable $x \in X$ where $\nu(x) \leq c$ we have $fract(\nu(x)) = 0$ iff $fract(\nu'(x)) = 0$.*

For a clock valuation ν, let $[\nu]$ denote the region to which it belongs. Let $\mathcal{R}_{X,c}$ denote the (finite) set of regions associated with X and c. Given regions $reg, reg' \in \mathcal{R}_{X,c}$, $reg' \in \mathrm{Reach}(reg)$ iff there exists $\nu \in reg$ and $\tau \in \mathsf{R}_+$ such that $\nu + \tau \in reg'$. Finally, for a finite timed transition system $\mathcal{T}$ an extended

[2]We use $\lfloor x \rfloor$ for the largest integer smaller than or equal to x and $fract(x) := x - \lfloor x \rfloor$.

state is defined as any pair $\langle s, reg\rangle$, where s is a state of $\mathcal{T}$ and reg is a region over the set of clock variables of $\mathcal{T}$.

Proposition 6.3 *Consider finite timed transition systems $\mathcal{T}$ and $\mathcal{T}'$ with clock variables X and X' respectively, and let c be an integer constant greater than or equal to the largest constant referred to in transition constraint expressions in $\mathcal{T}$ and $\mathcal{T}'$.*

For any $\mathcal{T}$-constraint expression δ and any region $reg \in \mathcal{R}_{X,c}$, $[\![reg]\!]_X \subseteq [\![\delta]\!]_X$ iff $[\![reg]\!]_X \cap [\![\delta]\!]_X \neq \emptyset$. For any $reg' \in \mathcal{R}_{X,c}$, and any $\nu, \nu' \in reg$, reg' is reachable from ν iff it is reachable from ν'.

Consider a morphism (m, η) from $\mathcal{T}$ to $\mathcal{T}'$ with $reg, reg' \in \mathcal{R}_{X,c}$, then

- $\eta^{-1}(reg) \in \mathcal{R}_{X',c}$
- *if* $reg' \in \mathrm{Reach}(reg)$ *then* $\eta^{-1}(reg') \in Reach(\eta^{-1}(reg))$

Proof First two properties follow from e.g. [13]. The regional properties of morphisms follow by simple calculation.

□

Our operations on clock evaluations can be extended to regions which will be used below. We can now give a characterization of open maps in terms of extended states.

Theorem 6.7 *Consider finite timed transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$ with clock variables X_1 and X_2 respectively, and associated regions defined with respect to some integer constant greater than or equal to the largest constant referred to in transition constraint expressions in $\mathcal{T}_1$ and $\mathcal{T}_2$. A morphism $(m, \eta) : \mathcal{T}_1 \to \mathcal{T}_2$ is open iff for all reachable extended states $\langle s_1, reg\rangle$ in $\mathcal{T}_1$, and for all $reg' \in Reach(reg)$, whenever there is a transition $m(s_1) \xrightarrow[\delta_2,\lambda_2]{\sigma} s_2'$ such that $[\![\eta^{-1}(reg')]\!]_{X_2} \subseteq [\![\delta_2]\!]_{X_2}$, then there exists a transition $s_1 \xrightarrow[\delta_1,\lambda_1]{\sigma} s_1'$ such that $m(s_1') = s_2'$, $\lambda_2 = \eta^{-1}(\lambda_1)$ and $[\![reg']\!]_{X_1} \subseteq [\![\delta_1]\!]_{X_1}$.*

Proof Follows from Theorem 6.5 and Proposition 6.3.

□

Notice that Theorem 6.7 implies the following decidability result of openness of a morphism between two finite timed transition systems.

Theorem 6.8 *Given two finite timed transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$ and a morphism $(m, \eta) : \mathcal{T}_1 \to \mathcal{T}_2$, openness of (m, η) is decidable.*

Proof Follows immediately form Theorem 6.7 and Proposition 6.3.

□

For untimed transition systems, decidability of bisimulation follows e.g. from the fact that a span of open maps between two finite transition systems imply a span with a vertex being a subsystem of their product, see [95]. Unfortunately, this result does not generalize completely to our setting here. However, we still have the following.

Theorem 6.9 *Given two finite timed transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$, if there exists a span of open maps*

$$\begin{tikzcd}
& \mathcal{T} \arrow[dl, "(m{,}\eta)"'] \arrow[dr, "(m'{,}\eta')"] & \\
\mathcal{T}_1 & & \mathcal{T}_2
\end{tikzcd}$$

then there is a finite vertex $\mathcal{T}\times_{\mathcal{R}}$ of size bounded by the size of $\mathcal{T}_1$ and $\mathcal{T}_2$ and with open morphisms

$$\begin{tikzcd}
& \mathcal{T} \arrow[ddl, "(m{,}\eta)"'] \arrow[ddr, "(m'{,}\eta')"] & \\
& \mathcal{T}\times_{\mathcal{R}} \arrow[dl, "(p{,}\eta_p)"] \arrow[dr, "(q{,}\eta_q)"'] & \\
\mathcal{T}_1 & & \mathcal{T}_2
\end{tikzcd}$$

Proof Assume without loss of generality that the clock variables of $\mathcal{T}_1$ and $\mathcal{T}_2$ are disjoint. If ν and ν' are clock evaluations for $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively we shall write $\nu \uplus \nu'$ for the combined clock evaluation over the disjoint union of the clock variables of $\mathcal{T}_1$ and $\mathcal{T}_2$, satisfying $(\nu \uplus \nu') := \nu(x)$ if $x \in X_1$ and $(\nu \uplus \nu') := \nu'(x)$ if $x \in X_2$. Let c be an integer constant greater than or equal to the largest constant mentioned in transition constraint expressions in $\mathcal{T}_1$ and $\mathcal{T}_2$, and let all regions in the following be defined with respect to c. The timed transition system $\mathcal{T}\times_{\mathcal{R}}$ is defined in the following way.

- $\mathrm{S}\times_{\mathcal{R}}$ is the set of pairs $\langle s_1, s_2\rangle$ for which there exists a reachable configuration $\langle s, \nu\rangle$ in $\mathcal{T}$ such that $m(s) = s_1$ and $m'(s) = s_2$.
- The initial state $\mathrm{s}^{\mathrm{in}}\times_{\mathcal{R}}$ is $\langle s_1^{in}, s_2^{in}\rangle$ where s_1^{in} is the initial state of $\mathcal{T}_1$ and s_2^{in} the initial state of $\mathcal{T}_2$.
- $\mathrm{X}\times_{\mathcal{R}} = X_1 \uplus X_2$.
- The transitions of $\mathcal{T}\times_{\mathcal{R}}$ are defined as follows. For all runs in $\mathcal{T}$

  $$\langle s^{in}, \nu_0\rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, \nu_1\rangle \xrightarrow[\tau_2]{\sigma_2} \ldots \xrightarrow[\tau_n]{\sigma_n} \langle s, \nu\rangle$$

  with an extended run of the form

  $$\langle s^{in}, \nu_0\rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, \nu_1\rangle \xrightarrow[\tau_2]{\sigma_2} \ldots \xrightarrow[\tau_n]{\sigma_n} \langle s, \nu\rangle \xrightarrow[\tau]{\sigma} \langle s', \nu'\rangle$$

  we introduce a transition

  $$\langle m(s), m'(s)\rangle \xrightarrow[\hat{\delta},\hat{\lambda}]{\sigma} \langle m(s'), m'(s')\rangle$$

  in $\mathcal{T}\times_{\mathcal{R}}$, where $\hat{\lambda}$ consists of all clock variables x from $\mathcal{T}_1$ and x' from $\mathcal{T}_2$, for which ${\nu'}^{-1}(\eta(x))) = 0$ or ${\nu'}^{-1}(\eta'(x'))) = 0$, and $\hat{\delta}$ is the logical expression defining the region to which $\eta^{-1}(\nu + (\tau - \tau_n)) \uplus {\eta'}^{-1}(\nu + (\tau - \tau_n))$ belongs.

This completes the definition of $\mathcal{T}\times_{\mathcal{R}}$. Obviously, the size is bounded by the size of $\mathcal{T}_1$ and $\mathcal{T}_2$. The number of states is bounded by $|S_1| * |S_2|$. The number of regions over the disjoint union of X_1 and X_2 with constant c, is $|X|! * 2^{|X|} * (2c+2)^{|X|}$ where $|X| = |X_1| + |X_2|$, so there are at most $|\Sigma| * (|X|! * 2^{|X|} * (2c+2)^{|X|}) * 2^{|X|}$ transitions between any two states.

The morphisms from $\mathcal{T}\times_{\mathcal{R}}$ to $\mathcal{T}_1$ and $\mathcal{T}_2$ are the projections (p, η_p) and (q, η_q) respectively, where $p(\langle s_1, s_2\rangle) = s_1$ and similarly for q. The function η_p is the identity function on the clock variables of $\mathcal{T}_1$ and η_q is the identity function on the clock variables of $\mathcal{T}_2$. We need to verify that these are morphisms and that they are open. The proof for (p, η_p) will be shown here, and the arguments for (q, η_q) are symmetric.

To verify that the projection (p, η_p) is a morphism, consider a transition $\langle m(s), m'(s)\rangle \xrightarrow[\hat{\delta},\hat{\lambda}]{\sigma} \langle m(s'), m'(s')\rangle$ in $\mathcal{T}\times_{\mathcal{R}}$ as defined above. From definition and Theorem 6.1 this implies the existence in $\mathcal{T}_1$ of some transition $m(s) \xrightarrow[\delta_1,\lambda_1]{\sigma} m(s')$ realizing $\langle m(s), \eta^{-1}(\nu)\rangle \xrightarrow[\tau]{\sigma} \langle m(s'), \eta^{-1}(\nu')\rangle$, i.e. such that $[\![\eta^{-1}(\nu)+(\tau-\tau_n)]\!]_{X_1} \in [\![\delta_1]\!]_{X_1}$ and $\lambda_1 = \{x_1 \in X_1 \,|\, \eta^{-1}(\nu')(x_1) = 0\}$. This implies $[\![\eta_p^{-1}(\hat{\delta})]\!]_{X_1} \subseteq [\![\delta_1]\!]_{X_1}$, and hence $[\![\hat{\delta}]\!]_{X_\times} \subseteq [\![\eta^{-1}(\hat{\delta})[\eta_p(x)/x]]\!]_{X_\times} \subseteq [\![\delta_1[\eta_p(x)/x]]\!]_{X_\times}$, and $\lambda_1 = \eta_p^{-1}(\hat{\lambda})$.

To show that (p, η_p) is open, we show that it has the property from Theorem 6.7. Notice first that from construction, for any reachable extended state in $\mathcal{T}\times_{\mathcal{R}}$ of the form $(\langle s_1, s_2\rangle, reg)$ there exists a reachable configuration $\langle s, \nu\rangle$ in $\mathcal{T}$ such that $m(s) = s_1$, $m'(s) = s_2$, and $\eta^{-1}(\nu) \cup \eta'^{-1}(\nu) \in reg$.

Assume the extended state $\langle s_1, s_2, reg\rangle$ is reachable in $\mathcal{T}\times_{\mathcal{R}}$. Consider $reg' \in Reach(reg)$ and a transition $s_1 \xrightarrow[\delta_1,\lambda_1]{\sigma} s_1'$ in $\mathcal{T}_1$ for which $[\![\eta_p^{-1}(reg')]\!]_{X_1} \subseteq [\![\delta_1]\!]_{X_1}$. We must show the existence of a transition in $\mathcal{T}\times_{\mathcal{R}}$ of the form $\langle s_1, s_2\rangle \xrightarrow[\hat{\delta},\hat{\lambda}]{\sigma} \langle s_1', s_2'\rangle$, such that $[\![reg']\!]_{X_\times} \subseteq [\![\hat{\delta}]\!]_{X_\times}$ and $\lambda_1 = \eta_p^{-1}(\hat{\lambda})$.

Since $\langle s_1, s_2, reg\rangle$ is reachable we have a reachable configuration $\langle s, \nu\rangle$ in $\mathcal{T}$ such that $m(s) = s_1, m'(s) = s_2$, and $\eta^{-1}(\nu) \uplus \eta'^{-1}(\nu) \in reg$. Let τ be such that $\eta^{-1}(\nu+\tau) \uplus \eta'^{-1}(\nu+\tau) \in reg'$, and hence $[\![\eta^{-1}(\nu+\tau)]\!]_{X_1} \in [\![\eta_p^{-1}(reg')]\!]_{X_1} \subseteq [\![\delta_1]\!]_{X_1}$. We obtain from Theorem 6.5 the existence in $\mathcal{T}$ of a transition $s \xrightarrow[\delta,\lambda]{\sigma} s'$ such that $m(s') = s_1'$, $[\![\nu+\tau]\!]_X \in [\![\delta]\!]_X$ and $\lambda_1 = \eta^{-1}(\lambda)$. Hence from construction we have $\langle s_1, s_2\rangle \xrightarrow[\hat{\delta},\hat{\lambda}]{\sigma} \langle m(s'), m'(s')\rangle$, where $[\![\hat{\delta}]\!]_{X_\times} = [\![reg']\!]_{X_\times}$ and $\lambda_1 = \eta^{-1}(\lambda) = \eta_p^{-1}(\hat{\lambda})$.

□

From the proof of Theorem 6.9, we have the following corollary.

Corollary 6.1 *Given two finite timed transition systems, timed bisimulation is decidable.*

6.5 Extension with Invariants

In this section we will extend the timed transition systems with invariants [109] on the states and argue that the results from the preceding sections can be generalized to the extended model without problems. We will state the results for the new model and hints to some of the proofs, all of which are simple extensions of the proofs for the model without invariants on states.

Definition 6.14 (Timed Transition Systems with invariants) *A timed transition system with invariants is six tuple $(S, \Sigma, s_0, X, T, I)$ where the first five components are as in Definition 6.1 and* I *assigns to each state an invariant. Invariants are given by the same syntax as constraints, so the invariant for state* s*, ι_s, can be generated by the grammar Δ from Definition 6.1.*

The meaning of a invariant ι_s, $[\![\iota_s]\!]_X$, is defined in the same way as the meaning of a constraint. In the definition of runs over a timed transition system with invariants, the invariant of a state must be satisfied when the state is entered and until the next state is entered. More formally, in the definition of a run

$$\langle s_0, \nu_0 \rangle \xrightarrow[\tau_1]{\sigma_1} \langle s_1, \nu_1 \rangle \xrightarrow[\tau_2]{\sigma_2} \cdots \xrightarrow[\tau_n]{\sigma_n} \langle s_n, \nu_n \rangle$$

we require $\forall i \in \{0, 1, \ldots, n-1\}, \forall \tau \in [0, \tau_i - \tau_{i-1}) : [\![\nu_i + \tau]\!]_X \in [\![\iota_{s_i}]\!]_X$ where $\tau_0 = 0$, and for the last state $[\![\nu_n]\!]_X \in [\![\iota_{s_n}]\!]_X$. We define a new kind of morphism which is going to be an extension of the ones from Definition 6.5 taking invariants into account.

Definition 6.15 *A morphism (m, η) between two timed transition systems with invariants $\mathcal{T}$ and $\mathcal{T}'$ consists of the same components as the morphisms in Definition 6.5 with one extra constraint:*

- *If a state* s *in $\mathcal{T}$ is mapped by m to a state $m(s)$ in $\mathcal{T}'$ then $[\![\iota_s]\!]_X \subseteq [\![\iota_{m(s)}[\eta(x)/x]]\!]_X$.*

This definition ensures that if an invariant is satisfied in some configuration in $\mathcal{T}$ the invariant of the simulating configuration is also satisfied. This implies that we still have morphisms as simulations as stated in Theorem 6.1.

With this notion of morphisms we have a category as in Definition 6.7, which we denote $\mathrm{CTTS}^{\iota}_{\Sigma}$.

Proposition 6.4 $\mathrm{CTTS}^{\iota}_{\Sigma}$ *has products and pullbacks.*

The construction of $\mathcal{T}_1 \times_{\mathcal{T}} \mathcal{T}_2$ follows the one in the proof of Theorem 6.3 where the invariant of the state $\langle s_1, s_2 \rangle$ is defined such that $[\![\iota_{\langle s_1, s_2 \rangle}]\!]_{X_\times} = [\![\iota_{s_1}]\!]_{X_\times} \cap [\![\iota_{s_2}]\!]_{X_\times}$. The invariants on the states in the product is defined in the same way.

As our category of computations we would again like to choose timed words over Σ with word extensions. Like for timed transition systems we choose a representation of these in terms of our models following the approach of [95]. This is going to look very much like the representation we defined for timed transition systems, we just need to add invariants to all the states.

Definition 6.16 *Given a timed word $\alpha = (\sigma_1, \tau_1)\ (\sigma_2, \tau_2)\ (\sigma_3, \tau_3) \cdots (\sigma_n, \tau_n)$ we define a timed transition system $\mathcal{T}^{\iota}_{\alpha}$ $0 \xrightarrow[\delta_1, \lambda_1]{\sigma_1} 1 \xrightarrow[\delta_2, \lambda_2]{\sigma_2} \cdots \xrightarrow[\delta_n, \lambda_n]{\sigma_n} n$, as in Definition 6.8, where the invariants are defined inductively to be of the form $\bigwedge_{x \in X}(c_x \leq x < c'_x)$. The initial the invariant is*

$$\bigwedge_{x \in X} (0 \leq x < \tau_1)$$

Assume the invariant on the state i-1 is $\bigwedge_{x\in X}(c_x^{i-1} \leq x < \tilde{c}_x^{i-1})$, *then the invariant for state i is*

$$\bigwedge_{x\in X}(\textit{if } x \in \lambda_i \textit{ then } (0 \leq x < \tilde{\tau}_i) \textit{ else } (\tilde{c}_x^{i-1} \leq x < \tilde{c}_x^{i-1} + \tilde{\tau}_i))$$

where $\tilde{\tau}_i = \tau_i - \tau_{i-1}$. *The constraint on the final state is*

$$\bigwedge_{x\in X}(\textit{if } x \in \lambda_i \textit{ then } (x = 0) \textit{ else } (x = \tilde{c}_x^{i-1}))$$

Using this construction we still get an embedding of the category of timed words with extensions into $\mathrm{CTTS}^{\iota}_{\Sigma}$, with properties as in Theorem 6.4.

The characterization of open maps is a little more complicated to state with the invariants. The proof though is again just a simple extension of the proof of Theorem 6.5 using the constraint for the invariants and the condition for runs.

Proposition 6.5 *A morphism* $(m, \eta) : \mathcal{T}_1 \to \mathcal{T}_2$ *is open iff for all reachable configurations* $\langle s_1, \nu\rangle$ *for all* $\nu' = \nu+\tau$ *such that* $\forall \tau' : \tau' < \tau \Rightarrow [\![\eta^{-1}(\nu+\tau')]\!]_{X_2} \in [\![\iota_{m(s_1)}]\!]_{X_2}$ *whenever there is a transition* $m(s_1) \xrightarrow[\delta_2,\lambda_2]{\sigma} s_2'$ *if* $[\![\overline{\nu}]\!]_{X_2} \in [\![\delta_2]\!]_{X_2}$ *and* $[\![\overline{\nu}]\!]_{X_2} \in [\![\iota_{s_2'}]\!]_{X_2}$ *for* $\overline{\nu} = \eta^{-1}(\nu')$ *then there exists a transition* $s_1 \xrightarrow[\delta_1,\lambda_1]{\sigma} s_1'$ *such that* $m(s_1') = s_2'$, $[\![\nu']\!]_{X_1} \in [\![\delta_1]\!]_{X_1}$, $\lambda_2 = \eta^{-1}(\lambda_1)$, *and* $\forall \tau' < \tau : [\![\nu + \tau']\!]_{X_1} \in [\![\iota_{s_1}]\!]_{X_1}$, $[\![\nu']\!]_{X_1} \in [\![\iota_{s_2}]\!]_{X_1}$.

We also have a characterization in terms of extended states equivalent to Theorem 6.7, using the property that if one clock evaluation in a region satisfies an invariant then all the clock evaluations of that region satisfy the invariant. Given this, the proof of the theorem for extended states follows directly from the proof of Theorem 6.7. With the characterization of the open maps in terms of extended states, we again have the decidability of openness for morphisms between finite timed transition systems with invariants, and we can construct the finite vertex if such one exists.

Theorem 6.10 *Given to finite timed transition systems with invariants* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *if there exists a span of open maps*

$$\begin{tikzcd} & \mathcal{T} \arrow[dl, "{(m,\eta)}"'] \arrow[dr, "{(m',\eta')}"] & \\ \mathcal{T}_1 & & \mathcal{T}_2 \end{tikzcd}$$

then there is a finite $\mathcal{T}\times_{\mathcal{R}}$ *giving a span of open maps*

$$\begin{tikzcd} & \mathcal{T} \arrow[ddl, "{(m,\eta)}"'] \arrow[ddr, "{(m',\eta')}"] & \\ & \mathcal{T}\times_{\mathcal{R}} \arrow[dl, "{(p,\eta_p)}"] \arrow[dr, "{(q,\eta_q)}"'] & \\ \mathcal{T}_1 & & \mathcal{T}_2 \end{tikzcd}$$

The construction of $\mathcal{T}\times_{\mathcal{R}}$ is almost the same as in the proof of Theorem 6.10. The meaning of the invariant of the state $\langle s_1, s_2\rangle$ is the intersection $[\![\iota_{s_1}]\!]_{X_\times} \cap [\![\iota_{s_2}]\!]_{X_\times}$ as in the construction used for the pullback.

Again the decidability of the bisimulation follows directly from the construction of the vertex and the decidability of openness.

6.6 Conclusion

We have illustrated how to apply the general framework of open maps to the setting of timed systems, providing a way of expressing a bisimulation purely within the framework of timed transition systems. Furthermore, a decision procedure for bisimulation was presented within this framework.

We propose the span of open maps idea as a useful way of expressing timed bisimulations for finite systems. On the other hand, we do not claim that our alternative decision procedure as presented here is more efficient than existing ones, e.g. [103, 143].

The categorical formulations in terms of open maps suggest applying general results from the categorical setting to concrete timed bisimulations, like the one studied here. One particularly interesting example is the characteristic path logic obtained from [95]. It would be interesting to study this logic and its relation to existing timed logics from the literature.

Part III

Timed Reachability Analysis

Chapter 7

Distributing Uppaal

The paper *Distributing Timed Model Checking – How the Search Order Matters* presented in this chapter has been published as a conference paper [30]. A technical report is in preparation, however, some experiments needs to be finished before the technical report can be finalized.

[30] G. Behrmann, T. Hune, and F. Vaandrager. Distributing Timed Model Checking – How the Search Order Matters. In *Proc. Computer Aided Verification, CAV 2000*, pages 216–231, 2000.

This chapter extends the conference paper with more explanation of the algorithm and how the search order can influence the state-space. Also a new termination algorithm is presented. Except from some new experiments this will be the content of the coming technical report.

Distributing Timed Model Checking — How the Search Order Matters[§]

Gerd Behrmann[*] Thomas Hune[†] Frits Vaandrager[‡]

Abstract

In this paper we address the problem of distributing model checking of timed automata. We demonstrate through four real life examples that the combined processing and memory resources of multi-processor computers can be effectively utilized. The approach assumes a distributed memory model and is applied to both a network of workstations and a symmetric multiprocessor machine. However, certain unexpected phenomena have to be taken into account. We show how in timed model checking the search order of the state-space is crucial for the effectiveness and scalability of the exploration. An effective heuristic to counter the effect of the search order is provided. Some of the results open up for improvements in the single processor case.

7.1 Introduction

The technical challenge in model checking is in devising algorithms and data-structures that allow one to handle large state-spaces. Over the last two decades numerous approaches have been developed that address this problem: symbolic methods such as BDDs, methods that exploit symmetry, partial order reduction techniques, etc [53]. One obvious approach that has been applied successfully by a number of researchers is to parallelize (or distribute) the state-space search [7, 138]. Distributed reachability analysis and state-space generation has also been investigated in the related field of performance analysis in the context of stochastic Petri nets [47, 71] (see the second paper for further references). Since the state-of-the-art in model checking and performance analysis is still progressing very fast, it does not make sense to develop parallel or distributed tools from scratch. Rather, the goal should be to view parallelization as an orthogonal feature, which can always be easily added when the appropriate hardware is available.

[*]**B**asic **R**esearch **I**n **C**omputer **S**cience, BRICS, (www.brics.dk), funded by the Danish National Research Foundation, University of Aalborg, Denmark, Email: `behrmann@cs.auc.dk`

[†]**B**asic **R**esearch **I**n **C**omputer **S**cience, BRICS, (www.brics.dk), funded by the Danish National Research Foundation, University of Århus, Denmark, Email: `baris@brics.dk`

[‡]Computing Science Institute, University of Nijmegen, Email: `fvaan@cs.kun.nl`

[§]Research supported by Esprit Project 26270, Verification of Hybrid Systems (VHS).

To some extent this goal has been achieved in the work of [47, 138, 71], all with very similar solutions. Stern and Dill [138], for example, present a simple but elegant approach to parallelize the Murφ tool [61] using the message passing paradigm. In parallel Murφ, the state table, which stores all reached protocol states, is partitioned over the nodes of the parallel machine. Each node maintains a work queue of unexplored states. When a node generates a new state, the *owning* node for this state is calculated with a hash function and the state is sent to this node; this policy implements randomized load balancing. In the case of Murφ, the algorithm of Stern and Dill achieves close to linear speedup. We applied the approach of Stern and Dill to parallelize UPPAAL [109], a model checker for networks of extended timed automata. We experimented with parallel UPPAAL using four existing case studies: DACAPO [115], communication [70] and power-down [69] protocols used in B&O audio/video equipment, and a model of a buscoupler.

In the case of timed automata, the state-space is uncountably infinite, and therefore one is forced to work with *symbolic* states, which are finite representations of possibly infinite sets of concrete states. A key problem we had to face in our work is that the number of symbolic states that has to be explored depends on the order in which the state exploration proceeds. In particular, the number of states tends to grow if state-space exploration is parallelized. The main contribution of this paper consists an effective heuristic which takes care that the growth of the number of states remains within acceptable bounds. As a result we manage to obtain super linear speedups for the B&O protocols and the buscoupler. For the DACAPO example the speedup is not so good, probably because the state-space is so small that only a few nodes are involved in the computation at a time. Some of the results open up for improvements in the single processor case.

The rest of this paper is structured as follows: Section 7.2 reviews the notion of timed automata. Section 7.3 describes our approach to distributed timed model checking, Section 7.4 presents experimental results, and Section 7.5 summarizes some of the conclusions.

7.2 Model Checking Timed Automata

In this section we briefly review the notion of timed automata that underlies the UPPAAL tool. For a more extensive introduction we refer to [13, 99]. For reasons of simplicity and clarity in presentation we have chosen to only give the semantics and exploration algorithms for timed automata. The techniques described in this paper extend easily to networks of timed automata, even when extended with shared variables as is the case in UPPAAL.

Timed automata are finite automata extended with real-valued clocks. Figure 7.1 depicts a simple two node timed automaton. As can be seen both the locations and edges can be labeled with constraints on the clocks. Given a set of clocks C, we use $\mathcal{B}(C)$ to stand for the set of formulas that are conjunctions of atomic constraints of the form $x \bowtie n$ and $x - y \bowtie n$ for $x, y \in C$, $\bowtie \in \{<, \leq, =, \geq, >\}$ and n being a natural number. Elements of $\mathcal{B}(C)$ are

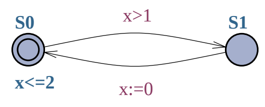


Figure 7.1: A simple two state timed automaton with a single clock x.

called clock constraints over C. $\mathcal{P}(C)$ denotes the power set of C.

Definition 7.1 *A timed automaton A over clocks C is a tuple (L, l_0, E, I) where L is a finite set of locations, l_0 is the initial location, $E \subseteq L \times \mathcal{B}(C) \times \mathcal{P}(C) \times L$ is the set of edges, and $I : L \to \mathcal{B}(C)$ assigns invariants to locations. In the case of $(l, g, r, l') \in E$, we write $l \xrightarrow{g,r} l'$.*

Formally, clock values are represented as functions called clock assignments from C to the non-negative reals $\mathbf{R}_{\geq 0}$. We denote by $\mathbf{R}^C$ the set of clock assignments for C. The semantics of a timed automaton is defined in terms of a labeled transition system with states $L \times \mathbf{R}^C$, and initial state (l_0, u_0) where u_0 assigns zero to all clocks. The transition relation is given by:

- $(l, u) \to (l, u + d)$ if $u \in I(l)$ and $u + d \in I(l)$
- $(l, u) \to (l', u')$ if there exist g and r s.t. $l \xrightarrow{g,r} l', u \in g$, $u' = [r \mapsto 0]u$, and $u' \in I(l')$

where for $d \in \mathbf{R}$, $u + d$ maps each clock x in C to the value $u(x) + d$, and $[r \mapsto 0]u$ denotes the assignment for C which maps each clock in r to the value 0 and agrees with u over $C \setminus r$. In short, the first rule describes *delay* and the second *edge* transitions. It is easy to see that the state-space is uncountable. However, it is a well-known fact that timed automata have a finite-state symbolic semantics [13] based on countable symbolic states of the form $L \times \mathcal{B}(C)$. The initial state is $(l_0, D_0^{\uparrow} \wedge I(l_0))$, where D_0 represents that all the clocks have value zero. The transition relation for the symbolic semantics is given by:

- $(l, D) \to (l, \mathrm{norm}(M, (D \wedge I(l))^{\uparrow} \wedge I(l)))$
- $(l, D) \to (l', r(g \wedge D) \wedge I(l'))$ if $l \xrightarrow{g,r} l'$.

where $D^{\uparrow} = \{u + d \mid u \in D \wedge d \in \mathbf{R}_{\geq 0}\}$ (the *future* operation), and $r(D) = \{[r \mapsto 0]u \mid u \in D\}$. The function norm : $\mathbf{N} \times \mathcal{B}(C) \to \mathcal{B}(C)$ normalizes the clock constraints with respect to the maximum constant M of the timed automaton. Normalizing the clock constraints guarantees a finite state-space. We refer to [13, 99] for an in-depth treatment of the subject.

The state-space exploration algorithm is shown in Fig. 7.2. Central to the algorithm are two data-structures: the waiting list, which contains unexplored but reachable symbolic states, and the passed list, which contains all explored symbolic states (similar to the state table in Murφ). An important but in the literature often ignored optimization is to check for state coverage in both lists.

```
PASSED := $\emptyset$
WAITING := $\{(l_0, D_0)\}$
**repeat**
    get $(l, D)$ from WAITING
    **if for all** $(l, D') \in$ PASSED : $D \not\subseteq D'$ **then**
        add $(l, D)$ to PASSED
        SUCC := $\{(l', D') : (l, D) \to (l', D') \wedge D' \neq \emptyset\}$
        **for all** $(l', D') \in$ SUCC **do**
            put $(l', D')$ in WAITING
        **od**
    **end if**
**until** WAITING = $\emptyset$
```

Figure 7.2: Sequential symbolic state-space exploration for timed automata.

Instead of only checking whether a symbolic state is already included in the list, UPPAAL searches for states in the list that either cover the new state or is covered by it. In the first case the new state is discarded and in the latter case it replaces the existing state covered by it. We will return to this matter in Section 7.3.

7.3 Distributed Model Checking of Timed Automata

The approach we have used for distributing the exploration algorithm is similar to the one presented in [47, 138, 71]. The state-space is partitioned according to a distribution function h mapping symbolic states, (l, D), to nodes, i, resulting in a waiting list fragment, WAITING$_i$, and a passed list fragment, PASSED$_i$, on each node, i. Initially, all fragments are empty, except for WAITING$_{h(l_0,D_0)}$ which contains the initial state.

Each node executes an algorithm described by the pseudo-code in Fig. 7.3 (a variant of the sequential algorithm shown in Fig. 7.2). Compared to the sequential algorithm, the distributed algorithm reads exclusively from its local waiting list, but writes into all waiting lists according to the distribution function, h. Also, the algorithm needs to access all waiting lists in order to guarantee termination.

Since we assume a distributed memory model, all variables are local and non-local variables can only be accessed by passing messages between nodes. The passed list fragments are only accessed locally, but every node can add states to any waiting list fragment, hence requires a message containing the state to be sent to the respective node. Fortunately, this can be done asynchronously, i.e., without blocking the sender. Evaluating the termination condition is more difficult, since it requires knowledge about the size of all waiting lists *at the exact same point in time*. Worse, the network acts as buffer, i.e., even though all waiting lists are empty, any node might receive a state currently being transfered. The distributed termination algorithm proposed in Section 7.3.4 solves both problems without incurring any significant overhead.

```
Explore$_i$ =
    Passed$_i$ := $\emptyset$
    Waiting$_i$ := $\emptyset$
    **repeat**
        pick $(l, D)$ from Waiting$_i$
        **if for all** $(l, D') \in$ Passed$_i$ : $D \not\subseteq D'$ **then**
            add $(l, D)$ to Passed$_i$
            **for all** $(l', D') : (l, D) \to (l', D') \wedge D' \neq \emptyset$ **do**
                put $(l', D')$ into Waiting$_{h(l',D')}$
            **od**
        **end if**
    **until** $\sum_{i=1}^{n} |$Waiting$_i| = 0$
```

Figure 7.3: The distributed state-space exploration algorithm.

7.3.1 Nondeterminism and Search Orders

In general, communication delay, message retransmission, message ordering, and system load causes the search order of the distributed algorithm to be nondeterministic. Making it deterministic would require much more synchronization and is undesirable. This is different from the sequential algorithm, where the search order is deterministic and easily controllable. Here, choosing between breadth-first and depth-first amounts to using a queue or a stack implementation of the waiting list, respectively.

In the distributed setting, we only distinguish between nodes exploring states in *FIFO-order* and *FILO-order*, i.e., states received first are explored first and vice versa, respectively. The actual search order is crucially influenced by the choice of either FIFO-order or FILO-order on each node.

In FIFO-order the states are explored in order of arrival at each node. However, the order in which states arrive is nondeterministic. If the workload is distributed evenly and the number of nodes is low, the actual search order will resemble breadth-first order, but as the number on nodes increases the differences will become larger.

In FILO-order the states are explored in reverse order of arrival. To some extend, the resulting search order resembles depth-first, but many paths are explored simultaneously. In fact, any node can and will interrupt other nodes by sending states to them. Since we use FILO-order, arriving states will be explored first. Thus, each node will continuously jump to new paths depending on the received states. The exploration is highly sensitive to the order in which states are received. Assume two states α and β arrive at a node while the node is busy exploring a state, with α arriving last. The successors of α are generated and sent to their owning nodes. One or more of these may go to the local node itself which means that they are explored before β, and the same for their successors and so on. Thus, it can happen that β has to wait a long time before it is explored even though it has arrived at almost the same time as α. Hence small changes in the order of arrival of states may change the search order drastically.

The difference between the actual search orders caused by FIFO-order and FILO-order is similar to the difference between breadth-first order and depth-first order. States "deep" in the state-space (only reachable using a long path) will be searched faster using FILO-order than with FIFO-order. All states close to the initial state will be searched faster using FIFO-order.

7.3.2 Why the Search Order Matters

In a distributed state-space search the number of states explored (and thereby the work done) may differ from run to run. This is because whether a state is explored or not depends on the states previously encountered. As an example, consider two states (l, D) and (l, D') with the same location vector l but different time zones satisfying $D \subseteq D'$. If (l, D) is explored before arrival of (l, D'), then (l, D') will also be explored since it is not covered by any state in the passed list (assuming that there are no other states covering it). Since the successors of (l, D') are very likely to have larger time zones than the successors of (l, D) these will also be explored later. However, if (l, D') is explored before (l, D) arrives, then (l, D) will not be explored because it is covered by a state in the passed list. This also means that no successors of (l, D) will be generated or explored.

Example 7.1 *To illustrate how the search order can influence the state-space we will look at the automaton in Figure 7.4. Searching the state-space of the*

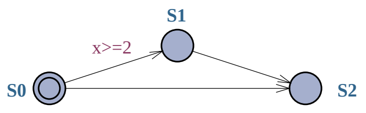


Figure 7.4: A timed automaton with one clock, x.

automaton in a depth-first order results in the state-space in Figure 7.5. States consist of a location part and a zone part. Since there is only one clock in the timed automaton the zones are one dimensional, represented by a shaded part of the x-axis. The number on the transitions show in what order the states will be searched. The initial symbolic state is the leftmost state in the figure consisting of the location S0 and the zone covering all of the x-axis. We have to define an ordering on the transitions from location S0 of the automaton to decide which transition should be explored first. The transition to location S1 have been chosen to be the first, so the symbolic state consisting of location S1 and the zone covering the x-axis from two to infinity is the second reachable state in the state-space. The part of the x-axis from zero to two is not included in the zone because of the guard $x \geq 2$ on the transition from location S0 to S1. Since we are doing a depth-first search we now explore the successors of this new state. There is only one which is the state consisting of location S2 and the same zone since the transition has no guards or invariants. This state has no successors, so we go back to the state with location S1. From this state there are no successors either, so we are back at the initial state. Here we have one transition more, from location S0 to location S2.Taking this transition *we can reach the symbolic state with location S2 and the same time zone as the initial state, since there are no guards or invariants. This state covers the already explored state with location S2, which is therefore replaced by the new larger state. Since this state has no successors, and the initial state has no more successors, the exploration has finished.*

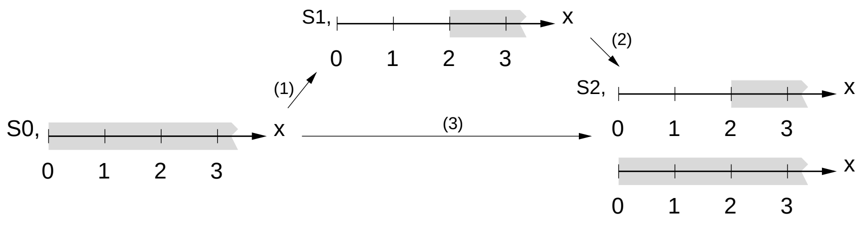


Figure 7.5: The state-space arising from a depth-first search of the automaton in Figure 7.4.

If we instead search the state-space in a breadth-first manner we get the state-space in Figure 7.6. From the initial state, which is equal to the initial state of the depth-first search, two states are reachable. The state with location S1 and zone represented by the x-axis from two to infinity arises from taking the transition from location S0 to S1, and the state with location S2 and zone represented by all of the x-axis arises from taking the transition from location S0 to S2. The state with location S1 has one successor with location S2 and the same zone. This state is included in the state with location S2 which has already been found. The state with location S2 has no successors and the search is finished.

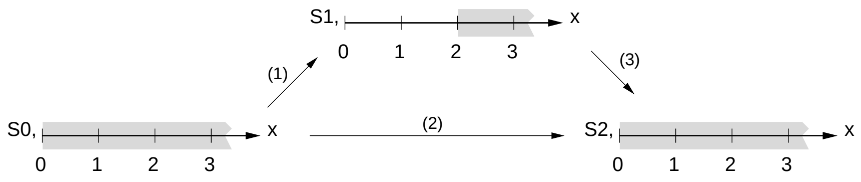


Figure 7.6: The state-space arising from a depth-first search of the automaton in Figure 7.4.

The difference in this example between doing a depth-first and a breadth-first search might seem minor. However, assume there was a transition from location S2 to another part of the automaton with a large state-space. In a depth-first search when the first state with location S2 was encountered, the successor(s) of this state would be explored. Therefore the large state-space reachable from location S2 would be explored. When this was finished we would return to the initial state and then again encounter a state with location S2 but this time with a larger time zone. Therefore we would again have to explore the successors until either we encounter a state which is covered by an already explored state or all states have been searched. This might mean that all of the possibly large state-space reachable from location S2 would have to be explored again.

This is just one example showing how the search order can influence the number of states explored when generating the state-space of a timed automaton. One can easily construct examples where a depth-first search generates fewer states than a breadth-first search. □

Earlier experiments with the sequential version of UPPAAL have shown that breadth-first search is often much faster than depth-first search when generating the complete state-space. We believe this comes from the fact that depth-first search order causes higher degree of fragmentation of the zones than breadth-first order. This results in a higher number of symbolic states being generated because when a new state is generated it is less likely to be included in one of the states previously explored. It might be included in the union of several states with the same location vector, but the union of two zones represented by DBMs can in general not be represented by a DBM.[1]

As noted above, the distributed algorithm neither realizes a strict breadth-first nor depth-first search. As noticed, using FIFO-order the algorithm approximates breadth-first search, but as we increase the number of nodes, chances increase that the nondeterministic nature of the communication causes the ordering to be such that some states with a large depth (distance from the initial state) are explored before other states with a smaller depth. In cases where breadth-first is actually the optimal search order, increasing the number of nodes is bound to increase the number of symbolic states explored.

Since it seems that breadth-first order in most cases is the optimal search order, we propose a heuristic for more accurately approximating breadth-first order. The heuristic keeps the states in each waiting list ordered by depth. Extending a state with a field containing the number of transitions used for reaching that state and implementing the waiting list as a priority queue is one way of realizing the heuristic. This guarantees that the state in the waiting list with the smallest depth is explored first. In Section 7.4, we will demonstrate that this heuristic drastically reduces the rate at which the number of symbolic states increases when the number of nodes grows. In some cases it actually decreases the number of states explored. This implies that breadth-first order is not always optimal.

7.3.3 Distribution Functions and Locality

On one hand, a good distribution function should guarantee a uniform work load for all nodes, and on the other hand it should minimize communication. In most cases these two objectives contradict each other. Therefore we will considered several distribution functions and try to find a suitable tradeoff.

As in [138], most of our results are based on using a hash function as the distribution function. However, to make the inclusion checks of the time zones in the waiting and the passed lists possible, states with the same location vector and the same values of the integer variables must be mapped to the same node. The hash value of a symbolic state is therefore only based on the location vector and not on the complete state, i.e., $h(l, D)$ only depends on l.

[1]DBMs can only represent convex sets.

We have made experiments with distribution functions based on the complete state. However, the increase in the number of states explored arising from fewer successful inclusion checks, negates the advantage of the increased processing power available.

One possible hash function is the one already implemented in Uppaal and used when states are stored in the passed list. It uniquely maps each state to an integer modulo the size of the hash table. Experiments have shown that it distributes location vectors uniformly. Using a uniform distribution of the states makes almost as much communication as possible. When using n nodes in the exploration, each node sends $(n-1)/n$ of the states it generates to other nodes and only $1/n$ states to itself. Trying to increase locality of the distribution function, it should be possible to use the fact that each transitions changes at most two entries in the location vector and only some transitions change the integer variables. We suggest two new general distribution functions both based on the hash function implemented in Uppaal. One of the new distribution functions only calculates the hash value based on every second entry in the location vector (the location of every second automaton in the network). The other distribution function calculates the hash value only based on the value of the integer variables, so it only makes sense to use this in a system with integer variables. Section 7.4 reports on experiments with these two new distribution functions.

Some experiments with model specific distribution functions have been made. Competing with the generic distribution functions has proven to be extremely difficult in the cases we have tried. However, we have only experimented with small to medium size models. Using user supplied model specific distribution functions is unlikely to scale to large systems, since the task of specifying the distribution function becomes intrinsically more difficult.

Within Uppaal the techniques described in [111] for reducing memory consumption by only storing loop entry points in the passed list are quite important for verifying large models. The idea is to keep a single state from every static loop (which are simple to compute). This guarantees termination while giving considerable reductions in memory consumption for some models. Uppaal implements two variations of this techniques. The most aggressive one is described in [111] which only stores loop entry points. While reducing memory consumption this technique may increase the number of states explored, since certain states are explored more than once. A less aggressive approach is to also store all non-committed states (in which no automaton is in a committed location) in the passed list. Experiments show that this is a good compromise between space and speed.

We propose using these techniques to increase locality in the exploration. Since non loop entry points are not stored on the passed list they might as well be explored by the node which computed the state in the first place instead of sending it to another node, thereby increasing locality. Consider, for example, a state α and its successor β. If β is not a loop entry point and therefore is not going to be stored on the passed list, we may as well explore β on the same node as α. Section 7.4 reports on experiments with this technique.

7.3.4 Termination

The model checker should terminate when either an error state has been encountered or all states have been explored. In the first case, the node encountering the error state will broadcast a the termination signal. Detecting the latter case is much more difficult.

Two conditions need to be satisfied before the algorithm can terminate:

1. All waiting lists must be empty, i.e., $\sum_{i=1}^{n} |\textsc{Waiting}_i| = 0$. In this case we say that all nodes are idle.

2. No states should be in the process of being passed between nodes.

The latter condition is necessary since the network acts as a buffer quite similar to the waiting lists. It can be checked for by letting each node count the number of sent and received states. When all sent counters sum up to the same value as all receive counters the second condition is satisfied. The challenge is to detect both conditions without unnecessarily slowing down the verifier. One simple but inefficient approach is to periodically check the two conditions by taking a snapshot of all counters. Typically, one of the nodes is responsible for this. In order to guarantee that the snapshot is taken at the exact same time on all nodes, it is necessary to temporarily halt the verification. This is essentially the approach taken in [138].

We propose a distributed detection algorithm, see Fig. 7.7. It is based on a token, that is passed from node to node. The idea is that we can conclude upon the two conditions when the token has visited all nodes. The token is on the form $(origin, sent, received)$, where $origin$ is a node ID, and $sent$ and $received$ are integers. Initially, the token is assigned to a random node (this might be the node with the ID 0). As long as the node is busy, it keeps the token. When it becomes idle, the token is sent to the next node in order, inserting the sending nodes ID and the two counter values. When receiving the token, the next action depends on whether the node is idle. If it is not, it simply keeps the token until it becomes idle at which point the scenario starts over. If it is idle, the local sent and received counters are added to the respective values in the token and the token is passed on to the next node. When a node receives a token containing the nodes own ID, we know that the token has visited each node and that each node was idle at the moment it received the token.

Now, if both termination conditions are satisfied, the token will indeed visit each node and return to its origin at which point the two integers will be equal (the total number of sent and received states, respectively). Unfortunately, we cannot conclude termination from this, since it only ensures that all nodes where idle at some point. But if the token successfully traverses all nodes a second time and returns to its origin with the exact same sums, we can conclude that no node has sent or received any states in the meantime and hence has been idle since the previous visit.

```
when idle do
    if has token then
        sent (ID, SENT, RECEIVED)

when receiving token (ORIGIN, S, R) do
    if idle then
        if ORIGIN $\neq$ ID then
            sent (ORIGIN, S + SENT, R + RECEIVED)
        else if S == R then
            if first check then
                S' = S
                R' = R
                sent (ID, SENT, RECEIVED)
            else if S == S' $\wedge$ R == R' then
                broadcast termination signal
            else
                sent (ID, SENT, RECEIVED)
        else
            sent (ID, SENT, RECEIVED)
```

Figure 7.7: The distributed termination algorithm passes a token $(origin, sent, received)$ from node to node. The two event handlers shown illustrate the algorithm.

7.3.5 Generating Shortest Traces

An important feature of a model checker is its ability to provide good debugging information in case a certain property is not satisfied. For a failed invariant property this is commonly a trace to the state violating the invariant. Providing a short trace increases the value of a trace. One of the features of UPPAAL is that when the algorithm from Fig. 7.2 is used with a breadth-first search order, the trace to the error state is the shortest possible, since all states that can be reached with a shorter trace have been explored before. It would be nice to have this feature also in a distributed version of the tool. However, as described above, the order of a distributed state-space search is non-deterministic, and this may lead to non-minimal traces. Fortunately, with little extra computational effort a shortest trace can be found regardless of the search order. The idea is once again to record for each symbolic state its "depth", i.e., the length of the shortest trace leading to this state. When a violating state is found the algorithm does not stop, but instead sends the depth of the violating state to all nodes and continues to search for violating states that can be reached with a shorter trace. Whenever a new violating state is found with a smaller depth, this new depth is send to all the nodes. We need to make sure that the inclusion checks performed on the waiting and passed lists do not discard violating states with a potential shorter trace. When a new state (l, D) is added to the waiting or passed list, we normally compare it to every state (l, D') on the list, and if an inclusion exists we keep the larger of the two states. In order not to discard potential traces, we add the restriction that a state is only replaced/discarded if it does not have a smaller depth than the state it is compared to. The same idea is used for the decision whether or not to explore a state when looking it up in the passed list: we only decide not to explore a state if its clock constraints are included in the clock constraints of another state with the same location vector *and* at the same time does not have a smaller depth than the state it is included in. The corresponding line in the algorithm changes to:

$$\textbf{if } D \not\subseteq D' \textbf{ or } \text{depth}(l, D) < \text{depth}(l, D') \textbf{ for all } (l, D') \in \text{PASSED} \textbf{ then}$$

The search stops when all reachable states with smaller depth than the smallest depth of a violating states has been searched. With these changes the algorithm in Fig. 7.3 can find shortest traces independently of the ordering used on the waiting list. As described above we have implemented a heuristic which approximates breadth-first search. In Section 7.4, we demonstrate that when using this heuristic the extra cost for finding the shortest trace is minor and we keep good speedups.

7.4 Experimental Results

Communication between the nodes is implemented using the *Message Passing Interface* (MPI) [137]. Since the algorithm assumes a distributed memory model and we use MPI for communication, porting and running the program on different kinds of machines and architectures is easy. We have conducted experiments on two different machines: a Sun Enterprise 10000 with 24 333Mhz processors, which has a shared memory architecture using the Sun implementation of MPI, and on a Linux Beowulf cluster of 10 450Mhz Pentium III CPUs using the LAM[2] implementation of MPI. Executables for both platforms can be obtained from the UPPAAL website (www.uppaal.com).

7.4.1 Nondeterminism and Search Orders

As one of the first examples, the distributed UPPAAL was tried on a model of a batch plant [82] constructed to verify schedulability of a batch production process. The verification, which on one processor took several hours, surprisingly took less than five minutes on 16 nodes on the Sun Enterprise. This is more than a linear speedup and therefore came as a surprise to us. Verifying schedulability in this model means searching for a state where all batches have been processed. For this particular model we had previously identified depth-first search as the fastest strategy on one node, and therefore we used the FILO-order. In this particular model, the verification benefited from the nondeterministic search order. Using the FILO-order the same goal state was not found as when running the verification on a single processor, and in fact the number of states searched was not the same in the two cases. It is possible to achieve a similar effect with the sequential algorithm by introducing randomness into the search order. First experiments with using a kind of random depth-first search have been promising [26].

[2]See http://www.mpi.nd.edu/lam/

Because of this property of checking for a set of states, we have in the remaining experiments chosen to generate the complete state-space of the given system using a FIFO-order search. Generating the complete state-space reduces the impact of the nondeterministic search order because one cannot find a "lucky" path which reaches the state searched for quickly. This makes the results from different runs comparable.

7.4.2 Speedup Gained

We have chosen to focus our experiments around four known industrial UPPAAL case studies: the start-up algorithm of the DACAPO [115] protocol which has a quite small state-space compared to the other examples, but on the other hand it showed some interesting behavior as will be discussed later; a communication protocol used in B&O audio/video equipment (CP) [70]; a power-down protocol also used in B&O audio/video equipment (PD) [69]; and a model of a buscoupler (which thus far has not been published). The reason not to look further at the model of the batch plant is that the state-space is too big to be generated completely. We also tried all other UPPAAL examples we could find, but these were so small that the complete state-space can be generated in a matter of seconds using a few processors, and were therefore considered too small to be of interest.

The examples were run on the Sun Enterprise on 1, 2, 3, 4, 5, 8, 11, 14, 17, 20 and 23 nodes; and on the Beowulf on 1 to 10 nodes to the extend it was possible (only the DACAPO model could be run on a single node because of memory usage). Since the search order (and thereby the work done) is nondeterministic we repeated one experiment several times. The observed running times[3] and number of states generated varied less than 3%. Running the rest of the experiments only once therefore seemed reasonable.

When generating the complete state-space for a number of examples using the FIFO-order a general pattern occurred. In most cases the number of states generated increased with the number of nodes used in the search, and in all cases the smallest number of states was generated using one node. It therefore seems that in most cases breadth-first is close to the optimal search order for generating the complete state-space. In most cases the increase in the number of states was minor (less than 10%), but for a few examples the increase was substantial. In the DACAPO example the number of states more than doubled — from 45000 states to more than 110000 states using 17 nodes (see Table 7.1).

To counter this phenomenon, we applied the heuristic described in Section 7.3.2 and used a priority queue to order the states waiting on each node such that the states with the shortest path to the initial state was searched first. Not only did this counter the increase in the number of states, it actually decreased the number of states generated in some cases. This shows that there is still room for improvement with respect to the search order also when using a single processor. Table 7.1 and 7.2 show in terms of explored states, the effect of applying the heuristic to our examples. As can be seen from the tables, the

[3]When talking about the running time we always consider the time of the slowest node.

Table 7.1: States generated with (Priority) and without (FIFO) use of heuristic on Sun Enterprise.

#	States							
	DACAPO		CP		Buscoupler		PD	
	FIFO	Priority	FIFO	Priority	FIFO	Priority	FIFO	Priority
1	45001	44925	3466548	3010244	6502804	6436543	7992048	7992098
2	45754	44863	5505161	3027728	8042882	6199274	8004165	8003477
3	69141	45267	5472878	3070491	8064519	6243785	8001670	7997859
4	62541	45177	5454067	3086016	8123748	6171125	8004717	8004439
5	78008	45667	5583368	3077890	8651090	6481067	8002412	7998607
8	77396	46510	5452888	3113378	8359647	6185288	8004898	8004898
11	84598	46318	5642463	3059169	8968257	6184329	8004888	8004892
14	108344	49741	5653134	3102709	8914300	6278855	8004888	8004888
17	110634	52247	5270822	3082967	9049252	6243571	8001813	7996979
20	98266	47573	5449055	3111333	9271401	6251283	8004881	8004880
23	104945	52457	5535724	3065916	9146026	6103629	8004714	8004651

Table 7.2: States generated with (Priority) and without (FIFO) use of heuristic on Beowulf.

#	States							
	DACAPO		CP		Buscoupler		PD	
	FIFO	priority	FIFO	priority	FIFO	priority	FIFO	priority
1	45858	45748	N/A	N/A	N/A	N/A	N/A	N/A
2	48441	46899	N/A	3028368	N/A	N/A	N/A	N/A
3	74882	47671	5605882	3053837	N/A	N/A	N/A	N/A
4	62398	47640	5533159	3058230	15832617	12794520	9473496	9409935
5	79899	47678	5454676	3060070	16637609	13603603	9432828	9287527
6	92678	49438	5684749	3133769	20443824	13896789	9511548	9482742
7	97065	49739	5702856	3074131	20329057	13797531	9513477	9441041
8	97662	50477	5358514	3106414	22430748	14442925	9527173	9488775
9	92642	49284	5449403	3071827	21086691	14455201	9535657	9515920
10	92400	48821	5532205	3060705	20704595	15507978	9526732	9500000

heuristic performs well in three of the four examples and in the PD example it has no effect. We also tried to use a FILO-order search order, and to 'reverse' the heuristic, i.e. first explore the states with the longest path to the initial state during FILO-order search. In both cases the number of states generated was increased substantially. Therefore these search orders were discarded for the remaining experiments.

An important question is of course how well the distribution of the search scales in terms of number of nodes used for the search. Tables 7.3 and 7.4 show the running times in seconds for the different examples on the Sun Enterprise and the Beowulf, respectively. When running on the Sun Enterprise we were able to generate the complete state-space on a single node for all the examples. We can therefore calculate the speedup with respect to running on a single node. The speedups we have calculated are normalized with respect to the number of states explored, to clarify the effect of the distribution. The speedup for i nodes is calculated as

$$\frac{\textit{time on one node/states on one node}}{\textit{time on i nodes/states on i nodes}}$$

where *time on one node* is the time for generating the complete state-space using the distributed version running on one node, and *time on i nodes* is the time of the slowest node when running on i nodes.

Table 7.3: Run time with (Priority) and without (FIFO) use of heuristic on Sun Enterprise.

#	Run time							
	DACAPO		CP		Buscoupler		PD	
	FIFO	Priority	FIFO	Priority	FIFO	Priority	FIFO	Priority
1	8.6	9.0	804.0	732.0	2338.6	2213.8	3362.8	3195.4
2	5.2	5.0	725.8	351.6	1506.5	861.4	1507.1	1101.2
3	5.4	3.7	446.4	238.6	773.0	559.4	943.0	649.8
4	3.9	2.9	317.9	175.2	596.4	413.4	713.4	467.6
5	4.0	2.5	266.9	142.0	501.2	342.5	453.5	373.1
8	2.8	2.1	152.8	86.8	283.0	202.3	231.6	226.9
11	2.6	1.9	121.7	65.3	221.4	148.0	159.9	161.4
14	2.7	2.0	95.5	53.9	172.2	118.0	127.3	133.4
17	2.7	2.1	74.2	43.1	145.2	97.7	106.9	102.4
20	2.4	2.3	66.5	38.8	127.6	83.6	93.0	92.1
23	2.2	2.4	60.2	34.3	112.4	72.7	76.9	79.6

Table 7.4: Run time with (Priority) and without (FIFO) use of heuristic on Beowulf.

#	Run time							
	DACAPO		CP		Buscoupler		PD	
	FIFO	Priority	FIFO	Priority	FIFO	Priority	FIFO	Priority
1	3.88	4.15	N/A	N/A	N/A	N/A	N/A	N/A
2	3.20	3.16	N/A	682.57	N/A	N/A	N/A	N/A
3	3.49	2.37	934.44	349.86	N/A	N/A	N/A	N/A
4	2.88	2.02	540.19	218.94	1060.09	799.69	616.85	541.89
5	2.71	1.64	390.02	169.93	836.09	646.02	413.45	401.30
6	2.62	1.52	337.79	144.50	1796.23	563.08	453.20	377.39
7	2.55	2.47	285.30	124.69	811.78	476.69	343.49	315.69
8	2.51	1.39	200.50	97.84	782.28	440.87	283.07	274.41
9	2.23	1.38	178.75	87.38	619.84	394.91	244.72	242.16
10	2.00	1.19	173.07	82.44	536.74	387.27	214.03	217.98

For the DACAPO example the speedup decreases from being linear already in the case of 5 nodes. However, it only takes 2.5 seconds to generate the complete states space using 5 nodes. Since the states space is small not all nodes can be kept busy and relatively much time is spent to start and close down the exploration. Therefore, a poor speedup was to be expected. For the CP example the speedup is close to linear. However, for the buscoupler and the

PD examples the speedup is super linear, which is surprising since the speedup has been normalized with respect to the total number of states. Figure 7.8 shows the graphs for the speedups of the CP and buscoupler examples. We can offer two possible explanations for this. The reason should most likely be found in a combination of these two.

The passed list storing the states which have been explored, is implemented using a hash table. When fragmenting the passed list into several passed lists on different machines the size of the hash table for the passed list is effectively made larger. This results in fewer collisions in the hash table and therefore checking whether a state is in the passed list is faster. We have made a few experiments with a fixed overall size of the hash table by letting the sum of the sizes of all the hash tables used be constant. This decreases the super linear speedup to an almost linear speedup in the cases we have tried showing this has effect on the overall performance. Sometime the speed up is a little less that linear sometimes a little more that linear. Unfortunately, we have not been able to run a complete series of test with 'fixed' size of the hash table yet. But this is something we will do in the near future.

Accessing main memory is considered to be a bottleneck. When the number of nodes used in an exploration increases so does the amount of cache available (on the Sun Enterprise each node has 4Mb of cache).

This will give faster access to a larger part of the passed list. The same kind of super linear speedups were not encountered by Stern and Dill [138]. As mentioned in their paper, Murφ has implemented a wide range of techniques for minimizing the state-space. This means that, compared to UPPAAL, Murφ spends less time on looking up states and accessing memory, and therefore Murφ does not gain the same speedup from the larger hash tables and larger cache.

On the Beowulf it was in most cases not possible to generate the complete state-space using only one processor. We have therefore chosen to present the amount of work done, where work for i nodes is defined as *the time on i nodes* times i divided by *the number of states on i nodes*, to normalize with respect to the number of node generated. A horizontal line then corresponds to a linear speedup. As expected the line for the DACAPO example increases, so we do not have a linear speedup. The speedup looks better for the CP on the Beowulf example but since we do not have the time on one node (this could not complete due to memory shortage) it is hard to judge whether the work is approaching the work in one node or really is decreasing below that. The same is the case for the buscoupler and the PD example. Figure 7.9 shows the work for the CP and buscoupler examples. One interesting point to notice is that for six nodes with the FIFO-order the buscoupler performs very poorly. The Beowulf cluster we have been using consists of ten processors placed on five boards sharing the memory two and two. When using six nodes two will be located on the same board sharing the same memory. This means that 1/3 of the state-space should be store in the memory on one board with two active processors. However, it is not possible to store 1/3 of the state-space in the memory on one board forcing the nodes on this board to start swapping and thereby slowing down the verification.

The explanations we suggest for the super linear speedups we encounter on the Beowulf are the same as for the Sun Enterprise: larger hash table size and access to a larger amount of local (cache) memory.

7.4.3 Distribution Functions and Locality

In most of the experiments, states are distributed evenly among nodes using the hash function from UPPAAL. However, for small models we observed that some nodes explore twice as many states as others because some location vectors have more reachable symbolic states than others, which means that some nodes have more states allocated than others. Counting the number of different location vectors on the different nodes, the distribution again looks uniform. This effect does not show up in larger models.

We ran experiments for different distribution functions: a function hashing on the discrete part of a state (D0), a function hashing on the complete state (D1), a function hashing on the integer variables (D2), and a function hashing on every second location (D3). We also ran experiments for different settings of the state-space reduction technique described in Section 7.3.3, where only states that are actually stored in the passed list are mapped to different nodes: storing all states (S0), storing non-committed or loop entry points (S1), and storing only loop entry points (S2). Table 7.5 shows for the buscoupler and the power-down models the percentage of states explored on the same node they were generated on. These experiments were run on the Sun Enterprise with 8 CPUs, but similar results were obtained using the Beowulf cluster.

Table 7.5: Percent of locally explored states for different distribution and storage policies for the buscoupler model (left) and the power-down protocol (right) when verified on a Sun Enterprise using 8 nodes.

Bus	D0	D1	D2	D3
S0	14%	n/a	52%	42%
S1	36%	n/a	60%	58%
S2	55%	n/a	62%	62%

PD	D0	D1	D2	D3
S0	4%	n/a	76%	22%
S1	34%	n/a	76%	48%
S2	60%	n/a	78%	86%

For the buscoupler with D0 and S0 we almost obtain the expected uniform distribution (100%/8 = 12.5%). This was not the case for the power-down model although the total load on the nodes was uniform. None of the D1 experiments terminated within a reasonable time frame. This was expected since much fewer inclusion checks can succeed with this distribution function and hence a much higher number of symbolic states will be generated. Both S1/S2 and D2/D3 improve locality. What cannot be seen is that both S1 and S2 increase the number of states generated. For the buscoupler using S2 generates three times as many states as using S0, so even though each node 'sends' more new nodes to itself that to all the other nodes together, it is faster using the S0 option than the S2. D2 is surprisingly uniform while increasing locality, but the load distribution of D3 was observed to be highly non-uniform, resulting in poor performance. For the buscoupler D2 and S1 turned out to be the fastest combination. For the power-down model D2 and S2 turned out to be the fastest combination.

7.4.4 Generating Shortest Traces

For the buscoupler system we tried the version finding the shortest trace on four different properties (finding a particular state not generating the complete state-space) on the Sun Enterprise. The speedups are displayed in Fig. 7.10. As for the DACAPO system the speedup for properties one and two suffer from too few states being explored. The speedup for properties three and four are much better but here more states are searched to find the state satisfying the property. So we can conclude that also the version finding shortest trace scales quite well, as long as sufficiently many states need to be generated.

7.5 Conclusions

This paper demonstrates the feasibility of distributed model checking of timed automata. A side effect of the distribution was an altered search order, which in turn increased the number of symbolic states generated when exploring the reachable state-space. We have proposed explicit ordering of the states in the waiting list as an effective heuristic to improve the scalability of the approach. In addition we propose an algorithm for finding shortest traces that performs well in a distributed model checker. Some of our results suggests possible improvements to the sequential state-space exploration algorithm for timed automata.

In several cases we obtained super linear speedups. We have suggested some explanations, which based on the experiments we have made so far seems to be able to explain this. However, more experiments needs to be conducted to clarify this. These experiments can hopefully take place on a new Beowulf cluster being set up at University of Aalborg. The single nodes in this cluster will have more memory than the ones on the cluster we have been using so far. This will enable us to run the verifications also on one node and therefore calculate the speedup gained by the distribution.

Appendix: Results

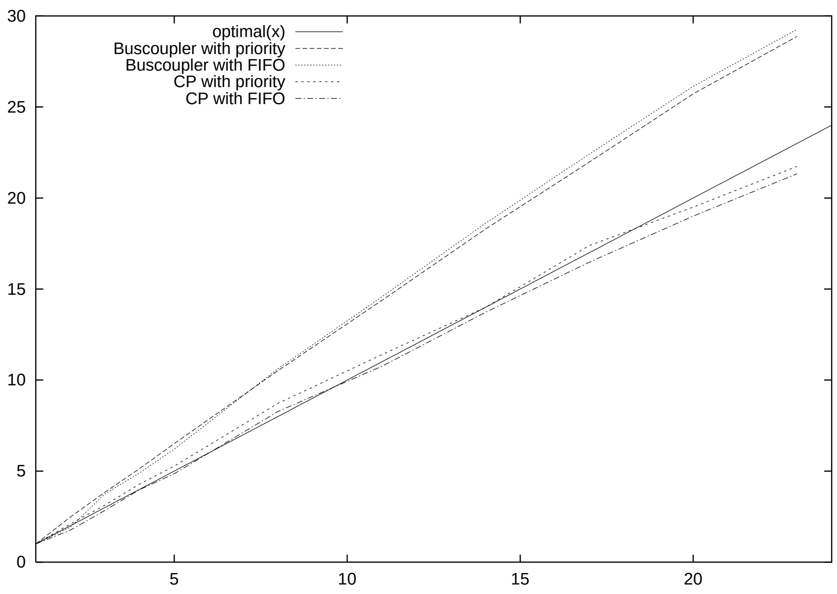


Figure7.8. Speedup for CP and buscoupler on Sun Enterprise

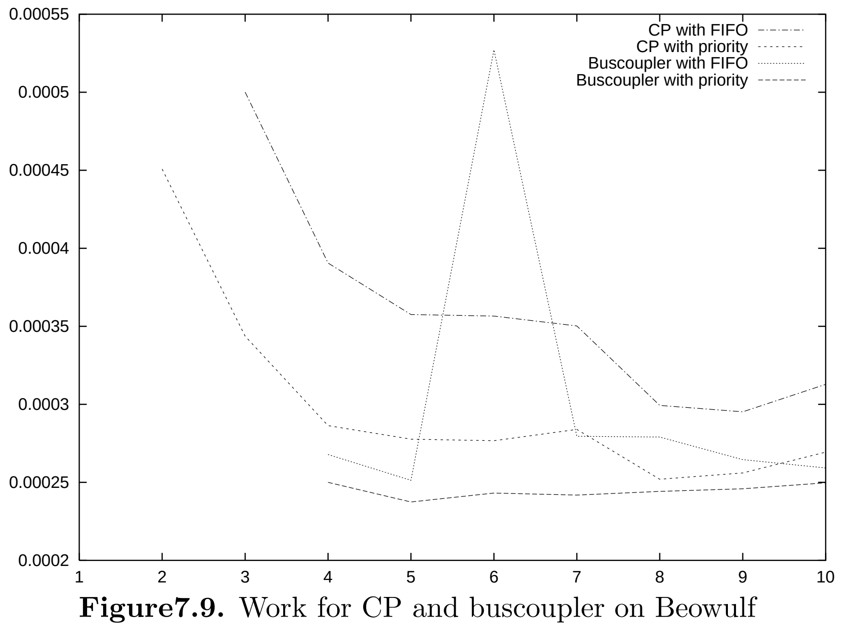


Figure7.9. Work for CP and buscoupler on Beowulf

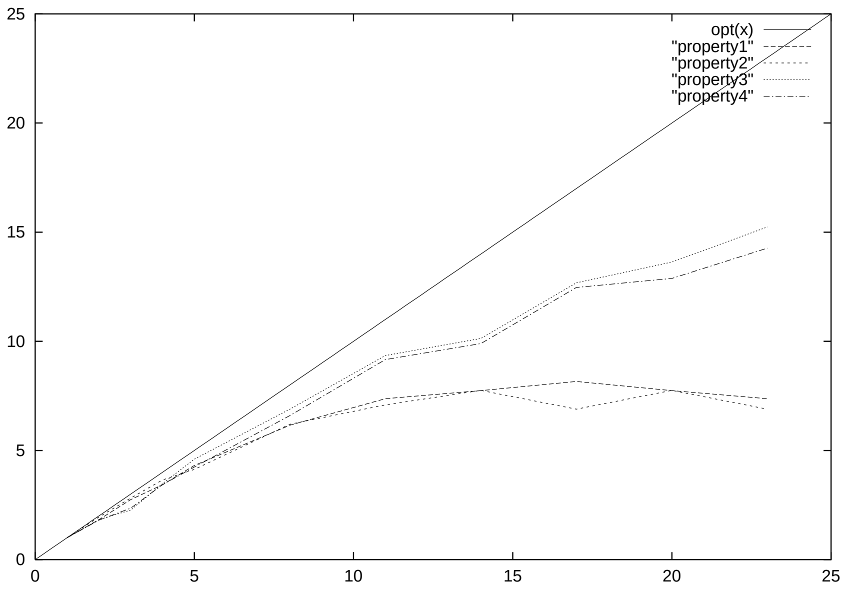


Figure7.10. Speedup for finding shortest trace in Buscoupler model.

Chapter 8

Guiding Uppaal for Synthesizing Control Programs

The paper *Guided Synthesis of Control Programs using* UPPAAL presented in this chapter has been published as a technical report [81] and a conference paper [82]. A journal version has been accepted for publication [83].

[82] T. Hune, K. G. Larsen, and P. Pettersson. Guided Synthesis of Control Programs Using UPPAAL. In *Proc. of the IEEE ICDCS International Workshop on Distributed Systems Verification and Validation*, pages E15–E22, 1998.

[81] T. Hune, K. G. Larsen, and P. Pettersson. Guided Synthesis of Control Programs for a Batch Plant using UPPAAL. Technical Report RS-00-37, BRICS, December 2000.

[83] T. Hune, K. G. Larsen, and P. Pettersson. Guided synthesis of control programs using UPPAAL. Accepted for *Nordic Journal of Computing*, 2001.

The journal version extends the conference paper by adding some more explanation of the guiding which is extended further in the technical report. Except for minor typographical changes the content of this chapter is equal to the technical report [81].

Guided Synthesis of Control Programs for a Batch Plant using Uppaal*

Thomas Hune† Kim G. Larsen‡ Paul Pettersson§

Abstract

In this paper we address the problem of scheduling and synthesizing distributed control programs for a batch production plant. We use a timed automata model of the batch plant and the verification tool Uppaal to solve the scheduling problem.

In modeling the plant, we aim at a level of abstraction which is sufficiently accurate in order that synthesis of control programs from generated timed traces is possible. Consequently, the models quickly become too detailed and complicated for immediate automatic synthesis. In fact, only models of plants producing two batches can be analyzed directly! To overcome this problem, we present a general method allowing the user to *guide* the model-checker according to heuristically chosen *strategies*. The guidance is specified by augmenting the model with additional guidance variables and by decorating transitions with extra guards on these. Applying this method have made synthesis of control programs feasible for a plant producing as many as 60 batches.

The synthesized control programs have been executed in a physical plant. Besides proving useful in validating the plant model and in finding some modeling errors, we view this final step as the ultimate litmus test of our methodology's ability to generate executable (and executing) code from basic plant models.

8.1 Introduction

In this paper we suggest a solution to the problem of synthesizing and verifying valid scheduling control programs for resource allocation, based on a batch

*This work is partially supported by the European Community Esprit-LTR Project 26270 VHS (Verification of Hybrid Systems).

†**B**asic **R**esearch **I**n **C**omputer **S**cience, BRICS, Centre of the Danish National Research Foundation, University of Århus, Denmark, Email: `baris@brics.dk`

‡**B**asic **R**esearch **I**n **C**omputer **S**cience, BRICS,Centre of the Danish National Research Foundation, University of Aalborg, Denmark, Email: `kgl@cs.auc.dk`

§Department of Computer Systems, Information Technology, Uppsala University, E-mail: `paupet@docs.uu.se`.

plant of SIDMAR [39, 64], which is a case study of the VHS project[1]. We model the plant in a network of timed automata, with the different components of the plant (e.g. batches, recipes, casting machine, cranes, etc.) constituting the individual timed automata. The scheduling problem is formulated as a time-bounded reachability question allowing us to apply the real-time model-checking tool UPPAAL [106, 109] to derive a schedule. An overview of the methodology is shown in Figure 8.1.

UPPAAL offers a trace with actions of the model and timing information of the actions. The remaining effort required in transforming such a model trace into an executable control program depends heavily on the accuracy of the model with respect to the control programming language and the physical properties of the plant. In the UPPAAL model given in [64], movement along tracks and on the cranes was assumed to be instantaneous. Making a schedule containing timing information from a trace like that is very hard, and sometimes it is not possible at all. However, given a sufficiently high level of accuracy of the plant model, a schedule can be obtained from a trace by projection, and synthesis of the control program from a schedule amounts to textual substitution. Unfortunately model suitable for such program synthesis becomes very detailed as all the necessary information about the plant, such as the timing bounds and the physical constraints for movements of loads, cranes etc, have be to specified. As an immediate drawback, synthesizing schedules for several batches quickly becomes infeasible. Even for the more abstract models presented in [64] this problem was also encountered.

To deal with this (unavoidable) problem we introduce a method, allowing the user to *guide* the model-checking according to certain chosen *strategies*. Each strategy will contribute with a reduction of the search-space, but in contrast to fully automatic reduction methods like partial order reduction [33] it is up to the user to 'guarantee' preservation of schedulability. However, if a schedule is identified via the guided search, the schedule is indeed a valid one for the original model. Since we are not interested in optimal schedules this is sufficient.

To be able to run the generated control programs in a physical plant, we consider a LEGO® MINDSTORMS™ plant, instead of the original plant of SIDMAR. We have used the plant to successfully run synthesized control programs and by doing so increased our confidence in the plant model. We view this final, scientifically rather simple, step as the ultimate litmus test of our methodology's ability to generate executable (and executing) code from rather natural plant models.

The SIDMAR plant has been studied by several other researchers. Our timed automata model is based on the model in [64], which is similar to ours but more abstract in the sense that some information, such as delays for the moving of batches, is not included. A Petri net model of the plant is presented in [39]. In [139], constraint programming techniques are used to generate schedules of the SIDMAR plant for up to 30 batches. This result is achieved by reducing the size of the plant model using techniques similar to the guiding techniques

[1]See the web site `http://www-verimag.imag.fr//VHS/main.html`.

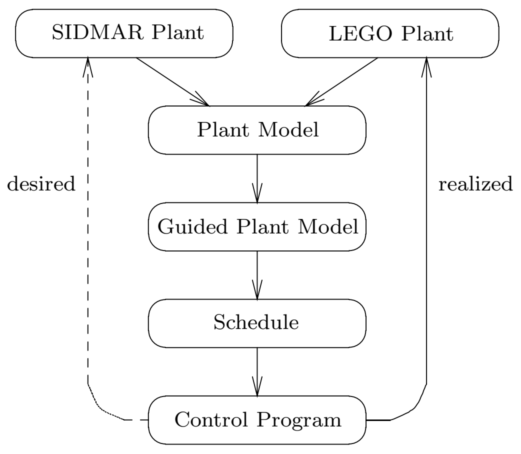


Figure 8.1: Overview of methodology.

presented in this paper. Other work applying the model of timed automata and Uppaal to analyze and solve planning problems of batch plants include [100] in which an experimental batch plant is studied.

The rest of this paper is organized as follows: In the next two sections we describe the scheduling problem and how it has been modeled in Uppaal. In Section 8.4 and 8.5 we present the guiding techniques and evaluate their effect on the plant model. In Section 8.6 we describe experiments with the LEGO® plant and how programs are synthesized for the plant. Section 8.7 concludes the paper. Finally, timed automata descriptions of four plant components are enclosed in the appendix.

8.2 The Scheduling Problem

Our plant is based on a part of the SIDMAR steel production plant located at Gent in Belgium. We will consider the part of the plant between the blast furnace and the continuous casting machine where molten pig iron is converted into steel of different qualities. The process is started when pig iron is poured into ladles by one of two converter vessels. The iron is transported in the ladles while it is being processed. By treatments in different machines the iron is converted into steel and finally casted in the casting machine. Depending on the machines used and how long the treatment in the machines last, different qualities of steel are produced. When the steel in a ladle has been casted the empty ladle must be moved to a storage place. From here the ladles are cleaned and reused. However, this is not part of our model, where ladles are stored at the storage place but not reused. The physical components of the process are: two converter vessels where molten iron is poured into ladles, five machines, tracks connecting these, two cranes running on one overhead track, a buffer place, a storage place for empty ladles, and one casting machine. The layout of the plant can be seen in Figure 8.2.

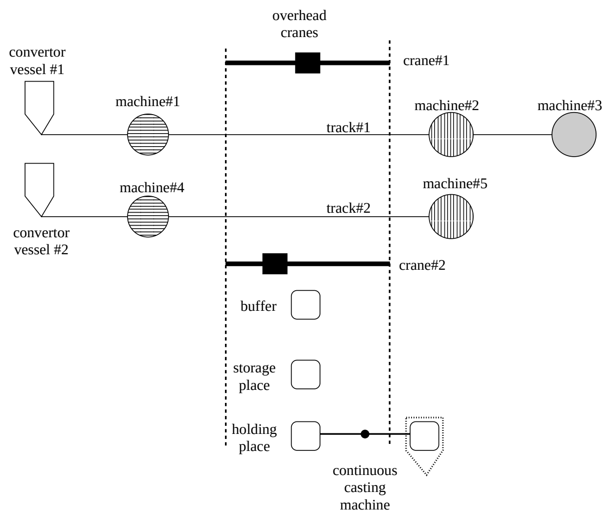


Figure 8.2: Layout of the plant.

Machines number one and four are of the same type and so are machines number two and five. Each crane can only hold one ladle and they cannot overtake each other. On each track and in each machine there is room for at most one ladle. This means that the ladles cannot overtake each other without using one of the cranes.

The steel must sustain a minimum temperature during the process. This gives an upper bound on the time a batch is allowed to spend in the plant from it is poured and until it is casted. Casting takes a fixed time and must be continuous. Therefore a new ladle filled with steel must be waiting in the holding place of the casting machine when casting of a ladle has finished.

Steel of different qualities can be produced depending on which types of machines are visited and for how long. For each batch this is specified by a recipe. The problem to be solved can now be stated as:

Given an ordered list of recipes, if possible synthesize a control program for the plant such that steel specified by the recipes are produced in the right order and within a given time.

The major part of solving this problem is finding a schedule for the production if one exists. A schedule for the plant defines which action takes place in the plant e.g. moving of batches and cranes, and when the actions take place.

8.3 Scheduling with Timed Automata

Finding a schedule for producing an ordered list of steel qualities is the main part of the problem. It can be solved in a number of ways. Here we chose

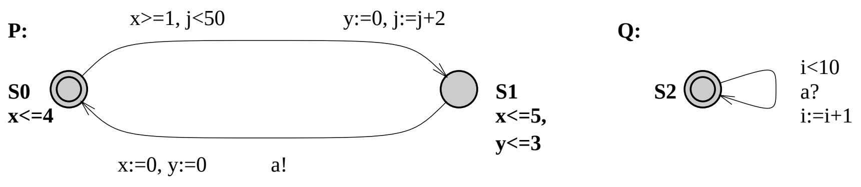


Figure 8.3: A Network of Timed Automata.

to model the plant using timed automata [13] and use the verification tool UPPAAL [106, 109] to solve the scheduling problem[2]. The use of timed automata for modeling the plant enables the scheduling problem to be reformulated as a reachability problem which can be solved by UPPAAL. A discussion of this approach to scheduling can be found in [64].

The modeling language in UPPAAL is networks of timed automata extended with data variables [109]. To meet requirements from various case-studies the language has been further extended with the notion of committed locations [32], urgent synchronization actions [109], and data structures such as arrays of data-variables etc. In this section we give a brief informal description of the modeling language of UPPAAL. For a detailed description we refer the reader to [109].

8.3.1 Networks of Timed Automata

Consider the network of timed automata P and Q shown in Figure 8.3.1. Automaton P has two control locations **S0** and **S1**, two real-valued clocks x and y, and a data variable j. A *state* of the automaton is of the form (l, s, t, k), where l is a control location, s and t are non-negative reals giving the value of the two clocks x and y, and k is a natural number giving value to the data variable j. A control location is labelled with a condition (the location invariant) on the clock values that must be satisfied for states involving this location. Assuming that the automaton starts to operate in the state $(\mathbf{S0}, 0, 0, 0)$, it may stay in location **S0** as long as the invariant $x \leq 4$ of **S0** is satisfied. During this time the values of the clocks increase synchronously. Thus from the initial state, all states of the form $(\mathbf{S0}, t, t, 0)$, where $t \leq 4$, are reachable. The edges of a timed automaton may be decorated with a condition (guard) on the clocks and the data variable values that must be satisfied in order for the edge to be enabled. Thus, only for the states $(\mathbf{S0}, t, t, k)$, where $1 \leq t \leq 4$ and $k < 50$, is the edge from **S0** to **S1** enabled. Additionally, edges may be labelled with assignments and synchronization labels. An assignment may reset the value of the clocks and update the data variables. For example, when following the edge from **S0** to **S1** the clock y is reset to 0 and the data variable j is incremented by 2, leading to states of the form $(\mathbf{S1}, t, 0, 2)$, where $1 \leq t \leq 4$. The synchronization label is used to establish synchronization between automata. For example

[2]See the web site `http://www.uppaal.com/` for more information about UPPAAL.

the transition from **S1** to **S0** of automaton P is labeled with *a!*, requiring the transition to be synchronized with the transition of automaton Q offering the complementary action *a?*.

In general, a timed automaton is a finite-state automata extended with a finite collection $\mathbb{C}$ of real-valued clocks ranged over by x, y etc. and a finite set of data variables $\mathcal{D}$ ranged over by i, j etc. We use $\mathcal{B}(\mathbb{C})$ ranged over by g to stand for the set of formulas that can be an atomic constraint of the form: $x \sim n$ or $x - y \sim n$ for $x, y \in \mathbb{C}$, $\sim \in \{<, \leq, =, \geq >\}$ and n being a natural number, or a conjunction of such formulas. Similarly, we use $\mathcal{B}(\mathcal{D})$ to stand for the set of *data-variable constraints* that are the conjunctive formulas of $i \sim j$ or $i \sim k$, where $\sim \in \{<, \leq, =, \neq, \geq, >\}$ and k is an integer number. To denote the set of formulas that are conjunctions of clock constraints and a data-variable constraints we use $\mathcal{B}(\mathbb{C},\mathcal{D})$ (ranged over by g). The elements of $\mathcal{B}(\mathbb{C},\mathcal{D})$ are called *constraints* or *guards.*

An assignment in UPPAAL is a sequence of operations of the form $x := 0$, or $i := Expr$, where x is a clock, i is a data variable, and $Expr$ is an integer expression, e.g. $2 * (i - j) + 3$ (where j is a data variable). We shall use $\mathcal{R}$ to denote the set of assignments. Furthermore, we use $\mathcal{A}ct$ to denote a finite set of actions ranged over by a, $a?$, $a!$, $b?$, $b!$, etc.

Definition 8.1 (Timed Automata) *A timed automaton A over clocks $\mathbb{C}$ and data variables $\mathcal{D}$ is a tuple $\langle N, l_0, \longrightarrow, I\rangle$ where N is a finite set of (control-) locations, l_0 is the initial location, $\longrightarrow \subseteq N \times \mathcal{B}(\mathbb{C},\mathcal{D}) \times \mathcal{A}ct \times \mathcal{R} \times N$ corresponds to the set of edges and finally, $I : N \mapsto \mathcal{B}(\mathbb{C})$ assigns invariants to locations. In the case, $\langle l, g, a, r, l'\rangle \in \longrightarrow$, we write $l \xrightarrow{g,a,r} l'$.*

To formalize the semantics we use variable assignments. A variable assignment is a mapping which maps the clocks $\mathbb{C}$ to the non-negative reals and the data variables $\mathcal{D}$ to integers. A semantical *state* of an automaton A is now a tuple (l, u), where l is a location of A and u is a an assignment for $\mathbb{C}$ and $\mathcal{D}$, and the semantics of A is given by a transition system with the following two types of transitions (corresponding to delay-transitions and action-transitions):

- $(l, u) \longrightarrow (l, u \oplus d)$ if $I(l)(u)$ and $I(l)(u \oplus d)$
- $(l, u) \longrightarrow (l', u')$ if there exist g and r such that $l \xrightarrow{g,a,r} l'$, $g(u)$, $u' = r[u]$, and $I(l')(u')$

where d is a non-negative real number, $u \oplus d$ denotes the assignment which maps each clock x in $\mathbb{C}$ to the value $u(x) + d$ and leaves each data variable i with the unchanged value $u(i)$, and $r[u]$ denotes the result of updating the clocks $\mathbb{C}$ and the data-variables in $\mathcal{D}$ according to $r \in \mathcal{R}$.

Finally, we briefly introduce the notion of *networks of timed automata* [146, 106]. A network is a finite set of automata composed in parallel with a CCS-like parallel composition operator [121]. For a network with the timed automata

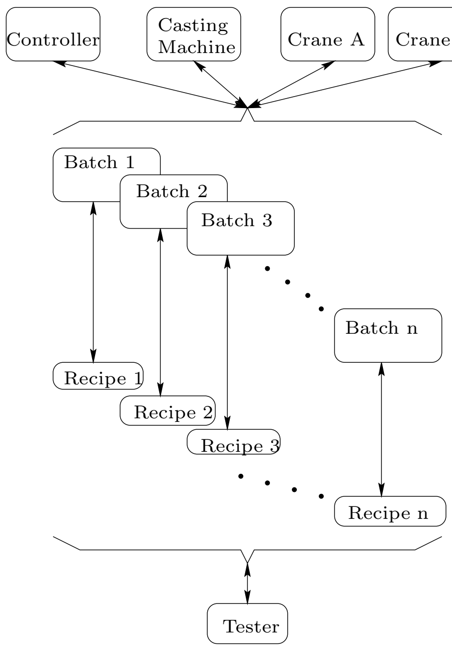


Figure 8.4: Synchronization between the automata of a model.

$A_1, \ldots, A_n$ the intuitive meaning is similar to the CCS parallel composition of $A_1, ..., A_n$ with *all* actions being restricted, that is, $(A_1|...|A_n)\backslash\mathcal{A}ct$. Thus an edge labelled with action *a must* synchronize with an edge labelled with an action complementary to *a*, and edges with the silent τ action are internal, so they do not synchronize. In UPPAAL '?' and '!' are used to represent complementary actions, so *a?* and *a!* are considered complementary and can synchronize.

8.3.2 Analysis

Given a network of timed automata and a set of states, UPPAAL can analyze whether or not one of the states is reachable from the initial state of the network. If the answer is positive, UPPAAL produces a trace with action- and delays-transitions leading from the initial state to one of the specified states.

For the model of the plant, which will be presented in the following, a trace defines a schedule for the plant since it specifies *what* happens in the plant (the synchronization actions) and *when* (the delays). From a schedule a working program controlling the plant may be generated. The level of detail in the trace (and therefore in the schedule) influences the work needed to generate the program. In [64] the traces generated did not include time for the moving of batches, making the generation of executable programs from the schedules hard. To minimize the effort needed during the translation, we produce traces with detailed and precise information about timing of all actions in the plant.

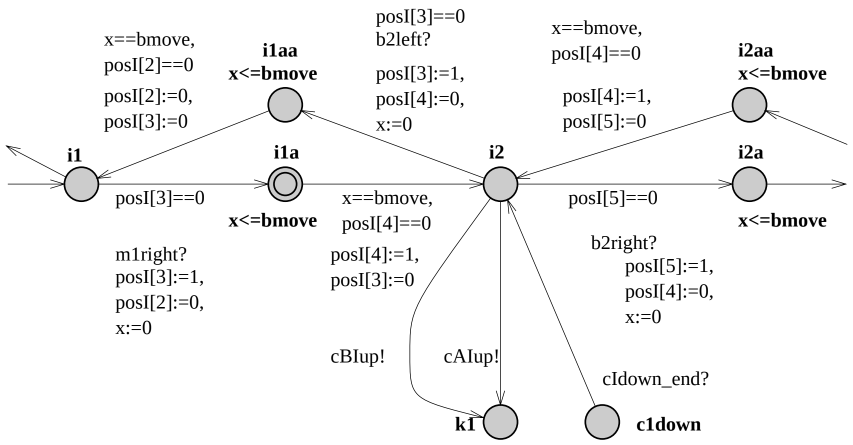


Figure 8.5: Part of the unguided batch automaton.

8.3.3 A Model for Scheduling the Plant

An instance of the problem is given by a list of qualities of steel (or recipes) and a maximal production time. A model of a problem instance consists of: for each recipe one automaton representing the recipe and one automaton representing the movement of the batch; one automaton for each of the two cranes; one automaton testing that the recipes finish in the correct order; one automaton for making some actions synchronizing; and one automaton modeling the casting machine. Figure 8.4 shows the synchronizations between the different automata. The batch automata communicate with each other through two shared arrays and the two cranes also share an array. These arrays will be described in more detail later.

The most complex of the automata is the one modeling the possible behaviors of a batch (see Figure 8.16 of the appendix[3])[4]. The batch automaton reflects the topology of the plant (shown in Figure 8.2) as well as the physical constraints on the movements of a batch. Basically, there is one location for each position of the plant a batch can be located at. A position is either a machine, a track segment, the storage place, the casting machine, or a position on the overhead track. Positions on the overhead track are over one of the two tracks, the storage place, the casting machine, or in between any of these. A batch automaton has a clock named x associated to it which is used to measure the time spend on moving along a track. The time spend is the worst case time measured in the physical plant which is given by the constant *bmove*. Shared among all the

[3]Unless stated otherwise, guided versions of the automata are shown since these have been used for most of the experiments.

[4]Pictures of all the automata and the LEGO® plant can be found at the web site `http://-www.brics.dk/~baris/CaseStudy/`.

batch automata in a model are the two binary arrays *posI* and *posII*, which are used for storing which positions are occupied on the two tracks. These are used to ensure that each position is occupied by at most one batch at a time. Figure 8.5 shows the part of the unguided batch automaton modeling the position named **i2**, between machines number one and two on track one. Moving a batch between positions in the model is done in two steps. First a transition is taken to an intermediate position, e.g. from **i2** to **i1aa**. A batch can only start to move to a position if this position is free, which in this case is ensured by checking the array *posI* using the guard *posI[3]*==0. Taking the transition resets the clock x and updates which positions are occupied by the assignment $posI[3] := 1, posI[4] := 0$. The batch can stay in the intermediate position at most *bmove* time units because of the invariant $x \leq bmove$ in the location. However, it cannot leave the location before *bmove* time units have passed because of the guard $x == bmove$ on the transition leaving the intermediate location. This means that moving a batch along a track is modeled as taking exactly *bmove* time units. A batch can also move when it is carried by a crane. The time spend during such moving is measured by the crane automaton.

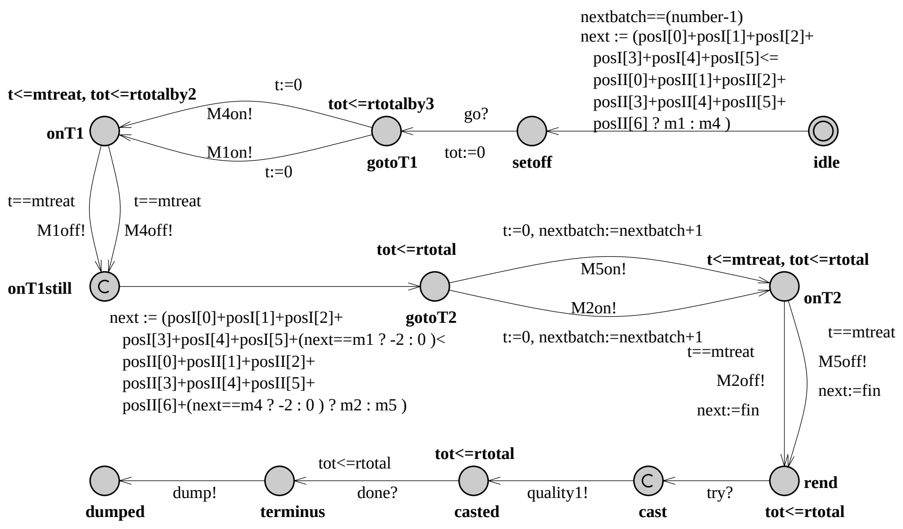

nextbatch==(number-1)
next := (posI[0]+posI[1]+posI[2]+
posI[3]+posI[4]+posI[5]<=
posII[0]+posII[1]+posII[2]+
posII[3]+posII[4]+posII[5]+
posII[6] ? m1 : m4)
t<=mtreat, tot<=rtotalby2
onT1
t:=0
M4on!
M1on!
t:=0
tot<=rtotalby3
gotoT1
go?
tot:=0
setoff
idle
t==mtreat
M1off!
t==mtreat
M4off!
onT1still
tot<=rtotal
gotoT2
t:=0, nextbatch:=nextbatch+1
M5on!
M2on!
t:=0, nextbatch:=nextbatch+1
t<=mtreat, tot<=rtotal
onT2
next := (posI[0]+posI[1]+posI[2]+
posI[3]+posI[4]+posI[5]+(next==m1 ? -2 : 0)<
posII[0]+posII[1]+posII[2]+
posII[3]+posII[4]+posII[5]+
posII[6]+(next==m4 ? -2 : 0) ? m2 : m5)
t==mtreat
M2off!
next:=fin
t==mtreat
M5off!
next:=fin
dumped
dump!
terminus
tot<=rtotal
done?
tot<=rtotal
casted
quality1!
cast
try?
rend
tot<=rtotal

Figure 8.6: An example recipe automaton.

Each batch has a recipe associated to it (a recipe using machine type one and two is shown in Figure 8.6). The recipe defines which machines should be visited, in which order, and for how long. It also measures the overall time the batch has spend in the plant. A recipe has two clocks associated to it. One, *tot*, is reset as the batch starts in the plant and measures the overall time spend in the plant by the batch. The other clock, *t*, is used for measuring the time of the different treatments the batch goes through. When a batch is located at a machine of the right type according to the recipe, the batch and the recipe can synchronize to start the machine. This resets the clock measuring the time of treatments. When the specified time for the treatment has passed the recipe and the batch synchronize to turn the machine off. When the treatments are completed and the batch is ready to be casted the recipe synchronizes with the test automaton to ensure that the production order is kept. Here it is also checked that the batch has not spend too much time in the plant.

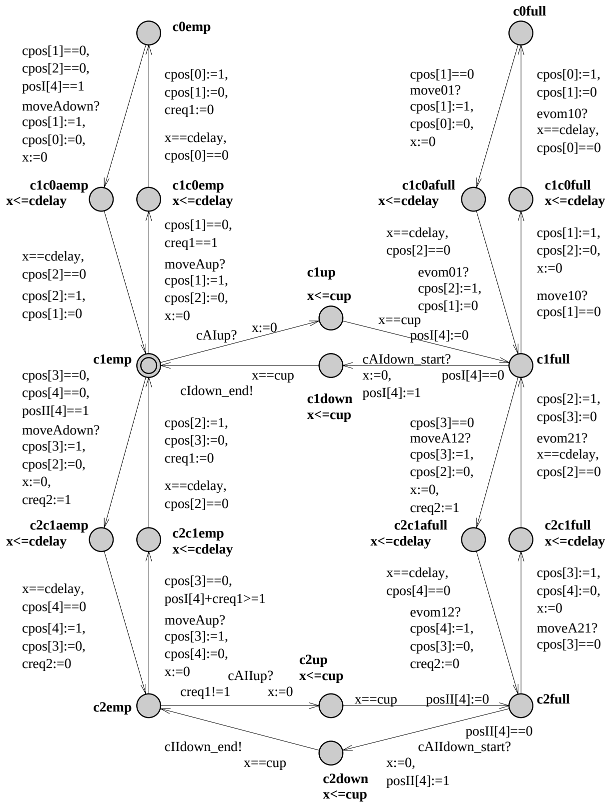

c0emp
c0full
cpos[1]==0, cpos[2]==0, posI[4]==1
moveAdown? cpos[1]:=1, cpos[0]:=0, x:=0
cpos[0]:=1, cpos[1]:=0, creq1:=0
x==cdelay, cpos[0]==0
cpos[1]==0 move01? cpos[1]:=1, cpos[0]:=0, x:=0
cpos[0]:=1, cpos[1]:=0
evom10? x==cdelay, cpos[0]==0
c1c0aemp x<=cdelay
c1c0emp x<=cdelay
c1c0afull x<=cdelay
c1c0full x<=cdelay
x==cdelay, cpos[2]==0
cpos[2]:=1, cpos[1]:=0
cpos[1]==0, creq1==1
moveAup? cpos[1]:=1, cpos[2]:=0, x:=0
c1up x<=cup
x==cdelay, cpos[2]==0
evom01? cpos[2]:=1, cpos[1]:=0
cpos[1]:=1, cpos[2]:=0, x:=0
move10? cpos[1]==0
cAIup?
x:=0
x==cup
posI[4]:=0
c1emp
c1full
cAIdown_start? x:=0, posI[4]:=1
posI[4]==0
x==cup
cIdown_end!
c1down x<=cup
cpos[3]==0, cpos[4]==0, posII[4]==1
moveAdown? cpos[3]:=1, cpos[2]:=0, x:=0, creq2:=1
cpos[2]:=1, cpos[3]:=0, creq1:=0
x==cdelay, cpos[2]==0
cpos[3]==0 moveA12? cpos[3]:=1, cpos[2]:=0, x:=0, creq2:=1
cpos[2]:=1, cpos[3]:=0
evom21? x==cdelay, cpos[2]==0
c2c1aemp x<=cdelay
c2c1emp x<=cdelay
c2c1afull x<=cdelay
c2c1full x<=cdelay
x==cdelay, cpos[4]==0
cpos[4]:=1, cpos[3]:=0, creq2:=0
cpos[3]==0, posI[4]+creq1>=1
moveAup? cpos[3]:=1, cpos[4]:=0, x:=0
x==cdelay, cpos[4]==0
evom12? cpos[4]:=1, cpos[3]:=0, creq2:=0
cpos[3]:=1, cpos[4]:=0, x:=0
moveA21? cpos[3]==0
cAIIup? creq1!=1
x:=0
c2up x<=cup
x==cup
posII[4]:=0
c2emp
c2full
posII[4]==0 cAIIdown_start? x:=0, posII[4]:=1
cIIdown_end!
x==cup
c2down x<=cup

Figure 8.7: The upper crane.

As mentioned the positions of a crane are over the two tracks, over the storage place, over the casting machine and in between these. An automaton modeling a crane has two locations for each of these positions, one modeling the crane being empty and one modeling the crane carrying a batch. The automaton modeling the upper crane which is only moving between the two tracks is shown in Figure 8.7 (the automaton modeling the other crane can be seen in Figure 8.15 of the appendix.) A crane picking up a batch is modeled by the two automata synchronizing. Similarly when a crane moves or sets a batch down. Each crane automaton has one clock which is used for measuring time when the crane is moving. The movement of a crane between two positions is modeled like movement between two positions in the batch automaton with an intermediate location where the time for the movement passes. The two crane automata share a binary array like the batch automata for storing which positions are occupied

The test automaton synchronizes with a recipe automaton just before the recipe allows the batch to enter the casting machine. This ensures that the order of

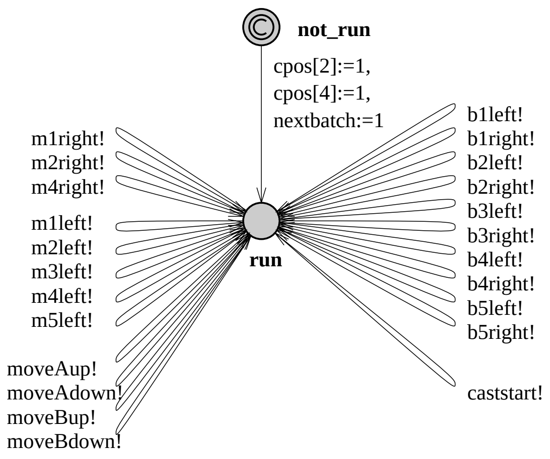


Figure 8.8: The automaton ensuring synchronization.

the production as stated in the problem description is kept (Figure 8.14 in the appendix shows a test automaton).

There is also one automaton which has no influence on the overall behavior of the model (shown in Figure 8.8). However, since we will use the traces obtained from the model for generating schedules, it is important that the all actions of the plant affecting the schedule appear directly in the trace. Some of these actions are internal actions in the batch automaton and will therefore not appear in the generated traces. An example is the movements of a batch on the belts. The purpose of this automaton is to synchronize with the internal actions (modified to external actions) to make them appear in the traces.

Finally there is an automaton modeling the casting machine (see Figure 8.13 of the appendix). It synchronizes with a batch to start the casting. After a specific time when the batch has been casted, the casting machine and the batch should synchronize again to let the batch leave the casting machine. Then the casting machine is ready to synchronize with the next batch which must be waiting, unless the production has finished.

8.4 Guiding Timed Automata

The timed automata described in the previous section models the steel production plant at a high level of accuracy. The details in the model are needed to allow generation of schedules from model traces by projection, and to allow generation of control programs from schedules by textual substitution. However, the fact that the model is detailed and consisting of a many parallel timed automata with several clocks is also a serious problem, as the model is too big and complicated for automatic analysis. In fact, finding traces of a plant model with just a few batches is infeasible in practice (see Section 8.5). The

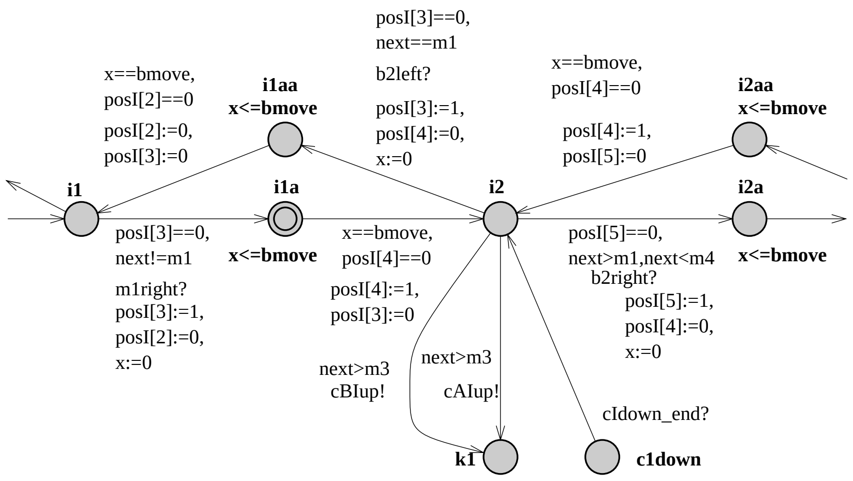


Figure 8.9: Guided part of the batch automaton.

limiting factor is the amount of time and memory consumed during the analysis to (symbolically) explore and store the reachable state-space of the analyzed model. To solve this problem we introduce a way of user directed *guiding* of a state-space exploration algorithm according to a number of certain chosen *strategies*.

8.4.1 Guiding

The overall idea of guiding an automata model is to let the user implement reduction strategies by augmenting the automata with a set of additional clocks, data variables, constraints and assignments[5]. Each strategy will contribute to the reduction of the state-space by constraining the behavior of the model. However, in contrast to automatic state-space reduction techniques, the guiding technique trust the user to preserve schedulability of the plant model.

Assume a network of timed automata over clocks $\mathbb{C}$ and data variables $\mathcal{D}$. The automata are guided by introducing a set of new clocks $\mathbb{C}_G$ and integer variables $\mathcal{D}_G$. We call $\mathbb{C}_G \cup \mathcal{D}_G$ *guiding variables*. A guide is implemented by conjuncting new constraints from $\mathcal{B}(\mathbb{C}_G \cup \mathbb{C}, \mathcal{D}_G \cup \mathcal{D})$ to the existing guards of the automata, new clock constraints from $\mathcal{B}(\mathbb{C}_G \cup \mathbb{C})$ to the location invariants, and adding new assignments of variables in $\mathbb{C}_G \cup \mathcal{D}_G$ to the resets. Thus, the guides may test the values of all the clocks and the data variables in the transition guards and the location invariants of the automata. A guide may also assign the guiding variables in the reset sets. However, the original clocks and data variables of the timed automata (i.e. $\mathbb{C} \cup \mathcal{D}$) should not be assigned. This ensures the essential

[5]The technique of adding guiding variables presented in this paper is reminiscent of the notion of history and prophesy variables used in traditional program verification, as in the work of Abadi and Lamport [2].

property that a trace generated from a guided network of timed automata indeed is a valid trace of the original network of timed automata. In the plant model this means that the schedules generated from the guided plant model is guaranteed to also be valid in the original plant model.

8.4.2 Implemented Strategies

We have used guiding to implement a number of strategies in the plant model. In the following we describe the strategies abstractly, in terms of the physical plant, and give some detailed examples of how the guides are introduced in the plant model. We emphasize that many of the strategies are heuristics and most of them could in fact reduce the number of valid schedules of the plant model. However, this is not a problem as long as it is still possible to generate valid schedules from the model (as we are not concerned with finding optimal schedules).

The implemented strategies are based on the general observation that the plant model described in the previous section models *all* possible behaviors of the plant. This also includes several behaviors that should not (or are unlikely to) appear in a valid schedule. The implemented strategies aim at reducing these ‘unwanted’ behaviors.

Strategy 1: Ordering of Batches. When the scheduling problem is stated the production order of the steel qualities is given. One strategy is to use this order when starting new batches in the plant. According to the engineers at SIDMAR the same strategy is used there.

To implement the strategy we introduce the new guiding variable *nextbatch* in the recipe automaton associated to each batch, to control which batch is allowed to start next. A recipe automaton is shown in Figure 8.6. The guard *nextbatch==(number-1)* on the first transition of the automaton, where *number* is a unique constant number associated to each recipe, implements the guide. The guide ensures that the recipe starts the batch when the value of *nextbatch* is equal to *number-1*. The recipe automaton increments the *nextbatch* variable on a transition from location **goT2** to **onT2** (see also Strategy 2 below) to allow the next batch to start.

Strategy 2: Delaying of Batches. Related to the first strategy is the starting time of batches. Since there is an upper bound on the time a batch is allowed to spend in the plant, all batches should not be started at the same time. Therefore, we prevent a batch from starting based on the progress of the batch just before it. If too much time passes before the batch undergoes the treatments in the recipe, the time bound will also be violated. Based on the progress in the recipe we can check whether it is still possible for the batch to reach the casting machine in time.

The strategy is implemented in the recipe automata by delaying the update of the *nextbatch* variable. In the recipe1 automaton shown in Figure 8.6 the *nextbatch* guiding variable is incremented on the transition from location **goT2** to **onT2** instead of immediately after the test on the first transition. This prevents the next batch to start before the batch has been treated by two machines. To make sure that it is still possible for the batch to reach the casting machine in time, invariants on the clock *tot* are added to some of the locations of the recipe.

Strategy 3: Global Routing of Batches. To make the movement of the batches more deterministic we choose a target for where the batches should move to next. The recipe of each batch chooses a target for the batch which is the machine the batch should visit next (including the casting machine when the treatment of the iron has finished). When there is a choice of machines the recipe will chose the machine on the track with fewest batches present.

We introduce a new guiding variable named *next* for each batch to realize this strategy. The value of *next* specifies where the batch should go next, based on the next machine treatment specified in its recipe. For example the choice of the first machine (there is a machine of this type on each track, so there is a choice here) is implemented by a guiding expression on the first transition of the recipe automaton:

$$\textbf{if } (track1 \leq track2) \textbf{ then } \mathit{next}{:=}\,\mathit{m1} \textbf{ else } \mathit{next}{:=}\,\mathit{m4}$$

where $track1$ is the number of batches present on track one and $track2$ the number of batches on track two. In the recipe automaton in Figure 8.6 the value of $track1$ and $track2$ are computed as the sum of active bits in the bit vectors *posI* and *posII* respectively (recall from the previous section that *posI* and *posII* are used to ensure mutex on the positions of the two production tracks).

Strategy 4: Local Routing of Batches. The possible movements of the batches are further reduced by a strategy deciding how a batch should move between two given position. The implemented strategy selects the only direct route between two positions.

To implement the strategy in the plant model we use the guiding variable *next*. A guard constraining the value of *next* is added to all transitions of the batch automata leaving a location modeling a physical position in the plant. Figure 8.9 shows a part of the guided batch automaton corresponding to the partial original automaton shown in Figure 8.5. Machine one is the only machine located to the left of position **i2** on track 1. Therefore, the guides require the *next* variable to have value *m1* (representing machine 1) to move in the left direction. This is ensured by the guard *next*== *m1* on the transition from location **i2** to **i1aa**. The transitions from location **i2** to **k1** represents the batch being picked up by one of the cranes. When this is the case the next destination of the batch should not be a machine on track one (i.e. not machine 1, 2, or 3) therefore *next* is required to be greater than *m3*.

Strategy 5: Moving of Cranes. When a crane is carrying a batch it always follows the strategy of the batch. If a crane is empty, the strategy is to move only when something is ready to be picked up, or if it is blocking the other crane. To allow a crane to move when it is blocking the other crane, we enable the two cranes to communicate.

Guiding guards in the crane automata testing bits in *posI* and *posII* ensure that the cranes move towards the pick up positions on the tracks only when a batch is waiting to be picked up (see e.g. the transition from location **c2emp** to **c2c1emp** in Figure 8.15 of the Appendix). To allow an empty crane to move in other situations the guiding variables *creq1* and *creq2* are introduced. Guards testing their value are introduced on some transitions to allow the crane to move from certain positions in a specified directions when the variables are non-zero. The variables are typically assigned by the other crane to indicate that it is moving towards a (possibly) occupied position that must be empty. For example, in the craneB automaton shown in Figure 8.15 the variable *creq1* is assigned on the transitions from location **c2emp** to **c1emp** to allow crane 1 to leave crane position 1 (modeled by the locations **c1emp**, **c1up**, **c1down**, and **c1full** in the craneB automaton).

Other Strategies. It is possible to imagine other strategies and other experiments that would be interesting. However, the strategies presented here have been very effective as shown by the results in the next section. Using the approach to guiding presented here allows for easy adding and changing of guides. This is important since guides are based on heuristics so experimenting is sometimes needed for finding good strategies.

8.5 Experimental Results

The plant models described in the previous sections have been analyzed in the validation and verification tool UPPAAL [106, 109]. In this section we present the results of the analysis for three versions of the model, with varying number of guides and batches. In particular, we present the measured time and space needed by UPPAAL to perform the analysis. Comparing the requirements for the different models allows us to evaluate the benefits of the presented guiding techniques. To evaluate the effect of adding guides, we use the standard UNIX programs `time` and `top` to measure the CPU time and the memory consumed by UPPAAL when generating a trace from the two models.

The three analyzed models are: the original plant model without the guides described in Section 8.3, the plant model with all guides added described in Section 8.4, and a model with all guides added except the once using the *nextbatch* variable described in Section 8.4.

#	All Guides				Some Guides				No Guides			
	DFS		BSH		DFS		BSH		DFS		BSH	
	sec	MB	sec	MB	sec	MB	sec	MB	sec	MB	sec	MB
1	0.1	0.9	0.1	0.9	0.1	0.9	0.1	0.9	0.8	2.2	3.9	3.3
2	0.1	1.0	0.1	1.1	2.1	4.4	7.8	1.2	19.5	36.1	-	-
3	3.2	6.5	3.4	1.4	72.4	92.1	901	3.4	-	-	-	-
4	4.0	8.2	4.6	1.8	-	-	-	-	-	-	-	-
5	5.0	10.2	5.5	2.2	-	-	-	-	-	-	-	-
10	13.3	25.3	16.1	9.3	-	-	-	-	-	-	-	-
15	31.6	51.2	48.1	22.2	-	-	-	-	-	-	-	-
20	61.8	89.6	332	46.1	-	-	-	-	-	-	-	-
25	104	144	87.2	83.3	-	-	-	-	-	-	-	-
30	166	216	124.2	136	-	-	-	-	-	-	-	-
35	209	250	-	-	-	-	-	-	-	-	-	-

Table 8.1: Time and space requirements for generating schedules.

Uppaal offers a number of options to control the internal verification algorithm applied in the tool [109]. When analyzing the plant models we have used the compact data-structure for constraints [111], the control-structure reduction [111], and a recently implemented version of the (in-)active clock reduction [58]. In addition we experiment with using depth-first search strategy (DFS), or depth-first search in combination with bit-state hashing (BSH) [77][6].

Table 8.1 shows the time (in seconds) and space (in MB) consumed by Uppaal version 3.0.12 [7] when generating schedules from the three models. The numbers in the leftmost column corresponds to the number of batches in the model (and in the generated schedule). We use the marker "-" to indicate that the corresponding execution requires more than 256MB of memory, more than two hours of execution time, or that a suitable hash table size has not been found. When applying the hash table technique, we have used table sizes from 1048577 to 33554441 bits. The reported results corresponds to the most suitable hash table sizes found.

As can be seen in Table 8.1, the use of guides significantly increases the size of models that can be analyzed. In the guided model, schedules can be generated for 35 batches using 250 MB in 3.5 minutes, whereas no schedule can be generated for three batches (or more) when no guides are used. We also observe that adding some guides improves the situation by enabling analysis of systems with three batches.

It can also be observed that the bit-state hashing technique does not allow analysis of larger models in this experiment, even though it performs well space-wise on most models. We experienced that finding suitable hash table sizes is very tedious for large system models. The largest system analyzed in the

[6]The bit-state hashing technique generates a sub set of the reachable state-space. A feasible schedule found with this technique is therefore guaranteed to also be feasible in the original plant model.

[7]The tool was installed on a Linux Redhat 5.2 machine equipped with a Pentium III processor and 256MB of memory.

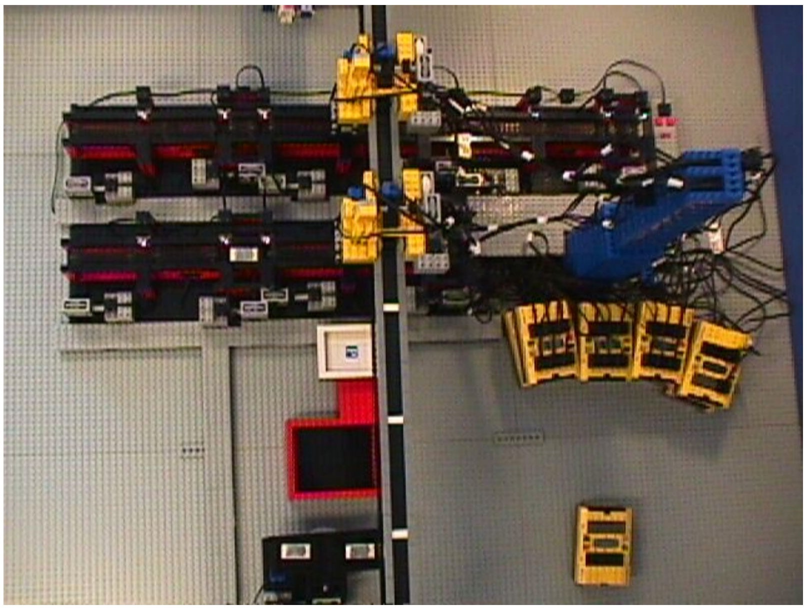


Figure 8.10: The LEGO® plant.

experiment is therefore a guided model using depth-first search strategy but without the bit-state hashing technique.

We also experimented with the breadth-first search strategy. For the model with all the guides two batches could be analyzed in this way whereas only models with one batch could be analyzed in models with only some of the guides or no guides.

We have also installed UPPAAL on a Sun Ultra machine equipped with 1024MB of memory. On this machine, a schedule for 60 batches can be generated from the guided model in 2 257 seconds.

8.6 Synthesis of Control Programs

We did not expect to be able to run the generated control programs in the original plant of SIDMAR. Therefore we have used a LEGO® plant (see Figure 8.10) to run the synthesized programs in. This allows for experimenting with the plant to validate the model and it also makes it easy to find answers to a number of questions about the plant (e.g. measuring time bounds).

The plant consists of a number of distributed units, each controlled locally by one RCX™ [112] brick. There are three types of units used in the plant: a crane, a machine with a track segment, and the casting machine. For the cranes there is an overhead track. The interface to the units consists of a set of commands like *MoveTrackRight*, *TurnOnMachine*, and *LiftBatch*. Commands are send to the local units by one central controller which is running the synthesized program. Ideally, one would want the local controllers to give feedback to the central

```
...                       Delay(5)
Load1.Track1Right         Crane1.Move1Left
Delay(10)                 Delay(5)
Load1.Machine1On          Load1.Machine2On
Load2.Track5Right         Delay(1)
Delay(4)                  Crane1.Move1Left
Crane1.Move1Left          Delay(6)
Delay(6)                  Crane1.Move1Left
Load1.Machine1Off         Delay(3)
Load1.Track2Right         Load1.Machine2Off
Crane1.Pickup1            ...
```

Figure 8.11: Part of a generated schedule.

controller when actions have finished or when an error occurs. However, since the communication between the RCX™ bricks is slow and unreliable especially if more than one brick tries to send at one time, the only feedback from the local controllers are acknowledgements of the received commands from the global controller. This has big influences on the generated programs. There are no loops or branches in the control of the plant, only to implement communication between the RCX™ bricks are loops and branches used.

As a result of the model checking in Uppaal a trace containing information about synchronizations between automata and delays is obtained. Some of the synchronizations in the model, like the recipe synchronizing with the test automaton, are not relevant for the generated schedule. To get a schedule for the plant we project the trace to the actions relevant for the plant. Given some numbering of tracks and machines, part of a possible schedule looks like in Figure 8.11. There is a one-to-one correspondence between a schedule of this kind and the commands of the synthesized central control program. Each line with a `Delay` action is translated into a delay in the control program (in RCX™ code there is a `Wait` instruction doing this). For the rest of the lines only the second part is used, which defines what unit the command should be send to and what the command is. For example in the line `Load1.Track2Right`, the part `Track2Right` is translated to a command *MoveTrackRight* and sent to the local controller of track two.

The projection and the translation have been implemented using the pattern scanning and processing language `gawk`. Since the RCX™ language does not offer reliable communication primitives, each line in the schedule is translated into a code segment implementing such communication. Figure 8.12 shows a part of a synthesized control program. The language does not support functions or procedures therefore the code implementing the communication has to be inlined for each instruction send to a local unit.

The synthesized programs have been executed in the plant. This was mainly intended as validation of the Uppaal model of the plant. During the validation we found three errors in the model: the crane started to move horizontally too early when an empty ladle was picked up from the casting machine, causing the crane to collide with the casting machine and accidently drop the lifted

```
''''moveAup();
''''Crane A - Move UP
PB.PlaySystemSound 1
PB.SendPBMessage 2, 99 ' Move up, on C1
PB.SetVar 1, 15, 0 'Wait for ack
PB.While 0, 1, 3, 2, 99
  PB.Wait 2, 20
  PB.SetVar 1, 15, 0 'Read the message
  PB.ClearPBMessage
  PB.SumVar 2, 2, 1
  PB.If 0, 2, 2, 2, 20 'If looped 20 times
    PB.PlaySystemSound 1
    PB.SendPBMessage 2, 99 'Then Send message,
       again same as sendig 0
    PB.SetVar 2, 2, 0
  PB.EndIf
PB.EndWhile

''''Delay 12
PB.Wait 2, 1200
```

Figure 8.12: Part of a synthesized program.

ladle, so here a delay was missing in the model; when two cranes were located at positions next to each other and started to move in the same direction they could collide because the crane 'in front' was started last; in systems with only one batch the casting machine did not turn correctly. These problems were corrected in the model and new control programs were synthesized.

At one point during the experiments with the plant the batteries running the crane started to wear out. This meant that the initial timing information obtained from the plant was inaccurate because the cranes were moving slower. At this point having the complete process from generating traces to synthesizing control programs fully automated proved especially useful. New times for the moving of the cranes were measured and put into the model. Since scheduling still was possible, new programs were quickly synthesized and were running in the plant as expected.

Performing the experiments also validate the implementation of the translation from schedules to programs and here no problems were found. Our confidence in the correctness of the model has been significantly increased by conducting these experiments.

8.7 Conclusion

In this paper, we have used timed automata and the verification tool Uppaal to synthesize control programs for a batch production plant. To deal with the unavoidable complexity of a plant model suitably accurate for program synthesis, we suggest and apply a general approach of guiding a model according to certain strategies. With this technique, we have been able to synthesize schedules for as many as 60 batches on a machine with 1024 MB of memory. Applying bit-state hashing the space consumption may be decreased even further.

Based on traces from the model checking tool UPPAAL, schedules are generated. From theses schedules, control programs are synthesized and later executed in a physical plant. During execution a few modeling errors were detected. After correcting the model, new schedules were generated and correct programs were synthesized and executed in the plant.

The presented method for guiding model-checking has proved very successful in significantly increasing the size of models which can be analyzed. The largest model we analyze consists of 125 timed automata and a total of 183 clocks. The notion of guides allows the user to add heuristics for controlling the behavior of the plant, and we believe that the approach is applicable and useful for model checking in general and reachability checking in particular. The validation of the model by running the synthesized programs also proved useful: having access to the (a) physical plant during the design of the model, allowed a number of questions to be readily answered.

Based on the traces generated from the UPPAAL model other types of control programs can be synthesized. Here it would be especially interesting to study how more communication between the distributed controllers can be used, e.g. for generating more optimal programs, and for detecting run-time errors. The guides added here decrease the size of the state-space. However, most of the strategies presented here could also be realized by changing the search order to first search the states which are most likely to lead to a goal state. This would not delete possible solutions, which would be particular useful if one were searching for optimal schedules. Searching for optimal schedules, a notion of cost should be added to the model. Here an obvious choice would be the time passed.

Acknowledgements: The authors wish to thank Ansgar Fehnker and Kåre Jelling Kristoffersen for fruitful discussions and many useful suggestions.

Appendix

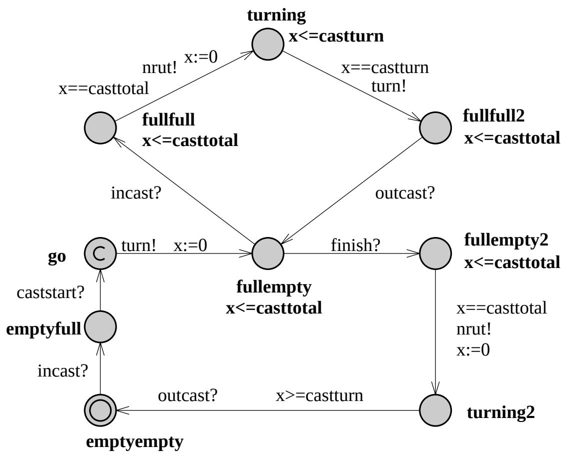


Figure 8.13: The casting machine.

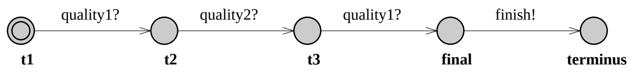


Figure 8.14: A test automaton for producing three batches.

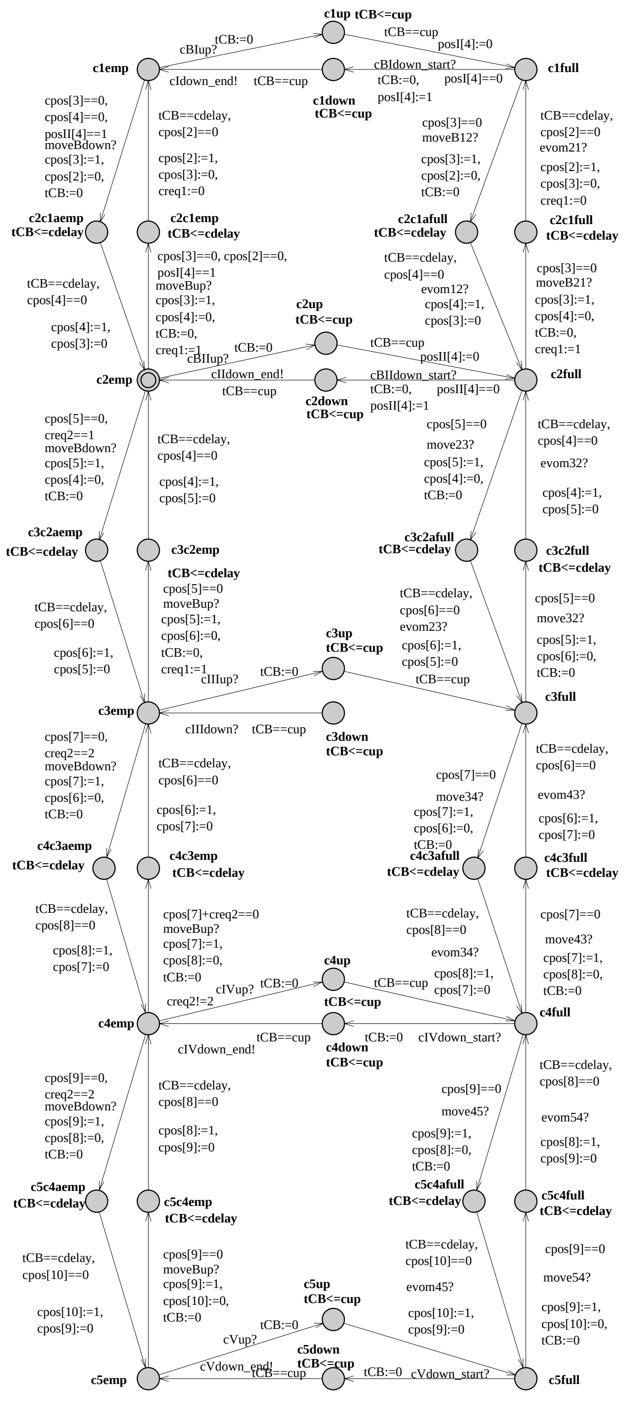


Figure 8.15: The lower crane.

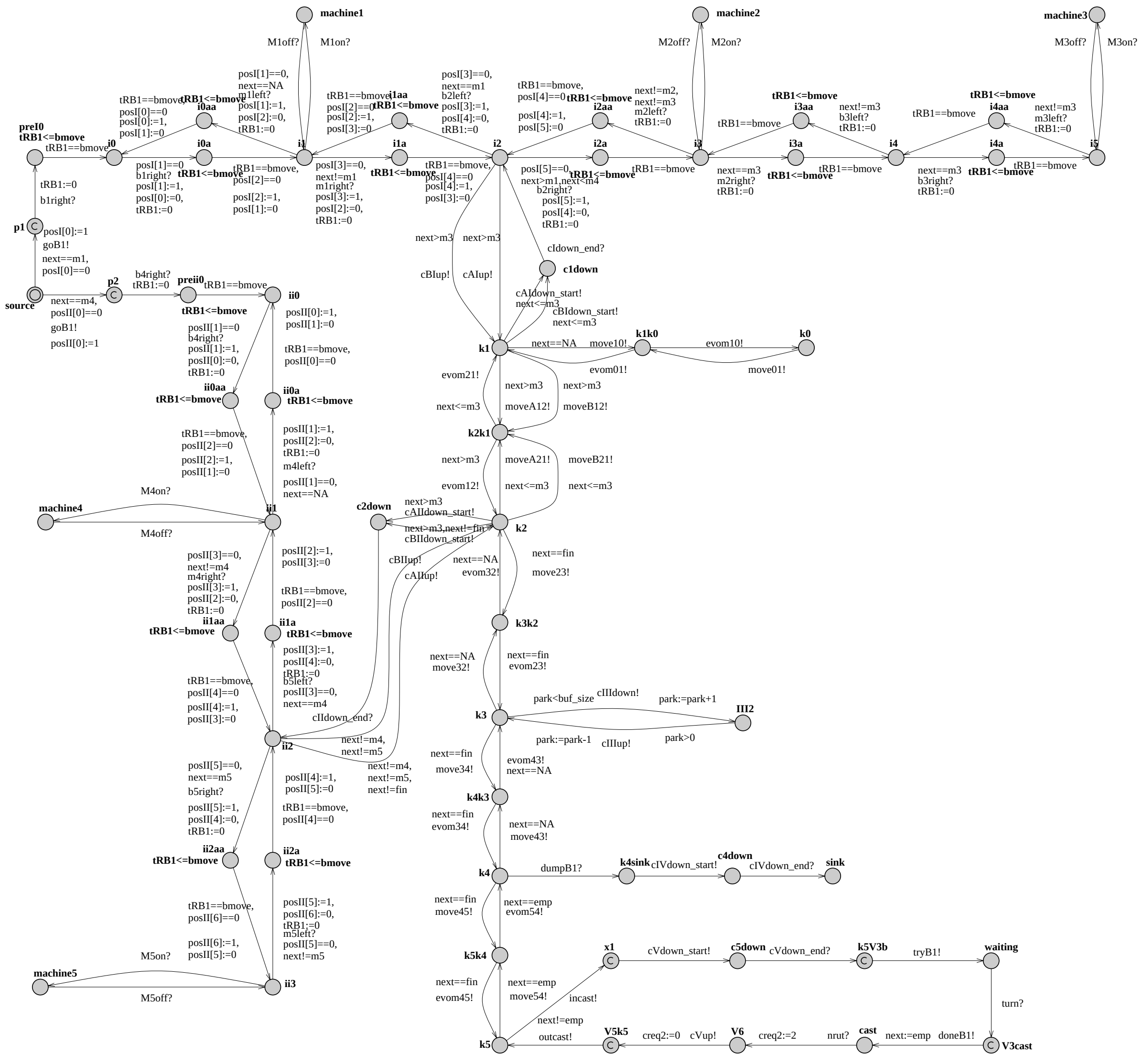


Figure 8.16: The batch automaton.

Chapter 9

Efficient Representation of Uniform Cost

The paper *Efficient Guiding Towards Cost-Optimality in* UPPAAL presented in this chapter has been published in part as a technical report [27] and a conference paper [26].

[26] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson and J. Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001*, pages 174–188, 2001.

[27] G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson and J. Romijn. Efficient Guiding Towards Cost-Optimality in UPPAAL. Technical Report RS-01-04, BRICS, January 2001.

The technical report extends the conference paper with proofs and more elaboration on the conducted experiments. Except for minor typographical changes the content of this chapter is equal to the technical report [27].

Efficient Guiding Towards Cost-Optimality in Uppaal[||]

Gerd Behrmann[*] Ansgar Fehnker[‡][**] Thomas Hune[†]
Kim Larsen[§] Judi Romijn[‡]
Paul Pettersson[¶][††]

Abstract

In this paper we present an algorithm for efficiently computing the minimum cost of reaching a goal state in the model of Uniformly Priced Timed Automata (UPTA). This model can be seen as a submodel of the recently suggested model of linearly priced timed automata, which extends timed automata with prices on both locations and transitions. The presented algorithm is based on a symbolic semantics of UTPA, and an efficient representation and operations based on difference bound matrices. In analogy with Dijkstra's shortest path algorithm, we show that the search order of the algorithm can be chosen such that the number of symbolic states explored by the algorithm is optimal, in the sense that the number of explored states can not be reduced by any other search order based on the cost of states. We also present a number of techniques inspired by branch-and-bound algorithms which can be used for limiting the search space and for quickly finding near-optimal solutions.

The algorithm has been implemented in the verification tool Uppaal. When applied on a number of experiments the presented techniques reduced the explored state-space with up to 90%.

[*]**B**asic **R**esearch **I**n **C**omputer **S**cience, BRICS, (www.brics.dk), funded by the Danish National Research Foundation, University of Aalborg, Denmark, Email: `behrmann@cs.auc.dk`

[†]**B**asic **R**esearch **I**n **C**omputer **S**cience, BRICS, (www.brics.dk), funded by the Danish National Research Foundation, University of Århus, Denmark, Email: `baris@brics.dk`

[‡]Computing Science Institute, University of Nijmegen, Email: `[ansgar,judi]@cs.kun.nl`

[§]Department of Computer Systems, University of Twente, Email: `brinksma@cs.utwente.nl`. On sabbatical from Basic Research in Computer Science, Aalborg University.

[¶]Department of Information Technology, Uppsala University, Email: `paupet@docs.uu.se`.

[||]This work is partially supported by the European Community Esprit-LTR Project 26270 VHS (Verification of Hybrid Systems)

[**]Research supported by Netherlands Organization for Scientific Research (NWO) under contract SION 612-14-004.

[††]Research partly sponsored by the AIT-WOODDES Project No IST-1999-10069.

9.1 Introduction

Recently, formal verification tools for real-time and hybrid systems, such as UPPAAL [109], KRONOS [41] and HYTECH [74], have been applied to solve realistic scheduling problems [64, 82, 127]. The basic common idea of these works is to reformulate a scheduling problem to a reachability problem that can be solved by verification tools. In this approach, the automata based modeling languages of the verification tools serve as the input language in which the scheduling problem is described. These modeling languages have been found to be very well-suited in this respect, as they allow for easy and flexible modeling of systems consisting of several parallel components that interact in a time-critical manner and constrain the behavior of each other in a multitude of ways.

A main difference between verification algorithms and dedicated scheduling algorithms is in the way they search a state-space to find solutions. Scheduling algorithms are often designed to find optimal (or near optimal) solutions and are therefore based on techniques such as branch-and-bound to identify and prune parts of the states-space that are guaranteed to not contain any optimal solutions. In contrast, verification algorithms do normally not support any notion of optimality and are designed to explore the entire state-space as efficiently as possible. The verification algorithms that do support notions of optimality are restricted to simple trace properties such as shortest trace [107], or shortest accumulated delay in trace [126].

In this paper we aim at reducing the gap between scheduling and verification algorithms by adopting a number of techniques used in scheduling algorithms in the verification tool UPPAAL. In doing so, we study the problem of efficiently computing the minimal cost of reaching a goal state in the model of *Uniformly Priced Timed Automata* (UPTA). This model can be seen as a restricted version of the recently suggested model of *Linearly Priced Timed Automata* (LPTA) [28], which extends the model of timed automata with *prices* on all transitions and locations. In these models, the cost of taking an action transition is the price associated with the transition, and the cost of delaying d time units in a location is $d \cdot p$, where p is the price associated with the location. The cost of a trace is simply the accumulated sum of costs of its delay and action transitions. The objective is to determine the minimum cost of traces ending in a goal state.

The infinite state-spaces of timed automata models necessitates the use of symbolic techniques in order to simultaneously handle sets of states (so-called symbolic states). For pure reachability analysis, tools like UPPAAL and KRONOS use symbolic states of the form (l, Z), where l is a location of the timed automaton and $Z \subseteq \mathsf{R}^{\mathbb{C}}$[1] is a convex set of clock valuations called a *zone*. For the computation of minimum costs of reaching goal states, we suggest the use of *symbolic cost states* of the form (l, C), where $C : \mathsf{R}^{\mathbb{C}} \rightarrow (\mathsf{R}_{\geq 0} \cup \{\infty\})$ is a cost function mapping clock valuations to real valued costs or ∞. The intention is

[1]$\mathbb{C}$ denotes the set of clocks of the timed automata, and $\mathsf{R}^{\mathbb{C}}$ denotes the set of functions from $\mathbb{C}$ to $\mathsf{R}_{\geq 0}$.

```
COST := $\infty$
PASSED := $\emptyset$
WAITING := $\{(l_0, C_0)\}$
while WAITING $\neq \emptyset$ do
    select $(l, C)$ from WAITING
    if $(l, C) \models \varphi$ and $min(C) <$ COST then
        COST := $min(C)$
    if for all $(l, C')$ in PASSED: $C' \not\sqsubseteq C$ then
        add $(l, C)$ to PASSED
        for all $(m, D)$ such that $(l, C) \leadsto (m, D)$: add $(m, D)$ to WAITING
return COST
```

Figure 9.1: Abstract Algorithm for the Minimal-Cost Reachability Problem.

that, whenever $C(u) < \infty$, reachability of the symbolic cost state (l, C) should ensure that the state (l, u) is reachable with cost $C(u)$.

Using the above notion of symbolic cost states, an abstract algorithm for computing the minimum cost of reaching a goal state satisfying φ of a uniformly priced timed automaton is shown in Fig. 9.1. The algorithm is similar to a standard state-space traversal algorithm that uses two data-structures WAITING and PASSED to store states waiting to be examined, and states already explored, respectively. Initially, PASSED is empty and WAITING holds an initial (symbolic cost) state. In each iteration, the algorithm proceeds by selecting a state (l, C) from WAITING, checking that none of the previously explored states (l, C') has a "smaller" cost function, written $C' \sqsubseteq C$[2], and if this is the case, adds it to PASSED and its successors to WAITING. In addition the algorithm uses the global variable COST, which is initially set to ∞ and updated whenever a goal state is found that can be reached with a lower cost than the current value of COST. The algorithm terminates when WAITING is empty, i.e. when no further states are left to be examined. Thus, the algorithm always searches the entire state-space of the analyzed automaton.

In [28] an algorithm for computing the minimal cost of reaching designated goal states was given for the full model of LPTA. However, the algorithm is based on a cost-extended version of regions, and is thus guaranteed to be extremely inefficient and highly sensitive to the size of constants used in the models. As the first contribution of this paper, we give for the subclass of UPTA an efficient zone representation of symbolic cost states based on *Difference Bound Matrices* [60], and give all the necessary symbolic operators needed to implement the algorithm. As the second contribution we show that, in analogy with Dijkstra's shortest path algorithm, if the algorithm is modified to always select from WAITING the (symbolic cost) state with the smallest minimum cost, the state-space exploration may terminate as soon as a goal state is explored. This means that we can solve the minimum-cost reachability problem without necessarily searching the entire state-space of the analyzed automaton. In fact, it can even be shown that the resulting algorithm is optimal in the sense that choosing to search a symbolic cost state with non-minimal minimum cost can never reduce the number of symbolic cost states explored.

[2]Formally $C' \sqsubseteq C$ iff $\forall u. C'(u) \leq C(u)$.

The third contribution of this paper is a number of techniques inspired by branch-and-bound algorithms [21] that have been adopted in making the algorithm even more useful. These techniques are particularly useful for limiting the search space and for quickly finding solutions near to the minimum cost of reaching a goal state. To support this claim, we have implemented the algorithm in an experimental version of the verification tool UPPAAL and applied it to a wide variety of examples. Our experimental findings indicate that in some cases as much as 90% of the state-space searched in ordinary breadth-first order can be avoided by combining the techniques presented in this paper. Moreover, the techniques have allowed pure reachability analysis to be performed in cases which were previously unsuccessful.

The rest of this paper is organized as follows: In Section 9.2 we formally define the model of uniformly priced timed automata and give the symbolic semantics. In Section 9.3 we present the basic algorithm and the branch-and-bound inspired techniques. The experiments are presented in Section 9.4. We conclude the paper in Section 9.5.

9.2 Uniformly Priced Timed Automata

In this section linearly priced timed automata are formalized and their semantics are defined. The definitions given here resemble those of [28], except that the symbolic semantics uses cost functions whereas [28] uses priced regions. Zone-based data-structures for compact representation and efficient manipulation of cost functions are provided for the class of uniformly priced timed automata.

9.2.1 Linearly Priced Timed Automata

Formally, linearly priced timed automata (LPTA) are timed automata with prices on locations and transitions. We also denote prices on locations as rates. Let $\mathbb{C}$ be a set of clocks. Then $\mathcal{B}(\mathbb{C})$ is the set of formulas that are conjunctions of atomic constraints of the form $x \bowtie n$ and $x - y \bowtie n$ for $x, y \in \mathbb{C}$, $\bowtie \in \{<, \leq, =, \geq, >\}$ and n being a natural number. Elements of $\mathcal{B}(\mathbb{C})$ are called clock constrains over $\mathbb{C}$. $\mathcal{P}(\mathbb{C})$ denotes the power set of $\mathbb{C}$.

Definition 9.1 (Linearly Priced Timed Automata) *We define a linearly priced timed automaton A over clocks $\mathbb{C}$ and actions Act as a tuple (L, l_0, E, I, P) where L is a finite set of locations, l_0 is the initial location, $E \subseteq L \times \mathcal{B}(\mathbb{C}) \times Act \times \mathcal{P}(\mathbb{C}) \times L$ is the set of edges, where an edge contains a source, a guard, an action, a set of clocks to be reset, and a target, $I : L \to \mathcal{B}(\mathbb{C})$ assigns invariants to locations, and $P : (L \cup E) \to \mathbb{N}$ assign prices to both locations and edges. In the case of $(l, g, a, r, l') \in E$, we write $l \xrightarrow{g,a,r} l'$.*

Following the common approach to networks of timed automata, we extend LPTA to networks of LPTA by introducing a synchronization function f :

$(Act \cup \{0\}) \times (Act \cup \{0\}) \hookrightarrow Act$, where 0 is a distinguished no-action symbol.[3] In addition, two functions $h_L, h_E : \mathsf{N} \times \mathsf{N} \to \mathsf{N}$ for combining prices of transitions and locations respectively are introduced.

Definition 9.2 (Parallel Composition) *Let $A_i = (L_i, l_{i,0}, E_i, I_i, P_i)$, $i = 1, 2$ be two LPTA. Then the parallel composition is defined as $A_1 \mid^f_{h_L,h_E} A_2 = (L_1 \times L_2, (l_{1,0}, l_{2,0}), E, I, P)$, where, $l = (l_1, l_2)$, $I(l) = I_1(l_1) \wedge I_2(l_2)$, $P(l) = h_L(P_1(l_1), P_2(l_2))$, and $l \xrightarrow{g,a,r} l'$ iff there exist g_i, a_i, r_i such that $f(a_1, a_2) = a$, $l_i \xrightarrow{g_i,a_i,r_i}_i l'_i$, $g = g_1 \wedge g_2$, $r = r_1 \cup r_2$, and $P((l, g, a, r)) = h_E(P((l_1, g_1, a_1, r_1)), P((l_2, g_2, a_2, r_2)))$.*

Useful choices for h_L and h_E guaranteeing commutativity and associativity of parallel composition are summation, minimum and maximum.

Clock values are represented as functions called clock valuations from $\mathbb{C}$ to the non-negative reals $\mathsf{R}_{\geq 0}$. We denote by $\mathsf{R}^{\mathbb{C}}$ the set of clock valuations for $\mathbb{C}$.

Definition 9.3 (Semantics) *The semantics of a linearly priced timed automaton A is defined as a labeled transition system with states $L \times \mathsf{R}^{\mathbb{C}}$ with initial state (l_0, u_0) (where u_0 assigns zero to all clocks in $\mathbb{C}$) and with the following transition relation:*

- $(l, u) \xrightarrow{\epsilon(d),p} (l, u + d)$ *if* $\forall 0 \leq e \leq d : u + e \in I(l)$, *and* $p = d \cdot P(l)$,
- $(l, u) \xrightarrow{a,p} (l', u')$ *if there exists* g, r *s.t.* $l \xrightarrow{g,a,r} l'$, $u \in g$, $u' = u[r \mapsto 0]$, $u' \in I(l)$, *and* $p = P((l, g, a, r, l'))$,

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

The transitions are decorated with a delay-quantity or an action, together with the cost of the transition. The cost of an execution trace is simply the accumulated cost of all transitions in the trace.

Definition 9.4 (Cost) *Let $\alpha = (l_0, u_0) \xrightarrow{a_1,p_1} (l_1, u_1) \cdots \xrightarrow{a_n,p_n} (l_n, u_n)$ be a finite execution trace. The cost of α, $cost(\alpha)$, is the sum $\Sigma_{i=1}^n p_i$. For a given state (l, u) the minimum cost $mincost(l, u)$ of reaching the state, is the infimum of the costs of finite traces ending in (l, u). For a given location l the minimum cost $mincost(l)$ of reaching the location, is the infimum of the costs of finite traces ending in (l, u) for some u*

$$mincost(l) = \inf\{cost(\alpha) \mid \alpha \text{ ends in a state } (l, u)\}$$

[3]We extend the edge set E such that $l \xrightarrow{tt,0,\emptyset,0} l$ for any location l. This allows synchronization functions to implement internal τ actions.

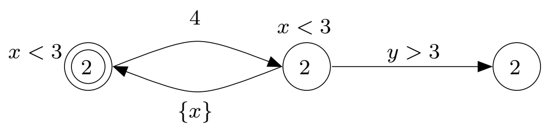


Figure 9.2: The LPTA from Example 9.1.

Example 9.1 *An example of a LPTA can be seen in Fig. 9.2. The LPTA has three locations and two clocks, x and y. The number inside the locations is the rate of the location, and the number on the transition from the leftmost location is the cost of the transition. The two other transitions have no cost. The initial location is the leftmost location.*

Because of the invariants on the locations, a trace reaching the rightmost location must first visit the middle location and then go back to the initial location. The minimal cost of reaching the rightmost location is 14. Note that there is no trace actually realizing the minimum cost because of the strict inequality on the transition to the rightmost location. However, because of the infimum in the definition of minimum cost, we will say that the minimum cost of reaching the rightmost location is 14. □

9.2.2 Cost Functions

The semantics of an LPTA yields an uncountable state-space and is therefore not suited for state-space exploration algorithms. To overcome this problem, the algorithm in Fig. 9.1 uses symbolic cost states, quite similar to how timed automata model checkers like Uppaal use symbolic states.

Typically, symbolic states are pairs on the form (l, Z), where $Z \subseteq \mathsf{R}^{\mathbb{C}}$ is a convex set of clock valuations, called a zone, representable by *Difference Bound Matrices* (DBMs) [60]. The operations needed for forward state-space exploration can be efficiently implemented using the DBM data-structure. However, the operations might as well be defined in terms of characteristic functions, $\mathsf{R}^{\mathbb{C}} \rightarrow \{0, 1\}$, see Table 9.1. For example, let χ be the characteristic function of a zone Z. Then delay can be defined as $\chi^{\uparrow} : u \mapsto \max\{\chi(v) \mid \exists d \in \mathsf{R}_{\geq 0} : v + d = u\}$, that is, u is in $\chi^{\uparrow}$ if there is a clock valuation v, that can delay into u. Looking at zones in terms of their characteristic functions extends nicely to symbolic cost states, but here zones are replaced by mappings, called cost functions, from clock valuations to real valued costs. For this we suggest the use of *symbolic cost states*, (l, C), where C is a cost function mapping clock valuations to real valued costs. Thus, within a symbolic cost state (l, C), the cost of a state (l, u) is given by $C(u)$.

Definition 9.5 (Cost Function) *A cost function $C : \mathsf{R}^{\mathbb{C}} \rightarrow \mathsf{R}_{\geq 0} \cup \{\infty\}$ assigns to each clock valuation, u, a positive real valued cost, c, or infinity. The support $sup(C) = \{u \mid C(u) < \infty\}$ is the set of valuations mapped to a finite cost.*

Table 9.1: Common operations on clock valuations and zones.

Operation	Clock Valuation ($\mathsf{R}^{\mathbb{C}}$)	Zone ($\mathcal{P}(\mathsf{R}^{\mathbb{C}})$)
Delay	$u+d, d \in \mathsf{R}_{\geq 0}$	$Z^{\uparrow} = \{u+d \mid u \in Z \wedge d \in \mathsf{R}_{\geq 0}\}$
Reset	$u[r \mapsto 0]$	$r(Z) = \{r(u) \mid u \in Z\}$
Satisfaction	$u \models g$	$g(Z) = \{u \in Z \mid u \models g\}$
Comparison	$u = v$	$Z_1 \subseteq Z_2 \overset{def}{\Leftrightarrow} \forall u : u \in Z_1 \Rightarrow u \in Z_2$

Table 9.2: Common operations on cost functions.

Operation	Cost Function ($\mathsf{R}^{\mathbb{C}} \to \mathsf{R}_{\geq 0}$)	
Delay	$delay(C,p)$	$: u \mapsto \inf\{C(v) + p \cdot d \mid d \in \mathsf{R}_{\geq 0} \wedge v + d = u\}$
Reset	$r(C)$	$: u \mapsto \inf\{C(v) \mid u = r(v)\}$
Satisfaction	$g(C)$	$: u \mapsto \min\{C(v) \mid v \models g \wedge u = v\}$
Increment	$C + k$	$: u \mapsto C(u) + k, k \in \mathsf{N}$
Comparison	$D \sqsubseteq C \overset{def}{\Leftrightarrow} \forall u : D(u) \leq C(u)$	
Infimum	$min(C) = \inf\{C(u) \mid u \in \mathsf{R}^{\mathbb{C}}\}$	

Table 9.2 summarizes several operations that are used by the symbolic semantics and the algorithm in Fig. 9.1. In terms of the support of a cost function, the operations behave exactly as on zones; e.g. $sup(r(C)) = r(sup(C))$. The operations effect on the cost value reflect the intent to compute the minimum cost of reaching a state, e.g., $r(C)(u)$ is the infimum of $C(v)$ for all v that reset to u.

9.2.3 Symbolic Semantics

The symbolic semantics for LPTA is very similar to the common zone based symbolic semantics used for timed automata.

Definition 9.6 (Symbolic Semantics) *Let $A = (L, l_0, E, I, P)$ be a linearly priced timed automaton. The symbolic semantics is defined as a labelled transition system over symbolic cost states on the form (l, C), l being a location and C a cost function with the transition relation:*

- $(l, C) \xrightarrow{\epsilon} \Big(l, I(l)\big(delay(C, P(l))\big)\Big)$,
- $(l, C) \xrightarrow{a} \big(l', I(l')\big(r(g(C))\big) + p\big)$ *iff* $l \xrightarrow{g,a,r} l'$, *and* $p = P((l, g, a, r, l'))$.

The initial state is $(l_0, I(l_0)(C_0))$ where $sup(C_0) = \{u_0\}$ and $C_0(u_0) = 0$.

Notice that the support of any cost function reachable by the symbolic semantics is a zone.

Lemma 9.1 *Given LPTA A, for each trace α of A that ends in state (l, u), there exists a symbolic trace β of A, that ends up in a symbolic cost state (l, C), such that $C(u) = cost(\alpha)$.*

Proof. By induction in the length of the run α. The base case, a run of length 0, is trivial.

For the induction step assume we have a trace α ending in a state (l, u) and a symbolic trace ending in a symbolic state (l, C), such that $C(u) = cost(\alpha)$. We look at two cases:

- The trace α is extended with a delay transition $(l, u) \xrightarrow{\epsilon(d),p} (l, u+d)$ such that $\forall 0 \leq e \leq d : u + e \in I(l)$ where $p = d * P(l)$. The cost of reaching $(l, u+d)$ in this way is $cost(\alpha) + p$. This can be matched in the symbolic semantics by a delay transition $(l, C) \xrightarrow{\epsilon} \Big(l, I(l)\Big(delay\big(C, P(l)\big)\Big)\Big)$. Using the definition of $delay(C, p)$ in Table 9.2 we get that the cost of $u+d$ after the delay is $C(u) + d * P(l)$. Since $C(u) = cost(\alpha)$ from the induction hypothesis we have the desired result.

- The trace α is extended with an action transition $(l, u) \xrightarrow{a,p} (l', u')$ using a transition $l \xrightarrow{g,a,r} l'$ where $u \in g$, $u' = u[r]$, and $u' \in I(l')$. The cost of reaching (l, u') is $cost(\alpha) + p$ where $p = P((l, g, a, r, l'))$. This can be matched in the symbolic semantics by an action transition $(l, C) \xrightarrow{a} \Big(l', I(l)\big(r(g(C))\big) + p\Big)$. Since u satisfies the guard g, $u' = u[r]$, and u' satisfies the invariant $I(l')$, then $I(l')(r(g(C)))(u') = C(u)$. Therefore according to Table 9.2 $(I(l')(r(g(C))) + p)(u') = C(u) + p$, where $p = P((l, g, a, r, l'))$.

□

Lemma 9.2 *Given an LPTA A, for each symbolic trace, β, ending in a symbolic state (l, C), for each $u \in sup(C)$, there exist a trace α ending in state (l, u) such that and $cost(\alpha) \leq C(u)$.*

Proof. We prove this by induction in the length of the symbolic trace leading to (l, C). The base case, a trace of length zero, is trivial.

For the induction we assume that there is a symbolic trace ending in (l, C) and for $u \in sup(C)$ there is a trace α ending in state (l, u), such that and $cost(\alpha) \leq C(u)$. We look at two cases:

- The symbolic trace β is extended with a delay transition $(l, C) \xrightarrow{\epsilon} C'$ where $C' = \Big(l, I(l)\Big(delay\big(C, P(l)\big)\Big)\Big)$. For the valuations u which were in $sup(C)$ before the delay, the cost has not changed (in the definition of $delay(C, p)$ in Table 9.2, choose $d = 0$). The valuations u' which are in $sup(I(l)\Big(delay\big(I(l)(C), P(l)\big)\Big))$ but not in $sup(C)$, are reachable from a valuation u in $sup(C)$ by a delay d. The cost of u' is $\inf\{C(u) + d \cdot p \mid u \in sup(C)\}$. This can be match in the concrete semantics by a delay transition from (l, u). Since $cost(l, u) \leq C(u)$ and the delay is the same, $cost(l, u') \leq C'(u')$.

- The symbolic trace β is extend with an action transition $(l, C) \xrightarrow{a} (l', C')$ where $C' = I(l)\big(r(g(C))\big) + p$ using the transition $l \xrightarrow{g,a,r} l'$ with cost p. The same transition can be used to extend the trace, also with cost p. Since $cost(l, u) \leq C(u)$ from the assumption and the same transition is used, we are finished.

$\square$

Theorem 9.1 *$mincost(l) = \min\{min(C) \mid (l, C)$ is reachable$\}$*

Theorem 9.1 ensures that the algorithm in Fig. 9.1 indeed does find the minimum cost, but since the state-space is still infinite there is no guarantee that the algorithm ever terminates. For zone based timed automata model checkers, termination is ensured by normalizing all zones with respect to a maximum constant M [133], but for LPTA ensuring termination also depends on the representation of cost functions.

9.2.4 Representing Cost Functions

As stated in the introduction, we provide an efficient implementation of cost functions for the class of Uniformly Priced Timed Automata (UPTA).

Definition 9.7 (Uniformly Priced Timed Automata) *We define an uniformly priced timed automata to be an LPTAs where all locations have the same rate. We refer to this rate as the rate of the UPTA.*

Lemma 9.3 *Any UPTA A with non-zero positive rate can be translated into an UPTA B with rate 1 such that $mincost(l)$ in A is identical to $mincost(l)$ in B.*

Proofsketch. Let A be an UPTA with positive rate r. Now, let B be like A except that all constants on guards and invariants are multiplied by r and set the rate of B to 1. $\square$

Thus, in order to find the infimum cost of reaching a satisfying state in UPTA, we only need to be able to handle rate zero and rate one.

In case of rate zero, all symbolic states reachable by the symbolic semantics have very simple cost functions: The support is mapped to the same integer (because the cost is 0 in the initial state and only modified by the increment operation). This means that a cost function C can be represented as a pair (Z, c), where Z is a zone and c an integer, s.t. $C(u) = c$ when $u \in Z$ and ∞ otherwise. Delay, reset and satisfaction are easily implementable for zones using DBMs. Increment is a matter of incrementing c and a comparison $(Z_1, c_1) \sqsubseteq (Z_2, c_2)$ reduces to $Z_2 \subseteq Z_1 \wedge c_1 \leq c_2$. Termination is ensured by normalizing all zones with respect to a maximum constant M.

In case of rate one, the idea is to use zones over $\mathbb{C} \cup \{\delta\}$, where δ is an additional clock keeping track of the cost, s.t. every clock valuation u is associated with *exactly one* cost $Z(u)$ in zone Z[4]. Then, $C(u) = c$ iff $u[\delta \mapsto c] \in Z$. This is possible because the continuous cost advances at the same rate as time. Delay, reset, satisfaction and infimum are supported directly by DBMs. Increment $C + c$ translates to $Z[\delta \mapsto \delta + k] = \{u[\delta \mapsto u(\delta) + k] \mid u \in Z\}$ and is also realizable using DBMs. For comparison between symbolic cost states, notice that $Z_2 \subseteq Z_1 \Rightarrow Z_1 \sqsubseteq Z_2$, whereas the implication in the other direction does not hold in general, see Fig. 9.3. However, it follows from the following Lemma 9.4 that comparisons can still be reduced to set inclusion provided the zone is extended in the δ dimension, see Fig. 9.3.

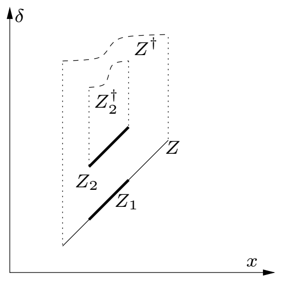


Figure 9.3: Let x be a clock and let δ be the cost. In the figure, $Z \sqsubseteq Z_1 \sqsubseteq Z_2$, but only Z_1 is a subset of Z. The $()^\dagger$ operation removes the upper bound on δ, hence $Z_2^\dagger \subseteq Z^\dagger \Leftrightarrow Z \sqsubseteq Z_2$.

Lemma 9.4 *Let* $Z^\dagger = \{u[\delta \mapsto u(\delta) + d] \mid u \in Z \wedge d \in \mathsf{R}_{\geq 0}\}$. *Then* $Z_1 \sqsubseteq Z_2 \Leftrightarrow Z_2^\dagger \subseteq Z_1^\dagger$.

Proof. By definition $Z_1 \sqsubseteq Z_2 \Leftrightarrow \forall u : Z_1(u) \leq Z_2(u)$. First, assume $Z_1 \sqsubseteq Z_2$ and let $u[\delta \mapsto c] \in Z_2^\dagger$. Then $Z_1(u) \leq Z_2(u) \leq c$ and by definition $u[\delta \mapsto Z_1(u) + d] \in Z_1^\dagger$ for $d \in R_{\geq 0}$ implying $u[\delta \mapsto c] \in Z_1^\dagger$. This proofs one direction of the lemma. Second, assume $Z_2^\dagger \subseteq Z_1^\dagger$. By definition $u[\delta \mapsto Z_2(u)] \in Z_2^\dagger \subseteq Z_1^\dagger$ and it follows that $Z_1(u) \leq Z_2(u)$. □

It is straightforward to implement the $()^\dagger$-operation on DBMs. However, a useful property of the $()^\dagger$-operation is, that its effect on zones can be obtained without implementing the operation. Let $(l_0, Z_0^\dagger)$, where Z_0 is the zone encoding

[4]We define $Z(u)$ to be ∞ if u is not in Z.

C_0, be the initial symbolic state. Then $Z = Z^\dagger$ for any reachable state (l, Z) — intuitively because δ is never reset and no guards or invariants depend on δ. Note that in a zone $Z^\dagger$ the cost of a clock valuation is $C(u) = c$ where $c = \inf\{c \mid u[\delta \mapsto c] \in Z^\dagger\}$.

Termination is ensured if all clocks except for δ are normalized with respect to a maximum constant M. It is important that normalization never touches δ. With this modification, the algorithm in Fig. 9.1 will essentially encounter the same states as the traditional forward state-space exploration algorithm for timed automata, except for the addition of δ.

9.3 Improving the State-Space Exploration

As mentioned the major drawback of using the algorithm in Fig. 9.1 to find the minimum cost of reaching a goal state is that the complete states space has to be searched. However, this can in most cases be improved in a number of ways. Realizing the connection between Dijkstra's shortest path algorithm and the UPPAAL state-space search leads us to stop the search as soon as a goal state has been found. However, this is based on a kind of breadth first search which might not be possible for systems with very large state-spaces. In this case using techniques inspired by branch and bound algorithms can be helpful.

9.3.1 Minimum Cost Order

In realizing the algorithm of Fig. 9.1, and in analogy with Dijkstra's algorithm for finding the shortest path in a directed weighted graph, we may choose always to select a (symbolic cost) state (l, C) from WAITING for which C has the smallest minimum cost. With this choice, we may terminate the algorithm as soon as a goal state is selected from WAITING. We will refer the search order arising from this strategy as the Minimum Cost order (MC order).

Lemma 9.5 *Using the MC order, an optimal solution is found by the algorithm in Fig. 9.1 when a goal state is selected from* WAITING *the first time.*

Proof. When a state is taken from WAITING using the MC order, no states with lower cost are reachable. Therefore, when the first goal state is taken from WAITING no (goal) states with lower cost are reachable, so the optimal solution has been found. □

When applying the MC order, the algorithm in Fig. 9.1 can be simplified since the variable COST is not needed any more.

Again in analogy with Dijkstra's shortest path algorithm, the MC ordering finds the minimum cost of reaching a goal state with guarantee of its optimality, in a manner which requires exploration of a *minimum number* of symbolic cost states.

Lemma 9.6 *Finding an the optimal cost of reaching a location and proving it to be optimal using the algorithm in Fig. 9.1, it can never reduce the number of explored states to prefer exploration of a symbolic cost state of* WAITING *with non-minimal minimum cost.*

Proof. Assume on the contrary that this would be the case. Then at some stage, the exploration of a symbolic cost state (l, C) of WAITING with non-minimal cost should be able to reduce the subsequent exploration of one of the symbolic cost states (m, D) of WAITING with smaller minimum cost. That is, some derivative of (l, C) should be applicable in pruning the exploration of some derivative of (m, D), or more precisely, $(l, C) \rightsquigarrow^* (l', C')$ and $(m, D) \rightsquigarrow^* (m', D')$ with $l' = m'$ and $C' \sqsubseteq D'$. By definition of $\sqsubseteq$ and since $\rightsquigarrow$ never decreases minimum cost, it follows that $min(C) \leq min(C') \leq min(D')$. But then, first exploring the states from WAITING with the smallest minimum cost (using the MC order) would also explore (l, C) and (l', C') before (m', D') and hence lead to the same pruning of (m', D') contradiction the assumed superiority of the non-MC search order. □

In situations when WAITING contains more than just one symbolic cost state with smallest minimum cost, the MC order does not offer any indication as to which one to explore first. In fact, for exploration of the symbolic state-space for timed automata without cost, we do not know of a definite strategy for choosing a state from WAITING such that the fewest number of symbolic states are generated. However, any improvements gained with respect to the search-order strategy for the state-space exploration of timed automata will be directly applicable in our setting with respect to the strategy for choosing between symbolic cost states with same minimum cost.

9.3.2 Using Estimates of the Remaining Cost

From a given state one often has an idea about the cost remaining in order to reach a goal state. In branch-and-bound algorithms this information is used both to delete states and to search the most promising states first. Using information about the remaining cost can also decrease the number of states searched before an optimal solution is reached.

For a state (l, u) let $rem((l, u))$ be the minimum cost of reaching a goal state from that state. In general we cannot expect to know exactly what the remaining cost of a state is. We can instead use an estimate of the remaining cost as long as the estimate does not exceed the actual cost. For a symbolic cost state (l, C) we require that $\textsc{Rem}(l, C)$ satisfies $\textsc{Rem}(l, C) \leq \inf\{rem((l, u)) \mid u \in sup(C)\}$, i.e. $\textsc{Rem}(l, C)$ offers a lower bound on the remaining cost of all the states with location l and clock valuation within the support of C.

Combining the minimum cost $min(C)$ of a symbolic cost state (l, C) with the estimate of the remaining cost $\textsc{Rem}(l, C)$, we can base the MC order on the sum of $min(C)$ and $\textsc{Rem}(l, C)$. Since $min(C) + \textsc{Rem}(l, C)$ is smaller than the actual cost of reaching a goal state, the first goal state to be explored is guaranteed to have optimal cost. We call this the MC+ order but it is also known as Least-Lower-Bound order. In Section 9.4 we will show that even simple estimates of the remaining cost can lead to large improvements in the number of states searched to find the minimum cost of reaching a goal state.

One way to obtain a lower bound is for the user to specify an initial estimate and annotate each transition with updates of the estimate. In this case it is the responsibility of the user to guarantee that the estimate is actually a lower bound in order to ensure that the optimal solution is not deleted. This also allows the user to apply her understanding and intuition about the system.

To obtain a lower bound of the remaining cost in an *automatic* and *efficient* manner, we suggest to replace one or more automata in the network with "more abstract" automata. The idea is that this should result in an abstract network which (1) contains (at least) all runs of the original one, and (2) with no larger costs. Thus computing the minimum cost of reaching a goal state in the abstract network will give the desired lower bound estimate of reaching a goal state in the original network. Moreover, the abstract network should (3) be substantially simpler to analyze than the original network making it possible to obtain the estimate efficiently. We are currently working with different ideas of implementing this idea. In Section 9.4 we have used the idea when guiding systems manually.

9.3.3 Heuristics and Bounding

It is often useful to quickly obtain an upper bound on the cost instead of waiting for the minimum cost. In particular, this is the case when faced with a state-space too big for the MC order to handle. As will be shown in Section 9.4, the techniques described here for altering the search order using heuristics are very useful. In addition, techniques from branch-and-bound algorithms are useful for improving the upper bound once it has been found. Applying knowledge about the goal state has proven useful in improving the state-space exploration [132, 82], either by changing the search order from the standard depth or breadth-first, or by leaving out parts of the state-space.

To implement the MC order, a suitable data-structure for WAITING would be a priority queue where the priority is the minimum cost of a symbolic cost state. We can obviously generalize this by extending a symbolic cost state with a new field, *priority*, which is the priority of the state used by the priority queue. Allowing various ways of assigning values to *priority* combined with choosing either to first select a state with large or small priority opens for a large variety of search orders.

Annotating the model with assignments to *priority* on the transitions, is one way of allowing the user to guide the search. Because of its flexibility it proves to be a very powerful way of guiding the search. The assignment works like a normal assignment to integer variables and allows for the same kind of expressions.

Example 9.2 *An example of a strategy which we have used in Section 9.4 for the Biphase Mark protocol, is first to search a limited part of the state-space in a breadth-first manner. The sender of the protocol can either send the value 0 or 1. We are mainly interested in the part where a 1 is send because we suspect that errors will occur in this case. Therefore, we want to search the part of the state-space where a 1 is send before searching the part where 0 is send, and we want to do this in a breadth first manner.*

Using guiding a standard breadth-first search can be obtained by adding the assignment `priority := priority - 1` to each transition and select the symbolic state with the highest value from WAITING. *This can be done by adding a global assignment to the model. Giving very low priority to the part of the state-space where a 0 has been send we will obtain the desired search order. The choice of what to send is made in one place in the model of the sender. On the transition choosing to send a 0 we add the assignment `priority := priority - 1000` which will give this state and all its successors very low priority and therefore these will be explored last. In this way we do not leave out any part of the state-space, but first search the part we consider to be interesting.* □

When searching for an error state in a system a *random* search order might be useful. We have chosen to implement what we call *random depth-first order* which as the name suggests is a variant of a depth-first search. The only difference between this and a standard depth-first search is that before pushing all the successors of a state on to WAITING (which is implemented as a stack), the successors are randomly permuted.

Once a reachable goal state has been found, an upper bound on the minimum cost of reaching a goal state has been obtained. If we choose to continue the search, a smaller upper bound might be obtained. During state-space exploration the cost never decreases therefore states with cost bigger than the best cost found in a goal state cannot lead to an optimal solution, and can therefore be deleted. The estimate of the remaining cost defined in Section 9.3.2 can also be used for pruning exploration of states since whenever $min(C) + \textsc{Rem}(l, C)$ is larger than the best upper bound, no state covered by (l, C) can lead to a better solution than the one already found.

All of the methods described in this section have been implemented in UPPAAL. Section 9.4 reports on experiments using these new methods.

9.4 Experiments

In this section we illustrate the benefits of extending UPPAAL with heuristics and costs through several verification and optimization problems. All of the examples have previously been studied in the literature.

9.4.1 The Bridge Problem

The following problem was proposed by Ruys and Brinksma [134]. A timed automaton model of this problem is included in the standard distribution of UPPAAL[5].

Four persons want to cross a bridge in the dark. The bridge is damaged and can only carry two persons at the same time. To cross the bridge safely in the darkness, a torch must be carried along. The group has only one torch to share. Due to different physical abilities, the four cross the bridge at different speeds. The time they need per person is (one-way) 25, 20, 10 and 5 minutes, respectively. The problem is to find a schedule, if possible, such that all four cross the bridge within a given time. This can be done with standard UPPAAL. With the proposed extension, it is also possible to find the fastest time for the four to cross the bridge, and a schedule achieving this.

We compare four different search orders: Breadth-First (BF), Depth-First (DF), Minimum Cost (MC) and an improved Minimum Cost (MC+). In this example we choose the lower bound on the remaining cost, REM(C), to be the time needed by the slowest person, who is still on the "wrong" side of the bridge.

For the different search orders, Table 9.3 shows the number of states explored to find the initial and the optimal time, and the values of the times. It can be seen that BF explores 4491 states to find an initial schedule and 4539 to prove what the optimal solution is. This number is reduced to 4493 explored states if we prune the state-space, based on the estimated remaining cost (third column). Thus, in this case only two additional states are explored after the initial solution is found. DF finds an initial solution (with high costs) quickly, but explores 25779 states to find an optimal schedule, which is much more than the other heuristics. Most likely, this is caused by encountering many small and incomparable zones during DF search. In any case, it appears that the depth-first strategy always explores many more states than any other heuristic.

Table 9.3: Bridge problem by Ruys and Brinksma.

	Initial Solution		**Optimal Solution**		**With est. remainder**	
	states	cost	states	cost	states	cost
BF	4491	65	4539	60	4493	60
DF	169	685	25780	60	5081	60
MC	1536	60	1536	60	N/A	N/A
MC+	404	60	404	60	N/A	N/A

Searching with the MC order does indeed improve the results, compared to BF and DF. It is, however, outperformed by the MC+ heuristic that explores only 404 states to find a optimal schedule. Note that pruning based on the estimate of the remaining cost does not apply to MC and MC+ order, since the first explored goal state has the optimal value.

Without costs and heuristics, UPPAAL can only show whether a schedule exists. The extension allows UPPAAL to find the optimal schedule and explores with

[5]The distribution can be obtained at `http://www.uppaal.com`.

the MC+ heuristic only about 10% of the states that are needed to find a initial solution with the breadth-first heuristic.

9.4.2 Job Shop Scheduling

A well known class of scheduling problems are the Job Shop problems. The problem is to optimally schedule a set of *jobs* on a set of *machines*. Each job is a chain of operations, usually one on each machine, and the machines have a limited capacity, also limited to one in most cases. The purpose is to allocate starting times to the operations, such that the overall duration of the schedule, the *makespan*, is minimal. Many solution methods such as local search algorithms like simulated annealing [1], shifting bottleneck [21], branch-and-bound [21] or even hybrid methods have been proposed [93].

We apply UPPAAL to 25 of the smaller Lawrence Job Shop problems.[6] Our models are based on the timed automata models in [63]. In order to estimate the lower bound on the remaining cost, we calculate for each job and each machine the duration of the remaining operations. These estimates may be seen as obtained by abstracting the model to one automaton as described in Section 9.3.2. The final estimate of the remaining cost is then estimated to be the maximum of these durations. Table 9.4 shows results obtained for the search orders MC+, DF, Random DF, and a combined heuristic. The latter is based on depth-first but also takes into account the remaining operation times and the lower bound on the cost, via a weighted sum which is assigned to the priority field of the symbolic states. We also experimented with using BF and MC order, but here no single instance was completed in 60 seconds. Even when allowed to spend more than 30 minutes using more than 2Gb of memory no solution was found using these search orders. It is important to notice that the combined heuristic used includes a clever choice between states with the same values of cost plus remaining cost. This is the reason it is able to outperform the MC+ order which is only able to find solution to two instances within the time limit of 60 seconds.

As can be seen from the table UPPAAL is handling the first 15 examples quite well finding the optimal solution in 11 cases and in 10 of these showing that it is optimal. This is much more than without the added guiding features. For the 10 largest problems (la16 to la25) with 10 machines we did not find optimal solutions though in some cases we were very close to the optimal solution. Since branch-and-bound algorithms generally do not scale too well when the number of machines and jobs increase, this is not surprising. The branch-and-bound algorithm for [21], who solves about 10 out the 15 problems in the same setting, faces the same problem. Note that the results of this algorithm depend sensitively on the choice of an initial upper bound. Also the algorithm used in [44], which combines a good heuristic with an efficient branch-and-bound algorithm and thus solves all of these 15 instances, does not find solutions for the larger instances with 15 jobs and 10 machines or larger.

[6]These and other benchmark problems for Job Shop scheduling can be found on

Table 9.4: Results for the 15 Job Shop problems with 5 machines and 10 jobs (la1-la5), 15 jobs (la6-la10) and 20 jobs (la11-la15), and 10 problems with 10 machines, 10 jobs (la16-20) and 15 jobs (la21-25). The table shows the best solution found by different search orders within 60 seconds cputime on a Pentium II 300 MHz. If the search terminated also the number of explored states is given. The last row gives the makespan of an optimal solution.

problem	MC+		DF		RDF		comb. heur.		minimal
instance	cost	states	cost	states	cost	states	cost	states	makespan
la01	-	-	2466	-	842	-	666	292	666
la02	-	-	2360	-	806	-	672	-	655
la03	-	-	2094	-	769	-	626	-	597
la04	-	-	2212	-	783	-	639	-	590
la05	593	9791	1955	-	696	-	593	284	593
la06	-	-	3656	-	1076	-	926	480	926
la07	-	-	3410	-	1113	-	890	-	890
la08	-	-	3520	-	1009	-	863	400	863
la09	-	-	3984	-	1154	-	951	425	951
la10	-	-	3681	-	1063	-	958	454	958
la11	-	-	4974	-	1303	-	1222	642	1222
la12	-	-	4557	-	1271	-	1039	633	1039
la13	-	-	4846	-	1227	-	1150	662	1150
la14	1292	10653	5145	-	1377	-	1292	688	1292
la15	-	-	5264	-	1459	-	1289	-	1207
la16	-	-	4849	-	1298	-	1022	-	945
la17	-	-	4299	-	938	-	786	-	784
la18	-	-	4763	-	1034	-	922	-	848
la19	-	-	4566	-	1140	-	904	-	842
la20	-	-	5056	-	1378	-	964	-	902
la21	-	-	7608	-	1326	-	1149	-	(1040,1053)
la22	-	-	6920	-	1413	-	1047	-	927
la23	-	-	7676	-	1357	-	1075	-	1032
la24	-	-	7237	-	1346	-	1061	-	935
la25	-	-	7141	-	1290	-	1070	-	977

9.4.3 The Sidmar Steel Plant

Proving schedulability of an industrial plant via a reachability analysis of a timed automaton model was firstly applied to the SIDMAR steel plant, which was included as case study of the VHS project. It deals with the part of the plant in-between the blast furnace and the hot rolling mill. The plant consists of five machines placed along two tracks and a casting machine where the finished steel leaves the system. The two tracks and the casting machine are connected via two overhead cranes on one track. Figure 9.4 depicts the layout of the plant. Each quantity of raw iron enters the system in a ladle and depending on the desired steel quality undergoes treatments in the different machines of different durations. The aim is to control the plant in particular the movement of the ladles with steel between the different machines, taking the topology of the plant into consideration.

We use a model based on the models and descriptions in [36, 64, 80]. A full model of the plant that includes all possible behaviors was however not immediate suitable for verification. Its state-space is very large since at each point in time many different possibilities are enabled. Consequently, depth-first (and

`ftp://ftp.caam.rice.edu/pub/people/applegate/jobshop/`.

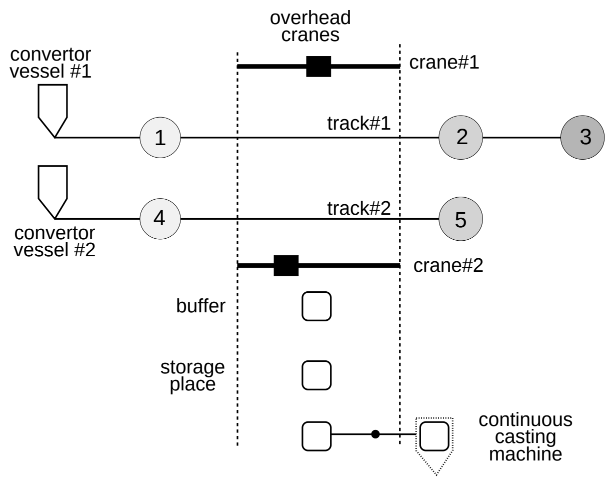


Figure 9.4: Layout of the SIDMAR plant

breadth-first) search gives either no answer within reasonable time or comes up with a solution that is far from optimal. In this way we were only able to find schedules for models of the plant with two ladles.

Priorities can be used to influence the search order of the state-space, and thus to improve the results. Based on a depth-first strategy, we reward transitions that are likely to serve in reaching the goal, whereas transitions that may spoil a partial solution result in lower priorities. For instance, when a batch of iron is being treated by a machine, it pays off to reward other scheduling activities rather than wait for the treatment to finish.

A schedule for three ladles was produced in [64] for a slightly simplified model using Uppaal. In [80] schedules for up to 60 ladles were produced also using Uppaal. However, in order to do this, additional constraints were included that reduce the size of the state-space drastically, but also prune possibly sensible behavior. A similar reduced model was used by Stobbe in [139], who uses constraint programming to schedule 30 ladles. All these works only consider ladles with the same quality of steel and the initial solutions cannot be improved.

A lower bound for the time duration of any feasible schedule is given by the time the first load needs for all treatments plus the least time one needs to cast all batches. Analogously, an upper bound is given by the maximal time during which the first batch is allowed to stay in the system, plus the maximal time needed to cast the ladles. For the instance with ten batches, these bounds are 291 and 425, respectively. With the sketched heuristic, our extended Uppaal is able to find a schedule with duration 355, within 60 seconds cputime on a Pentium II 300 MHz. The initial solution found is improved by 5% within the time limit. Importantly, in this approach we do not rule out optimal solutions. Allowing the search to go on for longer, models with more ladles can be handled.

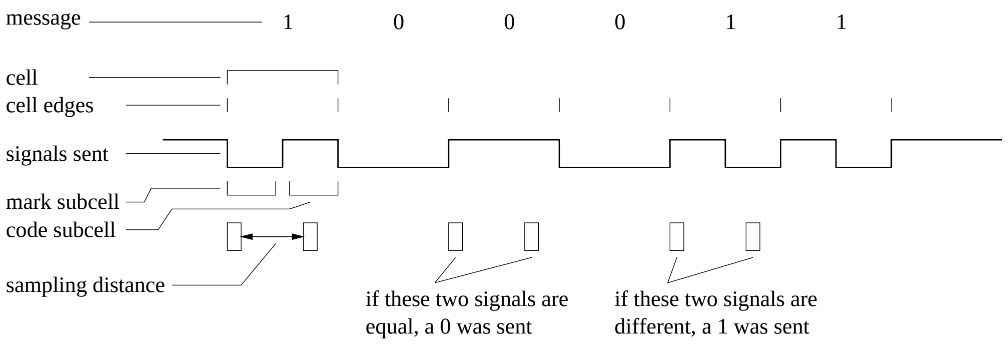

message
1
0
0
0
1
1
cell
cell edges
signals sent
mark subcell
code subcell
sampling distance
if these two signals are equal, a 0 was sent
if these two signals are different, a 1 was sent

Figure 9.5: Biphase mark terminology

9.4.4 Pure Heuristics: The Biphase Mark Protocol

The Biphase Mark protocol is a convention for transmitting strings of bit and clock pulses simultaneously as square waves. This protocol is widely used for communication in the ISO/OSI physical layer; for example, a version called “Manchester” is used in the Ethernet. The protocol ensures that strings of bits can be submitted and received correctly, in spite of clock drift, jitter and filtering by the channel. A formal parameterized timed automaton model of the Biphase Mark Protocol was given in [141], where also necessary and sufficient conditions on the correctness for a parametric model were derived. We will use the corresponding UPPAAL models to investigate the benefits of heuristics in pure reachability analysis.

The model assumes that sender and receiver have both their own clock with drift and jitter. The sender encodes each bit in a *cell* of length c clock cycles (see Fig. 9.5). At the beginning of each cell, the sender toggles the voltage. The sender then waits for m clock cycles, where m stands for the *mark* subcell. If the sender has to encode a “0”, the voltage is held constant throughout the whole cell. If it encodes a “1” it will toggle the voltage at the end of the mark subcell. The signal is unreliable during a small interval after the sender generates an edge. Reading it during this interval may produce any value.

The receiver waits for an edge that signals the arrival of a cell. Upon detecting an edge, the receiver waits for a fixed number of clock cycles, the *sampling distance* s, and samples the signal. We adopt the notation $bpm(c, m, s)$ for instances of the protocol where the cell size c, mark size m and sampling distance s.

There are three kind of errors that may occur in an incorrect configuration. Firstly, the receiver may not detect the mark subcell. Secondly, the receiver may sample too early, before or right after the sender left the mark subcell. Finally, the receiver may also sample too late, i.e. the sender has already started to transmit the next cell. The first two errors can only occur if there is an edge after the mark subcell. This is only the case if input ”1” is offered to the coder. The third error seems to be independent of the offered input.

Since two of the three errors occur only if input ”1” is offered to the coder, and the third error can occur in any case, it seems worthwhile to choose a heuristic that searches for states with input "1" first, rather than exploring state-space for both possible inputs concurrently. We apply a heuristic which is a mixture of only choosing input 1 and the breadth-first order, see Example 9.2, to erroneous modifications of the (correct) instances BPM$(16, 6, 11)$, BPM$(18, 5, 10)$ and BPM$(32, 16, 23)$. Table 9.5 gives the results in terms of the number of symbolic states explored to find an error state. It turns out that using the heuristic a bit more than half the number of symbolic states is explored to find an error state, compared to using breadth-first order. This is due to the fact that for input "1", there is more activity in the protocol. The corresponding diagnostic traces show that the errors were found within the first cell or at the very beginning of the second cell, thus at a stage were only one bit was sent and received. An exception to this rule is the fifth instance BPM$(18, 6, 10)$, which produces an error after one and a half cell, and shows consequently a larger reduction when verified with the heuristic.

Table 9.5: Results for nine erroneous instances of the Biphase Mark protocol. Numbers of state explored before reaching an error state

	nondetection mark subcell			sampling early			sampling late		
	(16,3,11)	(18,3,10)	(32,3,23)	(16,9,11)	(18,6,10)	(32,18,23)	(15,8,11)	(17,5,10)	(31,16,23)
breadth first	1931	2582	4049	990	4701	2561	1230	1709	3035
`in==1` heuristic	1153	1431	2333	632	1945	1586	725	1039	1763

9.5 Conclusion

On the preceding pages, we have contributed with (1) a cost function based symbolic semantics for the class of linearly priced timed automata; (2) an efficient, zone based implementation of cost functions for the class of uniformly priced timed automata; (3) an optimal search order for finding the minimum cost of reaching a goal state; and (4) experimental evidence that these techniques can lead to dramatic reductions in the number of explored states. In addition, we have shown that it is possible to quickly obtain upper bounds on the minimum cost of reaching a goal state by manually guiding the exploration algorithm using priorities.

Chapter 10

Efficient Representation of Linear Cost

The paper *As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata* presented in this chapter has not been published yet, but is accepted for Computer Aided Verification 2001.

[105] K. G. Larsen, G. Behrmann, E. Briksma, A. Fehnker, T. Hune, P. Pettersson and J. Romijn. As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata. To appear in *Proceedings of Computer Aided Verification*, 2001.

The content of this chapter is a slightly modified version of the submitted paper. My contribution to this paper has mainly been in the initial phase of the work.

As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata

Kim Larsen[†][*] Gerd Behrmann[*] Ed Brinksma[†]
Ansgar Fehnker[§] Thomas Hune[‡] Paul Pettersson[¶]
Judi Romijn[§]

Abstract

In this paper we present an algorithm for efficiently computing optimal cost of reaching a goal state in the model of Linearly Priced Timed Automata (LPTA). In recent papers, this problem have been shown to be computable using a priced extension of the traditional notion of regions for timed automata. However, for efficiency it is imperative that the computation is based on so-called zones (i.e. convex set of clock valuations) rather than regions. The central contribution of this paper is a priced extension of zones. This, together with a notion of facets of a zone, allows the entire machinery for symbolic reachability in terms of zones to be lifted to cost-optimal reachability using priced zones. We report on experiments with a cost-optimizing extension of Uppaal on a number of examples, including a range of aircraft landing problems.

10.1 Introduction

Well-known formal verification tools for real-time and hybrid systems, such as Uppaal [109], Kronos [41] and HyTech [74], use symbolic techniques to deal with the infinite state-spaces that are caused by the presence of continuous variables in the associated verification models. The use of such techniques,

[*]**B**asic **R**esearch **I**n **C**omputer **S**cience, BRICS, (www.brics.dk), funded by the Danish National Research Foundation, University of Aalborg, Denmark, Email: `[kgl,behrmann]@cs.auc.dk`

[†]Department of Computer Systems, University of Twente, Email: `brinksma@cs.utwente.nl`

[‡]**B**asic **R**esearch **I**n **C**omputer **S**cience, BRICS, (www.brics.dk), funded by the Danish National Research Foundation, University of Århus, Denmark, Email: `baris@brics.dk`

[§]Computing Science Institute, University of Nijmegen, Email: `[ansgar,judi]@cs.kun.nl`

[¶]Department of Information Technology, Uppsala University, Email: `paupet@docs.uu.se`.

however, generally does not result in small symbolic state-spaces. In fact, symbolic model checkers share the “state-space explosion problem” with their non-symbolic counterparts as the major obstacle for their application to non-trivial problems. A lot of research, therefore, is devoted to the containment of this problem by sophisticated techniques, such as data structures for compact state-space representation, smart state-space search strategies, etc.

An interesting idea for model checking of reachability properties that has received more attention recently is to “guide” the exploration of the (symbolic) state-space such that “promising” sets of states are visited first. In a number of recent publications [64, 82, 26, 127, 43] model checkers have been used to solve a number of non-trivial scheduling problems. Scheduling problems can often be reformulated in terms of reachability, viz. as the (im)possibility to reach a state that improves on a given optimality criterion. Such criteria distinguish scheduling algorithms from classical, full state-space exploration model checking algorithms. They are used together with, for example, branch-and-bound techniques [21] to prune parts of the search tree that are guaranteed not to contain optimal solutions. This observation motivates research into the extension of model checking algorithms with optimality criteria. They provide a basis for the guided exploration of state-spaces, and improve the potential of model checking techniques for the resolution of scheduling problems.

We believe that such extensions can be interesting for real-life applications of both model checking and scheduling. As full verification by model checking is usually not a practical option for real systems because of the implied combinatorial explosion, it is often restricted to a form of systematic debugging, in which one checks for a certain class of errors in a certain number of scenarios. Such errors usually appear as reachable faulty states of the relevant verification model(s). In many cases time- or cost-driven exploration strategies can help to find (more) errors more efficiently than standard depth- or breadth-first searches. To a much larger extent than model checking, scheduling is backed up by an impressive body of combinatorial mathematics that can guarantee (near) optimal results in many cases. In practice their usefulness is often restricted when problems fail to meet the particular assumptions of the relevant theory. In such cases (extended) model checking techniques provide a very generic approach to finding solutions. Moreover, if the dimensions of the problem do not exceed the available resources right from the start, it is scalable in the sense that one may approach optimal solutions by repeated checking for reachable states of successively improving quality.

Based on similar observations an extension of the timed automata model with a notion of *cost*, the *Linearly Priced Timed Automata* (LPTA), was already introduced in [28]. This model allows for a reachability analysis in terms of accumulated cost of traces, i.e. the sum of the costs of the individual transitions in the trace. Each action transitions has an associated price p determining its cost. Likewise, each location has an associated rate r and the cost of delaying d time units is $d \cdot r$. In [28] a symbolic semantics of LPTA is provided, based on which computability of minimal-cost reachability is demonstrated. The symbolic semantics is based on *linearly priced clock regions* extending the traditional region construct with cost information. Similar and independent work has been presented by Alur et al. [17], which additionally provides complexity bounds for the problem.

Although ensuring computability, the region construction is known to be very inefficient and highly sensitive to the size of constants used in the models. Tools like Uppaal and Kronos use symbolic states of the form (l, Z), where l is a location of the timed automaton and Z is a *zone*, i.e. a convex set of clock valuations. The central contribution of this paper is the extension of this concept to that of *priced zones*. Priced zones are attributed with an (affine) linear function of clock valuations that defines the cost of reaching a valuation in the zone. We show that the entire machinery for symbolic reachability in terms of zones can be lifted to cost-optimal reachability for priced zones. It turns out that some of the operations on priced zones force us to split them into parts with different price attributes, giving rise to a new notion, viz. that of the *facets* of a zone.

The suitability of the LPTA model for scheduling problems was already illustrated in [26], using the more restricted *Uniformly Priced Timed Automata* (UPTA) model. This model allows only a uniform cost rate for all locations, which admits an efficient priced zone implementation via *Difference Bound Matrices* [60]. The model was used to consider traces for the time-optimal scheduling of a steel plant and a number of job shop problems. The greater expressivity of LPTA also supports other measures of cost, like idle time, weighted idle time, mean completion time, earliness, number of tardy jobs, tardiness, etc. We take an aircraft landing problem [25] and an extension of the bridge problem [134] as the application examples for this paper.

The structure of the rest of this paper is as follows. In Section 10.2 we give an abstract account of symbolic optimal reachability in terms of *priced transition systems*, including a generic algorithm for optimal reachability. In Section 10.3 we introduce the model of linearly priced timed automata (LPTA) as a special case of the framework of Section 10.2. We also introduce here our running application example, the aircraft landing problem. Section 10.4 contains the definition of the central concept of priced zones. The operations that we need on priced zones and facets are provided in Section 10.5. The implementation of the algorithm, and the results of experimentation with our examples are reported in Section 10.6. Our conclusions, finally, are presented in Section 10.7.

10.2 Symbolic Optimal Reachability

Analysis of infinite state systems require symbolic techniques in order to effectively represent and manipulate sets of states simultaneously (see e.g. [66, 4, 5, 65, 50]). For analysis of cost-optimality, additional information of *costs* associated with individual states needs to be represented. In this section, we describe a general framework for symbolic analysis of cost-optimal reachability on the abstract level of priced transition systems. In the following sections we shall instantiate the framework to linearly priced timed automata.

A *priced transition system* is a structure $\mathcal{T} = (S, s_0, \Sigma, \rightarrow)$, where S is a (infinite) set of states, $s_0 \in S$ is the initial state, Σ is a (finite) set of labels, and, $\rightarrow$ is a partial function from $S \times \Sigma \times S$ into the non-negative reals, $\mathsf{R}_{\geq 0}$, defining the possible transitions of the systems as well as their associated costs. We write $s \xrightarrow{a}_p s'$ whenever $\rightarrow (s, a, s')$ is defined and equals p. Intuitively, $s \xrightarrow{a}_p s'$ indicates that the system in state s has an a-labeled transition to the state s' with the cost of p. We denote by $s \xrightarrow{a} s'$ that $\exists p \in \mathsf{R}_{\geq 0}.\, s \xrightarrow{a}_p s'$, and, by $s \rightarrow s'$ that $\exists a \in \Sigma.\, s \xrightarrow{a} s'$.

Now, an execution of $\mathcal{T}$ is a sequence $\alpha = s_0 \xrightarrow{a_1}_{p_1} s_1 \xrightarrow{a_2}_{p_2} s_2 \cdots \xrightarrow{a_n}_{p_n} s_n$. The *cost* of α, $\mathsf{cost}(\alpha)$, is the sum $\sum_{i \in \{1 \ldots n\}} p_i$. For a given state s, the *minimal cost* of reaching s, $\mathsf{mincost}(s)$, is the infimum of the costs of finite executions starting in the initial state s_0 and ending in s. Similar, the minimal cost of reaching a designated set of states $G \subseteq S$, $\mathsf{mincost}(G)$, is the infimum of the costs of finite executions ending in a state of G.

To decide reachability and compute minimum-cost reachability in the infinite state cases, we need to represent and simultaneously handle (certain) sets of states. Representable sets of states – referred to as so-called symbolic states – should in particular be closed w.r.t. the following notion of successor-set. For $A \subseteq S$ and $a \in \Sigma$, we denote by $post_a(A)$ the set of all a-successors to states of A, i.e. $\{s' \mid \exists s \in A.\, s \xrightarrow{a} s'\}$. Starting from the singleton set $\{s_0\}$, repeated application of these *post*-operators leads to exploration of the symbolic state-space and hence identification of the reachable set of states. Provided symbolic states ordered by set-inclusion constitutes a well-quasi ordering,[1] termination may be obtained by refraining from exploration of symbolic states included in already explored ones.

To compute minimum-cost reachability, we suggest the use of *priced symbolic states* of the form (A, π), where $A \subseteq S$ is a set of states, and $\pi : A \longrightarrow \mathsf{R}_{\geq 0}$ assigns (non-negative) costs to all states of A. The intention is that, reachability of the priced symbolic state (A, π) should ensure, that any state s of A is reachable with cost arbitrarily close to $\pi(s)$. As we are interested in minimum-cost reachability, π should preferably return as small cost values as possible. This is obtained by the following extension of the *post*-operators to priced symbolic states: for (A, π) a priced symbolic state and $a \in \Sigma$, $Post_a(A, \pi)$ is the priced symbolic state (B, η), where $B = post_a(A)$ and η is given by:

$$\eta(s) = \inf\{\pi(s') + p \mid s' \in A \wedge s' \xrightarrow{a}_p s\} \tag{10.1}$$

That is, η essentially gives the cheapest cost for reaching states of B via states in A, assuming that these may be reached with costs according to π. A symbolic execution of a priced transition system, $\mathcal{T}$, is a sequence $\beta = (A_0, \pi_0)$, $(A_1, \pi_1), \ldots, (A_n, \pi_n)$, where for $i < n$, $(A_{i+1}, \pi_{i+1}) = Post_{a_i}(A_i, \pi_i)$ for some $a_i \in \Sigma$, and $A_0 = \{s_0\}$ and $\pi_0(s_0) = 0$. It is not difficult to see, that there is a very close connection between executions and symbolic executions: for any execution α of $\mathcal{T}$ ending in a state s, there is a symbolic execution β of $\mathcal{T}$, that

[1]An ordering $(A, \sqsubseteq)$ is well-quasi iff for any infinite sequence $a_0, a_1, a_2, \ldots, a_l, \ldots$ of elements of A it holds that $a_j \sqsubseteq a_k$ for some $j < k$.

ends in a priced symbolic state (A, π), such that $s \in A$ and $\pi(s) \leq \mathsf{cost}(\alpha)$. Dually, for any symbolic execution β of $\mathcal{T}$ ending in priced symbolic state (A, π), whenever $s \in A$, then $\mathsf{mincost}(s) \leq \pi(s)$. We formalize this in the following two lemmas.

Lemma 10.1 *Let $\mathcal{T}$ be a priced transition system. Then for any execution α of $\mathcal{T}$ ending in state s, there is a symbolic execution β of $\mathcal{T}$, that ends in a priced symbolic state (A, π), such that $s \in A$ and $\pi(s) \leq \mathsf{cost}(\alpha)$.*

Proof. We prove this by induction on the length of α. The base case, $|\alpha| = 0$, is obvious. For the induction-step consider the following execution of length $n+1$:

$$\alpha = s_0 \xrightarrow{a_1}_{p_1} s_1 \xrightarrow{a_2}_{p_2} s_2 \cdots \xrightarrow{a_n}_{p_n} s_n \xrightarrow{a_{n+1}}_{p_{n+1}} s_{n+1}$$

By the induction hypothesis, there exists a symbolic execution β ending in a priced symbolic state (A, π) such that $s_n \in A$ and $\pi(s_n) \leq \sum_{i \leq n} p_i$. Now, extend β with $(B, \eta) = Post_{a_{n+1}}(A, \pi)$. Then clearly $s_{n+1} \in B$ and $\eta(s_{n+1}) \leq \pi(s_n) + p_{n+1} \leq \mathsf{cost}(\alpha)$. □

Lemma 10.2 *Let $\mathcal{T}$ be a priced transition system. Then, for any symbolic execution β of $\mathcal{T}$ ending in priced symbolic state (A, π), whenever $s \in A$, then $\mathsf{mincost}(s) \leq \pi(s)$.*

Proof. We prove this by induction on the length of β. The base case, $|\beta| = 0$, is obvious. For the induction-step consider a symbolic execution of length $n+1$, $\beta = (A_0, \pi_0), \ldots, (A_n, \pi_n), (A_{n+1}, \pi_{n+1})$, where for $i \leq n$, $(A_{i+1}, \pi_{i+1}) = Post_{a_i}(A_i, \pi_i)$ for some $a_i \in L$, and $A_0 = \{s_0\}$ and $\pi_0(s_0) = 0$. Now, let $s_{n+1} \in A_{n+1}$. Completion of the induction-step is obtained from the following:

$$\begin{array}{rcl} \pi_{n+1}(s_{n+1}) & = & inf\{\, \pi_n(s_n) + p_{n+1} \,|\, s_n \in A_n \wedge s_n \xrightarrow{a_{n+1}}_{p_{n+1}} s_{n+1} \,\} \\ & \geq_{IH} & inf\{\, \mathsf{mincost}(s_n) + p_{n+1} \,|\, s_n \in A_n \wedge s_n \xrightarrow{a_{n+1}}_{p_{n+1}} s_{n+1} \,\} \\ & \geq^* & \mathsf{mincost}(s_{n+1}) \end{array}$$

where $*$ follows from $inf\{inf(A_i) + p_i : i \in I\} = inf(A_i + p_i : i \in I\}$, whenever $A_i \subseteq \mathsf{R}_{\geq 0}$, $p_i \in \mathsf{R}_{\geq 0}$ for all $i \in I$, and $A+p$ is shorthand for the set $\{q+p : q \in A\}$. □

Combining Lemma 10.1 and Lemma 10.2, we obtain directly that the symbolic semantics using priced symbolic states accurately captures minimum cost reachability:

Theorem 10.1 $\mathsf{mincost}(G) = \inf\{\mathsf{mincost}(A \cap G, \pi) : (A, \pi)$ *is reachable*$\}$

Let (A, π) and (B, η) be priced symbolic states. We write $(A, \pi) \sqsubseteq (B, \eta)$ if $B \subseteq A$ and $\pi(s) \leq \eta(s)$ for all $s \in B$, informally expressing, that (A, π) is "as

```
COST := $\infty$
PASSED := $\emptyset$
WAITING := $\{(\{s_0\}, \pi_0)\}$
while WAITING $\neq \emptyset$ do
    select $(A, \pi)$ from WAITING
    if $A \cap G \neq \emptyset$ and $minCost(A \cap G, \pi) <$ COST then
        COST := $minCost(A \cap G, \pi)$
    if for all $(B, \eta)$ in PASSED: $(B, \eta) \not\sqsubseteq (A, \pi)$ then
        add $(A, \pi)$ to PASSED
        add $Post_a(A, \pi)$ to WAITING for all $a \in \Sigma$
return COST
```

Figure 10.1: Abstract Algorithm for the Minimal-Cost Reachability Problem.

big and cheap" as (B, η). Also, we denote by $minCost(A, \pi)$ the infimum costs in A w.r.t. π, i.e. $\inf\{\pi(s) \mid s \in A\}$.

Now using the above notion of priced symbolic state and associated operations, an abstract algorithm for computing the minimum cost of reaching a designated set of goal states G is shown in Fig.10.1. The algorithm is similar to the iterative procedure for deciding reachability. It uses two data-structures WAITING and PASSED to store priced symbolic states waiting to be examined, and priced symbolic states already explored, respectively. Initially, PASSED is empty and WAITING holds an initial priced symbolic state. In each iteration, the algorithm proceeds by selecting a priced symbolic state (A, π) from WAITING, checking that none of the previously explored states (B, η) are bigger and cheaper, i.e. $(B, \eta) \not\sqsubseteq (A, \pi)$, and adds it to PASSED and its successors to WAITING. In addition, the algorithm uses the global variable COST, which is initially set to ∞ and updated whenever a goal state is found that can be reached with lower cost than the current value of COST. The algorithm terminates when WAITING is empty, i.e. when no further priced symbolic states are left to be examined.

We denote by $(A, \pi) \rightsquigarrow (B, \eta)$ that $(B, \eta) = Post_a(A, \pi)$ for some $a \in L$, and we say that (A, π) can reach (B, η) if $(A, \pi) \rightsquigarrow^* (B, \eta)$. Now, $\sqsubseteq$ is a simulation preorder in the following sense:

Lemma 10.3 *Whenever $(B_1, \eta_1) \sqsubseteq (A_1, \pi_1)$ and $(A_1, \pi_1) \rightsquigarrow (A_2, \pi_2)$, then $(B_1, \eta_1) \rightsquigarrow (B_2, \eta_2)$ for some (B_2, η_2) such that $(B_2, \eta_2) \sqsubseteq (A_2, \pi_2)$.*

Now, assume that the algorithm Fig. 10.1 terminates. Let $\mathcal{W}$ denote the total (finite) collection of priced symbolic states, which will have been in the WAITING list at some stage during the execution of the algorithm. $\mathcal{W}$ is closed with respect $\rightsquigarrow$ up to $\sqsubseteq$ in the following sense:

Lemma 10.4 *Whenever $(A, \pi) \in \mathcal{W}$ and $(A, \pi) \rightsquigarrow^* (B, \eta)$, then there exists $(C, \rho) \in \mathcal{W}$ such that $(C, \rho) \sqsubseteq (B, \eta)$.*

Proof. Due to Lemma 10.3 it suffices to consider the case when $(A, \pi) \rightsquigarrow (B, \eta)$ (the general Lemma will then follow as an easy induction exercise). At the moment when (A, π) was selected from the WAITING list two scenarios could have taken place: either (A, π) was explored (i.e. all *Post*-successors added to WAITING), in which case, the Lemma holds trivially, or exploration of (A, π) was discarded due to the fact, that a better (in terms of $\sqsubseteq$) priced symbolic state (D, ψ) already had been previously explored and was present in PASSED. However, as $(D, \psi) \sqsubseteq (A, \pi)$, it follows from Lemma 10.3, that $(D, \psi) \leadsto (C, \rho)$ for some (C, ρ) with $(C, \rho) \sqsubseteq (B, \eta)$. As (D, ψ) is explored obviously $(C, \rho) \in \mathcal{W}$. □

We are now in a position where we can prove that the algorithm is partially correct in the following sense:

Theorem 10.2 *When the algorithm of Fig. 10.1 terminates, the value of* COST *equals* $\mathsf{mincost}(G)$.

Proof. Using Lemma 10.1, we must prove that upon termination COST equals $inf\{\mathsf{mincost}(A \cap G, \pi) : (A, \pi) \text{ is reachable}\}$. Assume that this does not hold. Then there exists a reachable priced symbolic state (A, π) where $\mathsf{mincost}(A \cap G, \pi) <$COST (since we approximate COST from above). Thus, (A, π) never appeared in the WAITING list; rather, the algorithm must at some point have discarded a state (A', π') on the path to (A, π) due to the fact, that a better (bigger and cheaper) priced symbolic state already had been explored and was present in PASSED. However, as $(A_0, \pi_0) \leadsto^* (A, \pi)$ and obviously $(A_0, \pi_0) \in \mathcal{W}$, it follows from Lemma 10.4, that $(B, \eta) \sqsubseteq (A, \pi)$ for some $(B, \eta) \in \mathcal{W}$. But then COST$\leq \mathsf{mincost}(B \cap G, \eta) \leq \mathsf{mincost}(A \cap G, \pi)$ contradiction our assumption that $\mathsf{mincost}(A \cap G, \pi) <$COST. □

Termination of the algorithm will be guaranteed provided $\sqsubseteq$ is a well-quasi ordering on priced symbolic states.

The above framework may be instantiated by providing concrete syntax for priced transition systems, together with data-structures for priced symbolic states allowing for computation of the *Post*-operations, *minCost*, as well as $\sqsubseteq$ (which should be well-quasi). In the following sections we provide such an instantiation for a priced extension of timed automata.

10.3 Priced Timed Automata

Linearly priced timed automata (LPTA) [28, 26, 17] extend the model of timed automata [12] with *prices* on all edges and locations. In these models, the cost of taking an edge is the price associated with it, and the price of a location gives the cost-*rate* applied when delaying in that location.

Let $\mathbb{C}$ be a set of clocks. Then $\mathcal{B}(\mathbb{C})$ is the set of formulas that are conjunctions of atomic constraints of the form $x \bowtie n$ and $x - y \bowtie m$ for $x, y \in \mathbb{C}$, $\bowtie \in \{\leq, =, \geq\}$,[2] n a natural number, and m an integer. Elements of $\mathcal{B}(\mathbb{C})$ are called clock

[2]For simplicity we do not deal with strict inequalities in this short version.

constraints or zones over $\mathbb{C}$. $\mathcal{P}(\mathbb{C})$ denotes the power set of $\mathbb{C}$. Clock values are represented as functions from $\mathbb{C}$ to the non-negative reals $\mathsf{R}_{\geq 0}$, called clock valuations. We denote by $\mathsf{R}^{\mathbb{C}}$ the set of clock valuations for $\mathbb{C}$. For $u \in \mathsf{R}^{\mathbb{C}}$ and $g \in \mathcal{B}(\mathbb{C})$, we denote by $u \in g$ that u satisfies all constraints of g.

Definition 10.1 (Linearly Priced Timed Automata) *We define a linearly priced timed automaton $\mathcal{A}$ over clocks $\mathbb{C}$ as a tuple (L, l_0, E, I, P), where L is a finite set of locations, l_0 is the initial location, $E \subseteq L \times \mathcal{B}(\mathbb{C}) \times \mathcal{P}(\mathbb{C}) \times L$ is the set of edges, where an edge contains a source, a guard, a set of clocks to be reset, and a target, $I : L \to \mathcal{B}(\mathbb{C})$ assigns invariants to locations, and $P : (L \cup E) \to \mathsf{N}$ assigns prices to both locations and edges. In the case of $(l, g, r, l') \in E$, we write $l \xrightarrow{g,r} l'$.*

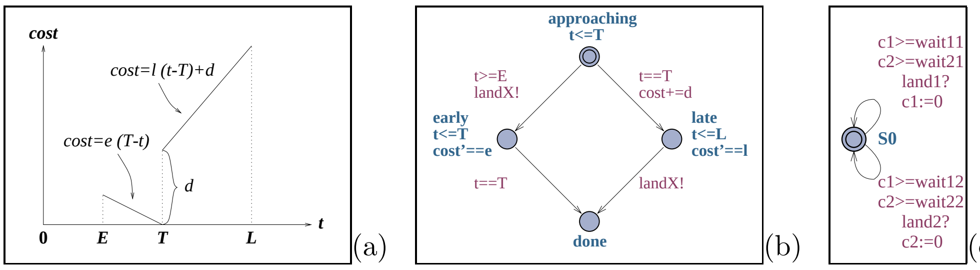


Figure 10.2: Figure (a) depicts the cost of landing a plane at time t. Figure (b) shows an LPTA modeling the landing costs. Figure (c) shows an LPTA model of the runway.

The semantics of a linearly priced timed automaton $\mathcal{A} = (L, l_0, E, I, P)$ may now be given as a priced transition system with state-space $L \times \mathsf{R}^{\mathbb{C}}$ with the initial state (l_0, u_0) (where u_0 assigns zero to all clocks in $\mathbb{C}$), and with the finite label-set $\Sigma = E \cup \{\delta\}$. Thus, transitions are labelled either with the symbol δ (indicating some delay) or with an edge e (the one taken). More precisely, the priced transitions are given as follows:

- $(l, u) \xrightarrow{\delta}_p (l, u + d)$ if $\forall 0 \leq e \leq d : u + e \in I(l)$, and $p = d \cdot P(l)$,
- $(l, u) \xrightarrow{e}_p (l', u')$ if $e = (l, g, r, l') \in E$, $u \in g$, $u' = u[r \mapsto 0]$, $u' \in I(l')$, and $p = P(e)$,

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

Example 10.1 (Aircraft Landing Problem) *As an example of the use of LPTAs we consider the problem of scheduling aircraft landings at an airport, due to [25]. In the example we assume that several automata $A_1, ..., A_n$ can be composed in parallel with a CCS-like parallel composition operator [121] to a network $(A_1, ..., A_n) \backslash Act$, with all actions Act being restricted. We further assume that the cost of delaying in the network is the sum of the cost of delaying* *in the individual automata. For each aircraft there is a maximum speed and a most fuel efficient speed which determine an earliest and latest time the plane can land. In this interval, there is a preferred landing time called target time at which the plane lands with minimal cost. The target time and the interval are shown as T and $[E, L]$ respectively in Fig. 10.2(a). For each time unit the actual landing time deviates from the target time, the landing cost increases with rate e for early landings and rate l for late landings. In addition there is a fixed cost d associated with late landings.*

In Fig. 10.2(b) the cost of landing an aircraft is modeled as an LPTA. The automaton starts in the initial location `approaching` *and lands at the moment one of the two transitions labeled* `landX!` *are taken. In case the plane lands too early it enters location* `early` *in which it delays exactly $T - t$ time units. In case the plane is late the cost is measured in location* `late` *(i.e. the delay in location* `late` *is 0 if the plane is on target time). After L time units the automaton always ends in location done.*

Figure 10.2(c) models the runway which ensures that two consecutive landings takes place with a minimum separation time. It will be explained in detail in Section 10.6. □

10.4 Priced Zones

Typically,[3] reachability of a (priced) timed automaton, $\mathcal{A} = (L, l_0, E, I, P)$, is decided using symbolic states represented by pairs of the form (l, Z), where l is a location and Z is a zone. Semantically, (l, Z) represents the set of all states (l, u), where $u \in Z$. Whenever Z is a zone and r a set of clocks, we denote by $Z^{\uparrow}$ and $\{r\}Z$ the set of clock valuations obtained by delaying and resetting (w.r.t. r) clock valuations from Z respectively. That is, $Z^{\uparrow} = \{u + d \mid u \in Z, d \in \mathsf{R}_{\geq 0}\}$ and $\{r\}Z = \{u[r \mapsto 0] \mid u \in Z\}$. It is well-known – using a canonical representation of zones such as *Difference Bounded Matrices* (DBMs) [60] – that in both cases the resulting set is again effectively representable as a zone.[4] Using these operations together with the obvious fact, that zones are closed under conjunction, the *post*-operations may now be effectively realized using the zone-based representation of symbolic states as follows:

- $post_\delta\big((l, Z)\big) = \big(l, (Z \wedge I(l))^{\uparrow} \wedge I(l)\big)$,
- $post_e\big((l, Z)\big) = \big(l', I(l') \wedge (\{r\}(Z \wedge g))\big)$ whenever $e = (l, g, r, l')$.

Now, the framework given in Section 10.2 for symbolic computation of minimum-cost reachability calls for an extension of our zone-based representation of symbolic states, which assigns costs to individual states. For this, we introduce the following notion of a *priced* zone, where the *offset*, Δ_Z, of a zone Z is the

[3]As is the case in the verification tools KRONOS [41] and UPPAAL [109].

[4]Thus, we shall not distinguish between zones (i.e. a conjunction of simple constraints) and sets of clock valuations representable as zones.

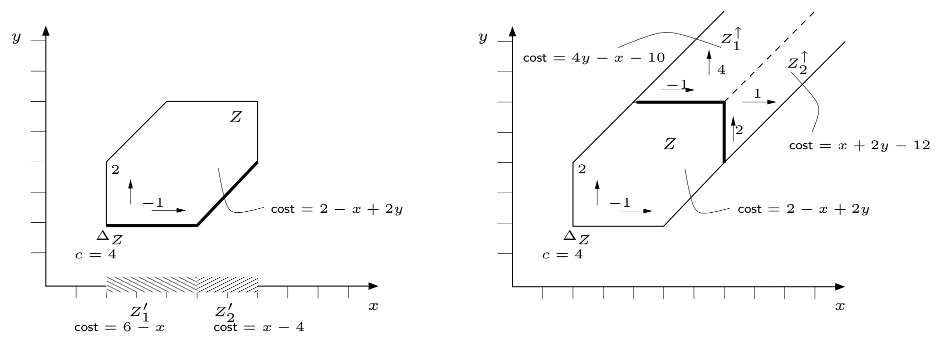


Figure 10.3: A Priced Zone and Successor-Sets

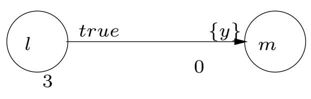


Figure 10.4: Part of a small LPTA $\mathcal{A}$.

unique clock valuation of Z satisfying $\forall u \in Z . \forall x \in \mathbb{C}.\, \Delta_Z(x) \leq u(x)$. Using the canonical DBM representation of Z, Δ_Z is easily computed.

Definition 10.2 (Priced Zone) *A priced zone $\mathcal{Z}$ is a tuple (Z, c, r), where Z is a zone, $c \in \mathsf{N}$ describes the cost of the offset, Δ_Z, of Z, and $r : \mathbb{C} \to \mathsf{Z}$ assigns a cost-rate $r(x)$ for any clock x. We write $u \in \mathcal{Z}$ whenever $u \in Z$. For any $u \in \mathcal{Z}$ the cost of u in $\mathcal{Z}$, $\mathsf{cost}(u, \mathcal{Z})$, is defined as $c + \sum_{x \in \mathbb{C}} r(x) \cdot (u(x) - \Delta_Z(x))$.*

Thus, the cost assignments of a priced zone define a linear plane over the underlying zone and may alternatively be described by a linear expression over the clocks. Figure 10.3 illustrates the priced zone $\mathcal{Z} = (Z, c, r)$ over the clocks $\{x, y\}$, where Z is given by the six constraints $2 \leq x \leq 7$, $2 \leq y \leq 6$ and $-2 \leq x - y \leq 3$, the cost of the offset ($\Delta_Z = (2, 2)^5$) is $c = 4$, and the cost-rates are $r(x) = -1$ and $r(y) = 2$. Hence, the cost of the clock valuation $(5.1, 2.3)$ is given by $4 + (-1) \cdot (5.1 - 2) + 2 \cdot (2.3 - 2) = 1.5$. In general the costs assigned by $\mathcal{Z}$ may be described by the linear expression $2 - x + 2y$.

Now, priced symbolic states are represented in the obvious way by pairs $(l, \mathcal{Z})$, where l is a location and $\mathcal{Z}$ a priced zone. More precisely, $(l, \mathcal{Z})$ represents the priced symbolic state (A, π), where $A = \{(l, u) \mid u \in \mathcal{Z}\}$ and $\pi(l, u) = \mathsf{cost}(u, \mathcal{Z})$.

Unfortunately, priced symbolic states are *not* directly closed under the *Post*-operations. To see this, consider the part of an LPTA $\mathcal{A}$ in Fig. 10.4. Here the cost of the edge is zero and the cost-rate of l is 3. Now, let $\mathcal{Z} = (Z, c, r)$ be the priced zone depicted in Fig. 10.3 and consider the associated priced symbolic state $(l, \mathcal{Z})$. Assuming that the e-successor set, $Post_e(l, \mathcal{Z})$, was expressible as a single priced symbolic state $(l', \mathcal{Z}')$, this would obviously require $l' = m$ and $\mathcal{Z}' = (Z', c', r')$ with $Z' = \{y\}Z$. Furthermore, following

[5]Here we identify clock valuations over x and y with pairs.

(10.1) in our framework of Section 10.2, the cost-assignment of $\mathcal{Z}'$ should be such that $\mathsf{cost}(u', \mathcal{Z}') = inf\{\mathsf{cost}(u, \mathcal{Z}) \mid u \in \mathcal{Z} \wedge u[y \mapsto 0] = u'\}$ for all $u' \in \mathcal{Z}'$. Since $r(y) > 0$, it is obvious that these infima are obtained along the lower boundary of Z with respect to y (see Fig. 10.3 left). E.g. $\mathsf{cost}((2,0), \mathcal{Z}') = 4$, $\mathsf{cost}((4,0), \mathcal{Z}') = 2$, and $\mathsf{cost}((6,0), \mathcal{Z}') = 2$. In general $\mathsf{cost}((x,0), \mathcal{Z}') = \mathsf{cost}((x,2), \mathcal{Z}) = 6 - x$ for $2 \leq x \leq 5$ and $\mathsf{cost}((x,0), \mathcal{Z}') = \mathsf{cost}((x, x-3), \mathcal{Z}) = x - 4$ for $5 \leq x \leq 7$. However, the disagreement w.r.t. the cost-rate of x (-1 or 1) makes it clear that the desired cost-assignment is *not* linear and hence not obtainable from any *single* priced zone. On the other hand, it is also shows that splitting $Z' = \{y\}Z$ into the sub-zones $Z'_1 = Z' \wedge 2 \leq x \leq 5$ and $Z'_2 = Z' \wedge 5 \leq x \leq 7$, allows the e-successor set $Post_e(l, \mathcal{Z})$ to be expressed using the union of *two* priced zones (with $r(x) = -1$ in Z'_1 and $r(x) = 1$ in Z'_2).

Similarly, priced symbolic states are *not* directly closed w.r.t. $Post_\delta$. To see this, consider again the LPTA $\mathcal{A}$ of Fig. 10.4 and the priced zone $\mathcal{Z} = (Z, c, r)$ depicted in Fig. 10.3. Clearly, the set $Post_\delta(l, \mathcal{Z})$ must cover the zone $Z^\uparrow$ (see Fig. 10.3). As for the cost-assignment to be prescribed by $Post_\delta(l, \mathcal{Z})$ it is crucial to compare the cost-rate of l (here $P(l) = 3$) with the sum of clock cost-rates of $\mathcal{Z}$ (here $r(x) + r(y) = 1$). As in this case $r(x) + r(y) \leq P(l)$, the clock valuations (x, y) in $Z^\uparrow \backslash Z$ are reached most cheaply from Z by delaying from the the *upper* boundary of Z, i.e. from valuations $(x' - y' + 6, 6)$ or $(7, y' - x' + 7)$ depending on whether $x - y \leq 1$ or $x - y \geq 1$ (see Fig. 10.3 right). The resulting cost is $4y - x - 10$ and $x + 2y - 12$ respectively. Clock valuations (x, y) in Z can obviously be reached by delay from all valuations within Z of the form $(x - \epsilon, y - \epsilon)$, $\epsilon \in \mathsf{R}_{\geq 0}$. However, again since the cost-rate of l exceeds $r(x) + r(y)$, it is clear that the minimum cost is obtained when $\epsilon = 0$. It follows that, although $Post_\delta(l, \mathcal{Z})$ is not expressible as a *single* priced symbolic state, it may be expressed as a *finite union* by splitting the zone $Z^\uparrow$ into the three sub-zones Z, $Z^\uparrow_1 = (Z^\uparrow \backslash Z) \wedge (x - y \leq 1)$, and $Z^\uparrow_2 = (Z^\uparrow \backslash Z) \wedge (x - y \geq 1)$.

10.5 Facets & Operations on Priced Zones

The universal key to expressing successor sets of priced symbolic states as finite unions is provided by the notion of *facets* of a zone Z. Formally, whenever $x \bowtie n$ ($x - y \bowtie m$) is a constraint of Z, the strengthened zone $Z \wedge (x = n)$ ($Z \wedge (x - y = m)$) is a facet of Z. Facets derived from lower bounds on individual clocks, $x \geq n$, are classified as *lower facets*, and we denote by $LF(Z)$ the collection of all lower facets of Z. Similarly, the collection of *upper facets*, $UF(Z)$, of a zone Z is derived from upper bounds of Z. We refer to lower as well as upper facets as *individual clock* facets. Facets derived from lower bounds of the forms $x \geq n$ or $x - y \geq m$ are classified as lower *relative facets* w.r.t. x. The collection of lower relative facets of Z w.r.t. x is denoted $LF_x(Z)$. The collection of upper relative facets of Z w.r.t. x, $UF_x(Z)$, is derived similarly. Figure 10.5(left) illustrates a zone Z together with its six facets: e.g. $\{Z_1, Z_6\}$ constitutes the lower facets of Z, and $\{Z_1, Z_2\}$ constitutes the lower relative facets of Z w.r.t. y.

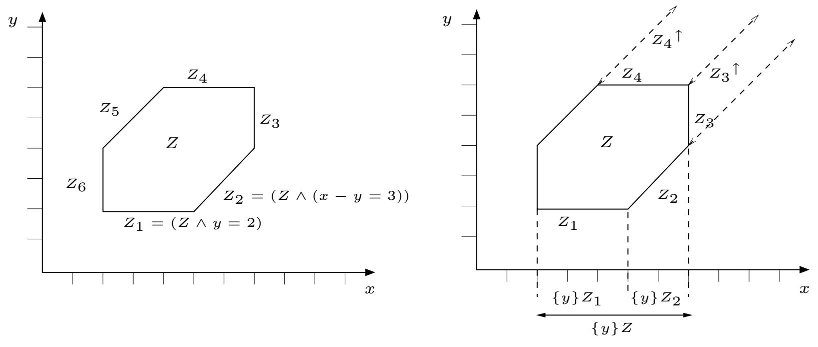


Figure 10.5: A Zone: Facets and Operations.

The importance of facets comes from the fact that they allow for decompositions of the delay- and reset-operations on zones as follows:

Lemma 10.5 *Let Z be a zone and y a clock. Then the following holds:*

i) $Z^{\uparrow} = \bigcup_{F \in LF(Z)} F^{\uparrow}$ *iii)* $\{y\}Z = \bigcup_{F \in LF_y(Z)} \{y\}F$

ii) $Z^{\uparrow} = Z \cup \bigcup_{F \in UF(Z)} F^{\uparrow}$ *iv)* $\{y\}Z = \bigcup_{F \in UF_y(Z)} \{y\}F$

Proof. As $F \subseteq Z$ whenever $F \in LF(Z)$ the $\supseteq$-direction of *i.* is trivial. For the $\subseteq$-direction of *i.* let $u \in Z^{\uparrow}$. Now let $d = min\{u(x) - l \mid (x \geq l) \in Z\}$. We claim that $u - d \in F$ for some $F \in LF(Z)$. Since $u \in Z^{\uparrow}$ there is some $u' \in Z$ and some e such that $u = u' + e$. Now $e \leq d$ since otherwise u' would violate one of the lower bounds of Z. Thus $u - d$ satisfies all upper bounds of Z on individual clocks (since u' does) and also all bounds on differences from Z are satisfied. Clearly, by the choice of d, all lower bounds of Z are satisfied by $u - d$. Thus $u - d \in Z$. However, it is also clear that $u - d \in F$ for some lower facet F.

As $F \subseteq Z$ whenever $F \in LF_y(Z)$ the $\supseteq$-direction of *iii.* is trivial. For the $\subseteq$-direction of *iii.* assume that $u \in \{y\}Z$. Now let $d = max\{u(x) - l \mid (y \geq x - l) \in Z\} \cup \{l \mid (y \geq l) \in Z\}$. We claim that $u[y \mapsto d] \in F$ for some $F \in LF_y(Z)$. Obviously, $\{y\}\big[u[y \mapsto d]\big] = u$. All constraints of Z not involving y are satisfied. Since $u \in \{y\}Z$ there is $u' \in Z$ such that $u = u'[y \mapsto 0]$. Now, $u'(y) \geq d$ since otherwise u' would violate some bound $(y \geq x - l) \in Z$. It follows that $u[y \mapsto d]$ satisfies all upper bounds for y (since u' does). Clearly, by the choice of d, all lower bounds of y are satisfied. □

Informally, *i)* and *ii)* express that any valuation reachable by delay from Z is reachable from one of the lower facets of Z, as well as reachable from one of the upper facets of Z or within Z, (see Fig. 10.5(right)). Any valuation in the projection of a zone will be in the projection of the lower (upper) facets of the zone relative to the relevant clock, which is expressed by *iii)* and *iv)*.

As a first step, the delay- and reset-operation may be extended in a straightforward manner to priced (relative) facets:

Definition 10.3 *Let $\mathcal{Z} = (F, c, r)$ be a priced zone, where F is a relative facet w.r.t. y in the sense that $y - x = m$ is a constraint of F. Then $\{y\}\mathcal{Z} = (F', c', r')$, where $F' = \{y\}F$, $c' = c$, and $r'(x) = r(y) + r(x)$ and $r'(z) = r(z)$ for $z \neq x$. In case $y = n$ is a constraint of F, $\{y\}\mathcal{Z} = (F', c, r)$ with $F' = \{y\}F$. In both cases $r'(y)$ may be set arbitrarily.*

Definition 10.4 *Let $\mathcal{Z} = (F, c, r)$ be a priced zone, where F is a lower or upper facet in the sense that $y = n$ is a constraint of F. Let $p \in \mathsf{N}$ be a cost-rate. Then $\mathcal{Z}^{\uparrow p} = (F', c', r')$, where $F' = F^{\uparrow}$, $c' = c$, and $r'(y) = p - \sum_{z \neq y} r(z)$ and $r'(z) = r(z)$ for $z \neq y$.*

Conjunction of constraints may be lifted from zones to priced zones simply by taking into account the possible change of the offset. Formally, let $\mathcal{Z} = (Z, c, r)$ be a priced zone and let $g \in \mathcal{B}(\mathbb{C})$. Then $\mathcal{Z} \wedge g$ is the priced zone $\mathcal{Z}' = (Z', c', r')$ with $Z' = Z \wedge g$, $r' = r$, and $c' = \mathsf{cost}(\Delta_{Z'}, \mathcal{Z})$. In particular, $\mathsf{cost}(u, \mathcal{Z}') = \mathsf{cost}(u, \mathcal{Z})$ for all $u \in \mathcal{Z}'$. For $\mathcal{Z} = (Z, c, r)$ and $n \in \mathsf{N}$ we denote by $\mathcal{Z} + n$ the priced zone $(Z, c + n, r)$. Thus, $\mathsf{cost}(u, \mathcal{Z} + n) = \mathsf{cost}(u, \mathcal{Z}) + n$ for all $u \in \mathcal{Z}$.

The constructs of Definitions 10.3 and 10.4 essentially provide the *Post*-operations for priced facets. More precisely, it is easy to show that:

$$Post_e(l, \mathcal{Z}_1) = (l', \{y\}(\mathcal{Z}_1 \wedge g) + P(e))$$

$$Post_\delta(l, \mathcal{Z}_2) = (l, (\mathcal{Z}_2 \wedge I(l))^{\uparrow P(l)} \wedge I(l))$$

if $e = (l, g, \{y\}, l')$, $\mathcal{Z}_1$ is a priced relative facet w.r.t. to y and $\mathcal{Z}_2$ is an individual clock facet. Now, the following extension of Lemma 10.5 to priced symbolic states provides the basis for the effective realization of *Post*-operations in general:

Theorem 10.3 *Let $\mathcal{A} = (L, l_0, E, I, P)$ be an LPTA. Let $e = (l, g, \{y\}, l') \in E$[6] with $P(e) = q$, $P(l) = p$, $I(l) = J$, $\sum_{x \in \mathbb{C}} r(x) = r_\Sigma$, and let $\mathcal{Z} = (Z, c, r)$ be a priced zone. Then:*

$$Post_e((l, \mathcal{Z})) = \begin{cases} \{\, (l', \{y\}Q + q) \mid Q \in LF_y(\mathcal{Z} \wedge g) \,\} & \text{if } r(y) \geq 0 \\ \{\, (l', \{y\}Q + q) \mid Q \in UF_y(\mathcal{Z} \wedge g) \,\} & \text{if } r(y) \leq 0 \end{cases}$$

$$Post_\delta((l, \mathcal{Z})) = \begin{cases} \{(l, \mathcal{Z})\} \,\cup\, \{\, (l, Q^{\uparrow p} \wedge J) \mid Q \in UF(\mathcal{Z} \wedge J)\} & \text{if } p \geq r_\Sigma \\ \{\, (l, Q^{\uparrow p} \wedge J) \mid Q \in LF(\mathcal{Z} \wedge J)\} & \text{if } p \leq r_\Sigma \end{cases}$$

In the definition of $Post_e$ the successor set is described as a union of either lower or upper relative facets w.r.t. to the clock y being reset, depending on the rate of y (as this will determine whether the minimum is obtained at the lower of

[6]For the case with a general reset-set r, the notion of relative facets may be generalized to sets of clocks.

upper boundary). For similar reason, in the definition of $Post_\delta$, the successor-set is expressed as a union over either lower or upper (individual clock) facets depending on the rate of the location compared to the sum of clock cost-rates.

To complete the instantiation of the framework of Section 10.2, it remains to indicate how to compute $minCost$ and $\sqsubseteq$ on priced symbolic states. Let $\mathcal{Z} = (Z, c, r)$ and $\mathcal{Z}' = (Z', c', r')$ be priced zones and let $(l, \mathcal{Z})$ and $(l', \mathcal{Z}')$ be corresponding priced symbolic states. Then $minCost(l, \mathcal{Z})$ is obtained by minimizing the linear expression $c + \sum_{x \in \mathbb{C}}(r(x) \cdot (x - \Delta_Z(x))$ under the (linear) constraints expressed by Z. Thus, computing $minCost$ reduces to solving a simple Linear Programming problem. Now let $\mathcal{Z}' \backslash \mathcal{Z}$ be the priced zone (Z^*, c^*, r^*) with $Z^* = Z$, $c^* = c' - \mathsf{cost}(\Delta_{Z'}, \mathcal{Z})$ and $r^*(x) = r'(x) - r(x)$ for all $x \in \mathbb{C}$. It is easy to see that $\mathsf{cost}(u, \mathcal{Z}' \backslash \mathcal{Z}) = \mathsf{cost}(u, \mathcal{Z}') - \mathsf{cost}(u, \mathcal{Z})$ for all $u \in \mathcal{Z}'$, and hence that

$$(l, \mathcal{Z}) \sqsubseteq (l', \mathcal{Z}') \iff (l = l') \land (Z' \subseteq Z) \land (minCost(\mathcal{Z}' \backslash \mathcal{Z}) \geq 0)$$

Thus, deciding $\sqsubseteq$ also reduces to a Linear Programming problem.

In exploring LPTAs using the algorithm of Fig. 10.1, we will only need to consider priced zones $\mathcal{Z}$ with non-negative cost assignments in the sense that $\mathsf{cost}(u, \mathcal{Z}) \geq 0$ for all $u \in \mathcal{Z}$. In addition, any LPTA $\mathcal{A}$ may be transformed into an equivalent "bounded" LPTA $\mathcal{A}^*$ in the sense that there is a bound on the possible (reachable) values of clocks of $\mathcal{A}^*$. Now, application of Higman's Lemma [76] ensures that $\sqsubseteq$ is a well-quasi ordering on priced symbolic states for bounded LPTA. We refer to [28] for more detailed arguments.

10.6 Implementation & Experiments

In this section we give further details on a prototype implementation within the tool Uppaal [109] of priced zones, formally defined in the previous sections. Also, we report on a number of experiments on the aircraft landing problem, an extended bridge problem and other examples conducted with the prototype tool. For brevity, we omit a detailed description of the algorithms.

The prototype implements the $Post_e$ (reset), $Post_\delta$ (delay), $minCost$, and $\sqsubseteq$ operations. The algorithms are extensions of the DBM algorithms outlined in [133]. As such, it is possible to realize reset of a clock x in a priced zone $\mathcal{Z}$ in $O(n^2)$ time, n being the number of clocks, provided that either the rate of x is zero or $\mathcal{Z}$ is a relative facet w.r.t. x. If neither is the case, $\mathcal{Z}$ will be split resulting in an $O(m \cdot n^2)$ algorithm, where m is the number of resulting priced zones (bounded by n). To keep this number to a minimum, we make heavy use of the (alternative) canonical representation of zones in terms a *minimal* set of constraints given in [111]. Delay can be implemented in $O(n)$ time given that the priced zone is an individual clock facet. This can be assured by adding an extra clock and resetting it on each transition. The $\sqsubseteq$ operation can be reduced to $minCost$ in $O(n^2)$ time, which in turn can be reduced to an LP problem. For dealing with LP problems, our prototype currently uses a free

Table 10.1: Results for seven instances of the aircraft landing problem. No result for instance 4 with 3 runways; cputime exceeded 8 hours.

runways	problem instance	1	2	3	4	5	6	7
	number of planes	10	15	20	20	20	30	44
	number of types	2	2	2	2	2	4	2
1	optimal value	700	1480	820	2520	3100	24442	1550
	explored states	481	2149	920	5693	15069	122	662
	cputime (secs)	4.19	25.30	11.05	87.67	220.22	0.60	4.27
2	optimal value	90	210	60	640	650	554	0
	explored states	1218	1797	669	28821	47993	9035	92
	cputime (secs)	17.87	39.92	11.02	755.85	1085.08	123.72	1.06
3	optimal value	0	0	0	130	170	0	
	explored states	24	46	84	207715	189602	62	N/A
	cputime (secs)	0.36	0.70	1.71	14786.19	12461.47	0.68	
4	optimal value				0	0		
	explored states	N/A	N/A	N/A	65	64	N/A	N/A
	cputime (secs)				1.97	1.53		

available implementation of the simplex algorithm.[7] Also, here the use of the (alternative) canonical form in [111] proves useful in reducing the size of the LP problems needed to be solved.

Many of the techniques described in [26] are directly applicable to the algorithm in Fig. 10.1, i.e., pruning the state space according to the COST variable, computing a lower bound on the remaining cost, exploring states with minimum cost first (known as the *minimum cost order*), and using heuristics to quickly guide the search to a goal state. These techniques have been used extensively in modelling and verifying the following examples.

Example 10.2 (Aircraft Landing Problem (continued)) *Recall the aircraft landing problem partially described in Example 10.1 of this paper. An LPTA model of the costs associated with landing a single aircraft is shown in Fig. 10.2(b). When landing several planes the schedule has to take into account the separation times between planes to avoid that the turbulence of one plane affecting another. The separation times depend on the types of the planes that are involved. Large aircrafts for example generate more turbulence than small ones, and successive planes should consequently keep a bigger distance. To model the separation times between two types of planes we introduce an LPTA of the kind shown in Fig. 10.2(c). The automaton uses two clocks `c1` and `c2` to measure and constrain the time between two consecutive landings. Clock `cX` (where `X` is 1 or 2) is always reset when a landing of a plane of type `X` takes place. For example, the guard `c1>=wait11` of the upper transition specifies that the time between two landings of planes of type 1 must be at least `wait11` (a constant integer value).*

Table 10.1 presents the results of an experiment were the prototype was applied to seven instances of the aircraft landing problem taken from [25][8]. For each

[7] lp_solve 3.1a by Michael Berkelaar, `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

[8] These and other benchmarks are available at `ftp://mscmga.ms.ic.ac.uk/pub/`.

instance, which varies in the number of planes and plane types, we compute the cost of the optimal schedule. In cases the cost is non-zero we increase the number of runways until a schedule of cost 0 is found[9]*. In all instances, the state space is explored in minimal cost-order, i.e. we select from the waiting list the priced zone $(l, \mathcal{Z})$ with lowest $minCost(l, \mathcal{Z})$. Equal values are distinguished by selecting first the zone which results from the largest number of transitions, and secondly by selecting the zone which involves the plane with the shortest target time. All numbers reported in Table 10.1 are measured on a 500MHz Pentium III running Linux.*

As can be seen from the table, our prototype implementation is able to deal with all the tested instances. In Beasley et.al. [25] all problem instances are solved using a linear programming based tree search algorithm. In 7 of the 15 instances with optimal solution greater than zero we are faster than the method in [25]. However, it should be noted that our solution-times are quite incomparable to those in [25]. For some instances our approach is up to 25 times slower that the one in [25], while for others we are up to 50 times faster. □

Example 10.3 (Extended Bridge Problem) *The original version of this problem was proposed by Ruys and Brinksma in [134]. In the following we present a variation of the problem in which non-uniform cost functions are needed. A version of the problem with uniform costs is studied in [26]. The problem is the following:*

Four persons want to cross a bridge in the dark. The bridge is damaged and can only carry two persons at the same time. To cross the bridge safely in the darkness, a torch must be carried along. The group has only one torch to share. Due to different physical abilities, the four cross the bridge at different speeds. The time they need per person is (one-way) 5, 10, 20 and 25 minutes, respectively. The original problem is to find a schedule such that all four cross the bridge within a given time. To extend the problem we introduce a cost associated with staying on the initial side of the bridge. The problem is now to find a way for the four persons to cross the bridge which results in the lowest possible cost.

In Table 10.2 we show the minimum costs for five instances of the extended bridge problem. For each instance we give the four costs assigned to the persons for residing on the initial side, the schedule (in which step 2 and 4 always represent a single person crossing the bridge back to the initial side), the minimum cost, the time of the generated schedule with minimum cost, and the number of explored states. All results have been obtained by searching the state space in minimal-cost order.

The first result in Table 10.2 were measured with a model for finding a time-optimal solution to the problem. The results on the first line shows that the time-optimal schedule requires 60 minutes and that 1762 (symbolic) states are explored to find the solution with the cost-extended version of Uppaal. *The*

[9]This is always possible as the cost of landing on target time is 0 and the number of runways can be increased until all planes arrive at target time.

Table 10.2: Schedules and minimum costs for cost extended versions of the Bridge Problem.

Cost-rates				**Schedule**					**Cost**	**Time**	**States**
A_5	B_{10}	C_{20}	D_{25}						-		
Min Time				BA	A	CD	B	BA	-	60	1762
Min Time				BA	A	CD	B	BA	-	60	(1538)
1	1	1	1	BA	A	CA	A	AD	55	65	252
9	2	3	10	AD	A	BA	A	CA	195	65	149
1	2	3	4	BA	A	CD	B	BA	140	60	232
1	2	3	10	BD	B	BC	B	BA	170	65	263

second line shows (within parenthesis) the number of symbolic states need to solve the same problem with an special-purpose time-optimizing version of Uppaal *[26]. Thus, in comparison the general cost version increments the number of explored states with less than 15% in this example. The four last lines of the table show the number of explored states when optimizing costs instead of time. From these results we observe that considering general costs seems to significantly reduce the number of symbolic states explored.* □

The cost-extended version of Uppaal has been (and is currently being) applied to other examples, including an optimal broadcast problem and a testing problem. Due to lack of space we omit the details but mention them briefly here.

In the optimal broadcast problem, Uppaal is applied to find schemes for broadcasting messages in a network consisting of several routers connected with two communication channels. The cost and time of using the two channels differ and the problem is to find a time or cost-optimal schedule for broadcasting a set of messages to all subscribed routers. So far, we have been able to solve this problem (with varying communication costs) for up to six routers.

In the testing example, the problem is to find a minimal set of test sequences to fully cover different aspects of the sender component of the audio protocol in [32]. This can be done by annotating the model with edges and testing variables which are set when an aspect of the specification has been covered. The cost extended version of Uppaal can then be applied to find the cheapest possible path trough the specification which sets all the testing variables. We have been able to apply this technique in Uppaal to generate optimal testing sequences for covering e.g. all location, all synchronization actions, or all edges of the protocol specification.

10.7 Conclusion

In this paper we have considered the minimum-cost reachability problem for LPTAs. The notions of priced zones, and facets of a zone are central contributions of the paper underlying our extension of the tool Uppaal. Our initial experimental investigations based on a number of examples are quite encouraging.

Compared with the existing special-purpose, time-optimizing version of UPPAAL [26], the presented general cost-minimizing implementation does only marginally down-grade performance. In particular, the theoretical possibility of uncontrolled splitting of zones does not occur in practice. In addition, the consideration of non-uniform cost seems to significantly reduce the number of symbolic states explored.

The single, most important question, which calls for future research, is how to exploit the simple structure of the LP-problems considered. We may benefit significantly from replacing the currently used LP package with some package more tailored towards small-size problems.

Chapter 11

Parametric Real-Time Model Checking

The paper *Linear Parametric Model Checking of Timed Automata* presented in this chapter has been published in part as a technical report [87] and a conference paper [86].

[86] T. Hune, J. Romijn, M. Stoelinga and F. Vaandrager. Linear Parametric Model Checking of Timed Automata. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, TACAS2001*, pages 189–203, 2001.

[87] T. Hune, J. Romijn, M. Stoelingaand F. Vaandrager. Linear Parametric Model Checking of Timed Automata. Technical Report RS-01-05, BRICS, January 2001.

The technical report extends the conference paper with proofs of the representation of the parametric state-space and more explanation of the experiments. Except for minor typographical changes the content of this chapter is equal to the technical report [87].

Linear Parametric Model Checking of Timed Automata[‡]

Thomas Hune[*] Judi Romijn[†] Mariëlle Stoelinga[†]
Frits Vaandrager[†]

Abstract

We present an extension of the model checker Uppaal capable of synthesize linear parameter constraints for the correctness of parametric timed automata. The symbolic representation of the (parametric) state-space is shown to be correct. A second contribution of this paper is the identification of a subclass of parametric timed automata (L/U automata), for which the emptiness problem is decidable, contrary to the full class where it is know to be undecidable. Also we present a number of lemmas enabling the verification effort to be reduced for L/U automata in some cases. We illustrate our approach by deriving linear parameter constraints for a number of well-known case studies from the literature (exhibiting a flaw in a published paper).

11.1 Introduction

During the last decade, there has been enormous progress in the area of timed model checking. Tools such as Uppaal [109], Kronos [41], and PMC [116] are now routinely used for industrial case studies. A disadvantage of the traditional approaches is, however, that they can only be used to verify concrete timing properties: one has to provide the values of all timing parameters that occur in the system. For practical purposes, one is often interested in deriving the (symbolic) constraints on the parameters that ensure correctness. The process of manually finding and proving such results is very time consuming and error prone (we have discovered minor errors one of the examples we have been looking at). Therefore tool support for deriving the constraints *automatically* is very important.

[*]**B**asic **R**esearch **I**n **C**omputer **S**cience, BRICS, (www.brics.dk), funded by the Danish National Research Foundation, University of Århus, Denmark, Email: `baris@brics.dk`

[†]Computing Science Institute, University of Nijmegen, Email: `[judi,marielle,fvaan]@cs.kun.nl`

[‡]Research supported by Esprit Project 26270, Verification of Hybrid Systems (VHS), and by PROGRESS Project TES4199, Verification of Hard and Softly Timed Systems (HaaST).

In this paper, we study a parameterized extension of timed automata, as well as a corresponding extension of the forward reachability algorithm for timed automata. We show the theoretical correctness of our approach, and its feasibility by application to non-trivial case studies. For this purpose, we have implemented a prototype extension of Uppaal, an efficient real-time model checking tool [109]. The algorithm we propose and have implemented is a semi-decision algorithm which will not terminate in all cases. In [15] the problem of synthesizing values for parameters such that a property is satisfied, was shown to be undecidable, so this is the best we can hope for.

A second contribution of this paper is the identification of a subclass of parameterized timed automata, called *lower bound/upper bound (L/U) automata*, which appears to be sufficiently expressive from a practical perspective, while it also has nice theoretical properties. Most importantly, we show that the emptiness problem for parametric timed automata, shown to be undecidable in [15], is decidable for L/U automata. We also establish a number of lemmas which allow one to reduce the number of parameters when tackling specific verification questions for L/U automata. The application of these lemmas has already reduced the verification effort drastically in some of our experiments.

Related work Our attempt at automatic verification of parameterized real-time models is not the only one. Henzinger et al. aim at solving a more general problem with HyTech [74], a tool for model checking hybrid automata, exploring the state-space either by partition refinement, or forward reachability. The tool has been applied successfully to relatively small examples such as a railway gate controller. Experience so far has shown that HyTech cannot cope with larger examples, such as the ones considered in this paper.

Toetenel et al. have made an extension of the PMC [116] real-time model checking tool called LPMC [24]. LPMC is restricted to linear parameter constraints as is our approach, and uses the partition refinement method, like HyTech. Other differences with our approach are that LPMC also allows for the comparison of non-clock variables to parameter constraints, and for more general specification properties (full TCTL with fairness assumptions). Since LPMC is a quite recent tool, not many applications have been presented yet. However, a model of the IEEE 1394 root contention protocol inspired by [136] has been successfully analyzed in [24].

A more general attempt than LPMC and our Uppaal extension has been made by Annichini et al. [20]. They have constructed and implemented a method which allows non-linear parameter constraints, and uses heavier, third-party, machinery to solve the arising non-linear constraint comparisons. Independently, we have used the same data-structure (a direct extension of DBMs [60]) for the symbolic representation of the state space, as in [20]. For speeding up the exploration, a method for guessing the effect of control loops in the model is presented. It appears that this helps termination of the method, but it is unclear under what circumstances this technique can or cannot be used. The feasibility of this approach has been shown on a few rather small case studies.

One of these is Fischer's protocol with two processes, for which the state space is constructed in about 3 minutes cpu time.

The remainder of this paper is organized as follows. Section 11.2 introduces the notion of parametric timed automata. Section 11.3 gives the symbolic semantics, which is the basis for our model checking algorithm, presented in Section 11.3.5. Section 11.4 is an intermezzo that states some helpful lemmas and decidability results on an interesting subclass. Finally, Section 11.5 reports on experiments with our tool.

11.2 Parametric Timed Automata

11.2.1 Parameters and Constraints

Throughout this paper, we assume a fixed set of *parameters* $P = \{p_1, \ldots, p_n\}$.

Definition 11.1 (Constraints) *A* linear expression *e is either an expression of the form $t_1 p_1 + \cdots + t_n p_n + t_0$, where $t_0, \ldots, t_n \in \mathsf{Z}$, or ∞. We write E to denote the set of all linear expressions. A* constraint *is an inequality of the form $e \sim e'$, with e, e' linear expressions and $\sim \in \{<, \leq, >, \geq\}$. The* negation *of constraint c, notation $\neg c$, is obtained by replacing relation signs $<$, $\leq$, $>$, $\geq$ by $\geq$, $>$, $\leq$, $<$, respectively. A* (parameter) valuation *is a function $v : P \to \mathsf{R}^{\geq 0}$ assigning a nonnegative real value to each parameter. There is a one-to-one correspondence between valuations and points in $(\mathsf{R}^{\geq 0})^n$. In fact we often identify a valuation v with the point $(v(p_1), \ldots, v(p_n)) \in (\mathsf{R}^{\geq 0})^n$.*

If e is a linear expression and v is a valuation, then $e[v]$ denotes the expression obtained by replacing each parameter p in e with $v(p)$. Likewise, we define $c[v]$ for c a constraint. Valuation v satisfies *constraint c, notation $v \models c$, if $c[v]$ evaluates to true. The* semantics *of a constraint c, notation $[\![c]\!]$, is the set of valuations (points in $(\mathsf{R}^{\geq 0})^n$) that satisfy c. A finite set of constraints C is called a* constraint set*. A valuation* satisfies *a constraint set if it satisfies each constraint in the set. The* semantics *of a constraint set C is given by $[\![C]\!] := \bigcap_{c \in C} [\![c]\!]$. We write $\top$ to denote any constraint set with $[\![\top]\!] = (\mathsf{R}^{\geq 0})^n$, for instance the empty set. We use $\perp$ to denote any constraint set with $[\![\perp]\!] = \emptyset$, for instance the constraint set $\{c, \neg c\}$, for some arbitrary c.*

Constraint c covers *constraint set C, notation $C \models c$, iff $[\![C]\!] \subseteq [\![c]\!]$. Constraint set C* is split by *constraint c iff neither $C \models c$ nor $C \models \neg c$.*

During the analysis questions arise of the kind: given a constraint set C and a constraint c, does c hold, i.e., does constraint c cover C? There are three possible answers to this, *yes*, *no*, and *split*. A split occurs when c holds for some valuations in the semantics of C and $\neg c$ holds for some other valuations. We will not discuss methods for answering such questions: in our implementation we use an oracle to compute the following function.

Definition 11.2 (Oracle)

$$\mathcal{O}(c, C) = \begin{cases} \text{yes} & \textit{if } C \models c \\ \text{no} & \textit{if } C \models \neg c \\ \text{split} & \textit{otherwise} \end{cases}$$

Observe that using the oracle, we can easily decide semantic inclusion between constraint sets: $[\![C]\!] \subseteq [\![C']\!]$ iff $\forall c' \in C' : \mathcal{O}(c', C) = \text{yes}$. The oracle that we use is a Linear Programming (LP) solver that was kindly provided to us by the authors of [24], who built it for their LPMC model checking tool. This LP solver is geared to perform well on small, simple sets of constraints rather than large, complicated ones.

11.2.2 Parametric Timed Automata

Throughout this paper, we assume a fixed set of clocks $X = \{x_0, \ldots, x_m\}$ and a fixed set of actions $A = \{a_1, \ldots, a_k\}$. The special clock x_0, which is called the *zero clock*, always has the value 0 (and hence does not increase with time).

A *simple guard* is an expression f of the form $x_i - x_j \prec e$, where x_i, x_j are clocks, $\prec \in \{<, \leq\}$, and e is a linear expression. We say that f is *proper* if $i \neq j$. We define a *guard* to be a (finite) conjunction of simple guards. We let g range over guards and write G to denote the set of guards. A *clock valuation* is a function $w : X \to \mathsf{R}^{\geq 0}$ assigning a nonnegative real value to each clock, such that $w(x_0) = 0$. We will identify a clock valuation w with the point $(w(x_0), \ldots, w(x_m)) \in (\mathsf{R}^{\geq 0})^{m+1}$. Let g be a guard, v a parameter valuation, and w a clock valuation. Then $g[v, w]$ denotes the expression obtained by replacing each parameter p with $v(p)$, and each clock x with $w(x)$. A pair (v, w) of a parameter valuation and a clock valuation *satisfies* a guard g, notation $(v, w) \models g$, if $g[v, w]$ evaluates to true. The *semantics* of a guard g, notation $[\![g]\!]$, is the set of pairs (v, w) such that $(v, w) \models g$.

A *reset* is an expression of the form, $x_i := b$ where $i \neq 0$ and $b \in \mathsf{N}$. A *reset set* is a set of resets containing at most one reset for each clock. The set of reset sets is denoted by R.

We now define an extension of timed automata [13, 147] called parametric timed automata. Similar models have been presented in [15, 20, 24].

Definition 11.3 (PTA) *A* parametric timed automaton (PTA) *over set of clocks X, set of actions A, and set of parameters P, is a quadruple $\mathcal{A} = (Q, q_0, \to, I)$, where Q is a finite set of* locations*, $q_0 \in Q$ is the* initial location*, $\to \subseteq Q \times A \times G \times R \times Q$ is a finite* transition relation*, and function $I : Q \to G$ assigns an* invariant *to each location. We abbreviate a $(q, a, g, r, q') \in \to$ consisting of a source location, an action, a guard, a reset set, and a target location as $q \xrightarrow{a,g,r} q'$. For a simple guard $x_i - x_j \prec e$ to be used in an invariant it must be the case that $x_j = x_0$, that is, the simple guard represents an upper bound on a clock.*

Example 11.1 *A parametric timed automaton with clocks x, y and parameters p, q can be seen in Fig. 11.1. The initial state is $S0$ which has invariant $x \leq p$, and the transition from the initial location to $S1$ has guard $y \geq q$ and reset set $x := 0$. There are no actions on the transitions. Initially the transition from $S0$ to $S1$ is only enabled if $p \leq q$, otherwise the system will be deadlocked.*

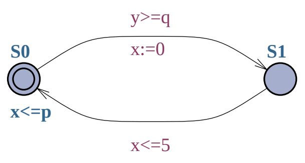


Figure 11.1: A parametric timed automaton

To define the semantics of PTAs, we require two auxiliary operations on clock valuations. For clock valuation w and nonnegative real number d, $w + d$ is the clock valuation that adds to each clock (except x_0) a delay d. For clock valuation w and reset set r, $w[r]$ is the clock valuation that resets clocks according to r.

$$(w+d)(x) = \begin{cases} 0 & \text{if } x = x_0 \\ w(x)+d & \text{otherwise} \end{cases} \qquad (w[r])(x) = \begin{cases} b & \text{if } x := b \in r \\ w(x) & \text{otherwise.} \end{cases}$$

Definition 11.4 (LTS) *A* labeled transition system (LTS) *over a set of symbols Σ is a triple $\mathcal{L} = (S, S_0, \rightarrow)$, with S a set of* states*, $S_0 \subseteq S$ a set of* initial states*, and $\rightarrow \subseteq S \times \Sigma \times S$ a* transition relation*. We write $s \xrightarrow{a} s'$ for $(s, a, s') \in \rightarrow$. A* run *of $\mathcal{L}$ is a finite alternating sequence $s_0 a_1 s_1 a_2 \cdots s_n$ of states $s_i \in S$ and symbols $a_i \in \Sigma$ such that $s_0 \in S_0$ and, for all $i < n$, $s_i \xrightarrow{a_{i+1}} s_{i+1}$. A state is* reachable *if it is the last state of some run.*

Definition 11.5 (Concrete semantics) *Let $\mathcal{A} = (Q, q_0, \rightarrow, I)$ be a PTA and v be a parameter valuation. The* concrete semantics of $\mathcal{A}$ under v, *notation $[\![\mathcal{A}]\!]_v$, is the labeled transition system (LTS) $(S, S_0, \rightarrow)$ over $A \cup \mathsf{R}^{\geq 0}$ where*

$$\begin{aligned} S &= \{(q,w) \in Q \times (X \rightarrow \mathsf{R}^{\geq 0}) \mid w(x_0) = 0 \wedge (v,w) \models I(q)\}, \\ S_0 &= \{(q,w) \in S \mid q = q_0 \wedge w = \lambda x.0\}, \end{aligned}$$

and transition predicate $\rightarrow$ is specified by the following two rules, for all (q,w), $(q',w') \in S$, $d \geq 0$ and $a \in A$,

- $(q,w) \xrightarrow{d} (q',w')$ *if* $q = q'$ *and* $w' = w + d$.
- $(q,w) \xrightarrow{a} (q',w')$ *if* $\exists g, r : q \xrightarrow{a,g,r} q' \wedge (v,w) \models g \wedge w' = w[r]$.

Note that the LTS $[\![\mathcal{A}]\!]_v$ has at most one initial state (at most, since we require that all states satisfy the location invariants).

11.2.3 The Problem

In its current version, UPPAAL is able to check for reachability properties, in particular whether certain combinations of locations and constrains on clock variables are reachable from the initial configuration. Our parameterized extension of UPPAAL handles exactly the same properties. However, rather than just telling whether a property holds or not, our tool looks for constraints on the parameters which ensure that the property holds.

Definition 11.6 (Properties) *The sets of* system properties *and* state formulas *are defined by, respectively,*

$$\psi ::= \forall\Box\phi \mid \exists\Diamond\phi \qquad\qquad \phi ::= x - y \prec b \mid q \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi$$

where $x, y \in X$, $b \in \mathsf{N}$ and $q \in Q$. Let $\mathcal{A}$ be a PTA, v a parameter valuation, s a state of $[\![\mathcal{A}]\!]_v$, and ϕ a state formula. We write $s \models \phi$ if ϕ holds in state s, we write $[\![\mathcal{A}]\!]_v \models \forall\Box\phi$ if ϕ holds in all reachable states of $[\![\mathcal{A}]\!]_v$, and we write $[\![\mathcal{A}]\!]_v \models \exists\Diamond\phi$ if ϕ holds for some reachable state of $[\![\mathcal{A}]\!]_v$.

The problem that we address in this paper can now be stated as follows: *Given a parametric timed automaton $\mathcal{A}$ and a system property ψ, compute the set of parameter valuations v for which $[\![\mathcal{A}]\!]_v \models \psi$.*

Timed automata [13, 147] arise as a special case of PTAs for which the set P of parameters is empty. If $\mathcal{A}$ is a PTA and v is a parameter valuation, then the structure $\mathcal{A}[v]$ that is obtained by replacing all linear expressions e that occur in $\mathcal{A}$ by $e[v]$ is a timed automaton.[1] It is easy to see that in general $[\![\mathcal{A}]\!]_v = [\![\mathcal{A}[v]]\!]$. Since the reachability problem for timed automata is decidable [13], this implies that, for any $\mathcal{A}$, integer valued v and ψ, $[\![\mathcal{A}]\!]_v \models \psi$ is decidable.

11.2.4 Example: Fischer's Mutual Exclusion Protocol

Figure 11.2 shows a PTA model of Fischer's mutual exclusion protocol [102]. The purpose of this protocol is to guarantee mutually exclusive access to a critical section among competing processes P_1, $P_2, \ldots$ P_n. In this protocol, a shared variable lock is used for communication between the processes, with each process P_i running the following algorithm.

```
lock := 0;
REPEAT
    while lock ≠ 0 do skip;
    lock := i;
    delay
UNTIL lock = i;
critical section;
lock := 0
```

[1] Strictly speaking, $\mathcal{A}[v]$ is only a timed automaton if v assigns an integer to each parameter.

The correctness of this algorithm crucially depends on the timing of the operations. The key idea for the correctness is that any process P_i that sets $\mathsf{lock} := i$ is made to wait long enough before checking $\mathsf{lock} = i$ to ensure that any other process P_j that tested $\mathsf{lock} = 0$, before P_i set lock to its index, has already set lock to its index j, when P_i finally checks $\mathsf{lock} = i$.

Assume that read/write access to the global variable (in the operations $\mathsf{lock} = i$ and $\mathsf{lock} := 0$) takes between min_rw and max_rw time units and assume that the delay operation (including the timed needed for the the assignment $\mathsf{lock} := i$) takes between min_delay and max_delay time units. If we assume the basic constraints $0 \leq min\_rw < max\_rw \wedge 0 \leq min\_delay < max\_delay$, then mutual exclusion is guaranteed if and only if $max\_rw \leq min\_delay$.

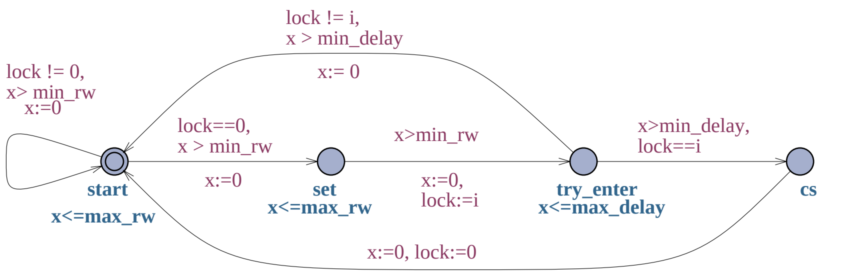


Figure 11.2: A PTA model of Fischer's Mutual Exclusion Protocol

Now consider the PTA in Fig. 11.2. (Several different models of this protocol exist [3, 15, 117] and one in the standard distribution of UPPAAL, our model is closest to the one in [117].) It consists of four locations *start* (which is initial), *set*, *try_enter* and *cs*; four parameters, min_rw, max_rw, min_delay and max_delay; one clock x and a shared variable $lock$. By convention, x and $lock$ are initially 0. Note that the process can remain in the locations *start* and *set* for at least min_rw and at most max_rw time units. Similarly, the process can remain in $lock$ for at least min_delay and at most max_delay time units.

The shared variable, which is not a part of the definition of PTAs, is syntactic sugar which allows for an efficient encoding of the protocol as a PTA and the notion of parallel composition for PTAs is standard. We refer the reader to [109] for their definitions.

11.3 Symbolic State Exploration

Our aim is to use basically the same algorithm for parametric timed model checking as for timed model checking. We represent sets of states symbolically in a similar way and support the same operations used for timed model checking. In the non-parametrized case, sets of states can be efficiently represented using matrices [60]. Similarly, in this paper we represent sets of states symbolically as *(constrained) parametric difference-bound matrices.*

Basically the same approach was followed in [20], although not worked out in detail. New in our presentation is the systematic use of structural operational semantics to deal with the nondeterministic computation that takes place in the parametrized case.

11.3.1 Parametric Difference-Bound Matrices

In the non-parametrized case, a difference-bound matrix is a $(m+1) \times (m+1)$ matrix whose entries are elements from $(\mathsf{Z} \cup \{\infty\}) \times \{0,1\}$. An entry $(c,1)$ for D_{ij} denotes a nonstrict bound $x_i - x_j \leq c$, whereas an entry $(c,0)$ denotes a strict bound $x_i - x_j < c$. Here, instead of using integers in the entries, we will use linear expressions over the parameters. Also, we find it convenient to view the matrix slightly more abstractly as a set of guards.

Definition 11.7 (PDBM) *A* parametric difference-bound matrix (PDBM) *is a set D which contains, for all $0 \leq i,j \leq m$, a simple guard D_{ij} of the form $x_i - x_j \prec_{ij} e_{ij}$. We require that, for all i, D_{ii} is of the form $x_i - x_i \leq 0$. Given a parameter valuation v, the* semantics *of D is defined by $[\![D]\!]_v = [\![\bigwedge_{i,j} D_{ij}]\!]_v$. We say that D is* satisfiable *for v if $[\![D]\!]_v$ is nonempty. If f is a proper guard of the form $x_i - x_j \prec e$ then we write $D[f]$ for the PDBM obtained from D by replacing D_{ij} by f. If i,j are indices then we write D^{ij} for the pair $(e_{ij}, \prec_{ij})$; we call D^{ij} a* bound *of D. Clearly, a PDBM is fully determined by its bounds.*

Definition 11.8 (Constrained PDBM) *Let C be a constraint set and D a PDBM, then the pair (C,D) is a* constrained PDBM. *The* semantics *of a constrained PDBM is defined by $[\![C,D]\!] = \{(v,w) \mid v \in [\![C]\!] \wedge w \in [\![D]\!]_v\}$.*

PDBMs with the tightest possible bounds are called *canonical*. To formalize this notion, we define an addition operation on linear expressions by

$$\begin{aligned}(t_1p_1 + \cdots + t_np_n + t_0) \quad &+ \quad (t'_1p_1 + \cdots + t'_np_n + t'_0)\\ &= \quad (t_1 + t'_1)p_1 + \cdots + (t_n + t'_n)p_n + (t_0 + t'_0).\end{aligned}$$

Also, we view Boolean connectives as operations on relation symbols $\leq$ and $<$ by identifying $\leq$ with 1 and $<$ with 0. Thus we have, for instance, $(\leq \wedge \leq) = \leq$, $(\leq \wedge <) = <$, $\neg \leq = <$, and $(\leq \implies <) = <$. Our definition of a canonical form of a constrained PDBM is essentially equivalent to the one for standard DBMs.

Definition 11.9 (Canonical Form) *A constrained PDBM (C,D) is* in canonical form *iff for all i,j,k, $C \models e_{ij}\ (\prec_{ij} \implies \prec_{ik} \wedge \prec_{kj})\ e_{ik} + e_{kj}$.*

The proof of the following technical lemma is immediate from the definitions.

Lemma 11.1

1. *If $v \models e \prec e'$ and $v \models e' \prec e''$ then $v \models e\ (\prec \wedge \prec')\ e''$.*

2. *If $(v, w) \models x - y \prec e$ and $v \models e \prec' e'$ then $(v, w) \models x - y\ (\prec \wedge \prec')\ e'$.*

3. *If $v \models e\ (\prec \wedge \prec')\ e'$ then $v \models e \prec e'$.*

4. *If $(v, w) \models x - y\ (\prec \wedge \prec')\ e$ then $(v, w) \models x - y \prec e$.*

5. *If $(v, w) \models x - y \ \prec\ e$ and $(v, w) \models y - z \ \prec'\ e'$ then $(v, w) \models x - z\ (\prec \wedge \prec')\ e + e'$.*

6. *$v \models \neg(e \prec e')$ iff $v \models e'\ (\neg \prec)\ e$.*

The next lemma, which basically carries over from the unparameterized case, states that canonicity of a constrained PDBM guarantees satisfiability. We recall the proof, since we will need the same argument later on in this section.

Lemma 11.2 *Suppose (C, D) is a constrained PDBM in canonical form and $v \in [\![C]\!]$. Then D is satisfiable for v.*

Proof. Inductively we construct a valuation $(t_0, \ldots, t_i)$ for variables $(x_0, \ldots, x_i)$ such that all constraints D_{jk} for $0 \leq j, k \leq i$ are met.

To begin with, we set $t_0 = 0$. Then, trivially, $(v, x_0 \mapsto t_0) \models D_{00}$.

For the induction step, suppose that for some $i < n$ we have a valuation $(t_0, \ldots, t_i)$ for variables $(x_0, \ldots, x_i)$ such that all constraints D_{jk} for $0 \leq j, k \leq i$ are met. In order to extend this valuation to x_{i+1}, we have to find a value t_{i+1} such that the following simple guards hold for valuation $(v, x_0 \mapsto t_0, \ldots, x_{i+1} \mapsto t_{i+1})$:

$$D_{i+1,0} \quad \cdots \quad D_{i+1,i} \quad D_{0,i+1} \quad \cdots \quad D_{i,i+1} \quad D_{i+1,i+1} \tag{11.1}$$

Here the first $i + 1$ simple guards give upper bounds for t_{i+1}, the second $i + 1$ simple guards give lower bounds for t_{i+1}, and the last simple guard is trivially met by any choice for t_{i+1}. We claim that each of the upper bounds is larger than each of the lower bounds. In particular, the minimum of the upper bounds is larger than the maximum of the lower bounds. This gives us a nonempty interval of possible values for t_{i+1} to choose from. Formally, we claim that, for all $0 \leq j, k < i + 1$, the following formula holds for valuation $(v, x_0 \mapsto t_0, \ldots, x_i \mapsto t_i)$:

$$x_j - e_{j,i+1} \quad \prec_{j,i+1} \wedge \prec_{i+1,k} \quad x_k + e_{i+1,k} \tag{11.2}$$

To see why (11.2) holds, observe that by induction hypothesis $(v, x_0 \mapsto t_0, \ldots, x_i \mapsto t_i) \models$

$$x_j - x_k \quad \prec_{jk} \quad e_{jk} \tag{11.3}$$

Furthermore, since (C, D) is canonical,

$$e_{jk} \quad (\prec_{jk} \Longrightarrow \prec_{j,i+1} \wedge \prec_{i+1,k}) \quad e_{j,i+1} + e_{i+1,k} \tag{11.4}$$

Combination of (11.3) and (11.4), using Lemma 11.1(1), gives $(v, x_0 \mapsto t_0, \dots, x_i \mapsto t_i) \models$

$$x_j - x_k \quad \prec_{j,i+1} \wedge \prec_{i+1,k} \quad e_{j,i+1} + e_{i+1,k}$$

which is equivalent to (11.2). This means that we can choose t_{i+1} in accordance with all the guards of (11.1), which completes the proof of the induction step and thereby of the lemma. □

Also the following lemma essentially carries over from the unparameterized case, see for instance [60]. As a direct consequence, semantic inclusion of constrained PDBMs is decidable for canonical PDBMs (using the oracle function).

Lemma 11.3 *Suppose $(C, D), (C', D')$ are constrained PDBMs and (C, D) is canonical, then*

$$[\![C, D]\!] \subseteq [\![C', D']\!] \;\Leftrightarrow\; ([\![C]\!] \subseteq [\![C']\!] \wedge \forall i, j : C \models e_{ij}(\prec_{ij} \Longrightarrow \prec'_{ij})e'_{ij}).$$

11.3.2 Operations on PDBMs

Our algorithm requires basically four operations to be implemented on constrained PDBMs: adding guards, canonicalization, resetting clocks and computing time successors.

Adding Guards

In the case of DBMs, adding a guard is a simple operation. It is implemented by taking the conjunction of a DBM and the guard (which is also viewed as a DBM). The conjunction operation just takes the pointwise minimum of the entries in both matrices. In the parametric case, adding a guard to a constrained PDBM may result in a set of constrained PDBMs. We define a relation $\Lleftarrow$ which relates a constrained PDBM and a guard to a collection of constrained PDBMs that satisfy this guard. For this we need an operation $\mathcal{C}$ that takes a PDBM and a simple guard, and produces a constraint stating that the bound imposed by the guard is larger than the corresponding bound in the PDBM, so let $D^{ij} = (e_{ij}, \prec_{ij})$ then

$$\mathcal{C}(D, x_i - x_j \prec e) \quad = \quad e_{ij}\ (\prec_{ij} \Longrightarrow \prec)\ e.$$

Relation $\Lleftarrow$ is defined as the smallest relation that satisfies the following rules:

$$(R1)\ \frac{\mathcal{O}(\mathcal{C}(D,f),C) = \text{yes}}{(C,D) \overset{f}{\Lleftarrow} (C,D)} \qquad\qquad (R2)\ \frac{\mathcal{O}(\mathcal{C}(D,f),C) = \text{no},\ f \text{ proper}}{(C,D) \overset{f}{\Lleftarrow} (C,D[f])}$$

$$(R3)\ \frac{\mathcal{O}(\mathcal{C}(D,f),C) = \text{split}}{(C,D) \overset{f}{\Mapsfrom} (C \cup \{\mathcal{C}(D,f)\}, D)} \qquad (R4)\ \frac{\mathcal{O}(\mathcal{C}(D,f),C) = \text{split},\ f \text{ proper}}{(C,D) \overset{f}{\Mapsfrom} (C \cup \{\neg\mathcal{C}(D,f)\}, D[f])}$$

$$(R5)\ \frac{(C,D) \overset{g}{\Mapsfrom} (C',D')\ ,\ (C'D') \overset{g'}{\Mapsfrom} (C'',D'')}{(C,D) \overset{g \wedge g'}{\Mapsfrom} (C'',D'')}$$

If the oracle replies "yes", then adding a simple guard will not change the constrained PDBM. If the answer is "no" then we tighten the bound in the PDBM according to the simple guard. With the answer "split" there are two possibilities and two pairs with updated constraint systems are returned. The side condition "f proper" in rules $R2$ and $R4$ ensures that the diagonal bounds in the PDBM always remain equal to $(0, \leq)$. If we update a bound in D then the semantics of the PDBM may become empty. The following lemma characterizes $\Mapsfrom$ semantically.

Lemma 11.4 $[\![C,D]\!] \cap [\![g]\!] \ = \ \bigcup\{[\![C',D']\!] \mid (C,D) \overset{g}{\Mapsfrom} (C',D')\}$.

Proof. "$\subseteq$". Assume $(v,w) \in [\![C,D]\!] \wedge (v,w) \models g$. By structural induction on g we prove that there exists a constrained PDBM (C',D') such that $(C,D) \overset{g}{\Mapsfrom} (C',D')$ and $(v,w) \in [\![C',D']\!]$.

For the induction basis, suppose g is of the form $x_i - x_j \prec e$. We consider four cases:

- $\mathcal{O}(\mathcal{C}(D,g),C) = \text{yes}$. Let $C' = C$ and $D' = D$. Then trivially $(v,w) \in [\![C',D']\!]$ and, by rule $R1$, $(C,D) \overset{g}{\Mapsfrom} (C',D')$.
- $\mathcal{O}(\mathcal{C}(D,g),C) = \text{no}$. By contradiction we prove that g is proper. Suppose g is not proper. Then, since $i = j$ and $v \models \neg e_{ij}(\prec_{ij} \implies \prec)e$, $v \models \neg(0 \prec e)$. By Lemma 11.1(6), $v \models e\neg \prec 0$. But $(v,w) \models g$ implies $v \models 0 \prec e$. Hence, by Lemma 11.1(1), $v \models 0 < 0$, a contradiction. Let $C' = C$ and $D' = D[g]$. Then, by rule $R2$, $(C,D) \overset{g}{\Mapsfrom} (C',D')$. Since $(v,w) \in [\![C,D]\!]$ and $(v,w) \models g$, it follows that $(v,w) \in [\![C',D']\!]$.
- $\mathcal{O}(\mathcal{C}(D,g),C) = \text{split}$ and $v \models \mathcal{C}(D,g)$. Let $C' = C \cup \{\mathcal{C}(D,g)\}$ and $D' = D$. Then, by rule $R3$, $(C,D) \overset{g}{\Mapsfrom} (C',D')$. Moreover, by the assumptions, $(v,w) \in [\![C',D']\!]$.
- $\mathcal{O}(\mathcal{C}(D,g),C) = \text{split}$ and $v \models \neg\mathcal{C}(D,g)$. By contradiction we prove that g is proper. Suppose g is not proper. Then, since $v \models \neg\mathcal{C}(D,g)$, $v \models \neg(0 \prec e)$. By Lemma 11.1(6), $v \models e\neg \prec 0$. But $(v,w) \models g$ implies $v \models 0 \prec e$. Hence, by Lemma 11.1(1), $v \models 0 < 0$, a contradiction. Let $C' = C \cup \{\neg\mathcal{C}(D,g)\}$ and $D' = D[g]$. Then, by rule $R4$, $(C,D) \overset{g}{\Mapsfrom} (C',D')$. By the assumptions $(v,w) \in [\![C',D']\!]$.

For the induction step, suppose that g is of the form $g' \wedge g''$. Then $(v,w) \models g'$. By induction hypothesis, there exist C'', D'' such that $(C,D) \overset{g'}{\Mapsfrom} (C'',D'')$ and

$(v, w) \in [\![C'', D'']\!]$. Since $(v, w) \models g''$, we can use the induction hypothesis once more to infer that there exist C', D' such that $(C'', D'') \overset{g''}{\Lleftarrow} (C', D')$ and $(v, w) \in [\![C', D']\!]$. Moreover, by rule $R5$, $(C, D) \overset{g}{\Lleftarrow} (C', D')$.

"$\supseteq$" Assume $(C, D) \overset{g}{\Lleftarrow} (C', D')$ and $(v, w) \in [\![C', D']\!]$. By induction on size of the derivation of $(C, D) \overset{g}{\Lleftarrow} (C', D')$, we establish $(v, w) \in [\![C, D]\!]$ and $(v, w) \models g$. There are five cases, depending on the last rule r used in the derivation of $(C, D) \overset{g}{\Lleftarrow} (C', D')$.

1. $r = R1$. Then $C = C'$, $D = D'$ and $C \models \mathcal{C}(D, g)$. Let g be of the form $x_i - x_j \prec e$. Hence $(v, w) \in [\![C, D]\!]$ and $v \models \mathcal{C}(D, g)$. By the first statement $(v, w) \models x_i - x_j \prec^D_{ij} e^D_{ij}$, and by the second statement $v \models e^D_{ij} \ (\prec^D_{ij} \implies \prec) \ e$. Combination of these two observations, using parts (2) and (4) of Lemma 11.1 yields $(v, w) \models g$.

2. $r = R2$. Then $C = C'$, $D' = D[g]$ and $C \models \neg\mathcal{C}(D, g)$. Hence $(v, w) \models g$ and $v \models \neg\mathcal{C}(D, g)$. Let g be of the form $x_i - x_j \prec e$. By Lemma 11.1(6), $v \models e \ \neg(\prec^D_{ij} \implies \prec) \ e^D_{ij}$. Using parts (2) and (4) of Lemma 11.1, combination of these two observations yields $(v, w) \models x_i - x_j \prec^D_{ij} e^D_{ij}$. Since trivially (v, w) is a model for all the other guards in D, $(v, w) \in [\![C, D]\!]$.

3. $r = R3$. Then $C' = C \cup \{\mathcal{C}(D, g)\}$ and $D' = D$. Let g be of the form $x_i - x_j \prec e$. We have $(v, w) \in [\![C, D]\!]$. This implies $(v, w) \models x_i - x_j \prec^D_{ij} e^D_{ij}$. We also have $v \models e^D_{ij} \ (\prec^D_{ij} \implies \prec) \ e$. Combination of these two observations, using parts (2) and (4) of Lemma 11.1 yields $(v, w) \models g$.

4. $r = R4$. Then $C' = C \cup \{\neg\mathcal{C}(D, g)\}$ and $D' = D[g]$. We have $v \models \neg\mathcal{C}(D, g)$ and $(v, w) \models g$. Let g be of the form $x_i - x_j \prec e$. By Lemma 11.1(6), $v \models e \ \neg(\prec^D_{ij} \implies \prec) \ e^D_{ij}$. Using parts (2) and (4) of Lemma 11.1 yields $(v, w) \models x_i - x_j \prec^D_{ij} e^D_{ij}$. Since trivially (v, w) is a model for all other guards in D, $(v, w) \in [\![C, D]\!]$.

5. $r = R5$. Then g is of the form $g' \wedge g''$ and there are C'', D'' such that $(C, D) \overset{g'}{\Lleftarrow} (C'', D'')$ and $(C'', D'') \overset{g''}{\Lleftarrow} (C', D')$. By induction hypothesis, $(v, w) \in [\![C'', D'']\!]$ and $(v, w) \models g''$. Again by induction hypothesis, $(v, w) \in [\![C, D]\!]$ and $(v, w) \models g'$. It follows that $(v, w) \models g$.

□

Canonicalization

Each DBM can be brought into canonical form using classical algorithms for computing all-pairs shortest paths, for instance the Floyd-Warshall (FW) algorithm [56]. In the parametric case, we also apply this approach except that now we run FW *symbolically*.

The algorithm repeatedly compares the difference between two clocks to the difference obtained by looking at the difference when an intermediate clock is taken into account (the comparison used in Definition 11.9). In the symbolic case the result is, in general, a (possibly empty, finite) set of constrained PDBMs, rather than just a single matrix.

Below, we describe the computation steps of the symbolic FW algorithm in SOS style. Recall that the FW algorithm consists of three nested for-loops, for indices k, i and j, respectively. Correspondingly, in the SOS description of the symbolic version, we use configurations of the form (k, i, j, C, D), where (C, D) is a constrained PDBM and $k, i, j \in [0, m+1]$ record the values of indices. In the rules below, k, i, j range over $[0, m]$.

$$\frac{(C, D) \overset{x_i - x_j \ \prec_{ik} \wedge \prec_{kj} \ e_{ik} + e_{kj}}{\Leftarrow\!\shortmid} (C', D')}{(k, i, j, C, D) \rightarrow_{FW} (k, i, j+1, C', D')}$$

$$(k, i, m+1, C, D) \rightarrow_{FW} (k, i+1, 0, C, D)$$

$$(k, m+1, 0, C, D) \rightarrow_{FW} (k+1, 0, 0, C, D)$$

We write $(C, D) \rightarrow_c (C', D')$ if there exists a sequence of $\rightarrow_{FW}$ steps leading from configuration $(0, 0, 0, C, D)$ to configuration $(m+1, 0, 0, C', D')$. In this case, we say that (C', D') is an *outcome* of the symbolic Floyd-Warshall algorithm on (C, D). It is easy to see that the set of all outcomes is finite and can be effectively computed. If the semantics of (C, D) is empty, then the set of outcomes is also empty. We write $(C, D) \overset{g}{\Leftarrow\!\shortmid}_c (C', D')$ iff $(C, D) \overset{g}{\Leftarrow\!\shortmid} (C'', D'') \rightarrow_c (C', D')$, for some C'', D''.

The following lemma says that if we run the symbolic Floyd-Warshall algorithm, the union of the semantics of the outcomes equals the semantics of the original constrained PDBM.

Lemma 11.5 $[\![C, D]\!] = \bigcup\{[\![C', D']\!] \mid (C, D) \rightarrow_c (C', D')\}$.

Proof. By an inductive argument, using Lemma 11.4 and the fact that, for any valuation (v, w) in the semantics of (C, D),

$$\begin{array}{lll} (v, w) & \models & x_i - x_k \ \prec_{ik} \ e_{ik} \ \text{ and} \\ (v, w) & \models & x_k - x_j \ \prec_{kj} \ e_{kj}, \ \text{ and therefore by Lemma 11.1(5)} \\ (v, w) & \models & x_i - x_j \ \prec_{ik} \wedge \prec_{kj} \ e_{ik} + e_{kj}. \end{array}$$

□

Lemma 11.6 *Each outcome of the symbolic Floyd-Warshall algorithm is a constrained PDBM in canonical form.*

Proof. As in [56]. □

Non-parametric DBMs can be canonicalized in $\mathcal{O}(n^3)$, where n is the number of clocks. In the parametric case, however, each operation of comparing the bound $D(x, x')$ to the weight of another path from x to x' may give rise to two new PDBMs if this comparison leads to a split. Then the two PDBMs must both be canonicalized to obtain all possible PDBMs with tightest bounds. Still, that part of these two PDBMs which was already canonical, does not need to be investigated again. So in the worst case, the cost of the algorithm becomes $\mathcal{O}(2^{n^3})$. In practice, it turns out that this is hardly ever the case.

Resetting Clocks

A third operation on PDBMs that we need is resetting clocks. Since we do not allow parameters in reset sets, the reset operation on PDBMs is essentially the same as for DBMs, see [147]. If D is a PDBM and r is a singleton reset set $\{x_i := b\}$, then $D[r]$ is the PDBM obtained by (1) replacing each bound D^{ij}, for $j \neq i$, by $(e_{0j} + b, \prec_{0j})$; (2) replacing each bound D^{ji}, for $j \neq i$, by $(e_{j0} - b, \prec_{j0})$. We generalize this definition to arbitrary reset sets by

$$D[x_{i_1} := b_1, \ldots, x_{i_h} := b_h] \quad = \quad D[x_{i_1} := b_1] \ldots [x_{i_h} := b_h].$$

It easily follows from the definitions that resets preserves canonicity.

Lemma 11.7 *If (C, D) is canonical then $(C, D[r])$ is canonical as well.*

The following lemma characterizes the reset operation semantically.

Lemma 11.8 *Let (C, D) be a constrained PDBM in canonical form, $v \in [\![C]\!]$, and w a clock valuation. Then $w \in [\![D[r]]\!]_v$ iff $\exists w' \in [\![D]\!]_v : w = w'[r]$.*

Proof. We only prove the lemma for singleton resets. Using Lemma 11.7, the generalization to arbitrary resets is straightforward. Let $r = \{x_i := b\}$ and $D' = D[r]$.

"$\Leftarrow$" Suppose $w' \in [\![D]\!]_v$ and $w = w'[r]$. In order to prove $w \in [\![D']\!]_v$, we must show that $(v, w) \models D'_{kj}$, for all k and j. There are four cases:

1. $k \neq i \neq j$. Then $D'_{kj} = D_{kj}$. Since $(v, w') \models D_{kj}$ and w and w' agree on all clocks occurring in D_{kj}, $(v, w) \models D'_{kj}$.
2. $k = i = j$. Then $D'_{kj} = D_{kj}$. Since $(v, w') \models D_{ii}$, $0 \prec_{ii} e_{ii}[v]$. Hence $(v, w) \models D'_{kj}$.
3. $k \neq i = j$. Then $D'_{kj} = x_k - x_j \prec_{k0} e_{k0} - b$. Using that $(v, w') \models D_{k0}$, we derive $w(x_k) - w(x_j) = w'(x_k) - b \prec_{k0} e_{k0}[v] - b$. Hence $(v, w) \models D'_{kj}$.

4. $k = i \neq j$. Then $D'_{kj} = x_k - x_j \prec_{0j} e_{0j} + b$. Using that $(v, w') \models D_{0j}$, we derive $w(x_k) - w(x_j) = b - w'(x_j) \prec_{0j} e_{0j}[v] + b$. Hence $(v, w) \models D'_{kj}$.

"$\Rightarrow$" Suppose $w \in [\![D']\!]_v$. We have to prove that there exists a clock valuation $w' \in [\![D]\!]_v$ such that $w = w'[r]$. Clearly we need to choose w' in such a way that, for all $j \neq i$, $w'(x_j) = w(x_j)$. This means that, for any choice of $w'(x_i)$, for all $j \neq i \neq k$, $v, w' \models D_{jk}$. Using the same argument as in the proof of Lemma 11.2, we can find a value for $w'(x_i)$ such that also the remaining simple guards of D are satisfied. $\square$

Time Successors

Finally, we need to transform PDBMs for the passage of time, notation $D\uparrow$. As in the DBMs case [60], this is done by setting the $x_i - x_0$ bounds to $(\infty, <)$, for each $i \neq 0$, and leaving all other bounds unchanged. We have the following lemma.

Lemma 11.9 *Suppose (C, D) is a constrained PDBM in canonical form, $v \in [\![C]\!]$, and w a clock valuation. Then $w \in [\![D\uparrow]\!]_v$ iff $\exists d \geq 0\ \exists w' \in [\![D]\!]_v : w' + d = w$.*

Proof. "$\Leftarrow$" Suppose $d \geq 0$, $w' \in [\![D]\!]_v$ and $w' + d = w$. We claim that $w \in [\![D\uparrow]\!]_v$. For this we must show that for each guard f of $D\uparrow$, $(v, w) \models f$. Let f be of the form $x_i - x_j \prec e$. We distinguish between three cases:

- $i \neq 0 \wedge j = 0$. In this case, by definition of $D\uparrow$, f is of the form $x_i - x_0 < \infty$, and so $(v, w) \models f$ trivially holds.
- $i = 0$. In this case f is also a constraint of D. Since $w' \in [\![D]\!]_v$ we have $(v, w') \models f$, and thus $-w'(x_j) \prec e[v]$. But since $d \geq 0$, this means that $-w(x_j) = -w'(x_i) - d \prec e[v]$ and therefore $(v, w) \models f$.
- $i \neq 0 \wedge j \neq 0$. In this case f is again a constraint of D. Since $w' \in [\![D]\!]_v$ we have $(v, w') \models f$, and therefore $w'(x_i) - w'(x_j) \prec e[v]$. But this means that $w'(x_i) - w'(x_j) = (w(x_i) - d) - (w(x_j) - d) \prec e[v]$ and therefore $(v, w) \models f$.

"$\Rightarrow$" Suppose $w \in [\![D\uparrow]\!]_v$. If $m = 0$ (i.e., there are no clocks) then $D\uparrow = D$ and the rhs of the implication trivially holds (take $w' = w$ and $d = 0$). So assume $m > 0$. For all indices i, j with $i \neq 0$ and $j \neq 0$, $(v, w) \models D_{ij}$. Hence $w(x_i) - w(x_j) \prec_{ij} e_{ij}[v]$. Thus, for any real number t, $w(x_i) - t - (w(x_j) - t) \prec_{ij} e_{ij}[v]$. But this means $(v, w - t) \models D_{ij}$. It remains to be shown that there exists a value d such that in valuation $(v, w - d)$ also the remaining guards D_{0i} and

D_{i0} hold. Let

$$\begin{aligned} t_0 &= \max(0, w(x_1) - e_{10}[v], \ldots, w(x_n) - e_{n0}[v]) \\ t_1 &= \min(w(x_1) + e_{01}[v], \ldots, w(x_n) + e_{0n}[v]) \\ d &= (t_0 + t_1)/2 \\ w' &= w - d \end{aligned}$$

Intuitively, t_0 gives the least amount of time one has to go backwards in time from w to meet all upper bounds of D (modulo strictness), whereas t_1 gives the largest amount of time one can go backwards in time from w without violating any of the lower bounds of D (again modulo strictness). Value d sits right in the middle of these two. We claim that d and w' satisfy the required properties. For any i, since $(v, w) \models D_{0i}$, trivially

$$0 \quad \prec_{0i} \quad w(x_i) + e_{0i}[v] \tag{11.5}$$

Using that D is canonical we have, for any i, j,

$$e_{ji}[v] \; (\prec_{ji} \Longrightarrow \prec_{j0} \wedge \prec_{0i}) \; e_{j0}[v] + e_{0i}[v]$$

and, since $v, w \models D_{ji}$,

$$w(x_j) - w(x_i) \prec_{ji} e_{ji}[v].$$

Using these two observations we infer

$$w(x_j) - e_{j0}[v] \; (\prec_{ji} \Longrightarrow \prec_{j0} \wedge \prec_{0i}) \; w(x_j) - e_{ji}[v] + e_{0i}[v] \prec_{ji} w(x_i) + e_{0i}[v].$$

Hence

$$w(x_j) - e_{j0}[v] \prec_{j0} \wedge \prec_{0i} w(x_i) + e_{0i}[v] \tag{11.6}$$

By inequalities (11.5) and (11.6), each subterm of max in the definition of t_0 is dominated by each subterm of min in the definition of t_1. This implies $0 \leq t_0 \leq t_1$.

Now either $t_0 < t_1$ or $t_0 = t_1$. In the first case it is easy to prove that in valuation (v, w) the guards D_{0i} and D_{i0} hold, for any i:

$$w'(x_i) = w(x_i) - d < w(x_i) - t_0 \leq w(x_i) - (w(x_i) - e_{i0}[v]) = e_{i0}[v]$$

and thus $w'(x_i) < e_{i0}[v]$ and $v, w' \models D_{i0}$. Also

$$-w'(x_i) = -w(x_i) + d < -w(x_i) + t_1 \leq -w(x_i) + (w(x_i) + e_{0i}[v]) = e_{0i}[v]$$

and so $-w'(x_i) < e_{0i}[v]$ and $v, w' \models D_{0i}$.

In the second case, fix an i. If $w(x_i) - e_{i0}[v] < t_0$ then

$$w'(x_i) = w(x_i) - d = w(x_i) - t_0 < w(x_i) - (w(x_i) - e_{i0}[v]) = e_{i0}[v]$$

and thus $w'(x_i) < e_{i0}[v]$ and $v, w' \models D_{i0}$. Otherwise, if $w(x_i) - e_{i0}[v] = t_0$ observe that by $t_0 = t_1$, inequality (11.6) and the fact that, $t_1 = w(x_j) + e_{0j}[v]$, for some j, $\prec_{i0} = \leq$. Since

$$w'(x_i) = w(x_i) - d \leq w(x_i) - t_0 \leq w(x_i) - (w(x_i) - e_{i0}[v]) \leq e_{i0}[v]$$

and thus $w'(x_i) \leq e_{i0}[v]$ this implies $v, w' \models D_{i0}$.

The proof that $v, w' \models D_{0i}$, for all i, in the case where $t_0 = t_1$ proceeds similarly. □

11.3.3 Symbolic Semantics

With the four operations on PDBMs, we can describe the semantics of a parametric timed automaton symbolically.

Definition 11.10 (Symbolic semantics) *The* symbolic semantics *of PTA $\mathcal{A} = (Q, q_0, \rightarrow, I)$ is an LTS. The states are triples (q, C, D) with q a location from Q and (C, D) a constrained PDBM in canonical form. The set of initial states is*

$$\{(q_0, C, D) \mid (\top, \mathsf{E}\!\uparrow) \overset{I(q_0)}{\Lleftarrow}_c (C, D)\},$$

where E is the PDBM with $\mathsf{E}^{ij} = (0, \leq)$, for all i, j. The transitions are defined by the following rule:

$$\frac{q \xrightarrow{a,g,r} q' \ , \ (C, D) \overset{g}{\Lleftarrow}_c (C'', D'') \ , \ (C'', D''[r]\!\uparrow) \overset{I(q')}{\Lleftarrow}_c (C', D')}{(q, C, D) \rightarrow (q', C', D')}.$$

Using Lemma 11.4 and Lemma 11.5, it follows by a simple inductive argument that if state (q, C, D) is reachable in the symbolic semantics and $(v, w) \in [\![C, D]\!]$ then $(v, w) \models I(q)$. It is also easy to see that the symbolic semantics of a PTA is a finitely branching transition system. It may have infinitely many reachable states though. Our search algorithm explores the symbolic semantics in an “intelligent” manner, and for instance stops whenever it reaches a state whose semantics is contained in the semantics of a state that has been encountered before. Despite this, our algorithm need not terminate.

Each run in the symbolic semantics can be simulated by a run in the concrete semantics.

Proposition 11.1 *For each parameter valuation v and clock valuation w, if there is a run in the symbolic semantics of $\mathcal{A}$ reaching state (q, C, D), with $(v, w) \in [\![C, D]\!]$, then this run can be simulated by a run in the concrete semantics $[\![\mathcal{A}]\!]_v$ reaching state (q, w).*

Proof. By induction on the number of transitions in the run.

As basis we consider a run with 0 transitions, i.e., a run that consists of a start state of the symbolic semantics. So this means that (q, C, D) is a start state.

Using the fact that $(v, w) \in [\![C, D]\!]$, the definition of start states, Lemma 11.5 and Lemma 11.4, we know that $q = q_0$, $(v, w) \models I(q_0)$ and $(v, w) \in [\![\top, \mathsf{E}\!\uparrow]\!]$. By Lemma 11.9, we get that there exists a $d \geq 0$ and $w' \in [\![\mathsf{E}]\!]_v$ such that $w' + d = w$. Since $(v, w) \models I(q_0)$ and invariants in a PTA only give upper bounds on clocks, also $(v, w') \models I(q_0)$. It follows that (q_0, w') is a state of the concrete semantics $[\![\mathcal{A}]\!]_v$ and $(q_0, w') \xrightarrow{d} (q_0, w)$. Since $w' \in [\![\mathsf{E}]\!]_v$, w' must be of the form $\lambda x.0$, so (q_0, w') is an initial state of the concrete semantics. This completes the proof of induction basis.

For the induction step, assume that we have a run in the symbolic semantics, ending with a transition $(q', C', D') \to (q, C, D)$. Using the fact that $(v, w) \in [\![C, D]\!]$, the definition of transitions in the symbolic semantics, Lemma 11.5 and Lemma 11.4, we know that there is a transition $q' \xrightarrow{a,g,r} q$ in $\mathcal{A}$, and there are C'', D'' such that $(v, w) \models I(q)$, $(v, w) \in [\![C'', D''[r]\!\uparrow]\!]$ and $(C', D') \overset{g}{\Lsh} (C'', D'')$. By Lemma 11.9, we get that there exists a $d \geq 0$ and $w' \in [\![D''[r]]\!]_v$ such that $w' + d = w$. Since $(v, w) \models I(q)$ and invariants in a PTA only give upper bounds on clocks, also $(v, w') \models I(q)$. It follows that (q, w') is a state of the concrete semantics $[\![\mathcal{A}]\!]_v$ and $(q, w') \xrightarrow{d} (q, w)$. Using Lemma 11.8 we get that there exists a $w'' \in [\![D'']\!]_v$ such that $w' = w''[r]$. Using Lemma 11.5 and Lemma 11.4 again, it follows that $(v, w'') \models g$ and $(v, w'') \in [\![C', D']\!]$. Following Definition 11.10, we already observed that the location invariant holds for any reachable state in the symbolic semantics. In particular, $(v, w'') \models I(q')$. Hence, by definition of the concrete semantics, (q', w'') is a state of the concrete semantics and $(q', w'') \xrightarrow{a} (q, w')$ is a transition in the concrete semantics. By induction hypothesis, there is a path in the concrete semantics leading up to state (q', w''). Extension of this path with the transitions $(q', w'') \xrightarrow{a} (q, w')$ and $(q, w') \xrightarrow{d} (q, w)$ gives the required path in the concrete semantics.

□

For each path in the concrete semantics, we can find a path in the symbolic semantics such that the final state of the first path is semantically contained in the final state of the second path.

Proposition 11.2 *For each parameter valuation v and clock valuation w, if there is a run in the concrete semantics $[\![\mathcal{A}]\!]_v$ reaching a state (q, w), then this run can be simulated by a run in the symbolic semantics reaching a state (q, C, D) such that $(v, w) \in [\![C, D]\!]$.*

Proof. In any execution in the concrete semantics, we can always insert zero delay transitions at any point. Also, two consecutive delay transitions $(q, w) \xrightarrow{d} (q, w + d)$ and $(q, w + d) \xrightarrow{d'} (q, w + d + d')$ can always be combined to a single delay transition $(q, w) \xrightarrow{d+d'} (q, w + d + d')$. Therefore, without loss of generality, we only consider concrete executions that start with a delay transition, and in which there is a strict alternation of action transitions and delay transitions. The proof is by induction on the number of action transitions.

As basis we consider a run $(q_0, w_0) \xrightarrow{d} (q_0, w_0 + d)$, where $w_0 = \lambda x.0$, consisting of a single time-passage transition. By definition of the concrete semanctics, $(v, w_0 + d) \models I(q_0)$. Using Lemma 11.9, we have that $(v, w_0 + d) \in [\![\top, \mathsf{E}\uparrow]\!]$ since $(v, w_0) \in [\![\top, \mathsf{E}]\!]$. From $(v, w_0 + d) \in [\![\top, \mathsf{E}\uparrow]\!]$ and $(v, w_0 + d) \models I(q_0)$, using Lemma 11.4 and Lemma 11.5 we get that there exists C, D such that $(\top, \mathsf{E}\uparrow) \overset{I(q_0)}{\Lleftarrow} (C, D)$ and $(v, w_0 + d) \in [\![C, D]\!]$. By definition, (C, D) is an initial state of the symbolic semantics. This completes the proof of the induction basis.

For the induction step, assume that the run in the concrete semantics of $[\![\mathcal{A}]\!]_v$ ends with transitions $(q'', w'') \xrightarrow{a} (q', w') \xrightarrow{d} (q, w)$. By induction hypothesis, there exists a run in the symbolic semantics ending with a state (q'', C'', D'') such that $(v, w'') \in [\![C'', D'']\!]$.

By definition of the concrete semantics, there is a transition $q'' \xrightarrow{g,a,r} q'$ in $\mathcal{A}$ such that $(v, w'') \models g$ and $w' = w''[r]$. Moreover, we have $q' = q$ and $w = w' + d$ and $(v, w) \models I(q)$. Using Lemma 11.4 and Lemma 11.5 gives that there exists C', D' such that $(C'', D'') \overset{g}{\Lleftarrow}_c (C', D')$ and $(v, w'') \in [\![C', D']\!]$. By Lemma 11.8, $w' \in [\![D'[r]]\!]_v$. Moreover, by Lemma 11.9, $w \in [\![D'[r]\uparrow]\!]_v$. Using $(v, w) \models I(q)$, Lemma 11.4 and Lemma 11.5, we infer that there exists C, D such that $(v, w) \in [\![C, D]\!]$ and $(C', D'[r]\uparrow) \overset{I(q)}{\Lleftarrow}_c (C, D)$. Finally, using the definition of the symbolic semantics, we infer the existence of a transition $(q'', C'', D'') \rightarrow (q, C, D)$.

□

Example 11.2 *Figure 11.2 shows the symbolic state-space of the automaton in Fig. 11.1 represented by constrained PDBMs. In the initial state the invariant $x \leq p$ limits the value of x, and since both clocks have the same value also the value of y. When taking the transition from S0 to S1 we have to compare the parameters* p *and* q. *This leads to a split where in the one case no state is reachable since the region is empty, and in the other (when $q \leq p$) S1 can be reached. From then on no more splits occur and only one new state is reachable.*

11.3.4 Evaluating Properties

We now define the relation $\overset{\phi}{\Lleftarrow}$ which relates a symbolic state and a state formula ϕ to a collection of symbolic states that satisfy ϕ.

In order to check whether a property holds, we break it down into the small basic formulas, namely checking locations and clock guards. Checking that a

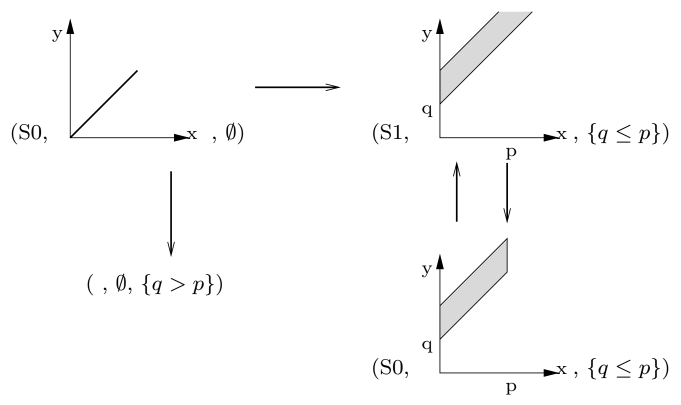


Figure 11.3: The symbolic state space of the PTA in Fig. 11.1.

clock guard holds relies on the definition given earlier, of adding that clock guard to the constrained PDBM. We rely on a special normal form of the state formula, in which all $\neg$ signs have been pushed down to the basic formulas.

Definition 11.11 *State formula ϕ is in* normal form *if all $\neg$ signs in ϕ appear only in front of a subformula that checks a location: $\neg q$.*

Since each simple guard with a $\neg$ sign in front can be rewritten to equivalent simple guard without, for each state formula there is an equivalent one in normal form.

In the following, let f be a simple guard, and ϕ be in normal form.

$$(Q_1)\frac{}{(q,C,D)\overset{q}{\Mapsfrom}(q,C,D)} \qquad (Q_2)\frac{q \neq q'}{(q,C,D)\overset{\neg q'}{\Mapsfrom}(q,C,D)}$$

$$(Q_3)\frac{(C,D)\overset{f}{\Mapsfrom}_c(C',D')}{(q,C,D)\overset{f}{\Mapsfrom}(q,C',D')}$$

$$(Q_4)\frac{(q,C,D)\overset{\phi_1}{\Mapsfrom}(q,C',D')\ ,\ (q,C',D')\overset{\phi_2}{\Mapsfrom}(q,C'',D'')}{(q,C,D)\overset{\phi_1\wedge\phi_2}{\Mapsfrom}(q,C'',D'')}$$

$$(Q_5)\frac{(q,C,D)\overset{\phi_1}{\Mapsfrom}(q,C',D')}{(q,C,D)\overset{\phi_1\vee\phi_2}{\Mapsfrom}(q,C',D')} \qquad (Q_6)\frac{(q,C,D)\overset{\phi_2}{\Mapsfrom}(q,C',D')}{(q,C,D)\overset{\phi_1\vee\phi_2}{\Mapsfrom}(q,C',D')}$$

The following lemma gives the soundness and completeness of relation $\overset{\phi}{\Mapsfrom}$.

Lemma 11.10 *Let $[\![\phi,q]\!]$ denote the set $\{(v,w) \mid (q,w) \models \phi\}$. Then for all properties ϕ in normal form $[\![C,D]\!] \cap [\![\phi,q]\!] = \bigcup \{[\![C',D']\!] \mid (q,C,D) \overset{\phi}{\Mapsfrom} (q,C',D')\}$.*

Proof.

$\subseteq$: We prove that, for all C, D, v, w and q, if $(v,w) \in [\![C,D]\!] \wedge (q,w) \models \phi$ then there are C', D' such that $(v,w) \in [\![C',D']\!] \wedge (q,C,D) \overset{\phi}{\Mapsfrom} (q,C',D')$. We use induction on $|\phi|$, where $|\phi|$ yields the depth of ϕ, as follows. For a location q and a simple guard f, we have $|q| = |\neg q| = |f| = 0$ and for composed properties we have $|\phi_1 \wedge \phi_2| = |\phi_1 \vee \phi_2| = 1 + \max(|\phi_1|, |\phi_2|)$.

- Base cases. Let $|\phi| = 0$ and let $(v,w) \in [\![C,D]\!]$ and $(q,w) \models \phi$.
  * Suppose $\phi = q'$. As $(q,w) \models q'$, clearly, $q = q'$. Since $(q,C,D) \overset{q}{\Mapsfrom} (q,C,D)$, we can take $C = C'$ and $D = D'$ and the result follows.
  * Suppose $\phi = \neg q'$. Similar to the previous case.
  * Suppose $\phi = f$ with f a simple guard. Then $(v,w) \in [\![C,D]\!]$ and $(q,w) \models f$. By Lemma 11.4 we have that there exist C'', D'' such that $(C,D) \overset{f}{\Mapsfrom} (C'',D'')$ and $(v,w) \in [\![C'',D'']\!]$ and Lemma 11.5 yields C', D' with $(C'',D'') \rightarrow_c (C',D')$ and $(v,w) \in [\![C',D']\!]$. Thus, we also have $(q,C,D) \overset{f}{\Mapsfrom} (q,C',D')$.
- Induction step. Let $|\phi| = n+1$ and let $(v,w) \in [\![C,D]\!]$ and $(q,w) \models \phi$.
  * Suppose $\phi = \phi_1 \wedge \phi_2$. Clearly, $(q,w) \models \phi_1$ and $(q,w) \models \phi_2$. By applying the induction hypothesis on ϕ_1, C and D, we derive that there exist C'', D'' such that $(q,C,D) \overset{\phi_1}{\Mapsfrom} (q,C'',D'')$ and $(v,w) \in [\![C'',D'']\!]$. Applying the induction hypothesis on ϕ_2, C'' and D'' yields C', D' such that $(q,C'',D'') \overset{\phi_2}{\Mapsfrom} (q,C',D')$ and $(v,w) \in [\![C',D']\!]$. Then also $(q,C,D) \overset{\phi_1 \wedge \phi_2}{\Mapsfrom} (q,C',D')$.
  * Suppose $\phi = \phi_1 \vee \phi_2$. Clearly, $(q,w) \models \phi_1$ or $(q,w) \models \phi_2$. Suppose $(q,w) \models \phi_1$. The induction hypothesis yields C', D' such that $(q,C,D) \overset{\phi_1}{\Mapsfrom} (q,C',D')$ and $(v,w) \in [\![C',D']\!]$. Then $(q,C,D) \overset{\phi_1 \vee \phi_2}{\Mapsfrom} (q,C',D')$. The case $(q,w) \models \phi_2$ is similar.

$\supseteq$: By induction on the structure of the derivation of $\overset{\phi}{\Mapsfrom}$, we establish that for all v, w, C, D, C', D', if $(q,C,D) \overset{\phi}{\Mapsfrom} (q,C',D')$ and $(v,w) \in [\![C',D']\!]$ then $(v,w) \in [\![C,D]\!]$ and $(q,w) \models \phi$.

- Base cases. The derivation consists of a single step r. Assume $(q,C,D) \overset{\phi}{\Mapsfrom} (q,C',D')$ and $(v,w) \in [\![C',D']\!]$.
  * $r = Q_1$. Then $\phi = q$, $C = C'$, $D = D'$. Then clearly, $(v,w) \in [\![C,D]\!]$ and $(q,w) \models q$.
  * $r = Q_2$. Similar to the previous case.
  * $r = Q_3$. Suppose $\phi = f$ with f a simple guard. Then $(q,C,D) \overset{\phi}{\Mapsfrom} (q,C',D')$ has been derived from $(C,D) \overset{f}{\Mapsfrom}_c (C',D')$. Then there

exist C'', D'' such that $(C, D)\overset{f}{\Lleftarrow}(C'', D'')$ and $(C'', D'')\rightarrow_c(C', D')$. By Lemma 11.5 we have $(v, w) \in [\![C'', D'']\!]$. Then we have by Lemma 11.4 that $(v, w) \models f$ and $(v, w) \in [\![C, D]\!]$.

– Induction step. Assume $(q, C, D) \overset{\phi}{\Lleftarrow} (q, C', D')$ and $(v, w) \in [\![C', D']\!]$ and consider the last rule r used in the derivation of $(q, C, D) \overset{\phi}{\Lleftarrow} (q, C', D')$.

  * $r = Q_4$. Then $\phi = \phi_1 \wedge \phi_2$ and $(q, C, D) \overset{\phi_1}{\Lleftarrow} (q, C'', D'')$ and $(q, C'', D'') \overset{\phi_2}{\Lleftarrow} (q, C', D')$ for some C'', D''. Applying the induction hypothesis to the second statement yields that $(q, w) \models \phi_2$ and $(v, w) \in [\![C'', D'']\!]$. Then applying the induction hypothesis to the first statement yields $(q, w) \models \phi_1$ and $(v, w) \in [\![C, D]\!]$. Then also $(v, w) \models \phi_1 \wedge \phi_2$.
  * $r = Q_5$. Then $\phi = \phi_1 \vee \phi_2$. Then $(q, C, D) \overset{\phi_1}{\Lleftarrow} (q, C', D')$. By the induction hypothesis we have that $(q, w) \models \phi_1$ and $(v, w) \in [\![C, D]\!]$.
  * $r = Q_6$. Similarly to the previous case.

□

11.3.5 Algorithm

We are now in a position to present our model checking algorithm for parametric timed automata. The algorithm displayed in Fig. 11.4 describes how our tool explores the symbolic state-space and searches for constraints on the parameters for which a reachability formula $\exists\Diamond\phi$ holds in a PTA $\mathcal{A}$. The result returned by the algorithm is a set of symbolic states, all of which satisfy ϕ, for any valuation of the parameters and clocks in the state. For invariance properties $\forall\Box\phi$, the tool performs the algorithm on $\neg\phi$, and the result is then a set of symbolic states, none of which satisfies ϕ. The answer to the model checking problem, stated in Section 11.2.2, is obtained by taking the union of the constraint sets from all symbolic states in the result of the algorithm; in the case of an invariance property we take the complement of this set.

A difference between the algorithm in Fig. 11.4 and the standard timed model checking algorithm is that we continue the exploration until either no more new states are found or all paths end in a state satisfying the property. This is because we want to find all the possible constraints on the parameters for which the property holds. Also, the operations on non-parametric DBMs only change the DBM they are applied to, whereas in our case, we may end up with a set of new PDBMs and not just one.

Some standard operations on symbolic states that help in exploring as little as possible, have also been implemented in our tool for parametric symbolic states. Before starting the state-space exploration, our implementation determines the

```
algorithm Reachable($\mathcal{A}$, $\phi$)
    RESULT := $\emptyset$
    PASSED := $\emptyset$
    WAITING := $\{(q_0, C, D) \mid (\top, \mathsf{E}\uparrow) \overset{I(q_0)}{\Lleftarrow}_c (C, D)\}$
    while WAITING $\neq \emptyset$ do
        select $(q, C, D)$ from WAITING
        RESULT := RESULT $\cup \{(q', C', D') \mid (q, C, D) \overset{\phi}{\Lleftarrow} (q', C', D')\}$
        FALSE := $\{(q', C', D') \mid (q, C, D) \overset{\neg\phi}{\Lleftarrow} (q', C', D')\}$
        for each $(q', C', D')$ in FALSE do
            if for all $(q'', C'', D'')$ in PASSED: $(q', C', D') \not\subseteq (q'', C'', D'')$ then
                add $(q', C', D')$ to PASSED
                for each $(q'', C'', D'')$ such that $(q', C', D') \rightarrow (q'', C'', D'')$ do
                    WAITING := WAITING $\cup \{(q'', C'', D'')\}$
    return RESULT
```

Figure 11.4: The parametric model checking algorithm

maximal constant for each clock. This is the maximal value to which the clock is compared in any guard or invariant in the PTA. When the clock value grows beyond this value, we can ignore its real value. This enables us to identify many more symbolic states, and helps termination[2].

11.4 Reducing the Complexity

This section introduces the class of lower bound/upper bound automata and describes several (rather intuitive) observations that simplify the model checking of PTAs in this class. Our results allow us to eliminate parameters in certain cases. Since the complexity of parametric model checking grows very fast in the number of parameters, this is a relevant issue. Secondly, our observations yield a decidability result for lower bound/upper bound automata whereas the corresponding problem for general PTAs is undecidable.

Informally, a positive occurrence of a parameter in a guard or an invariant of a PTA enforces (or contributes to) an upper bound on a clock difference, for instance p in $x - y < 2p$. A negative occurrence of a parameter contributes to a lower bound on a clock difference, for instance q and q' in $y - x > q + 2q'$ ($\equiv x - y < -q - 2q'$) and in $x - y < 2p - q - 2q'$. Hence, a PTA containing the guards $x - y \leq p - q$ and $z < q - p$ is not an L/U automaton.

Definition 11.12 *A parameter $p_i \in P$ is said to* occur *in the linear expression $e = t_0 + t_1 \cdot p_1 + \cdots t_n \cdot p_n$ if $t_i \neq 0$; p_i* occurs positively *in e if $t_i > 0$ and p_i* occurs negatively *in e if $t_i < 0$. A* lower bound parameter *of a PTA $\mathcal{A}$ is a parameter that only occurs negatively in the expressions of $\mathcal{A}$ and an* upper bound parameter *of $\mathcal{A}$ a parameter that only occurs positively in $\mathcal{A}$. We call $\mathcal{A}$ a* lower bound/upper bound (L/U) automaton *if every parameter occurring*

[2]For purely timed model checking this *guarantees* termination.

in $\mathcal{A}$ is either a lower bound parameter or an upper bound parameter of $\mathcal{A}$, but not both.

Example 11.3 *The automaton in Fig. 11.5 is an L/U automaton where* min *is a lower bound parameter and* max *is an upper bound parameter. Also the model of Fischer protocol in Fig. 11.2 is an L/U automaton. Here* min_rw *and* min_delay *are lower bound parameters and* max_rw *and* max_delay *are the upper bound parameters.*

From now on, we work with a fixed set $L = \{l_1, \dots l_K\}$ of lower bound parameters and a fixed set $U = \{u_1, \dots u_M\}$ of upper bound parameters with $L \cap U = \emptyset$ and $L \cup U = P$.

We consider, apart from parameter valuations, also *extended parameter valuations*. Intuitively, an extended parameter valuation is a parameter valuation with values in $\mathsf{R}^{\geq 0} \cup \{\infty\}$, rather than in $\mathsf{R}^{\geq 0}$. Extended parameter valuations are useful in certain cases to solve the verification problem (over non-extended valuations) stated in Section 11.2.3. Working with extended parameter valuations may cause the evaluation of an expression to be undefined. For example, the expression $e[v]$ is not defined for $e = p - q$ and $v(p) = v(q) = \infty$. We require that an extended parameter valuation of an L/U automaton does not assign the value ∞ both to a lower bound parameter and to an upper bound parameter. Then the expression $e[v]$ is defined for every extended valuation of an L/U automaton.

Therefore, we can easily extend notions $e[v]$, $(v, w) \models e$ and $\mathcal{A}[v]$ (defined in Section 11.2) to extended valuations, by using the conventions that $0 \cdot \infty = 0$, that $x - y \prec \infty$ evaluates to true and $x - y \prec -\infty$ to false. In particular, we have $[\![\mathcal{A}]\!]_v = \mathcal{A}[v]$ for extended valuations v and L/U automata $\mathcal{A}$. Moreover, we extend the orders $\sim$ to $\mathsf{R} \cup \{\infty\}$ in the usual way and to extended valuations via point wise extension (i.e. $v \sim v'$ iff $v(p) \sim v'(p)$ for all $p \in P$). We denote an extended valuation of an L/U automaton by a pair (λ, μ), which equals the function λ on the lower bound parameters and μ on the upper bound parameters. We write 0 and ∞ for the functions assigning respectively 0 and ∞ to each parameter.

The following proposition is based on the fact that weakening the guards in $\mathcal{A}$ (i.e. decreasing the lower bounds and increasing the upper bounds) yields an automaton whose reachable states include those of $\mathcal{A}$. Dually, strengthening the guards in $\mathcal{A}$ (i.e. increasing the lower bounds and decreasing the upper bounds) yields an automaton whose reachable states are a subset of those of $\mathcal{A}$. We claim that this proposition, formulated for L/U automata, can be generalized to lower bound and upper bound parameters present in general PTAs. It is however crucial that (by definition) state formulae do not contain parameters. The usefulness of this property (and of several other properties in this section) lies in the fact that it allows to conclude the satisfaction of a property for infinitely many parameter valuations from the satisfaction of that property for one valuation.

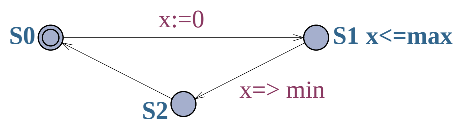


Figure 11.5: Reducing parametric to non-parametric model checking

Proposition 11.3 *Let $\mathcal{A}$ be an L/U automaton and ϕ a state formula. Then*

1. $[\![\mathcal{A}]\!]_{(\lambda,\mu)} \models \exists\Diamond\phi \iff \forall\lambda' \leq \lambda, \mu \leq \mu' : [\![\mathcal{A}]\!]_{(\lambda',\mu')} \models \exists\Diamond\phi.$
2. $[\![\mathcal{A}]\!]_{(\lambda,\mu)} \models \forall\Box\phi \iff \forall\lambda \leq \lambda', \mu' \leq \mu : [\![\mathcal{A}]\!]_{(\lambda',\mu')} \models \forall\Box\phi.$

Proof. (sketch) Both parts of the proposition can be proven by induction on the length of runs in the L/U automata. The crucial observation is that for parameter valuations $\lambda' \leq \lambda$ and $\mu \leq \mu'$ and linear expression e we have that $e[\lambda', \mu] \leq e[\lambda, \mu]$ and $e[\lambda, \mu] \leq e[\lambda, \mu']$. Therefore whenever $((\lambda, \mu), w) \models g$ then $((\lambda', \mu'), w) \models g$.

$\Box$

The following example illustrates how Proposition 11.3 can be used to eliminate parameters from L/U automata.

Example 11.4 *The automaton in Fig. 11.5 is an L/U automaton. Its location S_1 is reachable irrespective of the parameter values. By setting the parameter min to ∞ and max to 0, one checks with a non-parametric model checker that $\mathcal{A}[(\infty, 0)] \models \exists\Diamond S_1$. Then Proposition 11.3 (together with $[\![\mathcal{A}]\!]_v = \mathcal{A}[v]$) yields that S_1 is reachable in $[\![\mathcal{A}]\!]_{(\lambda,\mu)}$ for all extended parameter valuations $0 \leq \lambda, \mu \leq \infty$.*

Clearly, $[\![\mathcal{A}]\!]_{(\lambda,\mu)} \models \exists\Diamond S_2$ iff $\lambda(min) \leq \mu(max) \wedge \lambda(min) < \infty$. We will see in this running example how we can verify this property completely by non-parametric model checking. Henceforth, we construct the automaton $\mathcal{A}'$ from $\mathcal{A}$ by substituting the parameter max by the parameter min yielding a (non L/U) automaton with one parameter, min. If we show that $[\![\mathcal{A}']\!]_v \models \exists\Diamond S_2$ for all valuations v, this essentially means that $[\![\mathcal{A}]\!]_{(\lambda,\mu)} \models \exists\Diamond S_2$ for all λ, μ such that $\mu(max) = \lambda(min) < \infty$ and then Proposition 11.3 implies that $[\![\mathcal{A}]\!]_{(\lambda,\mu)} \models \exists\Diamond S_2$ for all λ, μ with $\lambda(min) \leq \mu(max)$ and $\lambda(min) < \infty$.

The question whether there exists a (non-extended) parameter valuation such that a given (final) location q is reachable, is known as the *emptiness problem* for PTAs. In [15], it is shown that the emptiness problem is undecidable for PTAs with three clocks or more. The following proposition implies that we can solve the emptiness problem for a L/U automaton $\mathcal{A}$ by only considering $\mathcal{A}[(0, \infty)]$, which is a non-parametric timed automaton. Since reachability is decidable for timed automata [13], the emptiness problem is decidable for L/U automata. Then it follows that the dual problem is also decidable for L/U automata. This is the *universality problem* for invariance properties, asking whether an invariance property holds for all parameter valuations.

Proposition 11.4 *Let $\mathcal{A}$ be an L/U automaton. Then $\mathcal{A}[(0,\infty)] \models \exists\Diamond q$ if and only if there exist a (non-extended) parameter valuation (λ,μ) such that $[\![\mathcal{A}]\!]_{(\lambda,\mu)} \models \exists\Diamond q$.*

Proof. The "if"–part is an immediate consequence of Proposition 11.3 and the fact that $\mathcal{A}[(0,\infty)] = [\![\mathcal{A}]\!]_{(0,\infty)}$. For the "only if"–part, assume that α is a run of $\mathcal{A}[(0,\infty)]$ that reaches q. Let T' be the smallest constant occurring in $\mathcal{A}$ and T be the maximum clock value occurring in α. (More precisely, if $\alpha = s_0a_1s_1a_2\dots a_Ns_N$ and $s_i = (q_i,w_i)$, then $T = \max_{i\le N, x\in C}\{w_i(x)\}$; T' compensates for negative constants t_0.) Now, take $\lambda(l_j) = 0$ and $\mu(u_j) = T + |T'| + 1$. Then for every guard or invariant g occurring in $\mathcal{A}$ we have that $((0,\infty),w_i) \models g \implies ((\lambda,\mu),w_i) \models g$. Hence, α is a run of $[\![\mathcal{A}]\!]_{(\lambda,\mu)}$, so $[\![\mathcal{A}]\!]_{(\lambda,\mu)} \models \exists\Diamond q$.

□

Corollary 11.1 *The emptiness problem is decidable for L/U automata.*

Definition 11.13 *A PTA $\mathcal{A}$ is* fully parametric *if clocks are only reset to* 0 *and every linear expression in $\mathcal{A}$ of the form $t_1\cdot p_1+\dots+t_n\cdot p_n$, where $t_i \in \mathsf{Z}$.*

The following proposition is basically the observation in [13], that multiplication of each constant in a timed automaton and in a system property with the same positive factor preserves satisfaction.

Proposition 11.5 *Let $\mathcal{A}$ be fully parametric PTA. Then for all parameter valuations v and all system properties ψ*

$$[\![\mathcal{A}]\!]_v \models \psi \iff \forall t \in \mathsf{R}^{>0} : [\![\mathcal{A}]\!]_{t\cdot v} \models t\cdot\psi,$$

where $t\cdot v$ denotes the valuation $p \mapsto t\cdot v(p)$ and $t\cdot\psi$ the formula obtained from ψ by multiplying each number in ψ by t.

Proof. It is easy to see that for all $t \in \mathsf{R}^{>0}$, $\alpha = s_0a_1s_1a_2\dots a_Ns_N$ with $s_i = (q_i,w_i)$ is a run of $[\![\mathcal{A}]\!]_v$ if and only if $s'_0a_1s'_1\dots a_Ns'_N$ is a run of $[\![\mathcal{A}]\!]_{t\cdot v}$, where $s'_i = (q_i, t\cdot w_i)$ and $t\cdot w_i$ denotes $x \mapsto t\cdot w_i(x)$. □

Then for fully parametric PTAs with one parameter and system properties ψ without constants (except for 0), we have $[\![\mathcal{A}]\!]_v \models \psi$ for all valuations v of P if and only if both $\mathcal{A}[0] \models \psi$ and $\mathcal{A}[1] \models \psi$. The fact that the 0-case has to be treated separatly is illustrated by the (fully parametric) automaton with a single transition equipped with the guard $x < p$. The target location of the transition is not reachable for $p = 0$.

Corollary 11.2 *For fully parametric PTAs with one parameter and properties ψ without constants (except 0), it is decidable whether $\forall v \in [\![C]\!] : [\![\mathcal{A}]\!]_v \models \psi$.*

Example 11.5 *The PTA $\mathcal{A}'$ mentioned in Example 11.4 is a fully parametric timed automaton and the property $\exists\Diamond S_2$ is without constants. We establish that $\mathcal{A}'[0] \models \exists\Diamond S_2$ and $\mathcal{A}'[1] \models \exists\Diamond S_2$. Then Proposition 11.5 implies that $\mathcal{A}'[v] \models \exists\Diamond S_2$ for all v. As shown in Example 11.4, this implies that $[\![\mathcal{A}]\!]_{(\lambda,\mu)} \models \exists\Diamond S_2$ for all λ, μ with $\lambda(min) = \mu(max) < \infty$.*

In the running example, we would like to use the same methods as above to verify that $[\![\mathcal{A}]\!]_{(\lambda,\mu)} \not\models \exists\Diamond S_2$ if $\lambda(min) > \mu(max)$. We can in this case not fill in for $min = max$, since the bound in the constraint is a strict one. The following definition and results allows us to move the strictness of a constraint into the PTA.

Definition 11.14 *Let $P' \subseteq P$ be a set of parameters. Define $\mathcal{A}^{<}_{P'}$ as the automaton obtained by replacing every inequality $x - y \leq e$ in $\mathcal{A}$ by a strict inequality $x - y < e$, provided that e contains at least one parameter from P'. Similarly, define $\mathcal{A}^{\leq}_{P'}$ as the automaton from $\mathcal{A}$ obtained by replacing every inequality $x - y < e$ by a non–strict inequality $x - y \leq e$, provided that e contains at least one parameter from P'. For $\prec = <, \leq$, write $\mathcal{A}^{\prec}$ for $\mathcal{A}^{\prec}_{P}$. Moreover, define $v \prec_{P'} v'$ by $v(p) \prec v'(p)$ if $p \in P'$ and $v(p) = v'(p)$ otherwise.*

Proposition 11.6 *Let $\mathcal{A}$ be an L/U automaton. Then*

1. $[\![\mathcal{A}^{\leq}]\!]_{(\lambda,\mu)} \models \exists\Diamond\phi \implies \forall\lambda' < \lambda, \mu < \mu' : [\![\mathcal{A}]\!]_{(\lambda',\mu')} \models \exists\Diamond\phi.$
2. $[\![\mathcal{A}^{<}]\!]_{(\lambda,\mu)} \models \forall\Box\phi \iff \forall\lambda < \lambda', \mu' < \mu : [\![\mathcal{A}]\!]_{(\lambda',\mu')} \models \forall\Box\phi.$

Proof.

1, $\implies$ Let e be a linear expression occuring in $\mathcal{A}$. Then we can write $e = t_0 + e_1 + e_2$, where $t_0 \in \mathsf{Z}$, e_1 is an expression over the upperbound parameters and e_2 an expression over the lower bound parameters. Then we have

$$\begin{aligned}
\mu \leq \mu' &\implies e_1[\mu] \leq e_1[\mu'],\\
\lambda' \leq \lambda &\implies e_2[\lambda'] \leq e_2[\lambda],\\
\lambda' \leq \lambda, \mu \leq \mu' &\implies e[(\lambda,\mu)] \leq\ e[(\lambda',\mu')].
\end{aligned}$$

If there is at least one parameter occuring in e_1 or e_2 respectively then respectively

$$\begin{aligned}
\mu < \mu' &\implies e_1[\mu] < e_1[\mu']\\
\lambda' < \lambda &\implies e_2[\lambda] < e_2[\lambda'].
\end{aligned}$$

Thus if there is at least one parameter occuring in e, then

$$\lambda' < \lambda, \mu < \mu' \implies e[(\lambda,\mu)] < e[(\lambda',\mu')].$$

Now, let (λ,μ) be an extended valuation. Let $g \equiv x - y \prec e$ be a simple guard occuring in $\mathcal{A}^{\leq}$ and let $g' \equiv x - y \prec' e$ be the corresponding guard in $\mathcal{A}$. Assume that $(w,(\lambda,\mu)) \models g$, i.e. $w(x) - w(y) \prec e[(\lambda,\mu)]$. We show that $(w,(\lambda,\mu)) \models g'$. We distinguish two cases.

case 1: There exists a parameter occuring in e. Then $w(x) - w(y) \prec e[(\lambda,\mu)] < e[(\lambda',\mu')]$. Then certainly $(w,(\lambda,\mu)) \models g' \equiv x - y \prec' e$.

case 2: The expression e does not contain any parameter. Then $g' \equiv g$ and hence $(w,(\lambda,\mu)) \models g'$.

Now it easily follows that every run of $[\![\mathcal{A}^{\leq}]\!]_{(\lambda,\mu)}$ is also a a run of $[\![\mathcal{A}]\!]_{(\lambda',\mu')}$. Thus, if a state satisfying ψ is reachable in $[\![\mathcal{A}^{\leq}]\!]_{(\lambda,\mu)}$ then it is also reachable in $[\![\mathcal{A}]\!]_{(\lambda',\mu')}$.

2, $\implies$: This follows from 1. Assume that $[\![\mathcal{A}^{<}]\!]_{(\lambda,\mu)} \models \forall\Box\phi$ and let λ', μ' be such that $\lambda < \lambda'$, $\mu' < \mu$. Since $[\![\mathcal{A}^{<}]\!]_{(\lambda,\mu)} \not\models \exists\Diamond\neg\phi$, we have

$$\neg\forall\lambda'' < \lambda', \mu' < \mu'' : [\![\mathcal{A}^{<}]\!]_{(\lambda'',\mu'')} \models \exists\Diamond\neg\phi.$$

Then contraposition of statement (1) of this proposition together with $(\mathcal{A}^{<})^{\leq} = \mathcal{A}^{\leq}$ yields $[\![\mathcal{A}^{\leq}]\!]_{(\lambda',\mu')} \not\models \exists\Diamond\neg\phi$. As $\mathcal{A}$ imposes stronger bounds than $\mathcal{A}^{\leq}$, also $[\![\mathcal{A}]\!]_{(\lambda',\mu')} \not\models \exists\Diamond\neg\phi$, i.e. $[\![\mathcal{A}]\!]_{(\lambda',\mu')} \models \forall\Box\phi$.

2, $\Longleftarrow$: Let (λ,μ) be an extended valuation and assume that $[\![\mathcal{A}]\!]_{(\lambda',\mu')} \models \forall\Box\phi$ for all $\lambda' > \lambda$, $\mu' < \mu$. Assume that α is a run of $[\![\mathcal{A}^{<}]\!]_{(\lambda,\mu)}$. We construct $\lambda' > \lambda$ and $\mu' < \mu$ such that α is also a run of $[\![\mathcal{A}]\!]_{(\lambda',\mu')}$. (Then we are done: since $[\![\mathcal{A}]\!]_{(\lambda',\mu')} \models \forall\Box\phi$, the last state of α satisfies ϕ. Hence every reachable state of $[\![\mathcal{A}]\!]_{(\lambda,\mu)}$ satisfies ϕ, i.e. $[\![\mathcal{A}]\!]_{(\lambda,\mu)} \models \forall\Box\phi$.)

We use the following notation. We write $v = (\lambda,\mu)$ and $v' = (\lambda',\mu')$. For a run α, we write $\alpha = s_0 a_1 s_1 a_2 \ldots a_N s_N$ with $s_i = (q_i, w_i)$, $I(q_i) = \wedge_j^{J'} I_{ij}$, $I_{ij} = x_{ij} \prec_{ij} E_{ij}$. As α is a run, there exists a transition $q_{i-1} \xrightarrow{g_i,a_i,r_i} q_i$ for each i, $1 \leq i \leq N$. We write the guard on this transition by $g_i = \wedge_j^J g_{ij}$, $g_{ij} = x_{ij} - y_{ij} \prec_{ij} e_{ij}$. Finally, if g is a guard or invariant in $\mathcal{A}$, then we denote the corresponding guard or invariant in $\mathcal{A}^{<}$ by $g^{<}$, i.e. the guard that is obtained as in Definition 11.14.

If neither the guards g_{ij} nor the invariants I_{ij} contains a parameter, then we can take v' arbitrarily and we have that α is a run of $[\![\mathcal{A}]\!]_{v'}$. Therefore, assume that at least one of the guards g_{ij} or invariants I_{ij} contains a parameter. Then, by definition of $\mathcal{A}^{<}$, this guard or invariant contains a strict bound. In this case, we construct $\lambda' > \lambda$ and $\mu' < \mu$ such that $w_i(x - y) < e[(\lambda',\mu')] < e[(\lambda,\mu)]$ if $g = x - y < e$ is an invariant I_{ij} or guard g_{ij} as above. Informally, we use the minimum "distance" $e[(\lambda,\mu)] -$

$w_i(x-y)$ occurring in α to slightly increase the lower bounds and slightly decrease the upper bounds yielding $\lambda < \lambda'$ and $\mu < \mu'$.

Let

$$\begin{aligned} T_0 &= \min_{i\le N, j\le J'} \{E_{ij}[v] - w_i(x_{ij}) \mid \prec_{ij} = <\}, \\ T_1 &= \min_{i\le N, j\le < J} \{e_{ij}[v] - w_i(x_{ij} - y_{ij}) \mid \prec_{ij} = <\}, \\ 0 < &T < \min\{T_0, T_1\}, \end{aligned}$$

with the convention that $\min\emptyset = \infty$. At least one of the inequalities $\prec_{ij}$ is strict, since at least one of the guards contains a parameter. Hence $T_0 < \infty$ or $T_1 < \infty$. Since $(v, w_i) \models I_{ij} \wedge g_{ij}$, we have we have that $T_0 \geq 0$ and $T_1 \geq 0$. Hence $\infty > \min\{T_0, T_1\} > 0$ and the requested T exists. The crucial property is that if $g_{ij} \equiv x_{ij} - y_{ij} < e_{ij}$ or $g_{ij} \equiv x_{ij} - y_{ij} < E_{ij}$ we have respectively

$$\begin{aligned} T &< e_{ij}[v] - w_i(x_{ij} - y_{ij}) \\ T &< E_{ij}[v] - w_i(x_{ij} - y_{ij}). \end{aligned}$$

Now, let T' be the sum of the constants appearing in the guards and invariants that appear in the run α i.e.

$$T' = \sum_{i\le N, j\le J'} \text{sum\_of\_const}(E_{ij}) + \sum_{i\le n, j\le J} \text{sum\_of\_const}(e_{ij}),$$

where $\text{sum\_of\_const}(t_0 + t_1 \cdot p_1 + \cdots + t_n \cdot p_n) = |t_1| + \cdots + |t_n|$. Since at least one of the guards or invariants contains a parameter, we have $T' > 0$.

Now, take $v' = (\lambda + \frac{T}{T'}, \mu - \frac{T}{T'})$ and consider $g_{ij} \equiv x_{ij} - y_{ij} \prec_{ij} e_{ij}$. We claim that $(v', w_i) \models g_{ij}$.

case 1: The expression g_{ij} does not contain any parameter. Then $g_{ij} = g_{ij}^{<}$ and $e_{ij}[v] = e_{ij}[v']$. Since $(w_i, v) \models g_{ij}$, also $(w_i, (v')) \models g_{ij}^{<}$.

case 2: There exists a parameter occurring in e. We can write $e = t_0 + t_1 \cdot u_1 + \cdots + t_M \cdot u_M - t_1' \cdot l_1 - \cdots - t_K' \cdot l_K$, with $t_i \geq 0$, $t_i' \geq 0$ for $i > 0$. Then

$$\begin{aligned} e_{ij}[v'] &= t_0 + \sum_{k=1}^{M} t_k \cdot (\mu_k' - \frac{T}{T'}) - \sum_{k=1}^{K} t_k \cdot (\lambda_k' + \frac{T}{T'}) \\ &= (t_0 + \sum_{k=1}^{M} t_k \cdot \mu_k' - \sum_{k=1}^{K} t_k \cdot \lambda_k') - \frac{T}{T'} \cdot (\sum_{k=1}^{M} t_k + \sum_{k=1}^{K} t_k') \\ &\geq e_{ij}[v] - T \\ &> e_{ij}[v] - (e_{ij}[v] - w_i(x_{ij} - y_{ij})) \\ &= w_i(x_{ij} - y_{ij}). \end{aligned}$$

Therefore $(w_i, v') \models x_{ij} - y_{ij} < g_{ij}^{<}$ and then also $(w_i, v') \models x_{ij} - y_{ij} \prec_{ij} g_{ij}^{<}$.

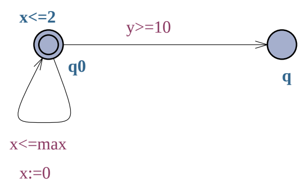


Figure 11.6: The converse of Proposition 11.6 (1) does not hold.

Combining the cases (1) and (2) yields that for all i, j, $(w_i, v') \models x_{ij} - y_{ij} \prec_{ij} g_{ij}^{<}$. Similarly, one proves that $(w_i, v') \models x_{ij} - y_{ij} \prec_{ij} I_{ij}$. Therefore, α is a fun of $[\![\mathcal{A}]\!]_{(\lambda',\mu')}$.

□

The previous result concerns the automaton that is obtained when all the strict inequalities in the automaton are changed into nonstrict ones, (or the other way around). Sometimes, we want to 'toggle' only a some of the inequalities. Then the following result can be applied.

Corollary 11.3 *Let $\mathcal{A}$ be an L/U automaton and $P' \subseteq P$.*

1. $[\![\mathcal{A}_{P'}^{\leq}]\!]_{(\lambda,\mu)} \models \exists\Diamond\phi \implies \forall\lambda' <_{P'} \lambda, \mu <_{P'} \mu' : [\![\mathcal{A}]\!]_{(\lambda',\mu')} \models \exists\Diamond\phi.$
2. $[\![\mathcal{A}_{P'}^{<}]\!]_{(\lambda,\mu)} \models \forall\Box\phi \iff \forall\lambda <_{P'} \lambda', \mu' <_{P'} \mu : [\![\mathcal{A}]\!]_{(\lambda',\mu')} \models \forall\Box\phi.$

Proof. Let (λ, μ) be an extended valuation. Let $\mathcal{A}_0$ be the automaton obtained from $\mathcal{A}$ by substituting p by $(\lambda, \mu)(p)$ for every $p \notin P'$. Then $\mathcal{A}_{P'}^{<} = \mathcal{A}_0{}^{<}$ and $\mathcal{A}_{P'}^{\leq} = \mathcal{A}_0{}^{\leq}$. Now the result follows by applying Proposition 11.6 to $\mathcal{A}_0$. □

The following example shows that the converse of Proposition 11.6, item 1 does not hold.

Example 11.6 *Consider the automaton $\mathcal{A}$ in Fig. 11.6. Recall that the clocks x and y are initially 0. Then $\mathcal{A} = \mathcal{A}^{\leq}$ and the location q is reachable if $max > 0$ but not if $max = 0$. Thus $\forall\lambda' < 0, 0 < \mu' : [\![\mathcal{A}]\!]_{(\lambda',\mu')} \models \exists\Diamond\phi$, but not $[\![\mathcal{A}^{\leq}]\!]_{(0,0)} \models \exists\Diamond\phi$.*

We believe the class of L/U automata can be very useful in practice. Several examples known from the literature fall into this class, or can be modeled slightly differently to achieve this. We mention the IEEE Root Contention protocol [91], Fischer's mutual exclusion protocol [102], the (toy) rail road crossing example from [15], the Bounded Retransmission protocol (when considering a fixed value for the integer variables), and the Biphase Mark protocol (with minor adaptations). Moreover, the MMT models from [117] can be encoded straightforwardly into L/U automata.

We expect that many other distributed systems and protocols can be modeled with L/U automata, since it is quite natural to have the duration of an event (such as the communication delay in a channel, the computation time needed to produce a result, the time required to open the gate in a rail road crossing) lying between a lower bound and an upper bound and these bounds are often the parameters of the system.

Section 11.4.1 and Section 11.5 show that the techniques discussed in this section to eliminate parameters in L/U models reduce the verification effort significantly and possibly leads to a completely non-parametric model.

11.4.1 Verification of Fischer's Mutual Exclusion Protocol

In this section, we apply the results from the previous section to verify the Fischer protocol with 2 processes. We can establish the sufficiency of the protocol constraints by non-parametric model checking and the necessity of the constraints by eliminating three of the four parameters.

Consider the Fischer protocol from Section 11.2.4 again. In this section, we consider a system $\mathcal{A}$ consisting of two parallel processes P_1 and P_2. It is clear that $\mathcal{A}$ is a fully parametric L/U automaton: *min_rw* and *min_delay* are lower bound parameters and *max_rw* and *max_delay* upper bound parameters.

The mutual exclusion property is expressed by the formula

$$\Phi_{ME} \equiv \forall\Box\neg(P_1.cs \wedge P_2.cs).$$

Recall that assuming the basic constraints $B_{ME} \equiv 0 \leq \mathit{min\_rw} < \mathit{max\_rw} \wedge 0 \leq \mathit{min\_delay} < \mathit{max\_delay}$, mutual exclusion is guaranteed if and only if $C_{ME} \equiv \mathit{max\_rw} \leq \mathit{min\_delay}$. Thus we prove that for all valuations v: $v \models B_{ME} \implies (\llbracket\mathcal{A}\rrbracket_v \models \Phi_{ME} \iff v \models C_{ME})$.

Sufficiency of the Constraints

We show that the constraints assure mutual exclusion, that is

$$\text{if } v \models C_{ME} \wedge B_{ME}, \text{ then } \mathcal{A}[v] \models \Phi_{ME}.$$

We perform the substitution

$$\mathit{min\_rw} \mapsto 0, \mathit{max\_delay} \mapsto \infty, \mathit{min\_delay} \mapsto \mathit{max\_rw}$$

to obtain a fully parametric automaton $\mathcal{A}'$ with one parameter, *max_rw*. We have established by non-parametric model checking that $\mathcal{A}'[0] \models \Phi_{ME}$ and $\mathcal{A}'[1] \models \Phi_{ME}$. Now Proposition 11.5 yields that $\llbracket\mathcal{A}'\rrbracket_v \models \Phi_{ME}$ for all valuations

v (where only the value of max_delay matters). This means that $[\![\mathcal{A}]\!]_v \models \Phi_{ME}$ if $v(min\_rw) = 0$, $v(max\_rw) = v(min\_delay)$ and $v(max\_delay) = \infty$. Then Proposition 11.3 yields that the mutual exclusion property, which is an invariance property, also holds if the lower bound parameters min_rw and min_delay are increased and if the upper bound parameter max_rw is decreased. More precisely, Proposition 11.3 implies that $[\![\mathcal{A}]\!]_v \models \Phi_{ME}$ for all v with $0 \leq v(min\_rw)$, $v(max\_rw) \leq v(min\_delay)$ and $v(max\_delay) \leq \infty$. Then, in particular, if $v \models C_{ME} \wedge B_{ME}$, then $[\![\mathcal{A}]\!]_v \models \Phi_{ME}$.

Necessity of the Constraints:

We show that if

$$v \models B_{ME} \wedge \neg C_{ME} \implies \mathcal{A}[v] \models \neg\Phi_{ME},$$

i.e. that if $v \models min\_rw < max\_rw \ \wedge \ min\_delay < max\_delay \ \wedge \ min\_delay < max\_rw$, then $\mathcal{A}[v] \models \neg\Phi_{ME} \equiv \exists\Diamond(P_1.cs \wedge P_2.cs)$. We construct the automaton $\mathcal{A}^{\leq}$ and proceed in two steps.

Step 1 Let v_0 be the valuation $v_0(min\_delay) = v_0(max\_delay) = 0$ and $v_0(min\_rw) = v_0(max\_delay) = 1$. By non-parametric model checking we have established that

$$\mathcal{A}^{\leq}[0] \models \neg\Phi_{ME} \tag{11.7}$$

$$\mathcal{A}^{\leq}[v_0] \models \neg\Phi_{ME}. \tag{11.8}$$

We show that it follows that for all v

$$v \models 0 = min\_delay = max\_delay \leq min\_rw = max\_rw \implies \mathcal{A}^{\leq}[v] \models \neg\Phi_{ME}. \tag{11.9}$$

Assume $v \models 0 = min\_delay = max\_delay \leq min\_rw = max\_rw$. Consider $t = v(min\_rw)$. If $v(min\_rw) = 0$, then (11.7) shows that $[\![\mathcal{A}^{\leq}]\!]_v \models \neg\Phi_{ME}$. Therefore, assume $v(min\_rw) > 0$ and consider $\frac{v}{t} \equiv \lambda x.\frac{v(x)}{t}$. It is not difficult to see that

$$\frac{v}{t} \models 0 = min\_delay = max\_delay \leq min\_rw = max\_rw = 1.$$

Therefore, (11.8) yields $[\![\mathcal{A}^{\leq}]\!]_{\frac{v}{t}} \models \neg\Phi_{ME}$. Since $\mathcal{A}^{\leq}$ is a fully parametric PTA, Proposition 11.5 yields that $[\![\mathcal{A}^{\leq}]\!]_v \models \neg\Phi_{ME}$.

Step 2 Let $\mathcal{A}'$ be the automaton that is constructed from $\mathcal{A}^{\leq}$ by performing the following substitution $min\_delay \mapsto 1$, $max\_delay \mapsto 1$, $min\_rw \mapsto max\_rw$. By parametric model checking we have established

$$v \models 1 \leq max\_rw \implies [\![\mathcal{A}']\!]_v \models \neg\Phi_{ME}. \tag{11.10}$$

This means that if

$$v \models \mathit{min\_delay} = \mathit{max\_delay} = 1 \leq \mathit{min\_rw} = \mathit{max\_rw} \implies [\![\mathcal{A}^{\leq}]\!]_v \models \neg\Phi_{ME}.$$

By a argument similar to the one we used to prove (11.9), (where now the case $v(\mathit{min\_delay}) = 0$ is covered by statement (11.9) in Step 1.), we can use Proposition 11.5 to show that

$$v \models \mathit{min\_delay} = \mathit{max\_delay} \leq \mathit{min\_rw} = \mathit{max\_rw} \implies [\![\mathcal{A}^{\leq}]\!]_v \models \neg\Phi_{ME}.$$

Now, Proposition 11.3 yields that $\neg\Phi_{ME}$ – which is a reachability property – also holds if the values for the lower bounds are decreased and the values for the upper bounds are increased. Note that we may increase *max_delay* as much as we want; $v(\mathit{max\_delay})$ may be larger than $v(\mathit{min\_rw})$. Thus we have

$$\begin{aligned} v \models \mathit{min\_rw} \leq \mathit{max\_rw} \,\wedge\, \mathit{min\_delay} \leq \mathit{max\_delay} \,\wedge\, \mathit{min\_delay} \leq \mathit{max\_rw} \\ \implies [\![\mathcal{A}^{\leq}]\!]_v \models \neg\Phi_{ME} \end{aligned}$$

and then Proposition 11.6 yields that

$$\begin{aligned} v \models \mathit{min\_rw} < \mathit{max\_rw} \,\wedge\, \mathit{min\_delay} < \mathit{max\_delay} \,\wedge\, \mathit{min\_delay} < \mathit{max\_rw} \\ \implies [\![\mathcal{A}]\!]_v \models \neg\Phi_{ME}. \end{aligned}$$

We have checked the result formulated in statement 11.10 with our prototype implementation. The experiment was performed on a SPARC Ultra in 2 seconds CPU time and 7.7 Mb of memory. We also tried to verify the protocol model without any substitutions or changing of bounds with our prototype, which did not terminate within 20 hours. Since we observed that the constraints lists of the states explored kept growing, we concluded that this experiment would not terminate at all. (Recall that parametric verification is undecidable.) Therefore, we can conclude that in some cases, the techniques for L/U automata yield results even if the state-space exploration algorithm seemingly does not terminate on the original model.

The substitutions and techniques used in the verification to eliminate parameters are ad hoc. We believe however that more general strategies can be applied, especially in this case, where the constraints are L/U–like (i.e. can be written in the form $e \prec 0$ such that every p occurring negatively in e is a lower bound parameter and every p occurring positively in e is an upper bound parameter).

11.5 Experiments

11.5.1 A Prototype Extension of Uppaal

In this section, we report on the results of experimenting with a prototype extension of Uppaal described in the previous sections.

Our prototype allows the user to give some initial constraints on the parameters. This is particularly useful when explorations cannot be finished due to lack of memory or time resources, or because a non-converging series of constraint sets is being generated. Often, partial results can be derived by observing the constraint sets that are generated during the exploration. Based on partial results, the actual solution constraints can be established in many cases. These partial results can then be checked by using an initial set of constraints. Always, for each parameter p the constraint $p \geq 0$ is added as initial constraint.

11.5.2 The Root Contention Protocol

The root contention protocol is part of a leader election protocol in the physical layer of the IEEE 1394 standard (FireWire/i-Link), which is used to break symmetry between two nodes contending to be the root of a tree, spanned in the network topology. The protocol consists of first drawing a random number (0 or 1), then waiting for some time according to the result drawn, followed by the sending of a message to the contending neighbor. This is repeated by both nodes until one of them receives a message before sending one, at which point the root is appointed.

We use the Uppaal models of [140, 136], turn the constants used into parameters, and experiment with our prototype implementation (see Fig. 11.7 for results[3]). In both models, there are five constants, all of which are parameters in our experiments. The *delay* constant indicates the maximum delay of signals sent between the two contending nodes. The *rc_fast_min* and *rc_fast_max* constants give the lower and upper bound to the waiting time of a node that has drawn 1. Similarly, the *rc_slow_min* and *rc_slow_max* constants give the bounds when 0 has been drawn. It is reasonable to assume that initially, the constraints $rc\_fast\_min \leq rc\_fast\_max \leq rc\_slow\_min \leq rc\_slow\_max$ hold.

We have checked for safety with the following property:

$$\forall\Box \,.\, (\neg(\mathrm{Node}_1.\mathrm{root} \wedge \mathrm{Node}_2.\mathrm{root}) \wedge \neg(\mathrm{Node}_1.\mathrm{child} \wedge \mathrm{Node}_2.\mathrm{child}))$$

Safety for [140] It is shown in [140], that the safety property holds, if the parameters obey the following relation: $delay < rc\_fast\_min$. We have checked that the error states, expressed in the safety property, are indeed unreachable when this parameter constraint is met. We have also checked whether error states are reachable when we assume the constraint $delay = rc\_fast\_min$. This turns out not to be the case. In fact, it is argued in Remark 2 in [140], that the mentioned constraint is not *needed* for the correctness of the protocol. Rather than checking this on the parameterized model without any initial constraints, which is a large task, we experiment with a non-parametric version of the model *without any timing constraints*. It turns out that this model satisfies the safety property, hence we deduce that the parametric model, in which guards and

[3]All experiments were performed on a 366 MHz Celeron, except the liveness property which was performed in a 333 MHz SPARC Ultra Enterprise.

invariants have been added, satisfies the safety property for *any* valuation of the parameters.

Safety for [136] A different model of the root contention protocol is proposed in [136], in which it is shown that the relation between the parameters for the safety property to hold, should obey: $2{*}\mathit{delay} < \mathit{rc\_fast\_min}$. In fact, the model satisfies the safety property already when $\mathit{delay} < \mathit{rc\_fast\_min}$, but the stronger constraint is needed for proper behavior of the connecting wires. The necessity and sufficiency of these constraints is shown in [136] by applying standard Uppaal to several valuations for the parameters, and presented as an experimental result.

We have checked that the error states, expressed in the safety property, are indeed unreachable when either of these parameter constraints are met. We have also checked whether error states are reachable when we assume the constraint $\mathit{delay} = \mathit{rc\_fast\_min}$, which turns out to be the case as well. In fact, the union of the constraint sets of reachable states reported, can be rewritten to the constraint $\mathit{delay} = \mathit{rc\_fast\_min}$. As a double-check, we have ascertained for some parameter valuations, satisfying $\mathit{delay} = \mathit{rc\_fast\_min}$, that standard Uppaal also reaches an error state.

Since the model used for safety is a L/U automaton, we can experiment with Proposition 11.3, as follows. We show that our invariant property is satisfied by a more general model of root contention, and deduce with part 2 of Proposition 11.3 that it holds for the constraints we are after. We first identify the sets $L = \{\mathit{rc\_fast\_min}, \mathit{rc\_slow\_min}\}$ and $U = \{\mathit{delay}, \mathit{rc\_fast\_max}, \mathit{rc\_slow\_max}\}$. We substitute infinity for both *rc_fast_max* and *rc_slow_max*, *rc_fast_min* for *rc_slow_min*. The new model, together with either initial constraint $\mathit{delay} < \mathit{rc\_fast\_min}$, or $2{*}\mathit{delay} < \mathit{rc\_fast\_min}$, satisfies the invariant property. This allows us to conclude that the original model satisfies the invariant property for any valuation of the parameters where $\mathit{rc\_fast\_min} \leq \mathit{rc\_slow\_min}$, and the given initial constraint are satisfied. This includes the special case $\mathit{rc\_fast\_min} \leq \mathit{rc\_fast\_max} \leq \mathit{rc\_slow\_min} \leq \mathit{rc\_slow\_max}$.

We can do even better by applying Proposition 11.6, if we first change each guards or invariants for *delay* to a *strict* version, and then substitute infinity for both *rc_fast_max* and *rc_slow_max*, and *rc_fast_min* for both *delay* and *rc_slow_min*. Now we have a model with only one parameter and no constants, which we can verify non-parametrically with standard Uppaal, for two valuations of the parameter *rc_fast_min*, namely 0 and a non-zero value. The invariant property is satisfied, hence, by Proposition 11.5, we can deduce that it holds for all valuations of *rc_fast_min*, hence the original model satisfies the invariant property for any valuation of the parameters where $\mathit{rc\_fast\_min} \leq \mathit{rc\_slow\_min}$, and $\mathit{delay} < \mathit{rc\_fast\_min}$. Likewise, we can substitute $\mathit{rc\_fast\_min}/2$ for *delay*, and derive the other constraint. As can be seen in Fig. 11.7, the speed-up in terms of memory and time is drastic.

Finally, we can combine the results for initial constraints $\mathit{delay} < \mathit{rc\_fast\_min}$ and $\mathit{delay} = \mathit{rc\_fast\_min}$ with the fact that our model is a L/U automaton,

model	*initial constr.?*	*reduced?*	*property*	UPPAAL	*time*	*memory*
[140]	yes	no	safety	param	2.9 h	185 Mb
[140]	yes	completely	safety	std	1 s	800 Kb
[136]	yes	no	safety	param	1.6 m	36 Mb
[136]	yes	partly	safety	param	11 s	13 Mb
[136]	yes	completely	safety	std	1 s	800 Kb
[136]	yes	no	liveness	param	2.6 h	308 Mb

Figure 11.7: Experimental results for the root contention protocol

and derive the *necessity* of constraint *delay* < *rc_fast_min*, as follows. Suppose that a parameter valuation for *delay* and *rc_fast_min* exists, such that (1) the safety property holds, but (2) the constraint *delay* < *rc_fast_min* is not satisfied. Assume this valuation assigns d to *delay* and r to *rc_fast_min*. By our results, we know that $d \neq r$, so $d > r$. We now apply Proposition 11.3, and deduce that for each parameter valuation that assigns a value to upper bound parameter *delay* which is *smaller* than d, and a value to lower bound parameter *rc_fast_min* which is *larger* than r, the safety property must hold. This includes valuations that satisfy constraint *delay* = *rc_fast_min*, which contradicts our results. We conclude that only for parameter valuations that satisfy constraint *delay* < *rc_fast_min*, the safety property holds.

Liveness for [136] In [136], it is also shown that a refinement relation between the model of the most detailed level, and a model which is a bit more abstract, holds when the following relations are obeyed: 2∗*delay* < *rc_fast_min*, and 2∗*delay* < *rc_slow_min* - *rc_fast_max*. The refinement relation is such that it preserves both safety and liveness properties for the root contention protocol (which is proved in [136]). Again, the necessity and suffiency of the constraints is shown by experimenting with standard UPPAAL for several valuations for the parameters, and presented as an experimental result.

We have checked for a completely parameterized version of the system with the detailed model and the test automaton of the more abstract model, that error states in the test automaton are unreachable, that is, that the refinement relation holds. We have also checked, whether error states are reachable, that is, that the refinement relation does not hold, in the following two cases: either 2∗*delay* = *rc_fast_min*, and 2∗*delay* < *rc_slow_min* - *rc_fast_max*, or 2∗*delay* < *rc_fast_min*, and 2∗*delay* = *rc_slow_min* - *rc_fast_max*. This turns out to be the case as well. In fact, in both cases, the union of the constraint sets of reachable states reported, can be rewritten to these initial constraints. Again, this has been double checked by feeding parameter valuations that satisfy either of the above constraint sets to standard UPPAAL, which comes up with error states as well. Since the models for liveness use constraints that fall outside the scope of L/U automata, we cannot apply Proposition 11.6 here.

model from	*initial constraints*	*property*	UPPAAL	*time*	*memory*
[57]	yes	safety1	param	1.3 m	34 Mb
[57]	no	safety2	param	11 m	180 Mb
[57]	yes	safety2	param	3.5 m	64 Mb

Figure 11.8: Experimental results for the bounded retransmission protocol

11.5.3 The Bounded Retransmission Protocol

This protocol was designed by Philips for communication between remote controls and audio/video/TV equipment. It is a slight alteration of the well-known alternating bit protocol, to which timing requirements and a bound on the retry mechanism have been added. In [57] constraints for the correctness of the protocol are derived by hand, and some instances are checked using UPPAAL. Based on the models in [57], an automatic parametric analysis is performed in [20], however, no further results are given.

For our analysis we have also used the timed automata models from [57]. In [57] three different constraints are presented based on three properties which are needed to satisfy the safety specification of the protocol. We are only able to check two of these since one of the properties contain a parameter which our prototype version of UPPAAL is not able to handle yet.

One of the constraints derived in [57] is that $\mathrm{TR} \geq 2 \cdot \mathrm{MAX} \cdot \mathrm{T}_1 + 3 \cdot \mathrm{TD}$, where TR is the timeout of the receiver, T_1 is the timeout of the sender, MAX is the number of resends made by the sender, and TD is the delay of the channel. This constraint is needed to ensure that the receiver does not time out prematurely before the sender has decided to abort transmission. The sender has a parameter SYNC which decides for how long the sender waits until it expects that the receiver has realized a send error and reacted to it. In our parametric analysis we used TR and SYNC as parameters and instantiated the others to fixed values. Using our prototype we did derive the expected constraint $\mathrm{TR} \geq 2 \cdot \mathrm{MAX} \cdot \mathrm{T}_1 + 3 \cdot \mathrm{TD}$, however, we also derived the additional constraint $\mathrm{TR} - 2 \leq \mathrm{SYNC}$ which was not stated in [57] for this property. The necessity of this constraint was verified by trying models with different fixed values for the parameters.

The full set of constraints derived in [57] includes a constraint $\mathrm{TR} \geq \mathrm{SYNC}$ which is based on the property we cannot check. Therefore the error we have encountered is only present in an intermediate result, the complete set of constraints derived is correct. The authors of [57] have acknowledged the error and provided an adjusted model of the protocol, for which the additional constraint is not necessary.

The last constraint derived in [57] arises from checking that the sender and receiver are not sending messages too fast for the channel to handle. In this model we treat T_1 as a parameter and derive the constraint $\mathrm{T}_1 > 2 \cdot \mathrm{TD}$ which is the same as is derived in [57].

11.5.4 Other Experiments

We have experimented with parameterized versions of models included in the standard Uppaal distribution, namely Fischer's mutual exclusion protocol, a train gate controller, and a car gear box controller.

In the case of Fischer's protocol (which is the version of the standard Uppaal distribution, and not the one discussed in the rest of this paper), we parameterized a model with two processes, by turning the bound on the period the processes wait, before entering the critical section, into a parameter. We were able to generate the constraints that ensure the mutual exclusion within 2 seconds of CPU time on a 266 MHz Pentium MMX. Using these constraints as initial constraints and checking that now indeed the mutual exclusion is guaranteed, is done even faster. Fischer's protocol with two processes was also checked in [20], which took about 3 minutes.

Part IV

Development Methods

Chapter 12

Automatic Modeling a Language for Embedded Systems

The paper *Modeling a Language for Embedded Systems in Timed Automata* presented in this chapter has been published in part as a technical report [79] and a conference paper [78].

[78] T. Hune. Modeling a real-time language. In *Proceedings of Workshop on Formal Methods for Industrial Critical Systems, FMICS'99*, pages 259–282, 1999.

[79] T. Hune. Modeling a Language for Embedded Systems in Timed Automata. Technical Report RS-00-17, BRICS, August 2000.

The technical report extends the conference paper with a section discussing how to prove the correctness of the translation. Except for minor typographical changes the content of this chapter is equal to the technical report [79].

Modeling a Language for Embedded Systems in Timed Automata[†]

Thomas Hune[*]

Abstract

We present a compositional method for translating real-time programs into networks of timed automata. Programs are written in an assembly like real-time language and translated into models supported by the tool UPPAAL. We have implemented the translation and give an example of its application on a simple control program for a car. Some properties of the behavior of the control program are verified using the generated model.

12.1 Introduction

Reasoning about real-time systems can be very difficult and even more so if they consist of several concurrent processes. Tools for formal reasoning about such systems have been successfully developed [109, 74, 41] and applied in a number of cases (see [109, 73, 41] for lists of case studies). Before applying such tools one has to define an appropriate model of the system in question. This can in many case be a time consuming and error prone process. Methods and tools for defining such models based on an (informal) description of the system or parts of the system are an important help in the process of modeling.

One can divide models of embedded system into two groups. The first consisting of systems where the control program (if any) and the physical systems are mixed into one description (the water level monitor and the leaking gas burner, see e.g. [9], are examples of this). Systems in the second group have a clear distinction between the control program and the hardware/environment (some versions of the train gate controller, e.g. the one in [72], belong to this group). Here we will consider systems belonging to the second group. More precisely we will consider a method for modeling the control programs of such systems.

We have defined and implemented a translation from control programs written in the RCX™ language to networks of timed automata [108] used by UPPAAL

[*]Basic Research in Computer Science, **BRICS**, Centre of the Danish National Research Foundation. Department of Computer Science, University of Aarhus, Denmark, E-mail:{baris}@brics.dk

[†]This work is partially supported by the European Community Esprit-LTR Project 26270 VHS (Verification of Hybrid Systems)

[109]. Such a translation allows easier access to the verification power of a tool like Uppaal since the model of the program comes for free. The implementation has been tested on a number of programs and the properties of these programs have been verified by Uppaal.

The programs we are considering, are written in a language called the RCX™ language, which is an assembly like language with some high-level features. The RCX™ language runs on a processor in the LEGO® RCX™ brick which is part of LEGO® MINDSTORMS™ and LEGO® ROBOLAB™. The RCX™ brick is basically a (big) LEGO® brick with a computer inside. The brick has three input and three output ports, a speaker and an infrared sender and receiver for communication. Four different types of sensors are available for the RCX™ brick: touch, light, temperature, and rotation. Programming in the RCX™ language takes place on a PC where the programs are translated into byte code and downloaded to the RCX™ for execution.

In Section 2 the RCX™ language is described in more detail and an example of a control program for a car is given. Section 3 describes how RCX™ programs are executed especially with respect to scheduling. The translation is described in Section 4. The correctness of the translation is shortly discussed in Section 5 and and some aspects of the implementation in Section 6. In Section 7 the example is revisited and Section 8 contains a conclusion and ideas for future work.

12.2 The RCX™ Language

The language we are considering here is the RCX™ language running on the LEGO® RCX™ brick. It is a kind of assembly language but with some features from high-level languages. The language is mainly used as a target language for compilers form other languages like the ones in MINDSTORMS™ and ROBOLAB™. We have chosen to look at the RCX™ language for several reasons. First of all, it is a fairly simple language, but with most standard assembler operations. However, there is only one addressing mode making modeling a lot simpler. Programs consist of a fixed number of tasks running concurrently with a simple scheduling algorithm (see Section 12.3).

Even though the language is simple it can be used to write interesting control programs. Since the RCX™ is part of LEGO® one can build physical embedded systems for the control programs. This gives the opportunity of conducting experiments with complete embedded systems, and to study the relationship between the behavior of the complete embedded system and the formal model.

For these reasons we believe that the RCX™ language is suitable for a first try of defining an automatic translation from control programs to formal models.

12.2.1 Program Structure

An RCX™ program consists of a number of tasks. The number is restricted to a maximum of ten tasks, numbered 0 to 9. There are other restrictions imposed by the language, the most sever one being that one can only use 32 integer variables for data (it is not possible to address more). The body of a task is defined between `BeginOfTask(i)` and `EndOfTask()`. During execution a task is either blocked or enabled, initially only task 0 is enabled. A task can be started by the command `StartTask(i)` and blocked by `StopTask(i)`. Whenever `StartTask(i)` is executed, task i is restarted from the beginning independent of the state of the task, so there is always at most one copy of each task.

Next we will give a small example of a control program for a car, and following that, informally[1], present the part of the language we have considered. We present the instructions of the language in two groups, one containing instructions for control of flow and one for commands.

12.2.2 Example

As an example we will look at a simple control program for a car equipped with one touch sensor on each side of the front and one motor on each side driving one wheel. At the front and the back there is a ball instead of a wheel to make turning smoother. The car is turned by the motors running in different directions. Figure 12.1 is a sketch of the car. The control program consists of

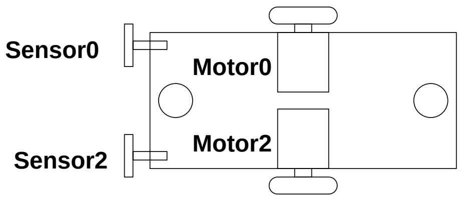


Figure 12.1: Sketch of the car.

three tasks. Task 0 first sets up the sensors and starts the car. After having started the car the task enters an infinite loop waiting for the reading from one of the sensors to change from zero to one. The change of a sensor reading from 0 to 1 will be considered an event which should be handled. Depending on which of the two sensors change, a task is started to handle the event.

```
BeginOfTask(0)
  SetFwd("02")          # Setup the output ports(motors)
  SetPower("02",2,4)    # to forward and power 4
  SetSensorType(0,1)    # Setup the input ports
```

[1]No formal semantics is available for the language. An informal description of the language can be found in [112]

```
    SetSensorMode(0,1,0)   # to touch sensor,
    SetSensorType(2,1)     # boolean mode
    SetSensorMode(2,1,0)
    SetVar(0,2,0)          # Var0:=0 (oldSensor0)
    SetVar(2,2,0)          # Var2:=0 (oldSensor2)
    On("02")               # Start the motors
    Loop(2,0)              # Begin the infinite loop
      SetVar(3,9,0)        # var3:=Sensor0
      If(0,3,2,2,1)        # If var3 = 1
        If(0,0,3,2,1)      # If var0 <> 1
          StartTask(1)     # Start task 1
          SetVar(0,0,3)    # Var0:=Var3
        EndIf()
      Else                 # Sensor0 was 0
        SetVar(0,0,3)      # Var0:=Var3
      EndIf()
      SetVar(4,9,2)        # Var4:=Sensor2
      If(0,4,2,2,1)        # If var4 = 1
        If(0,2,3,2,1)      # If var2 <> 1
          StartTask(2)     # Start task 2
          SetVar(2,0,4)    # Var2:=Var4
        EndIf()
      Else                 # Sensor2 was 0
        SetVar(2,0,4)      # Var2:=Var4
      EndIf()
    EndLoop()
  EndOfTask(0)
```

Task 1 and 2 are supposed to back the car a little and then turn away from the obstacle. The only difference between the tasks is the direction in which the car is turned, so only task 1 is shown.

```
  BeginOfTask(1)
    Off("02")       # Stop the motors
    SetRwd("02")    # Set the motors to go backwards
    On("02")        # Start the motors
    Wait(2,40)      # Wait while the car goes backwards
    SetFwd("0")     # Make the car turn
    Wait(2,30)      # Wait while it is turning
    SetFwd("2")     # Go forward again
  EndOfTask()
```

In line six of task 2 the argument for `SetFwd` is '`2`' and in line eight it is '`0`'.

12.2.3 Commands

The commands of the language can be divided into three categories. There are commands for manipulating variables, for setting up the sensors, and for controlling output.

The command for assignment is `SetVar(i,j,k)` where the number of the target variable (there are no symbolic names) is $i \in \{0, 1, \ldots, 31\}$, j is the type of the source of the assignment and k is the source. The most used sources (and the only ones we will consider), are variables (where k is the number of the variable), constants (where k is the value), sensor readings (where $k \in \{0, 1, 2\}$ is the number of the input port), and messages on the communication port[2] (where k does not have a meaning). All the commands for manipulating variables have this form, but the sensor reading and the message can only be used in assignment. The other possible manipulations are addition, subtraction, multiplication, division, bitwise conjunction and bitwise disjunction.

Two commands can be used for setting up the type of the sensors. To specify that input port i should be read as a sensor of type j the command `SetSensorType(i,j)` can be used, where j is one of the four types described previously. Using the command `SetSensorMode(i,j,k)` one can also set the type of readings from a sensor. Again i specifies which input port is set, j is the type of input, e.g. a raw ten bit integer value, a boolean or a percentage value. If boolean is chosen, k specifies how the value is calculated.

There are two types of output ports. Three ports for motors or lights (since we will concentrate on these, we will in the following call these output ports) and one for the speaker. The most basic commands for controlling the output ports are `On(li)` and `Off(li)` where *li* is a list of output ports. These simultaneously turn on respectively turn off the ports specified by *li*. The commands `SetFwd(li)`, `SetRwd(li)` and `AlterDir(li)` can be used for changing the 'direction' of the output of the ports specified by *li*. Finally, one can change the power of the output from the ports by `SetPower(li,j,k)` where *li* specifies the ports and j and k specifies the power in the same way as j and k specifies a value in the commands for manipulating variables. The value for the power can only be chosen in the range $0, 1, \ldots, 8$. For using the speaker port the two commands, `PlaySystemSound(i)`, and `PlayTone(i,j)` are available. The first plays one of six predefined sounds or beeps, and the second plays a tone with frequency i for duration j, given in units of ten milliseconds.

12.2.4 Flow Control

Two kinds of iterations exist in the language. `Loop(j,k)` loops the number of times specified by j and k, where j is the type of the source and k is the source. As source only variables and constants can be used. If j and k specifies the constant zero, the loop is infinite. The other possibility is `While(i,j,k,l,m)` where k specifies a comparison operator from the set $\{<, >, =, \neq\}$, and *i,j* and *l,m* specifies the values to be compared. Here variables, constants and sensor readings can be used.

An *if* statement with an optional else part, `If(i,j,k,l,m)` having the same type of arguments as the *while* statement is also available.

[2]This will always be the last message received.

Finally, one can block a task for a specified time using the `Wait(j,k)` command. Here j and k specifies the time for which the task will be blocked in ten milliseconds units. This is the only 'real-time' command in the sense that it refers directly to time.

12.3 Execution and Scheduling

The RCX™ is running a small operating system with processes for handling I/O and *one* process running an RCX™ interpreter. The process running the interpreter has the lowest priority. Since (almost) all the handling of I/O is periodic we assume that the interpreter gets a fixed portion of the CPU time.

Initially all tasks but task 0 are blocked. Each enabled task executes *one* instruction and then leaves control to the next task in a round robin fashion. One could imagine a number of other scheduling policies for RCX™ and experimenting with this would be interesting.

A task context switch takes place between interpretation of instructions. We have made some experiments measuring the timing of the execution of programs on the RCX™. Based on these we have concluded that the number of context switches does not depend on the number of tasks in a program. In a program with only one task, an instruction is interpreted approximately every 2 milliseconds.

The scheduling policy is very important when reasoning about the behavior of programs. If we want to guarantee a given response time of some action like pressing a touch sensor, we must first of all know how often the control program reads the input from the sensor[3]. As one can imagine, the response time will depend on the number of enabled tasks making it difficult to give precise bounds, though upper bounds can be given. We will return to this question in Section 12.7.

12.4 Modeling

The type of model we are using to model programs is networks of timed automata extended with integer variables which is supported by the tool UPPAAL. The type of model UPPAAL supports is a real-time model which we find appropriate for our purpose. The only real-time feature in the language (the wait command) could be handled by introducing a kind of tick, but we find it more natural to introduce a clock variable to model time. More importantly, we aim at modeling more than control programs. We hope to use our models of control programs together a with a model of the environment they are controlling, and time will be needed for describing this. We might need more general constructions than clock variables to get a satisfactory model of the environment.

[3]This of course also depends on how often the underlying operating system polls the input ports but since we do not know this, we assume this is done 'often enough'.

However, the tools we know of supporting more general models does not seem to be mature enough yet.

A network of automata is a collection of automata running in parallel and communicating by handshake. Channels are either internal, input, or output. Internal channels do not have a name, while the other channels do. The names of input and output channels are suffixed with '?' or '!' respectively. Communication takes place between one output channel and one input channel. Internal channels do not synchronize. Figure 12.2 is an example of a network of two automata. Here the right automaton can communicate with the left on

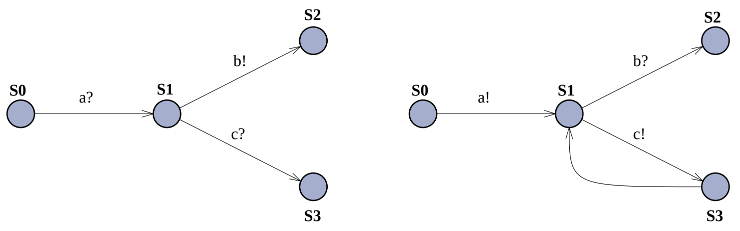


Figure 12.2: A network of two automata

the a channel. After communicating both automata are in location S1 where the right automaton can send on channel c and the left on channel b. If a communication on the b channel occurs nothing more can happen, but if the c channel is used for communication the right automaton has the possibility to move back to location S1, using the internal channel. Here it can input on the b channel and output on the c channel, but in this system, there is no automaton to communicate with so the system is deadlocked.

Time is added in the standard way [12, 13] by introducing a number of real valued clock variables. In Fig. 12.3 we have added clock variables x and y to the automata. Initially both clock variables have value zero and they progress

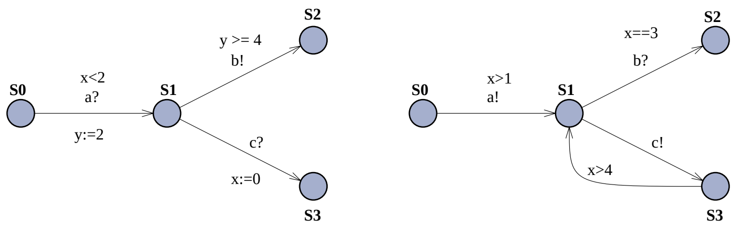


Figure 12.3: A network of two timed automata

at the same speed during the execution. The a channel is enabled in the left automaton until two time units have passed and in the right after one time unit has passed from the beginning of the execution. After the communication has taken place the value of the clock variable y is set to two. This should give an idea about how one can specify, when a channel is enabled by adding guards to the channel and how one can reset the value of clock variables. Only integer values are allowed in guards and resettings.

In Uppaal one also has integer variables which do not change with time but can be part of guards and assignments. On the right hand side of assignments, expressions of the form $E ::= Var \mid Num \mid E \text{ OP } E$ where OP $\in \{+, -, *, /\}$ can be used. We will make heavy use of this, when modeling operations on variables in the language.

12.4.1 Structure of Model

In our model of a program we will make one automaton for each task. All channel names are suffixed with the name of the task (we will assume that all instructions are from task 0 in the following figures if nothing else is stated). This makes it possible for the scheduler, also represented by one automaton, to control which task is allowed to execute. All information concerning time spent on interpreting instructions is handled by the scheduler.

Before we look at how single instructions are modeled, we have to look at the modeling of I/O. Since we are modeling a program and not a complete system our knowledge about the surroundings is very limited. The program gets readings from sensors after A/D conversion and some preprocessing but it cannot relate these readings to the true values in the environment[4]. Therefore the reading from a sensor is represented by an integer variable, named *Sensori*, where *i* is the number of the sensor. This is all the program has knowledge of. For output we do something similar. The status of each output port is modeled by three integer variables *Motor_Pow*, *Motor_Dir* and *Motor_On* representing the power, the direction and whether the port is turned on or not, respectively. The effect this has on what is connected to the port is not modeled. For the port to the speaker there is also three integer variables *SystemSound*, *Frequency*, and *Duration*.

In the following subsections we will describe the transformation, which is compositional, and hopefully it should be clear that implementing this has been straightforward. In all the figures the location named S0 is either the last location of the model of the previous instruction or the initial location of the automaton, if the instruction is the first of the task.

12.4.2 Commands

All the commands are modeled in a similar way by one channel and one new location. The name of the channel is the name of the command. The commands for setting up the sensors do not update any integer variables since the type and mode of the sensors are not used in the models we have defined so far.

[4]Such readings might not make sense at all. If the programmer has specified that sensor 1 is a touch sensor and someone plugs a light sensor to port 1, how should the program relate such readings to the real values?

If needed, this may be included later. For a command like `On(''02'')` the variables *Motor_On0* and *Motor_On2* are set to 1 as shown in Fig. 12.4. The

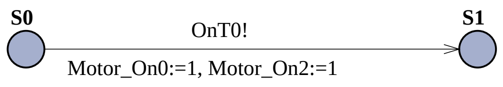


Figure 12.4: The model for the single command `On(''02'')`.

other commands for controlling the output ports are modeled similarly. Also the sound commands for the speakers are handled in this way.

The commands handling variables are modeled in almost the same way though here the type of the argument must be determined. For a command like `SetVar(23,2,0)` we should assign the constant 0 to variable number 23 (remember that the second argument specifies the type of the third argument, 2 means a constant). The command `SumVar(23,0,21)` for adding variable number 21 to variable number 23 will have the assignment $Var23 := Var23{+}Var21$.

If we look at the sequence of commands like the first three commands from task 1 of the example we get the automaton in Fig. 12.5. There is one channel for

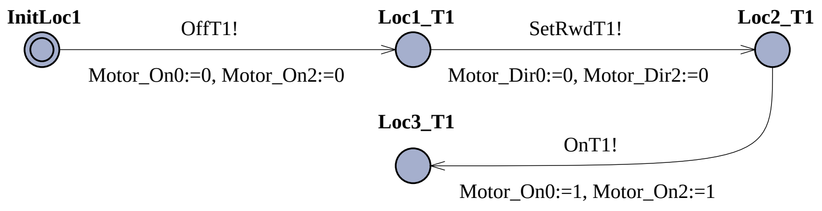

InitLoc1
OffT1!
Motor_On0:=0, Motor_On2:=0
Loc1_T1
SetRwdT1!
Motor_Dir0:=0, Motor_Dir2:=0
Loc2_T1
Loc3_T1
OnT1!
Motor_On0:=1, Motor_On2:=1

Figure 12.5: The model of the first three commands of task 1.

each command labelled with the name of the command. Each channel updates the value of the variables specified by the arguments of the command.

12.4.3 Flow Control

When translating the instructions for flow control we need more that one new channel and one new location. In the model of some of these instructions there must be room for connecting models of the inner parts of the instruction (like the body of a loop). For modeling a *loop* statement like

```
Loop(2,5)
    .
    .
EndLoop()
```

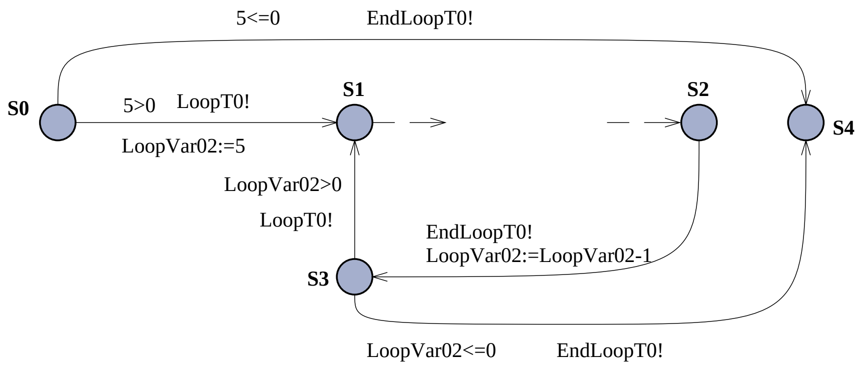


Figure 12.6: The model for a *loop* statement.

where .. is the body of the loop, we use four new locations. Figure 12.6 shows the model of such a loop statement. In the example we have a loop always running five times but the number of times can also be specified by a variable. In the model this would mean changing the constant 5 to the specified variable. The locations S1 and S2 are the initial and final location of the model of the body respectively. From the location S0 we have two channels, one for entering the loop and one for leaving it. Only one of these is enabled at a time. If the loop is entered we assign the number of times the loop must be executed to a new loop variable. Every time the end of the body is reached, the loop variable is decremented and location S3 is entered. From S3 there is a channel leading to the beginning of the body and one to S4. Again only one of the channels is enabled based on whether the loop is finished or not. When the loop is finished location S4 is entered. Decrementing the loop variable and testing it could of course have been done in one step but in the implementation of the interpreter this is interpreted as two instructions. This means that the task needs to be scheduled twice to restart a loop.

The model of a *while* statement is very similar to that of a loop though there is no need for a new variable. An example of a *while* statement is

```
While(0,3,2,2,0)
    .
    .
EndWhile()
```

where the arguments means `While Var3==0`. Again .. is the body of the while. In Fig. 12.7 the model of the *while* statement can be seen. As before the location S1 is the initial location of the body and S2 is the final location. From the location S0 we can move to the initial location of the body if the condition of the while is satisfied and otherwise the channel to the location S4 is enabled. Only one of these channels is enabled. From the final location of the body there is a channel to the S0 location where the condition is tested again. As for the

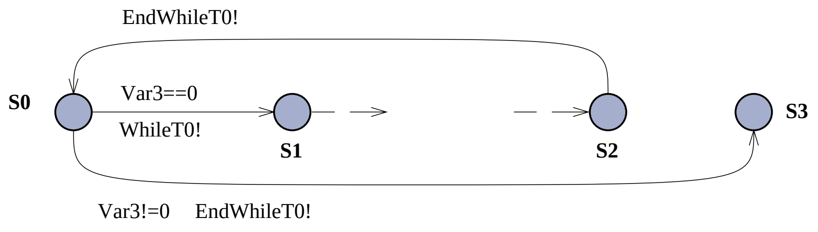


Figure 12.7: The model for a *while* statement.

loop this could have been done in one step (one channel) but the interpreter uses two steps.

An *if* statement (without an else) like

```
If(0,3,2,2,0)
    .
    .
EndIf()
```

is modeled as in Fig. 12.8. Again the initial location of the body is S1 and the

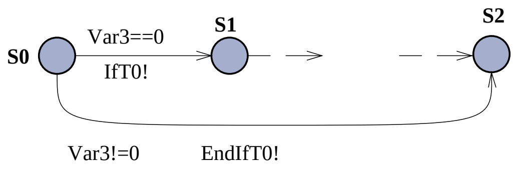


Figure 12.8: The model for an *if* statement.

final location of the body is S2. From the S0 location a channel to S1 is enabled if the condition is satisfied. If this is not the case, a channel to location S2 is enabled.

The model of an *if-else* statement follows the same idea, though now there are two parts or bodies. A model of the *if-else* statement

```
If(0,3,2,2,0)
    .
    .
Else()
    .
    .
EndIf()
```

is shown in Fig. 12.9. As in the model of the *if* statement there are two channels from the S0 location. One with label *IfT0* which is enabled if the condition is satisfied and one labelled *ElseT0* which is enabled otherwise. The initial

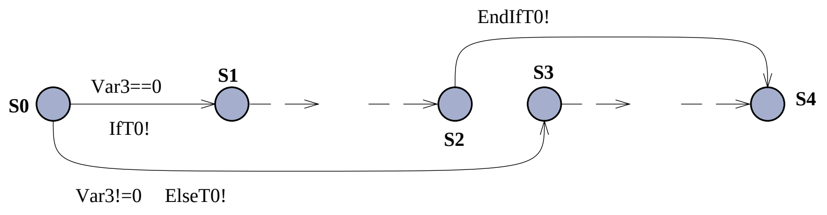


Figure 12.9: The model for an *if-else* statement.

location of the model of the *if* part (the first . .) is S1 and the final location is S2. From here there is an *EndIf* channel to S4 which is the final location of the model for an *if-else* statement. The initial location of the model of the *else* part (the second set of . .) is S3 and the final location is S4.

The *wait* statement is a little different from the others. This is the only instruction in the language referring directly to time and also the only part of the model of a task referring to time (represented by a clock). The model of a wait instruction, see Fig. 12.10, consists of a channel from the S0 location to location S1 where a clock variable *xT0* is reset to zero, and a channel labelled *SkipT0* from S1 to itself. The *SkipT0* channel is only used to synchronize with the

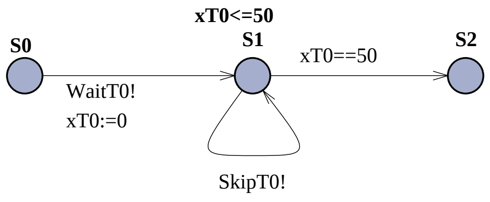


Figure 12.10: The model for a *wait* statement.

scheduler without any time passing. There is also a channel without label from S1 to S2. With this construction the scheduler does not need to keep a list of tasks blocked by wait. The location S1 has an invariant forcing the automaton to leave the location when the task is no longer blocked by the wait. When the task is no longer blocked the channel to location S2 is enabled. The channel from S1 to S2 is not enabled when the task is blocked.

12.4.4 The Scheduler

Applying the translation described so far we get an automaton for each task modeling the execution of that task. To get a model of the execution of the complete program we must combine the executions of the individual automata according to the scheduling policy described in Section 12.3. We define one automaton controlling the execution of the other automata (by synchronizing with these) implementing the scheduling policy on the RCX.

When the `StartTask(i)` command is executed task number i is restarted. Therefore we need a way of getting to the initial location of a task from all the other locations in the task. For this purpose we add a channel labelled *RSTi* from all the locations (including the initial location) to the initial location. The scheduler also needs to realize when a task has finished its execution, and hence we add a channel from the last location to itself with label *FinTi*.

As mentioned the scheduler lets each task which is not blocked execute one instruction in round robin. A task can be blocked if it has not been started (initially only task 0 is started), if it has finished, because of a *StopTask* statement, or because of a wait statement. In the first three cases our model of the scheduler will skip the task but in the case of the wait, the *Skip* channel with no delay will be used for communication. This means that the scheduler does not need to manage a list of tasks blocked by wait.

Our timing experiments suggest that the time spent on interpreting the different commands is almost the same for all commands. Since the time is almost independent on the parameters for the command we will not take this into account when modeling. The time measured for interpreting an instruction is less that 0.2 milliseconds which is less than the time spend on the context switch. In the model we have chosen milliseconds as our basic time unit. Therefore we will not model the time spent on interpreting each command but say that interpreting one command from each task in the program takes one millisecond, no matter how many tasks are enabled. The overhead of instructions for all the tasks is two milliseconds, so interpreting one instruction for all the tasks takes three milliseconds including context switches. This limits the precision of our model since we are using three milliseconds steps. So the best guarantees we can give using the model is within three milliseconds.

Figure 12.11 shows the structure of the scheduler for a program with three tasks, where only task 0 can start the other tasks. The initial location in the figure is the one with a ring inside. In Uppaal one can define a location to be *committed* which means that the automaton must leave the location before any other action takes place. Especially, this means that time cannot pass while an automaton is in a committed location. Committed locations are marked by a 'C' inside the location. The committed locations in the scheduler can be seen as a kind of control locations used for deciding who should be allowed to execute next.

For each task i there is an integer variable *Ti* taking values 0 or 1. If the task is enabled the value is 1, otherwise it is 0. The channel from the initial location to *ChooseT0* initializes these variables such that only task 0 is enabled. When the scheduler is in location *ChooseTi* the next task to execute according to the round robin schedule is task i. If this task is enabled the scheduler can move to location *RunTi* where it is possible to execute one instruction from the task. In case the task is blocked the scheduler can move to test the next task. Since all the *ChooseTi* locations (except *ChooseT0*) are committed this does not take any time.

In location *RunTi* the next instruction of task i can be executed. The channel

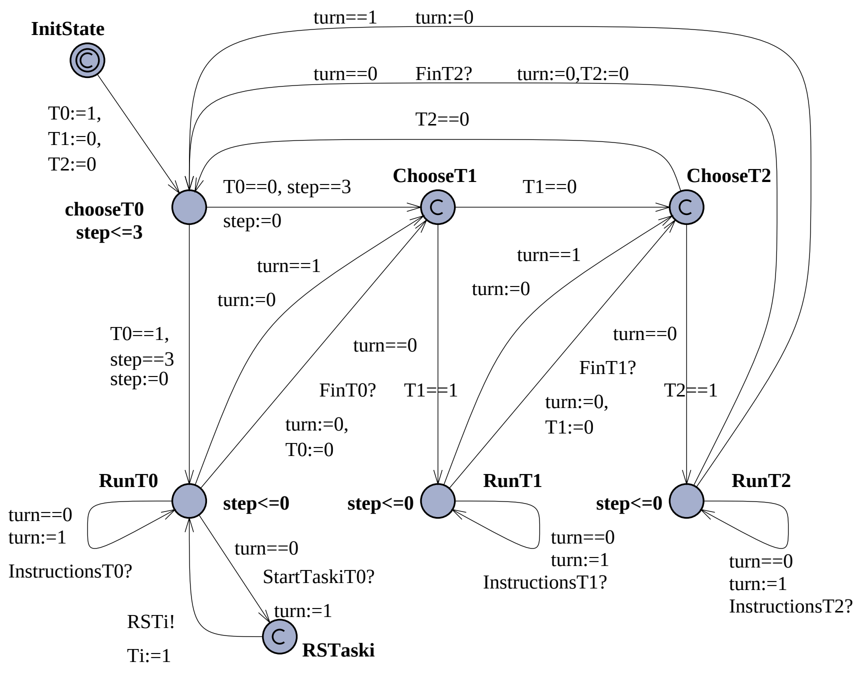


Figure 12.11: Model of a scheduler.

labelled *InstructionsTi* from *RunTi* to itself represents a number of channels, one for each kind of instruction in task *i*. All these channels have the same guard and assignment but different labels. The clock *step* is used to synchronize the execution such that each round through all the tasks takes three milliseconds. In location *ChooseT0* there is an invariant $step \leq 3$ and guards on the outgoing channels such that exactly three milliseconds must pass in this location. In the *RunTi* location there is an invariant $step \leq 0$ making sure that time does not pass in these locations. We have chosen not to make these locations committed because the environment should have the possibility of doing something. To make sure that a task only executes one instruction each time it gets the control, the integer variable *turn* is used. Depending on the value of *turn* it is possible to execute the next instruction of the task or leave control for the next task. Changing the model slightly would allow for different instructions to take a different amount of time. More instructions could be allowed by changing the guards involving *turn*.

The *StartTaskiT0* instructions is treated specially. This instruction executes as the other instructions but it must also restart task *i*. Therefore the *StartTaskiT0* channel ends in an intermediate location *RSTaski* which is committed. From this location there is a channel restarting task *i* and setting the variable *Ti* to one. After this control is back in the *RunT0* location, as if a normal instruction had been executed.

The scheduling policy on the RCX™ is very simple but with our approach one can model more complicated policies. As mentioned the number of instructions executed by each task could easily be changed. One could also define a time slice instead of counting instructions. Modeling a scheduler with fixed priorities is also possible. After a task finishes the scheduler should move control to the location allowing the task with the highest priority to start. If this task is enabled it will start, otherwise the other tasks should be checked according to their priority until one can be started.

12.4.5 I/O

Handling of the I/O is not part of the model. We have modeled each output port by three integer variables and the speaker port by three integer variables as well. How fast the motor goes or what sounds the speaker plays will not be part of our model, and the effects this might have on the environment and thereby the inputs, will not be modeled either. Note that this is important for modeling a complete system, but based only on the program, we cannot hope to do this automatically.

The input is not modeled either though generating a simple model of this letting the sensor reading behave arbitrarily would be easy. Based on the sensor mode one can define an automaton which at any time can assign any value in the range to the integer variable representing the sensor reading. In most cases this can be determined by a static analysis of the program. Programs can change the sensor mode dynamically making such an analysis more difficult to realize automatically.

We have defined these automata by hand for the experiments we have made. In the example there are two touch sensors which can give the reading zero or one. The simplest way of representing the behavior of these two sensors is shown in Fig. 12.12. This does not place any restrictions on the readings from

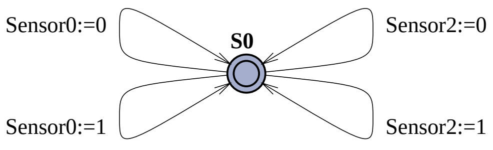


Figure 12.12: Model of simple environment for two touch sensors.

the sensors. If we had knowledge of how often the underlying operating system is polling the sensors, this would be a natural constraint to put on the channels.

12.5 Correctness

When addressing the correctness of the translation we must consider two different aspects. The relationship between the input of a program (values on the input ports) and the program variables and values written to the output ports.

In our case this also depends on the scheduling of the processes. The second aspect is the timing information of an execution and in the model.

No formal semantics of the RCX™ language is available but for most of the commands the semantics is clear from the informal description given in [112]. The few points which were not clear from the description has been clarified by some simple experiments with the language.

One could give a formal operational semantics to a task describing the output and changes of program variables with respect to old program variables, old output, and input. Assuming functions Var, In, and Out representing these environments, rules would have the form

$$(\mathsf{Var}, \mathsf{In}, \mathsf{Out}) \xrightarrow{\texttt{SetVar(3,0,5)}} (\mathsf{Var}[3 \mapsto \mathsf{Var}(5)], \mathsf{In}, \mathsf{Out}).$$

With rules like this for all the instructions of the language, one can for each instruction in the language prove that the translation described in Section 12.4 satisfies these rules. Given a rule for sequential composition it would be possible to prove correctness of the individual tasks with respect to the semantic given. Properties of a program containing more than one task must be based on semantic rules taking the scheduling into account. If we disregard the `Wait` instruction this could also be done without complicated modifications. If we also want rules for the `Wait` instruction some notion of time is needed in the semantic. Relating such a notion of time in the semantic to the notion of time in Uppaal would be much harder than relating values of variables.

Proving the translation of the individual tasks correct should therefore be simple but tedious. Proving the scheduling of tasks without the wait should also be possible though more challenging.

With respect to the timing of the execution it might be more appropriate to talk about accuracy than correctness. Should we talk about correctness we would need precise information about the operating systems and the how much time is spend on handling I/O. This might in the end depend on the input to the sensors. Also the precision of the clock would have to be taken into consideration. For these reasons we will talk about accuracy of the timing information. There are two immediate problems with the timing information in the model. First of all, the number of tasks enabled is not taken into consideration when calculating the time spend interpreting the commands. Secondly, time only passes in one location. In solving the second problem we would have to solve the first as well. To solve this the time unit would have to be changed, but this would enable models with better precision in general. With the current information we have on the timing of the execution of instructions, it does not make much sense to allow for specification with such a precision. Our assumption about all commands taking the same time might not be valid with such a fine grained measure of time. Much more precise timing information would be needed for models and results obtained from these models to be useful.

Therefore we have chosen to keep milliseconds as the basic time unit and have the three milliseconds intervals when modeling the execution. With this as basic time unit, we find that there is a good correspondence between the timing specified in the model of a program and the actual execution of the program. However, since there is a small inaccuracy this can be added up during long executions. One should of course be aware of this when modeling and proving properties.

12.6 Implementation

From the description in Section 12.4 it should be clear that the translation can be implemented. We have made an implementation in ML which translates a RCX™ program file to a file containing a textual description of a network of timed automata (called *xta* format). The *xta* format is the format UPPAAL uses for describing automata. There is no graphical information in the *xta* format but the newest version of the graphical interface to UPPAAL can read a file in *xta* format and display the corresponding network of timed automata.

The program works in two phases. First the program file is parsed and a data type for the program is built. This type looks as one would expect with a statement being one of the instructions described in Section 12.2 and the body of the control statements consisting of statements.

The second phase is a recursive descent of the data type for the program. Since our translation is compositional a statement can be translated only knowing the last location of the model of the previous statement.

Along the way through the data type one also needs to collect the names of channels, clock variables, integer variables and locations since these must be defined in the *xta* file.

We have successfully tested the translation program on a number of RCX™ programs.

12.7 Example Revisited

We have used the translator to get a model of the control program in Section 12.2.2. Figure 12.13 shows the automaton for task 0. The loop testing the input from the sensors begins at location *Loc9_T0*, the channels before this location models the initialization. The location *Loc23_T0* is the final location which is not reachable since the loop is infinite. There are no restart channels in this model (except the one from the initial location) because this task is never restarted.

The models of task 1 and 2 are very similar so we will only show the model of task 1 (Fig. 12.14). If one abstracts away from the restart channels (the channels labelled *RST1?*) it should be easy to follow the one path through the model. It should also be easy to see that this models the commands in the task.

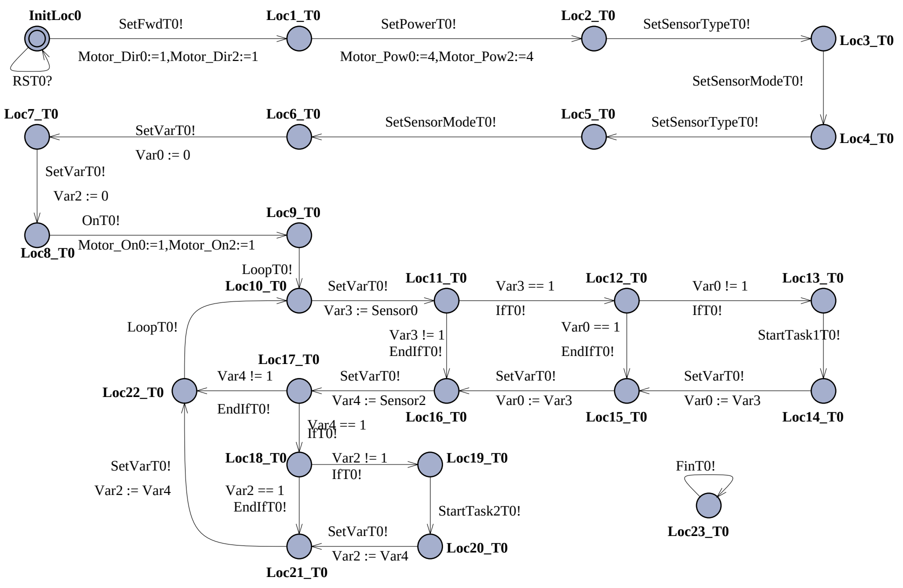


Figure 12.13: Model of task 0.

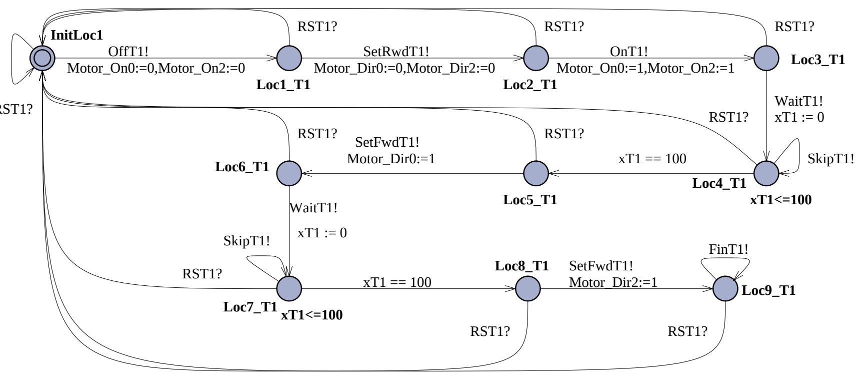


Figure 12.14: Model of task 1.

We will not show the model of the scheduler since this looks very much like the model in Fig. 12.11. The only difference being that the *InstructionsTi* channel has been replaced with a number of channels - one for each type of instruction in the task.

If we model the input from the sensors by the automaton in Fig. 12.12 and use this together with the automata for the tasks and the scheduler we have a model of the complete system. We cannot use this model to reason directly about the movements of the car. What we can do, is reason about how the output ports react to input from the sensors. With this model we can first of all simulate the behavior of the program. Given the very liberal model of the environment we have defined, we can test how our program reacts under all possible sequences of input.

We can also answer some very basic questions about the program such as whether it is possible for the output ports to be turned on or whether it is possible for the output ports to be in reverse. This is done by checking the formulas

$$E <> (Motor\_On0 == 1 \textit{ and } Motor\_On2 == 1)$$

and

$$\begin{aligned} E <> \quad & (Motor\_On0 == 1 \textit{ and } Motor\_On2 == 1 \textit{ and} \\ & Motor\_Dir0 == 0 \textit{ and } Motor\_Dir2 == 0) \end{aligned}$$

respectively. Both properties are satisfied and we get a trace leading to a satisfying state.

We can also try to find out how fast the program will respond when an input is read. This can not be done directly by writing one formula in Uppaal. Instead we have to make what is called a test automaton. This is an automaton which only monitors the behavior of the system. Figure 12.15 shows a test automaton for testing whether the response time from sensor 0 has been read with value one by task 0 and until task 1 responds is less than 16 milliseconds. Some auxiliary channels are added to the model to synchronize with the test

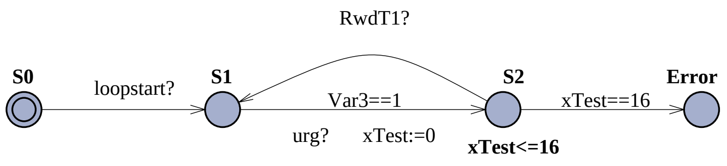


Figure 12.15: A test automaton.

automaton. Firstly, we will only consider the program after the initialization is finished therefore the first channel. The channel *urg* is only used to make the channel what is called an *urgent* channel. An urgent channel is a special kind of channel which, when enabled, must be executed without any time delay. This channel is enabled when the program reads that the sensor has been pressed. If the output ports are set in reverse before 16 milliseconds have passed we enter the location *S2* again and otherwise the *Error* location is entered. All channels but the one to the error location are urgent. We can now ask whether the *Error* location is reachable. If this is not the case the response always arrives within 16 milliseconds. In this case the *Error* location is not reachable but if 16 is changed to 15 then the *Error* location is reachable. If we wanted to test the response time from the sensor was touched we need a more complex test automaton taking into account that the reading of the sensor in this model can be set to one and then zero before the program reads it.

We might also want to verify that the output ports are set in reverse direction and turned on for a given time when the sensor reading changes to one. For this we need another model of the environment. When one of the readings change from zero to one the output ports are set in reverse. While the output ports are in reverse the reading from the other sensor might also change from zero to one. As a response to this the ports would be stopped. In general we will have to define a more precise model of the sensor readings if we want to prove more involved properties about our program or the movement of the car itself.

12.8 Conclusion

We have presented a method for translating RCX™ programs to networks of timed automata in a format readable by Uppaal. Applying this translation gives the possibility of reasoning formally about the behavior of the program using Uppaal. The translation have been implemented and tested on a number of examples with success.

Even though the method described here is specific to the RCX™ language, we believe that the principles can be carried over to most other assembly like (real-time) languages. There is a number of things one should take into consideration before trying to do this. Modeling other addressing forms like indirect addressing will be a lot more involved though it can be done. A detailed knowledge of the execution of programs or a formal semantics is needed for the model to make sense. If one wants to prove strong timing bounds for programs, precise timing information of the instructions will be needed in the model.

Exploring how good a relationship we can get between the behavior defined by timed automata and the behavior of LEGO® systems will be interesting. We cannot model the behavior of the physical system completely but we hope to be able to model it in such a way, that it makes sense to relate a number of properties of the formal model to the real system. In doing this we will have to define more detailed models of the environment.

Acknowledgements We would like to thank Michael Andersen who is working at LEGO® for useful comments on the RCX™.

Chapter 13

Using Automata in Control Synthesis

The paper *Using Automata in Control Synthesis – a Case Study* presented in this chapter has been published in part as a technical report [89] and a conference paper [88].

[88] T. Hune, and A. Sandholm. A Case Study on using Automata in Control Synthesis. In *Proceedings of Fundamental Approaches to Software Engineering (FASE 2000)*, pages 349–362, 2000.

[89] T. Hune, and A. Sandholm. Using Automata in Control Synthesis – a Case Study. Technical Report RS-00-22, BRICS, September 2000.

The technical report extends the conference paper by adding more description of the logical formulae used. Some of the code implementing the control automata has also been included. Except for minor typographical changes the content of this chapter is equal to the technical report [89].

Using Automata in Control Synthesis – a Case Study

Thomas Hune[*] Anders Sandholm[*]

Abstract

We study a method for synthesizing control programs. The method merges an existing control program with a control automaton. For specifying the control automata we have used monadic second order logic over strings. Using the Mona tool, specifications are translated into automata. This yields a new control program restricting the behavior of the old control program such that the specifications are satisfied. The method is presented through a concrete example.

13.1 Introduction

This paper presents some practical experience on synthesizing programs for the LEGO® RCX™ system. The synthesis presented here is based partly on an existing simple program and partly on an automaton generated by the tool Mona [96][1].

Writing control programs can often be an error prone task, especially if a number of special cases must be taken into account. Often most of the time and effort is spent on taking care of special case or failure situations rather than solving the actual problem at hand. Different methods and tools have been developed to help in writing control programs. One well known method is based on a control automaton running in parallel with the actual program [131, 135]. The automaton controls the input and output events of the program. By doing this the sequences of I/O actions occurring is restricted.

The automata controlling the I/O actions can be specified in different ways, e.g. by specifying it directly in some suitable notation, or by a logical formula. We have chosen the latter approach. There are various logics which could be used as specification language. We have chosen to use monadic second order logic over strings (M2L) [45] for a number of reasons. First of all M2L has a number of nice properties such as being expressive and succinct. For instance, having second

[*]Basic Research in Computer Science, **BRICS**, Centre of the Danish National Research Foundation. Department of Computer Science, University of Aarhus, Denmark, E-mail:`{baris,sandholm}@brics.dk`

[1]`http://www.brics.dk/mona`

order quantification M2L is more expressive than LTL. Furthermore, there are succinct M2L-formulae of size n which have minimal corresponding automata of non-elementary size. Secondly, the tool Mona [96] implements a translation from M2L formulae to minimal deterministic automata (MDFA) accepting the language specified by the formula. The automata generated do not contain any acceptance condition for infinite executions so we will only be considering safety properties.

The method we study here is a variation of classical synthesis as described in e.g. [118, 131], in that the method is partly based on an existing control program. The aim of the synthesis described here is to restrict the behavior of an existing (hopefully very simple) control program such that it satisfies certain properties given by the user. The executions of the existing control program are restricted by the control automaton having I/O events as alphabet. These events define the interface between the existing control program and the specification.

For studying the method we will look at a control program for a moving crane. We have implemented the method for this example in the LEGO® RCX™ system [112]. Using the LEGO® RCX™ system is interesting for at least two reasons. First of all the environment of the RCX™ system and especially the programming language is quite restricted, so it is not obvious that implementing the method is feasible at all. Secondly, using the LEGO® RCX™ system one can build actual physical systems for testing the control programs. We have built the crane and used it with different control programs.

The language running on the LEGO® RCX™ brick (RCX™ language) is an assembly-like language with a few high level features, like a notion of task or process. Programs are written on a PC and downloaded to the RCX™ brick where they are interpreted.

13.1.1 Related Work

The use of finite state automata for controlling systems is not novel. Ramadge and Wonham [131] give a survey of classic results.

The method used in this paper has been used successfully in `<bigwig>` [135], a tool for specifying and generating interactive Web services. Our method for control synthesis is used as an integral part of `<bigwig>` to define safety constraints. In fact, via use of a powerful macro mechanism [42] the method has been used to extend the Web programming language in `<bigwig>` with concepts and primitives for concurrency control, such as, semaphores and monitors.

13.1.2 Outline of the Paper

In the following section we will outline the method. A short presentation of the LEGO® system is given in Section 13.3. In Section 13.4 the crane example is presented. The logic-based specification language is presented in Section 13.5, and the merge of automata with the RCX™ code in Section 13.6. Finally, Section 13.7 rounds off with conclusions, and suggestions for future work.

13.2 Outline of the Method

The two main components of the synthesis is a basic control program and an automaton. From these two components we generate a control program which is ready for use. We do not have any special requirements for what a control program is, such as no side effects, since in our case the control program is the only program running on the RCX™ brick. The interface between the two components is a predefined set of I/O actions. This will typically be all commands in the program for reading sensors or manipulating actuators.

Given a basic control program and an implementation of the automaton we merge these. Each instruction in the basic control program using one of the I/O actions is transformed to a sequence of instructions first calling the automaton and based on the response from the automaton performing the action or not. Section 13.6.3 will discuss different approaches to what should happen, when a given action is not allowed by the automaton.

Since the automaton is invoked only when the basic control program is about to make an I/O action, it can only restrict the possible I/O behaviors of the control program, not add I/O actions in new places. Only looking at sequences of I/O actions the basic control program must therefore be able to generate all sequences present in the solution. Since the automaton will prune away unwanted sequences, the basic control program might also generate unwanted sequences. The basic control program should not implement any kind of priority scheme, unless one is sure that combining this with the safety specification will not lead to deadlocks.

The hope is that writing such basic control programs should be a simple task. In general the basic control program could be one always being able to read any sensor and give any command to the actuators. This amounts to generating the star operation of the input alphabet. However, there will often be a correspondence between input and output which is naturally included in the basic control program. Often, adding details like these to the basic control program will make the specification of the automaton simpler. This is the case in the example shown later.

One could see the basic control program as implementing the method for controlling the sequences of I/O actions and the automaton defining the allowed policy for these. This suggests that with one implementation of a basic control program it is possible to test different specifications or strategies (policies) only by changing the control automaton. Therefore, a fast (automatic) way of getting an implementation of an automaton from a specification and merging this with the control program allows for testing different specifications fast.

13.3 The LEGO® System

The studies we have conducted are based on the LEGO® RCX™ system and the associated RCX™ language. The language is an assembly like language with some high level concepts like concurrent tasks. The language is restricted in a number of ways, e.g. it is possible to address only 32 integer variables and allows only ten tasks in a program. Furthermore, one cannot use symbolic names in programs. However, we have not encountered problems with the mentioned restrictions during our experiments.

A small operating system is running on the RCX™ with processes for handling I/O and one process running an interpreter for the RCX™ language. The RCX™ brick has three output ports (for motors and lights) and three input ports. Four kinds of sensors for the input ports are supplied by LEGO®: touch, temperature, rotation, and light.

13.3.1 The RCX™ Language

A program consists of a collection of at most ten tasks. There is no special way to communicate between tasks but all variables are shared, providing a way of communication. A task can start another task with the command `StartTask(i)` and stop it with the command `StopTask(i)`. Starting a task means restarting it from the beginning. That is, there is no command for resuming the execution of a task nor spawning an extra "instance" of a task.

The language has some commands for controlling the output ports, the main ones being `On(li)` and `Off(li)` where `li` is a list of ports. The commands `SetFwd(li)` and `SetRwd(li)` sets the direction of the ports in `li` to forward and reverse respectively. There are also a number of instructions for manipulating variables. All of these take three integer arguments. The first argument specifies the target variable, the second the type of the source, and the third the source. The most important types of sources are: variables (the third argument is then the number of the variable), constants (the third argument is then the value), and sensor readings (the third argument is then the number of the sensor). These types of sources can be used in the instruction `SetVar(i,j,k)` for assigning a value to a variable. In the instructions for calculating like `SumVar(i,j,k)`, `SubVar(i,j,k)`, and `MulVar(i,j,k)` sensor readings are not allowed.

Loops can be defined in two ways, either by the `Loop(j,k)` instruction or by the `While(j,k,l,m,n)` instruction. The arguments of the `Loop` indicates how many times the body should be iterated in the same way as the source of the instructions for calculating. The `While` loop is iterated as long as the condition specified by the arguments is satisfied. The first two and last two arguments specify the sources of a comparison as in an assignment and `l` specifies a relation from the set $\{=, <, >, \neq\}$.

There is also a conditional, `If(j,k,l,m,n)`, with the condition specified as in the `While` construct and an `Else` branch can be specified as well.

One can block a task for a given time using the `Wait(j,k)` statement. When the specified time has passed, execution of the task is resumed.

During execution a task is either enabled or blocked. A task can be blocked by a `StopTask(i)` instruction, by a `Wait(j,k)` instruction, or by finishing its execution (reaching the end of the code). Initially only task zero is enabled. The enabled tasks are executed in a round robin fashion, where each task executes one instruction and then leaves control for the next task.

The statements presented above constitute the part of the RCX™ language which we have used for implementing control automata.

13.4 Example

As an example we will look at a crane which we will program in the RCX™ language. We have built the crane and tested it with different control programs. The crane is run by three motors connected to the RCX™. One motor is driving the wheels, one is turning the turret around, and one is moving the hook up and down. The input for the three motors are three touch sensors, which is all the RCX™ brick has room for. This means we can only turn motors on and off. Therefore the crane alternates between moving forward and backward each time the motor is turned on. The direction of turret and the hook is controlled in a similar way.

A very basic control program for the crane consists of four tasks. One task for setting up the sensors and motors, and starting the other tasks. For each of the three inputs there is one task for monitoring input and controlling the motor correspondingly. Task 1 for monitoring sensor 0 is:

```
BeginOfTask 1
Loop 2, 0                    'An infinite loop
  SetFwd "0"                 'Set direction of motor 0 to forward
  SetVar 1, SENSOR, 0        'Var1 := Sensor0
  While VAR, 1, 3, CONST, 1 'While Var1 != 1
    SetVar 1, SENSOR, 0
  EndWhile
  On "0"                     'Start motor 0
  Wait CONST, 100            'Wait
  SetVar 1, SENSOR, 0        'Var1 := Sensor0
  While VAR, 1, 3, CONST, 1 'While Var1 != 1
    SetVar 1, SENSOR, 0
  EndWhile
  Off "0"                    'Stop motor 0
  Wait CONST, 100                'Wait
  ... repeat the code replacing SetFwd "0" with SetRwd "0" ...
EndLoop
EndOfTask
```

The `Wait` statements ensures that one touch of the sensor is not read as two touches. We could of course have tested for this but for our example this simple approach will do. The two other tasks for controlling the remaining two motors look similar, only the numbers of variables, sensors and motors are different.

For the purpose of illustrating the presented method we choose to place the following constraints on the behavior of the crane. First of all we only want one thing happening at a time, so we will not allow for two motors to be turned on at the same time. Pressing the touch sensor could now be seen as a request which the control program may grant (and start the motor) when all the motors are stopped. A motor can only be stopped by a request to stop that motor, not by requests to start other motors. Moreover, we want that moving the hook has higher priority than the wheels and the turret. Requests from the other two are handled in order of arrival. The first constraint on the behavior is basically mutual exclusion which is nontrivial to implement in the RCX™ language (this is an integrated part of the implementation of the automata-based approach described in Section 13.6). On top of this we have a mixed priority and queue scheme.

13.5 Logic-Based Specifications

Basically, we could keep the initial simple control program if we had a way of pruning out some unwanted executions. To be able to implement the constraints we have to change the initial control program slightly. This is done by considering touching a sensor as a request. The motor can be turned on or off when the request is accepted. Even with these changes the program is still simple to write. Execution of the program gives rise to a sequence of events. In our case we will consider input (requests), and two kinds of output (start and stop motor) as events. We then implement the automaton accepting the language over these events satisfying the introduced constraints. With this approach we can thus keep the control program simple.

Traditionally, control languages are described by automata which are in some cases a good formalism to work with. However, having experience in using logic for specifying properties, we will take that approach here. In this section we describe the use of a logic formalism from which we can automatically generate automata.

13.5.1 Terminology

An *automaton* is a structure $A = (Q, q^{\text{in}}, \Sigma, \rightarrow, F)$, where Q is a set of states with initial state $q^{\text{in}} \in Q$, Σ is a finite set of events, $\rightarrow \subseteq Q \times \Sigma \times Q$ is the transition relation, and $F \subseteq Q$ the set of acceptance states. We shall use $q_1 \overset{\sigma}{\rightarrow} q_2$ to denote $(q_1, \sigma, q_2) \in \rightarrow$. A sequence $w = \sigma_0\sigma_1 \ldots \sigma_{n-1} \in \Sigma^*$ is said to be *accepted* by the automaton A if there exists a run of A which reads the sequence w and ends up in an accepting state q. So we have $q_1, \ldots, q_{n-1} \in Q$ and $q \in F$, such that $q^{\text{in}} \overset{\sigma_0}{\rightarrow} q_1 \overset{\sigma_1}{\rightarrow} \ldots \overset{\sigma_{n-2}}{\rightarrow} q_{n-1} \overset{\sigma_{n-1}}{\rightarrow} q$. We shall denote by $L(A)$ the *language* recognized by an automaton, that is, $L(A) = \{\, w \in \Sigma^* \mid A \text{ accepts } w \,\}$.

In order to be able to define the notion of a legal control language, one partitions the event set Σ into *uncontrollable* and *controllable* events: $\Sigma = \Sigma_u \cup \Sigma_c$. The controllable events can be disabled by the control automaton at any time, whereas the uncontrollable ones are performed autonomously by the system without any possible interference by the control automaton. The automaton merely has to accept the fact that the particular uncontrollable event has occurred and maybe change its state. Thus a control language must in some sense, which is defined precisely below, respect the uncontrollableness of certain events. Furthermore, since our method only allows restrictions concerning safety properties, it does not make sense to have non-prefix-closed languages as control languages. That is, we define the notion of control language as follows.

Let *pre*(L) denote the prefix closure of a language L, and let *unc*(L) denote closure of L under concatenation of uncontrollable events. That is, let

$$\begin{aligned} \mathit{pre}(L) &= \{\, v \in \Sigma^* \mid \exists w \in \Sigma^* : vw \in L \,\} \text{ and} \\ \mathit{unc}(L) &= \{\, vw \in \Sigma^* \mid v \in L \wedge w \in \Sigma_u^* \,\}. \end{aligned}$$

A language, L over $\Sigma = \Sigma_u \cup \Sigma_c$ is called a *control language* if it satisfies the two properties $\mathit{pre}(L) = L$ and $\mathit{unc}(L) = L$.

When using deterministic finite state automata to specify sets of sequences, checking for prefix closedness is easy. One just has to make sure that all transitions from non-accepting states go to non-accepting states. Similarly, checking closure under concatenation of uncontrollable events is straightforward for deterministic automata.

What is new here, in comparison to the use of our method in [135], apart from the new domain of LEGO® RCX™ robots, is the partition into controllable and uncontrollable events and the resulting additional restrictions and computations.

13.5.2 Specification Logic

It would be nice if instead of converting the informal requirement in Section 13.4 into an automaton, one could write it formally in a specification formalism closer to natural language. That is, we would like to be able to write something like the following.

- Only one motor can be turned on at a time;
- If the wheels get turned on, then the hook must not be requesting and the wheels must have been the first to make a request; and
- If the turret gets turned on, then the hook must not be requesting and the wheels must have been the first to make a request.

We therefore turn to a formalism that is as expressive as finite state automata and yet still allows for separation of the declaratively specified requirements (previously our control automaton) and the operational part of the control program (the existing RCX™ program).

Experience has shown that logic is a suitable specification formalism for control languages. For the purpose of defining controllers for LEGO® RCX™ robots, we have chosen to use M2L. One might argue in favor of other specification formalisms such as high-level Petri Nets [94] or Message Sequence Charts [120]. Being a logic formalism, however, M2L has the advantage that specifications can be developed iteratively, that is, one can easily add, delete, and modify parts of a specification. It also has a readable textual format. Moreover, the formalism in use should be simple enough that a runtime checker, such as an automaton, can actually be calculated and downloaded to the RCX™ brick. Thus, M2L is powerful and yet just simple enough to actually subject it to automated computation.

Experience in using M2L as a language for defining control requirements has shown that only the first-order fraction of the logic is used in practice [42, 135]. We shall thus consider only first order quantifications, though second-order quantifications could be added at no extra cost.

The abstract syntax of the logic is given by the following grammar:

$$\begin{aligned}\phi &::= \exists p : \phi' \mid \forall p : \phi' \mid \neg\phi' \mid \phi' \wedge \phi'' \mid \phi' \vee \phi'' \mid \phi' \Rightarrow \phi'' \mid \sigma(t) \mid t < t' \\ t &::= p \mid t + 1\end{aligned}$$

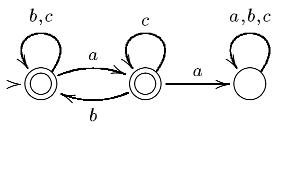
That is, M2L has the constructs: universal and existential quantifications over first order variables (ranging over positions in the sequence of events), standard boolean connectives such as negation, conjunction, disjunction, and implication, the basic formulae, $\sigma(t)$, to test whether an event σ can be found at position t, and $t < t'$, to test whether position t is before position t'. It also has operations on terms, such as, given a position t one can point out its successor $(t + 1)$, and simple term variables (p).

A formula ϕ in M2L over the event set Σ will – when interpreted over a finite sequence of events w – either evaluate to true or to false and we shall write this as $w \models \phi$ or $w \not\models \phi$, respectively. The *language* associated with ϕ is $L(\phi) = \{\, w \in \Sigma^* \mid w \models \phi \,\}$. The language associated with an M2L formula is guaranteed to be regular. In fact, it has been known since the sixties that M2L characterizes regularity [45, 62].

The Mona tool implements the constructive proof of the fact that for each M2L formula there is a minimal deterministic finite state automaton accepting the language of the formula. That is, Mona translates a particular M2L formulae, ϕ, into its corresponding minimal deterministic finite state automata (MDFA), A, such that $L(\phi) = L(A)$.

Example 13.1 *Let $\Sigma = \{a, b, c\}$. The M2L formula to the left*

$$\begin{aligned}&\forall p, p'' : (p < p'' \wedge a(p) \wedge a(p'')) \\ &\quad \Longrightarrow \exists p' : p < p' < p'' \wedge b(p')\end{aligned}$$



is true for sequences in which any two occurrences of a will have an occurrence of b in between. Using Mona*, one can compute the automaton corresponding to the formula above. The resulting automaton appears to the right.*

Example 13.2 *With this logic-based specification language in place, we can write a specification of the requirements given in the example. The logic-based specification looks quite complex at first. However, because of it's modular structure we find it easier to handle than the automaton. The basic formulae for the elements of the alphabet are* ***req1(t)****,* ***req2(t)****,* ***req3(t)****,* ***turnon1(t)****,* ***turnon2(t)****,* ***turnon3(t)****,* ***turnoff1(t)****,* ***turnoff2(t)****, and* ***turnoff1(t)****. The first three are uncontrollable events of the alphabet and the rest are controllable events of the alphabet. A predicate is true if the event at position* t *is the mentioned event. Using these basic formulae we can define some basic predicates like all motors are stopped by:*

$$\begin{aligned}\text{off1}(t) = {} & (\forall t' : t' < t \Rightarrow \neg\texttt{turnon1}(t'))\vee \\ & (\forall t' : (t' < t \wedge \texttt{turnon1}(t')) \Rightarrow \\ & \exists t'' : t' < t'' \wedge t'' < t \wedge \texttt{turnoff1}(t''))\end{aligned}$$

The predicate specifies that either there has never been a ***turnon1*** *action, or for every position where there is a* ***turnon1*** *action there is a* ***turnoff1*** *action at a later position. Similarly, we define predicates* ***off2(t)*** *and* ***off3(t)*** *and using these we can define a predicate,* ***alloff(t)****, specifying that all the motors are turned off.*

$$\text{alloff}(t) = \text{off1}(t) \wedge \text{off2}(t) \wedge \text{off3}(t)$$

We can specify that motor 1 has been requested to be turned on but has not yet been turned on by the following predicate:

$$\begin{aligned}\text{request1}(t) = {} & \exists t' : t' < t \wedge \texttt{req1}(t')\wedge \\ & (\forall t'' : (t' < t'' \wedge t'' < t) \Rightarrow \neg\texttt{turnon1}(t''))\end{aligned}$$

This is specified by stating that at some position there is a request ***req1*** *and at no later position is there a* ***turnon1*** *action, handling the request. Predicates* ***request2(t)*** *and* ***request3(t)*** *are specified similarly. Using this we can define a predicate specifying that the first request which has not been acknowledged is for motor one.*

$$\begin{aligned}\text{req1first}(t) = {} & \text{request1}(t) \wedge \forall t' : (t' < t \wedge \texttt{req1}(t')\wedge \\ & \forall t'' : (t' < t'' \wedge t'' < t) \Rightarrow \neg\texttt{req1}(t'')) \Rightarrow \\ & ((\text{request2}(t') \Rightarrow \\ & \exists t'' : t' < t'' \wedge t'' < t \wedge \neg\text{request2}(t''))\wedge \\ & (\text{request3}(t') \Rightarrow \\ & \exists t'' : t' < t'' \wedge t'' < t \wedge \neg\text{request3}(t'')))\end{aligned}$$

The predicate specifies that at the current position, motor one is requesting and that there is at position t' a ***req1*** *action which has not been handled. If it is the* *case that at position t' motor two (three) is already requesting then there is later position where motor two (three) is not requesting any more (so the request has been handled before the current position). Again, predicates `req2first(t)` and `req3first(t)` are specified similarly. With these basic predicates as building blocks we can give a specification closely related to the informal requirements of the example.*

$$
\begin{aligned}
&\forall t : (\textit{turnon1}(t) \vee \textit{turnon2}(t) \vee \textit{turnon3}(t)) \Rightarrow \mathrm{alloff}(t) \wedge \\
&\forall t : \textit{turnon2}(t) \Rightarrow (\neg \mathrm{request1}(t) \wedge \mathrm{req2first}(t)) \wedge \\
&\forall t : \textit{turnon3}(t) \Rightarrow (\neg \mathrm{request1}(t) \wedge \mathrm{req3first}(t)),
\end{aligned}
$$

An informal specification for the control of the crane containing three properties was given in Section 13.5.2. Each of these properties corresponds to one of the lines in the predicate above. For instance the first line of the predicate specifies that if a motor is turned on then all the motors are turned off. This corresponds to the first property that only one motor can be turned on at a time. For the remaining two lines, motor 2 (3) can be turned on, if there is no request for motor 1 and the request for motor 2 (3) was first.

Since our basic control program specifies the order of the events in the individual tasks (first `req`, then `turnon`, then `req`, and then `turnoff`), this specification will define the wanted behavior.

From this specification Mona generates the minimal deterministic automaton which can be seen in Figure 13.1.

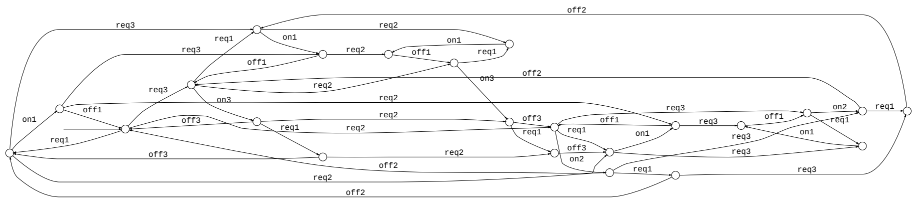


Figure 13.1: The automaton giving priority to motor one.

Had the order of events not been specified in the basic control program, there should also have been predicates specifying this.

Should we want to change the control language of our example in such a way that all three tasks have equal priority, the overall structure of the control automaton would change. As the following example will show, modifying the logical formula is indeed quite comprehensible in the case of the LEGO® crane requirements.

Example 13.3 *Say that we would like to change the requirements such that all motors are given equal priority, that is, they will be turned on in a first come first served manner. Using the logic-based specification, all we have to do is to* *change the last two lines of our specification slightly resulting in the following fifo requirement.*

$$\begin{aligned}
&\forall t : (\textit{\textbf{turnon1}}(t) \vee \textit{\textbf{turnon2}}(t) \vee \textit{\textbf{turnon3}}(t)) \Rightarrow \text{alloff}(t) \wedge \\
&\forall t : \textit{\textbf{turnon1}}(t) \Rightarrow \text{req1first}(t) \wedge \\
&\forall t : \textit{\textbf{turnon2}}(t) \Rightarrow \text{req2first}(t) \wedge \\
&\forall t : \textit{\textbf{turnon3}}(t) \Rightarrow \text{req3first}(t).
\end{aligned}$$

Note that the sub-formulae, such as, ***alloff()*** *and* ***req2first()*** *are reused from the previous specification. As we can see it is relatively easy to change the specification using the previously defined primitives.*

From this specification Mona *generates the automaton in Figure 13.2 which looks quite different from the one in Figure 13.1.*

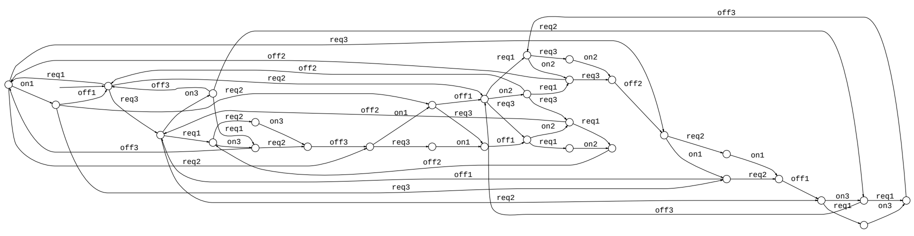


Figure 13.2: Control automaton giving equal priority to motor one, two, and three.

13.6 Merging Automata and RCX™ Code

Given a control automaton and the basic control program, one can synthesize the complete control program. In this section we describe how to translate an automaton into RCX™ code and how this is merged with the existing RCX™ control program. For our example we have done this by hand. It should be clear from this section that only standard techniques are used and these can easily be carried out automatically.

13.6.1 Wrapping the RCX™ Code

The execution of the basic control program is restricted to sequences allowed by a control automaton as follows. Firstly, RCX™ code is generated for the control automaton and then this code is merged with the existing RCX™ code. Merging RCX™ code with an automaton can in some sense be considered a program transformation. Each statement involving a request or writing to an output port is replaced by a block of code that tests whether the operation is legal according to the control automaton. For our example an action should be delayed if the control automaton does not allow it, waiting for the other motor(s) to be turned off. Transforming the code for turning motor 0 on, will lead to the following piece of code.

```
While VAR, 4, 3, CONST, 1 'While automaton has not accepted the command
  SetVar 31, CONST, 1     'Arg := on0, the argument for the automaton
  GoSub 0                 'Run the automaton
  SetVar 4, VAR, 22       'Local success:= global success
EndWhile
On "0"                    'Execution of the actual command
```

We have chosen to implement the automaton as a subroutine. Since arguments for subroutines are not supported by the language, passing an argument to the automaton has to be done via a global variable. Similarly, since a subroutine cannot return a value, return values are also placed in global variables for the process to read. The while loop delays the action until the automaton accepts execution of it.

13.6.2 Implementing Mutual Exclusion and Automata

However, the idea described above is not sufficient since we will not allow more tasks to use the automaton simultaneously. In the RCX™ language the problem is obvious since we are using shared variables for passing arguments and results. In general, we also need exclusive access to the automaton since the outcome of the automaton depends on its state when execution begins. If a process accesses the automaton while it is used by another process, the state variable might be corrupted. Therefore we must have exclusive access to the automaton.

In our implementation we have used Dijkstra's algorithm to implement mutual exclusion between several processes [59]. But any correct mutual exclusion algorithm will do. The algorithm uses some shared variables but this is no problem in the RCX™ language since all variables are shared. There are no gotos in the RCX™ language. Therefore, we have used an extra while loop and a success variable for each task. Except from these details, the algorithm is followed directly. Entering the critical region for process one is done as follows.

```
SetVar 27, 2, 0             'success(1):=0
While 0, 27, 2, 2, 0        'while success(1) == 0
  SetVar 30, 2, 1           '  flag(1):=1
  While 0, 24, 3, 2, 1      '  while turn <> 1 do
    If 0, 24, 2, 2, 2       '    if turn == 2 then
      If 0, 29, 2, 2, 0     '      if flag(2) == 0 then
        SetVar 24, 2, 1     '        turn:=1
      EndIf                 '      endif
    Else                    '    else      #turn is 3
      If 0, 28, 2, 2, 0     '      if flag(3) == 0 then
        SetVar 24, 2, 1     '        turn:=1
      EndIf                 '      endif
    EndIf                   '    endif
  EndWhile                  '  endwhile
  SetVar 30, 2, 2           '  flag(1):=2
  SetVar 27, 2, 1          '  success(1):=1
  If 0, 29, 2, 2, 2        '  if flag(2) == 2 then
    SetVar 27, 2, 0        '    success(1):=0
  EndIf                    '  endif
  If 0, 28, 2, 2, 2        '  if flag(3) == 2 then
    SetVar 27, 2, 0        '    success(1):=0
  EndIf                    '  endif
EndWhile                   'endwhile
```

The last two if statements is an 'unfolding' of the for loop

```
for j<>1 do
  if flag(j)==2 then success(1):=0
```

Leaving the critical region is simple, just setting ones own flag to zero.

An automaton is implemented in the standard way by representing the transition relation as nested conditionals of depth two, branching on the current state and the input symbol respectively. The current state and the input symbol is represented by one variable each. This gives us a way to combine the run of an automaton with the execution of standard RCX™ code with wrapped input/output statements. Implementation of the automaton in RCX™ code looks like.

```
BeginOfSub 0
SetVar 22, 2, 1         'Global success :=1
If 0, 23, 2, 2, 0       'Initial state
  If 0, 31, 2, 2, 1     'Label==on1
    SetVar 23, 2, 1     'New state := state 1
  Else
    If 0, 31, 2, 2, 2   'Label == on2
      SetVar 23, 2, 2   'New state :=state 2
    Else
      If 0, 31, 2, 2, 3 'Label==on3
        SetVar 23, 2, 3 'New state := state 3
      Else
        SetVar 22, 2, 0 'Global success :=0
      EndIf
    EndIf
  EndIf
Else
  If 0, 23, 2, 2, 1     'State 1
  ...
  EndIf
EndIf
EndOfSub
```

13.6.3 Variations of the Method

In the example an action is delayed if the control automaton does not grant permission at once. Depending on the problem to be solved the action taken when permission is not granted can vary. That is, there are various ways of handling this temporary lack of acknowledgment from the controller:

- as in the above example where the task is *busy waiting*, asking the controller over and over whether its label had been enabled; but
- one could also simply *cancel* or ignore the statement requesting permission and continue execution. This could be done by replacing the busy waiting while loop by an if statement.

The former would often be the preferred approach in cases where the internal state of the code is important, such as, in our example, or in a train gate controller. The latter would be a good choice in cases where the code is written in a reactive style, constantly changing output action based on newly read input, e.g. in autonomous robots.

The property implemented by the automaton in the example was specific to the problem. One could also imagine using the method for general properties e.g. for protecting hardware against malicious sequences of actions. This leaves at least two options of where to place the automaton:

- as in the example above where the automaton was put *alongside the wrapped code*. Placing the code implementing the automaton at this level seems a natural choice when dealing with properties about the behavior of a specific program solving a particular problem.
- If the property is of a more general kind, one should rather place the automaton *at the level of the operating system.*

So far we have only considered untimed properties. One could easily imagine using automata with discrete time as control automata. This would open for a whole new range of properties to be specified, e.g. a minimum delay between two actions. In the example it would be possible to specify properties like that a minimum time of 5 seconds should pass between stopping the crane and starting to move the hook.

On the RCX™ this could be realized by having a variable representing the discrete time. This variable could be updated by a task consisting of an infinite loop waiting for one time unit and then updating the variable. Assuming variable number zero represents the time, it could be updated by:

```
SetVar 0, CONST, 0       'Initialize the timer
Loop CONST, 0            'An infinite loop
  Wait CONST, 10         'Wait for 1 sec.
  SumVar 0, CONST, 1     'Update the timer
EndLoop
```

13.7 Conclusion

We have used control automata in conjunction with basic control programs for synthesizing complete control programs. Using this method one can add to a basic control program a control automaton which will ensure that certain safety properties are satisfied. We have used M2L to specify the control automata and the Mona tool to translate formulae into automata.

The approach has been implemented in the setting of the LEGO® RCX™ system. This has allowed for the possibility of testing the implementations on real physical systems.

Based on our experiments we find the method well suited for synthesis of programs ensuring safety properties like the ones we have used. We find the main advantage of the method is the ease of testing different specifications. The separation of the active control program and the restricting automaton also allows for ensuring (new) safety specifications to existing control programs. In critical systems one might consider the automaton only for monitoring actions, not restricting these, to avoid deadlocks.

The main disadvantage of the method is the restriction to safety properties. Since the all concurrent tasks must access the automaton there is a danger of this becoming bottleneck.

Future work There is an overhead connected with gaining exclusive access to the automaton and running it. How much time is spent on gaining access to the automaton of course depends on the arrival of input events. It would be interesting to calculate some specific times for this given some input sequences. A tool for translating RCX™ programs to timed models supported by Uppaal [109] exists [78]. Using Uppaal one can "measure" the time spent by a program from an input is read until the response arrives.

The example presented in this paper only has one component (one crane) and the control restrictions are consequently imposed on that particular component only. One could easily imagine having several components in a distributed environment working to achieve a common goal. By use of modular synthesis and distributed control [131] via independence analysis [135] one can statically infer information about which constraints to put locally on the components and which to put on the (most often necessary) central controller.

Bibliography

[1] E. Aarts, P. van Laarhoven, J. Lenstra, and N. Ulder. A Computational Study of Local Search Algorithms for Job-Shop Scheduling. *OSRA Journal on Computing*, 6(2):118–125, Spring 1994.

[2] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[3] M. Abadi and L. Lamport. An Old-Fashioned Recipe for Real Time. In *Proc. of REX Workshop "Real-Time: Theory in Practice"*, number 600 in Lecture Notes in Computer Science, 1992.

[4] P. Abdulla, K. Cerans, B. Jonsson, and T. Yih-Kuen. General decidability theorems for infinite-state systems. In *Proc. of IEEE Symp. on Logic in Computer Science*. IEEE Computer Society Press, 1996.

[5] P. Abdulla and B. Jonsson. Undecidability of verifying programs with unreliable channels. In *Proc. 21st Int. Coll. Automata, Languages, and Programming (ICALP'94)*, volume 820 of *LNCS*, 1994.

[6] P. A. Abdulla and A. Nylén. Better is better than well: On efficient verification of infinite-state systems. In *Proc. of the 14*th *IEEE Symp. on Logic in Computer Science*. IEEE, 2000.

[7] S. Aggarwal, R. Alonso, and C. Courcoubetis. Distributed reachability analysis for protocol verification environments. In *Discrete Event Systems: Models and Applications. IIASA Conference*, pages 40–56, 1987.

[8] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for Real-Time Systems. In *Proc. of Logic in Computer Science*, pages 414–425. IEEE Computer Society Press, June 1990.

[9] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[10] R. Alur, C. Courcoubetis, and T. A. Henzinger. Computing accumulated delays in real-time systems. In *Proc. of the 5th Int. Conf. on Computer Aided Verification*, number 697 in Lecture Notes in Computer Science, pages 181–193, 1993.

[11] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Sysetems*, number 736 in Lecture Notes in Computer Science, pages 209—229. Springer–Verlag, 1993.

[12] R. Alur and D. Dill. Automata for Modelling Real-Time Systems. In *Proc. of Int. Colloquium on Algorithms, Languages and Programming*, number 443 in Lecture Notes in Computer Science, pages 322–335, July 1990.

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

[14] R. Alur and T. A. Henzinger. Logics and Models of Real-Time: A Survey. In *Proc. of REX Workshop "Real-Time: Theory in Practice"*, number 600 in Lecture Notes in Computer Science. Springer–Verlag, 1992.

[15] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric Real-time Reasoning. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 592–601, 1993.

[16] R. Alur and R. P. Kurshan. Timing Analysis in COSPAN. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 220–231. Springer–Verlag, Oct. 1995.

[17] R. Alur, S. L. Torre, and G. J. Pappas. Optimal paths in weighted timed automata. To appear in Hybrid Systems: Computation and Control, 2001.

[18] H. R. Andersen and M. Mendler. An asynchronous process algebra with multiple clocks. In *Proceedings of ESOP'94*, volume 788 of *Lecture Notes in Computer Science*, pages 58–73. Springer–Verlag, 1994.

[19] J. H. Andersen, K. J. Kristoffersen, K. G. Larsen, and J. Niedermann. Automatic synthesis of real time systems. In *In Proc. of ICALP'95*, Lecture Notes in Computer Science. Springer–Verlag, 1995.

[20] A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In E. A. Emerson and A. P. Sistla, editors, *Proc. of the 12th Int. Conf. on Computer Aided Verification*, number 1855 in Lecture Notes in Computer Science, pages 419–434. Springer–Verlag, 2000.

[21] D. Applegate and W. Cook. A Computational Study of the Job-Shop Scheduling Problem. *OSRA Journal on Computing 3*, pages 149–156, 1991.

[22] E. Asarin, P. Caspi, and O. Maler. A Kleene theorem for timed automata. In *Proc. of IEEE Symp. on Logic in Computer Science*, pages 160–171. IEEE Computer Society Press, 1997.

[23] E. Asarin and O. Maler. As soon as possible: Time optimal control for timed automata. In F. Vaandrager and J. van Schuppen, editors, *Hybrid Systems: Computation and Control*, number 1569 in Lecture Notes in Computer Science, pages 19–30. Springer–Verlag, Mar. 1999.

[24] G. Bandini, R. Lutje Spelberg, and H. Toetenel. Parametric verification of the IEEE 1394a root contention protocol using LPMC. http://tvs.twi.tudelft.nl/, July 2000. Submitted for publication.

[25] J. Beasley, M. Krishnamoorthy, Y. Sharaiha, and D. Abramson. Scheduling aircraft landings - the static case. The Management School, Imperial College. Working paper, 1998. To appear in Transportation Science, vol. 34, 2000.

[26] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, and J. Romijn. Efficient guiding towards cost-optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001*, number 2031 in Lecture Notes in Computer Science, pages 174–188. Springer Verlag, 2001.

[27] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, and J. Romijn. Efficient guiding towards cost-optimality in UPPAAL. Technical Report RS-01-4, BRICS, Jan. 2001.

[28] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-cost reachability for priced timed automata. In M. D. D. Benedetto and A. Sangiovanni-Vincentelli, editors, *Proceedings of the 4th International Workshop on Hybrid Systems:Computation and Control*, number 2034 in Lecture Notes in Computer Science, pages 147–161. Springer Verlag, 2001.

[29] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-cost reachability for priced timed automata. Technical Report RS-01-03, BRICS, Jan. 2001.

[30] G. Behrmann, T. Hune, and F. Vaandrager. Distributing timed model checking – How the search order matters. In E. A. Emerson and A. P. Sistla, editors, *Proc. of the 12th Int. Conf. on Computer Aided Verification*, number 1855 in Lecture Notes in Computer Science, pages 216–231. Springer–Verlag, 2000.

[31] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In N. Halbwachs and D. Peled, editors, *Proc. of the 11th Int. Conf. on Computer Aided Verification*, number 1633 in Lecture Notes in Computer Science, pages 341–353. Springer–Verlag, 1999.

[32] J. Bengtsson, W. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In R. Alur and T. A. Henzinger, editors, *Proc. of the 8th Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer–Verlag, July 1996.

[33] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial Order Reductions for Timed Systems. In *Proc. of CONCUR '98: Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 485–500, 1998.

[34] L. Björnfot. Ada and timed automata. In *In Proc. of Ada in Europe*, number 1031 in LNCS, pages 389–405. Springer–Verlag, 1995.

[35] R. Boel. Automatic synthesis of schedules in a timed discrete event plant. In *Proceedings of ADPM 2000: 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems*, 2000.

[36] R. Boel and G. Stremersch. Report for VHS: Timed Petri Net Model of Steel Plant at SIDMAR. Technical report, SYSTeMS Group, University Ghent, 1999.

[37] R. Boel and G. Stremersch. VHS case study 5: modelling and verification of scheduling for steel plant at SIDMAR. draft, 1999.

[38] R. Boel and G. Stremersch. VHS case study 5: Timed Petri net model of steel plant at SIDMAR. VHS deliverable, 1999.

[39] R. Boel and G. Stremersch. VHS Case Study 5: Timed Petri net model of steel plant at SIDMAR. Technical report, SYSTeMS Group, Universiteit Gent, Technologiepark-Zwijnaarde 9, B-9052 Ghent, Belgium, 1999.

[40] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Are timed automata updatable? In E. A. Emerson and A. P. Sistla, editors, *Proc. of the 12th Int. Conf. on Computer Aided Verification*, number 1855 in Lecture Notes in Computer Science, pages 464–479. Springer–Verlag, 2000.

[41] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In *Proc. of the 10th Int. Conf. on Computer Aided Verification*, number 1427 in Lecture Notes in Computer Science, pages 546–550. Springer–Verlag, 1998.

[42] C. Brabrand. Synthesizing safety controllers for interactive Web services. Master's thesis, Department of Computer Science, University of Aarhus, December 1998. Available from `http://www.brics.dk/~brabrand/thesis/` .

[43] E. Brinksma and A. Mader. Verification and optimization of a plc control schedule. In *Proceedings of the 7th SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[44] P. Brucker, B. Jurisch, and B. Sievers. Code of a Branch & Bound Algorithm for the Job Shop Problem. Available at url `http://www.mathematik.uni-osnabrueck.de/research/OR/`, 1995.

[45] J. R. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.

[46] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In O. Grumberg, editor, *Proc. of the 9th Int. Conf. on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer–Verlag, June 1997. Haifa, Isreal, June 22-25.

[47] S. Caselli, G. Conte, and P. Marenzoni. Parallel state space exploration for GSPN models. In G. D. Michelis and M. Diaz, editors, *Proceedings 16th Int. Conf. on Application and Theory of Petri Nets,Turin, Italy*, volume 935 of *Lecture Notes in Computer Science*, pages 181–200. Springer–Verlag, June 1995.

[48] G. Cattani, M. Fiore, and G. Winskel. A theory of recursive domains with applications to concurrency. In *Proc. of IEEE Symp. on Logic in Computer Science*, pages 214–225. IEEE Computer Society Press, 1998.

[49] K. Čerāns. Decidability of Bisimulation Equivalences for Parallel Timer Processes. In *Proc. of CAV'92*, number 663 in Lecture Notes in Computer Science, pages 302–315, Berlin, 1992. Springer–Verlag.

[50] K. Cerans. Deciding properties of integral relational automata. In *Proceedings of ICALP 94*, volume 820 of *LNCS*, 1994.

[51] K. Cerans, J. C. Godskesen, and K. G. Larsen. Time Modal Specification – Theory and Tools. In *Proc. of the 5th Int. Conf. on Computer Aided Verification*, number 697 in Lecture Notes in Computer Science. Springer–Verlag, 1993.

[52] A. Cheng and M. Nielsen. Open maps (at) work. In Thiagarajan, editor, *In Proc. of FST&TCS '95*, volume 1026 of *Lecture Notes in Computer Science*, pages 263–278. Springer–Verlag, 1996.

[53] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, Cambridge, Massachusetts, 1999.

[54] J. C. Corbett. Modeling and analysis of real-time Ada tasking programs. In K. Ramamritham, editor, *Proc. of the* 15th *IEEE Real-Time Systems Symposium*, pages 132–141. IEEE Computer Society Press, December 1994.

[55] J. C. Corbett. Constructing abstract models of concurrent real-time software. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 250–260. IEEE Computer Society Press, Januar 1996.

[56] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw-Hill, Inc., 1991.

[57] P. D'Argenio, J.-P. Katoen, T. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In E. Brinksma, editor, *Proc. of the 3*rd *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1217 in Lecture Notes in Computer Science, pages 416–431. Springer–Verlag, Apr. 1997.

[58] C. Daws and S. 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*, number 1384 in Lecture Notes in Computer Science, pages 313–329. Springer–Verlag, 1998.

[59] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[60] D. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In J. Sifakis, editor, *Proc. of Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 197–212. Springer–Verlag, 1989.

[61] D. Dill. The Murφ Verification System. In R. Alur and T. A. Henzinger, editors, *Proc. of the 8th Int. Conf. on Computer Aided Verification*, number 1102 in Lecture Notes in Computer Science, pages 390–393. Springer–Verlag, 1996.

[62] C. Elgot. Decision problems of finite automata design and related arithm etics. *Transactions of the American Mathematical Society*, 98:21–52, 1961.

[63] A. Fehnker. Bounding and heuristics in forward reachability algorithms. Technical Report CSI-R0002, Computing Science Institute Nijmegen, 1999.

[64] A. Fehnker. Scheduling a steel plant with timed automata. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA99)*, pages 280–286. IEEE Computer Society, 1999.

[65] A. Finkel and P. Schnoebelen. Fundamental structures in well-structured infinite transition systems. In *Proc. 3rd Latin American Theoretical Informatics Symposium (LATIN'98)*, volume 1380 of *LNCS*, 1998.

[66] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere. Research Report LSV-98-4, Lab. Specification and Verification, ENS de Cachan, France, 1998. To appear in TCS., 1998.

[67] R. V. Glabeek and U. Goltz. Equivalence notions for concurrent systems and refinement of actions. In *Proc. of MFCS'89*, number 379 in Lecture Notes in Computer Science, pages 237–248. Springer–Verlag, 1989.

[68] V. Gupta, T. A. Henzinger, and R. Jagadeesan. Robust timed automata. In O. Maler, editor, *Hybrid and Real-Time Systems*, Lecture Notes in Computer Science, pages 331–345. Springer–Verlag, March 1997.

[69] K. Havelund, K. Larsen, and A. Skou. Formal verification of a power controller using the real-time model checker UPPAAL. In J.-P. Katoen, editor, *Formal Methods for Real-Time and Probabilistic Systems, 5th International AMAST Workshop, ARTS'99*, volume 1601 of *Lecture Notes in Computer Science*, pages 277–298. Springer-Verlag, 1999.

[70] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal Modeling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, Dec. 1997.

[71] B. Haverkort, A. Bell, and H. Bohnenkamp. On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets. In *Proceedings of the 8th International Workshop on Petri Nets and Performance Models (PNPM'99), Zaragoza, Spain*, pages 12–21. IEEE Computer Society Press, 1999.

[72] T. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, pages 278–292. IEEE Computer Society Press, 1996.

[73] T. A. Henzinger and P.-H. Ho. HyTech: The Cornell HYbrid TECHnology Tool. In *Proc. of TACAS, Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, 1995. BRICS report series NS–95–2.

[74] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A Model Checker for Hybird Systems. In O. Grumberg, editor, *Proc. of the 9th Int. Conf. on Computer Aided Verification*, number 1254 in Lecture Notes in Computer Science, pages 460–463. Springer–Verlag, 1997.

[75] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *Proc. of IEEE Symp. on Logic in Computer Science*, 1992.

[76] G. Higman. Ordering by divisibility in abstract algebras. *Proc. of the London Math. Soc.*, 2:326–336, 1952.

[77] G. Holzmann. *The Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[78] T. Hune. Modeling a real-time language. In S. Gnesi and D. Latella, editors, *Proc. of Workshop on Formal Methods for Industrial Critical Systems, FMICS'99*, volume 2, pages 259–282, 1999.

[79] T. Hune. Modeling a language for embedded systems in timed automata. Technical Report RS-00-17, BRICS, Aug. 2000.

[80] T. Hune, K. G. Larsen, and P. Pettersson. Guided synthesis of control programs using UPPAAL for VHS case study 5. VHS deliverable, 1999.

[81] T. Hune, K. G. Larsen, and P. Pettersson. Guided synthesis of control programs for a batch plant using UPPAAL. Technical Report RS-00-37, BRICS, Dec. 2000.

[82] T. Hune, K. G. Larsen, and P. Pettersson. Guided Synthesis of Control Programs Using UPPAAL. In T. H. Lai, editor, *Proc. of the IEEE ICDCS International Workshop on Distributed Systems Verification and Validation*, pages E15–E22. IEEE Computer Society Press, Apr. 2000.

[83] T. Hune, K. G. Larsen, and P. Pettersson. Guided synthesis of control programs using UPPAAL. *Nordic Journal of Computing*, 2001. Accepted for publication.

[84] T. Hune and M. Nielsen. Timed bisimulation and open maps. In *In Proc. of Mathematical Foundations of Computer Science, MFCS'98*, volume 1450 of *Lecture Notes in Computer Science*, pages 378–387. Springer–Verlag, 1998.

[85] T. Hune and M. Nielsen. Timed bisimulation and open maps. Technical Report RS-98-4, BRICS, Feb. 1998.

[86] T. Hune, J. Romijn, M. Stoelinga, and F. Vaandrager. Linear parametric model checking of timed automata. In T. Margaria and W. Yi, editors, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001*, number 2031 in Lecture Notes in Computer Science, pages 189–203. Springer Verlag, 2001.

[87] T. Hune, J. Romijn, M. Stoelinga, and F. Vaandrager. Linear parametric model checking of timed automata. Technical Report RS-01-05, BRICS, Jan. 2001.

[88] T. Hune and A. Sandholm. A case study on using automata in control synthesis. In T. Maibaum, editor, *Proc. of Fundamental Approaches to Software Engineering, FASE 2000*, number 1783 in Lecture Notes in Computer Science, pages 349–362. Springer–Verlag, March/April 2000.

[89] T. S. Hune and A. B. Sandholm. Using automata in control synthesis — A case study. Technical Report RS-00-22, BRICS, Sept. 2000.

[90] D. Hung. Modelling and verification of biphase mark protocols using PVS. In *Proceedings of the International Conference on Applications of Concurrency to System Design (CSD'98), Aizu-wakamatsu, Fukushima, Japan, March 1998*, pages 88–98. IEEE Computer Society Press, 1998.

[91] IEEE Computer Society. IEEE Standard for a High Performance Serial Bus. Std 1394-1995, Aug. 1996.

[92] T. K. Iversen, K. J. Kristoffersen, K. G. Larsen, M. Laursen, R. G. Madsen, S. K. Mortensen, P. Pettersson, and C. B. Thomasent. Model-checking real-time control programs. In *In Proceedings of the 12th Euromicro Conference on Real-Time Systems (ECRTS'2000)*, pages 147–155. IEEE Computer Society Press, June 2000.

[93] A. Jain and S. Meeran. Deterministic job-shop scheduling; past, present and future. *European Journal of Operational Research*, 1999. to appear in volume 113, issue 2.

[94] K. Jensen and G. Rozenberg, editors. *High-level Petri Nets – Theory and Application.* Springer-Verlag, 1991.

[95] A. Joyal, M. Nielsen, and G. Winskel. Bisimulation from open maps. *Information and Computation*, 127,2:164–185, 1996.

[96] N. Klarlund and A. Møller. *MONA Version 1.3 User Manual.* BRICS Notes Series NS-98-3 (2.revision), Department of Computer Science, University of Aarhus, October 1998.

[97] S. Kowalewski. Description of VHS case study 1 "Experimental Batch Plant". Draft. University of Dortmund, Germany, July 1998.

[98] K. Kristoffersen and J. Niedermann. User's manual for Epsilon. Available via anonymous ftp at cs.auc.dk, Dec. 1994.

[99] K. J. Kristoffersen, F. Laroussinie, K. G. Larsen, P. Pettersson, and W. Yi. A Compositional Proof of a Real-Time Mutual Exclusion Protocol. In *Proc. of the 7*th *Int. Joint Conf. on the Theory and Practice of Software Development*, Apr. 1997.

[100] K. J. Kristoffersen, K. G. Larsen, P. Pettersson, and C. Weise. Experimental batch plant - VHS case study 1 using uppaal. VHS deliverable, May 1999. Draft.

[101] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. In J.-P. Katoen, editor, *Proc. of ARTS'99*, number 1601 in Lecture Notes in Computer Science, pages 75–95. Springer–Verlag, 2000.

[102] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, Feb. 1987.

[103] F. Laroussinie, K. G. Larsen, and C. Weise. From Timed Automata to Logic — and Back. In *Proc. of MFCS'95*, Lecture Notes in Computer Sciencie, 1995. Also BRICS report series RS–95–2.

[104] K. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991.

[105] K. G. Larsen, G. Behrmann, E. Briksma, A. Fehnker, T. Hune, P. Pettersson, and J. Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. Accepted for CAV 2001.

[106] K. G. Larsen, P. Pettersson, and W. Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, Dec. 1995.

[107] K. G. Larsen, P. Pettersson, and W. Yi. Diagnostic Model-Checking for Real-Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 575–586. Springer–Verlag, Oct. 1995.

[108] K. G. Larsen, P. Pettersson, and W. Yi. Model-Checking for Real-Time Systems. In *Proc. of Fundamentals of Computation Theory*, number 965 in Lecture Notes in Computer Science, pages 62–88, Aug. 1995.

[109] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[110] K. G. Larsen, C. Weise, W. Yi, and J. Pearson. Clock difference diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999.

[111] F. Larsson, K. G. Larsen, P. Pettersson, and W. Yi. Efficient Verification of Real-Time Systems: Compact Data Structures and State-Space Reduction. In *Proc. of the 18*th *IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, Dec. 1997.

[112] LEGO. *Software developers kit*, November 1998. See `http://www.legomindstorms.com/`.

[113] H. Lescow. On polynomial-size games winning finite-state games. In *Proc. of the 7th Int. Conf. on Computer Aided Verification*, number 939 in Lecture Notes in Computer Science, pages 239–252. Springer–Verlag, 1995.

[114] M. Lin. Syntehsis of control software in a layered architecture from hybrid automata. In F. Vaandrager and J. van Schuppen, editors, *Hybrid Systems: Computation and Control*, number 1569 in Lecture Notes in Computer Science, pages 152–164. Springer–Verlag, Mar. 1999.

[115] H. Lönn and P. Pettersson. Formal Verification of a TDMA Protocol Startup Mechanism. In *Proc. of the Pacific Rim Int. Symp. on Fault-Tolerant Systems*, pages 235–242, Dec. 1997.

[116] R. Lutje Spelberg, W. Toetenel, and M. Ammerlaan. Partition refinement in real-time model checking. In A. Ravn and H. Rischel, editors, *Proc. of Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*, pages 143–157. Springer–Verlag, 1998.

[117] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Fransisco, California, 1996.

[118] Z. Manna and A. Pnueli. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, January 1984.

[119] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic petri nets.* Wiley series in parallel computing. John Wiley & Sons, 1995.

[120] S. Mauw and M. A. Reniers. An algebraic semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.

[121] R. Milner. *Communication and Concurrency.* Prentice Hall, Englewood Cliffs, 1989.

[122] R. Milner and D. Sangiorgi. Barbed bisimulation. In *In Proc. of ICALP'92*, Lecture Notes in Computer Science, pages 685–695. Springer–Verlag, July 1992.

[123] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. Technical Report IT-TR-1999-023, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, Feb. 1999.

[124] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Fully symbolic model checking of timed systems using difference decision diagrams. In *Proceedings First International Workshop on Symbolic Model Checking*, volume 23-2 of *Electronic Notes in Theoretical Computer Science*, pages 89–108, Trento, Italy, July 1999.

[125] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. *Acta Informatica*, 30:pages 181–202, 1993.

[126] P. Niebert, S. Tripakis, and S. Yovine. Minimum-time reachability for timed automata. In *IEEE Mediteranean Control Conference*, 2000. Accepted for publication.

[127] P. Niebert and S. Yovine. Computing optimal operation schemes for multi batch operation of chemical plants. VHS deliverable, May 1999. Draft.

[128] M. Nielsen and T. Hune. Bisimulation and open maps for timed transition systems. *Fundamenta Informatica, special issue dedicated to Professor Arto Salomaa*, pages 61–77, 1999.

[129] N.V. SIDMAR. Planung und Synchronisation der Anlagen im Stahlwerk SIDMAR. Ghent, Belgium, October 1998.

[130] D. Park. Concurrency and automata on infinite sequences. In *Proc. of 5th GI Conference*, number 104 in Lecture Notes in Computer Science, Berlin, 1981. Springer–Verlag.

[131] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[132] F. Reffel and S. Edelkamp. Error Detection with Directed Symbolic Model Checking. In *Proc. of Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 195–211. Springer–Verlag, 1999.

[133] T. G. Rokicki. *Representing and Modeling Digital Circuits.* PhD thesis, Stanford University, 1993.

[134] T. C. Ruys and E. Brinksma. Experience with Literate Programming in the Modelling and Validation of Systems. In B. Steffen, editor, *Proceedings of the Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, number 1384 in Lecture Notes in Computer Science (LNCS), pages 393–408, Lisbon, Portugal, Apr. 1998. Springer-Verlag, Berlin.

[135] A. Sandholm and M. I. Schwartzbach. Distributed safety controllers for Web services. In E. Astesiano, editor, *In Proc. of Fundamental Approaches to Software Engineering, FASE'98*, number 1382 in Lecture Notes in Computer Science, pages 270–284. Springer–Verlag, March/April 1998. Also available as BRICS Technical Report RS-97-47.

[136] D. Simons and M. Stoelinga. Mechanical verification of the IEEE 1394a root contention protocol using Uppaal2k. Technical Report CSI-R0009, Computing Science Institute, University of Nijmegen, May 2000. Conditionally accepted for *STTT.*

[137] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI:The Complete Reference.* MIT Press, Cambridge, Massachusetts, 1996.

[138] U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In O. Grumberg, editor, *Proc. of the 9th Int. Conf. on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267. Springer–Verlag, June 1997. Haifa, Isreal, June 22-25.

[139] M. Stobbe. Results on scheduling the sidmar steel plant using constraint programming. Internal report, 2000.

[140] M. Stoelinga and F. Vaandrager. Root contention in IEEE 1394. In J.-P. Katoen, editor, *Proceedings 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems, Bamberg, Germany*, volume 1601 of *Lecture Notes in Computer Science*, pages 53–74. Springer–Verlag, 1999.

[141] F. Vaandrager. Analysis of a biphase mark protocol with UPPAAL. Slides, available from http://www.cs.kun.nl/ fvaan/publications.html, 2000.

[142] F. Wang. Efficient data structure for fully symbolic verification of real-time software systems. In S. Graf and M. Schwartzbach, editors, *Proc. of the 6*th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1785 in Lecture Notes in Computer Science, pages 157–171. Springer–Verlag, 2000.

[143] C. Weise and D. Lenzkes. Efficient scaling-invariant checking of timed bisimulation. In *Proc. of STACS'97*, volume 1200 of *Lecture Notes in Computer Science*, pages 177–188. Springer–Verlag, 1997.

[144] G. Winskel and M. Nielsen. Presheaves as transition systems. In *Partial order methods in verification*, DIMACS, pages 129–140. 1996.

[145] W. Yi. Real–time behaviour of asynchronous agents. In *Proc. of CONCUR '90, Theories of Concurrency: Unification an d Extension*, number 458 in Lecture Notes in Computer Science, pages 502–520. Springer–Verlag, 1990.

[146] W. Yi, P. Pettersson, and M. Daniels. Automatic Verification of Real-Time Communicating Systems By Constraint-Solving. In D. Hogrefe and S. Leue, editors, *Proc. of the 7*th *Int. Conf. on Formal Description Techniques*, pages 223–238. North–Holland, 1994.

[147] S. Yovine. Model-Checking Timed Automata. In G. Rozenberg and F. Vaandrager, editors, *Embedded Systems*, number 1494 in Lecture Notes in Computer Science, pages 114–152. Springer–Verlag, 1998. SBN 3-540-65193-4.