Brno University of Technology
Faculty of Information Technology

PhD. Thesis

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Symbolic Data Structures for Parametric Verification

by

Ing. Petr Matoušek

Supervisor: Prof. Ing. Miroslav Švéda, CSc.

Submitted on: 27th May 2005

State doctoral exam passed on: 13th June 2000

The thesis is available at the library of the Faculty of Information Technology.

Abstract

Hardware and software systems will inevitably grow in scale and functionality. Because of this increase in complexity, the likelihood of subtle errors is much greater. Moreover, some of these errors may cause a loss of money, time, or even human life. A major goal of software engineering is to enable developers to construct systems that operate reliably despite this complexity. One way of achieving this goal lies in using *formal methods*, which are mathematically-based languages, techniques, and tools for specifying and verifying such systems. Use of formal methods does not *a priori* guarantee correctness. However, they can greatly increase our understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might otherwise go undetected.

Specification is the process of describing a system and its desired properties. Formal specification uses a language with a mathematically-defined syntax and semantics. The kinds of system properties might include functional behavior, timing behavior, performance characteristics, or internal structure. So far, specification has been most successful for behavioral properties. One current trend is to integrate different specification languages, each able to handle a different aspect of a system. Another one is to handle non-behavioral aspects of a system like its performance, real-time constraints, security policy, and architectural design.

In this thesis we work with real time systems augmented with parameters. Traditional approaches to the analysis and verification of real-time systems deal with timed models where time is expressed by variables that are compared with explicite values (e.g., integers).

Parametric timed and counter models use parameters to define constraints over clocks or counters. Verification of automata with parameters is generally undecidable. However, it is decidable for some restricted classes of parametric systems and moreover many practical systems outside these classes may be successfully verified using semi-algorithms.

Analysis mostly depends on the efficient data structure that is used to express behavior of the system. In this work we deal with parameterized timed and counter systems and discuss data structures that are used for their representation.

The main goal of the thesis is to introduce a new data structure called parametric hypercubes based on parameterized intervals and operations that are needed for verification. This structure makes operations over parametric counter automata simpler in comparison with other approaches. The advantage of this structure is that it reduces the space needed to represent data and simplifies some operations (emptiness test, intersection, etc.) with respect to other structures (e.g., parametric DBM or polyhedra). Another advantage is that the structure extends expressivity of constraints (guards) on transitions in comparison with the original definition of timed and counter automata. It allows constraints of the form $x_1 + \ldots + x_n \prec t$ while DBMs allow comparisons of terms with only two variables $x_i - x_j \prec t$. This new structure was implemented as a library of the verification tool TReX [BCAS01].

Another contribution of the thesis is a part about analysis of parametric systems. This part discusses fundamentals of modelling, analysis and verification of parameterized timed and counter systems. It introduces a methodology for modelling communicating systems that include three basic components—environment, buffers, and executive units. We show models of a simple bounded FIFO queue, a delayed queue and a lossy queue that can be reused for automata-based verification languages where queues are not part of the language or where a customization is needed.

These proposed principles are demonstrated on a case study—verification of a multicast protocol PGM. The verification was done using combination of three verification tools (HyTech, TReX, and Uppaal). In this work we present our experience with parametric verification, our results, and several sources of complexity with suggestions how to deal with them.

Preface

For the LORD gives wisdom,
and from his mouth come knowledge and understanding.
He holds victory in store for the upright,
he is a shield to those whose walk is blameless,
for he guards the course of the just
and protects the way of his faithful ones
Bible

This work has been carried out at the Faculty of Information Technology, Brno, Czech republic and at the laboratory of LIAFA, Paris, France. I am grateful to Professor Miroslav Švéda for supervising my work here in Czech republic and to Professor Ahmed Bouajjani for his leading my work in Paris during my stay there.

My thanks belong to my close co-workers and friends Dr. Tomáš Vojnar (FIT) and Mihaela Sighireanu (LIAFA) whose valuable hints, notes and encouragement were important for me during my work on this thesis.

I thank my friend Břeťa for his careful reading of the English text. I very appreciate his moral and prayer support which was essential to me in the time of finishing this work. Thanks to my friends who are not mentioned here for their valuable support.

I am also grateful to my mother for her support and her love.

Petr Matoušek
Brno, 26th May 2005

Contents

1 Introduction **1**
1.1 A Motivation for the Thesis 2
1.2 The Goal of the Thesis 2
1.3 The Structure of the Thesis 3

2 Model Checking of Timed Systems **4**
2.1 Model Checking 4
2.2 Model Checking of Timed Systems 5
2.3 Timed Automata 7
2.4 Data Structures for Timed Automata 10
2.4.1 Regions 10
2.4.2 Zones 11
2.4.3 Difference Bound Matrix (DBM) 13
2.4.4 Example 16

3 Parametric Real-Time Reasoning **19**
3.1 Parametric Timed Automata 20
3.2 Parametric Counter Automata 22
3.3 Comparison of Parametric Timed and Counter Automata 23
3.4 Verification 23
3.5 Parametric DBMs 24
3.5.1 Operations on Constrained PDBMs 27
3.5.2 Decidability 31
3.6 Example 32
3.7 Polyhedra 35
3.7.1 Operations over Polyhedra 37
3.8 Discussion 39

4 Parameterized Hypercubes **42**
4.1 Parameterized Intervals 42
4.1.1 Arithmetical Term, Formula 43
4.1.2 Numerical Bounds 44
4.1.3 Parameterized Bounds 49

4.1.4 Parameterized Intervals . . . 51
4.2 Parameterized Hypercubes . . . 55
4.3 Operations on pHCubes . . . 56
4.4 Implementation in TReX . . . 62
4.5 Discussion . . . 66

5 How to Analyse Parameterized Timed Systems 68
5.1 Formal Analysis of Real-time Systems . . . 68
5.2 Modelling a System . . . 70
5.2.1 Modelling FIFO Queues . . . 71
5.3 Properties to be Verified . . . 73
5.4 Summary . . . 76

6 Case Study 77
6.1 Formal Analysis . . . 77
6.2 Modelling PGM . . . 78
6.3 Tools for Parametric Verification . . . 83
6.3.1 HyTech . . . 83
6.3.2 TReX . . . 86
6.3.3 Uppaal . . . 87
6.4 Discussion . . . 89

7 Conclusion 92

Glossary 94

Bibliography 96

Chapter 1

Introduction

Rapid expansion of Internet, communication systems and embedded applications in different fields of human activities create increasing demands on new applications, tools and equipment that are fast, safe and reliable in terms of functionality. There is a challenge for many software and hardware companies to create reliable and robust systems. Reliability and safety of both software and hardware turn our special attention to testing, simulation and verification. While testing and simulation can prove that a system gives expected output values for specific inputs, verification helps to prove that a given property (e.g., safety) holds for every configuration of the system.

There are two basic approaches to formal verification—theorem proving and model checking. Theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. Verification is a process of finding a proof of a property from the axioms of the system. Model checking is a technique that relies on building a finite model of a system and automatic checking that a desired property holds in that model. During verification a model checker searches a generated state space of all possible behaviors of the system.

Many tools based on model checking and state exploration are efficient and widely used for verification of finite systems. However, real-time systems are in general not finite-state and cannot be fully analyzed. Especially, state space of real-time systems is either infinite or too large for finite-state verification techniques. Since the 90's there exists a large community of scientists and engineers who try to formally describe and prove temporal properties of real-time systems using model checking. There exist several verification tools for timed verification—both for discrete and continuous time—that have been successfully used for verification of non–trivial systems.

One direction of research in the area of timed systems is parametric analysis and verification. Parametric analysis works with systems that contain special variables that are not changed during the execution—parameters. In parametric models, clocks and counters can be compared with parameters. Parameters are used in transitions where they define lower and upper bounds on clocks and counters. Parameters may range over infinite domains and are related by a set of constraints. Using parametric reasoning, we can either verify that a system satisfies a property for all possible values of parameters, or we can find constraints on the parameters that define the set of all possible values for which the system satisfies the property.

Verification of timed automata with parameters is generally undecidable. However, it is decidable for some restricted classes of parametric systems. Moreover, many practical systems outside these classes may be successfully verified using semi-algorithms. Analysis of such systems depends on the efficient data structures that are used to express dynamic behavior of the system. In this work we deal with parameterized timed and counter systems. We explore current data structures that are used for their representation and operations over them. The goal of the thesis is a design and an implementation of a new structure for data representation that extends possibilities of analysis and verification of parametric systems. Such a structure—parameterized hypercubes—was designed and implemented as a library for verification tool TReX. The thesis also contains a part concerning verification of parametric system where we summarize our experience from several projects and propose a methodology for parametric verification of timed system.

1.1 A Motivation for the Thesis

Motivation for the research in the field of verification of parametric timed systems here presented was the extension of formal verification of PGM protocol. Reliability of transmission depended on several parameters—the length of transmission window, rate of the channel, the length of intermediate buffer etc. Our task was to find relations among these parameters that satisfy a desired property. However, with current verification tools we were not able to find a solution because the model contained non-linear relation between parameters as was proved by manual analysis of the model.

We noticed that current implementation of data for parametric reasoning does not allow assignments of the form $x := x + y + c$ where x, y are variables and c is a constant. Thus we decided to design a new data structure based on intervals that would extend a class of analyzed systems. This structure implements operation for symbolic model checking with symbolic domain (reals, integers etc.) and is compatible with current data structure—parametric DBMs.

1.2 The Goal of the Thesis

The main contribution of the thesis is a proposal and implementation of a new data structure called parameterized hypercubes based on parameterized intervals. We introduce here operations that are needed for verification. This new structure makes operations over parametric counter automata simpler in comparison with other approaches. The advantage of this structure is that it reduces the space needed to represent data and simplifies some operations (emptiness test, intersection, etc.) with respect to other structures (e.g., parametric DBM or polyhedra).

Another advantage of this structure is that it extends expressivity of constraints (guards) over transitions in comparison with the original definition of parametric timed and counter automata. Parameterized hypercubes allow constraints of the form $x_1 + \ldots + x_n \prec t$ while parametric DBMs (PDBMs) allow comparisons of terms with only two variables

$x_i - x_j \prec t$. However, adding general constraints is not an exact operation—we show an algorithm that is approximate. We as well extend an operation of linear assignment of the form $x := x_0t_0 + \ldots + x_nt_n + t_{n+1}$ whereas PDBMs use only simple assignment $x := t$.

The second contribution of the thesis is an overview of basic principles of formal analysis and verification of timed and counter systems with parameters. We summarize our experience and give recommendations for analysis of such systems. These principles are demonstrated on a non–trivial example of multicast PGM protocol and from Liberouter project [MSV05].

1.3 The Structure of the Thesis

The thesis is structured into three basic parts. First part covers model checking of timed systems (chapter 2) and parametric reasoning of timed systems (chapter 3). The second part presents a proposed data structure—parameterized hypercube—and operations over it. The third part describes practical experience with verification of real-time systems with parameters and shows a case study of parametric analysis and verification of PGM protocol.

Further, in the first part we give an overview of symbolic model checking of timed systems and show two basic structures for data representation—regions and zones implemented as Difference Bound Matrices (DBMs). Operations over DBMs are important for us because they form the basis which is then extended for parameters. The theory of timed automata and methods for their analysis is known for many years, so we give only short overview of the problem of symbolic model checking of timed systems. Then we move to systems with parameters. We remind the definition and semantics of parametric timed and counter automata and show two data structures that are used for parametric reasoning—parametric DBMs (PDBMs) and polyhedra. We describe operations over them in details and show how parametric analysis differs from non-parametric one. Both PDBMs and polyhedra were implemented in verification tools TReX, resp. HyTech and are used for parametric verification. We mention our experience with both tools and show the need for their extension.

The main part of the thesis (chapter 4) contains definitions of proposed parameterized intervals and hypercubes. We start from well-known numerical intervals that we extend to intervals with bounds formed by parametric terms. We define new operators over them—total order, *min* operator etc. Then we define constrained parameterized intervals and formulas. A set of constrained parameterized intervals forms a parameterized hypercube. This structure is used to represent the valuation of parameters and variables during symbolic analysis of the system. We create operations over them with respect to operations needed for timed verification as showed on example in the DBMs, PDBMs and polyhedra.

The third part of the thesis contains two chapters describing practical experience with parametric analysis (chapter 5) and verification of a real protocol (chapter 6). These two chapters summarize our experience and give recommendations and practical steps for verification of parametric timed systems.

The last chapter (chapter 7) summarizes the contribution of the thesis and points out directions for future research in this area.

Chapter 2

Model Checking of Timed Systems

Model checking is one of the methods used for formal verification of the concurrent systems. The main idea is to generate the state space of the system's behaviour and to check properties on these states. Symbolically it means to check whether our system is a model for desired requirements:

$$system \models requirements$$

Requirements are expressed in temporal logics and the system is modelled as a finite state transition system. Analysis of the system is based on an exhaustive state space search over all possible behaviours of the system. If the model is finite, the termination is guaranteed. Even for infinite timed systems there exist several symbolic techniques that allow us to terminate the analysis–as demonstrated later in the chapter.

The main advantage of model checking is that it is fully automatic. That means it does not require an experienced user to analyse the property of the described system. The major disadvantage of model checking is the state space explosion. The number of configurations increases exponentially with the number of variables, their size and the number of processes of the model. However, there exist powerful techniques that reduce size of the state space and make analysis of the state space possible. Even for (continuous) timed systems we can use model checking techniques for verification.

In this chapter we recall basics of the model checking with focus on timed systems. We show the theory of timed automata and the way how to represent infinite data domains using finite structures. On the example of Difference Bound Matrices (DBMs), we point out basic operations over data structures that are needed for verification. Design and implementation of these operations for our new data structure form the foundation of the thesis.

2.1 Model Checking

Model checking is a technique for verifying finite state concurrent systems such as communication protocols. In comparison with theorem proving it can be performed fully automatically. The procedure uses an exhaustive search of the state space of a system to determine if some specification is true or not. The procedure can terminate with a yes/no answer.

The main disadvantage of model checking consists in the state explosion. There are various techniques to decrease the size of the state space of a system - partial order reduction, binary decision diagrams (BDDs), abstraction, symmetries etc. In this section we will remember some of these techniques.

Partial order reduction. This technique is based on the following observation: computations that differ in the ordering of independently executed events are usually indistinguishable by the specification and can be considered equivalent. There are various approaches to the partial order reduction mentioned in [CGP99]: the stubborn sets proposed by A. Valmari in [Val90], the ample sets presented by D. Peled in [Pel94], or the persistent sets of P. Godefroid introduced in [God90]. Among other techniques belong K. McMillan's unfolding [McM92]. Using these methods we can decrease the size of the state space. As practice shows, partial order reduction is a very successful method for software verification.

Binary decision diagrams (BDDs). One of the approaches to deal with the state explosion problem is using a compact representation of the state space called binary decision diagrams and symbolic model checking over them. Binary decision diagrams introduced by R.E. Bryant in [Bry86] provide a canonical form for boolean formulas that is more compact than conjunctive or disjunctive normal form, and very efficient algorithms for manipulating them. Because the symbolic representation captures some of the regularity in the state space determined by circuits and protocols, it is possible to verify systems with an extremely large number of states. BDDs are very successful especially in hardware verification.

Abstraction. Predicate abstraction is a technique that is used to prove properties of infinite state systems. It is a combination of theorem proving and model checking techniques. Given a concrete infinite state system and a set of abstraction predicates, a conservative finite state abstraction is generated. It is conservative in the sense that for every execution in the concrete system there is a corresponding execution in the abstract system. The abstract version of the verification condition is model checked in this abstract system. If the property is verified then it holds in the concrete system. Otherwise an abstract counter-example trace is generated. More about abstraction can be found in [CGP99, chapter 13].

2.2 Model Checking of Timed Systems

Modelling continuous time. For timed systems, clocks may be real values. The timed transition system is infinite and cannot be simply used for automated verification. However, D. Dill and R. Alur in [AD94] introduced the notion of region equivalence over clock assignments and proved that reachability problem is decidable for timed automata. The main idea of the region technique is that it is possible to find a finite representation of the valuation graph which represents all the necessary reachability information symbolically. At first, infinitely many symbols of the transition system are reduced to the finite number using time abstraction where time increments over time-transitions are hidden (see section 2.3 for details). Then, using equivalence over the state space the number of states (infinite) is represented by the finite number of classes according to the equivalence. In other words, the state space is factorized into a finite number of regions.

Difference Bound Matrices (DBMs). A more efficient representation of the state-space for timed systems is based on the notion of zones ([Dil89],[AD94]). A clock zone φ is a conjunction of inequalities that compare either a clock value or the difference between two clock values to an integer. We allow inequalities of the following type: $x \prec c, c \prec x, x - y \prec c$ where $\prec \in \{<, \leq\}, x, y \in \mathcal{X}$, $\mathcal{X}$ is a set of clocks, and $c \in \mathbb{N}$. A key property of the set of clock zones is closure property under three operations—the intersection, time elapsing and clock reset.

Clock zones can be efficiently represented using matrices [Dil89]. In order to express a uniform notation for clock zones, we introduce a special clock x_0 that is always 0. Then, any clock zone φ can be written as a conjunction of constraints of the form $x - y \prec c$, for $x, y \in \mathcal{X}, \prec \in \{<, \leq\}$ and $n \in \mathbb{N}$.

Let $\mathcal{D}$ be a difference bound matrix representing clock zone φ. Each entry $\mathcal{D}_{i,j}$ is in the form $(d_{i,j}, \prec_{i,j})$ and represents the inequality $x_i - x_j \prec d_{i,j}$, where $\prec_{i,j}$ is either $<, \leq$ or $(\infty, <)$, if no such bound is known.

In this work we closely investigate parametric DBMs (PDBMs). Parametric DBMs are an extension of DBMs with parameters. We show later in the text that our proposed structure—a parametric hypercube—is more efficient to represent data for counter automata than PDBMs with respect to space and processors requirements.

Reachability analysis. The main effort on verification of timed systems has been put on safety properties that can be checked using reachability analysis by exploring the state space of timed automata. Symbolically, we can describe reachability analysis for timed automata with the following algorithm:

```
R := $S_0$                    // a set of reached states
F := $F_0$                    // a set of final states
while $((R \cap F) = \emptyset)$          // reached states are different from final
  R' := post(R)               // compute the set of successors of R
  if $(R' \subseteq R)$ return "No"   // the whole state space was reached
  R := R'                     // add successors to the set of all states
end while
return "Yes"                  // final state was reached
```

Reachability analysis consists of two basic steps: computing the set of successors of a set of reached states - a result of procedure *post()*, and searching for states that satisfy or contradict given properties - expressed by an intersection of a set of reached states and a set of final states. In figure above, the analysis ends if the final state is reached, or if the entire state space is generated without being intersected with a set of final states.

Symbolic reachability analysis. Symbolic reachability analysis is a powerful paradigm for verification of infinite-state systems, such as parameterized communicating systems [BCALS01]. Symbolic reachability analysis uses finite structures to represent infinite sets of configurations, and iterative exploration procedures to compute the set of all reachable configurations, or an upper approximation of this set. This technique is used for verification of infinite systems like time systems are.

Acceleration. Verification of an infinite state system can be enhanced by acceleration in order to help termination. Instead of repeating the same transition in the reachability graph we can replace these transitions by an acceleration step. The acceleration step corresponds to the computation of an upper approximation of the set (in some cases of an exact set) of reachable configurations by iterating a sequence of transitions an arbitrary number of times. For instance, starting with initial value x = 0, the iteration of a transition which increments x by 2 leads to the set of configurations $\{0, 2, 4, \ldots\}$ which can be represented by constraint $x = 2n$, with $n \geq 0$. Acceleration techniques allow us to compute a finite representation in one step instead of computing the infinite sequence of approximations $\{0\}, \{0, 2\}, \{0, 2, 4\}, \ldots$. As we will see in next sections acceleration plays an important role in parametric verification.

2.3 Timed Automata

Timed automata are finite-state machines with clocks which are used to constrain the accepting runs by imposing timing requirements on the transitions. While ordinary automata generate sequence of events (states), time automata are constrained by timing requirements and generate timed sequences. All clocks proceed at the same rate and measure the amount of time that has elapsed since they were started or reset. Each transition of the automaton may reset some of the clocks, and it puts certain constraints on the values of the clocks: a transition can be taken only if the current clock values satisfy the corresponding constraints.

Timed automata introduced by R. Alur and D. Dill in [AD94] serve as a technique for modelling finite state machines (FSM) with explicit time which is essential for specification and analysis of real-time systems. Here, we briefly show the theory of timed automata.

Timed Automata. To express system behaviour with timing constraints, we consider finite graphs augmented with a finite set of (real-valued) *clock*. The vertices of a graph are called *locations*, and the edges are called *switches*. While the switches are instantaneous, time can elapse in a location. A clock can be reset to zero simultaneously with any switch. At any instant, the reading of a clock equals the time elapsed since the last clock reset. With each switch we associate a clock constraint, and we require that the switch may be taken only if the current values of the clock satisfy the constraint. With each location we associate a clock constraint called an *invariant*, and we require that time can elapse in a location only if its invariant stays true.

Clock constraints and clock interpretation. For a set $\mathcal{X}$ of clocks, the set $\Phi(\mathcal{X})$ of *clock constraints* φ is defined by grammar

$$\varphi := x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi_1 \wedge \varphi_2,$$

where x is a clock in $\mathcal{X}$ and c is a constant in $\mathbb{Q}$. A *clock interpretation* ν for a set $\mathcal{X}$ of clocks assigns a real value to each clock; that is, it is a mapping from $\mathcal{X}$ to the set $\mathbb{R}^{\geq 0}$ of non-negative reals. For $\Delta \in \mathbb{R}, \nu + \Delta$ denotes the clock interpretation which maps every clock x to the value $\nu(x) + \Delta$. For $Y \subseteq \mathcal{X}, \nu[Y := 0]$ denotes the clock interpretation for $\mathcal{X}$ which assigns 0 to each $x \in Y$, and agrees with ν over the rest of the clocks.

Syntax. A *timed automaton* A is a tuple $\langle L, L_0, \Sigma, \mathcal{X}, I, E\rangle$, where

- L is a finite set of locations,
- L_0 is a finite set of initial locations,
- Σ is a finite set of labels,
- $\mathcal{X}$ is a finite set of clocks,
- I is a mapping that labels each location s with a clock constraint $\Phi(\mathcal{X})$, and
- $E \subseteq L \times \Sigma \times 2^{\mathcal{X}} \times \Phi(\mathcal{X}) \times L$ is a set of switches. A switch $e = \langle s, a, \varphi, \lambda, s'\rangle$ represents an edge from location s to location s' on symbol a. φ is a clock constraint over $\mathcal{X}$ that specifies when the switch is enabled, and the set $\lambda \subseteq \mathcal{X}$ gives the clocks to be reset with the switch.

In Figure 2.1, there is an example of a simple timed automaton where $L = \{s_0, s_1\}$, $L_0 = \{s_0\}$, $\Sigma = \{a, b\}$, $\mathcal{X} = \{x\}$, $I = \{s_1 \rightarrow x < 20\}$, and $E = \{(s_0, a, \epsilon, x, s_1), (s_0, b, x > 10, \epsilon, s_2)\}$.

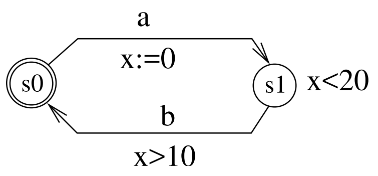


Figure 2.1: A simple timed automaton

Semantics. The semantics of timed automaton A is defined by a transition system $S_A = (\mathcal{Q}_A, \mathcal{Q}_0, \rightarrow)$ associated with it. A state q of S_A is a pair (s, ν) such that s is a location of A and ν is a clock interpretation for $\mathcal{X}$ such that ν satisfies the invariant $I(s)$. The set of all states of S_A is denoted $\mathcal{Q}_A$. There are two types of transitions in S_A:

1. **Time transition** $\xrightarrow{\Delta}$: for a state (s, ν) and a real-valued time increment $\Delta \geq 0$ there holds $(s, \nu) \xrightarrow{\Delta} (s, \nu + \Delta)$ if for all $0 \leq \Delta' \leq \Delta, \nu + \Delta'$ satisfies the invariant $I(s)$.

2. **Action transition** $\xrightarrow{a}$: for a state (s, ν) and a switch $\langle s, a, \varphi, \lambda, s' \rangle$ such that ν satisfies φ there holds $(s, \nu) \xrightarrow{a} (s', \nu[\lambda := 0])$ and ν satisfies $I(s')$.

A transition system of our example is depicted in Figure 2.2.

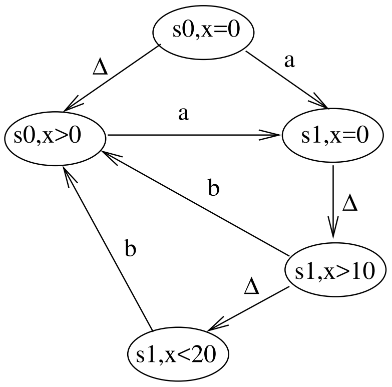


Figure 2.2: Semantics

Executions. An execution or run r of A is an infinite sequence of states and transitions:

$$r = q_0 \xrightarrow{l_0} q_1 \xrightarrow{l_1} \dots$$

where $i \in \mathbb{N}, q_i \in \mathcal{Q}_A$, and $l_i \in (\Sigma \cup \mathbb{R}^+)$. We denote by $\mathcal{R}_A(q)$ the set of runs starting at $q \in \mathcal{Q}_A$, and by $\mathcal{R}_A = \bigcup_{q \in \mathcal{Q}_A} \mathcal{R}_A(q)$ the set of runs of A.

A state q' is reachable from state q if it belongs to some run starting at q. We define $Reach_A(q)$ to be a set of states reachable from q:

$$Reach_A(q) = \{q' \in \mathcal{Q}_A \mid \exists r = q_0 \xrightarrow{l_0} q_1 \xrightarrow{l_1} \dots \xrightarrow{l_i} q_i \in \mathcal{R}_A(q), i \in \mathbb{N}\ ,\ q_i = q'\}.$$

Given a state $q \in \mathcal{Q}_A$ we are interested in computing the set of states $Reach_A(q)$.

Verification. The semantics of the timed automata—a timed transition system—is the basis for verification of timed automata. Model checking algorithms can be viewed as algorithms that search for particular states among all the possible states of the system. Conceptually, the search can be performed in either of two ways: forwards or backwards. Forward search consists in traversing the state-space by moving from one state to its successors and computing the set of reachable states $Reach_A(q_0)$. The backward search analyses the graph by exploring the predecessors of a state.

Having the set of all the possible states of the system we can analyse specified properties of the system. Properties can be expressed using a formula of the temporal logic. Verification means to check whether the property specified by the formula is satisfied over the state space. Methods for verification are out of scope of this work, so we recommend, for instance, [Yov98] for deep study of verification of timed systems.

2.4 Data Structures for Timed Automata

For representation of clock valuation of the transition system we can exploit different data structures. First data structure proposed by R.Alur and D.L.Dill [AD94] is based on regions. The more popular and efficient from the point of view of analysis is a formal data structure based on zones. Zones are represented using matrices—difference bound matrices (DBMs).

The core of my thesis is a design of a new data structure and operations over it. Thus, it is important to understand basic operations on other data structures like DBMs are. Similar operations like for DBMs were implemented in a new proposed data structure, as described later in this document.

2.4.1 Regions

Consider timed automaton A as a static model of a system and transition system S_A that represents behaviour of that system, as defined in the previous chapter. The transition system S_A is to be analysed using model checking. However, this system has infinitely many states and infinitely many symbols. The region analysis uses two techniques to change these infinite sets to finite ones [Alu99]:

- *time abstraction*

  Time abstraction forms a new time-abstract transition system where all time transitions are hidden. We get a new transition system U_A with finite number of symbols. In the reachability problem for timed automata, we wish to determine reachability of target locations. It follows that to solve reachability problems, we can consider the time-abstract transition system U_A instead of S_A.

- *stable quotient*

  To reduce infinitely many states to finite number R. Alur proposed equivalence relation over the state space Q_A called stable partition $\sim$. The quotient of U_A with respect to a stable partition is the transition system $[U_A]_\sim$. States of this system are the equivalence classes of $\sim$.

Then, we define an equivalence relation on the state space of an automaton $\cong$ that equates two states with the same location if they agree on the integral parts of all clock values and on the ordering of the fractional parts of all clock values. The integral parts of the clock values are needed to determine whether a particular clock constraint is met or not, whereas the ordering of the fractional parts is needed to decide which clock will change its integral part first.

Region Equivalence $\cong$. For a clock $x \in \mathcal{X}$, let c_x be a constant (maximal constant of clock x). For $d \in \mathbb{R}$, $fr(d)$ denotes the fractional part of d and $\lfloor d \rfloor$ denotes the integral part of d. Two clock assignments ν and ν' are region equivalent with respect to c_x, c_y, $\nu \cong \nu'$, iff all the following conditions hold:

1. For all $x \in \mathcal{X}$, either $\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor$ or both $\nu(x)$ and $\nu'(x)$ exceed c_x.
2. For all x, y, if $v(x) \leq c_x$ and $v(y) \leq c_y$, $fr(\nu(x)) \leq fr(\nu(y))$ iff $fr(\nu'(x)) \leq fr(\nu'(y))$.
3. For all $x \in \mathcal{X}$, if $\nu(x) \leq c_x$, then $fr(\nu(x)) = 0$ iff $fr(\nu'(x)) = 0$.

Region. An equivalence class of clock interpretation $[\nu]$ induced by $\cong$ is called a region.

For instance, consider a timed transition system with two clocks x and y with $c_x = 2$ and $c_y = 1$, where c_x, c_y are the largest constraints over clocks in the system. The clock regions are shown in Figure 2.3. The regions include 6 corner points, e.g. $(0, 1)$, 14 open line segments, e.g. $0 < x = y < 1$, and 8 open regions, e.g. $0 < x < y < 1$. The number of clock regions is exponential in the number of clocks and [Alu99].

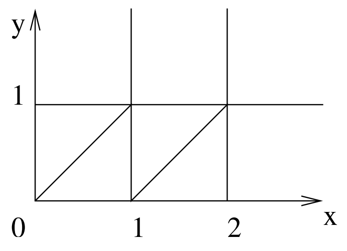


Figure 2.3: Clock regions

Reachability can be solved in time linear in the number of vertices and edges of the region automaton, which is linear in the number of locations, exponential in the number of clocks, and exponentional in the encodings of the constants. The reachability problem is PSPACE-complete.

2.4.2 Zones

Clock zones [Alu99] are a way how to obtain finite representation for the infinite state space S_A. Zones are clock constraints that can be represented and stored in memory as DBMs (Difference Bound Matrices). In a zone graph, instead of regions, zones are used to denote symbolic states. Their representation is more compact than regions.

A clock zone φ is a conjunction of inequalities that compare either a clock value or the difference between two clock values to an integer. We allow inequalities of the following type:

$$\boxed{x \prec c, c \prec x, x - y \prec c}$$

where $\prec \in \{<, \leq\}, x, y \in \mathcal{X}, c \in \mathbb{N}$.

If S_A has k clocks, then set φ is a convex set in the k-dimensional Euclidean space.

Operations over Zones

Every clock constraint used in invariant of an automaton location or in the guard of a transition is a clock zone (see definition above). They can be used as the basis for various state reachability analysis algorithms for timed automata.

The reachability analysis using zones is usually expressed in terms of the following three operations:

- **Intersection**
  For two clock zones φ and ψ, $\varphi \wedge \psi$ denotes the intersection of the two zones. Because φ and ψ are clock zones, they can be expressed as conjunctions of clock constraints.
- **Elapsing of Time**
  For a clock zone φ, $\varphi \Uparrow$ denotes the set of interpretations $\nu + \delta$ for $\nu \in \varphi$ and $\delta \in \mathbb{R}$.
- **Clock Reset**
  For a subset λ of clocks and a clock zone φ, $\varphi[\lambda := 0]$ denotes the set of clock interpretations $\nu[\lambda := 0]$ for $\nu \in \varphi$.

A key property of the set of clock zones is closure under the above three operations.

Zone Automata

Zone. A zone is a pair (s, φ), where s is a location and φ a clock zone. We build a transition system called *zone automaton* $\mathcal{Z}(A)$ whose states are zones of A. For every initial location s of A the zone $(s, [\mathcal{X} := 0])$ is an initial location of $\mathcal{Z}(A)$, and for every switch $e = (s, a, \psi, \lambda, s')$ of A and every clock zone φ, there is a transition $((s, \varphi), a, (s', succ(\varphi, e)))$.

Operation succ(). Model checking is based on computation of all possible states of the system and checking properties on these states. The core principle of the reachability analysis is to compute a set of successors of a current state. This is done through an operation succ(); in some papers it is called post() operation.

The set $succ(\varphi, e)$ can be computed using the three operations on clock zones as follows:

$$succ(\varphi, e) = (((\varphi \wedge I(s)) \Uparrow) \wedge I(s) \wedge \psi)[\lambda := 0]$$

This formula can be interpreted by following steps during analysis of a time automaton:

1. Intersect φ with the invariant of location s to find the set of possible clock assignments for the current state.
2. Let time elapse in location s using the operator $\Uparrow$.

3. Take the intersection with the invariant of location s again to find the set of clock assignments that still satisfy the invariant.

4. Take the intersection with the guard ψ of the transition e to find the clock assignments that are permitted by the transition.

5. Set all of the clocks in λ that are reset by the transition to 0.

Construction of a zone automaton starts from the initial zone (s, φ) by computing new clock zones $\varphi' = succ(\varphi, e)$ for every edge $e = (s, a, \psi, \lambda, s')$ starting in the state s of the timed automaton A. This new zone (s', φ') is then checked if it is already included in the automaton $\mathcal{Z}(A)$.

If the zone (s', φ') does not exists in $\mathcal{Z}(A)$, a new state (s', φ') is added to the graph $\mathcal{Z}(A)$ together with the edge $((s, \varphi), a, (s', \varphi')$. If the zone (s', φ') is already present in $\mathcal{Z}(A)$, only the edge $((s, \varphi), a, (s', \varphi'))$ is added to $\mathcal{Z}(A)$.

2.4.3 Difference Bound Matrix (DBM)

Clock zones can be efficiently represented using matrices [Dil89]. In order to express a uniform notation for clock zones we introduce a special clock x_0 that is always 0. Then, any clock zone φ can be written as a conjunction of constraints of the form $x - y \prec c$, for $x, y \in \mathcal{X}, \prec \in \{<, \leq\}$, and $n \in \mathbb{Z}$.

Let $\mathcal{D}$ is a difference bound matrix representing clock zone φ. Each entry $\mathcal{D}_{i,j}$ has the form $(d_{i,j}, \prec_{i,j})$ and represents the inequality $x_i - x_j \prec d_{i,j}$, where $\prec_{i,j}$ is either $<, \leq$ or $(\infty, <)$, if no such bound exists.

Since variable x_0 is always 0, it can be used for expressing constraints that only involve a single variable. Thus, $\mathcal{D}_{j,0} = (d_{j,0}, \prec)$ means $x_j \prec d_{j,0}$. Similarly, $\mathcal{D}_{0,j} = (d_{0,j}, \prec)$ means $-x_j \prec d_{0,j}$.

For instance, consider the clock zone:

$$\varphi = (x_1 - x_2 < 2) \wedge (0 < x_2 \leq 2) \wedge (1 \leq x_1)$$

That clock zone can be represented by the matrix $\mathcal{D}$:

$$\mathcal{D} = \left| \begin{array}{cccc} & x_0 & x_1 & x_2 \\ x_0 & (0, \leq) & (\mathbf{-1}, \leq) & (\mathbf{0}, <) \\ x_1 & (\infty, <) & (0, \leq) & (\mathbf{2}, <) \\ x_2 & (\mathbf{2}, \leq) & (\infty, <) & (0, \leq) \end{array} \right|$$

that encodes following inequalities:

$$\left| \begin{array}{ccc} (x_0 - x_0 \leq 0) & (\mathbf{x_0 - x_1 \leq -1}) & (\mathbf{x_0 - x_2 < 0}) \\ (x_1 - x_0 < \infty) & (x_1 - x_1 \leq 0) & (\mathbf{x_1 - x_2 < 2}) \\ (\mathbf{x_2 - x_0 \leq 2}) & (x_2 - x_1 < \infty) & (x_2 - x_2 \leq 0) \end{array} \right|$$

$$= \left| \begin{array}{ccc} (0 \leq 0) & (\mathbf{-x_1 \leq -1}) & (\mathbf{-x_2 < 0}) \\ (x_1 < \infty) & (0 \leq 0) & (\mathbf{x_1 - x_2 < 2}) \\ (\mathbf{x_2 \leq 2}) & (x_2 - x_1 < \infty) & (0 \leq 0) \end{array} \right|$$

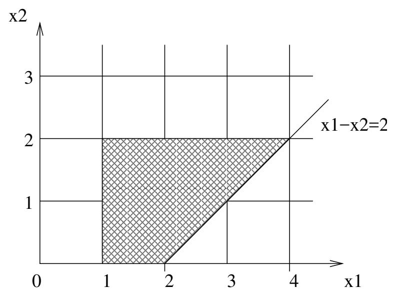


Figure 2.4: Clock zone $\varphi = (x_1 - x_2 < 2) \wedge (0 < x_2 \leq 2) \wedge (1 \leq x_1)$

In Figure 2.4 you can see the graphical representation of the zone φ.

The representation of a clock zone by a difference bound matrix is not unique. For instance, if we change $\mathcal{D}_{1,0}$ to $(4, <)$ we obtain an alternative DBM for the same zone, because $x_1 < 4$ includes $x_1 - x_2 < 2$ and $x_2 \leq 2$.

In general, the sum of upper bounds on the clock difference $x_i - x_j$ and $x_j - x_k$ is an upper bound on the clock difference $x_i - x_k$. As seen from previous example, the sum of $x_1 - x_2 < 2$ and $x_2 - x_0 \leq 2$ gives the upper bound 4 on $x_1 - x_0$, i.e., $x_1 - x_0 < 4$.

More formally, if $x_i - x_j \prec_{i,j} d_{i,j}$ and $x_j - x_k \prec_{j,k} d_{j,k}$, then

$$\boxed{x_i - x_k \prec'_{i,k} d'_{i,k}}$$

where

$$d'_{i,k} = d_{i,j} + d_{j,k} \text{ and } \prec'_{i,k} = \begin{cases} \leq & \prec_{i,j} = '\leq' \text{ and } \prec_{j,k} = '\leq' \\ < & \text{otherwise} \end{cases}$$

Finding minimum constraint $d'_{i,k}$ on each clock difference is called *tightening* the difference bound matrix. Deriving the tightest constraint on a pair of clocks is equivalent to finding the shortest path between two nodes. By repeating that operation on the matrix until the matrix is changed we get a *canonical* representation for the clock zone under consideration. Finding the canonical form can be done using the Floyd-Warshall algorithm which is of cubic complexity. Details about computation of canonical form can be found at [BY04, chapter 4].

Transformation operations over DBMs are defined with respect to operations on clock zones from the previous section as follows:

- **Intersection**

  The operation of intersection is used to add a constraint to a zone in state-space exploration.

  We define $\mathcal{D} = \mathcal{D}^1 \wedge \mathcal{D}^2$. Let $D^1_{i,j} = (c_1, \prec_1)$ and $\mathcal{D}^2_{i,j} = (c_2, \prec_2)$. Then $\mathcal{D}_{i,j} = (min(c_1, c_2), \prec)$, where $\prec$ is defined as follows:

  - If $c_1 < c_2$ then $' \prec' = ' \prec'_1$.

  - If $c_2 < c_1$ then $' \prec' =' \prec_2'$.
  - If $c_1 = c_2$ and $' \prec_1' =' \prec_2'$, then $' \prec' =' \prec_1'$.
  - If $c_1 = c_2$ and $' \prec_2' \neq' \prec_2'$, then $' \prec' =' <'$.

- **Elapsing of Time**
  This operation computes the strongest post condition of a zone with respect to delay, i.e., contains the time assignment that can be reached from $\mathcal{D}$ by delay.

  Algorithmically, the operation is computed by removing the upper bounds on all individual clocks. The property that all clocks proceed at the same speed is ensured by the fact that constraints on the differences between clocks are not altered by the operation.

  We define $\mathcal{D}' = \mathcal{D} \Uparrow$ as follows:

  - $\mathcal{D}'_{i,0} = (\infty, <)$ for any $i \neq 0$.
  - $\mathcal{D}'_{i,j} = \mathcal{D}_{i,j}$ if $i = 0$ or $j \neq 0$.

  Note: The complement operation for $pre()$ operation computes the weakest precondition of $\mathcal{D}$ with respect to delay. Formally, $\mathcal{D}' = \{u | u + d \in \mathcal{D}, d \in \mathbb{R}_+\}$, i.e., the set of time assignment that can reach $\mathcal{D}$ by some delay d.

  Algorithmically, this operation is computed by setting the lower bound on all individual clocks to $(0, \leq)$.

  This operation may produce non-canonical DBMs.

- **Clock Reset**
  This operation is used to set clocks to zero.

  We define $\mathcal{D}' = \mathcal{D}[\lambda := 0]$, where $\lambda \subseteq \mathcal{X}$ as follows:

  - If $x_i, x_j \in \lambda$ then $D'_{i,j} = (0, \leq)$.
  - If $x_i \in \lambda, x_j \notin \lambda$ then $\mathcal{D}'_{i,j} = \mathcal{D}_{0,j}$.
  - If $x_j \in \lambda, x_i \notin \lambda$ then $\mathcal{D}'_{i,j} = \mathcal{D}_{i,0}$.
  - If $x_i, x_j \notin \lambda$ then $\mathcal{D}'_{i,j} = \mathcal{D}_{i,j}$.

  Note: This operation can be extended to set clocks to any specific values, so that $\mathcal{D}' = \mathcal{D}[\lambda := m]$, where $m \in \mathbb{R}$.

- **Removing constraints**
  This operation removes all constraints on a given clock, i.e., the clock may take any positive value. Formally, $\mathcal{D}' = \{u[x = d] \mid u \in \mathcal{D}, d \in \mathbb{R}_+\}$. This operation is used in combination with conjunction to implement reset operations on clocks. It can be used in both forwards and backwards exploration. It is similar to *reset clocks* operation.

- **Copy**
  Copy is used in forward state-space exploration to copy the value of one clock to another. Formally, $\mathcal{D}' = \{u[x = u(y)] \mid u \in \mathcal{D}\}$. Copy can be implemented by assignment $\mathcal{D}_{x,y} = (0, \leq), \mathcal{D}_{y,z} = (0, \leq)$, which removes all other bounds on x.

- **Shift clocks**
  This reset operation is used for shifting a clock, i.e., adding or subtracting a clock with an integer value. Formally, $\mathcal{D}' = \{u[x = u(x) + m] \mid u \in D, m \in \mathbb{Z}\}$.

Additionally, we introduce other two classes of operation. One is the *property-checking* that includes checking the consistency of a DBM, the inclusion between zones, and whether a zone satisfies a given atomic constraint. The last class is *normalization* that is used to obtain a finite zone graph.

Property-checking operations on DBMs are as follows:

- **Consistency test**
  The most basic operation on a DBM is to check if it is consistent, i.e., if the solution set is non-empty. In state-space exploration this operation is used to remove inconsistent states from exploration.

  Inconsistent zone has at least one pair of clocks where the upper bound on their difference is smaller than the lower bound.

- **Inclusion test**
  Another key operation in state space exploration is inclusion checking for the solution sets of two zones. For DBMs in canonical form, the condition that $\mathcal{D}_{i,j} \leq \mathcal{D}'_{i,j}$ for all clocks $i, j \in \mathcal{X}$ is necessary and sufficient to conclude that $\mathcal{D} \subseteq \mathcal{D}'$.

Normalization operation is used to obtain a finite zone graph with respect to the maximal constant each clock is compared to in the automaton. Here we describe the normalization operation for automata that contain no constraints over clock differences. For the general case, we refer to [BY03].

Given a clock zone $\mathcal{D}$ and a set of maximal constants $k = \{k_x, k_y, \dots\}$, where k_x denotes the maximal constant for clock x, the normalized zone of $\mathcal{D}$ is computed as follows:

1. Removing all constraints of the form $x \prec m, x - y \prec m$, where $m > k_x, \prec \in \{<, \leq\}$.

2. Replacing all constraints of the form $x \succ m, x - y \succ m$, where $m > k_x, \succ \in \{>, \geq\}$ with $x > k_x$ and $x - y > k_x$, respectively.

2.4.4 Example

Consider a simple timed system: a light switch with delays. To switch the light on it is needed to press twice the switch within ten seconds. Then the light shines bright until the maximum time delay passes. Then the light is automatically switched off. If the switch

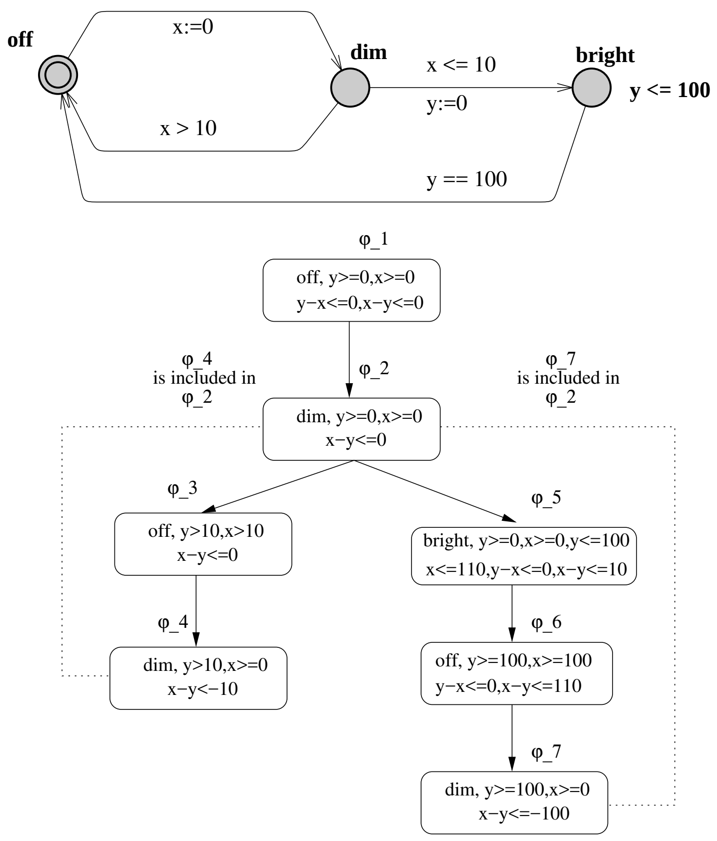


Figure 2.5: A light switch—timed automaton A and its transition system S_A

button is pressed just once, the light is dim and after next press the button is switched off. The system can be modelled by timed automaton A—see Figure 2.5.

Behaviour of the system can be described using transition system S_A. This graph contains nodes in the form (s, ν) where s is the location of the automaton S and ν is a clock valuation for $\mathcal{X}$. Clock valuation is represented using zones. The transition system S_A in Figure 2.5 represents states generated using post() analysis. Normal lines means transition of a S_A, dotted lines depicts relation inclusion. For example, the state $(\text{off}, \varphi_4) \subseteq (\text{off}, \varphi_2)$. These two states have the same location s (in our case "off"), and $\varphi_4 \subseteq \varphi_2$.

Two zones with the same location and witch clock zones φ that are included are symbolically represented as one zone. The graph corresponds to the zone automaton $\mathcal{Z}(A)$. Graphical representation of clock zones $\varphi_1 - \varphi_7$ is in Figure 2.6.

Clock zones are represented in the form of difference bound matrices (DBMs) as described in the previous section. For our system $\mathcal{Z}(A)$ is represented by the following matrices:

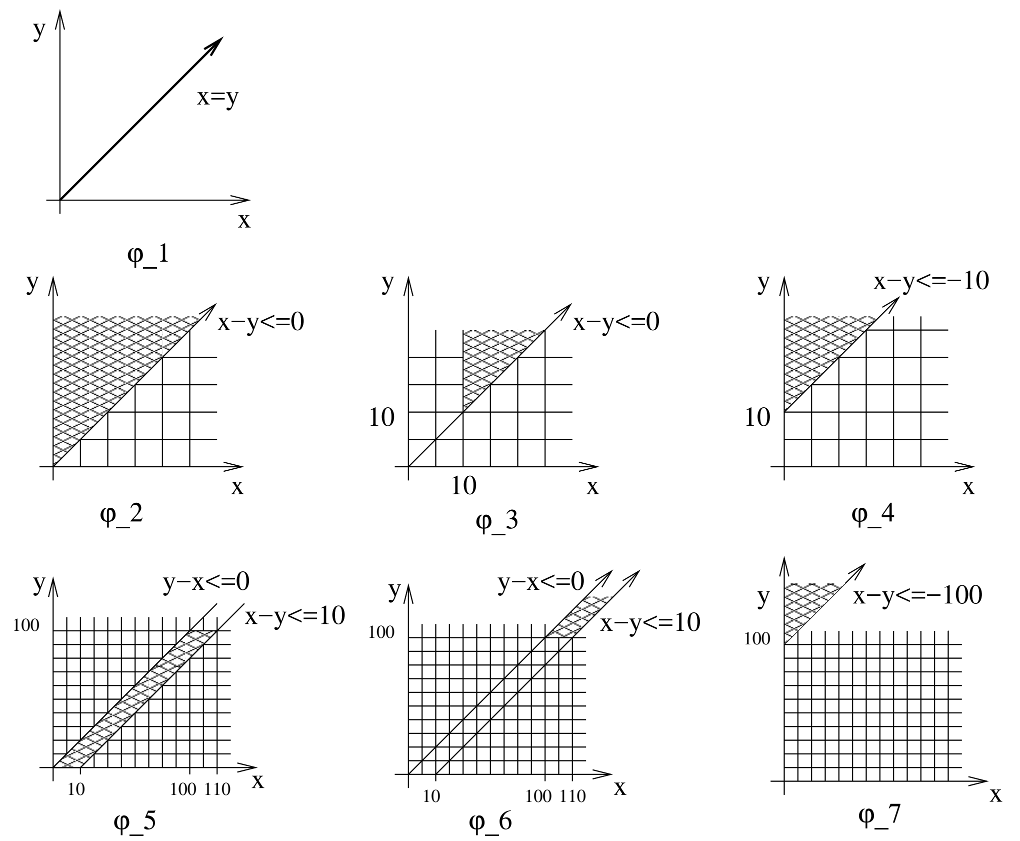


Figure 2.6: A graphical representation of clock zones

$$\varphi_1 = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0,\leq) & (0,\leq) & (0,\leq) \\ x & (\infty,<) & (0,\leq) & (0,\leq) \\ y & (\infty,<) & (0,\leq) & (0,\leq) \end{array}\right), \varphi_2 = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0,\leq) & (0,\leq) & (0,\leq) \\ x & (\infty,<) & (0,\leq) & (0,\leq) \\ y & (\infty,<) & (\infty,<) & (0,\leq) \end{array}\right)$$

$$\varphi_3 = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0,\leq) & (-10,<) & (-10,<) \\ x & (\infty,<) & (0,\leq) & (0,\leq) \\ y & (\infty,<) & (\infty,<) & (0,\leq) \end{array}\right), \varphi_4 = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0,\leq) & (0,\leq) & (-10,<) \\ x & (\infty,<) & (0,\leq) & (-10,<) \\ y & (\infty,<) & (\infty,<) & (0,\leq) \end{array}\right)$$

$$\varphi_5 = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0,\leq) & (0,\leq) & (0,\leq) \\ x & (110,\leq) & (0,\leq) & (10,\leq) \\ y & (100,\leq) & (0,\leq) & (0,\leq) \end{array}\right), \varphi_6 = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0,\leq) & (-100,\leq) & (-100,\leq) \\ x & (\infty,<) & (0,\leq) & (110,<) \\ y & (\infty,<) & (0,\leq) & (0,\leq) \end{array}\right)$$

$$\varphi_7 = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0,\leq) & (0,\leq) & (-100,\leq) \\ x & (\infty,<) & (0,\leq) & (-100,\leq) \\ y & (\infty,<) & (\infty,<) & (0,\leq) \end{array}\right)$$

Chapter 3

Parametric Real-Time Reasoning

Parametric analysis works with systems that contain special variables that are not changed during the execution—parameters. In parametric models, clocks and counters can be compared with parameters. Parameters are used in transitions where they define lower and upper bounds on clocks or counters. Parameters may range over infinite domains and are related by a set of constraints. For instance, we say variable $x < MAXDELAY$ where $MAXDELAY$ is a parameter. This parameter can be constrained by a relation $MAXDELAY >= 1$.

Using parametric reasoning, we can either verify that a system satisfies some property for all possible values of the parameters, or we can find constraints on the parameters that define the set of all possible values for which the system satisfies a property.

As mentioned in [AHV93], the important question for parametric automata is the emptiness: given a parametric timed system, are there concrete values for the parameters so that the automaton has an accepting run? This question is generally undecidable but there exist algorithms for checking the emptiness of restricted classes of parametric timed automata. Using semi-algorithms the termination is not ensured, however, in many practical cases the analysis terminates. There are various techniques that enforce the convergence of the analysis like widening technique or convex hull [ACH$^+$95] implemented in HyTech, or extrapolation based on control loops introduced in [AAB00] and implemented in TReX.

As showed in [AD94], the verification problem for timed automata is decidable. It has been shown in [AHV93] that for parametric timed automata, the reachability problem is undecidable. However, in [AAB00], the authors propose a semi-algorithmic approach that allows to deal with parametric counter and timed systems. They define a new symbolic representation called Parametric DBMs (PDBMs) for use in reachability analysis, and provide powerful technique for computing representations of their sets of reachable configuration.

Another approach to parametric time automata can be found in [HRSV01]. The authors investigate symbolic representation of parametric timed automata. They also present a subclass of parametric timed automata (L/U automata) for which the emptiness problem is decidable.

In our text, this chapter gives an overview of parametric timed and counter systems as defined in [AAB00] and their representation using PDBMs. It explains main points of parametric reasoning and points out the significance of a symbolic data representation. We show two data structures mainly used to represent abstract data domain for parametric verification—Parametric DBMs and polyhedra. We focus on operations over these structures and compare them with operations on DBMs (non-parametric data structure). We discuss efficiency of implementation of PDBMs and polyhedra in the last section of this chapter.

3.1 Parametric Timed Automata

Classical timed automata where clocks can only be compared to constants do not allow parametric reasoning. Here, we consider models of extended automata supplied with real valued clocks. Clocks can be reset using simple operations, and they can be tested using simple parametric constraints (they can be compared to parameters in these constraints). The following definition of parametric timed and counter automata were first introduced in [AAB00].

A Parametric Timed Automaton (PTA) is a tuple $\mathcal{T} = \langle L, L_0, \mathcal{X}, \mathcal{P}, I, \delta \rangle$, where

- L is a finite set of locations,
- L_0 is a finite set of initial locations, $L_0 \subseteq L$,
- $\mathcal{X} = \{x_0, \dots, x_{n_x-1}\}$ is a finite set of clocks, each $x_i \in \mathcal{X}$ is interpreted over domain $\mathcal{D}$,
- $\mathcal{P} = \{p_0, \dots, p_{n_p-1}\}$ is a finite set of parameters, each $p_i \in \mathcal{P}$ is interpreted over domain $\mathcal{D}$,
- $I : L \to SC(\mathcal{X}, \mathcal{P})$ is a mapping that associates locations with invariants, $SC(\mathcal{X}, \mathcal{P})$ is a simple parametric constraint expressed as a conjunction of formulas of the form $x \prec t$ or $x - y \prec t$ where $x, y \in \mathcal{X}, \prec \in \{<, \leq\}$, $t \in AT(\mathcal{P})$ is an arithmetical term over $\mathcal{P}$ defined by the grammar $t ::= c \mid p \mid t - t \mid t + t \mid c * t$ where $p \in P$ is a parameter and $c \in \mathbb{Z}$ is a constant.
- δ is a set of transitions of the form (s_1, g, sop, s_2) where $s_1, s_2 \in L, g \in SC(\mathcal{X}, \mathcal{P})$ is a guard, and sop is a simple operation over $\mathcal{X}$—a special kind of assignment of the form $x := 0$ or $x := t$ where $x \in \mathcal{X}$ and $t \in AT(\mathcal{P})$

Clocks and parameters range over a set $\mathcal{D}$ which can be either the set of positive reals $\mathbb{R}^{\geq 0}$ (dense model time) or the set of non-negative integers $\mathbb{N}$ (discrete time model). We call valuation of clocks a vector $\vec{\nu} = (\nu_1, \dots, \nu_n)$, where $\nu_i \in \mathcal{D}$. Value $\vec{\nu}(x_i)$ denotes the value of variable x_i in the valuation $\vec{\nu}$.

A configuration of $\mathcal{T}$ is a triplet $\langle s, \vec{\nu}, \vec{\gamma} \rangle$ where $s \in L$ is a location, $\vec{\nu} : \mathcal{X}^{n_x} \to \mathcal{D}^{n_x}$ is a valuation of the clocks, and $\vec{\gamma} : \mathcal{P}^{n_p} \to \mathcal{D}^{n_p}$ is a valuation of parameters.

In Figure 3.1, there is an example of a parametric timed automaton where $L = \{s_0, s_1\}$, $L_0 = \{s_0\}$, $\mathcal{X} = \{x\}$, $I = \{s_1 \to x < p\}$, and $\delta = \{(s_0, \epsilon, x := 0, s_1), (s_0, x \geq q, \epsilon, s_2)\}$.

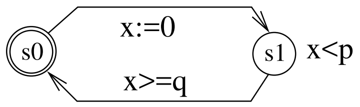


Figure 3.1: A parametric timed automaton

Semantics. Similarly to the timed automata, the semantics of PTA is a transition system $S_\mathcal{T} = (Q_\mathcal{T}, Q_0, \to)$ associated with PTA $\mathcal{T}$ where $Q_\mathcal{T} = L \times \mathcal{D}^{n_x} \times \mathcal{P}^{n_p}$ is a set of possible configurations of $\mathcal{T}$, Q_0 is a set of initial configurations, and $\to$: $Q_\mathcal{T} \times \delta \times Q_\mathcal{T}$ is a set of transitions of $S_\mathcal{T}$.

A state $q \in Q_\mathcal{T}$ is a tuple $(s, \vec{\nu}, \vec{\gamma})$, where $s \in L$, $\vec{\nu}$ is a clock interpretation for $\mathcal{X}$, and $\vec{\gamma}$ is a parameter valuation such that $(\vec{\nu}, \vec{\gamma}) \models I(s)$. We denote $Q_\mathcal{T}$ the set of all states of $S_\mathcal{T}$. There are two types of transitions $\to$:

1. **Time transition** $\xrightarrow{\Delta}$: $\langle s_1, \vec{\nu_1}, \vec{\gamma_1}\rangle \to \langle s_2, \vec{\nu_2}, \vec{\gamma_2}\rangle$ iff $s_1 = s_2$, $\vec{\gamma_1} = \vec{\gamma_2}$ and exists $\Delta \in \mathcal{D}$ such that $\vec{\nu_2} = \vec{\nu_1} + \Delta$ and for all $\Delta' \leq \Delta : (\vec{\nu_1} + \Delta', \vec{\gamma_1}) \models I(s_1)$.

2. **Action transition** $\xrightarrow{a}$: for a state $\langle s_1, \vec{\nu_1}, \vec{\gamma_1}\rangle$ and a transition $a = (s_1, g, sop, s_2) \in \delta$ we define a transition relation $\xrightarrow{a}$ between configurations as $\langle s_1, \vec{\nu_1}, \vec{\gamma_1}\rangle \xrightarrow{a} \langle s_2, \vec{\nu_2}, \vec{\gamma_2}\rangle$ such that $(\vec{\nu_1}, \vec{\gamma_1}) \models g$ and $\vec{\nu_2} = sop(\vec{\nu_1}) \wedge \vec{\gamma_1} = \vec{\gamma_2}$ and $(\vec{\nu_2}, \vec{\gamma_1}) \models I(s_2)$.

A graph of symbolic configurations of our example is depicted in Figure 3.2. In addition to timed automata defined in the previous chapter, each state of the transition system contains a valuation of parameters in the form of a quantifier-free formula over $\mathcal{P}$, $F(\mathcal{P})$, by grammar $\varphi ::= \mathit{true} \mid t \leq t \mid \neg\varphi \mid \varphi \wedge \varphi$, where $t \in AT(\mathcal{P})$. In our figure, operation $\wedge$ is denoted by comma, semicolon separates items of the tuple. In the initial state s_0 parameters are not bounded (constrained), so the formula is *true*.

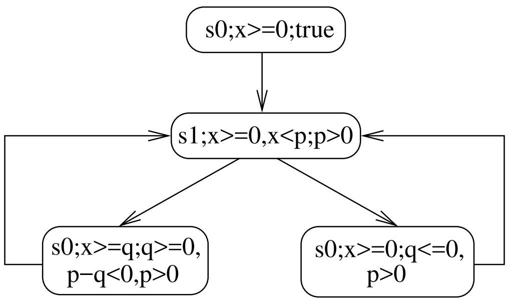


Figure 3.2: Semantics of the parametric timed automaton

Executions. An execution or run r of $\mathcal{T}$ is an infinite sequence of states and transitions:

$$r = q_0 \xrightarrow{l_0} q_1 \xrightarrow{l_1} \dots$$

where $i \in \mathbb{N}, q_i \in Q_{\mathcal{T}}$, and $l_i \in (\Sigma \cup \mathcal{D} \cup \mathcal{D})$. We denote by $\mathcal{R}_{\mathcal{T}}(q)$ the set of runs starting at $q \in Q_{\mathcal{T}}$, and by $\mathcal{R}_{\mathcal{T}} = \bigcup_{q \in Q_{\mathcal{T}}} \mathcal{R}_{\mathcal{T}}(q)$ the set of runs of $\mathcal{T}$.

A state q' is reachable from state q if it belongs to some run starting at q. We define $Reach_{\mathcal{T}}(q)$ to be a set of states reachable from q:

$$Reach_{\mathcal{T}}(q) = \{q' \in Q_{\mathcal{T}} \mid \exists r = q_0 \xrightarrow{l_0} q_1 \xrightarrow{l_1} \dots \xrightarrow{l_i} q_i \in \mathcal{R}_{\mathcal{T}}(q), i \in \mathbb{N}\ ,\ q_i = q'\}.$$

For verification we are interested in computing the set of states $Reach_{\mathcal{T}}(q_0)$.

3.2 Parametric Counter Automata

Parametric counter automata are similar to parametric timed automata. The only difference is that the function *post*() is defined without considering time-transitions. Variables (counters) and parameters are interpreted over naturals, $\mathcal{D} = \mathbb{N}$.

A Parametric Counter Automaton (PCA) is a tuple $\mathcal{C} = \langle L, L_0, C, P, I, \delta\rangle$, where

- L is a finite set of locations,
- L_0 is a finite set of initial locations, $L_0 \subseteq L$,
- C is a finite set of variables (counters) over integers,
- $\mathcal{P}$ is a finite set of parameters,
- $I : L \to SC(C, \mathcal{P})$ is a mapping that associates locations with invariants, $SC(C, \mathcal{P})$ is a simple parametric constraint expressed as a conjunction of formulas of the form $x \prec t$ or $x - y \prec t$ where $x, y \in C, \prec \in \{<, \leq\}$, t is an arithmetical linear term $AT(\mathcal{P})$ over the $\mathcal{P}$ defined by the grammar $t ::= c \mid p \mid t - t \mid t + t \mid c * t$ where $p \in \mathcal{P}$ is a parameter and $c \in \mathbb{Z}$ is a constant.
- δ is a set of transitions of the form (s_1, g, sop, s_2) where $s_1, s_2 \in L, g \in SC(C, \mathcal{P})$ is a guard, and *sop* is a simple operation over C—a special kind of assignment of the form $x := y + t$ or $x := t$ where $x, y \in C$ and $t \in AT(\mathcal{P})$.

Semantics. Semantics of PCA is a transition system $S_{\mathcal{C}} = (Q_{\mathcal{C}}, Q_0, \to)$ associated with PCA $\mathcal{C}$. A state q of $S_{\mathcal{C}}$ is a tuple $(s, \vec{\nu}, \vec{\gamma})$, where $s \in L$, $\vec{\nu} : C^{n_c} \to \mathcal{D}^{n_c}$ is a valuation of counters, and $\vec{\gamma} : \mathcal{P}^{n_p} \to \mathcal{D}^{n_p}$ is a valuation of parameters such that $(\vec{\nu}, \vec{\gamma}) \models I(s)$. We denote $Q_{\mathcal{C}}$ a set of all states of $S_{\mathcal{C}}$.

The set $\mathcal{D}$ denotes a domain of counter values, $\mathcal{D}$ is $\mathbb{N}$. Given a transition $a \in \delta$, we define an action relation $\xrightarrow{a}$: For a state $\langle s_1, \vec{\nu_1}, \vec{\gamma_1}\rangle$ and a transition $a = (s_1, g, sop, s_2) \in \delta$, we define a transition relation $\xrightarrow{a}$ between configurations as $\langle s_1, \vec{\nu_1}, \vec{\gamma_1}\rangle \xrightarrow{a} \langle s_2, \vec{\nu_2}, \vec{\gamma_2}\rangle$ such that $(\vec{\nu_1}, \vec{\gamma_1})$ satisfies g, and $\vec{\nu_2} = sop(\vec{\nu_1}) \wedge \vec{\gamma_1} = \vec{\gamma_2}$, and $(\vec{\nu_2}, \vec{\gamma_1}) \models I(s_2)$. The function $post_a$ is defined here without considering time-transitions.

3.3 Comparison of Parametric Timed and Counter Automata

The main difference between Parametric Timed Automata and Parametric Counter Automata is whether a parameter can be a clock variable or not. In Parametric Timed Automata parameters can be clock variables only. PTA define two types of transitions—action transitions and time transitions.

Parametric Counter Automata allow parameters only for non-clock variables that we call counters here. PCA define only one type of transitions—action transition.

However, it is possible to combine these two parametric automaton into one heterogenous systems. In such system we can have different types of variables defined over different domains. However, operations for every type of variables are defined separately.

In [AAB00] such model is called Parametric Timed Counter System (PTCS) that has both counters and clocks. PTCS is a straightforward extension of the definition of the PTA's and PCS's. It is defined as a tuple $\mathcal{M} = (L, C, \mathcal{X}, \mathcal{P}, I, \delta)$, where C is a finite set of counters, $\mathcal{X}$ is a finite set of clocks and $\mathcal{P}$ is a finite set of parameters. In this model comparisons between clocks and counters in guards and invariants are not allowed.

In both models—PCA and PTA—we can use Parametric DBMs for data representation. However, it is quite possible to have different data structures for clocks and counters. While PTA use mostly PDBMs or polyhedra, for PCA we can use simpler data structure. The design and implementation of such structure—a parametric hypercube—is one of the objectives of this work. This data structure is described later in the text.

3.4 Verification

In automata-theoretic verification, a state-space of the system is viewed as an automaton. The set of words accepted by the automaton corresponds to possible behaviours (runs) of the system. Properties of the system are specified by automaton too. Considering parametric systems we distinguish two goals of verification—parametric verification and parameter synthesis:

- For verification of parametric systems, we want to prove that a system satisfies its specification for all parameters values that meet a given set of constraints. Given a set $\Gamma \subseteq [\mathcal{P} \mapsto \mathcal{D}]$ of possible parameter valuations, we wish to verify that no $\gamma \in \Gamma$ breaks the requirements put on the system.
- In parameter synthesis, we want to find all parameter valuations $\Gamma(A)$ that satisfy desired properties of the system.

In parameterized models we represent states symbolically and support the same operations as for non-parameterized models. A set of states can be efficiently represented using matrices for non-parametric times systems. In the parameterized case, we use constrained parametric DBMs.

3.5 Parametric DBMs

Parametric Difference Bound Matrix (PDBM) is a symbolic data structure that is used to represent valuation of variables during system analysis. PDBMs—in comparison to DBMs—were designed to work with parameters. PDBMs were successfully implemented in verification tool TREX [BCAS01].

Parametric DBMs were first introduced in [AAB00]. Here, we recall its definition and show basic operations over PDBMs. For further details, we recommend [AC01].

Parametric DBMs use for its representation parameterized bounds. Let us show the definition of parameterized bounds for PDBMs. Its definition is then applied to parameterized hypercubes defined in chapter 4.1.3.

Parameterized bounds. Let $b = (t, \prec)$ be a pair of an arithmetical term with parameters $t \in AT(\mathcal{P})$ with a symbol from $\{<, \leq\}$. The set of *parameterized bounds* $\mathcal{PB}$ is defined as follows:

$$\mathcal{PB} = AT(\mathcal{P}) \times \{<, \leq\} \ \cup \ \{(-\infty, <), (\infty, <)\}.$$

In order to limit the set of values taken by parameters, we define the notion of *a constrained parameterized bound*—a parameterized bound with a constraint (formula) $F(\mathcal{P})$ that should be satisfied by the parameters. $F(\mathcal{P})$ is a set of quantifier-free formulas over $\mathcal{P}$ given by the grammar $\varphi ::= t \leq t \mid \neg\varphi \mid \varphi \wedge \varphi$ where $t \in AT(\mathcal{P})$.

The set of constrained parameterized bounds $\tilde{\mathcal{PB}}$ over $\mathcal{D}$ is defined as follows:

$$\tilde{\mathcal{PB}} = \mathcal{PB} \times F(\mathcal{P})$$

$\tilde{\mathcal{PB}}$ is defined on the domain of integers or reals.

Symbols $<$ and $\leq$ are totally ordered: $<$ is strictly less then $\leq$.

Total order over parameterized bounds $\subseteq^{\mathcal{PB}}$. We use the relation of total order $\subseteq^{\mathcal{PB}}$ over constrained parameterized bounds in the following way:

Let $\tilde{b_1} = ((t_1, \prec_1), \varphi_1)$, $\tilde{b_2} = ((t_2, \prec_2), \varphi_2) \in \tilde{\mathcal{PB}}$ are two constrained parameterized bounds. We say that $\tilde{b_1} \subseteq^{\mathcal{PB}} \tilde{b_2}$ if and only if the formula φ_{incl} defined below is satisfiable.

$$\varphi_{incl} = \forall p_i \in \mathcal{P} \ . \ \varphi_1 \Rightarrow \varphi_2 \ \wedge \ ((t_1 < t_2) \ \vee \ (t_1 = t_2 \wedge \prec_1 \leq \prec_2))$$

The bound $(\infty, <)$ satisfies all $\tilde{b} \in \mathcal{PB}$, i.e., $\tilde{b} \subseteq^{\mathcal{PB}} ((\infty, <), true)$ for any $\tilde{b} \in \mathcal{PB}$.

Definition of the strict order $\subset^{\mathcal{PB}}$ is similar. We say that $\tilde{b_1} \subset^{\mathcal{PB}} \tilde{b_2}$ if and only if the formula φ_{incls} given bellow is satisfiable.

$$\varphi_{incls} = \forall p_i \in \mathcal{P} \ . \ \varphi_1 \Rightarrow \varphi_2 \ \wedge \ ((t_1 < t_2) \ \vee \ (t_1 = t_2 \wedge \prec_1 < \prec_2))$$

Bounds $\tilde{b_1}$ and $\tilde{b_2}$ are equal if and only if $\tilde{b_1} \subseteq^{\mathcal{PB}} \tilde{b_2}$ and $\tilde{b_2} \subseteq^{\mathcal{PB}} \tilde{b_1}$.

Operators $\oplus, \otimes$. The following definition introduces operators $\oplus$ and $\otimes$. Operations $\oplus$ is needed for canonization and computation of the normal form, operation $\otimes$ for intersection over PDBMs. The satisfiability of parameterized formulas $F(\mathcal{P})$ plays an important role in the computation of these operations.

Let $\tilde{b_1} = ((t_1, \prec_1), \varphi_1)$, $\tilde{b_2} = ((t_2, \prec_2), \varphi_2) \in \tilde{\mathcal{PB}}$ are two constraint parameterized bounds. We define following operators according to [AC01]:

- Operator $\oplus : \tilde{\mathcal{PB}} \times \tilde{\mathcal{PB}} \rightarrow \tilde{\mathcal{PB}}$ such that

  $$\tilde{b_1} \oplus \tilde{b_2} = (t_1 + t_2, (min(\prec_1, \prec_2)), \varphi_1 \ \wedge \ \varphi_2)$$

  where for all $t \in AT(\mathcal{P})$ we define:

  $$\begin{array}{rcl} t + \infty & = & \infty \\ t + (-\infty) & = & -\infty \\ \infty + \infty & = & \infty \\ \infty + (-\infty) & = & \infty \\ (-\infty) + (-\infty) & = & -\infty \end{array}$$

  There are two special bounds:

  - Bound $((0, \leq), true)$ is a neutral element for $\oplus$. For all $\tilde{b} \in \tilde{\mathcal{PB}}, ((0, \leq), true) \oplus \tilde{b} = \tilde{b}$.
  - Bound $((\infty, <), false)$ is an absorbing element for $\oplus$. For all $\tilde{b} \in \tilde{\mathcal{PB}}, ((\infty, <), false) \oplus \tilde{b} = ((\infty, <), false)$.

  Operator $\oplus$ is needed when transforming PDBMs into canonical forms and for computation of normal forms.

- Operator $\otimes$.

  Before defining operator $\otimes$ used in intersection of PDBMs we have to determine the minimum between two terms. We will use following three formulas:

  $$\begin{array}{rcl} \Phi_< & \equiv & \exists p \in \mathcal{P}.\varphi_1 \wedge \varphi_2 \wedge t_1 < t_2 \\ \Phi_= & \equiv & \exists p \in \mathcal{P}.\varphi_1 \wedge \varphi_2 \wedge t_1 = t_2 \\ \Phi_> & \equiv & \exists p \in \mathcal{P}.\varphi_1 \wedge \varphi_2 \wedge t_1 > t_2 \end{array}$$

Operator $\otimes : \tilde{\mathcal{PB}} \times \tilde{\mathcal{PB}} \rightarrow 2^{\tilde{\mathcal{PB}}}$ is such that $\tilde{b_1} \otimes \tilde{b_2} = min(\tilde{b_1}, \tilde{b_2})$.

$$\begin{array}{rcl} min(\tilde{b_1}, \tilde{b_2}) & = & min_{<}(\tilde{b_1}, \tilde{b_2}, \Phi_{<}) \\ & \cup & min_{=}(\tilde{b_1}, \tilde{b_2}, \Phi_{=}) \\ & \cup & min_{>}(\tilde{b_1}, \tilde{b_2}, \Phi_{>}) \\ \text{where} & & \\ min_{<}(\tilde{b_1}, \tilde{b_2}, \Phi_{<}) & = & \begin{cases} \{((t_1, \prec_1), \varphi_1 \wedge \varphi_2 \wedge (t_1 < t_2))\} & \text{if } \Phi_{<} \\ \emptyset & \text{otherwise} \end{cases} \\ min_{=}(\tilde{b_1}, \tilde{b_2}, \Phi_{=}) & = & \begin{cases} \{(t_1, \prec_1), \varphi_1 \wedge \varphi_2 \wedge (t_1 = t_2))\} & \text{if } \Phi_{=} \wedge \prec_1 \leq \prec_2 \\ \{(t_2, \prec_2), \varphi_1 \wedge \varphi_2 \wedge (t_1 = t_2))\} & \text{if } \Phi_{=} \wedge \prec_2 < \prec_1 \\ \emptyset & \text{otherwise} \end{cases} \\ min_{>}(\tilde{b_1}, \tilde{b_2}, \Phi_{>}) & = & \begin{cases} \{((t_2, \prec_2), \varphi_1 \wedge \varphi_2 \wedge (t_1 > t_2))\} & \text{if } \Phi_{>} \\ \emptyset & \text{otherwise} \end{cases} \end{array}$$

The result of *min* operation may be a set of one, two or three constrained parameterized bounds. Their combination depends on satisfiability of Φ_1, Φ_2 and Φ_3.

There is as well a special bound:

– Bound $((\infty, <), true)$ is a neutral element for $\otimes$. For all $\tilde{b} \in \tilde{\mathcal{PB}}$, $((\infty, <), true) \otimes \tilde{b} = \{\tilde{b}\}$.

Parametric DBM $\mathcal{M}$. Let $\mathcal{T} = \langle L, L_0, \mathcal{X}, \mathcal{P}, I, \delta \rangle$ be a PTA, $\mathcal{X} = \{x_1, \ldots, x_n\} \cup \{x_0\}$ a set of its clocks, where x_0 is an additional clock with a special value 0. Using this special clock we can express the condition $x_i \leq t_i$ as $x_i - x_0 \leq t_i$.

Then, we define a parametric difference bound matrix (PDBM) $\mathcal{M}$ as a matrix that encodes the constraints in form $x_i - x_j \prec t$ (similar to DBMs) where $x_i, x_j \in \mathcal{X}$ are clocks, $t \in AT(\mathcal{P})$, and $\prec \in \{<, \leq\}$ such that:

- $\mathcal{M}$ is a square matrix of dimension $n + 1$ where n is the number of clock variables.
- Elements of $\mathcal{M}$ are parameterized bounds in the form $(t, \prec) \in \mathcal{PB}$ encoding constraints $x_i - x_j \prec t$.
- The first column encodes the upper bounds of the variables, the first row encodes the lower bounds of the variables.
- Element $(\infty, <)$ means that interval $x_i - x_j$ is unbounded (unrestricted).

The following parameterized matrix is taken from the example at page 33.

$$\mathcal{M} = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0, \leq) & (-10, <) & (-10, <) \\ x & (\infty, <) & (0, \leq) & (0, \leq) \\ y & (\infty, <) & (\infty, <) & (0, \leq) \end{array} \right)$$

Valuation of a PDBM $[\![\mathcal{M}]\!]$ is a set of valuations of counters, clocks and parameters $(\vec{\nu}, \vec{\gamma})$ that satisfy a formula ψ given bellow:

$$\psi = \wedge_{i=0}^{n_x} \wedge_{j=0}^{n_x} x_i - x_j \prec_{ij} t_{ij} \ \wedge \ \varphi,$$

for all variables $x_i, x_j \in \mathcal{X}$, where $M(i,j) = (\prec_{ij}, t_{ij})$. We say that $\mathcal{M}$ is *satisfiable* for parametric valuation $(\vec{\nu}, \vec{\gamma})$ if $[\![\mathcal{M}]\!]$ is nonempty.

Constrained PDBM $\tilde{\mathcal{M}}$. A Constrained PDBM is a pair $\tilde{\mathcal{M}} = (\mathcal{M}, \Phi)$ where $\mathcal{M}$ is a PDBM and $\Phi \in F(\mathcal{P})$ is a parameter constraint. $F(\mathcal{P})$ is a set of quantifier-free first order formulas given by grammar $\varphi ::= true \mid t \leq t \mid \neg\varphi \mid \varphi \ \wedge \ \varphi$. A symbolic configuration of a $\mathcal{T}$ automaton is a tuple $(q, \tilde{\mathcal{M}})$, where $q \in Q_{\mathcal{T}}$ is a control state (see Semantics of PTA at page 21), and $\tilde{\mathcal{M}}$ is a Constrained PDBM.

Example of a constrained PDBM that encodes following inequalities $\tilde{\mathcal{M}} = ((y >= 0) \wedge (x >= 0) \wedge (y \leq Max) \wedge (y - x \leq 0) \wedge (x \leq 10 + Max) \wedge (x - y) \leq 10), Max > 0)$ is depicted in Figure 3.3.

$$\tilde{\mathcal{M}} = (\mathcal{M}, \varphi) = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0, \leq) & (0, \leq) & (0, \leq) \\ x & (10 + Max, \leq) & (0, \leq) & (10, \leq) \\ y & (Max, \leq) & (0, \leq) & (0, \leq) \end{array} \right), -Max < 0$$

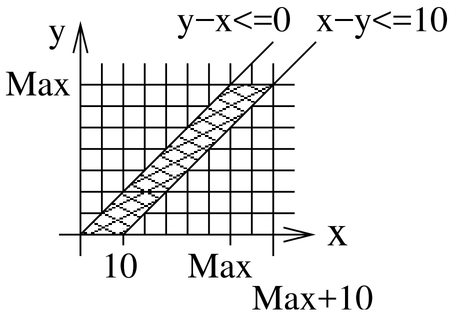


Figure 3.3: Graphical representation of Constrained PDBM $\tilde{\mathcal{M}}$

3.5.1 Operations on Constrained PDBMs

- **Transformation into a canonical form.** PDBMs with the tightest possible bounds are called canonical. Formally, a Constrained PDBM $(\mathcal{M}, \Phi)$ is in canonical form iff for all i,j,k, $\mathcal{M}$ satisfies $(M_{ij}, \Phi) \subseteq^{\mathcal{PB}} (M_{ik}, \Phi) \oplus (M_{kj}, \Phi)$, where $t \in AT(\mathcal{P})$. That means every element of the canonical matrix contains the minimal distance between the variables $x_1, x_2 \in \mathcal{X}$.

  Canonical forms of DBMs (non parametric case) are constructed using the Floyd Warshall algorithm which computes the minimum path between all pairs of entries.

In the parametric case, we follow the same principle by running a symbolic Floyd Warshall algorithm. During a computation, the algorithm needs to determine minimums between terms. For that, the algorithm assumes each of the two possible cases and checks their consistency with respect to the parameter constraints: given two terms t_1 and t_2, it considers the case where $min(t_1, t_2) = t_1$, resp. t_2, and adds $t_1 < t_2$, resp. $t_1 \geq t_2$ to the parameter constraints.

In order to check the consistency of each of the possible cases when computing the minimum between two terms, we have to test the satisfiability of formulas φ of the form $\Phi(P) \wedge t_1 \prec t_2$ where $\prec \in \{<, \leq\}$ and Φ is a parameter constraint. If Φ contains linear constraints or all parameters are real, then the test is decidable [AAB00]. If Φ is nonlinear formula mixing integer and real parameters, this test is undecidable. Nevertheless, we can test the satisfiability of Φ under the assumption that all parameters are reals. Further details are mentioned later in Section 3.5.2, or in [AC01].

The transformation into a canonical form is used to test emptiness.

- **Intersection.** Let $\tilde{M}_1 = (M_1, \Phi_1)$ and $\tilde{M}_2 = (M_2, \Phi_2)$. We already defined intersection on constrained parameterized bounds using the operator $\otimes$. Now we extend the operation on matrices.

  The intersection consists of computing the minimum for every i, j between two terms $M_1(i, j)$ and $M_2(i, j)$ under the parameter constraints $\Phi_1 \wedge \Phi_2$. This is done by splitting and checking the consistency of each case, as in the construction of canonical representation. For every two terms we can get one, two or three constrained parameterized bounds depending on the satisfiability of formulas $\Phi_<, \Phi_=$, or $\Phi_>$. The result will be a set of Constrained PDBMs as shown in the following example [AC01]:

  Let $\mathcal{X} = \{x, y, z\}$ be clocks and $\mathcal{P} = \{T_1, T_2, T_3, T_4\}$ parameters. Let two control states of the transition graph be given by conjunction of constrained parameterized bounds $\psi_1 = (T_1 \leq x \ \wedge\ z \leq T_2 \ \wedge\ x - y \leq 3,\ T_1 \geq 0 \ \wedge\ T_2 \geq 0)$ and $\psi_2 = (x \leq T_3 \ \wedge\ y \leq 5 \ \wedge\ z \leq T_4 \ \wedge\ y - z < 1,\ T_3 \geq 0 \ \wedge\ T_4 \geq 0)$.

  Corresponding PDBMs are $\tilde{M}_1, \tilde{M}_2$ where:

$$(M_1, \Phi_1) = \left(\begin{array}{ccccc} & x_0 & x & y & z \\ x_0 & (0, \leq) & (-T_1, \leq) & (0, \leq) & (0, \leq) \\ x & (\infty, <) & (0, \leq) & (3, \leq) & (\infty, <) \\ y & (\infty, <) & (\infty, <) & (0, \leq) & (\infty, <) \\ z & (T_2, \leq) & (\infty, <) & (\infty, <) & (0, \leq) \end{array}\right), T_1 \geq 0 \wedge T_2 \geq 0$$

$$(M_2, \Phi_2) = \left(\begin{array}{ccccc} & x_0 & x & y & z \\ x_0 & (0, \leq) & (0, \leq) & (0, \leq) & (0, \leq) \\ x & (T_3, \leq) & (0, \leq) & (\infty, <) & (\infty, <) \\ y & (5, \leq) & (\infty, <) & (0, \leq) & (1, <) \\ z & (T_4, \leq) & (\infty, <) & (\infty, <) & (0, \leq) \end{array}\right), T_3 \geq 0 \wedge T_4 \geq 0$$

The result of the intersection $\tilde{M_1} \otimes \tilde{M_2} = \{\tilde{M'}, \tilde{M''}, \tilde{M'''}\}$:

$$(M', \Phi') = \left(\begin{array}{ccccc} & x_0 & x & y & z \\ x_0 & (0, \leq) & (-T_1, \leq) & (0, \leq) & (0, \leq) \\ x & (T_3, \leq) & (0, \leq) & (3, \leq) & (\infty, <) \\ y & (5, \leq) & (\infty, <) & (0, \leq) & (1, <) \\ z & (T_2, \leq) & (\infty, <) & (\infty, <) & (0, \leq) \end{array}\right), T_1 \geq 0 \wedge T_2 \geq 0 \wedge T_3 \geq 0 \wedge T_2 < T_4$$

$$(M'', \Phi'') = \left(\begin{array}{ccccc} & x_0 & x & y & z \\ x_0 & (0, \leq) & (-T_1, \leq) & (0, \leq) & (0, \leq) \\ x & (T_3, \leq) & (0, \leq) & (3, \leq) & (\infty, <) \\ y & (5, \leq) & (\infty, <) & (0, \leq) & (1, <) \\ z & (T_4, \leq) & (\infty, <) & (\infty, <) & (0, \leq) \end{array}\right), T_3 \geq 0 \wedge T_4 \geq 0 \wedge T_1 \geq 0 \wedge T_2 > T_4$$

$$(M''', \Phi''') = \left(\begin{array}{ccccc} & x_0 & x & y & z \\ x_0 & (0, \leq) & (-T_1, \leq) & (0, \leq) & (0, \leq) \\ x & (T_3, \leq) & (0, \leq) & (3, \leq) & (\infty, <) \\ y & (5, \leq) & (\infty, <) & (0, \leq) & (1, <) \\ z & (T_4, \leq) & (\infty, <) & (\infty, <) & (0, \leq) \end{array}\right), T_1 \geq 0 \wedge T_2 \geq 0 \wedge T_3 \geq 0 \wedge T_2 = T_4$$

- **Inclusion.** Let $\tilde{M_1} = (M_1, \varphi_1)$ and $\tilde{M_2} = (M_2, \varphi_2)$ be PDBMs in a canonical form, $\mathcal{X} = \{x_0, \ldots, x_{n_x-1}\}$, $\mathcal{P} = \{p_0, \ldots, p_{n_p-1}\}$. The inclusion of $\tilde{M_1}$ in $\tilde{M_2}$ can be expressed by an extension of the total order $\subseteq^{\mathcal{PB}}$ over parameterized bounds to Constrained PDBMs:

  $$\tilde{M_1} \subseteq^{\mathcal{PB}} \tilde{M_2} \quad \text{iff} \quad \forall i,j \in \{0, n-1\} \; . \; (M_1(i,j), \varphi_1) \subseteq^{\mathcal{PB}} (M_2(i,j), \varphi_2)$$

  Definition of the operation $\subseteq^{\mathcal{PB}}$ over parameterized bounds uses a formula φ_{incl} to test the inclusion. Here, this operation is extended to matrices. Then, we get for the inclusion following relation:

  $$\begin{array}{lll} \tilde{M_1} \subseteq^{\mathcal{PB}} \tilde{M_2} & \text{iff} & \Phi_{incl} \text{ is satisfiable} \\ & & \Phi_{incl} = \forall i,j, \forall \vec{p} \in P^{n_p} \; . \; \varphi_1 \Rightarrow (\varphi_2 \;\wedge\; M_1(i,j) \leq M_2(i,j)) \end{array}$$

  This practically means we build the formula for every pair $M_1(i,j)$ and $M_2(i,j)$ and test its satisfiability. If the test is false, the matrices are not included. Otherwise we continue computing Φ_{incl} for other values i, j.

  Since $\tilde{M_1}, \tilde{M_2}$ have to be in the canonical form, the operation of inclusion is very computationally expensive.

- **Adding constraints.** Guards are conjunctions of constraints in the form $x_i - x_j \prec t$ where $x_i, x_j \in \mathcal{X}$ and $t \in AT(\mathcal{P})$. Guards can be represented by PDBMs.

  Let PDBM $\tilde{M_g} \in \tilde{\mathcal{M}}$ represent a guard g over a transition $(s_1, g, sop, s_2) \in \delta$. Let $\tilde{M}$ represents a current valuation of variables. We compute the intersection between the guard represented by $\tilde{M_g}$ and $\tilde{M}$ as follows:

  $$\tilde{M'} = \tilde{M_g} \otimes \tilde{M}$$

- **Linear assignment.** For resetting a clock or setting a counter, we use the operation in the form $x := A.x + \vec{b}$, where A is a matrix with one 1 on the row. Other elements on the row are 0's. Vector $\vec{b}$ symbolically represents variables that form term t_i. For example, $x_i := x_0 + t_i$.

  Let $\tilde{M}$ be a PDBM representing the current valuation of a system. Let $x := A.\mathcal{X} + \vec{b}$ where $x \in \mathcal{X}$ is the form of the clock assignment. A PDBM $\tilde{M}'$ that represents a new valuation after the assignment is computed as follows [AC01]:

$$\tilde{M}' = A * \tilde{M} * A^T + B.$$

  After the assignment we have to recalculate the canonical form of the result matrix.

- **Elapsing of time.** We define the operation $\Uparrow$ of passing time similarly as for DBMs. Elapsing time in a given clock means to set value $(<, \infty)$ in the corresponding clock variable. In a matrix, the first column represents the upper bounds on clocks. So we need to set these values to infinity.

  Let $\tilde{M} = (M, \varphi)$ be a constrained parameterized matrix. Matrix $\tilde{M}' = (M', \varphi') = \tilde{M} \Uparrow$ is defined as follows:

  - $M'_{i,0} = (\infty, <)$ if $i \neq 0 \ \wedge \ x_i$ is a clock.
  - $M'_{i,j} = M_{i,j}$ otherwise.

- **Test of emptiness.** To check emptiness we compute the canonical form of the matrix. If a canonical PDBM is not satisfied under the specified constraint Φ, the Constrained PDBM is empty.

- **Test of universality.** The test of universality checks if all bounds of $\tilde{M} = (M, \Phi)$ are unbounded. That means to compute if $\forall i, j \ . \ M_{ij} = (\infty, <)$.

- **Operation post()**. Operation post()—sometimes called succ()—is a procedure that computes a set of all configurations that are reachable from the given configuration in one step. This operation is essential for model checking. Operation *post*() applied on the given state (configuration) computes a set of immediate followers (states) according to transitions going out from the given state. By applying *post*() on new states we can extend this operation to *post**(). The operation *post**() is used to generate a set of all configurations of the system.

  Starting from initial state $q \in Q_0$ of a transition system $S_{\mathcal{T}}$ we construct a symbolic reachability graph where each vertex is a symbolic configuration $(q, \tilde{M})$ and edges $(q, \tilde{M}_g, (A, \vec{b}), q')$ correspond to transitions $(s_1, g, sop, s_2) \in \delta$ of $\mathcal{T}$ using *post**(). The vertices of the symbolic graph are treated according to a depth-first traversal. The construction stops when each symbolic configuration that can be generated is covered by some symbolic configuration that has already been computed. During this construction, we use acceleration in order to help termination.

Operation $post(q, \tilde{M})$ can be computed using the three operations over PDBMs as follows:

$$post(q, \tilde{M}) = \{(q', \tilde{M}') \mid (A * (\tilde{M} \otimes \tilde{M}_g) * A^T + B) \Uparrow\}$$

- **Acceleration.** As written above, acceleration is the technique that helps to speed up computation of $post^*()$. [AAB00] shows an extrapolation technique based on guessing automatically the effect of iterating a loop detected in the configuration graph an arbitrary number of times.

  The idea of acceleration is to detect a cycle in the graph and instead of iterating infinitely many times to substitute the detected loop with only one cycle augmented by a special iteration variable. Acceleration checks the distance between variables of the system in the cycle and if the distance is constant it is expected that after certain value n, the value of the variable will be equal to the initial value plus n.

  Let $\tilde{M} = (M, \varphi)$ be a PDBM and let $(q, \tilde{M})$ be a symbolic configuration where $\tilde{M}$ is valuation. Let θ be a control loop—a path $(q_1, g_1, sop_1, q'_1) \dots (q_n, g_n, sop, q'_n)$ in the transition graph $S_{\mathcal{T}}$ such that $q_1 = q'_n$ and $\forall i \in \{1, \dots, n-1\}, q'_i = q_{i+1}$.

  Suppose we have computed $\tilde{M}_1 = (M_1, \varphi_1), \tilde{M}_2 = (M_2, \varphi_2)$ such that $(q, \tilde{M}_1) = post_\theta(q, \tilde{M})$ and $(q, \tilde{M}_2) = post_\theta(q, \tilde{M}_1)$. Let $\Delta = M_1 - M$ and $\Delta' = M_2 - M_1$.

  Then, we suspect that the effect of iterating θ will be to add the same Δ to the original set at each step. After n iterations the set of reachable configurations will be $S + n\Delta$. This operation introduces a new parameter $n \in \mathbb{N}$ corresponding to numbers of iterations of the control loop.

  More formally, this extrapolation technique means to check whether the two following conditions hold:

$$\begin{aligned} C1: &\quad \forall \vec{p} \in \mathcal{P}^{n_p}, \forall \vec{n} \in \mathbb{N}^n \ . \ \varphi_2(p, n) \Rightarrow \Delta = \Delta' \\ C2: &\quad \forall n \geq 0 \ . \ post^2_\theta((q, \tilde{M} + n * \Delta)) = post_\theta((q, \tilde{M} + (n+1) * \Delta)) \end{aligned}$$

  Condition C1 expresses that the effect of θ after two iterations is to add the same Δ. This condition is decided under the constraint φ_2 which is stronger then φ_1 associated with M_1. This is due to the fact that each transition introduced new constraints over parameters and variables but never removed any. Condition C2 allows to check that each application of θ has an effect of adding Δ provided the guards and the invariants in Δ are satisfied.

  If C1 and C2 hold we add $post_\theta(q, \tilde{M} + n * \Delta)$ to the set of reachable states, and the edge $(q, \tilde{M}_1) \xrightarrow{\theta^*} post_\theta(q, \tilde{M} + n * \Delta)$ to the symbolic graph.

3.5.2 Decidability

In [AC01], the author discusses decidability of parametric systems. Decidability depends on satisfaction of formulas describing constraints. The formulas we have to test satisfaction on them can be of three types:

$$\begin{array}{ll} (1) & \forall p \in \mathcal{P} \ . \ \forall n \exists m \in \mathbb{N} \ . \ \varphi \\ (2) & \forall p \in \mathcal{P} \ . \ \forall n \in \mathbb{N} \ . \ \varphi \\ (3) & \exists p \in \mathcal{P} \ . \ \exists n \in \mathbb{N} \ . \ \varphi \end{array}$$

where $\varphi \in AT(\mathcal{P})$.

We have the following cases:

- *Parameters are integers.* If φ is linear, the three formulas above are formulas of Presburger arithmetics and are decidable.

  If φ contains at least one non-linear term we cannot test satisfiability of these formulas for integers. In this case, we expect all variables as reals. If a formula is not satisfiable for reals, it is not satisfiable for integers as well. If it is satisfiable for reals, we cannot prove that this is valid as well for integers. By interpreting formulas under reals, we consider upper approximation of the sets of possible configurations.

- *Parameters are reals.*
  - If formulas don't contain iteration variables, then all three formulas are satisfiable.
  - If they include iteration variables, we have to eliminate real parameters in the formula. If such formula is linear over integers, we can decide it. If not, we cannot say anything. This arithmetic is called half-linear.

- *Parameters are reals and integers.* The formulas contain half-linear arithmetics (see above). The method for testing satisfiability is the same as for the case above.

PDBMs use two-dimensional arrays for representing clocks and counters. However, for counters, we don't need two dimensions because it is not needed to store differences between every pair of counters. So we propose a new structure that reduces both space and time requirements for storing and manipulating data—parametric hypercubes—in chapter 4.

3.6 Example

To demonstrate the usage of parametric timed automata we show an example here. The example is the extension of the light switch from Figure 2.5. We replace the constant value 100 with a parameter MAX. A model of the parametric system will be more complicated—see Figure 3.4.

Behaviour of the system can be described using a transition system—a graph of symbolic configurations Q_A. A symbolic configuration is a pair $(q, (M, \Phi))$ where q is the location of the automaton A and M is a clock valuation for $\mathcal{X}$ in the form of PDBM and Φ is a parameter valuation in the form of a quantifier-free formula. We as well call Φ a

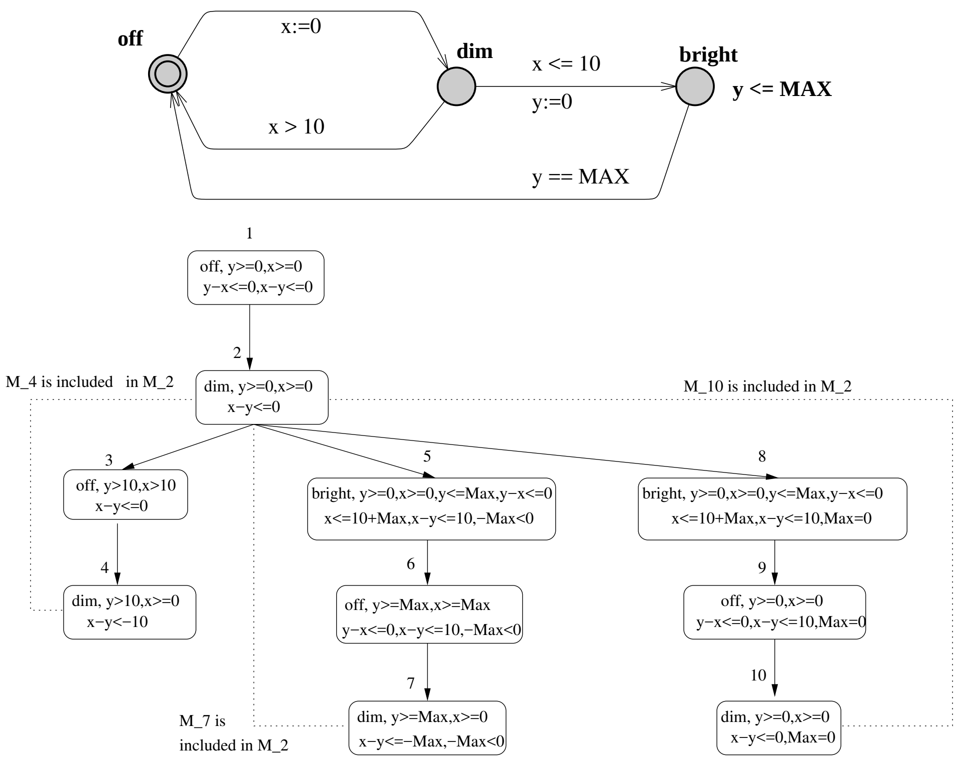


Figure 3.4: A light switch with a parameter–timed automaton and its transition system

parameter constraint and the tuple (M, Φ) a constrained PDBM. For our system Q_A is represented by the following pairs (M, Φ):

$$(M_1, \varphi_1) = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0, \leq) & (0, \leq) & (0, \leq) \\ x & (\infty, <) & (0, \leq) & (0, \leq) \\ y & (\infty, <) & (0, \leq) & (0, \leq) \end{array}\right), true \qquad (M_2, \varphi_2) = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0, \leq) & (0, \leq) & (0, \leq) \\ x & (\infty, <) & (0, \leq) & (0, \leq) \\ y & (\infty, <) & (\infty, <) & (0, \leq) \end{array}\right), true$$

$$(M_3, \varphi_3) = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0, \leq) & (-10, <) & (-10, <) \\ x & (\infty, <) & (0, \leq) & (0, \leq) \\ y & (\infty, <) & (\infty, <) & (0, \leq) \end{array}\right), true \quad (M_4, \varphi_4) = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0, \leq) & (0, \leq) & (-10, <) \\ x & (\infty, <) & (0, \leq) & (-10, <) \\ y & (\infty, <) & (\infty, <) & (0, \leq) \end{array}\right), true$$

$$(M_5, \varphi_5) = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0, \leq) & (0, \leq) & (0, \leq) \\ x & (10 + Max, \leq) & (0, \leq) & (10, \leq) \\ y & (Max, \leq) & (0, \leq) & (0, \leq) \end{array}\right), -Max < 0$$

$$(M_6, \varphi_6) = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0, \leq) & (-Max, \leq) & (-Max, \leq) \\ x & (\infty, <) & (0, \leq) & (10, \leq) \\ y & (\infty, <) & (0, \leq) & (0, \leq) \end{array}\right), -Max < 0$$

$$(M_7,\varphi_7) = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0,\leq) & (0,\leq) & (-Max,\leq) \\ x & (\infty,<) & (0,\leq) & (-Max,\leq) \\ y & (\infty,<) & (\infty,<) & (0,\leq) \end{array}\right), -Max < 0$$

$$(M_8,\varphi_8) = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0,\leq) & (0,\leq) & (0,\leq) \\ x & (10+Max,\leq) & (0,\leq) & (10,\leq) \\ y & (Max,\leq) & (0,\leq) & (0,\leq) \end{array}\right), Max = 0$$

$$(M_9,\varphi_9) = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0,\leq) & (0,\leq) & (0,\leq) \\ x & (\infty,<) & (0,\leq) & (10,\leq) \\ y & (\infty,<) & (0,\leq) & (0,\leq) \end{array}\right), Max = 0 \quad (M_{10},\varphi_{10}) = \left(\begin{array}{cccc} & x_0 & x & y \\ x_0 & (0,\leq) & (0,\leq) & (0,\leq) \\ x & (\infty,<) & (0,\leq) & (0,\leq) \\ y & (\infty,<) & (\infty,<) & (0,\leq) \end{array}\right), Max = 0$$

Graphical representation of clock zones $M_1 - M_7$ is in Figure 3.5.

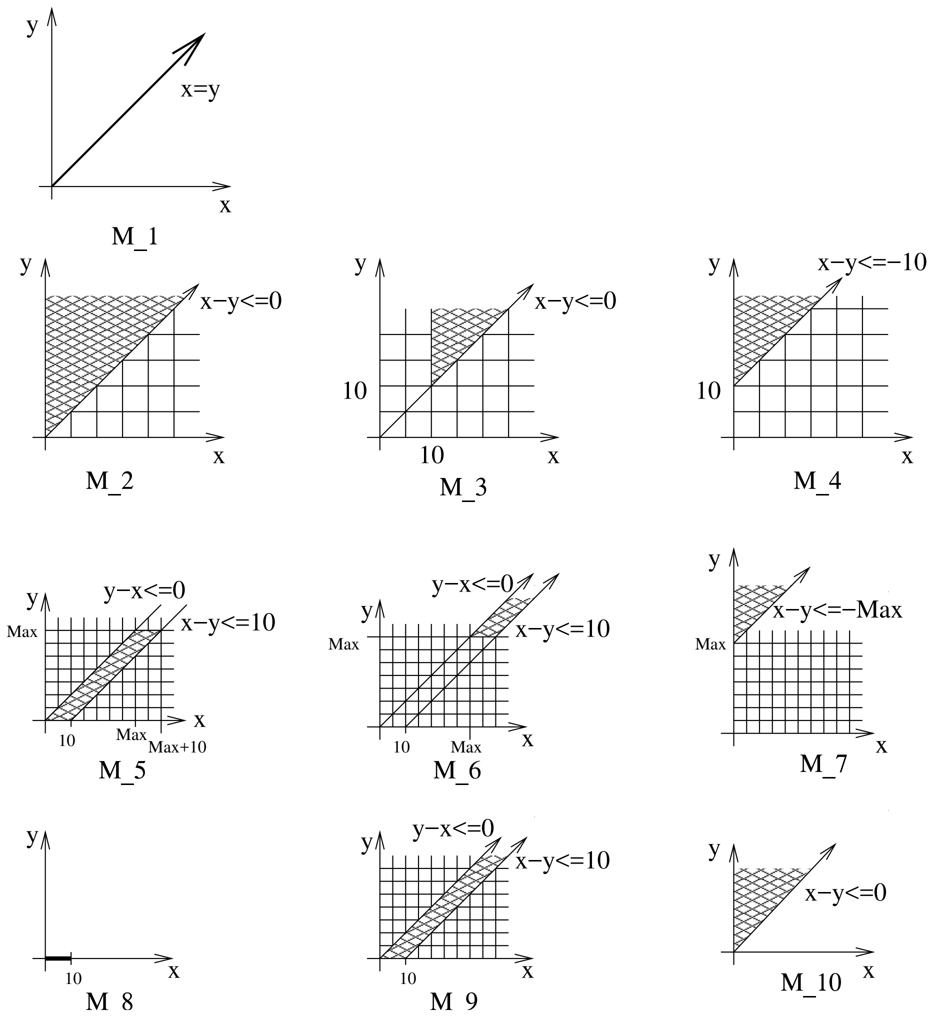


Figure 3.5: A graphical representation of clock zones

3.7 Polyhedra

Another interesting data structure used for parametric model checking is polyhedron. Polyhedra are used in HyTech [HHWT97] for representing clocks and other variables. As PDBMs, polyhedra implement parameters. This is the reason why we look deeply at this structure here. Our goal is to explore operation over polyhedra and compare them with operations over PDBMs and parametric hypercubes.

While the approach to the analysis of timed automata is based on regions, the analysis of linear hybrids systems is based on polyhedra [LSW97]. A polyhedron is a subset of the Euclidean space $\mathbb{R}^n$, which can be described by linear inequalities of the form $Ax \leq b$. Each guard and invariant of a linear hybrid systems describe a polyhedron.

Polyhedron is described by a linear formula. Given a location p of a linear hybrid system and a polyhedron z, we can compute all points reachable from (p, z) in the same way as we did with the region technique. The result of these operations is a pair (p', z') where p' is a control location of the linear hybrid system and z' is again a polyhedron. However, termination is not guaranteed anymore as there are generally infinitely many polyhedra.

When we cross the border of linearity and enter the field of non-linear systems, then in general we do not have any algorithm at all. However, even non-linear systems can be analysed. Often, a linear approximation of the system can be found. Approximations are also a good idea in the linear case if the analysis does not terminate. Then it might help to use over-approximation of polyhedra to force the algorithm to terminate. For details, we recommend [ACH+95].

Timed automata can be considered as special case of hybrid automata where clocks evolve with the fixed rate equal one. The symbolic model checking of a timed automaton requires the manipulation of certain linear constraints on clock values in the form $x \prec b$ and $x - y \prec b$ for clocks $x, y \in \mathcal{X}$ and $b \in \mathbb{Z}$.

In linear hybrid automaton, we admit more general linear constraints on continuous variables in the form $A * \vec{x} \prec \vec{c}$ where A is a constant matrix and c is a constant vector. Since the number of possible constraints is no longer finite, when moving from timed automata to linear hybrid automata, termination is no longer guaranteed.

Convex polyhedra. Convex polyhedra can be viewed as a set of solution of a linear system. It forms an abstraction for numerical (and parametric) domain. Polyhedra use two dual representations that are suitable for different operations over polyhedra. The dual representation are as follows:

- **Set of solutions of a system of linear constraints:**

  $$\mathcal{P} = \{x \mid Ax \geq B\}$$

  where A is a non-zero matrix and B is a vector.

  This representation is called an implicit form too. It is suitable for intersection, inclusion, image and pre-image.

- **Convex hull of a system of generators (V,R):**

$$\mathcal{P} \;=\; \{x \mid x = L\lambda + R\mu + V\nu,\ \mu,\nu \geq 0,\ \sum \nu = 1\}$$

  This representation is called a parametric representation and it is a collection of vertices, rays and lines. It is a linear combination of lines (matrix L), a convex combination of vertices (matrix V), and a positive combination of extreme rays (matrix R).

  This representation is suitable for inclusion, convex union and test of emptiness.

In Figure 3.6, there is a simple polyhedra with the following representation:

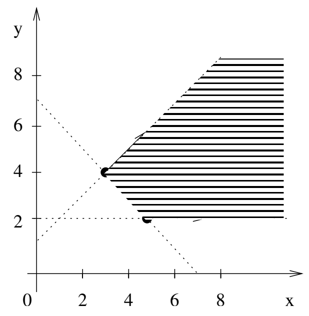


Figure 3.6: A simple polyhedra

- a set of inequalities:

$$\begin{aligned} x + y &\geq 7 \\ y &\geq 2 \\ -x + y &\leq 1 \end{aligned}$$

- a set of vertices and rays

$$v_1 = \begin{pmatrix} 5 \\ 2 \end{pmatrix} \qquad v_2 = \begin{pmatrix} 3 \\ 4 \end{pmatrix}$$
$$r_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad r_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

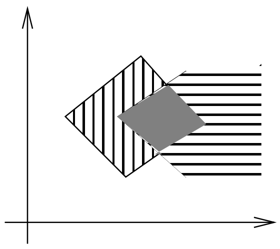
Since each form is optimal for different operations and the dual form computation is exponential $\mathcal{O}(N^{D/2})$, where D is the number of variables and N is the number of constraints. Both forms are usually used during verification.

3.7.1 Operations over Polyhedra

Polyhedra are closed under intersection, convex union, definition, and affine transformation. However, they are not closed under union since the union of any two polyhedra is not necessarily convex. Similarly, polyhedra are not closed under the difference operation. To obtain closure of these two operations, it is necessary to expand the model from a simple polyhedron to a finite union of polyhedra.

In this section we introduce basic operations on polyhedra and their comparison with PDBMs. Implementation of polyhedra operations is described in details in [Wil93].

- **Intersection**—on systems of constraints

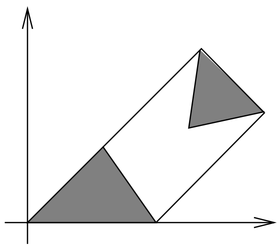


$$\{A_1 * X \geq B_1\} \cap \{A_2 * X \geq B_2\} = \{A_1 * X \geq B_1 \ \wedge \ A_2 * X \geq B_2\}$$

Intersection is performed by concatenating the lists of constraints from two or more polyhedra into one list, and finding the polyhedron which satisfies all of the combined constraints. This is done by finding the extremal rays which satisfy the combined constraints, and then reducing both the constraints and rays into one polyhedron.

With this operation there appears a lot of new inequalities.

- **Convex hull**—approximation of union, on systems of generators



$$[V_1, R_1] \sqcup [V_2, R_2] = [V_1 \cup V_2, R_1 \cup R_2]$$

The non-convex union operation simply combines two domains into one. The lists of polyhedra associated with the domains are combined into a single list. This may create non-minimal representations. In computation it is necessary to take into consideration that, for example, $\mathcal{P}_1 \supset \mathcal{P}_2$ and the union can be reduced simply to $\mathcal{P}_1$.

Convex union is performed by concatenating the lists of rays and lines of two polyhedra in a domain into one list, and finding the set of constraints which tightly bind all of those objects. This is done by finding the dual of the list of rays and lines, and reducing both the constraints and rays into one polyhedron.

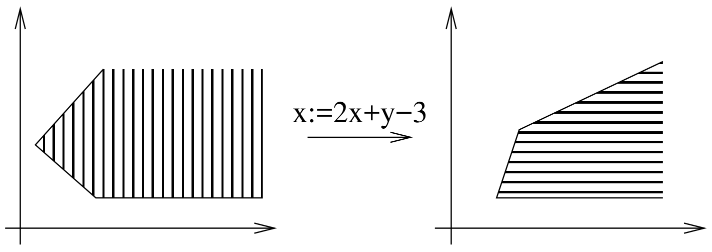
- **Difference**, on systems of linear constraints

  Domain difference $\mathcal{P}_1 - \mathcal{P}_2$ computes the domain which is a part of $\mathcal{P}_1$ but not a part of $\mathcal{P}_2$. It is equivalent to $\mathcal{P}_1 \cup \sim \mathcal{P}_2$, where $\sim \mathcal{P}_2$ is the complement domain of $\mathcal{P}_2$. If $\mathcal{P}_2$ is the intersection of a set of hyperplanes (representing equalities) and half-spaces (representing inequalities), then the universe of $\mathcal{P}_2$ is computed as follows:

$$\begin{aligned} \sim \mathcal{P}_2 &= \sim (\cap_i P_i) \\ &= \cup_i (\sim P_i) \\ &\quad \text{where} \\ \sim P_i &= \begin{cases} \{x \mid A^T x < 0\} & \text{when } P_i = \{x \mid A^T x \geq 0\} \\ \{x \mid (A^T x < 0 \cup A^T x > 0)\} & \text{when } P_i = \{x \mid A^T x = 0\} \end{cases} \end{aligned}$$

  The computation of difference is transformed to computation of union. Because polyhedra is not closed under union, so it is not closed under difference either.

- **Linear assignment to variables**, on systems of generators



$$[V, R] \xrightarrow{X := C * X + D} [C * V + D, C * R]$$

- **Test for inclusion**, on system of generators

$$[V, R] \subseteq \{A * X \geq B\} \Leftrightarrow \begin{cases} A * v \geq B, & \forall v \in V \\ A * r \geq 0, & \forall r \in R \end{cases}$$

- **Test for emptiness**, on systems of generators

$$[V, R] = \emptyset \Leftrightarrow V = \emptyset$$

  An empty domain is a polyhedron which includes no points. This happens when there is no point that can satisfy all of the constraints.

- **Test for universality**

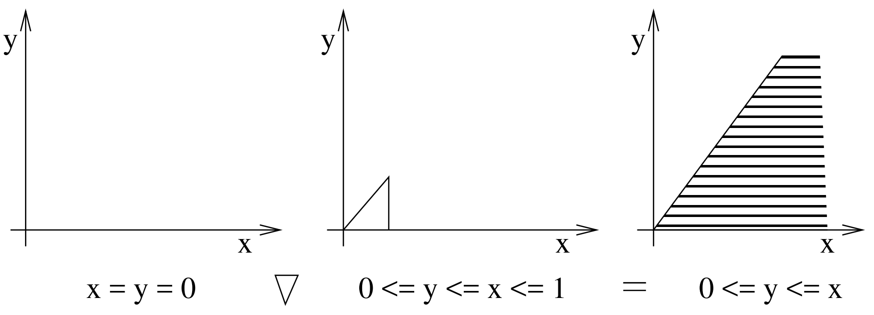
  A universal polyhedron is one that includes all points within a certain dimensional sub-space. It is unbounded in all directions. It is created if a system is not constrained at all.

- **Widening.** An approximation analysis of polyhedra is provided by abstract interpretation techniques—widening [CC77]. The idea is to extrapolate the limit of a sequence of polyhedra in such a way that an upper approximation of the limit is always reached in a finite number of iterations.

  We define a widening operator ∇ on polyhedra, such that:

  – For each pair (P, P') of polyhedra, $P \sqcup P' \subseteq P \nabla P'$

  – For each infinite increasing sequence $(P_0, P_1, \ldots, P_n, \ldots)$ of polyhedra, the sequence defined by $Q_0 = P_0, Q_{n+1} = Q_n \nabla P_{n+1}$ is not strictly increasing.



  Intuitively, the system of linear constrains of $P \nabla P'$ is made of exactly those constraints of P which are also satisfied by P'. The system of constraints of $P \nabla P'$ is obtained by removing from the system of P all the constraints not satisfied by P'.

3.8 Discussion

Polyhedra as well as PDBMs can be successfully used to represent symbolic data structures. The main difference is the way how they represent symbolic domains. While PDBMs use matrices to represent set of constraints, polyhedra use lists of vectors, rays and lines or a set of linear equations/non-equations. Both these structures can be used for timed domains with parameters.

Polyhedra allow constraints in the form $\sum_i a_i * x_i \prec c_i$ while PDBMs $x - y \prec c, x \prec c$. Data representation in polyhedral form is less coarse than difference matrices because we can describe a domain using general linear inequalities of the form $ax - y \prec b$, which is not possible for DBMs.

In Figure 3.7 we can see a polyhedron represented by a set of inequalities $y \geq 0.5x - 0.5, y \leq 3, y \geq 2x-5, y \geq 3x-3$. This cannot be represented by difference bound matrices. DBMs use over–approximation to express this. For example $0 < y \leq 3, 1 \leq x \leq 4, x - y \leq 2, y - x \leq 1$, see Figure 3.8.

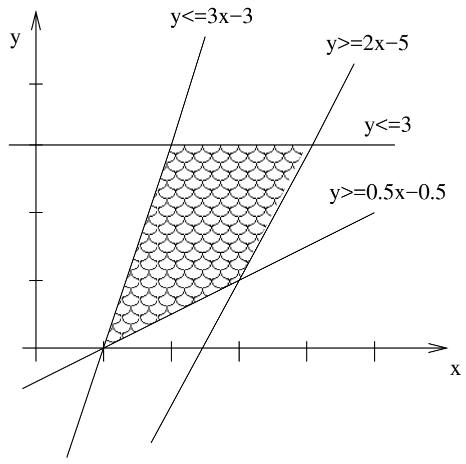


Figure 3.7: A polyhedra $y \geq 0.5x - 0.5, y \leq 3, y \geq 2x - 5, y \geq 3x - 3$

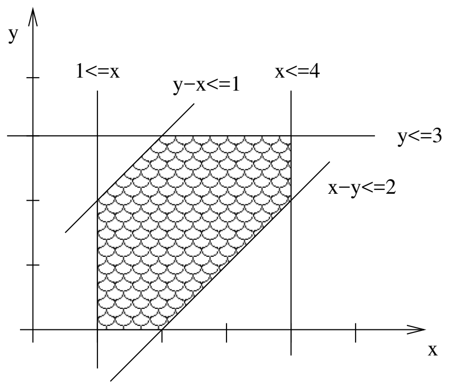


Figure 3.8: Over-approximation by DBM $0 < y \leq 3, 1 \leq x \leq 4, x - y \leq 2, y - x \leq 1$

On the other hand, this feature brings more complexity into polyhedra operations and computation of a set of successors during system analysis.

There are two main problems in the practical computation with polyhedra [ACH$^+$95]:

- Handling disjunctions of systems of linear inequalities—there is no easy way for deciding if a union of polyhedra is included into another.
- The fix-point computation may involve infinite iteration.

Polyhedra require space $3*(n+1)$ of memory to store its data for n variables while PDBM requires n^2. Thus, polyhedra are more compact to store data valuation. However, it is not easy to implement some operations, for instance acceleration. Without acceleration it is not possible to analyse systems with more parameters.

As acceleration, polyhedra use widening but this operation generates the set of configuration which is an upper approximation. Also because of difficult construction of union of polyhedra (operation is not closed) the set of inequalities becomes too large after computation of the set of successors. Then the system consumes too much memory and the analysis may finish with the lack of memory.

Polyhedra work well for hybrid systems with not many parameters but for timed or counter automata with many parameters they are too robust and inefficient.

Chapter 4

Parameterized Hypercubes

In previous chapters we described the basics of verification of timed systems with parameters and showed two data structures for representation of abstract data domains. We mentioned that successful analysis of a system—more precisely termination of the analysis—depends mostly on choice of abstract domains. We showed operation on PDBMs and polyhedra—two representations that are used for parametric verification to store clocks and counters.

In this chapter we introduce a new data structure based on intervals [ST02] that can be used to represent abstract domain for counters with parameters. The advantage of this structure is that it reduces the space needed to represent data with respect to PDBMs and simplifies some operations (test of emptiness, intersection etc.). Another advantage is that this structure allows more general constraints over variables. We can represent conditions of the form $x_1 + \ldots + x_n \prec t$ while PDBMs only allow comparison of two variables $x_i - x_j \prec t$. This extension of constraints was inspired by a real need that we found during a project where we analysed PGM protocol - see chapter 6.

The chapter is divided into three parts. At the beginning we define parameterized bounds and intervals—their syntax and interpretation. We as well define an operator of total order on bounds and intervals that is necessary for other operations. Then we form a new structure called parameterized hypercubes. We define basic operations over it. First proposal of these operations was published in [Mat04a]. The last part shows implementation of this structure in the verification tool TReX and its usage.

4.1 Parameterized Intervals

Parameterized hypercubes are defined as a tuple of constrained parameterized intervals that form a region of valuations of variables. In this section, we introduce intervals, parameterized intervals and interval formulas.

4.1.1 Arithmetical Term, Formula

Arithmetical term $AT(\mathcal{X})$. Let $\mathcal{X}$ be a finite set of variables, x a variable of this set and c a constant over numerical domain, e.g. $\mathbb{N}$. The set of *arithmetical terms* over $\mathcal{X}$, $AT(\mathcal{X})$ is defined by the following grammar:

$$t ::= c \mid x \mid t + t \mid t - t \mid c * t$$

Arithmetical term with parameters $AT(\mathcal{P})$. Let $\mathcal{P}$ be a finite set of parameters, p a variable of this set and c an integer. The set of *arithmetical terms* over $\mathcal{P}$, $AT(\mathcal{P})$ is defined by the following grammar:

$$t ::= c \mid p \mid t + t \mid t - t \mid c * t$$

Arithmetical formula $FO(\mathcal{X})$. The set of *first-order arithmetical formulas* over $\mathcal{X}$, $FO(\mathcal{X})$, is defined by the following grammar:

$$\varphi \quad ::= \quad \mathit{true} \mid t \leq t \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists x.\varphi \mid \mathit{isInt}(t)$$

where $t \in AT(\mathcal{X})$. For parametric analysis we consider $t \in AT(\mathcal{X} \cup \mathcal{P})$.

Formulas are interpreted over the set of reals, integers, or natural numbers. The predicate *isInt* expresses that a term has an integer value. Notations like *false*, $\vee$, $\forall$, and $\Rightarrow$ are derived as usual from the operators defined in the grammar above. We denote $F(\mathcal{X})$ *quantifier-free* formulas in $FO(\mathcal{X})$.

Semantics of an arithmetical formula. The semantics of formulas and terms are defined as follows. Let $\mathbb{I}$ be a numerical domain of variables, $\mathbb{I} = \mathbb{Z}, \mathbb{N}$, or $\mathbb{R}$. Let $\nu : \mathcal{X} \to \mathbb{I}$ be a valuation of variables in $\mathcal{X}$ into the numerical domain $\mathbb{I}$. We denote $[\![t]\!]\nu$ the value in $\mathbb{I}$ of the term t under the valuation ν. The satisfaction of formula in $FO(\mathcal{X})$ is defined as follows:

$$\begin{array}{lll}
\nu \models \mathit{true} & & \\
\nu \models t_1 \leq t_2 & \text{iff} & [\![t_1]\!]\nu \leq [\![t_2]\!]\nu \\
\nu \models \neg\varphi & \text{iff} & \nu \not\models \varphi \\
\nu \models \varphi_1 \wedge \varphi_2 & \text{iff} & \nu \models \varphi_1 \text{ and } \nu \models \varphi_2 \\
\nu \models \exists x.\varphi & \text{iff} & \exists v \in \mathbb{I}.[\![x/v]\!]\nu \models \varphi \\
\nu \models \mathit{isInt}(t) & \text{iff} & [\![t]\!]\nu \in \mathbb{Z}
\end{array}$$

Several fragments of $FO(\mathcal{X})$ are interesting for us:

- $RFO(\mathcal{X})$ is the fragment where the variables take up values in reals and the predicate *isInt* does not appear.
- $LFR(\mathcal{X})$ is the fragment where the variables take up values in reals and all terms used are linear.

- $FI(\mathcal{X})$ is the fragment where the variables take up values in integers.
- $LFO(\mathcal{X})$ means the *linear arithmetics* (Presburger arithmetics). It is the fragment of formulas from $FI(\mathcal{X})$ without multiplication.

The decidability results for each fragment are:

Fragment	*Sat*	*Complexity*
FO	undecidable	-
RFO	decidable [Tar51]	EXPTIME
LFR	decidable [Tar51]	EXPTIME
FI	undecidable	-
LFO	decidable [FR73]	3EXPTIME

Complexity of decision of linear first-order formulas is 3EXPTIME—it depends exponentially on the length of a formula φ. This means that time of system analysis depends mostly on satisfaction of formulas, than on efficiency of an algorithm for computing configuration states.

In our analysis of parametric counter systems we will use fragments of first-order logic that are decidable. Let φ is a formula from $AT(\mathcal{X})$ and $\vec{\nu}$ valuation of variables from $\mathcal{X}$. Satisfaction of the formula φ under valuation $\vec{\nu}$ can be expressed as $\vec{\nu} \models \varphi$. For example, let $\varphi = \exists x \,.\, x < 5$ and valuation of x be $\nu(x) = 3$. We can see that $[\![\exists x \,.\, x < 5]\!]_{x=3} =$ true, i.e., the formula is satisfied under given valuation.

Similarly to PDBMs we distinguish three cases where parameters are integers, reals or both. For discussion about this topic see Section 3.5.2.

4.1.2 Numerical Bounds

Let $\mathbb{I}$ be a numerical domain of variables, $\mathbb{I} = \mathbb{Z}, \mathbb{N}$, or $\mathbb{R}$. Suppose intervals over $\mathbb{I}$ of the form $[a, b], [a, \infty)$ representing the symbolic data structure for variables over $\mathbb{I}$. In our approach, we define the interval as a pair of the lower and the upper numerical bounds.

Numerical bounds. The set of *numerical bounds* $\mathcal{B}$ over $\mathbb{I}$ is given by following definition:

$$\mathcal{B} = \mathbb{I} \times \{<, \leq\} \ \cup \ \{(\infty, <)\}.$$

Numerical bounds represent either the lower, or the upper bound of the variable. Their denotation is as follows: if $b = (z, \prec)$ is the lower bound of x, $\prec \in \{<, \leq\}$ and $z \in \mathbb{I}$, the denotation (interpretation) of b is $-x \prec z$, i.e., $-z \prec x$. If b is the upper bound, the denotation is $x \prec z$. Such interpretation is efficient because it enables us to encode both the upper bounds and the lower bounds using operators $<$ and $\leq$ only.

For example, if $b_1 = (3, \leq)$ is a lower numerical bound on variable x its denotation is $-x \leq 3$, i.e., $-3 \leq x$. If $b_2 = (5, <)$ is an upper numerical bound of x it restricts the variable by $x < 5$.

Interval. We define interval $I(x)$ over $\mathbb{I}$ as a pair of numerical bounds. Let $b_i = (z_i, \prec_i), b_s{}^1 = (z_s, \prec_s) \in \mathcal{B}$, where $\prec_i, \prec_s \in \{<, \leq\}$, $z_i, z_s \in \mathbb{I} \cup \{\infty\}$, be numerical bounds over $\mathbb{I}$. The set of *intervals* $\mathcal{I}$ is a set of pairs of lower and upper bound:

$$\mathcal{I} = \mathcal{B} \times \mathcal{B}$$

We define *interval* $I(x) \in \mathcal{I}$ *for variable* x over $\mathbb{I}$ as follows:

$$I(x) = \langle b_i, b_s \rangle,$$

where $b_i \in \mathcal{B}$ is a lower bound of variable x and $b_s \in \mathcal{B}$ is an upper bound of variable x.

The semantics of this interval is $(-x \prec_i z_i) \wedge (x \prec_s z_s)$. Bound z_i is the infimum and z_s is the supremum of the interval. For instance, interval $I(x) = \langle (3, \leq), (7, <) \rangle$ means $-3 \leq x < 7$.

Definition of the interval over a variable can be extended to vectors. Let $\vec{x} = (x_1, x_2, \ldots, x_n)$ be a vector. The vector can be interpreted as a tuple of n variables. Interval $I(\vec{x})$ over $\mathbb{I}^n$ is the n-tuple of pairs of numerical bounds:

$$I(\vec{x}) = (\langle (b_i^1, \prec_i^1), (b_s^1, \prec_s^1) \rangle, \langle (b_i^2, \prec_i^2), (b_s^2, \prec_s^2) \rangle, \ldots, \langle (b_i^n, \prec_i^n), (b_s^n, \prec_s^n) \rangle)$$

This interval forms bounds over variables $x_1, \ldots, x_n$ in the form $-b_i^1 \prec_i^1 x \prec_s^1 b_s^1, \ldots, -b_i^n \prec_i^n x \prec_s^n b_s^n$.

For example, let $\vec{x} = (x_1, x_2)$. Interval $I(\vec{x}) = (\langle (-3, \leq), (4, <) \rangle, \langle (5, <), (10, \leq) \rangle)$ denotes data domain for x_1 and x_2 that can be written as $3 \leq x_1 < 4$ and $-5 < x_2 \leq 10$.

Relation of total order.

1. *Relation of total order* $\subseteq^\sharp$ *over* $\{<, \leq\}$. Let $\sharp_1, \sharp_2 \in \{<, \leq\}$. We define the relation of total order as follows:

   $$\sharp_1 \subseteq^\sharp \sharp_2 \iff (\sharp_1 = \sharp_2) \vee ((\sharp_1 =' <') \wedge (\sharp_2 =' \leq'))$$

   Thus, we can say that $<$ is less then $\leq$. For comparison over this relation we will use notation $\leq$ as usual.

   For strict inclusion, $\subset^\sharp$ the operator is defined as follows:

   $$\sharp_1 \subset^\sharp \sharp_2 \iff ((\sharp_1 =' <') \wedge (\sharp_2 =' \leq'))$$

   For example, $< \subseteq^\sharp \leq$, $< \subseteq^\sharp <$, $< \subset^\sharp \leq$.

[1] i stands for infimum and s for supremum

2. *Relation of total order $\subseteq^{\mathcal{B}}$ over numerical bounds.* Because the lower bounds and the upper bounds have different interpretation, we have to distinguish four different cases in the definition of the relation—ordering of lower bounds, ordering of upper bounds, ordering of a lower bound and an upper bound, and ordering of an upper bound and a lower bound. Let $a_1 = (z_i^1, \prec_i^1), a_2 = (z_i^2, \prec_i^2)$ be lower bounds and $b_1 = (z_s^2, \prec_s^2), b_2 = (z_s^2, \prec_s^2)$ be upper bounds, $a_1, a_2, b_2, b_2 \in \mathcal{B}$.

   We define the relation of total order $\subseteq^{\mathcal{PB}}$ over parameterized bounds as follows:

$$\begin{aligned}
[\![a_1 \subseteq^{\mathcal{B}} a_2]\!] &= (z_i^1 < z_i^2) \vee ((z_i^1 = z_i^2) \wedge (\prec_i^1 \leq \prec_i^2)) \\
[\![b_1 \subseteq^{\mathcal{B}} b_2]\!] &= (z_s^1 < z_s^2) \vee ((z_s^1 = z_s^2) \wedge (\prec_s^1 \leq \prec_s^2)) \\
[\![a_1 \subseteq^{\mathcal{B}} b_2]\!] &= (-z_i^1 < z_s^2) \vee ((-z_i^1 = z_s^2) \wedge (\prec_i^1 \leq \prec_s^2)) \\
[\![b_1 \subseteq^{\mathcal{B}} a_2]\!] &= (z_s^1 < -z_i^2) \vee ((z_s^1 = -z_i^2) \wedge (\prec_i^1 \leq \prec_2^2))
\end{aligned}$$

   Such interpretation is useful for inclusion over bounds of the same type—either lower or upper bounds. Their denotation of total order is the same and we don't need to take into account different interpretation of bounds. Only in cases where upper and lower bounds are mixed together we need to pay attention to different interpretation and distinguish the difference by the sign '-' for lower bounds.

   Test of inclusion of upper and lower bounds is an important operation, for example, for intersection over intervals (Section 4.3).

   For example, in Figure 4.1, we consider the following bounds as lower bounds (a) $(3, \leq) \subseteq^{\mathcal{B}} (7, <)$, (b) $(-2, <) \subseteq^{\mathcal{B}} (-2, \leq)$, (c) $(-2, \leq) \not\subseteq^{\mathcal{B}} (-2, <)$, (d) $(5, \leq) \not\subseteq^{\mathcal{B}} (-2, <)$, while in Figure 4.2 we take them as upper bounds.

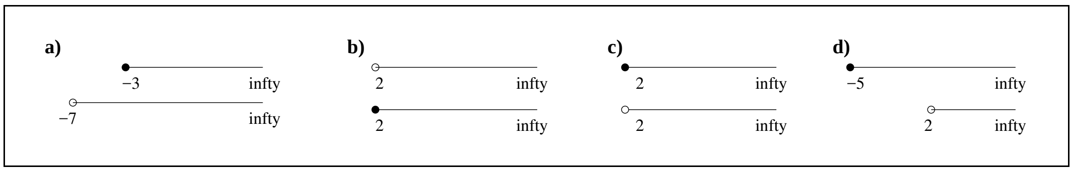


Figure 4.1: Lower numerical bounds

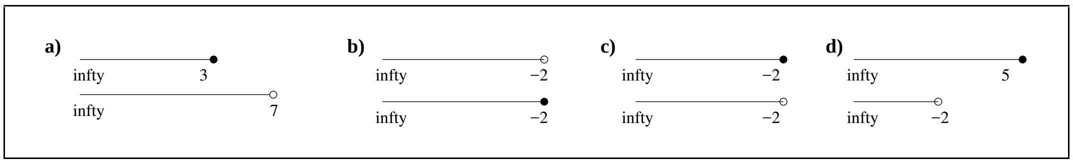


Figure 4.2: Upper numerical bounds

   Both interpretations of examples (a) and (b) show bounds properly ordered with respect to $\subseteq^{\mathcal{B}}$. Bounds (c) and (d) are not ordered.

Strict inclusion over bounds $\subset^{\mathcal{B}}$ is similar to the previous definition except for the relation over signs. Instead of $\leq$ (more precisely $\leq^{\sharp}$) there will be an operator $<$ (more precisely $<^{\sharp}$).

3. *Relation of total order $\subseteq$ over intervals.* We define a relation of total order over intervals similarly to total order over numerical bounds. Let $I(x) = \langle b_i, b_s \rangle, I(x') = \langle b'_i, b'_s \rangle$ be two intervals from $\mathcal{I}$. We define the relation of total order $\subseteq$ over intervals as follows:

$$I(x) \subseteq I'(x) \iff (b_i \subseteq^{\mathcal{B}} b'_i) \wedge (b_s \subseteq^{\mathcal{B}} b'_s)$$

Note that interpretation of inclusion of lower bounds is defined differently than inclusion over upper bounds.

For example, $\langle(3, \leq), (9, <)\rangle \not\subseteq \langle(-5, \leq), (7, <)\rangle$ because $3 \not< -5$ and $9 \not< 7$. Indeed, interval $-3 \leq x < 9$ is not included into $5 \leq x < 7$, see Figure 4.3. Relation $\langle(-5, \leq), (7, <)\rangle \subseteq \langle(3, \leq), (9, <)\rangle$ is valid.

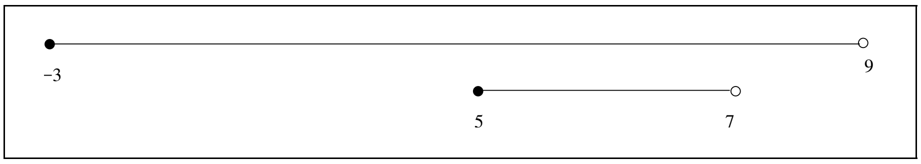


Figure 4.3: Inclusion of intervals $\langle(-5, \leq), (7, <)\rangle \subseteq \langle(3, \leq), (9, <)\rangle$

Interval formulas. Interval formulas are a dual representation of numerical intervals in a more compact form than intervals as introduced above. It is a data structure similar to Difference Bound Matrices (DBMs)—see [Dil89].

Let $\mathcal{X}$ be a finite set of variables, $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, and $\mathbb{I}$ be a numerical domain. We define *interval formulas* i over $\mathcal{X}$ and $\mathbb{I}$ by the following grammar:

$$i \ ::= \ true \mid x_j \prec a \mid -x_j \prec b \mid i \wedge i$$

where $a, b \in \mathbb{I} \cup \{\infty\}$, $j \in \{1, \ldots, n\}$, and $\prec \in \{<, \leq\}$. The denotation of the formula is defined by:

$$\begin{aligned}
[\![true]\!] &= \mathbb{I}^n \text{ (whole domain)} \\
[\![x_j \prec a]\!] &= \{(x_1, \ldots, c, \ldots, x_n) \mid c \prec a, c \in \mathbb{I}\} \\
[\![-x_j \prec b]\!] &= \{(x_1, \ldots, c, \ldots, x_n) \mid -b \prec c, c \in \mathbb{I}\} \\
[\![i_1 \wedge i_2]\!] &= [\![i_1]\!] \cap [\![i_2]\!]
\end{aligned}$$

For example, suppose $\mathcal{X} = \{x_1, x_2, x_3\}$ and $\mathbb{I} = \mathbb{Z}$. The following interval formula i defines data domain in $\mathbb{I}^3$:

$$i = x_1 \leq 5 \ \wedge -x_1 < 2 \ \wedge \ x_2 \leq 4$$

where x_1 is from interval $(-2, 5]$, x_2 from $(-\infty, 4]$ and x_3 is from $(-\infty, \infty)$. This formula encodes interval $I(\vec{x}) = (\langle(2, <), (5, \leq)\rangle), (\langle(\infty, 0), (4, \leq)\rangle), (\langle(\infty, <), (\infty, <)\rangle)$

As mentioned above, the interval i can be described using DBM. This representation is larger that the corresponding interval formula, because DBM represents a set of differences of every two variables in the form $x - y \prec c$, where $\prec = \{<, \leq\}$. There is as well a special bound x_0 that is always 0 and represents a constraint on a single variable. DBMs represent regions for clocks in timed automata. For counter automata, we don't use relation of the form $x - y \prec c$ (difference between counters). Relations of the form $x \prec c$ are mostly sufficient. So we can use intervals instead of DBMs to represent counters in counter automata.

For instance, $x_1 \leq 5$ is expressed by $x_1 - x_0 \leq 5$ in DBM. Unbounded variables use special bounds $(\infty, 0)$. Matrix D that represents interval formula $i = x_1 \leq 5 \ \wedge -x_1 < 2 \ \wedge \ x_2 \leq 4$ can be as follows:

$$D = \left|\begin{array}{ccccc} & x_0 & x_1 & x_2 & x_3 \\ x_0 & (0, \leq) & (\mathbf{2}, <) & (\infty, <) & (\infty, <) \\ x_1 & (\mathbf{5}, \leq) & (0, \leq) & (\infty, <) & (\infty, <) \\ x_2 & (\mathbf{4}, \leq) & (\mathbf{7}, <) & (0, \leq) & (\infty, <) \\ x_3 & (\infty, <) & (\infty, <) & (\infty, <) & (0, \leq) \end{array}\right|$$

Graphic representation of D is depicted in Figure 4.4.

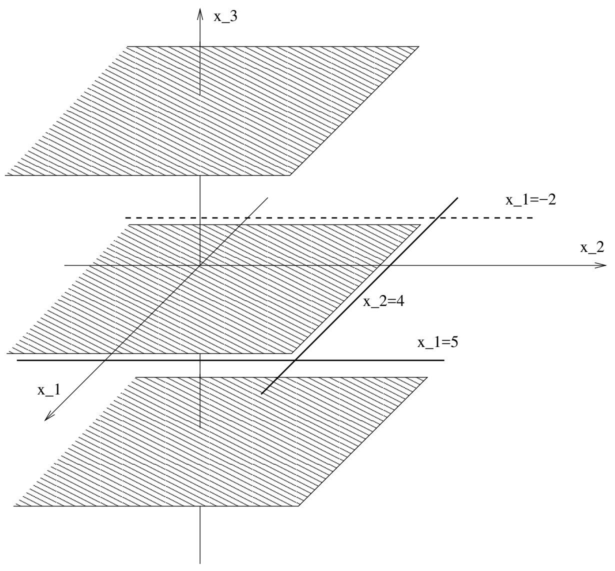


Figure 4.4: Graphical representation of data domain defined by $x_1 \leq 5 \wedge -x_1 < 2 \wedge x_2 \leq 4$

We can see that the space requirements for n variables is $2n$ for intervals while $(n+1)^2$ for DBMs.

4.1.3 Parameterized Bounds

Numerical bounds may be extended to parameterized bounds by replacing the numerical value by an arithmetical term $t \in AT(\mathcal{P})$ over the set of parameters $\mathcal{P} = \{p_1, \ldots, p_n\}$.

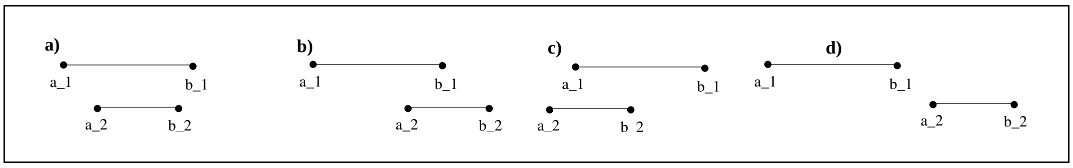
Parameterized bounds. Let $\mathcal{D}$ be a domain over which clocks and parameters range, $\mathcal{D}$ is the set of integers $\mathbb{Z}$. The set of *parameterized bounds* $\mathcal{PB}$ over $\mathcal{D}$ is defined as follows:

$$\mathcal{PB} = AT(\mathcal{P}) \times \{<, \leq\} \ \cup \ \{(\infty, <)\}.$$

A parameterized bound b represents a set of non-parameterized (numeric) bounds depending on the values taken by parameters. In order to constrain the set of values taken by parameters, we define the notion of *constrained parameterized bound*, which specifies for a parameterized bound the constraint (formula) $F(\mathcal{P})$ which should be satisfied by the parameters.

Total order over parameterized bounds $\subseteq^{\mathcal{PB}}$. This operation is similar to the ordering of numerical bounds as defined in Section 4.1.2, it is extended to terms from $AT(\mathcal{P})$. Here, we also distinguish four cases of the interpretation of total order. Let $a_1 = (t_i^1, \prec_i^1)$, $a_2 = (t_i^2, \prec_i^2)$ be lower bounds and $b_1 = (t_s^1, \prec_s^1)$, $b_2 = (t_s^2, \prec_s^2)$ be upper bounds, $a_1, a_2, b_2, b_2 \in \mathcal{PB}$.

$$\begin{aligned}
[\![a_1 \subseteq^{\mathcal{PB}} a_2]\!] &= (t_i^1 < t_i^2) \ \vee \ ((t_t^1 = t_i^2) \ \wedge \ (\prec_i^1 \leq \prec_i^2)) \ \ \text{(a)} \\
[\![b_1 \subseteq^{\mathcal{PB}} b_2]\!] &= (t_s^1 < t_s^2) \ \vee \ ((t_s^1 = t_s^2) \ \wedge \ (\prec_s^1 \leq \prec_s^2)) \ \ \text{(b)} \\
[\![a_1 \subseteq^{\mathcal{PB}} b_2]\!] &= (-t_i^1 < t_s^2) \ \vee \ ((-t_i^1 = t_s^2) \ \wedge \ (\prec_i^1 \leq \prec_s^2)) \ \ \text{(c)} \\
[\![b_1 \subseteq^{\mathcal{PB}} a_2]\!] &= (t_s^1 < -t_i^2) \ \vee \ ((t_s^1 = -t_i^2) \ \wedge \ (\prec_i^1 \leq \prec_2^2)) \ \ \text{(d)}
\end{aligned}$$



If t_2 is $+\infty$ (unbounded), then $a \subseteq^{\mathcal{PB}} b$. If t_1 is $+\infty$ and t_2 is not $+\infty$, then $a \not\subseteq^{\mathcal{PB}} b$.

Constrained parameterized bounds. The set of constrained parameterized bounds $\tilde{\mathcal{PB}}$ over $\mathcal{D}$ is defined as follows:

$$\tilde{\mathcal{PB}} = \mathcal{PB} \times F(\mathcal{P})$$

The interpretation (denotation) of a constrained parameterized bound is given by the following description:

$$\begin{aligned}
[\![((\infty, <), \varphi)]\!] &= \{(\infty, <)\} \\
[\![((t, \prec), \varphi)]\!] &= \{(v, \prec) \ \mid \ \exists \nu . \nu \models \varphi \ \wedge \ v = [\![t]\!]\nu\}
\end{aligned}$$

where $t \in AT(\mathcal{P})$ is a term over the set of parameters $\mathcal{P}$, $\varphi \in F(\mathcal{P})$, and $\nu : AT(\mathcal{P}) \to \mathcal{D}$ is valuation of parameterized terms into the domain $\mathcal{D}$.

We say that a constrained parameterized bound is satisfiable (or not empty) if there exists any valuation of parameters from $\mathcal{P}$ such that the valuation satisfies the constraint φ over parameters.

For example, the upper bound $\tilde{b} = ((<, 2*p-q), p \leq 3+q)$ of x can be interpreted as $x < 2*p-q \ \wedge \ p \leq 3+q$. This bound is not empty, because there exist many valuations over $\mathbb{N}$, for example $q = 3 \ \wedge \ p \leq 6$.

As for numerical bounds, the operators $<$ and $\leq$ are ordered: $<$ is strictly less than $\leq$.

Relation of total order $\subseteq$. Relation of total order over constrained parameterized bounds is an extension of the relation of total order over numerical intervals as defined in Section 4.1.2. Let $\tilde{b_1} = ((t_1, \prec_1), \varphi_1)$ and $\tilde{b_2} = ((t_2, \prec_2), \varphi_2)$ be constrained parameterized bounds from $\tilde{\mathcal{PB}}$, $\vec{p} \in \mathcal{P}^{n_p}$ is a vector of parameters from $\mathcal{P} = \{p_0, \ldots, p_{n_p-1}\}$. We define relation of total order $\subseteq$ on constrained parameterized bounds as follows:

$$\begin{aligned} \tilde{b_1} \subseteq \tilde{b_2} \quad &\Longleftrightarrow \quad \forall \vec{p} \in \mathcal{P}^{n_p} \ . \ \varphi_1 \Rightarrow (\varphi_2 \ \wedge \ (b_1 \subseteq^{\mathcal{PB}} b_2)) \\ &\Longleftrightarrow \quad \forall \vec{p} \in \mathcal{P}^{n_p} \ . \ \neg\varphi_1 \ \vee \ (\varphi_2 \ \wedge \ (b_1 \subseteq^{\mathcal{PB}} b_2)) \end{aligned}$$

The formula above says that for every valuation of parameters the constraint of the lesser bound is included in the constraint of the greater bound and that the terms are included in the context of specified constraints. The constraint φ_1 is stronger then φ_2, so the inclusion of terms is tested under this context (implication).

For computation of the inclusion, it is useful to test negative formula, i.e., to find a valuation ν that makes negative formula false. The non-inclusion will be as follows:

$$\tilde{b_1} \not\subseteq \tilde{b_2} \quad \Longleftrightarrow \quad \exists \vec{p} \in \mathcal{P}^{n_p} \ . \ \varphi_1 \ \wedge \ (\neg\varphi_2 \ \vee \ (b_1 \not\subseteq^{\mathcal{PB}} b_2))$$

This formula says that two constrained parameterized bounds are not included under context of φ_1, φ_2 if there exists any valuation of parameters such that either φ_2 is not valid or parameterized bounds are not included.

The strict order, $\subset$, between constrained parameterized bounds may be defined similarly.

With this definition, the bound $((\infty, <), \mathit{true})$ is the top element of the set of bounds with respect to both ordering relations. This means that every bound is included in $((\infty, <), \mathit{true})$.

The equality operator $=$. The equality operator $=$ over constrained parameterized bounds is defined as usual. For any $\tilde{b}_1, \tilde{b}_2 \in \tilde{\mathcal{PB}}$:

$$\tilde{b}_1 = \tilde{b}_2 \quad \Longleftrightarrow \quad (\tilde{b}_1 \subseteq \tilde{b}_2) \ \wedge \ (\tilde{b}_2 \subseteq \tilde{b}_1)$$

The minimum operator *min*. Similarly to PDBMs, we define operator *min* on parameterized intervals. It can be used in intersection and so-called correct form computation (see Section 4.1.4).

Let $\tilde{b}_1 = ((t_1, \prec_1), \varphi_1)$ and $\tilde{b}_2 = ((t_2, \prec_2), \varphi_2)$ be two elements of $\tilde{\mathcal{PB}}$, $\vec{p}$ a vector of parameters from $\mathcal{P}$. The definition of the *min* operator uses following three formulas:

$$\begin{array}{lcl} \Phi_< & \equiv & \exists \vec{p} \in \mathcal{P}^{n_p}.\varphi_1 \wedge \varphi_2 \wedge t_1 < t_2 \\ \Phi_= & \equiv & \exists \vec{p} \in \mathcal{P}^{n_p}.\varphi_1 \wedge \varphi_2 \wedge t_1 = t_2 \\ \Phi_> & \equiv & \exists \vec{p} \in \mathcal{P}^{n_p}.\varphi_1 \wedge \varphi_2 \wedge t_1 > t_2 \end{array}$$

The *min* operator is defined as follows:

$$min : \tilde{\mathcal{PB}} \times \tilde{\mathcal{PB}} \rightarrow 2^{\tilde{\mathcal{PB}}}$$

$$\begin{array}{rcl} min(\tilde{b}_1, \tilde{b}_2) & = & min_<(\tilde{b}_1, \tilde{b}_2, \Phi_<) \\ & \cup & min_=(\tilde{b}_1, \tilde{b}_2, \Phi_=) \\ & \cup & min_>(\tilde{b}_1, \tilde{b}_2, \Phi_>) \\ \text{where} & & \\ min_<(\tilde{b}_1, \tilde{b}_2, \Phi_<) & = & \begin{cases} \{((t_1, \prec_1), \varphi_1 \ \wedge \ \varphi_2 \ \wedge \ (t_1 < t_2))\} & \text{if } \Phi_< \\ \emptyset & \text{otherwise} \end{cases} \\ min_=(\tilde{b}_1, \tilde{b}_2, \Phi_=) & = & \begin{cases} \{((t_1, min(\prec_1, \prec_2)), \varphi_1 \ \wedge \ \varphi_2 \ \wedge \ (t_1 = t_2))\} & \text{if } \Phi_= \\ \emptyset & \text{otherwise} \end{cases} \\ min_>(\tilde{b}_1, \tilde{b}_2, \Phi_>) & = & \begin{cases} \{((t_2, \prec_2), \varphi_1 \ \wedge \ \varphi_2 \ \wedge \ (t_1 > t_2))\} & \text{if } \Phi_> \\ \emptyset & \text{otherwise} \end{cases} \end{array}$$

The result of *min* operator may be a set of one, two or three constrained parameterized bounds. If an operand of *min* is a special bound $((\infty, <), true)$—unlimited bound, the minimum is trivially the other operand:

$$min(((\infty, <), true), \tilde{b}) = \{\tilde{b}\}$$

Bound $((\infty, <), true)$ is a neutral element for *min*.

For example, let $\tilde{b}_1 = ((p + 3, <), p > 8)$ and $\tilde{b}_2 = ((q, \leq), q < p + 3)$. Then,

$$\begin{array}{rcl} min(\tilde{b}_1, \tilde{b}_2) & = & ((p+3, <), p > 8 \ \wedge \ q < p + 3 \ \wedge \ p + 3 < q) \\ & \cup & ((p+3, <), p > 8 \ \wedge \ q < p + 3 \ \wedge \ p + 3 = q) \\ & \cup & ((q, \leq), p > 8 \ \wedge \ q < p + 3 \ \wedge \ p + 3 > q) \end{array}$$

Because $\Phi_<$ and $\Phi_=$ are trivially false, the result is only one bound with a new constraint, i.e. $min(\tilde{b}_1, \tilde{b}_2) = ((q, \leq), p > 8 \ \wedge \ q < p + 3)$.

4.1.4 Parameterized Intervals

Constrained parameterized intervals. In the previous section we showed a relation between numerical bounds and intervals. This relation can be extended to parameterized bounds and intervals. Let $x \in \mathcal{X}$ be a variable and $\mathcal{P}$ a set of parameters.

The set of constrained parameterized intervals $\tilde{\mathcal{I}}$ for variables from $\mathcal{X}$ and parameters from $\mathcal{P}$ over $\mathcal{D}$ is defined as follows:

$$\tilde{\mathcal{I}}(\mathcal{X}, \mathcal{P}) = \mathcal{PB} \times \mathcal{PB} \times F(\mathcal{P})$$

A constrained parameterized interval $\tilde{I}(x, \mathcal{P})$ for variable x and parameters from $\mathcal{P}$ over $\mathcal{D}$ is a pair of constrained parameterized bounds

$$\tilde{I}(x, \mathcal{P}) = (\langle (t_i, \prec_i), (t_s, \prec_s) \rangle, \varphi)$$

where $\prec_i, \prec_s \in \{<, \leq\}, t_i, t_s \in AT(\mathcal{P}), \varphi \in F(\mathcal{P})$ such that

$$(-x \prec_i t_i) \ \wedge \ (x \prec_s t_s) \ \wedge \ \varphi$$

For example, let $\mathcal{P} = \{p, q\}$ and $\varphi = p > 0 \wedge q \leq p + 10$. Then $\tilde{I}(x, \mathcal{P}) = (\langle (3 - p, <), (q + 10, \leq) \rangle, \varphi)$ denotes the interval $(p - 3 < x \leq q + 10) \ \wedge \ (p > 0 \ \wedge \ q \leq p + 10)$. The interval is depicted on Figure 4.5.

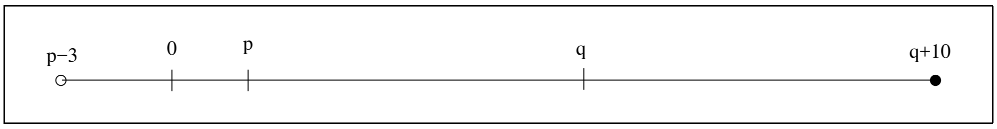


Figure 4.5: Parameterized interval $(p - 3 < x \leq q + 10) \ \wedge \ (p > 0 \ \wedge \ q \leq p + 10)$

Relation of total order over constrained parameterized intervals $\subseteq^{\tilde{I}}$. Let $i = (\langle a, b \rangle, \varphi)$ and $i' = (\langle a', b' \rangle, \varphi')$ be constrained parameterized intervals from $\tilde{\mathcal{I}}(\mathcal{X}, \mathcal{P})$, where $a, a' \in \mathcal{PB}$ are the lower bounds, $b, b' \in \mathcal{PB}$ are the upper bounds, and $\vec{p}$ is a vector for parameters from $\mathcal{P}$. Total order over constrained parameterized intervals is similar to total order over numerical intervals:

$$(\langle a, b \rangle, \varphi) \subseteq^{\tilde{I}} (\langle a', b' \rangle, \varphi') \iff \forall \vec{p} \in P^{n_p} \ . \ \varphi \Rightarrow (\varphi' \ \wedge \ (a \subseteq^{\mathcal{PB}} a') \ \wedge \ (b \subseteq^{\mathcal{PB}} b'))$$

Parameterized intervals can be extended to vectors. Let $\vec{x} = (x_1, x_1, \ldots, x_n)$ be a vector of variables $\{x_1, \ldots, x_n\}$. Constrained parameterized interval $\tilde{I}(\vec{x}, \mathcal{P})$ over $\mathcal{D}^n$ is an n-tuple of pairs of constrained parameterized bounds

$$\tilde{I}(\vec{x}, \mathcal{P}) = (\langle (t_i^1, \prec_i^1), (t_s^1, \prec_s^1) \rangle, \ldots, \langle (t_i^n, \prec_i^n), (t_s^n, \prec_s^n) \rangle, \varphi)$$

The minimum operator $min_{\mathcal{I}}$. Here we extend the operator min on constrained parameterized intervals. Let $I = (\langle a, b \rangle, \varphi)$ and $I' = (\langle a', b' \rangle, \varphi')$ be two constrained parameterized intervals. Minimum of intervals I and I', written $min_{\mathcal{I}}(I, I')$, is a union of constrained parameterized intervals given as follows:

$$\begin{array}{rcl} min_{\mathcal{I}}(I, I') & = & (\langle min_<(a, a', \Phi_<), min_<(b, b', \Phi^2_<)\rangle, \Phi_< \ \wedge \ \Phi^2_<) \\ & \cup & (\langle min_<(a, a', \Phi_<), min_=(b, b', \Phi^2_=)\rangle, \Phi_< \ \wedge \ \Phi^2_=) \\ & \cup & (\langle min_<(a, a', \Phi_<), min_>(b, b', \Phi^2_>)\rangle, \Phi_< \ \wedge \ \Phi^2_>) \\ & \cup & (\langle min_=(a, a', \Phi_=), min_<(b, b', \Phi^2_<)\rangle, \Phi_= \ \wedge \ \Phi^2_<) \\ & \cup & (\langle min_=(a, a', \Phi_=), min_=(b, b', \Phi^2_=)\rangle, \Phi_= \ \wedge \ \Phi^2_=) \\ & \cup & (\langle min_=(a, a', \Phi_=), min_>(b, b', \Phi^2_>)\rangle, \Phi_= \ \wedge \ \Phi^2_>) \\ & \cup & (\langle min_>(a, a', \Phi_>), min_<(b, b', \Phi^2_<)\rangle, \Phi_> \ \wedge \ \Phi^2_<) \\ & \cup & (\langle min_>(a, a', \Phi_>), min_=(b, b', \Phi^2_=)\rangle, \Phi_> \ \wedge \ \Phi^2_=) \\ & \cup & (\langle min_>(a, a', \Phi_>), min_>(b, b', \Phi^2_>)\rangle, \Phi_> \ \wedge \ \Phi^2_>) \end{array}$$

Finding the minimum on intervals means to find the minimum on the upper and the lower bounds using operator *min* as defined in Section 4.1.3. Since the minimum on each parameterized bound may result in three new constrained parameterized bounds, we have to make a combination with all of these bounds to get the minimum on intervals.

Parameterized interval formula. Let $\mathcal{X}$ be a finite set of variables, $\mathcal{X} = \{x_1, \ldots, x_n\}$. Let $\mathcal{P}$ be a set of parameters. The set of *parameterized interval formulas* over $\mathcal{X}$ and $\mathcal{P}$, $PIF(\mathcal{X}, \mathcal{P})$, is defined by the following grammar:

$$\iota ::= true \mid x \prec t \mid -x \prec t \mid \iota \wedge \iota$$

with $t \in AT(\mathcal{P})$.

Constrained parameterized interval formula. Like for bounds, we define a set of *constrained parameterized interval formulas*, $CPIF(\mathcal{X}, \mathcal{P})$, as follows:

$$CPIF(\mathcal{X}, \mathcal{P}) = PIF(\mathcal{X}, \mathcal{P}) \times F(\mathcal{P}).$$

The *denotation (semantics)* of the constrained parameterized interval formula (ι, φ) over the numerical domain $\mathbb{I}$ is defined by:

$$\begin{array}{rcl} [\![true, \varphi]\!] & = & \mathbb{I}^n \\ [\![x_i \prec t, \varphi]\!] & = & \{(x_1, \ldots, x_i, \ldots, x_n) \mid \exists \nu . \nu \models \varphi \wedge x_i \prec [\![t]\!]\nu\} \\ [\![-x_i \prec t, \varphi]\!] & = & \{(x_1, \ldots, x_i, \ldots, x_n) \mid \exists \nu . \nu \models \varphi \wedge -x_i \prec [\![t]\!]\nu\} \\ [\![\iota_1 \wedge \iota_2, \varphi]\!] & = & [\![\iota_1, \varphi]\!] \cap [\![\iota_2, \varphi]\!] \end{array}$$

where $x_i \in \mathcal{X}, t \in AT(\mathcal{P})$, and $\varphi \in F(\mathcal{P})$. If the constraint for a variable in $\tilde{\iota}$ represents an empty numerical interval, then $\tilde{\iota}$ is *empty*, $[\![\tilde{\iota}]\!] = \emptyset$.

A formula in $CPIF(\mathcal{X}, \mathcal{P})$ may contain none or several inequalities with respect to a variable of $\mathcal{X}$. For example, if $\mathcal{X} = \{x_1, x_2\}$ and $\mathcal{P} = \{p, q\}$, the following formula

$$((x_1 \leq p \ \wedge -x_1 \leq q \ \wedge \ x_1 \leq 3), (p \geq 1 \ \wedge \ q \leq 0))$$

contains no inequality for x_2 (which is equivalent to $x_2 < \infty \ \wedge -x_2 < \infty$) and several inequalities for x_1.

In the example above, the lower bound of x_1 is $-q$ and the upper bound is p and 3. To have a *correct form* of the interval (some sort of normal form with only one pair of bounds on a variable), we need to find minimum on both these bounds, that means to find $min(p, 3)$ in the context of $\varphi = p \geq 1 \ \wedge \ q \leq 0$. Because of parameters, there are two case—either p is the minimum, or 3 is the minimum; recall definition of min() in Section 4.1.3. Note that this splitting implies a stronger constraint on φ. In the first case (a), it adds the constraint $p < 3$ (p is the minimum), in the other case (b), the constraint $p \geq 3$ (3 is the minimum), see Figure 4.6. Surprisingly, the latter case implies interval of x_1 that does not depend on the value of parameter p. So, the interval for x_1 may be rewritten as follows:

$$\begin{aligned} & (x_1 \leq p \ \wedge -x_1 \leq q \ \wedge \ x_1 \leq 3, \ p \geq 1 \ \wedge \ q \leq 0) \\ = \quad & (x_1 \leq p \ \wedge -x_1 \leq q, \ 1 \leq p < 3 \ \wedge \ q \leq 0) \\ \vee \ & (x_1 \leq 3 \ \wedge -x_1 \leq q, \ 3 \leq p \ \wedge \ q \leq 0) \end{aligned}$$

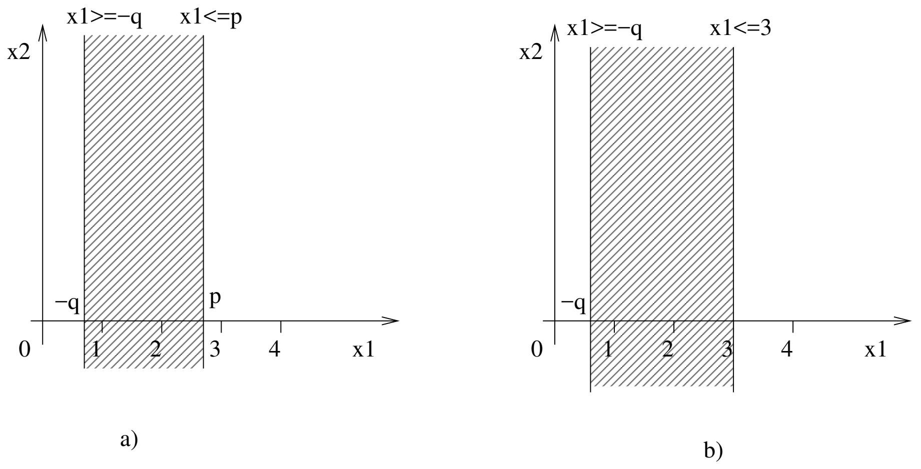


Figure 4.6: Minimum of constrained parameterized intervals.

Relation between the interval formula and bounds. For implementation purposes, it is more efficient to represent parameterized bounds of vector $\vec{x}$ using an interval formula from $CPIF(\mathcal{X}, \mathcal{P})$. A parameterized interval formula associated with a constrained parameterized interval $\tilde{I}(\vec{x}, \mathcal{P}) = (\langle (t_i^1, \prec_i^1), (t_s^1, \prec_s^1) \rangle, \ldots, \langle (t_i^n, \prec_i^n), (t_s^n, \prec_s^n) \rangle, \varphi)$ is define as follows:

$$cpif(\vec{x}, \mathcal{P}) = (-x_1 \prec_i^1 t_i^1 \ \wedge \ x_1 \prec_s^1 t_s^1 \ \wedge \ \ldots \ \wedge -x_n \prec_i^n t_i^n \ \wedge \ x_n \prec_s^n t_s^n, \varphi)$$

So, we can say, that the interval formula is a dual representation of constraints over variables defined by bounds of the interval. Both representation are important. We use conjunction of parameterized bound for internal representation of data domains of variables. Interval formula is more efficient for description of data operations.

Parameterized intervals and bounds presented here are used for a definition of a new abstract data structure, parameterized hypercubes, that will be introduced in the following section.

4.2 Parameterized Hypercubes

Parameterized hypercubes (pHCubes) symbolically represent valuation of variables $x_1, \ldots, x_n$. This symbolic data structure is efficient to represent counters in extended time and counter automata. Manipulation with this structure (intersection, union,widening) is easier in comparison with other parametric data structures like PDBMs or polyhedra.

This structure represents a set of possible values of every variable using parameterized intervals constrained by a formula φ. First, we recall the definition of a constrained parameterized interval over a variable x:

$$\tilde{I}(x, \mathcal{P}) = (\langle a, b\rangle, \varphi) = (\langle (t_i, \prec_i), (t_s, \prec_s)\rangle, \varphi)$$

where $\prec_i, \prec_s \in \{<, \leq\}, a = (t_i, \prec_i)$ and $b = (t_s, \prec_s)$ are constrained parameterized bounds such that $(-x \prec_i t_i) \ \wedge \ (x \prec_s t_s) \ \wedge \ \varphi$. Terms t_i, t_s (infimum and supremum) are from $AT(\mathcal{P}) \ \cup \ \{\infty\}$, constraint φ is a quantifier-free formula over $\mathcal{P}$ given by the grammar $\varphi ::= t \prec t \mid \neg\varphi \mid \varphi \wedge \varphi$.

A parameterized hypercube is an abstract data structure based on constrained parameterized intervals. It can be described using a parameterized interval formula in the form of a conjunction of constraints over parameterized terms as follows:

$$\bigwedge_j (t_i^j \ \prec_i \ x_j \ \prec_s \ t_s^j) \ \wedge \ \varphi$$

where $x_j \in \mathcal{X}$ is a variable, and t_i^j, resp. t_s^j, is the lower, resp. the upper bound.

Here, we will use dual representation using intervals. *A parameterized hypercube ph* over a set of variables $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a vector of parameterized bounds constrained by a formula φ:

$$ph(\mathcal{X}) = (\vec{I}, \varphi) = (I_1, \ldots, I_n, \varphi) = (\langle a_1, b_1\rangle, \ldots, \langle a_n, b_n\rangle, \varphi)$$

where a_i, resp. b_i are the lower, resp. the upper parameterized bound of variable x_i, φ is a constraint over parameters $\mathcal{P}$.

For example, let $\vec{v} = (x, y, z)$ be a set of variables over $\mathcal{X}$, $\mathcal{P} = \{p, q\}$, and $\varphi \in F(\mathcal{P})$. Then $ph(\vec{v})$ is a pHCube over vector $\vec{v}$ such that

$$ph(\vec{v}) = (\langle (<, 0), (\leq, 2*p+1)\rangle, \langle (<, \infty), (<, \infty)\rangle, \langle (\leq, -3), (<, 3+q)\rangle, -p \leq 1 \ \wedge -q \leq -1)$$

This structure represents valuation of three variables x, y, z. Variables x and z are bounded by intervals $(0 < x \leq 2*p+1)$, $(3 \leq z < 3+q)$, respectively. Variable y is unbounded $(-\infty < y < \infty)$ and covers the entire domain. Parameters $\{p, q\}$ are constrained by the formula $\varphi = p \geq -1 \ \wedge \ q \geq 1$. Graphical representation of $ph(\vec{v})$ is shown in Figure 4.7.

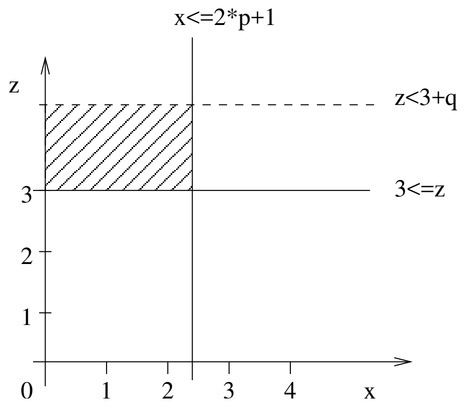


Figure 4.7: Parameterized hypercube $ph(\vec{v})$

4.3 Operations on pHCubes

In this section, we will define operations on pHCubes and their relation to the verification of counter automata. For brevity, we will use *ph* instead of $ph(\mathcal{X})$ to express an instance of pHCube over variables from $\mathcal{X}$ in the following text.

- **Emptiness test.** We test emptiness before applying any other operation during system analysis, for example, inclusion, equality, etc. Let $ph = (\langle a_1, b_1\rangle, \dots, \langle a_n, b_n\rangle, \varphi)$ be a pHCube and $\vec{p}$ be a vector of parameters from $\mathcal{P}$. We say that *ph is not empty* if the following formula ψ is satisfiable

$$\psi \;=\; \exists \vec{p} \in \mathcal{P}^{n_p} \;.\; \varphi \;\wedge\; \bigwedge_i (a_i \subseteq^{\mathcal{PB}} b_i)$$

  which can be written as comparison of terms

$$\psi \;=\; \exists \vec{p} \in \mathcal{P}^{n_p} \;.\; \varphi \;\wedge\; \bigwedge_i (-t_i \leq t_s)$$

  Note:
  More precisely, the second formula should contain $(-t_i < t_s) \;\vee\; ((-t_i = t_s) \;\wedge\; (\prec_i \leq \prec_s))$ according to definition of $\subseteq^{\mathcal{PB}}$. For shortening we use the expression $-t_i \leq t_s$.

  Conversely, *ph is empty* if the negation of this formula is satisfied.

- **Universality test.** A pHCube is universal if every bound is infinite. In this test we don't take into account a context expressed by formula φ. Universality does not depend on φ, it has to be satisfied in every context. The implementation of this operation is simple. pHCube *ph* **is universal (unbounded)** if and only if $\forall i \in \{1, \dots, n\} \;.\; a_i = \{<, \infty\} \;\wedge\; b_i = \{<, \infty\}$.

- **Inclusion.** The test of inclusion is a very important operation over pHCubes. After computing a set of new configurations using *post*(), we test if the new pHCubes are included in a set of the already known configurations represented by pHCubes.

  The inclusion of pHCubes is defined using the relation of total order $\subseteq^{\mathcal{PB}}$ over constrained parameterized bounds. Here, we extend this relation onto pHCubes. Let $ph = (\langle a_1, b_1\rangle, \ldots, \langle a_n, b_n\rangle, \varphi)$ and $ph' = (\langle a'_1, b'_1\rangle, \ldots, \langle a'_n, b'_n\rangle, \varphi')$ be two pHCubes. We define *inclusion* on pHCubes as follows:

  $$ph \subseteq ph' \iff \forall \vec{p} \in \mathcal{P}^{n_p} \ . \ \varphi \Rightarrow (\varphi' \ \wedge \ \bigwedge_i (a_i \subseteq^{\mathcal{PB}} a'_i \ \wedge \ b_i \subseteq^{\mathcal{PB}} b'_i))$$

  For implementation, it is more efficient to test non-inclusion. The new formula is

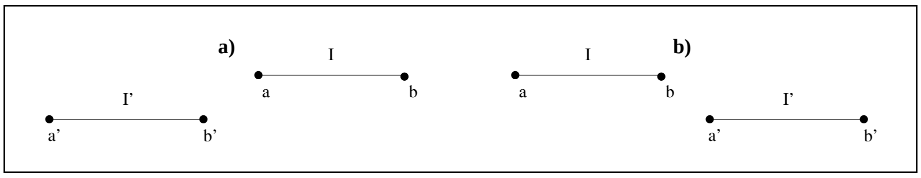
  $$\begin{aligned} ph \not\subseteq ph' \quad &\iff \quad \exists \vec{p} \in \mathcal{P}^{n_p} \ . \ \varphi \ \wedge \ (\neg\varphi' \ \vee \ \bigvee_i (\neg(a_i \subseteq^{\mathcal{PB}} a'_i \ \wedge \ b_i \subseteq^{\mathcal{PB}} b'_i)) \\ &\iff \quad \exists \vec{p} \in \mathcal{P}^{n_p} \ . \ \neg(\varphi \Rightarrow \varphi') \ \vee \ (\varphi \ \wedge \ \bigvee_i (\neg(a_i \subseteq^{\mathcal{PB}} a'_i \ \wedge \ b_i \subseteq^{\mathcal{PB}} b'_i)) \end{aligned}$$

  We test inclusion by testing satisfiability of the formula. If it is not satisfied, the pHCube are included. Before inclusion testing, the test of emptiness has to be done on both pHCubes, because empty pHCube is included in every pHCube. So, if *ph* is empty, $ph \subseteq ph'$. Otherwise, if ph' is empty, $ph \not\subseteq ph'$.

- **Equality.** The test of quality can be implemented via the operation of inclusion because $ph = ph' \iff (ph \subseteq ph') \ \wedge \ (ph' \subseteq ph)$.

- **Intersection.** Computation of intersection is an expensive operation. For every dimension, we need to test all possible cases of relations between two intervals. Here, we demonstrate intersection for pHCubes with only one variable. Let $ph = (I, \varphi)$ and $ph' = (I', \varphi')$ where $I = \langle a, b\rangle, I' = \langle a', b'\rangle, a = (t_i, \prec_i), b = (t_s, \prec_s), a' = (t'_i, \prec'_i), b' = (t'_s, \prec'_s)$. The result of the intersection will be a list of pHCubes $(I'', \varphi \ \wedge \ \varphi\prime \ \wedge \ \psi)$. We distinguish the following cases:

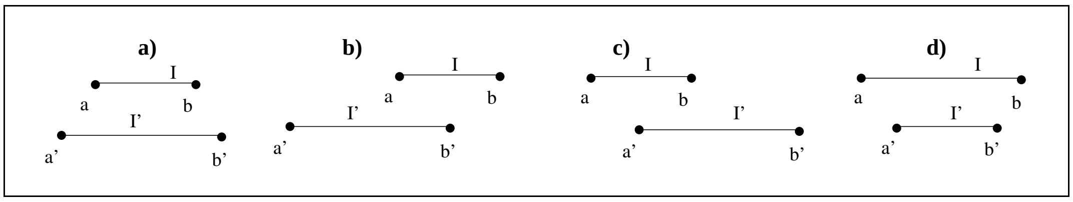
  1. If $I' < I$ (a) or $I < I'$ (b), then the intersection will be empty, i.e., $I \cap I' = \emptyset$. The following constraint must be satisfied: $\psi = b' \subset^{\mathcal{PB}} a \ \vee \ b \subset^{\mathcal{PB}} a'$, that means $\psi = t'_s < -t_i \ \vee \ t_s < t'_i$.

  2. If $I \cap I' \neq \emptyset$, the following four cases are possible:

     (a) $\psi = a' \subset^{\mathcal{PB}} a \ \wedge \ b \subset^{\mathcal{PB}} b'$, i.e., $\psi = t_i < t'_i \ \wedge \ t_s < t'_s$, $I'' = \langle a, b\rangle$

(b) $\psi = a' \subset^{\mathcal{PB}} a \ \wedge \ b' \subseteq^{\mathcal{PB}} b$, i.e., $\psi = t_i < t'_i \ \wedge \ t'_s \leq t_s$, $I'' = \langle a, b' \rangle$
(c) $\psi = a \subseteq^{\mathcal{PB}} a' \ \wedge \ b \subset^{\mathcal{PB}} b'$, i.e., $\psi = t'_i \leq t_i \ \wedge \ t_s < t'_s$, $I'' = \langle a', b \rangle$
(d) $\psi = a \subseteq^{\mathcal{PB}} a' \ \wedge \ b' \subseteq^{\mathcal{PB}} b$, i.e., $\psi = t'_i \leq t_i \ \wedge \ t'_s \leq t_s$, $I'' = \langle a', b' \rangle$



For computation, only case (2) is needed to implement. We don't need to detect case (1) because we are interested in intersection only. If case (1) happens none of the formulas from case (2) will be satisfied and the result will be an empty pHCube anyway.

- **Correct form (normal form).** Parameterized intervals are defined by conjunction of inequalities with parameters. Sometimes an interval can be expressed by more than one upper (lower) bound as the result of adding constraints, for example. In that case we are interested in finding normal form that is minimal and uniquely represents the interval.

  As shown on example in Section 4.1.4, the correct form of an interval is the minimum of the lower, resp. the upper bound of the interval. For computation, we apply the operator $min_{\mathcal{I}}$ defined at page 52.

  Let $ph(\mathcal{X}) = (\vec{I}, \varphi)$, where $\vec{I} = (I_1, \ldots, I_n)$, Suppose that variable x_i is bounded by another interval $I'_i(x)$. Then *correct form* $I_i^{cf}(x)$ of the interval is as follows:

$$I_i^{cf}(x) = min_{\mathcal{I}}(I_i(x), I'_i(x))$$

- **Adding constraints.** Guards or invariants in parametric counter automata are generally inequalities in the form $x_0 t_0 + \ldots + x_n t_n \prec t_{n+1}$, where $x_i \in \mathcal{X}$, $t_i \in AT(\mathcal{P})$ while PDBMs allow only expressions of the form $x_0 - x_1 \prec t$.

  Here, we will present adding a constraint on one variable. The extension to more variables is straightforward (only repeat this computation for all of them), thus we will not present it here.

  We distinguish two cases of constraints—a simple constraint $x \prec t$, and a general constraint $x_0 t_0 + \ldots + x_n t_n \prec t_{n+1}$:

  – *simple constraint* $x \prec t_g$

    Computation of the simple constraint is quite easy—we have to find intersection of current valuation of a variable and a guard. Let $I(x) = (\langle a, b \rangle, \varphi)$ be a constrained parameterized interval over variable x where $a = (t_i, \prec_i)$ is the lower bound and $b = (t_s, \prec_s)$ is the upper bound, $t \in AT(\mathcal{P})$. Let $x \prec_{gs} t_{gs} \wedge -x \prec_{gi} t_{gi}$ be a guard expressed as a constrained interval $I_g(x) = (\langle a_g, b_g \rangle, \varphi_g)$ where $a_g = (t_{gi}, \prec_{gi})$, $b_g = (t_{gs}, \prec_{gs})$. The resulting interval $I' = (\langle a', b' \rangle, \varphi')$ is the intersection of the lower, resp. upper bound of interval $I(x)$ with the guard $I_g(x)$ as follows:

$$I'(x) = I(x) \cap I_g(x)$$

– *general constraint* $x_0t_0 + \ldots + x_nt_n \prec t_{n+1}$
For a general constraint, the operation of adding a constraint is not exact. We have to compute an approximate value of bounds of a guard. We will show computation only for one variable x_0, for example. We require that term t_0 is a constant, $t_0 \neq 0$.

$$\begin{aligned} x_0t_0 + \ldots + x_nt_n &\prec t_{n+1} \\ x_0 &\prec \frac{t_{n+1}}{t_0} - \frac{t_1}{t_0}x_1 - \ldots - \frac{t_n}{t_0}x_n \end{aligned}$$

Now, we will substitute all variables $x_k, k \in \{0, \ldots, n\}$ on the right side by their upper bounds t_{ks}, if t_k is positive, or by their lower bounds t_{ki}, if t_k is negative. We get a bound of general constraint t_g:

$$x_0 \prec \underbrace{\frac{t_{n+1}}{t_0} - \frac{t_1}{t_0}\hat{t}_1 - \ldots - \frac{t_n}{t_0}\hat{t}_n}_{t_g} \quad \text{where } \hat{t_k} = \begin{cases} t_{ki} & \text{if } t_k \geq 0 \\ t_{ks} & \text{if } t_k < 0 \end{cases}$$

If t_0 is positive then we get an upper bound of the guard, i.e., $t_{gs} = t_g$. If t_0 is negative then we get a lower bound of the guard, i.e., $t_{gi} = t_g$. Now, we can specify a general constraint as in the previous case by a constrained parameterized interval $I_g = (\langle a_g, b_g \rangle, \varphi_g)$, where $a_g = (t_{gi}, \prec)$, $b_g = (t_{gs}, \prec)$. If the guard specifies only one bound on a variables (for example, the upper bound) the other bound will be an infinite bound, i.e., $(\infty, <)$.

A new interval $I'(\vec{x})$ after adding constraints will be the intersection of these two intervals as follows:

$$I'(\vec{x}) = I(\vec{x}) \cap I_g(\vec{x})$$

For example, suppose numerical intervals over x, y, z such that $1 < x < 3$, $2 < y < 5$, and $0 < z < 10$. Suppose a guard $x - 2y < 3$. Then, for variable x we get $x < 3 + 2y$, that is after substitution $x < 7$. For y we get $y > \frac{3}{2} + \frac{x}{2}$, i.e., $y > 3$. So, intersection of the guard and the original interval is $1 < x < 3$, $3 < y < 5$, and $0 < z < 10$.

- **Linear assignment.** Linear assignment is an operation used for setting a counter (variable) to a specific value. In parameterized counter automata, we allow a general assignment of the form $x := x_0t_0 + \ldots + x_nt_n + t_{n+1}$, where $t_i \in AT(\mathcal{P})$, $x_i \in \mathcal{X}$. The operation of assignment is exact for a simple assignment of the form $x := t$ where we set the upper and the lower bound of x to value $(t, \leq)$ and $(-t, \leq)$ so that $x \leq t$ and $-x \leq -t$ what is equivalent to $t \leq x \leq t$.

Let $I(x) = (\langle(t_i, \prec_i), (t_s, \prec_s)\rangle, \varphi)$ be a constrained parameterized interval over variable x. Let $x := t'$ be a simple linear assignment. Then a constrained parameterized interval $I'(x)$ after assignment $x := t'$ is as follows:

$$I'(x) = I(x)|_{x:=t'} = (\langle(-t', \leq), (t', \leq)\rangle, \varphi)$$

In case of general assignment of the form $x := x_0 t_0 + \ldots + x_n t_n + t_{n+1}$, we have to compute a new lower bound and a new upper bound on variable x. The lower bound t'_i is given by following computation:

$$t'_i = \hat{t_0}.t_0 + \ldots + \hat{t_n}.t_n + t_{n+1} \quad \text{where} \quad \hat{t_k} = \begin{cases} t_{ki} & \text{if } t_k \geq 0, k \in \{0, \ldots, n\} \\ -t_{ks} & \text{if } t_k < 0 \end{cases}$$

Term t_{ki} means the lower bound (infimum) of x_k, t_{ks} means the upper bound (supremum). The new lower bound t'_i is given by adding of lower bounds if coefficient t_k is positive and by minus of upper bounds if coefficient t_k is negative.

The upper bound t'_s is computed similarly:

$$t'_s = \hat{t_0}.t_0 + \ldots + \hat{t_n}.t_n + t_{n+1} \quad \text{where} \quad \hat{t_k} = \begin{cases} t_{ks} & \text{if } t_k \geq 0, k \in \{0, \ldots, n\} \\ -t_{ki} & \text{if } t_k < 0 \end{cases}$$

For example, suppose numerical intervals over variables x, y, z such that $1 < x < 3$, $2 < y < 5$ and $0 < z < 10$. Then, after general linear assignment $x := 2 - y + 2z$, a new interval of x will be $-3 < x < 20$, because $t'_i(x) = 2 - 5 + 0 = -3$ and $t'_s(x) = 2 - 2 + 2.10 = 20$.

Let $I(x) = (\langle(t_i, \prec_i), (t_s, \prec_s)\rangle, \varphi)$ be a constrained parameterized interval over variable x. Let $x := x_0 t_0 + \ldots + x_n t_n + t_{n+1}$ (shortly $x := \vec{t}$) be a general linear assignment. Then a constrained parameterized interval $I'(x)$ after assignment $x := \vec{t}$ on x is as follows:

$$I'(x) = I(x)|_{x:=\vec{t}} = (\langle(t'_i, \leq), (t'_s, \leq)\rangle, \varphi)$$

To define a linear assignment on parameterized hypercubes, we can simply consider a pHCube as a set of parameterized intervals and perform linear assignment on every interval.

Let $ph(\mathcal{X}) = (I_0(x_0), \ldots, I_n(x_n), \varphi)$ be a parameterized hypercube over variables $\mathcal{X} = (x_0, \ldots, x_n)$. Let $T(\mathcal{X})$ be a set of general linear assignments of the form $x_i := x_i^0 t_i^0 + \ldots + x_i^n t_i^n + t_i^{n+1}, i \in \{0, \ldots, n\}$. Then, a new parameterized hypercube $ph(\mathcal{X}')$ after assignment is as follows:

$$ph(\mathcal{X}) = ph'(\mathcal{X})|_{\mathcal{X}:=T(\mathcal{X})} = (I'_0(x_0), I'_1(x_1), \ldots, I'_n(x_n), \varphi)$$

where $I'_i(x_i) = I_i(x_i)|_{x_i:=\vec{t_i}}$.

- **Operation post().** Operation *post*() is a procedure similar to the one defined for PDBMs. It computes a set of configurations of the system starting from the given state of PCA C. For every state $s \in L$ and its valuation (ν, γ) expressed as parameterized hypercubes we go through all possible transitions given in the form (s_1, g, sop, s_2) (see definition of PCA at Section 3.2 for details) and compute a set of new configurations. Unlike timed variables, counters don't have time transitions, so we don't need an operation of elapsing of time as for PTAs.

Formally, the set of configurations form a transition system $S_C = (\mathcal{Q}_C, \mathcal{Q}_0, \rightarrow)$ of symbolic configurations where $q \in \mathcal{Q}_C$ is a node of S_C and $\tau = (q, ph_g, T, q')$ is a transition between q and q'. pHCube ph_g represents a guard over transition τ and T is a linear assignment.

Let $\mathcal{X} = (x_0, \dots, x_n)$ be a set of variables of the system and $ph(\mathcal{X})$ a current valuation of counters and parameters. Operation $post(q, ph(\mathcal{X}))$ over pHCubes can be computed as follows:

$$post(q, ph(\mathcal{X})) = \{(q', ph'(\mathcal{X})) \mid (ph(\mathcal{X}) \cap ph_g(\mathcal{X}))|_{\mathcal{X}:=T(\mathcal{X})}\}$$

where $ph_g(\mathcal{X})$ is a valuation of guards of the transition and $T(\mathcal{X})$ is a set of linear assignments over the transition.

- **Acceleration.** Acceleration over pHCubes uses an algorithm proposed in [AAB00] for PDBMs. Implementation requires operations manipulating pHCubes—subtraction, addition and multiplication to calculate control loops, conditions for acceleration and effect of acceleration.

  Let (q, ph) be a symbolic configuration and let θ be a control loop. Suppose that difference between (q, ph) and $post_\theta(q, ph)$, Δ, is equal to the difference between $post_\theta^2(q, ph)$ and $post_\theta(q, ph)$ Δ'. Formally, this condition is called $C1$ and is defined under the context of constraints expressed by φ_1 for (q, ph) and by φ_2 for $post(q, ph)$ as follows: $C1 : \forall \vec{p} \in \mathcal{P}^{n_p}, \forall \vec{n} \in \mathbb{N}^n \;.\; \varphi_2(p, n) \Rightarrow \Delta = \Delta'$ where $\vec{n} \in \mathbb{N}^n$ is a vector of extrapolation variables. We suspect that the effect of iterating θ will be the same at each iteration—to add increment Δ to the original set. This expectation is expressed formally by the following condition $C2 : \forall n \geq 0 \;.\; post_\theta^2((q, ph + n * \Delta)) = post_\theta((q, ph + (n+1) * \Delta))$. If C2 holds there is an exact approximation, i.e., by removing iteration and adding new extrapolation variables computation of acceleration is still exact—see [AC01] for prove.

  This is the main idea of acceleration. It helps to speed up computation of configurations.

  Implementation of acceleration requires simple operations *add*, *mul* and *sub* to add an effect of the control loop, multiplication by extrapolation variables or finding difference between two control states. These operations are simple and there are not presented in this text.

  In the following example we will demonstrate acceleration on a simple counter system. Suppose a system with one counter $x \in C$ and one parameter T with initial constraint $T > 0$, see Figure 4.8.

  The initial configuration of the system is $c_1 = (q_0, 0, T > 0)$ where 0 is the value of x and $T > 0$ is the initial constraint on the parameter. After applying control loop

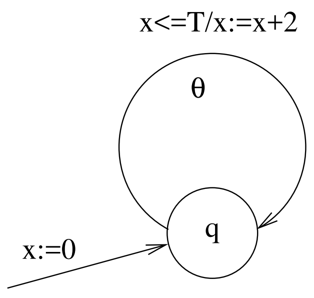


Figure 4.8: A simple parametric counter system

θ, a new configuration is c_2:

$$\begin{aligned} c_2 &= post_\theta(c_1) = \\ &= (q_0, (0 \leq x \leq 0) \cap (x \leq T), 0 < T)|_{x:=x+2} = \\ &= (q_0, 0 \leq x \leq 0, 0 < T)|_{x:=x+2} = \\ &= (q_0, 2 \leq x \leq 2, 0 < T) \end{aligned}$$

The difference of the valuations of c_2 and c_1 is the interval $\Delta_1 = \langle(2, \leq), (2, \leq)\rangle$. Another iteration of *post*() will have a similar effect:

$$\begin{aligned} c_3 &= post_\theta(c_2) = \\ &= (q_0, (2 \leq x \leq 2) \cap (x \leq T), 0 < T)|_{x:=x+2} = \\ &= (q_0, 2 \leq x \leq 2, 2 < T)|_{x:=x+2} = \\ &= (q_0, 4 \leq x \leq 4, 2 < T) \end{aligned}$$

The difference between valuations of c_3 and c_2 is $\Delta_2 = \langle(2, \leq), (2, \leq)\rangle$. We can see that $\Delta_1 = \Delta_2$, i.e., effect of the first and the second iteration of θ is to add the same value 2 to x. We check if condition $C2$ holds

$$\begin{aligned} &post_\theta(q_0, 2 * n \leq x \leq 2 * n, n < T \ \wedge \ 0 \leq n) = \\ &= (q_0, 2 * (n+1) \leq x \leq 2 * (n+1), 2 * n < T \ \wedge \ 0 \leq 2 * n) \end{aligned}$$

If it is satisfiable, we add a new control state to the set of configuration states and a new transition as shown in Figure 4.9.

Using acceleration, we reduce an infinite graph of symbolic configurations to the finite one what is a key point of successful parametric verification.

4.4 Implementation in TReX

The abstract data structure of a parameterized hypercube and operations over it that we presented in the previous section were implemented in the TReX verification tool [BCAS01] developed in LIAFA, Paris.

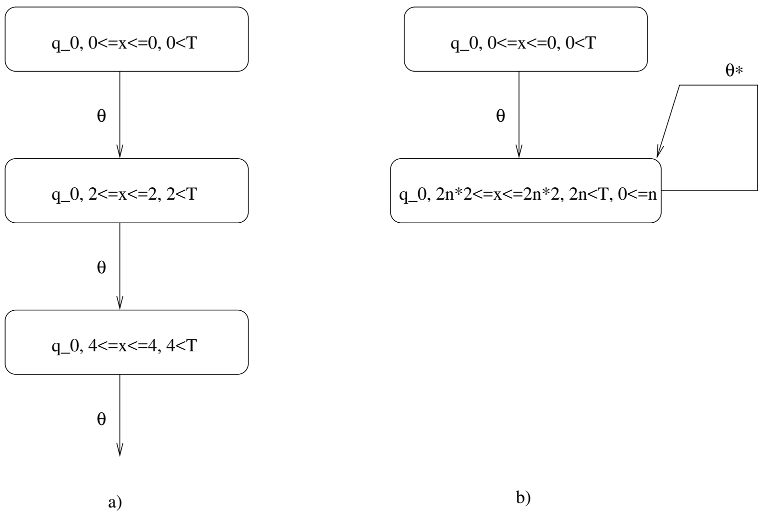


Figure 4.9: Effect of acceleration—before (a) and after (b) acceleration

TReX is a tool that enables an automatic analysis of automata-based models equipped with variables of different kinds of infinite-domain data structures with parameters. These models are parametric timed automata extended with integer counters and communicating through unbounded lossy FIFO queues.

Analysis of the model is based on symbolic reachability analysis. Symbolic structures are used to represent infinite sets of configurations, and forward/backward exploration procedures generate a symbolic reachability graph. The termination is not guaranteed, but efficient extrapolation techniques (e.g., acceleration, as showed before) are used to help it.

TReX allows to check on-the-fly safety properties as well as to generate the set of reachable configurations and a finite symbolic graph. The generated finite symbolic graph is a finite abstraction of the analysed model which can be used for finite-state model checking.

Architecture. The kernel algorithm used in TReX is generic and can be used for any kind of data structures for which it is possible to provide a symbolic representation structures, symbolic operations $post(), prec()$, and an extrapolation procedure—see Figure 4.10.

TReX is formed by three parts—modules:

1. Algorithms

   This module computes generic reachability states for every data structure. Algorithms implement operations post(), pre() and extrapolation techniques. Using such an algorithm it is possible to implement a new data structure and operation over it and add this data structure as a library to TReX. Before verification we can then specify what kind of data structure will represent given variables—we can set specific data domains for variables.

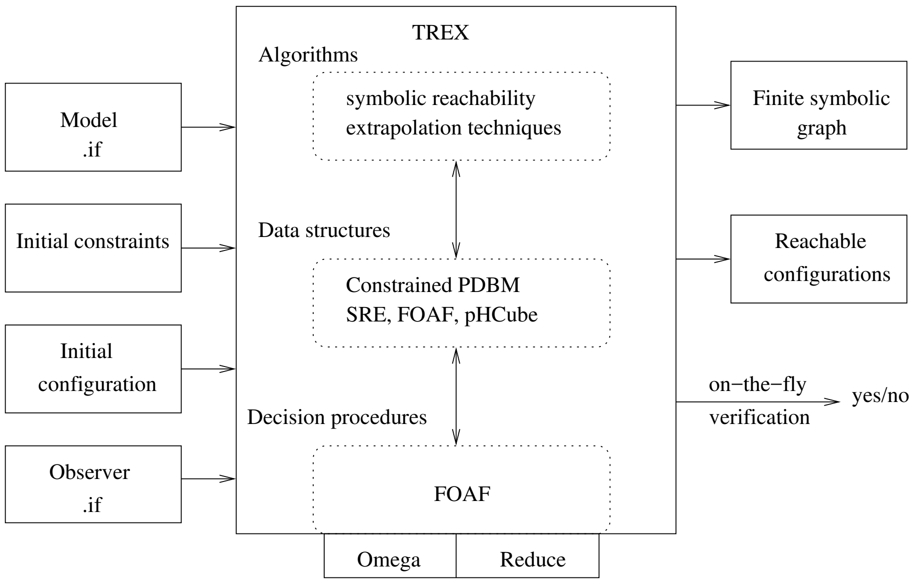


Figure 4.10: TREX architecture

2. Data structures

   Module Data structure includes several symbolical data structures—PDBM, SRE, FOAF. This module implements these abstract data structures and operations over them—e.g, inclusion, intersection, emptiness test etc.

3. Decision procedures

   For certain operations, TREX calls external decision procedures (REDUCE, OMEGA). This concerns manipulation of arithmetical formulas.

TREX provides the following data representations—simple regular expressions (SRE), see [AAB99, ABJ98], that represent configuration of lossy FIFO channels, and Constrained PDBMs [AAB00] that represent sets of configurations of clocks and counters with evolution depended on parameters. Efficient manipulation with first–order arithmetical formulas was implemented in a package FOAF (First–Order Arithmetical Formulas). This library is used to represent linear and non-linear constraints on parameters.

Satisfiability of formulas is checked using external decision procedures. TREX uses OMEGA [Ome96] for formulas over integers and REDUCE [Hea99] for formulas over reals.

Parameterized hypercubes. A library for manipulating pHCubes was created in C++. It implements intervals with symbolic bounds of generic type T defined using templates. A concrete specification of a class pHCube is as follows:

```
template <typename T>
class phcube {
 public:
    /* build/copy/destroy */
   phcube(var_t);                        /* universe */
   phcube(var_t, const bound_sym_t*);  /* constant for a bound */
   phcube(const phcube<T>&);           /* copy */
   ~phcube();
 public:
   bool is_included_in(const typename T::ty, const phcube<T>*, const typename T::ty) const;
   bool is_equal(const typename T::ty, const phcube<T>*, const typename T::ty) const;

protected:
    var_td  m_iDomain; /* domain of variables represented */
    var_t   m_iSize;   /* size of the vectors, computed using the domain */

    Vector<bound_sym_t *> *m_vlBound;   // lower bound
    Vector<bound_sym_t *> *m_vuBound;   // upper bound
};
```

The library provides basic operations required for manipulation of data structures. These operations formally introduced in the previous section are:

- `is_empty()`—test of emptiness,
- `is_universe()`—test of universality,
- `is_included()`—test of inclusion,
- `is_equal()`—test of equality,
- `intersection()`—intersection of two pHCubes,
- `add_constraint()`—intersection with the linear constraint,
- `assign_variable()`—linear assignment,
- `substitute_variable()`—transfer function for backward analysis,
- `set_bounds()`—set the upper and the lower bound of a variable,
- `add()`—addition of intervals,
- `sub()`—difference of intervals,
- `mul()`—multiplication of bounds,
- `get_extrapolation_vars()`—builds the list of extrapolation variables,
- `substitute_vars()`—an operation need by extrapolation.

Operations `intersection()`, `add_constraint()`, `assign_variable()`, `substitute_variable()` return as result lists of pHCubes because pHCubes are not closed under these operations.

A new package with a library for manipulating parameterized hypercubes will be a part of a new version of TReX.

Verification using TReX. TReX uses the If language [BFG+00] for model specification. It allows to specify initial constraints over a parameter, the initial symbolic configuration and the safety properties to be checked on-the-fly expressed by an observer.

On the output, TReX generates

- a finite symbolic graph in the Aldebaran format of Cadp [FGK+96] that can be used for further finite-state model checking and minimization, and
- a set of reachable configurations in format of InVeSt [BLO98] which is an invariant checker and can be used for verification of invariance properties of the model.

If we decide to perform an on-the-fly check of a safety property, the property should be given as an observer, ie. an extended automaton sharing transition labels with the input model. If the property is not satisfied, TReX generates a diagnosis trace showing the sequence of transitions from the initial state of the model to the state with bad behaviour. The symbolic configuration of the bad state can be used to synthesize constraints under which the safety property is satisfied.

TReX is also able to check some kind of liveness properties. It can synthesize fairness constraints stating the bounded iterability of some kind of loops. For further reading about liveness property we recommend [BCALS01] and [PS00].

4.5 Discussion

In this chapter, we introduced a new data structure for analysis and verification of parametric systems called parameterized hypercubes. This data structure is based on intervals and it is used mainly for counters in parametric counter automata. It cannot be used directly for timed automata because time domains require constraints of the form $x - y \prec t$ to specify time regions whilst counters do only $x \prec t$.

Original definition of parametric counter automata given in [AAB00] and shortly shown in Section 3.2 defines only simple guards and assignments over PCA. The reason is that a data structure used for PCA is implemented by PDBMs that permit only expressions of the form $x - y \prec t$ for constraints (guards) and $x := t$ for assignments.

Parameterized intervals allow constraints of the form $x_0t_0 + \ldots + x_nt_n \prec t_{n+1}$ and assignments $x := x_0t_0 + \ldots + x_nt_n + t_{n+1}$. This extends expressivity of the model towards systems based on polyhedra that generally allow linear inequalities for constraints.

On the other hand pHCubes utilize acceleration to speed up system analysis and effective implementation of operations like intersection, inclusion and test of emptiness. Because pHCubes encode only bounds of each variable (not their difference) they reduce space needed for data representation. While PDBMs require n^2 memory cells to represent data domain for n variables, pHCubes require only $2 * n$ (the lower and the upper bound).

The main contribution of parameterized hypercubes can be summarized into the following points:

- linear constraints in guards of transitions of the form $x_0 t_0 + \ldots + x_n t_n \prec t_{n+1}$,
- general actions (assignment) over transitions of the form $x := x_0 t_0 + \ldots + x_n t_n + t_{n+1}$,
- space reduction in comparison with PDBMs,
- normalization is not needed for intervals manipulation.

Chapter 5

How to Analyse Parameterized Timed Systems

Parameterized timed and counter systems are a subclass of timed/counter systems with specific features. In this chapter, we look more deeply to practical issues connected with analysing such systems. Some theoretical aspects were already mentioned in the chapter 3. Here, we focus more on practical hints and observations made by the author during formal analysis and verification he did within projects ADVANCE [Mat04a] and Liberouter [MSV05].

In the first section, we describe basic steps of formal analysis of a system. We point out issues raised by continuous time variables and parameters. The second part of this chapter shows our experience how to create a model of a system. We divide a model in three kinds of model components—a model of environment, a model of buffers, and a model of executive units. This was applied in Liberouter project [MSV05]. In the second section, we focus on buffers that were the main object of the author's research. In the third section, we give an overview of properties that can be analysed during verification.

5.1 Formal Analysis of Real-time Systems

Formal analysis of a system usually consists of four basic parts:

1. Creation of a model.
    - The model is an abstraction of the real system. The level of abstraction is the main question—we need to detect parts of a model that can be hidden in the abstract model without changing functionality of a system to be examined.
    - The model is described using formal language. Description is then unambiguous and precise.
    - Special request on an expression of a model is that description should be easy to read and understand. This contradicts with the previous point a little bit, especially for those people who are not familiar with reading of formal descriptions. A great advantage is to use tools with graphical interface where the

formal description is hidden behind simple figures as we can see, for instance, in UPPAAL.

- Description language can be usually an input language of a verification tool. There exist also convertors that translate descriptions from one language to another, for instance `SDL2IF` that translates a SDL description to IF language.

2. Validating the model.

   - The main issue of validation can be expressed by the question "Does the model correspond to the modelled system?" It is necessary to present the model to designers to see if it reflects their ideas of functionality of the system.

3. Definition of required properties.

   - Among such properties there can be logical correctness, response of the system etc.
   - Properties can be encoded as logical formulas in LTL or CTL, using some special syntax assertions (assertions—statements that have to be satisfied in every configuration of the system during its behaviour), invariants, reachability conditions, etc. These expressions are tightly connected to a verification tool we decided to use.

4. Simulation and/or verification of required behaviour.

   - Simulation is useful for making sure that the model corresponds to the modelling system. We can also observe the behaviour of the system for well-known conditions and input values and expected input values.
   - Nevertheless, simulation can never state that the system is correct or that a specified property is true for all input values and control states of the system.
   - Verification can prove that the property is valid for all input values and states of the system.
   - If verification ends with the result that the property is not satisfied we can find a counter-example showing why the property is violated. It helps us to detect flaws in system design. Even if the verification is not successful it may find different bugs than simulation due to a different way of traversing the state space of the system.

Timed and counter systems with parameters are generally undecidable. We showed some semi-algorithms (e.g. extrapolation) that helps termination of verification of parametric systems. Many times the analysis of such system does not finish and crashes because of the lack of memory. In this case, we can review our model in order to detect sources of complexity and eliminate them. Here we discuss some of common sources of complexity and possible ways how to deal with them.

Clocks. While analysing timed systems the complexity of computation increases by number of clocks. However, for many practical problems only a few clocks are sufficient. There exist algorithms for automated reduction of redundant clocks. Some tools (e.g., KRONOS) implement a technique that eliminate redundant clocks from a system—active clock reduction [DT98]. IF uses for clock reduction a technique based on slicing [Wei84]. However, in tools that do not implement these techniques and it is necessary to eliminate clocks manually.

Parameters. In parameterized systems values of variables are not computed precisely. Variables contain symbolic values (terms) with parameters that are constrained by logical formulas. During the analysis symbolic values are manipulated as shown in the previous chapters and also constraints develop. By passing a transition with guard or invariant a new constraint is added. Thus test of satisfiability of such formulas becomes more complicated and time consuming. This increases with the number of parameters.

If there are too many parameters, it is good to instantiate some of them (revert them to numerical constants) and try to analyse a model for some instances of such parameter. Of course, in this case we resign on parametric analysis but we can get partial results at least—as shown in the following case study.

Sometimes it is useful to define initial constraints or initial configuration of the system. Doing this we can reduce the size of a generated set of configurations.

5.2 Modelling a System

Modelling a system is to create an abstract model that describes the behaviour of the original system. The abstract model hides complexity of the original system and preserves just those system features that are interesting with respect to the system analysis. The level of abstraction is given by complexity of the system and also by expressivity of modelling language. Usually it reflects the description language of a tool we used for verification. For instance, dealing with timed systems, we can express time as discrete events (ticks) and model them using a counter.

While modelling communication systems (transmission protocols, programmable hardware) we noticed that a model of the class of such systems is useful to divide in three kinds of model components:

- environment—generators of incoming requests (packets), output units (waiting for a result),
- buffers (queues, channels)—they can be deterministic, stochastic, or non-deterministic; we can model lossy queues, delayed queues, etc.,
- executive units—processing units that perform operations over data in the system; they are mostly interesting from the point of view of delay on the processing requests, communication with environment, and data transformation.

Here, in this text we will be interested mostly in FIFO queues, for other components you can see [MSV05].

5.2.1 Modelling FIFO Queues

A FIFO queue (buffer, channel) is a typical abstract data structure that contains a sequence of stored data. FIFO queues are used to represent transmission channels, intermediate buffers between a processing unit and a memory, etc. Few verification tools (e.g. IF) implement FIFOs in such way that FIFOs are special data structure in the system which are manipulated using a language of the verification tool.

Here, we introduce patterns for modelling FIFO queues in the framework of automata without an explicit support of such structures which is the case of, e.g., UPPAAL. And even if queues are supported, the patterns may be used if one wants to customize the built-in implementation.

We can have lossy queues where some data may be lost. These are important to model communication channels. There are delayed queues where data are delayed. We can have bounded or unbounded queues, or we can also deal with queues where a symbolic constant value—a parameter—defines the maximum length of the queue. In parametric verification, we are then interested in the values for which the parameter satisfies the expected properties. This approach is called parameter synthesis [AHV93].

If we do not refer to the actual data stored in a queue we may abstract the queue to one value only—a counter of the number of items in the queue. This approach was used, for example, in the Liberouter project [MSV05].

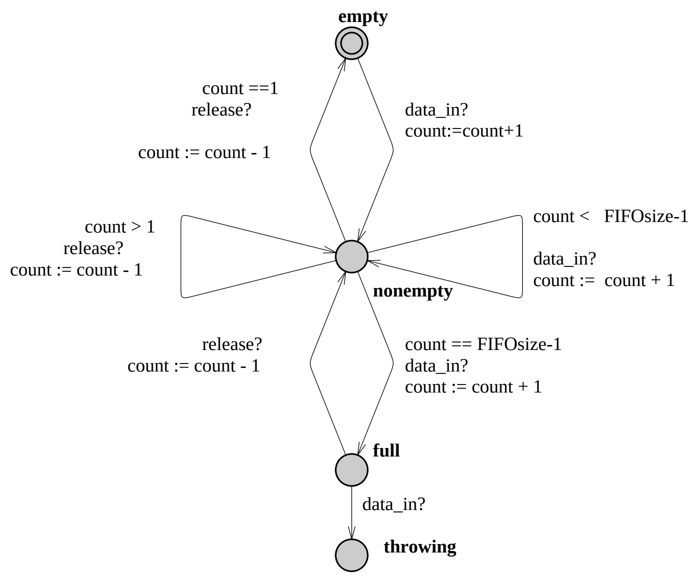


Figure 5.1: Model of a simple bounded FIFO queue

Simple Bounded FIFO Queues. A simple bounded FIFO queue can be modelled in an abstract way as a finite automaton with three control states—empty, nonempty, and full—augmented with a bounded counter *count* that represents a number of items currently present in the queue (Figure 5.1). If we add a new item in the queue, the counter *count* is incremented. If an item is removed from the queue, the counter is decremented. The maximum length of the queue is a constant value $FIFOsize$. Notation x? means reading from the queue into a variable x, x! means writing x into the queue.

Notice the special state named *throwing* that we have added to the model of a queue. This state forms a so-called observer—a state that is used for observing some features of the behaviour of the system being examined[1]. Here, we are interested in the queue overflow. If the queue overflows, the system moves to the state *throwing*. In the verification, we then define a property "the buffer will never overflow" as a logical statement "the system never enters the state *throwing*", $A \Box \neg throwing$ in the CTL logic.

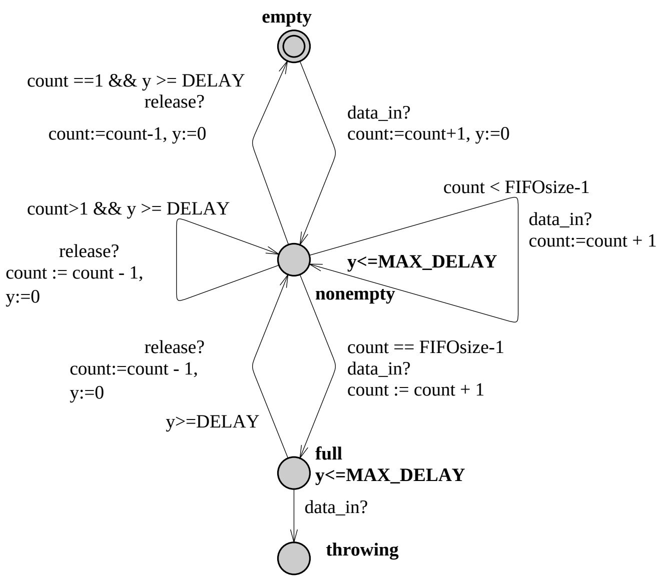


Figure 5.2: Model of a delayed FIFO queue

FIFO queues with delays. Delayed FIFO is a model of FIFO with time where it is guaranteed that every request is delayed by at least $DELAY$ time units. Delayed FIFO are modelled using Timed Automata [Alu99]. In figure 5.2 there is a delayed FIFO queue with clock y. Transitions that release an element of the queue are augmented with time constraints allowing to release an item only if the constraint is satisfied. The time of the release is non-deterministic $y \geq DELAY$. If we want to define a precise maximal delay of the queue we have to add an invariant $y \leq MAX\_DELAY$ to each state.

[1]Observers are usually independent automata that monitor the execution and decide whether a safety property is violated. In this case we reduce the observer to a state.

Lossy FIFO queue. Lossy FIFO queues can be with time or without time. In Figure 5.3 we consider untimed lossy FIFO queue.

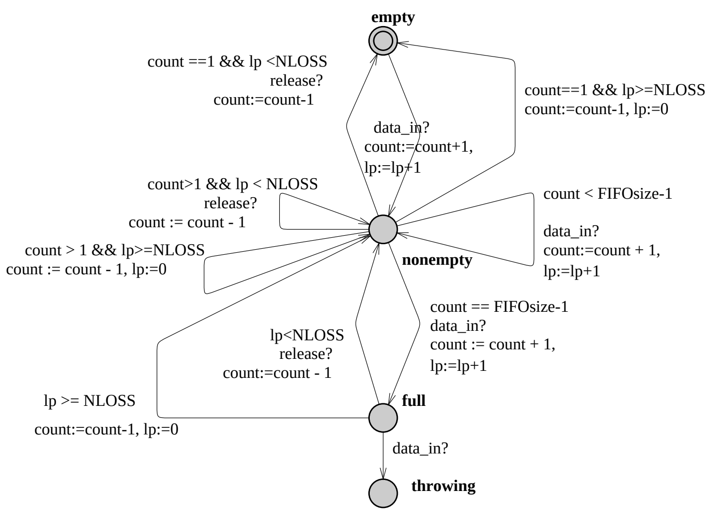


Figure 5.3: Model of a lossy FIFO queue

The model includes a special counter lossy period lp that counts a number of successfully sent elements. If the count reaches constant $NLOSS$, it generates a loss that means it decrements the number of items of the queue without sending it to the next block. In our model we have NLOSS equal to 5, i.e., every fifth element is lost. Using this abstraction we can model queues where we know the rate of losses. Distribution of losses is uniform, that means, we don't model chunks of losses.

5.3 Properties to be Verified

In this part we recall basic types of temporal properties to be verified, principles how they are implemented in verification tools and practical hints to describe common requirements using the following properties. The following classification of properties is based on [BBF$^+$01].

- **Reachability property**

  A reachability property states that "some particular state can be reached". Often we are mostly interested in negation of that property—we cannot reach a state that violates the property. In temporal logic it can be expressed as $EF\varphi$ that we can read as "there exists a path from the current state along states that satisfy φ".

  Reachability properties are relatively easy to verify. During model checking analysis we construct the reachability graph (configuration graph) of a system where we can check if a good/bad state is present.

For reachability analysis, we use two basic algorithms—forward and backward analysis.

- Forward analysis

  Starting from the initial states we build the set of reachable states using operation *post*(). We add iteratively successors of initial states until no new states can be added.
- Backward analysis

  Here we start from target states and add their predecessors using operation *pre*() until no new states can be added. Then we test whether some initial states are in the resulting set. The computation of predecessors in the case of automata with variables, guarded transitions and assignments is more complicated then forward analysis because it require solving equations, finding values of the variables, etc. During this computation, it is not sure if analysed states leads to the initial states because we get out of the well constrained states for which the system was designed. Another issue is to define a set of target states. This has to be done before backward analysis starts.

• **Safety properties**

A safety property is interpreted as "nothing bad ever happens". Many specifications are expressed naturally as safety properties.

A subset of safety properties is invariance. A system is said to be correct if and only if its reachable states all satisfy a φ requirement, that can be expressed in temporal logic as $AG\varphi$ for CTL, for instance. This is called invariants testing. Another approach is to use monitors (observers)—independent automata that monitor execution of the model of the system and accept only its legitimate behaviour.

Invariance is mostly implemented by testing reachability of negated formula, i.e, testing if $\neg\varphi$ is satisfied because $EF\varphi = \neg AG\neg\varphi$. One method is to perform forward reachability analysis from the initial states, and then check whether the intersection with the violating states $\neg\varphi$ is empty.

For other safety properties, i.e., $AG(\varphi_1 \Rightarrow (AG(\varphi_2 \Rightarrow AG\varphi_3)))$, the property is expressed using a property automaton. Then we test the intersection of the system automaton and the property automaton for emptiness.

In many cases it is not necessary to construct the entire state space of the modelled system. It is sufficient to generated only those states of the modelled system that are needed for testing the emptiness of the intersection of the modelled system with the property automaton. This technique is called on–the–fly model checking. The advantage of on–the–fly model checking is that when computing the intersection of the system automaton with the property automaton, some states may never be generated at all. Another advantage is that a counterexample may be found before completing the construction of the intersection of the two automata. Once a counterexample has been found and reported, there is no need to complete the construction.

If a state that violates the property is found, the analysis is stopped with the negative answer—property φ is not satisfied. Most tools generate also an error trace from the initial state so a designer can see conditions that violate property φ. In case of parametric analysis we can synthesize values of parameters that allows forbidding behaviour of the system.

- **Liveness properties**

  A liveness property states that "something good should eventually happen". The notion "something good" can be expressed as an configuration state R. Then analysis check if all states forward reachable from the initial states are backward reachable from a state R.

  In CTL we can express liveness property using combinator F. For instance "any request will ultimately be satisfied" is expressed as $AG(\text{req} \Rightarrow AF \text{ sat})$.

  In timed systems we are interested in so called *bounded liveness*. This is a liveness property that comes with a limited delay. For example, we can say that "any request will be responded within 15 seconds". However, this formula is in fact the safety property [BBF$^+$01].

  In model analysis we implement the bounded liveness using a monitor—a special automaton that checks if the requirement is violated. The monitor typically contains special states which are only reachable by violating executions. The monitor must act strictly as an observer of the original system, without changing its behaviour.

- **Deadlock-free property**

  Deadlock-freeness is a special property that states that "the system can never be in a situation where no progress is possible". This property is not safety property.

  It can be verified using formula $AGEX$ true in CTL but often special treatment is used.

- **Fairness**

  Fairness is a particular type of assumption that is very often needed for ensuring liveness. Numerous different notions of fairness has been defined. In [MP92] the authors introduce two kinds of fairness—weak fairness and strong fairness.

  Weak fairness (also known as justice) states that if transition t is enabled in every state from some point on, then it will eventually occur. The definition rules out executions where t never occurs although it is always ready to occur. Thus weak fairness is suitable for modelling assumptions such that each process of a system gets processor time.

  Strong fairness (or compassion) requires that if transition t is enabled infinitely many times, then it should also occur infinitely many times. With strong fairness one can specify, for instance, that a server does not systematically disfavour any one of its clients.

5.4 Summary

In this chapter, we discussed fundamentals of modelling, analysis and verification of a parameterized timed system. We recalled basic steps of formal analysis that include creation of a model, validation of the model, definition of required properties, and verification. While dealing with timed systems with parameters, number of clocks and parameters is important for the termination of the verification. It depends also on the relation between parameters (linear/non-linear), initial constraints over the parameters and number of processes of the model.

We also introduced a template to model communicating system. We distinguish three types of components of such system—environment, buffers, and executive units. In this text we dealt mainly with buffers and proposed a model of a simple bounded FIFO queue, a delayed FIFO queue and a lossy FIFO queue for automata-based verification languages where queues are not a part of the languages or where a customization is needed. These types of queues can be combined or enhanced by additional features.

In the last section we recalled basic types of properties for verification and principle of there usage and implementation. In the next chapter, we will show an application of this theoretical approach on a concrete example.

Chapter 6

Case Study

This part presents our experience with parametric verification of PGM protocol. The work was made at laboratory LIAFA under supervision of professor Ahmed Bouajjani and with collaboration of Mihaela Sighireanu from this institute. It was a part of ADVANCE project supported by the European Commission (FET project ADVANCE, contract No. IST-1999-29082). One of the deliverables of the project was to model and verify the PGM protocol [Bou02]. In this chapter we summarized results done by author of the thesis during this project. Full paper with results can be found at [Mat04b].

In our work we concentrated on three tools we used for analysis—UPPAAL, HYTECH, and TREX. Using combination of these tools we were able to automatically find relations between parameters of the protocol. Our results automatically obtained were consistent with previous results derived manually in [BS03].

This chapter is divided into four section. First section gives general comments on formal analysis of a system. It mentions steps to create a model, its specification in language of a verification tool and steps of verification. Second section shows on an example of protocol PGM the phase of modelling. Third section presents our verification in three verification tools and describes our results. The last section summaries our results and experience with verification of the timed system with parameters.

6.1 Formal Analysis

In this section we give an overview of basic steps that we did during analysis of PGM protocol—creation of model, specification of properties, and verification.

(i) System specification. We created an abstract model and described it using extended timed automata with parameters. The automata communicate using shared variables.

(ii) Implementation in verification tools. Formal model was translated into input languages of UPPAAL, HYTECH, and TREX. Transformation was straightforward, because UPPAAL and TREX use timed automata for model description. HYTECH models systems using hybrid automata. Timed automata can be considered as a subset of hybrid automata, so translation of the model to HyTech was not difficult too.

(iii) Verification. Because Uppaal does not support parameters, we instantiated parameters and verified the model for a set of different values. Parametric verification was made in HyTech and TReX. During verification we faced a problem of non termination. If the analysis did not terminate we used following algorithm:

1. We checked traces/runs to find a source of non-termination.
2. We refined a model—we put an additional restriction on parameters, instantiated some parameters, etc.
3. Verification was repeated.

Parametric verification resulted in relations between parameters (parameter synthesis) that satisfied the observed property.

6.2 Modelling PGM

This section deals with the modelling of PGM protocol. Our model of PGM was tuned during time of analysis—a few features were modelled not very precisely, first models were too large to finish verification etc. Here, we present the last version of our model where we were able to prove desired properties.

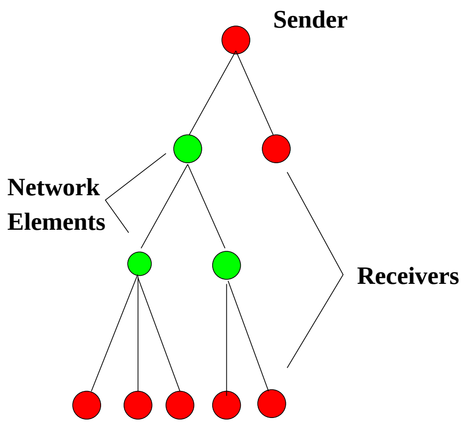


Figure 6.1: PGM—multicast transmission.

PGM protocol. PGM protocol defined in [SFC^{+}01] is a complex multicast protocol. It works on a network of nodes with multiple senders and multiple receivers. Its dynamic behaviour is depicted in Figure 6.1. Transport-layer originators of PGM data packets are referred to as senders, transport-layer consumers of PGM data packets are referred to as receivers, and network-layer entities in the intervening network are referred to as network elements.

In the normal course of data transfer, a sender multicasts sequenced data packets (ODATA), and receivers unicast selective negative acknowledgments (NAKs) for data packets detected to be missing from the expected sequence. Network elements forward NAKs hop-by-hop to the source, and confirm each hop by multicasting a NAK confirmation (NCF) in the response to the interface at which the NAK was received. Repairs (RDATA) may be provided by the sender in the response to a NAK.

Since NAKs provide the sole mechanism for reliability, PGM is particularly sensitive to their loss. To minimize the NAK loss, PGM defines a network-layer hop-by-hop procedure for reliable NAK forwarding—see Figure 6.2.

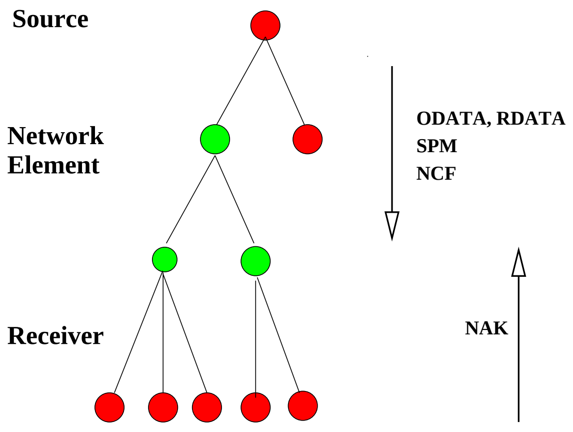


Figure 6.2: Data packets defined in PGM.

In our approach we abstract the model to a simple one-sender and one-receiver system. Joining and leaving multiple nodes during session can be considered as nodes missing data [BS03].

Abstract model. Analysing the full PGM protocol is beyond limits of current verification tools because of

- dynamic topology—joining/leaving out a node,
- multiple senders,
- a lot of different packet types (`SPM`, `NCF`, `NAK`),
- a lot of processes, counters and clocks.

Possible sources of complexity are the number of clocks and counters, the number of parameters, non-linear relations between variables. However, by combination of different tools for analysis we were able to prove the reliability property.

Our abstract model is based on a global view of the protocol running in the sender and one of its receivers, as presented on Figure 6.3. The intermediate network between the sender and the receiver is abstracted into a unreliable, unbounded FIFO queue. Only data packets (`ODATA`) are transmitted between the sender and the receiver, the other packets (`SPM`, `NAK`, `NCF`, `RDATA`) are abstracted also.

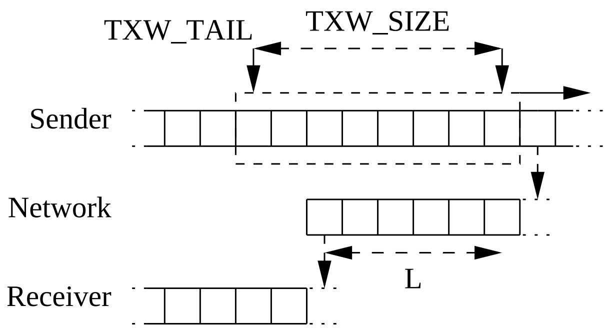


Figure 6.3: Abstract model of PGM.

Using the abstract model we can abstract from individual packet numbers. We are not interested in a precise sequence number to detect losses. When the receiver receives a packet it is informed by global variable that there was a loss (or multiple losses) preceding the incoming packet. The receiver knows an actual size of the data in the network (variable `L`—the length of FIFO queue), the number of old data in the sender's transmission window and the speed of the transmission. From these values it is possible to detect if a lost packet can be recovered. In Figure 6.3 we can see that the number of packets present in transmission window for recovery is `TXW_SIZE - L`. The real possibility of recovery depends on the speed of transmission expressed by parameter `RATE`.

Formal description. For verification purposes we use extended timed automata with parameters (see previous section) to describe our abstract model. The choice of the formalism was made with regard to verification tools we intended to use. Our prime goal was to synthesize parameters of a protocol. So we looked for tools that implement parametric verification. Because the observed system is a communication protocol working in real-time, the need of explicit continuous time was raised. Following these requirements we decided to use extended timed automata with parameters to formally describe PGM protocol.

Our PGM model is composed of three automata—a sender, a network and a receiver with six parameters, one finite variable, two clocks, two counters and two communication channels, see Figures 6.4, 6.5, and 6.6.

The automata work simultaneously and are synchronized by rendez-vous on gates SN and NR. They communicate using shared variables L and lp. The states labeled by C are urgent states, i.e., states where the time is not allowed to advance.

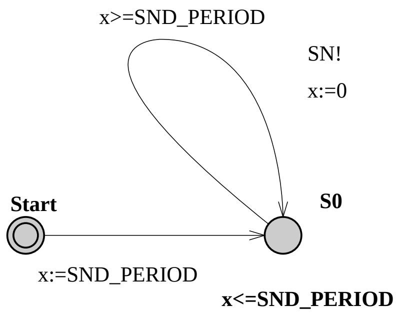


Figure 6.4: PGM model - sender

The sender. The sender generates new data each period (SND_PERIOD). The data sent are stored in the transmission window that advances each time new data are sent. The transmission window is fully opened during the session to recover as many data packets as possible. If data loss is detected, we test if an original data packet is in the transmission window. If not, a non-recoverable data loss has happened and the full recovery property is violated.

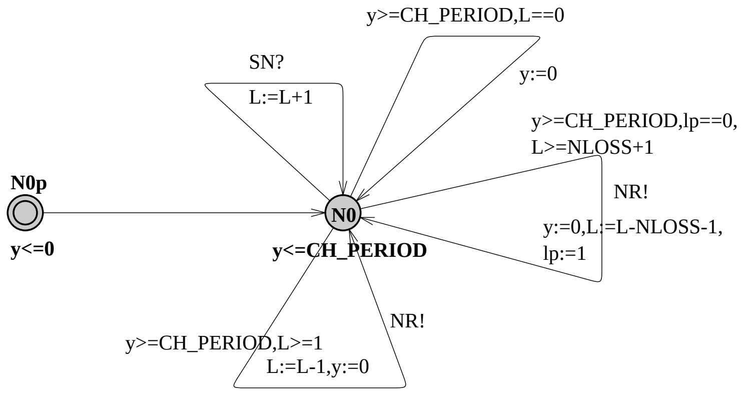


Figure 6.5: PGM model - network

The network. The network automaton models the transmission channel between the sender and the receiver with transmission delays and non-deterministic losses. The network receives data from the sender and increments the length of buffer L. The buffer is unbounded, it can grow without limitation. The network element either delivers data to the receiver with the speed defined by CH_PERIOD parameter or multiple data are discarded in order to model losses during transmission. The model allows `NLOSS` data packets to be lost, `NLOSS` being a parameter. The initial buffer length is set to `BUFFER_LENGTH`, which means the system is in process of communication—we don't model opening and closing stages of communication.

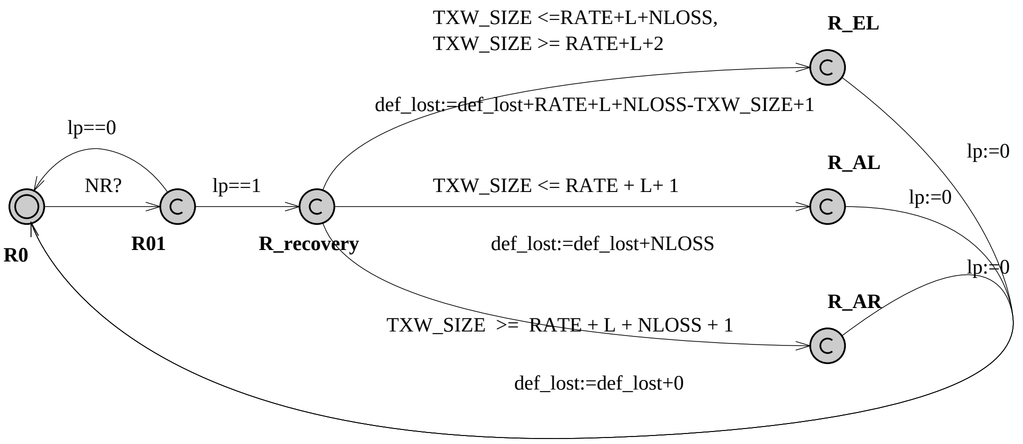


Figure 6.6: PGM model - receiver

The receiver. The receiver is informed about losses using a global variable lp. When a loss occurs the receiver calculates possibility of recovery. The result depends on `TXW_SIZE`, `BUFFER_LENGTH`, `RATE` and the current length of the buffer `L'`.

By reasoning about the recovery of transmitted data we distinguish three possible cases—every lost packet can be recovered, some lost packets can be recovered or nothing can be recovered. This depends on the size of the sender's transmission window, the speed of transmission, the delay in the network etc. These cases can be described by following manually obtained results:

∀R All lost packets may be recovered (full recovery) if $\mathtt{TXW\_SIZE} > \mathtt{RATE} + \mathtt{L}' + \mathtt{NLOSS}$, state R_AR,

∀L None of the `NLOSS` lost packets may be recovered (no recovery) if $\mathtt{TXW\_SIZE} \leq \mathtt{RATE} + \mathtt{L}' + 1$, state R_AL, or

∃R Some of the lost packets may be recovered (partial recovery) if $\mathtt{TXW\_SIZE} > \mathtt{RATE} + \mathtt{L}' + 1$ and $\mathtt{TXW\_SIZE} \leq \mathtt{RATE} + \mathtt{L}' + \mathtt{NLOSS}$ (state R_EL) .

The full recovery may be done for the first case if the parameters satisfy constraint $\mathtt{SND\_PERIOD} \geq \mathtt{CH\_PERIOD} \land \mathtt{TXW\_SIZE} \geq \mathtt{RATE} + \mathtt{BUFFER\_LENGTH}$. This constraint on parameters was obtained manually in [BS03]. In this paper we focus on automatic Al synthesis of parameters. However, it is interesting to compare the manually obtained results with output of verification tools listed in the following section. It can be seen that the results are consistent.

6.3 Tools for Parametric Verification

In parametric verification we used three tools—HyTech, TReX and Uppaal. In this part we introduce the tools and our results. As mentioned in [AHV93], a large class of parametric verification problems is undecidable. In [AAB00] the authors introduce a semi-logarithmic approach based on an expressive symbolic representation, parametric DBMs, and extrapolation techniques that allow one to speed up reachability analysis and help its termination. We will see how important an effective extrapolation technique is in comparison with TReX and HyTech.

6.3.1 HyTech

HyTech [HHWT95] is a tool for analysis of linear hybrid automata [ACHH93]. A hybrid automaton is a mathematical model for hybrid systems that models both their discrete and continuous behaviour. Hybrid automata can be considered a generalization of timed automata with continuous variables. Timed automata have one type of continuous variables—clocks. Generally, hybrid systems are undecidable [HHWT95]. Linear hybrid systems form a subclass of hybrid systems which can be analysed semi-automatically [ACHH93]. Invariants, guards and actions in linear hybrid systems depend linearly on time and other variables.

HyTech is a symbolic model checker for linear hybrid automata. The ability of HyTech to perform parametric analysis is an important feature. It is able to synthesize parameter values, i.e., to find the correct values for the parameters so that the system will satisfy a specified property.

Model description. HyTech takes a description of a model as in input in the form of linear hybrid system and analysis commands. System description contains variables of several types: *discrete, clock, stopwatch, parameter* and *analog.* Guards and constraints are composed of linear terms and expressions. Each automaton is composed of locations and their transitions. Locations are labeled with their invariants. Transitions contain guards with enabling conditions and the successor location. There must be provided an initial state of an automaton and an initial value of the variables.

Model analysis. Analysis in HyTech is specified by two parts: declaration of regions, and a sequence of analysis commands. Analysis commands provide a means of manipulating and outputting regions. At any time instant, the state of a hybrid automaton is specified by a location and constraints on variables. This is called a region. HyTech computes the forward reachable region by finding the limit of the infinite sequence *I, post(I),* $post^2(I)$, *...* of regions. All timed safety requirements, including bounded-time response requirements, can be verified using the reachability set. However, the iteration scheme is a semi-decision process: there is no guarantee of termination.

In our first approach, we computed the reachability set of the system. The property to be verified in the system was expressed in negative form using a region that violates the property: $final\_reg := def\_lost > 0$. Term $def\_lost > 0$ describes states where the recovery property is not satisfied, i.e. number of definitely lost packets is greater then zero. Firstly, HyTech generates a set of all reachable configurations of the system. Then intersection with specified property is applied on the set. If the property holds we get a non-empty result in the form of equations between parameters that satisfy our model and specified conditions. Declaration of analysed region and analysis commands in HyTech for the first approach is following:

```
-- definition of initial and final region
init_reg, final_reg: region;

-- region inizialization
init_reg := loc[sender] = S0 & x = SND_PERIOD & loc[Node] = N0 & y =
  0 & L = BUFFER_LENGTH & lp = 0 & loc[receiver] = R0 & def_lost = 0
  & RATE >= 1 & TXW_SIZE >= 1 & NLOSS >= 1 & BUFFER_LENGTH >= 1 &
  CH_PERIOD >= 1 ;

-- a violation state (final_reg)
final_reg :=  def_lost > 0;

-- analysis
reached := reach forward from init_reg endreach;
prints "------------";
print omit all locations
        hide non_parameters in reached & final_reg endhide;
```

For the first approach computation did not terminate. In symbolic model checking there are very important techniques like acceleration that help to speed up and terminate the analysis. For above written example HyTech had a problem to accelerate and after few hours the computation failed because of the lack of memory.

The second approach analyses the model on the fly. At first, the nearest reachable region is computed using *post()* operation and then, immediately intersection of the region and the undesirable property *final_region* is tested. If the intersection is non empty, non-reliable state was reached. If the intersection is empty, we continue in the iteration. We cannot find all states satisfying the property but we can determine states that violate the property and synthesize parameters for non-allowed states. In HyTech, the second approach is written as follows:

```
init_reg, reached,old, final_reg: region;

init_reg := loc[sender] = S0 & x = SND_PERIOD & loc[Node] = N0 & y =
  0 & L = BUFFER_LENGTH & lp = 0 & loc[receiver] = R0 & def_lost = 0
  & RATE >= 1 & TXW_SIZE >= 1 & NLOSS >= 1 & BUFFER_LENGTH >= 1 &
  CH_PERIOD >= 1 ;

final_reg :=  def_lost > 0;

-- initialize region reached:
reached := init_reg;
prints "------------";
while empty(reached & final_reg) do
  old:= reached;
  reached:=post(old);
  print diff(reached, old);
endwhile;

prints "reached & final_reg:";
print omit all locations hide non_parameters in reached & final_reg
endhide;
```

Results. During the analysis of PGM we distinguish four different cases depending on the speed

- *Case 1:* SND_PERIOD $>$ CH_PERIOD - the rate of arrivals is less than that of departures, the size of the queue converges to zero. Following constraints on parameters were synthesized:

  ```
  CH_PERIOD < SND_PERIOD & CH_PERIOD >= 1 & NLOSS >= 1 & NLOSS <= BUFFER_LENGTH
  & RATE >= 1 & TXW_SIZE >= 1 & TXW_SIZE + NLOSS <= RATE + BUFFER_LENGTH + 1
    |
  RATE >= 1 & NLOSS <= BUFFER_LENGTH & TXW_SIZE + NLOSS >= RATE + BUFFER_LENGTH + 2
  & CH_PERIOD < SND_PERIOD & CH_PERIOD >= 1 & TXW_SIZE <= RATE + BUFFER_LENGTH
  ```

  The result shows that for TXW_SIZE $\leq$ RATE$+$BUFFER_LENGTH$-$NLOSS (the first part of the formula) nothing can be recovered and for TXW_SIZE $>$ RATE $+$ BUFFER_LENGTH $-$ NLOSS (the second part of the formula) some losses can be recovered. This corresponds to results obtained by TReX - see later.

- *Case 2:* SND_PERIOD $=$ CH_PERIOD—arrivals are of the same speed as departures, the size of the queue decreases to zero by the number of losses NLOSS these are non-deterministic losses in the queue.

  The constrained obtained are the same as in the previous case.

- *Case 3:* CH_PERIOD/SND_PERIOD $>$ NLOSS—arrivals are faster then departures and losses, the queue grows beyond any limits.

  For this case and case 4, we introduce a new parameter $q =$ CH_PERIOD/ SND_PERIOD, and we consider that $q \geq 2$. Parameter synthesis obtained by HyTech for $q = 2$ as follows.

  ```
  q >= NLOSS + 1 & SND_PERIOD > 1 & BUFFER_LENGTH >= 1 & NLOSS <= BUFFER_LENGTH + 1
  & RATE >= 1 & TXW_SIZE + NLOSS <= RATE + BUFFER_LENGTH + 2 &TXW_SIZE>= 1 & NLOSS >= 1
   |
  q >= NLOSS + 1 & NLOSS <= BUFFER_LENGTH + 1 & RATE >= 1 & TXW_SIZE + NLOSS >= RATE
  + BUFFER_LENGTH + 3 & SND_PERIOD > 1 & TXW_SIZE <= RATE + BUFFER_LENGTH + 1
  ```

  This is for $q = 2$. If we set q equal to $\{3, 4, \ldots\}$ we obtain similar results that differ by constants in relation with TXW_SIZE.

- *Case 4:* NLOSS $>$ CH_PERIOD/SND_PERIOD > 1—arrivals are faster than departures but not enough to fill the losses between two deliveries, the size of the buffer does not grow fast enough because of the losses.

  The experiments and the results are similar to the third case.

6.3.2 TREX

TREX [BCAS01] is a tool that allows one to analyse automatically automata-based models equipped with variables of different kinds of infinite domain and with parameters. The models are parametric timed automata extended with integer counters and communicating through unbounded FIFO queues.

The verification technique is improved with an efficient extrapolation technique. TREX allows on-the-fly model checking as well as the generation of the set of reachable configuration and of a finite symbolic graph.

Model description and analysis. A model of the system is specified using an input language that is a subset of IF language [BFG$^+$00]. A model contains timed automata with counters, parameters and gates for synchronization. In .cnd file we specify initial constraints on parameters to help its termination. The output of the verification is a resulting finite graph (.sg), a set of symbolic configurations (.res) and a list of traces/runs (.tr) over a symbolic configuration graph.

Using a set of traces and a graph of symbolic configuration we can observe behaviour of the system and find a relation between parameters satisfying desired property. In our case we search for configurations where the number of definitely lost packets is zero. This configuration satisfies the full recovery property.

In HYTECH we were able to verify only counter-examples, i.e., configurations where the property was violated. We did not succeed to generate a full set of reachable configuration. On the contrary, TREX successfully generates a full graph of all reachable configurations. From this graph we can synthesize parameters satisfying the desired property. For instance, in Figure 6.7 we can see all possible traces (runs) of the model for which the desired property $def\_lost = 0$ holds. The graph was generated from TREX (.tr file) for the full recovery property (def_lost = 0) and `CH_PERIOD`= `SND_PERIOD`. We can observe a dependency of an initial value of the buffer `BUFFER_LENGTH` on current length L of the buffer for all recovery property.

Results. The results obtained from a set of symbolic configurations cover all three cases of recovering losses. We can see that relations in the case of no recovery and of partial recovery correspond with HYTECH results and full recovery with manually obtained relation. The example is for `SND_PERIOD` > `CH_PERIOD`:

- R_AR—full recovery
  txw_size ≥ rate + buffer_length and buffer_length ≥ nloss + 1

- R_EL—partial recovery
  txw_size ≥ rate + buffer_length -nloss - n3 - 1 and twx_size ≤ rate + buffer_length - n3 - 3 and buffer_length ≥ nloss + n3 - 3 and buffer_length ≥ n3 - 2 and n3 ≥ 0

- R_AL—no recovery
  txw_size ≤ rate -nloss + buffer_length - n3 - 1 and buffer_length ≥ nloss + n3 - 2 and buffer_length - n3 - 1 ≤ 0 and n3 ≥ 0

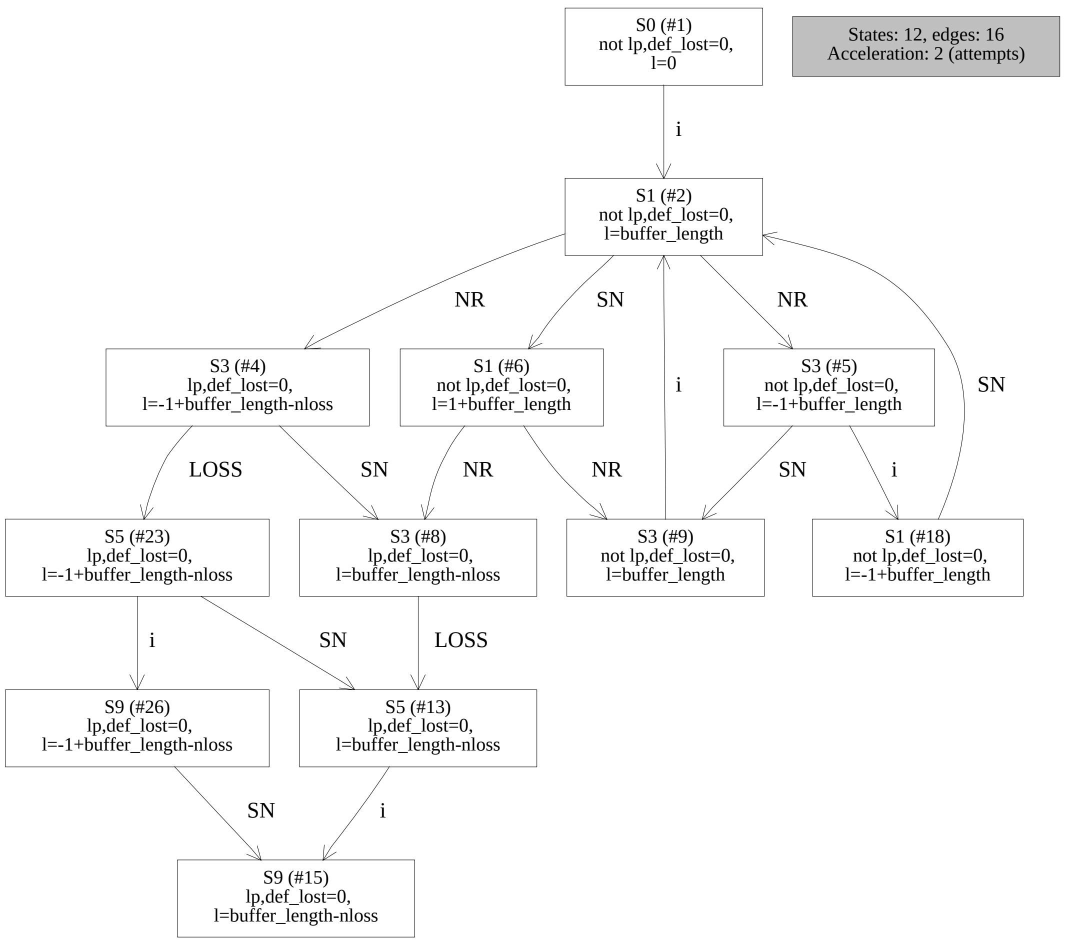


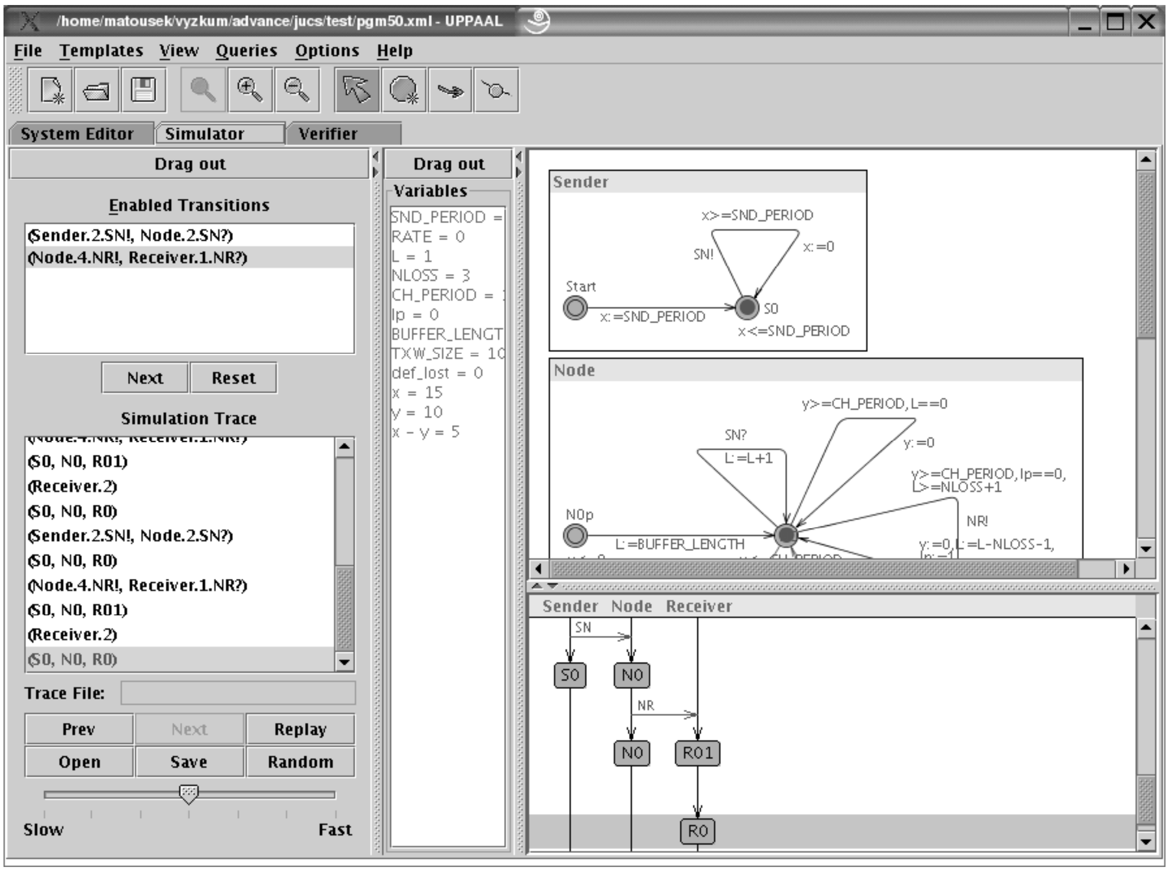
Figure 6.7: TREX—symbolic reachability graph

Variables $nX \in \mathbb{N}$ are called iteration variables. They are used by acceleration procedures to describe symbolically the number of iterations of detected loop.

6.3.3 UPPAAL

UPPAAL is a tool for validation and verification of RT systems developed by collaboration of Uppsala University in Sweden and Aalborg University in Denmark [PL00]. A model is described using timed automata. Verifier checks specified properties that are expressed using simple temporal logic with operators `E<>`, `A[]`, `E[]`, `A<>`. UPPAAL verifies an existence of a deadlock using special property `A[] not deadlock`.

Model description. A part of UPPAAL tool is a simulator that performs simulation on a specified model. We used the simulator especially in the first stage of the model specification where we tuned our model and compared it with the given protocol—see



Enabled Transitions:

```
(Sender.2.SN!, Node.2.SN?)
(Node.4.NR!, Receiver.1.NR?)
```

Simulation Trace:

```
(S0, N0, R01)
(Receiver.2)
(S0, N0, R0)
(Sender.2.SN!, Node.2.SN?)
(S0, N0, R0)
(Node.4.NR!, Receiver.1.NR?)
(S0, N0, R01)
(Receiver.2)
(S0, N0, R0)
```

Variables:

```
SND_PERIOD =
RATE = 0
L = 1
NLOSS = 3
CH_PERIOD =
lp = 0
BUFFER_LENGT
TXW_SIZE = 1(
def_lost = 0
x = 15
y = 10
x - y = 5
```

Figure 6.8: Simulation of PGM in Uppaal.

Figure 6.8. A model was described graphically using a built-in editor. The specification is visual and enables the first check of the consistency of the model. Specification of our model in UPPAAL is in Figures 6.4, 6.5, and 6.6.

Model analysis. In our project we did the verification using UPPAAL states where `def_lost > 0` and `def_lost = 0`. UPPAAL does not support parametric verification, so we instantiated parameters. Using UPPAAL we were able to prove that our results obtained by HYTECH and TREX are consistent and that our model is deadlock-free.

Results. In this part we briefly show obtained results.

- *Result 1:* relation between the current length of the buffer and the number of definitely lost packets for different values of parameters. constants: `CH_PERIOD`: 10, `SND_PERIOD`: 15, `RATE`: 0

TXW_SIZE	10	10	10	10	10	12	11	10	10	12	12	13	14	10	10	2
NLOSS	3	3	3	3	3	3	3	3	3	3	4	4	4	4	5	5
BUFFER_LENGTH	0	5	8	9	10	10	10	11	12	12	12	12	12	5	5	4
max L	1	6	9	10	11	11	11	12	13	13	13	13	13	6	6	5
def_lost	0	0	0	1	2	0	1	3	3	2	2	1	0	0	0	0

From the above table we see that:

1. max L = BUFFER_LENGTH+ 1
   Because of CH_PERIOD < SND_PERIOD the buffer cannot grow except for the initial phase of the transmission.
2. $L \geq$ NLOSS + 1
   We can produce losses only if the queue is greater then NLOSS+1 packets.

- *Result 2:* relation between TXW_SIZE and BUFFER_LENGTH.

TXW_SIZE	10	7	7	7	7
NLOSS	3	3	4	5	5
BUFFER_LENGTH	5	5	5	5	6
def_lost	0	0	0	0	1

The second table shows that we have a definitely lost packets if $L >=$ NLOSS + 1 and if TXW_SIZE > BUFFER_LENGTH + 1. Losses occur only if we reach L $\geq$ NLOSS + 1, so BUFFER_LENGTH $\geq$ NLOSS + 1.

- *Result 3:* relation for CH_PERIOD > SND_PERIOD.

  If we test our model for CH_PERIOD > SND_PERIOD, i.e, (CH_PERIOD= 2 SND_PERIOD, SND_PERIOD=10,CH_PERIOD=20), than we obtain following results:

  1. queue L grows beyond every limit,
  2. def_lost is at most NLOSS.

6.4 Discussion

Analysis and verification of parameterized models is difficult because the appropriate problems are usually, in general, undecidable. In this paper, we showed our experience and results of synthesis of the parameters for PGM protocol. We detected the following sources of complexity that prevent the tool to finish the verification:

The number of clocks and counters. The number of clocks and counters in the model can cause that reachability analysis does not terminate. One of the suggestion for dealing with it is to abstract out some of the clock or counter variables. There are as well several automated approaches—a technique based on slicing [Wei84], or active clock reduction [DT98]—that are implemented in some tools (e.g., KRONOS, IF). However, the issue of reducing the number of clocks and counters has not been very well explored yet and it is a possible area for future research.

Parameters. Parameters form another type of complexity. To speed up verification it is useful to set strict initial constraints (bounds) on parameters that limit the size of the generated state space. For instance, by setting parameter CH_PERIOD > 0 the tool is prevented to explore states where CH_PERIOD < 0.

Nonlinear relation between parameters. During the analysis PGM protocol we discovered a big issue concerning parameters that are related non-linearly. Current tools and verification techniques cannot solve this problem in a satisfactory way. As a solution we propose to instantiate parameters which are non-linear. For instance, we introduced substitution of SND_PERIOD$= 2 *$ CH_PERIOD, then for SND_PERIOD$= 3*$ CH_PERIOD, SND_PERIOD$=$ $4 *$ CH_PERIOD, etc.

Analysis does not terminate. It is not surprising when analysis does not terminate or crashes because of the lack of memory. Timed systems with parameters are generally not decidable. The termination of the analysis is sensitive to several aspects. In our case study we synthesize following recommendations:

- Explicit on-the-fly verification. This was extremely useful in verification with HYTECH.
- Analysis of traces. We used a symbolic reachability graph - at least a part of it—to find a beginning of the non-termination. By setting the initial constraints on parameters we can refine the model and narrow the state space. It may help the verification to terminate.

We worked with three different tools—UPPAAL, HYTECH and TREX. In combined analysis using these tools we were able to find constraints on the parameters that satisfied desired property—the full recovery property. We proved that for SND_PERIOD $\geq$ CH_PERIOD $\wedge$ TXW_SIZE $\geq$ RATE $+$ BUFFER_LENGTH the property is satisfied. In comparison with previous work [BS03], our result was obtained automatically.

Analysis using UPPAAL helped us to visually describe our model and simulate its behaviour. We used its verifier to prove the full recovery property for a model with instantiating parameters. Verification of deadlock detection proved the consistency of the model. For parametric analysis we used HYTECH and TREX. HYTECH had problems with termination so we verified only the negation of the property and detected configurations that violate that property. This covers two cases—partial recovery and no recovery. Using TREX, we obtained a full graph of symbolic configurations and observed relations between parameters. We synthesized parameters for all three cases—full recovery, partial recovery and no recovery. The results were consistent with those obtained manually in previous work, and those obtained using HYTECH and UPPAAL.

Tool	**Formal specification**	**Data structure**	**Params**	**Acceleration**	**Notes**
HyTech	hybrid automata	polyhedra	yes	not very good	problem with termination
Uppaal	extended timed automata (simple version)	DBMs	no	yes	includes simulator, graphical interface
TReX	extended timed automata	PDBMs	yes	yes	generates symbolic reachability graph

Table 6.1: Features of verification tools

In Table 6.1, a brief comparison of verification tools that were used for parametric verification of PGM protocol is shown.

Chapter 7

Conclusion

The main goal of the thesis was to explore common data structures used for parametric verification and propose a new structure that would increase expressivity of operations over transitions of timed and counter automata with parameters. Such operations are needed if we want, for example, to increment counters by linear expressions with more than one variable. This cannot be represented by widespread Difference Bound Matrices that allow only expressions of the form $x - y \prec t$ where $t \in \{<, \leq\}$. Another symbolic structure—polyhedra—allows such assignment and guards, nevertheless its implementation is not efficient for acceleration. In this work, we studied properties of data structures for parametric verification and operations over them. Then we proposed a new data structure based on intervals that can be used for representation of counters with parameters. This structure allows general guards and assignments and extends expressivity of the model further in comparison with parameterized DBMs. On the other hand, this structure remains simpler than, for example, polyhedra. It implements acceleration that speeds up analysis of a system.

This work builds upon the work of A. Bouajjani, E. Asarin, A. Collomb–Annichini, M. Sighireanu and others on symbolic techniques for parametric reasoning on parametric timed and counter automata, mainly introduced in [AAB00], [BCALS01], [BCAS01], [AC01], and [AHV93]. We extend definitions presented in these works and apply their results on a new data structure.

In our work we formalize the definition of parameterized bounds constrained by logical formulas and define basic operations over them. Then we extend the definition to intervals. We show operations of inclusion, intersection and finding a minimum on parameterized terms. Unlike numerical intervals we cannot evaluate the expression precisely because it contains symbolical values—parameters. We can only check if there exists any valuation of parameters that satisfies such expression and initial conditions over parameters (like $P > 0$). Practically it means to keep more than one case in a set of explored states and to check satisfiability of the constraints for given conditions.

Data structure for constrained parameterized terms contains not only terms with parameters but also logical formulas that specify constraints over parameters. Both these values change during the evolution of the system. This puts special requirements on data structures such as a compact and unique format (canonical form) and simple manipulation with values and constraints. In our text we introduced basic operations over parameterized data structures and their efficient implementation.

These structures have their limits. We mention that if a constraint is a nonlinear formula mixing integer and real parameters, the satisfiability is not decidable. Nevertheless, in such case we can suppose that parameters are reals and test the satisfiability in real arithmetics. This test is decidable.

There are many other abstract data structures that are used for timed verification—real vector automata [BRW98], numerical decision diagram [WB00], covering sharing trees [DRV01], clock difference diagrams [LWYP99], octagons [Min01], etc. It would be interesting to study these representations in detail and compare their expressivity, complexity of their operations, and possibility of extension with parameters. Another field of research can be extension of parametric timed and counter systems with probability.

In chapters 5 and 6, we discussed fundamentals of modelling, analysis and verification of parameterized timed and counter systems. We introduced a methodology for modelling communicating systems that include three basic components—environment, buffers, and executive units. We proposed models of a simple bounded FIFO queue, a delayed queue and a lossy queue that can be reused for automata-based verification languages where queues are not a part of the language or where a customization is needed.

These principles were applied in the project of parametric verification of PGM protocol that is described in the chapter 6. We describe steps how to create a model and the specification in a language of a verification tool. For verification we used three verification tools HyTech, TReX and Uppaal. We presented here our experience with combination of parametric verification in three tools and our results. At the end of the chapter we point to several sources of complexity of the systems and propose how to deal with them.

For future research in that area will be interesting to exploration further basic principles for parametric verification of real systems as outlined in chapters 5 and 6. It would be worthwhile to identify basic classes of verified systems—like communication protocols, hardware designs, embedded systems—and specify a set of common properties for these classes that are mostly required to be satisfied. And then to introduce a general methodology how to build a model of the system, how to specify and verify the examined properties using model checking and what kind of tools can be used. Nevertheless, this requires longer experience with verification of such systems.

Glossary

abstraction – simplification that hides some aspects of a modelled system

assignment – a transition can modify the value of variables; assignment specifies which variables will be set and how; for example $x := 5$

bound – an arithmetical term (can be with parameters) that restricts possible values of a variable; i.e., $x < 5 + t$ means that variable x of integer domain is restricted to interval $(-\infty, 5 + t)$ where $5 + t$ is a (non-strict) bound

configuration – a state of the transition system; it is identified by a state of an automaton related to this transition system and current values of the variables; for example (s, ν, ψ)

constraint – a condition in the form of logical formulas that has to be satisfied; i.e., $p \leq 5 \wedge p > t$ is a constraint on parameter p

counter – a special non-clock integer variable that increments beyond any limits; counters similarly to clocks can be a parameterized

guard – a constraint (condition) on the variables; it guards a transition t so that the transition cannot occur unless the condition on the variables is satisfied; for example, $x < 5 \ \wedge \ y = t + 2$

hybrid system – a transition system designed to provide access to dynamic variables (timed systems can be viewed as a special kind of hybrid systems); in hybrid automaton, with each control state is associated a relation of evolution with respect to time for each variable; for example, using differential equations $\dot{x} = 2$

invariant – a constraint over a state that has to be satisfied while activity is in the state; for example $x \leq 12$

model checking – an automatic technique for verifying finite state concurrent systems; it is based on an exhaustive search of the state space of a system; the main disadvantage of model checking is the state explosion (cf. to theorem proving)

parameter – a special symbolic variable that is not initialized; uninitialized constant

parametric verification – verification on a model where clocks or counters can be compared with parameters; during analysis and verification parameters are considered to be special symbols; they are not evaluated

specification – formal description of system properties to be verified

symbolic model checking – model checking method that attempts to represent the states and transitions of a configuration system for verification symbolically (not explicitly); using forward/backward exploration procedure a finite symbolic reachability graph is generated

symbolic representation – data structure that represents data symbolically (by formulas, for example) instead of explicitly (by enumeration); it is used to represent infinite sets of configurations

state space explosion – enormous increase of states during system analysis because of the interaction with one another

timed automata – finite automata that describe the system control states and the transitions between states; they include clocks to represent quantitative time and time constraints over transitions and states

temporal logic – logic with special operators that work with time (ordering of events); for example, CTL with operators X,F,G,U

transition system – a graph that describes the behaviour of an automaton; transition system represents the dynamics of the system while automaton shows its static behaviour

valuation – a mapping associating with each clock/counter its current value (real number, integer)

Bibliography

[AAB99] P.A. Abdulla, A. Annichini, and A. Bouajjani. Symbolic verification of lossy channel systems: Application to the bounded retransmission protocol. In *Proceedings of 5th TACAS*, volume 1579 of *LNCS*. Springer Verlag, 1999.

[AAB00] 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, *Proceedings of the 12th CAV*, volume 1855 of *LNCS*, pages 419–434. Springer Verlag, July 2000.

[ABJ98] P.A. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy, FIFO channels. In *Proceedings of the 10th CAV*, volume 1427, pages 305–317, 1998.

[AC01] A. Annichini-Collomb. *Vérification d'automates étendus: algorithmes d'analyse symbolique et mise en oeuvre.* PhD thesis, Joseph Fourier University, December 2001.

[ACH$^+$95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.

[ACHH93] 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 systems*, volume 736 of *LNCS*, pages 209–229. Springer Verlag, 1993.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AHV93] R. Alur, T.A. Henzinger, and M.Y. Vardi. Parametric real-time reasoning. In *ACM Symposium on Theory of Computing*, pages 592–601, 1993.

[Alu99] R. Alur. Timed automata. In *Proceedings of 11th CAV*, volume 1633 of *LNCS*, pages 8–22. Springer Verlag, 1999.

[BBF$^+$01] B. Bérard, M. Bidoit, A. Finkel, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie. *Systems and Software Verification. Model-Checking Techniques and Tools.* Springer Verlag, 2001.

[BCALS01] A. Bouajjani, A. Collomb-Annichini, Y. Lackneck, and M. Sighireanu. Analysing fair parametric extended automata analysis. In Patrick Cousot, editor, *Proceedings of Static Analysis Symposium*, volume 2126 of *LNCS*, pages 335–355. Springer Verlag, July 2001.

[BCAS01] A. Bouajjani, A. Collomb-Annichini, and M. Sighireanu. TReX: A tool for reachability analysis of complex systems. In *Proceedings of CAV*, volume 2102 of *LNCS*, pages 368–372. Springer Verlag, June 2001.

[BFG$^+$00] M. Bozga, J.-C. Fernandez, L. Girvu, S. Graf, J.-P. Krimm, and L. Mounier. IF: A validation environment for times asynchronous systems. In E.A. Emerson and A.P. Sistla, editors, *Proceedings of the 12th CAV*, volume 1855 of *LNCS*, pages 543–547. Springer Verlag, July 2000.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. InVeSt: A tool for the verification of invariants. In *Proceedings of the 10th CAV*, volume 1427, 1998.

[Bou02] A. Bouajjani. *ADVANCE Project Deliverable Report*, chapter Modeling and Verifying the PGM protocol. LIAFA, 2002.

[BRW98] B. Boigelot, S. Rassart, and P. Wolper. On the expressiveness of real and integer arithmetic automata. In *Proceedings of ICALP'98*, 1998.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35-8:677–691, 1986.

[BS03] Marc Boyer and Mihaela Sighireanu. Synthesis and verification of constraints in the pgm protocol. In Stefania Gnesi, editor, *Proceedings of International FME Symposium*, LNCS, pages 264–281. Springer Verlag, 2003.

[BY03] Johan Bengtsson and Wang Yi. On clock difference constraints and termination in reachability analysis of timed automata. In J. S. Dong and J. Woodcock, editors, *Proc. of ICFEM'03*, number 2885 in Lecture Notes in Computer Science. Springer Verlag, 2003.

[BY04] J. Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In W.Reisig and G.Rozenberg, editors, *Lecture Notes on Concurrency and Petri Nets*, volume 3098 of *LNCS*, 2004.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation fixpoints. In *Proceedings of the 4th Annual Symposium on Principles of Programming Languages.* ACM Press, 1977.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 1999.

[Dil89] D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proceedings of the 1st CAV*, volume 407 of *LNCS*, pages 197–212. Springer Verlag, 1989.

[DRV01] G. Delzanno, J.-F. Raskin, and L. VanBegin. Csts: Compact data structures for parameterized verification. *Software Tools for Technology Transfer*, 2001.

[DT98] C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS'98*, volume 1384, Lisbon, Portugal, 1998.

[FGK$^+$96] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP (cæsar/aldebaran development package): A protocol validation and verification toolbox. In R. Alur and T.A. Henzinger, editors, *Proceedings of the 8th CAV*, volume 1102, pages 437–440, August 1996.

[FR73] M.J. Fischer and M.O. Rabin. Super-exponential complexity of presburger arithmetic. In *Proceedings of a Symposium in Applied Mathematics of the American Mathematical Society and the Society for Industrial and Applied Mathematics*, Providence, RI, April 1973.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proceeding of the 2nd Workshop on Computer-Aided Verification*, volume 531 of *LNCS*, pages 176–185. Springer Verlag, 1990.

[Hea99] A.C. Hearn. *REDUCE — User's and Contributed Packages Manual.* Codemist Ltd., February 1999. version 3.7.

[HHWT95] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HYTECH. In *Proceedings of TACAS*, volume 1019 of *LNCS*, pages 41–71. Springer Verlag, 1995.

[HHWT97] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1(1):110–122, 1997.

[HRSV01] T. Hune, J. Romijn, M. Stoelinga, and F. Vaandrager. Linear parametric model checking of timed automata. In *Proceedings of TACAS'01*, 2001.

[LSW97] K.G. Larsen, B. Steffen, and C. Weise. Continuous modeling of real-time and hybrid systems: From concepts to tools. *International Journal on Software Tools for Technology Transfer*, 1(1-2):64–85, 1997.

[LWYP99] K.G. Larsen, C. Weise, W. Yi, and J. Pearson. Clock difference diagrams. *Nordic Journal of Computing*, 6(3):271–198, 1999.

[Mat04a] P. Matoušek. A New Data Structure Based on Intervals for Parametric Counter Automata. In *Proceedings of MOVEP'04*, pages 16–21. Université Libre de Bruxelles, December 2004.

[Mat04b] P. Matoušek. Tools for Parametric Verification. A Comparison on a Case Study. *Journal of Universal Computer Science*, 10 (2004):1469–1494, 2004.

[McM92] K. McMillan. Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits. In G.von Bochmann and D.K.Probst, editors, *Proceeding of Computer-Aided Verification (CAV) 1992*, volume 663 of *LNCS*, pages 164–177. Springer Verlag, 1992.

[Min01] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, 2001.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume I: Specification. Springer Verlag, 1992.

[MSV05] P. Matoušek, A. Smrčka, and T. Vojnar. High-level Modelling, Analysis and Verification on FPGA-based Hardware Design. Accepted to CHARME'05, 2005.

[Ome96] Omega Team. *The Omega Library*, November 1996. version 1.1.0.

[Pel94] D. Peled. Combining partial order reduction with on-the-fly model-checking. In D.Dill, editor, *Proceeding of the 1994 Workshop on Computer-Aided Verification*, volume 818 of *LNCS*, pages 377–390. Springer Verlag, 1994.

[PL00] P. Pettersson and K.G. Larsen. UPPAAL2k. *Bulletin of the European Association for Theoretical Computer Science*, 70:40–44, February 2000.

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parametrized verification. In *Proceedings of the 12th CAV*, volume 1855, 2000.

[SFC$^+$01] T. Speakman, D. Farinacci, J. Crowcroft, J. Gemmell, S. Lin, D. Leshchiner, M. Luby, A. Tweedly, N. Bhaskar, R. Edmonstone, T. Montgomery, L. Rizzo, R. Sumanasekera, and L. Vicisano. PGM reliable transport protocol specification. RFC 3208, IETF, Decembre 2001. 111 pages.

[ST02] Karsten Strehl and Lothar Thiele. Interval diagrams for efficient symbolic verification of process networks. *IEEE Transactions on Computer-Aided Design of Integrated Ciruits and Systems*, 2002.

[Tar51] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 2nd edition, 1951.

[Val90] A. Valmari. A stubborn attack on state explosion. In *Proceeding of the 2nd Workshop on Computer-Aided Verification*, volume 531 of *LNCS*, pages 156–165. Springer Verlag, 1990.

[WB00] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *Proceedings of the 6th Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785, 2000.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[Wil93] Doran K. Wilde. A library for doing polyhedral operations. Internal publication 785, IRISA, Rennes, France, December 1993.

[Yov98] S. Yovine. Model checking timed automata. In G. Rozenberg and F. Vaandrager, editors, *Lectures on Embedded Systems*, volume 1494 of *LNCS*, 1998.