Algorithms and data structures for parametric analysis of real time systems

Dissertation
zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultäten
der Georg-August-Universität zu Göttingen

vorgelegt von
Patryk Chamuczyński
aus Brzeg Dolny

Göttingen 2009

D7

Referent:	Prof. Dr. Hogrefe
Korreferent:	Prof. Dr. Castanet
Tag der mändlichen Prüfung:	

Algorithms and data structures for modeling and analysis of real-time systems

Abstract:

This document is intended to contribute to the area of validation and verification of communicating real time systems, with emphasis put on parametric reachability analysis of systems modeled using timed automata.

Reachability analysis is a crucial aspect of validation and verification of software and hardware systems. The reachability analysis for real time systems is area that is studied by many researchers in academic and industrial communities. However, not much work has been done for systems, where temporal constraints are expressed using parameters. This is serious disproportion with real world, where specifications of most of the communication protocols or embedded software and hardware systems are indeed parameterized.

This thesis presents a complete framework for forward and backward parametric reachability analysis. The solution presented here can be used as a base of algorithms for validation and verification of software and hardware real-time systems, modeled as timed automata with parameters. The results of the thesis can be easily applied to model checking or test generation tools and algorithms.

The core idea of the thesis is a concept of Extended Difference Bound Matrix (EDBM). This is a data structure that stores relations between all system's clocks and parameters. In contrast to Parametric DBM, that is the state-of-the-art data structure for parametric analysis, EDBM does not require storing constraints on clocks and constraints on parameters separately. This leads to significant benefits regarding memory consumption and time necessary to perform basic operations for symbolic analysis.

The maturity of the solution was proven by implementation of a proof-of-concept tool and by experiments performed with modern communication protocol. The results show that even complex systems can be efficiently handled by the framework.

Keywords: timed automata, Difference Bound Matrix, embedded systems, real time systems, parameterized verification, model checking, test generation

Acknowledgements

It is not possible to mention here all the people that I want to thank, so please do not be angry if you can not find your name below.

In the first place I would like to thank my supervisors - prof. Richard Castanet from University of Bordeaux and prof. Dieter Hogrefe from University of Göttingen for giving me an ultimate freedom in the direction of my research while always being ready with help and advice.

Big part of the research reported in this thesis was done at the University of Bordeaux and the LaBRI institute. There is a lot of people, that need to be mentioned for their help and support during those 18 months, that I spent in France. In the first place I want to thank Jean Louis Lassartesses for his tremendous help with all the required administration work. With your guidance even the legendary French bureaucracy was not scary to me. I want to thank Antoine Rollet for taking care of me during my first days at LaBRI, although they were his first days there as well. Special thanks must be said to Gosia Napierała (I guess that your name is Vincent now) for teaching me French. You really taught me more than I expected and deserved. And last but not least – Ismaïl Berrada. It was your initial idea to investigate the subject, that turned out to be worth of writing 170 pages long thesis. To all of you: merci beaucoup.

The list of people from Institute for Informatics in Göttingen that I want to thank is for sure not shorter from the French one. Carmen Scherbaum de Huamán for making German bureaucracy (definitely not less legendary then French) easier. Also for keeping me motivated to learn and practice my German. Udo Burghard for doing magical things with my computer when it refused to obey me. Some words must be said about Nikunj Modi for his permanent distracting me from work and entertaining me with Indian stories. And mostly for Omar Alfandi for his permanent ability to listen, help and support in all aspects of my stay in Germany. To all of you: danke sehr.

It must be said here, that a single page of this work could not be written without the support of my wife. I thank you the most. Dziękuję.

Mojej żonie

To my wife

Pour ma femme

Für meine Frau

Contents

1 Introduction

1.1 Formal methods

The recent technological revolution resulted in rapid expansion of Internet, communication systems and embedded applications in different fields of human life. Not more than 20 years ago, using computer was a privilege of small amount of specialist. Nowadays, our interaction with some kind of computational-based device is unavoidable. Consumer electronics, vehicles, telecommunication systems, medical equipment – these are only few examples of domains where the impact of the revolution in electronics cannot be overestimated.

A malfunctioning system may have different consequences. It may be as meaningless as irritation, when a pocket audio player does not want to handle a playlist correctly, or it may be as catastrophic as an explosion in a nuclear plant. For many of such systems, it is crucial that they provide a correct and efficient service. In order to gain confidence that such devices satisfy standards of service, it has been recognized that formal analysis has to be carried out as part of their development.

Formal Methods are mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. The phrase "mathematically rigorous" means that the specifications used in formal methods are well-formed statements in a mathematical logic and that the formal verifications are rigorous deductions in that logic (i.e. each step follows from a rule of inference and hence can be checked by a mechanical process). The value of formal methods is that they provide means to symbolically examine digital design (either hardware or software). There is a growing set of success stories in applying formal methods to real applications in automotive industry [73, 60], space industry [35, 79] or medicine [15, 40]. This work aims to contributing in the following of the formal methods:

- *Formal verification*, as the act of proving or disproving the correctness of intended algorithms underlying in a system with respect to a certain formal specification or property. Verification is done by providing a formal proof on an abstract mathematical model of the system, while the correspondence between the mathematical model and the nature of the system being otherwise known by construction. There can be distinguished two approaches to formal verification:
  - *Logical inference* – The property is verified by mathematical reasoning about the system, usually using automated proof checking software, like the family of HOL proof checkers [84, 85], or their successor family Isabelle [66]. The overview and comparison of proof checking tools is done in [83].
  - *Model checking* – The properties are verified by exhaustive analysis of the reachability space of the system states. Model checking is described in more details in the Section 4.1.
- *Testing* – In general, testing may be considered as a process of comparing behavior of implemented system to its specification. Testing may be considered in many aspects. For example *conformance testing* is based on checking whether a developed system conforms to its specification. It is done by observing system's implementation and comparing its behavior to a reference specification. Conformance testing may be done in active or passive way. Passive testing is done by deducing conformance of a system to its specification basing on monitoring system's behavior without any interaction between *tester* and the *system under test* (SUT). In case of active testing, a tester stimulates a system under test according to a *test case* that was derived using system formal specification. System's responses are then compared to the specification to check their conformance. Testing methods can be traditionally divided according to the accessability of the tester to internal structure of the SUT:
  - *Black box testing*: a tester have access only to an external interface of a tested system, with no direct access to its internal structure. This imposes that knowledge about currently occupied system state (values of variables etc.) or performed interaction between system's components must be deduced basing on observation of the interface.
  - *White box testing*: a tester may fully access and observe internal structure of a tested system (e.g. may stimulate interfaces between system's components, read

values of variables etc.)

– *Grey box testing*: this is an intermediate case between black and white box testing. In this case the tester has limited approach to the internal structure of SUT, e.g. may read the values of variables but is not allowed to see the interface between components, or may see them but is not allowed to stimulate them.

General constraints and methodology for conformance testing has been standardized by International Organization for Standardization (ISO) in [52].

1.2 Real time systems

The formal reasoning about systems becomes more complicated, if a description of the system's behavior, apart from sequences of events, contain also constraints on timing of those events. Such systems are referred to as *real time systems* (RTS). Examples of such systems may include:

- Automotive safety critical systems, e.g. ABS. Correct working of such a system depends on following strict constraints on how often and for how long the brakes of the car should be released in order to stop the car while preventing it from becoming uncontrollable. Violating those constraints may have catastrophic results: if the brakes are locked for too long, the car is locked and may not succeed in omitting an obstacle. If the brakes are released for too long, the car may not succeed in stopping and then hits an obstacle.
- Communication protocols. Functioning of many protocols (e.g. [74, 44]) depends on timing of messages from communicating nodes. Some other protocols (e.g. [53, 26, 34]) have time-triggered character, which means that performing node's actions depends on a state of its clock. Formal analysis of communication protocol may be an alternative or complement to analysis done by means of simulation (see [31, 30, 4]).
- An assembly line. This is a manufacturing process in which parts (usually interchangeable parts) are added to a product in a sequential manner to create a finished product much faster than with handcrafting-type methods. Scheduling this process in optimal way according to timing of performing each task may let performing many task in parallel and boosting the whole manufacturing process.

1.2.1 Models of real time systems

Modeling real time systems derives from methods used for untimed systems. Many models used with untimed systems have been extended for handling real time constraints. Examples of such models may be Timed Petri Nets [16], Timed Transition Systems [48] or Finite State Machines extended with Action Durations and Time-Outs [64]. There are also extensions of specification languages widely used for industrial purposes, like SDL-RT [3] or Real Time Profile for UML [46]. Behavior of real time systems may be also described using algebra notations [80, 86, 72].

This work concentrates on using Timed Automata [5] as a model of real time system. In the original theory of timed automata, a timed automaton is a finite state Büchi automaton extended with a set of real-valued variables modeling clocks. Constraints on the clock variables are used to restrict the behavior of an automaton, and Büchi accepting conditions are used to enforce progress properties. Due to its simplicity and power of expression, Timed Automata has been adopted in several verification tools, like UPPAAL [58], SPIN [50] or KRONOS [38, 38]. Those tools have been successfully used in industrial case studies, e.g. [11, 21].

The success of timed automata has been driving force for extending the theory to match new purposes. The examples may be probabilistic and stochastic automata [13, 36], hybrid automata [47, 49] or hierarchical timed automata [37].

1.2.2 Parametric real-time reasoning

Traditional approaches to the algorithmic verification of real-time systems are limited to checking program correctness with respect to concrete timing properties (i.e. time constraints are defined with concrete values – reals or integers). More realistic and more ambitious approach is when those constraints may be parameterized. In this case the constraints are defined with parameters. Value of a parameter is chosen from a predefined range at initial state and is fixed for entire execution. Such an approach reflects realistic scenario, where a system may behave in different ways, according to its configuration (e.g. the acknowledgement time-out in a radio transceiver may be configured differently according to specific radio conditions). The design of a robust system requires the verification of the desired behavior of the system without concrete values for parameters. Indeed, when

studying the literature on real-time protocols, one sees that the desired timing properties for protocols are almost invariably parametric [82, 12], because concrete timing constraints make sense only in the context of a given concrete environment. Using parametric reasoning, it is possible to either verify that a system satisfies a given property for all possible values of parameters, or to find constraints on the parameters that define the set of all possible values for which the the property is satisfied. The foundations of the theory of parametric reasoning about real time systems has been done in [6]. This work has been continued in [8] or [51].

1.3 Motivation of the thesis

The research work on the parametric verification and test case derivation for real time systems was motivated by a study on the FlexRay protocol [29]. FlexRay is a protocol designed for in-car communication purposes. It is based on distributed synchronization mechanisms and time triggered medium access control scheme, therefore fulfilling strict time constraints is critical for correct functioning of the protocol. FlexRay is designed to be scalable and flexible in configuration, what is manifested for example in freedom regarding definition of network topology, or in allowing user (that is usually a car manufacturer) to arbitrary allocate available bandwidth to network's nodes. This approach has consequences in plenitude of parameters that were used in the protocol specification. Most of the time constraints (e.g. length of communication time slot, duration of network idle time following transmition etc.) are defined using parameters. Verification of such a protocol or even efficient derivation of test suite able to cover most of the specification requires data structures that efficiently and compactly considers multiplicity of options that are introduced by plentifulness of parameters used in specification. Currently used approaches (discussed in Chapter 5) suffer explosion in terms of memory consumption and time cost in case of systems with many parameters that can take values from wide ranges.

The goal of this thesis is to design a data structure for symbolic analysis of parameterized system. The thesis introduce a compact structure that is extension of Difference Bound Matrix [20] and allows to constraint clocks and parameters within the same structure. Although initially bigger than other data structures, it remains in the same size during entire analysis, when other structures may grow into unmanageable sizes. This feature

simplifies basic operations for symbolic analysis what boosts the entire process. Other advantage of the new structure is that it extends expressiveness of guards over transitions. Standard structures, like PDBM (see chapter 5) allow constraining single clock or difference of two clocks, while the solution proposed in this thesis allow constraining difference of two sums of clocks.

The solution described in the thesis has been implemented in a proof-of-concept tool and successfully used for generating test cases for the FlexRay MAC process.

1.4 Structure of the thesis

The thesis is structured in 8 chapters.

The Chapter 2 introduces mathematical foundations of the concepts that were defined in later chapters. In the first section it presents the notation that will be used in predicate logic formulae or algorithms. Next sections recalls basic ideas of the set theory, graphs and dense spaces. Those concepts will have crucial meaning in the definition of the main thesis subjects.

The Chapter 3 covers basic ideas of modeling real time systems as timed automata. It introduces the basic model of a timed automaton and later shows possible extensions that improve expressibility of the model. Later part of this chapter shows modeling approaches for systems composed of concurrently working elements.

Symbolic analysis of systems defined with timed automata is described in Chapter 4. The basic concepts of model checking are introduced. Then the chapter describes in detailed way concepts of forward and backward symbolic path analysis. The last section of the chapter introduces the Difference Bound Matrix – a data structure widely used in many model checking tools for symbolic representation and manipulation of system state.

The Chapter 5 induces the concept of parametric verification. Starting from the parametric extension of timed automata it goes through the analysis methods for such sort of models. Later sections in this chapter present the state-of-the-art of data structures used in parametric reasoning.

The core of the thesis and the main innovation is presented in the Chapter 6. Here an Extended Difference Bound Matrix, the new data structure for parametric verification, is

presented. The chapter describes algorithms for manipulating this structure and compares them to the currently used techniques. Last section of the chapter shows methods for forward and backward symbolic path analysis that is crucial part of the model checking process.

An implementation of a proof-of-concept of the newly designed structure is reported in the Chapter 7. The chapter shows how the EDBM structure can be efficiently implemented together with basic manipulating operations. The structure's implementation was used in a toy-tool for analysis and simulation of real time systems – SMART. Later sections of the chapter describe the tool's architecture and document its input format. The chapter concludes with report on experiments that were done with specification of the FlexRay MAC process.

The last chapter contains conclusion of the thesis and perspectives for future work in this area.

1.5 Acknowledgements

The work in this thesis was done with a support of the TAROT network and EU IST-DAIDALOS II Framework Programme project.

TAROT (Training And Research On Testing) is a project within Marie Curie Research Training Network (MCRTN). It focuses on the protocols, services and systems testing, that is an essential but empirical and neglected domain of validation and Quality of Service (QoS). Then the TAROT network aims to strengthen and develop the collaboration among major European testing communities.

DAIDALOS II (Designing Advanced network Interfaces for the Delivery and Administration of Location independent, Optimized personal Services) is an EU Framework Programme 6 Integrated Project. During writing of the thesis the project was in its second phase. The Daidalos vision is to seamlessly integrate heterogeneous network technologies that allow network operators and service providers to offer new and profitable services, giving users access to a wide range of personalized voice, data, and multimedia services. 46 partners from industry and academia ambitiously work to achieve this vision.

2 Formalities

The chapter covers basics of mathematical concepts that are used in the thesis. After introducing the notation that will be used throughout the thesis in the Section 2.1, it goes through the fundamentals of the set theory (Section 2.2), graphs (Section 2.3) and dense spaces (Section 2.4). Familiarity with those domains is crucial for understanding the theory that is covered by further chapters.

Contents

2.1 Notation

2.1.1 Numbers

Throughout the document following notation is used for numerical domains:

- $\mathbb{N}$ denotes set of naturals with 0,
- $\mathbb{N}^+$ denotes set of positive naturals,
- $\mathbb{Z}$ denotes set of integers,
- $\mathbb{R}$ denotes set of reals,
- $\mathbb{R}^{\geq 0}$ denotes set of non-negative reals,

2.1.2 Predicate logic

As usually, the symbols $\forall$, $\exists$ and $\nexists$ will denote universal quantification ("for all"), existential quantification ("there exists") and negation of existential quantification ("there does not exist"). The symbol "|" will mean "such that", while ":" will mean "following is true:".

For example the predicate

$$\forall n \in \mathbb{N} \mid n > 2 \ : \ \nexists x, y, z \in \mathbb{Z} \mid x^n + y^n = z^n$$

should be read in the following way: "**for all** natural n **such that** $n > 2$ **following is true**: **there do not exist** integers x, y and z **such that** $x^n + y^n = z^n$.

The symbols $\wedge$ and $\vee$ will denote logical "and" and logical "or" respectively.

2.1.3 Algorithm notation

The algorithms are written using pseudocode. Sometimes the notation is derived from existing programming languages, but is kept rather intuitive. For example statement $i{+}{+}$ denotes increment of the variable i.

The statements *break* and *continue* used in loops are derived form corresponding statements in C language. The command *break* means: "stop executing the loop and go to the first line after the loop". The command *continue* means: "stop executing only this iteration of the loop; proceed with next iteration".

The statement **return** A exits the algorithm and returns the value A.

2.2 Sets, multisets and sequences

2.2.1 Sets

A *set* is an unordered collection of distinct objects that are called *elements*. Sets are noted by surrounding its elements with curly brackets: $\{\cdots\}$. Let $A = \{a, b, c\}$ and $B = \{b, c, d\}$. The notation of basic operations on sets is following:

- union - $A \cup B = \{a, b, c, d\}$,
- intersection - $A \cap B = \{b, c\}$,
- complement - $A \setminus B = \{a\}$,
- cardinality - $|A| = 3$,
- cartesian product - $A \times B = \{(a, b), (a, c), (a, d), (b, b), (b, c), (b, d), (c, b), (c, c), (c, d)\}$.

In the reminder of this document, the set-builder notation will be used whenever it is more convenient than traditional notation. The set-builder notation has following form:

$$A = \{x \mid \Phi(x)\}$$

which means: "A is a set that contains all elements x such that x satisfies predicates defined by $\Phi(x)$".

Equivalence

Two sets are equivalent if they contain exactly the same elements. For example sets $A = \{a, b, c\}$ and $B = \{a, c, b\}$ are equivalent, however sets $\{a, b, c\}$ and $\{a, b\}$ are not.

Subsets

A subset of set A is such a set B that contains only such elements that belong to the set A. The following notation is used to denote subsets:

- $B \subset A$ means that B is a proper subset of A, i.e. $B \neq A$,
- $B \subseteq A$ means that either B is a proper subset of A, or $B = A$.

Powersets

A powerset of a set A, noted by 2^A, is a set of all subsets of A. Formally:

$$2^A = \{B \mid B \subseteq A\}$$

The number elements of a powerset of the set A is equal to $2^{|A|}$.

2.2.2 Multisets

A multiset is a generalization of a set. An element of a multiset can have more than one membership in a multiset. Formally a multiset is defined as a pair $B = (A, m_B)$, where A is the *underlying set of elements* of the multiset B and $m_B : A \mapsto \mathbb{N}^+$ is a multiplicity function that for each element $a \in A$ assigns its multiplicity in B (number of occurrences of a in B).

In the remainder of this document multisets are noted in the following way: $A = \{m_1 a_1, \cdots, m_n a_n\}$ such that $m_i \in \mathbb{N}^+$ is the multiplicity of element a_i in the multiset A. For example multiset $\{a, b, b, c, c\}$ will be noted by $\{a, 2b, 2c\}$.

Let $A = \{a, 2b\}$ and $B = \{b, c, d\}$ be two multisets. Following notation is used for operations on multisets:

- union - $A \cup B = \{a, 2b, c, d\}$,
- sum - $A \uplus B = \{a, 3b, c, d\}$,
- intersection - $A \cap B = \{b\}$,
- complement - $A \setminus B = \{a, b\}$,

- cardinality - $|A| = 3$,
- cartesian product - $A \times B = \{(a, b), (a, c), (a, d), (b, b), (b, c), (b, d), (b, b), (b, c), (b, d)\}$.

Two multisets A and B are equivalent if, and only if, they contain exactly the same elements occurring in both sets exactly the same number of times.

A multiset A is a proper subset of a multiset B, written $A \subset B$, if and only if multiplicity of all elements in A is lower than multiplicity of the same elements in B. Notation $A \subseteq B$ means that the multiset A is either equal to multiset B or is its proper subset.

2.2.3 Sequence

A sequence is an ordered collection of elements. Sequences are noted using square brackets: $a = [a_1, a_2, a_3]$. Unless explicitly stated otherwise, a_i will denote the ith element of the sequence a. A *length* of a sequence is the number of its elements. $[1..n]$ will denote an increasing sequence of subsequent naturals from 1 to n.

For sequences $a = [a_1, \cdots, a_n]$ and $b = [b_1, \cdots, b_n]$ notation $a.b$ denotes concatenation of the two sequences: $a.b = [a_1, \cdots, a_n, b_1, \cdots, b_n]$.

2.3 Graphs

2.3.1 Fundamental definitions

Definition 1. (Oriented graph) *An oriented graph G is a pair (N, E), where N is a finite set of elements $(n_1, n_2, \cdots, n_k)$ called* nodes *and E is a finite set of elements of the cartesian product $N \times N$ called* edges. *A element $(n_i, n_j) \in E$, noted $n_i \rightarrow n_j$, represents an edge with source in n_i and destination in n_j.*

For an oriented graph G following operations are defined:

- $src : E \mapsto N$ defined by: $src(n_i \rightarrow n_j) = n_i$,
- $dest : E \mapsto N$ defined by: $dest(n_i \rightarrow n_j) = n_j$,
- $out : N \mapsto 2^E$ defined by: $out(n_i) = \{e \in E | src(e) = n_i\}$,
- $in : N \mapsto 2^E$ defined by: $in(n_i) = \{e \in E | dest(e) = n_i\}$,

– $\bar{\bullet}$: $E \mapsto E$ defined by: $\overline{(n_i, n_j)} = (n_j, n_i)$.

Intuitively, operation $src(e)$ (resp. $dest(e)$) returns the source (resp. destination) node of the edge e. Operation $out(n_i)$ (resp. $in(n_i)$) returns all edges of G that have source (resp. destination) in n_i. The operation $\overline{e}$ returns an *inverse* edge to e.

Definition 2. *The graph G is* labeled *by alphabet L if there exists a labeling function $\lambda_G : E \mapsto L$. In this case G is noted $G = (N, \lambda_G, E)$.*

From now the notation $n_i \xrightarrow{l \in L} n_j \in E$ will denote an edge with source in n_i, destination in n_j and labeled with l. If L is an ordered and additive set, l is called *weight* of the edge $n_i \xrightarrow{l} n_j$. The graph $G = (N, \lambda_G, E)$ is then called *weighted*. For weighted graphs, the labeling function λ_G will be noted by ω_G.

Definition 3. *The graph G is* complete*, if for all pairs $n_i, n_j \in N, n_i \neq n_j$ there exists an edge $(n_i, n_j) \in E$.*

2.3.2 Path

A path p of the graph $G = (N, \omega_G, E)$ (finite or infinite) is a sequence $[e_1, e_2, \cdots, e_n(, \cdots)]$ where $e_i \in E$ is an edge of G, such that $\forall e_i \in p : dest(e_i) = src(e_{i+1})$. From now $paths(G)$ will denote set of all paths of the graph G.

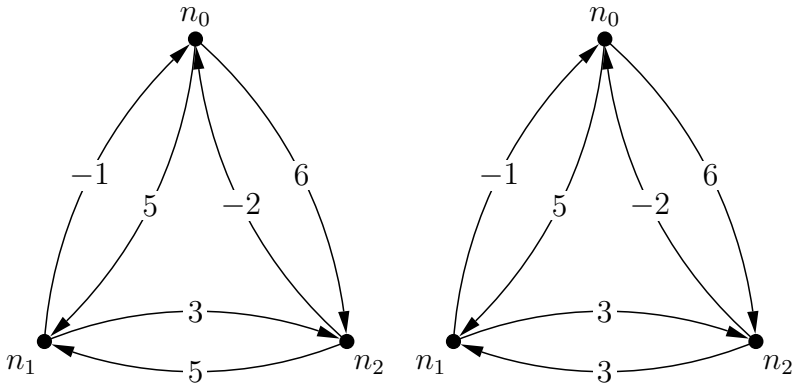
For a finite path $p = [e_1, \cdots, e_n]$, $src(p) = src(e_1)$ and $dest(p) = dest(e_n)$. A path p *traverses* node n if there exist an edge $e \in p$ such that $dest(e) = n$. Therefore a path p with source in the node n not necessarily traverses n.

Let $e \in E$. Then $path_G(e)$ will denote set of paths of the graph G such that $src(p) = src(e)$ and $dest(p) = dest(e)$. A *cycle* of node n is a path with source and destination in n. An *elementary cycle* is a cycle that does not traverse the same node more than once.

If G is weighted, a weight of a path is the sum of the weights of all edges in this path: $\omega_G(p) = \sum_{i \in [1,n]} \omega_G(p_i)$.

2.3.3 Minimal and positive graphs

Definition 4. (Positive graph) *Let $G = (N, \omega, E)$ be a weighted graph with real weights. G is said to be* positive *if and only if weights of all its cycles are not lower than zero.*



(a) G (b) G'

Figure 1: Complete and weighted graphs

Property 1. *A graph is positive if and only if weights of all its elementary cycles are non-negative.*

Proof. It is enough to say that if c is a cycle there exists a finite set $(ec_i)_{i\in[1,n]}$ of elementary cycles such that:

$$\omega(c) = \sum_{i=1}^{n} \omega(ec_i)$$

By consequence, if $\forall i \in [1,n] \ : \ \omega(ec_i) \geq 0$ then $\omega(c) \geq 0$. □

Definition 5. (Minimal graph) *Let $G_c = (N, \omega, E)$ be a complete weighted graph. G is said to be minimal if and only if $\forall e \in E, p \in path_G(e) \ : \ \omega(e) \leq \omega(p)$*

In other words, the graph G is minimal if and only if weight of each edge e of G is not higher than weight of any path connecting the same nodes that e.

Exemple 2.1. *Figure 1 shows two complete and weighted graphs with $N = \{n_0, n_1, n_2\}$. It can be noticed that:*

- *Weights of all elementary cycles of G and G' are non-negative. This means that both G and G' are positive.*
- *$\omega_G(n_2 \rightarrow n_1) > \omega_G(n_2 \rightarrow n_0 \rightarrow n_1)$ which means that the graph G is not minimal.*
- *The graph G' is minimal.*

Property 2. *Only positive graphs can be minimal.*

Proof. Assume that graph $G = (N, \omega, E)$ is not positive. It means that there exists a cycle c such that $\omega(c) < 0$. By consequence it is possible to find a path p between any two nodes of G such that $\omega(p) < k$ for any $k \in \mathbb{R}$ just by traversing the cycle c enough many times. Therefore it is not possible to find a weight for any edge that could satisfy the definition of minimal graph. □

2.3.4 Graph transformations

The function $minimal(G)$ transforms a complete weighted graph $G = (N, \omega_G, E)$ into minimal graph $G' = (N, \omega_{G'}, E)$ such that:

$$\forall e \in E \ : \ \omega_{G'}(e) = min(\{\omega_G(p) | p \in path_G(e)\})$$

In other words, weights of edges with source and destination respectively in n_i and n_j in the graph G' correspond to minimal weight of all paths in G from n_i to n_j. Note that the operation $minimal()$ is unambiguous which means that for given graph G there is only one graph G' that can be result of operation $minimal(G)$.

Exemple 2.2. *Consider again graphs from the Figure 1. The graph G' is the result of operation $minimal(G)$. As effect, G' differs from G in weight of the edges $n_2 \to n_1$ and $n_1 \to n_3$. Note that $\omega_{G'}(n_2 \to n_1) = \omega_G(n_2 \to n_3 \to n_1) = min(\omega_G(p) | p \in path(n_2 \to n_1))$ and $\omega_{G'}(n_1 \to n_3) = \omega_G(n_1 \to n_2 \to n_3) = min(\omega_G(p) | p \in path(n_1 \to n_3))$.*

2.3.5 Minimization algorithm

The algorithm for transforming weighted graph into minimal graph (Floyd-Warschall shortest path algorithm) was given by [43]. It is presented by Algorithm 2.1.

The resulting graph G' has a property that weight of each edge $e \in E$ is equal to minimal weight of any path $p \in path(e)$ such that p traverses each node at most once. Therefore if the graph G is positive then the graph G' is minimal.

Proof. Let $s(k, i, j)$ denote the shortest path (path with minimal weight) between nodes n_i and n_j, from all paths that traverse nodes from the set $\{n_1 \cdots n_k\}$ (not necessary all of them). Let Φ_k denotes following invariant: after kth iteration of the outer loop of the

Algorithm 2.1 Floyd-Warschall shortest path algorithm

Input: complete, oriented and weighted graph $G = (N, \omega, E)$

Output: $G' = minimal(G)$

```
G' = G
for all n_k ∈ N do
  for all n_i ∈ N do
    for all n_j ∈ N do
      ω_G'(n_i → n_j) = min(ω_G'(n_i → n_j), ω_G'(n_i → n_k) + ω_G'(n_k → n_j))
    end for
  end for
end for
return G'
```

algorithm, the weight of all edges $n_i \rightarrow n_j$ is equal to $\omega\big(s(k,i,j)\big)$. It is obvious that Φ_1 holds, because after the first iteration, all weights $\omega(n_i \rightarrow n_j)$ were either not changed, or changed to $\omega(n_i \rightarrow n_1) + \omega(n_1 \rightarrow n_j)$. Thus, to prove correctness of Floyd-Warshall algorithm, it is enough to prove that if Φ_{k-1} holds, Φ_k holds as well.

After $k-1$ iterations $\omega(n_i \rightarrow n_j) = \omega\big(s(k-1,i,j)\big)$, $\omega(n_i \rightarrow n_k) = \omega\big(s(k-1,i,k)\big)$ and $\omega(n_k \rightarrow n_j) = \omega\big(s(k-1,k,j)\big)$. If $s(k,i,j)$ traverses the node n_k, its weight equals $\omega\big(s(k-1,i,k)\big) + \omega\big(s(k-1,k,j)\big)$ and this weight will be assigned to $n_i \rightarrow n_j$ at kth iteration. Otherwise the weight will not be altered. In any case $\omega(n_i \rightarrow n_j)$ will equal to the shortest path between n_i and n_j that traverse nodes from the set $\{n_1 \cdots n_k\}$.

After the outer loop ends, the invariant Φ_n holds which means that the weight of any edge $n_i \rightarrow n_j$ equals the weight of the shortest path between n_i and n_j of all paths from $path(n_i \rightarrow n_j)$.

□

Property 3. *If the shortest path between nodes n_i and n_j goes through nodes n_k and n_l, then the section of the path between n_k and n_l defines the shortest path between those nodes.*

The consequence of Property 3 is that if the shortest path of $path(n_i \rightarrow n_j)$ contains edge $n_x \rightarrow n_y$ then the shortest path of $path(n_x \rightarrow n_y)$ does not contain the edge $n_i \rightarrow n_j$.

Property 4. *The graph G is positive if there is no elementary cycle ec of length 2 in $G' = minimal(G)$, such that $\omega_{G'}(ec) < 0$.*

Proof. Let $ec = \{e, \bar{e}\}$. It is known, that for any p in $path(e)$, such that p does not contain a cycle, $\omega_{G'}(e) \leq \omega_{G'}(p)$. Also, for any $\bar{p}$ in $path(\bar{e})$, such that $\bar{p}$ does not contain a cycle, $\omega_{G'}(\bar{e}) \leq \omega_{G'}(\bar{p})$. This means that $\omega_{G'}(e) + \omega_{G'}(\bar{e}) \leq \omega_{G'}(p) + \omega_{G'}(\bar{p})$. Because $p.\bar{p}$ is an elementary cycle, it means that if $\omega_{G'}(e) + \omega_{G'}(\bar{e}) \geq 0$ then $\omega_{G'}(p.\bar{p}) \geq 0$ and according to Property 1 the graph is positive. □

2.4 Dense spaces

2.4.1 Valuations

Let $V = \{x_1, x_2, \cdots, x_n\}$ be a finite set of variables ranged over $\mathbb{R}^{\geq 0}$ and let $V_0 = \{x_0, V\}$ be the set V extended with a variable x_0 which is always equal to 0. A *valuation* $\nu(V)$ is function $\nu : V \mapsto \mathbb{R}^{\geq 0}$ that assigns value to each element of V. In the following $\nu(x)$ will denote a valuation of single variable $x \in V$. $\mathcal{V}(V)$ will denote the space of all valuations over V. In the remaining, unless stated otherwise, ν will denote $\nu(V)$.

Let $X \subseteq V$, $d \in \mathbb{R}$ and $\nu \in \mathcal{V}(V)$. Then $\nu[X := 0]$ and $\nu + d$ are also valuations, defined respectively by:

- $\nu[X := 0](x) = \nu(x)$ if $x \notin X$, and $\nu[X := 0](x) = 0$ otherwise.
- $(\nu + d)(x) = \nu(x) + d$ for all variables $x \in V$.

In other words, $\nu[X := 0]$ sets each variable in X to 0 and leaves the rest unchanged; by operation $\nu + d$ a value d is added to each variable.

2.4.2 Polyhedra

An *atomic constraint* is an comparison of a variable or difference of variables to a constant. Atomic constraints over V are an expressions of a form:

$$x \bowtie n \text{ or } x - y \bowtie m \text{ with } (x, y) \in V^2, (n, m) \in \mathbb{R} \text{ and } \bowtie \in \{<, \leq, =, \geq >\}$$

Constraints in the form $x - y \bowtie m$ are called *diagonal constraints*.

A set of valuations that satisfy finite conjunction of atomic constraints is called a *polyhedron*[1]. $\Omega(\mathcal{V})$ will denote set of all polyhedra on $\mathcal{V}(V)$. From now **false** will denote an empty polyhedron, **true** will denote a polyhedron constrained by $\bigwedge_{x \in V} x \geq 0$ and **zero** will denote a polyhedron constrained by $\bigwedge_{x \in V} x = 0$.

By convention Z can be described by following set of constraints:

$$Z = \bigwedge_{x_i, x_j \in V_0, x_i \neq x_j} x_i - x_j \prec l_{i,j}$$

where $l_{i,j} \in \mathbb{R}$ is a constant and $\prec \in \{<, \leq\}$. Indeed, a constraint in the form $x_i \succ n$ can be noted as $x_0 - x_i \prec n$, $x_i = n$ can be noted as $x_i - x_0 \leq n \wedge x_0 - x_i \leq -n$. If a polyhedron does not define a constraint on $x_i - x_j$ it may be defined as $x_i - x_j \leq \infty$. If definition of polyhedron contains more than one constraint on the same variable only the tightest one is considered.

Exemple 2.3. *Let $V = \{x_1, x_2\}$ and $Z = (x_1 \geq 3) \wedge (x_2 < 5) \wedge (x_1 - x_2 \leq 4)$ be a polyhedron. Note that the constraint $x_1 \geq 3$ can be written as $x_0 - x_1 \leq -3$. Therefore Z can be defined as follows:*

$$Z = \begin{cases} x_0 - x_1 \leq & -3 \\ x_0 - x_2 \leq & 0 \\ x_1 - x_0 < & \infty \\ x_1 - x_2 \leq & 4 \\ x_2 - x_0 < & 5 \\ x_2 - x_1 < & \infty \end{cases}$$

We say that a polyhedron $Z \in \Omega(V)$ is bounded if there exists such a $d \in \mathbb{R}$ that $\forall \nu \in Z : \nu + d \notin Z$.

Intersection of polyhedra

Intersecting two polyhedra is intuitive. Formally for polyhedra Z and Z':

[1] Note that polyhedra are always convex

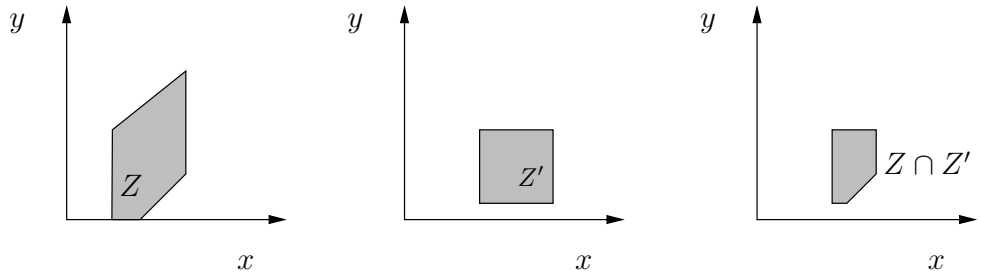


Figure 2: Intersection of two polyhedra

$$Z \cap Z' = \{\nu : \nu \in Z \wedge \nu \in Z'\}$$

This operation is illustrated in the Figure 2.

2.4.3 Numerical bounds

A numerical bound b is a pair in form $(m, \prec)$ where $m \in \mathbb{R} \cup \infty$ and $\prec \in \{<, \leq\}$. Set of all bounds will be noted by $\mathcal{B}$. Formally, the set $\mathcal{B}$ is defined by:

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

Ordering of bounds

Operators ‘ $<$ ‘ and ‘ $\leq$ ‘ are strictly ordered. The order is defined by “$<$” $<$ “$\leq$”. The ordering of bounds is defined as follows:

$$(n_1, \prec_1) \leq (n_2, \prec_2) \Leftrightarrow \begin{cases} n_1 < n_2, \text{ or} \\ (n_1 = n_2) \wedge (\prec_2 = \text{“} \leq'') \end{cases}$$

$$(n_1, \prec_1) < (n_2, \prec_2) \Leftrightarrow \begin{cases} n_1 < n_2, \text{ or} \\ (n_1 = n_2) \wedge (\prec_2 = ' <') \wedge (\prec_2 = ' \leq') \end{cases}$$

Operations on bounds

Let $b_1 = (n_1, \prec_1)$ and $b_2 = (n_2, \prec_2)$. The sum of two bounds is defined in the following way:

$$b_1 + b_2 = (n_1 + n_2, min(\prec_1, \prec_2))$$

The function $min(b_1, b_2)$ returns lower of two bounds and is defined by

$$min(b_1, b_2) = \begin{cases} b_1 & \text{, if } \quad b_1 \leq b_2 \\ b_2 & \text{, otherwise} \end{cases}$$

The multiplication operation of a real and a bound is defined as follows:

$$k \cdot (n, \prec) = (k \cdot n, \prec)$$

2.4.4 Constraint graph

A polyhedron $Z \in \Omega(\mathcal{V})$ can be represented by a *constraint graph*. This is a directed, complete and weighted graph, where nodes are labelled with variables of V_0 and weights of edges define bounds of difference of variables labelling nodes connected by the edge. Formally:

Definition 6. (Constraint graph) *Let Z be a polyhedron defined by:*

$$Z = \bigwedge_{x_i, x_j \in V_0, x_i \neq x_j} x_i - x_j \prec_{i,j} m_{i,j}$$

A constraint graph associated to Z is a directed, complete and weighted graph $G = (V_0, \omega, E)$, such that $\omega : E \mapsto \mathcal{B}$, where each edge $x_j \xrightarrow{(m_{i,j}, \prec_{i,j})} x_i$ represent the constraint $x_i - x_j \prec_{i,j} m_{i,j}$ from the definition of Z.

A constraint graph for the polyhedron from Example 2.3 is presented in the Figure 3.

Later in the document, names of variables will be used to refer to the nodes labelled with those variables in a constraint graph.

A constraint graph represents set of constraints that define polyhedron. Each edge $x_i \xrightarrow{(b, \prec)} x_j$ represent constraint $x_j - x_j \prec b$. Thus, the path $x_i \xrightarrow{(b_i, \prec_i)} x_{i+1} \xrightarrow{(b_{i+1}, \prec_{i+1})} x_{i+2} \cdots x_{k-1} \xrightarrow{(b_{k-1}, \prec_{k-1})} x_k$ in fact represents the following set of constraints:

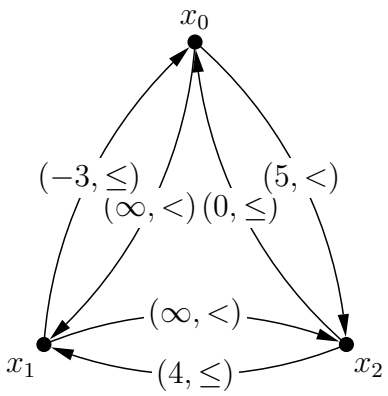


Figure 3: Constraint graph

$$\begin{cases} x_{i+1} - x_i & \prec_i & b_i \\ x_{i+2} - x_{i+1} & \prec_{i+1} & b_{i+2} \\ \cdots & & \\ x_k - x_{k-1} & \prec_{k-1} & b_{k-1} \end{cases}$$

Adding those constraints by sides will give following result:

$$x_k - x_i \prec b_i + b_{i+1} + \cdots + b_{k-1}$$

with $\prec = \leq$ if $\forall i \in [1..k-1] : \prec_i = \text{`} \leq \text{`}$. Otherwise $\prec = \text{`} < \text{`}$. Therefore the actual constraint on $x_j - x_i$ is determined by the weight of the shortest path from the set $path(n_i \to n_j)$.

From now, $expr(p)$ will denote the actual expression for which the path $p \in paths(G)$ determine constraint. For example, for a path $p = x_i \to \cdots \to x_j$, $expr(p) = x_j - x_i$.

2.4.5 Canonical form of a polyhedron

It is possible that two polyhedra defined by different sets of constraints represent the same portion of the space $\mathcal{V}(V)$. It is useful to define a *canonical form* of a polyhedron which defines the "tightest" set of constraint for a given polyhedron. Formally:

Definition 7. (Canonical form) *Let Z be a polyhedron defined by following set of constraints:*

$$\bigwedge_{x_i, x_j \in V_0, x_i \neq x_j} x_i - x_j \prec_{i,j} m_{i,j}$$

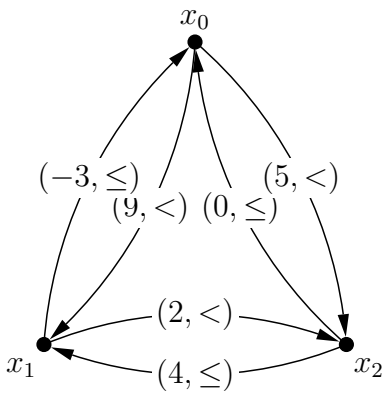


Figure 4: Minimal graph for polyhedron from Example 2.3

Z is in canonical form if and only if:

$$\forall x_i, x_j \ : \ \nexists (m, \prec) < (m_{i,j}, \prec_{i,j}) \mid Z \cap (x_i - x_j \prec m) = Z$$

Intuitively, Z is in its canonical form if the constraints that define it cannot be tightened without changing Z.

In a constraint graph for polyhedron Z, the actual bound of difference $x_i - x_j$ is determined by the shortest path of $path(x_j \rightarrow x_i)$. If there exists $p \in path(x_j \rightarrow x_i)$, such that $\omega(p) < \omega(x_j \rightarrow x_i)$ it means that constraint on $x_i - x_j$ is in fact determined by $\omega(p)$ and not $\omega(x_j \rightarrow x_i)$. This means that $\omega(x_j \rightarrow x_i)$ can be lowered as long as it is not lower than $\omega(p)$, without any consequence on the shape of Z.

On the other hand, if the constraint graph is minimal, weight of any edge e is not higher than weight of any $p \in path(e)$. The actual bound of the difference between variables represented by nodes connected by e is than determined by $\omega(e)$. Thus, lowering weight of e will cause that content of Z will change as well.

Corollary 1. *A polyhedron is in canonical form if and only if its constraint graph is minimal.*

Exemple 2.4. *Let us consider polyhedron defined in Example 2.3 with constraint graph depicted in the Figure 3. Note that the graph is not minimal, since $\omega(x_0 \rightarrow x_1) > \omega(x_0 \rightarrow x_2 \rightarrow x_1)$ and $\omega(x_1 \rightarrow x_2) > \omega(x_1 \rightarrow x_0 \rightarrow x_2)$. The minimal graph for Z is depicted in the Figure 4.*

Therefore, the canonical form of Z is following:

$$cf(Z) = \begin{cases} x_0 - x_1 \leq & -3 \\ x_0 - x_2 \leq & 0 \\ x_1 - x_0 < & 9 \\ x_1 - x_2 \leq & 4 \\ x_2 - x_0 < & 5 \\ x_2 - x_1 < & 2 \end{cases}$$

Theorem 1. (Emptiness test) *The polyhedron Z is not empty ($Z \nsim$ **false**) if and only if its constraint graph is positive.*

A polyhedron does not represent an empty portion of space only if its constraint graph does not contain negative cycles. Therefore, according to Property 4 emptiness of the polyhedron may be tested by checking weights of cycles of lengths 2 of its minimal constraint graph.

Proof. Z is not empty if and only if the constraints that define it are not contradicting. Assume that a constraint graph $G = (N, \omega, E)$ that represents Z is negative. It means that there exist a cycle $c = x_i \rightarrow \cdots \rightarrow x_i$ that has a weight $(b, \prec)$ such that $(b, \prec) \leq (0, \leq)$. The weight of the cycle c determines the bound of $x_i - x_i$, so it determines in fact bound of 0. The negative weight of c leads to following inequality: $0 \prec b$ that is contradicting when $(b, \prec) \leq (0, \leq)$. Therefore any negative cycle determines that Z is empty. □

2.4.6 Minimal constraint system

A set of constraints defining a polyhedron may be redundant in the sense that some of the constraint may be derived from others. For example for a set of constraints $(x - y \leq 2) \wedge (y - z \leq 5) \wedge (x - z \leq 7)$ the latter constraint is obviously redundant, since it may be derived from the first two. It is desirable to know the set of non-redundant constraints that define a polyhedron.

It is known, e.g. from [57], that for each polyhedron there is a minimal constraint system with the same solution set. Computing this minimal form for all polyhedra and storing them in memory using a sparse representation can reduce the memory consumption. This problem has been thoroughly investigated in [57], [68] and [59].

To define an algorithm for finding redundant constraints it is necessary to define *zero cycle*

as a cycle in a constraint graph which weight is zero. If a graph does not have zero cycles, finding the redundant constraints is trivial: an edge of a constraint graph represents a redundant constraint if its weight is equal to weight of any path with source and destination of given edge. Further, if the input graph is in minimal form all redundant edges can be located by considering alternative paths of length two. The Algorithm 2.2 defines a function $reduce^*()$ which removes redundant edges from a zero cycle free constraint graph and has $\mathcal{O}(n^3)$ complexity.

Algorithm 2.2 $reduce^*(G)$

Input: constraint graph G without zero cycles

Output: reduced graph

```
for all i ∈ [1, n] do
  for all j ∈ [1, n] do
    for all k ∈ [1, n] do
      if ω_G(n_i → n_j) ≤ ω_G(n_i → n_k) + ω_G(n_k → n_j) then
        Mark edge n_i → n_j as redundant;
      end if
    end for
  end for
end for
Remove all edges marked as redundant;
```

The problem is more complex, however, in case of graphs with zero cycles. The reason is that the set of redundant edges in a graph with zero-cycles is not unique. This is illustrated by Example 2.5 [22].

Exemple 2.5. *Consider the graph from Figure 5(a). Applying the reasoning for the graphs without zero cycles would remove edge* $x_0 \xrightarrow{(3,\leq)} x_2$ *basing on path* $x_0 \xrightarrow{(-2,\leq)} x_1 \xrightarrow{(5,\leq)} x_2$*, but also it would remove edge* $x_1 \xrightarrow{(5,\leq)} x_2$ *basing on path* $x_1 \xrightarrow{(2,\leq)} x_0 \xrightarrow{(3,\leq)} x_2$*. If both of those edges are removed it will not be possible to construct a path leading to* x_2*. There is a dependence between edges* $x_0 \xrightarrow{(3,\leq)} x_2$ *and* $x_1 \xrightarrow{(5,\leq)} x_2$ *so only one of them can be considered redundant.*

The solution to this problem is to partition the nodes according to zero-cycles and build a

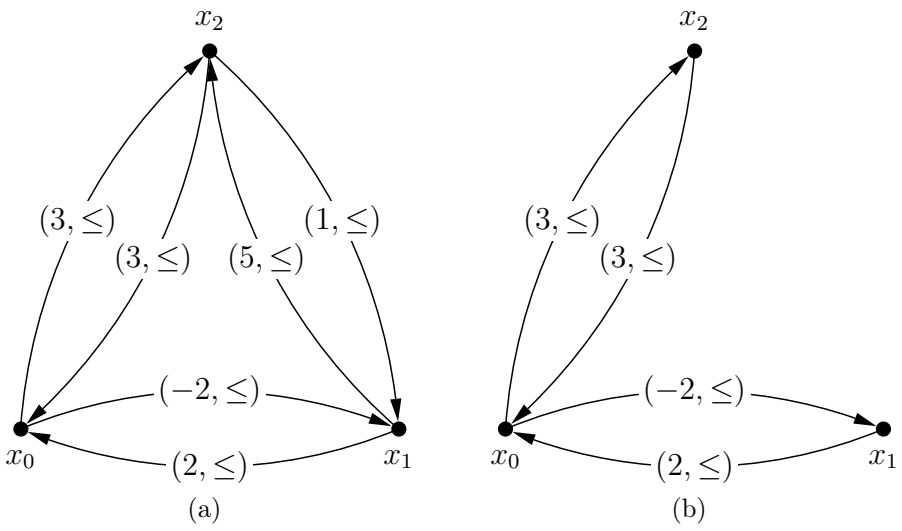


Figure 5: Constraints graph with zero cycle and its reduced version

super-graph where each node is a partition. The graph from Figure 5 has two partitions, one containing x_1 and x_2 and the other containing x_3. To compute the edges in the super-graph one representative for each partition must be picked and the edges between the partitions inherit the weights from edges between the representatives. The super-graph is zero-cycle free and can be reduced using Algorithm 2.2. The relation between the nodes within a partition is uniquely defined by the zero-cycle and all other edges may be removed. The reduced super-graph is connected to the reduced partitions. Figure 5(b) shows the reduced version of graph from the Figure 5(a). Pseudo-code for the *reduce*() function is cited after [22] in Algorithm 2.3.

2.4.7 Operations on polyhedra

Let $\nu^{\uparrow}$ (resp. $\nu^{\downarrow}$) be an operation that for valuation ν returns a polyhedron containing all valuations ν' such that $\nu' = \nu + d$ (resp. $\nu' = \nu - d$) for all $d \in \mathbb{R}^{\geq 0}$.

The operation $[X := 0]\nu$ returns a polyhedron containing valuations ν' such that $\nu'[X := 0] = \nu$.

Intuitively, $\nu^{\uparrow}$ (resp $\nu^{\downarrow}$) contains all valuations that can be obtained by adding (resp. subtracting) the same value to all elements of ν. $[X := 0]\nu$ results in such a polyhedron that assigning 0 to variables in X for all valuations in this polyhedron will result in obtaining

Algorithm 2.3 *reduce*(G)

Input: constraint graph G

Output: reduced graph G'

for all $i \in [1, n]$ **do**
  if n_i is not in a partition **then**
    $Eq_i = \emptyset$;
    for all $j \in [i, n]$ **do**
      if $\omega(n_i \to n_j) + \omega(n_j \to n_i) = (0, \leq)$ **then**
        $Eq_i = Eq_i \cup n_i$;
      end if
    end for
  end if
end for
Let G' be a graph without nodes;
for all Eq_i **do**
  Pick one representative node $n_i \in Eq_i$;
  Add n_i to G';
  Connect n_i to all nodes in G' using weights of G
end for
$reduce^*(G')$
for all Eq_i **do**
  Add one zero cycle containing all nodes in Eq_i to G';
end for

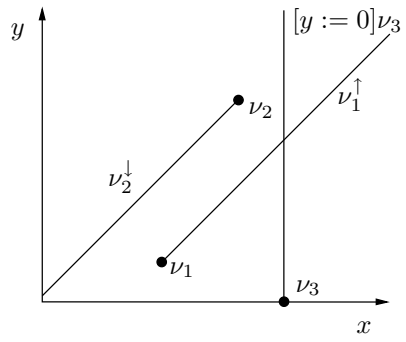


Figure 6: Operations on valuations

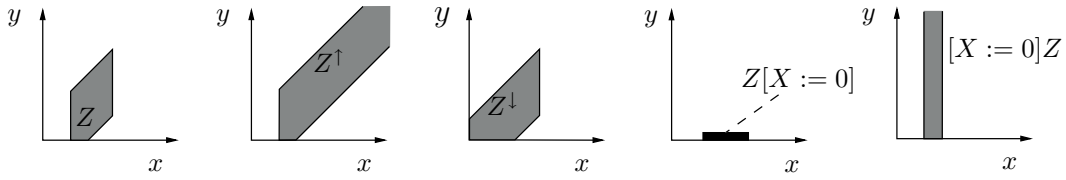


Figure 7: Operations on polyhedra

polyhedron containing only valuation ν. The operations $\nu^{\uparrow}$, $\nu^{\downarrow}$ and $[X := 0]\nu$ are illustrated in the Figure 6

Having defined operations on valuations, corresponding operations may be defined for polyhedra. The operations $Z^{\uparrow}$, $Z^{\downarrow}$, $Z[X := 0]$ and $[X := 0]Z$ are defined in the following way:

$$Z^{\uparrow} = \{\nu^{\uparrow} \mid \nu \in Z\},$$

$$Z^{\downarrow} = \{\nu^{\downarrow} \mid \nu \in Z\},$$

$$Z[X := 0] = \{\nu[X := 0] \mid \nu \in Z\},$$

$$[X := 0]Z = \{[X := 0]\nu \mid \nu \in Z\}.$$

Examples of operations on polyhedra are shown in the Figure 7.

Property 5. *If Z is a polyhedron and $X \subseteq V$, then $Z^{\uparrow}$, $Z^{\downarrow}$, $Z[X := 0]$ and $[X := 0]Z$ are also polyhedra.*

3 Modeling Real Time Systems

This chapter concentrates on modeling real time systems using timed automata. The Section 3.1 presents the background information about clocks and alphabets that need to be understood before introducing timed automata. The model is presented in the Section 3.2, together with some of its extensions. Finally, the Section 3.3 shows two approaches for modeling systems that are composed from more than one communicating real time elements.

Contents

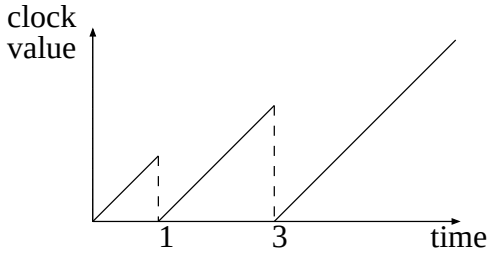


Figure 8: Behavior of a clock

3.1 Background

3.1.1 Clocks

Clocks are variables that measure time. The basic feature of a clock is that its value increases with the same tempo that all other clocks in the system. It means that within fixed period of time value of all system clocks will increase with the same value.

Two allowed operations on clock are comparing its value to a real constant and reset. By reset, a clock's value is set to 0. A behavior of a clock is illustrated in the Figure 8. A value of the clock increases with the same rate that the global time. The clock is reset at two moments: when a value of the global time reached 1 and 3.

3.1.2 Alphabets and timed sequence

Let Σ be a finite alphabet of symbols. Σ^* will denote the set of finite sequences of symbols from Σ and $\epsilon \in \Sigma^*$ is an empty sequence. τ will denote an event not in Σ and Σ_τ is the set $\Sigma \cup \{\tau\}$.

A timed event over Σ is a pair $u = (a, d)$ such that $a \in \Sigma$ and $d \in \mathbb{R}^{\geq 0}$. a is interpreted to denote an event occurrence and d is interpreted as the timestamp of the occurrence of a. $event(u)$ will denote the untimed event a associated to u and $time(u)$ the real d.

A timed sequence $\sigma = [(a_1, d_1)...(a_n, d_n)]$ over Σ is an element of $(\Sigma \times \mathbb{R}^{\geq 0})^*$ such that the sequence of timestamps is monotonically increasing. For example, $\sigma = [(a_1, 3), (a_2, 5)]$ is a timed sequence, however $\sigma' = [(a_1, 3), (a_2, 2)]$ is not. The set of timed sequences over Σ is noted $TS(\Sigma)$.

For $X \subseteq \Sigma$, $\sigma_{|X}$ is the sequence obtained by erasing from σ all timed events u, such that $event(u) \notin X$ (projection on X).

3.2 Timed Automata

3.2.1 Syntax and semantics of TA

Definition 8. *A Timed Automaton (TA) over an alphabet Σ is a 5-tuple $A = (L, l_0, \Sigma, \mathcal{C}, \rightarrow)$, where:*

- *L is a set of locations,*
- *l_0 is an initial location,*
- *Σ is an alphabet of events,*
- *$\mathcal{C}$ is a set of clocks,*
- *$\rightarrow \subseteq L \times \Omega(\mathcal{C}) \times \Sigma_\tau \times 2^C \times L$ is a set of transitions.*

Each transition $t \in \rightarrow$ of a TA has following form: $t = (l, Z, a, r, l')$ noted $l \xrightarrow{Z,a,r} l'$. The l and l' are source and destination locations respectively. Z is a guard of transition that is defined by conjunction of atomic constraints on system clocks. a is an event associated with the transition. r denotes set of clocks reset to 0 when the transition is executed. From now, $src(t)$ and $dest(t)$ will denote source and destination locations of the transition t.

Exemple 3.1. *An exemplary TA is presented in the Figure 9. The automaton has three locations – the initial location l_0, and two other locations: l_1 and l_2. The alphabet of events Σ consists of three events: a, b and c. There are two clocks used: x and y.*

Semantics of Timed Automaton

The semantics of TA $A = (L, l_0, \Sigma, \mathcal{C}, \rightarrow)$ is defined by a transition system [55, 69] $Q^A = (S, s_0, \Gamma, \rightarrow_A)$. A state s of Q^A is defined by a pair $s = (l, \nu)$ where l is current system's location and ν denotes values of all system's clocks. The initial state q_0 is defined by $(l_0, \mathbf{zero})$. The alphabet Γ is defined by $\Gamma = \Sigma_\tau \cup \{\epsilon(d) | d \in \mathbb{R}^{\geq 0}\}$.

There are two possible kinds of transitions between states: delay transition and action transition:

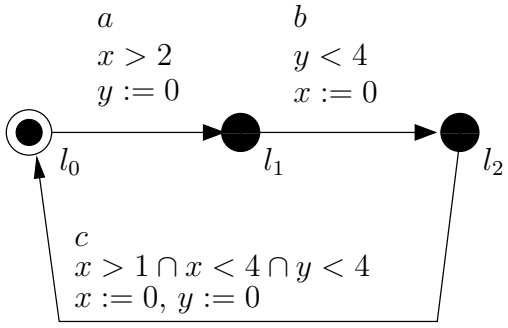


Figure 9: Example of Timed Automaton

- *delay transition* when a state is changed due to passage of time: for a state (l, ν) and $d \in \mathbb{R}^{\geq 0}$ $(l, \nu) \xrightarrow{\epsilon(d)} (l, \nu + d)$,
- *discrete transition* for a state (l, ν) and a transition $t = (l, Z, a, r, l')$, $(l, \nu) \xrightarrow{t} (l', \nu[r := 0])$ if $\nu \in Z$.

3.2.2 Computation

Let $A = (L, l_0, \Sigma, C, \rightarrow)$ be a TA and σ be a timed sequence, such that $|\sigma| = n$. A *computation* r of A over σ, noted $(\bar{s}, \bar{\nu})$ is a finite sequence defined in following form:

$$r : (l_0, \nu_0) \xrightarrow{\sigma_1} (l_1, \nu_1)...(l_{n-1}, \nu_{n-1}) \xrightarrow{\sigma_n} (l_n, \nu_n)$$

with $l_i \in L$ and $\nu \in \mathcal{V}(C)$, satisfying following conditions:

1. *Initiation:* for all $x \in C : \nu_0(x) = 0$
2. *Succession:* for all $i \in [1, n]$ there exists a transition t_i in A, such that $t_i = (l_{i-1}, Z_i, event(\sigma_i), r_i, l_i)$ and:
   - $\nu_{i-1} + (time(\sigma_i) - time(\sigma_{i-1})) \in Z_i$,
   - $\nu_i = \nu_{i-1} + (time(\sigma_i) - time(\sigma_{i-1}))[r_i := 0]$

Intuitively, the initial state is defined by $(l_0, \mathbf{zero})$. When a transition t_{i+1} is executed, valuations of clocks equal ν_i plus the time interval between events $event(\sigma_i)$ and $event(\sigma_{i+1})$. This valuation is checked against the transition guard for t_{i+1}. The valuation of clocks when entering location l_{i+1} must be equal to the valuation at the moment of executing t_{i+1} but with all clocks in r_{i+1} reset to 0.

Exemple 3.2. *Consider TA from the Figure 9 and following timed sequence σ:*

$$\sigma = [(a, 2.4), (b, 3), (c, 3.8)]$$

The corresponding computation for σ is presented below:

$$r : (l_0, [0, 0]) \xrightarrow{(a,2.4)} (l_1, [2.4, 0]) \xrightarrow{(b,3)} (l_2, [1.8, 0.6]) \xrightarrow{(c,3.8)} (l_0, [0, 0])$$

For the same automaton computation over sequence σ' would not be possible:

$$\sigma' = [(a, 2.4), (b, 3.2), (c, 4.0)]$$

A set of timed sequences that allow computation of A is noted by $Runs(A)$ and is defined by:

$$Runs(A) = \{\sigma \in TS \mid A \text{ allows computation over } \sigma\}$$

The projection of all elements of $Runs(A)$ onto alphabet Σ is called *timed traces* and noted by $TTrace(A)$. Formally:

$$TTrace(A) = \{\sigma' \mid \exists \sigma \in Runs(A) \mid \sigma' = \sigma_{|\Sigma}\}$$

Finally, $TTrace(A, n)$ denotes all elements of $TTrace(A)$ of length n.

3.2.3 Invariants

The specification of TA may be extended with *invariants*. In TA with invariants each location is associated with a polyhedron describing clock constraints which must be fulfilled to let the automaton reside in given location. Formally TA with invariants is defined as follows:

Definition 9. (TA with invariants) *A timed automaton with invariants is a 6-tuple $(L, l_0, \Sigma, C, Inv, \rightarrow)$, where:*

- *$(L, l_0, \Sigma, C, \rightarrow)$ is a TA in classical meaning*

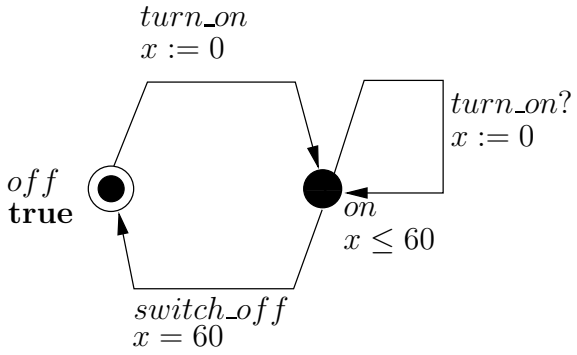


Figure 10: Example of a TA with invariants - light controller

- $Inv : L \mapsto \Omega(\mathcal{C})$ *is the function associating each location with an invariant in form of a polyhedron.*

The semantics of TA with invariants is the same as semantics for classical TA with a little difference regarding the possible types of transition:

- *Delay transition* – state is changed due to passage of time: for a state (l, ν) and $d \in \mathbb{R}^{\geq 0}$ $(l, \nu) \xrightarrow{\epsilon(d)} (l, \nu + d)$, if for all $0 < d' < d$, $\nu + d' \in Inv(l)$
- *Discrete transition* – for a state (l, ν) and an edge $t = (l, Z, a, r, l')$, $(l, \nu) \xrightarrow{t} (l', \nu[r := 0])$, if $\nu \in Z$ and $\nu[r := 0] \in Inv(l')$.

Exemple 3.3. *An example of TA with invariants is presented in the Figure 10. It is a classical example of light controller. The initial location of the automaton is* off. *Note that this location is associated with an invariant defined by* **true** *what means that the automaton may stay there for any time. If the automaton receives signal* turn_on *from the environment it goes to the location* on *and resets the clock x. The location* on *may be occupied only if the value of the clock x is lower or equal to* 60 *time units. If event* turn_on *is received within this period of time the clock x is reset. If it reaches value* 60 *the automaton switches off the light (by* switch_off *event) and goes to the location labelled with* off.

3.2.4 Urgent locations

In the tool UppAal, locations may be labelled as *urgent*. The time is not allow to pass in urgent locations - when the automaton enters such a location it must leave it immediately [19]. Semantically, urgent locations are equivalent to:

- adding an extra clock x, that is reset on every transition with destination in the urgent location, and
- adding an invariant $x \leq 0$ to the location. [18]

3.2.5 Time Input Output Automata

The basic model of TA does not allow distinction between emission and reception of action. It may be sometimes necessary to distinct, whether a transition of TA is executed due to stimulation from system's environment (input), or it was initiated by the system itself (output). To allow analysis of real time systems from this point of view, the extension of TA was proposed – Time Input/Output Automata ($TIOA$) [54]. $TIOA$ is a TA over alphabet $\Sigma_\tau = \Sigma^I \cup \Sigma^O \cup \tau$, where:

- Σ^I is a set of input actions (emitted by the environment)
- Σ^O is a set of output actions (emitted by the automaton)
- τ is an internal, unobservable event of the automaton.

The automaton $A = (L, l_0, \Sigma, \mathcal{C}, \rightarrow)$ is said to be *input complete* if it accepts every input in all states, i.e $\forall l \in L, a \in \Sigma^I \ : \ \exists t \in \rightarrow \ | \ src(t) = l \wedge action(t) = a$, where $action(t)$ denotes the action associated with transition t.

3.2.6 Extended TIOA

Most of the specifications of real systems apart clocks use also *variables*. Modeling such systems is possible with *Extended TIOA* [65]. Formally, an Extended TIOA A is a tuple $A = (L, l_0, \Sigma, \mathcal{C}, V, V_0, \rightarrow)$, where L, l_0, Σ and $\mathcal{C}$ are defined in the same way that for standard TIOA, and additionally:

- V is a set of variables (reals, integers, booleans etc.),
- V_0 is a set of initial values of variables from V,
- transitions in $\rightarrow$ have form: (l, Z, a, Upd, l') such that:
  - l and l' are source and destination locations respectively,
  - $a \in \Sigma$ is an action associated with the transition,

– Z is a guard of the transition in form conjunction of atomic constraints in the form:

$$x_i - x_j \prec f(V) \text{ or } f_1(V) \bowtie f_2(V)$$

– Upd is a set of updates in form:

$$\begin{cases} x := 0 & \text{if } x \in \mathcal{C} \\ x := f(V) & \text{if } x \in V \end{cases}$$

where $f(V)$, $f_1(V)$ and $f_2(V)$ are linear functions over variables of V.

The semantics of Extended TIOA A is defined by a transition system $TS(A) = Q^A = (S, s_0, \Gamma, \rightarrow_A)$. A state of Q^A is defined by a triple (l, ν, ϑ), where l denotes currently occupied location, ν is a valuation of system's clocks and ϑ is valuation of variables of V. The initial state is defined by $(l_0, \mathbf{zero}, V_0)$. The alphabet Γ is defined as in case of standard TA by $\Gamma = \Sigma_\tau \cup \{\epsilon(d) | d \in \mathbb{R}^{\geq 0}\}$.

There are two possible kinds of transitions between states: delay transition and action transition:

- *delay transition* as described in the Section 3.2.1 for standard TA,
- *discrete transition* for a state (l, ν, ϑ) and a transition $t = (l, Z, a, Upd, l')$, $(l, \nu, \vartheta) \xrightarrow{t} \big(l', \nu[r := 0], Upd(\vartheta)\big)$ if $\nu, \vartheta \in Z$.

3.3 Modeling parallel systems

TA or $TIOA$ allow to model behavior of single entity communicating with its environment. They do not allow to model parallel execution and communication of two or more entities. This is handled by a new, higher level modeling structures.

For needs of UPPAAL a network of $TIOA$ has been defined [22]. It allows to model several automata executed in parallel, however all of them on the same topological level - networks do not allow modeling nested structures.

Nested structures are possible using the model of Communicating System (CS) defined in [24]. A CS defines a communication topology for set of automata in the system. Viewed from outside, a CS has the same interface as TIOA, so it may communicate with them and can be nested in CSs of higher levels.

3.3.1 Networks of $TIOA$

A *network of* $TIOA$ is a set $N = \{A_1, ..., A_n\}$ of timed input output automata, called *processes*. Synchronous communication between the processes is done by hand-shake synchronization using input and output events - an output event of one process may be associated with an input event of some other process (here noted $a \| b$ for events a and b) in the network.

Events that are not associated with any other event in a network define an *interface* of the network. Interface of a network N will be denoted as $if(N)$. By definition $\tau \notin if(N)$.

A state of N is a pair $s = (\bar{l}, \nu)$, where $\bar{l}$ is a vector of locations occupied by all network's processes and ν is a valuation of all clocks in the network. A network may perform two types of transitions: a delay transition and discrete transition. The rule for delay transitions is similar that rule for delay transitions of single $TIOA$. There are, however two rules for discrete transitions of N. The first case is when a single process of N performs a transition that is associated with unobservable event τ or with an event that belongs to the interface of N. The second type of transition is when two processes synchronize and move simultaneously. In the second case one of processes performs a discrete transition with an output event that is associated to an input event of the transition performed by the other process.

Let $\bar{l}[l'_i/l_i]$ stand for a vector $\bar{l}$ where l_i has been substituted with l'_i. $Inv(\bar{l})$ means an intersection of all invariants of elements of $\bar{l}$. $t_i \parallel t_j$ denotes that event of transition t_i is an output event associated with input event of transition t_j. Then, all three types of transitions of N can be described as follows:

- *delay transition:* for a state $(\bar{l}, \nu)$ and $d \in \mathbb{R}^{\geq 0}$ $(l, \nu) \xrightarrow{\epsilon(d)}_A (\bar{l}, \nu+d)$, if for all $0 \leq d' \leq d$, $\nu + d' \in I(\bar{l})$,
- *local action transition:* for a state $(\bar{l}, \nu)$ and an edge $t = (l_i, Z, a, r, l'_i)$,

$$(\bar{l}, \nu) \xrightarrow{t} (\bar{l}[l'_i/l_i], \nu')$$

  if $\nu \in Z$, $\nu' = \nu[r := 0] \in I(\bar{l}[l'_i/l_i])$ and $a \in if(N) \cup \tau$,
- *synchronization transition:* for a state $(\bar{l}, \nu)$ and edges $t_1 = (l_i, Z_1, a_1, r_1, l'_i)$ and $t_2 = (l_j, Z_2, a_2, r_2, l'_j)$

$$(\bar{l}, \nu) \xrightarrow{t_1 \| t_2} (\bar{l}[l'_i/l_i][l'_j/l_j], \nu')$$

if $\nu \in Z_1 \cap Z_2$, $\nu' = \nu[r_1 := 0][r_2 := 0] \in I(\bar{l}[l'_i/l_i][l'_j/l_j])$.

Committed locations

The concept of *committed locations* increase expressiveness in modeling networks of automata in UPPAAL. Labelling location as committed has sense only, if the network consists of more than one process. If any process of a network is in a committed location, the next step must involve a transition from one of the committed locations. Furthermore, committed locations, similarly to urgent locations freeze time, which means that committed location must be left at the same time it was entered.

Committed locations are useful for creating atomic sequences and for encoding synchronization between more than two components. Notice that if several processes are in a committed location at the same time, then they will interleave.

3.3.2 Communicating System

Topology of communication

The topology of communication for set of automata is a model of communication between them. It describes dynamic system configuration and possible synchronizations between system's processes in any possible system configuration. The definition of topology is inspired by definition of *synchronization vectors* [10] and work of [17].

A topology defines *channels* for events exchanged by automata in the system. The channel has a form of a vector of events that define the synchronization. Such a vector is associated with one additional event that is seen by the entities outside the CS. Due to this features, topology has the same interace as $TIOA$ and can communicate with its environment.

Definition 10. (Topology) *The topology of communication* **Top** *for set of automata* $N = \{A_1, .., A_n\}$ *is defined by a 3-tuple* (Σ_G, Σ, Tr). *The* Σ_G *is a finite set of global actions,* $\Sigma = \{\Sigma_1, ..., \Sigma_n\}$ *is set of alphabets of automata. The* Tr *is an automaton, for which the alphabet of events* Σ^{Tr} *is defined by a set of vectors of* $n + 1$ *elements in the form* $< a_g, a_1, ..., a_n >$, *where:*

- $a_g \in \Sigma_G$ *is the global action assigned to the synchronization,*

- $\forall i \in [1, n] : a_i \in \Sigma_i \cup \{idle\}$

The synchronization vectors define for each automaton an action that must be performed in the synchronization. If the action for an automaton is *idle* it means that this certain automaton does not take part in the synchronization. The global action a_g is the action that is seen from the environment point of view.

If the automaton Tr has only one location, the topology is called *static*, otherwise it is called *dynamic*. In static topologies, all synchronizations are enabled regardles what happened in the past. The dynamic topologies allow to model more advanced systems, where communication channels are opened and closed depending on the system's history. The form of the synchronization vectors allow to model broadcast, unicast or multicast character of communication between processes.

Definition of Communicating System

In [23] a CS is defined using set of $TIOA$. In this work the definition of CS will be extended such that it may be composed of either primitive processes that are $TIOA$ or it may contain also other CSs of lower level. By this it is possible to model nested structures of many layers.

Definition 11. (Communicating System) *A Communicating System is a 3-tuple* $\{\overline{M}, SC, \mathbf{Top}\}$*, where:*

- $\overline{M}$ *is a set of modules that can be either* $TIOA$ *or* CS*,*
- CS *is a set of shared clocks that are visible for all modules in* $\overline{M}$*,*
- **Top** *is a topology of communication for elements of* $\overline{M}$*.*

If a CS contains a shared clock it must be also a shared clock for all its modules if they are defined as CS of lower level.

Semantics of CS

Semantics of communication system $CS = \{\overline{M}, SC, (\Sigma_G, \Sigma, Tr)\}$ is defined by a $TIOA$ $\xi(CS) = \{L^\xi, l_0^\xi, \Sigma^\xi, C^\xi, \rightarrow^\xi\}$, such that:

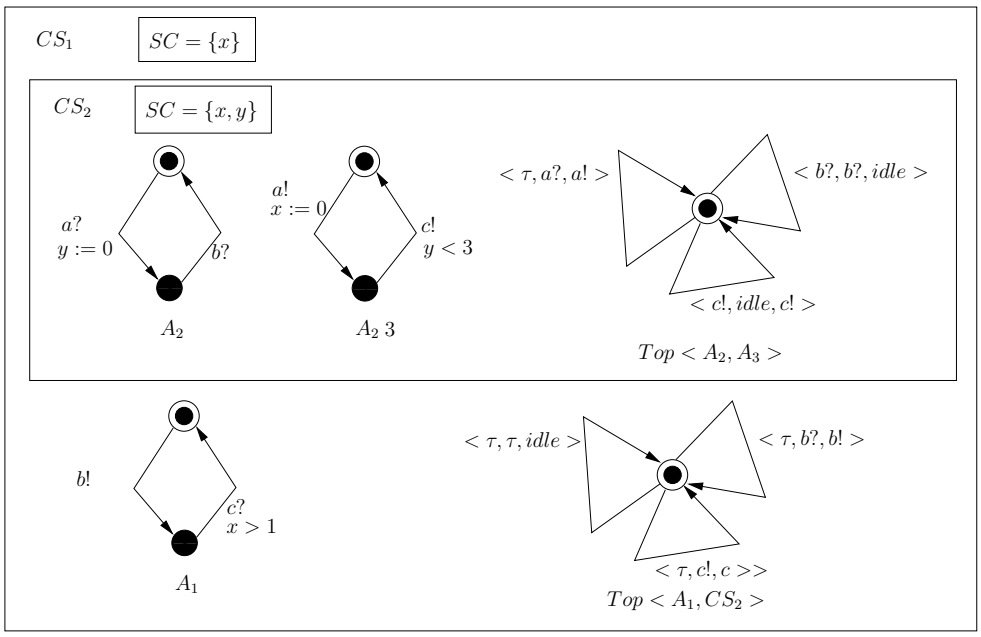


Figure 11: Example of a Communicating System

- $L^\xi = \{< l_{Tr}, l_1, ..., l_n > \mid l_{Tr} \in L(Tr),\ \forall i \in [1, n] : l_i \in L(M_i)\}$ is set of locations of all modules. $L(M_i)$ denotes set of locations of M_i and $L(Tr)$ denotes set of locations in the CS's topology,
- $l_0^\xi = \{l_0^{Tr}, l_0^1, ..., l_0^n\}$ is a vector of initial locations of the topology and modules of CS,
- $\Sigma^\xi = \Sigma$ is the alphabet of CS
- $C^\xi = SC(CS) \cup C(M_i),\ i \in [1, n]$, where $SC(CS)$ denotes set of shared clocks of CS and $C(M_i)$ is set of clocks of module M_i,
- $\rightarrow^\xi = \{(l_{Tr}, l_1, ..., l_n) \xrightarrow{a \in \Sigma^\xi, Z \in \Omega(C^\xi), r \in C^\xi} (l'_{Tr}, l'_1, ..., l'_n)$ is set of transitions between locations in L^ξ.

Exemple 3.4. *Let us consider the communicating system CS_1 from the Figure 11. It consists of another communicating system CS_2 and an automaton A_1. Those two modules share access to the clock x. The communication between CS_2 and A_1 is described by the topology $Top < A_1, CS_2 >$. The module CS_2 consists of two primitive processes A_2 and A_3 which share access to the clock y. The communication between them is described by $Top < A_2, A_3 >$.*

The $Top < A_1, CS_2 >$ *defines three possible synchronizations for* CS_1*. All of them are associated with global event* τ*, which means that they cannot be observed from the environment of* CS_1*. First synchronization is when* CS_2 *performs local action that is associated with* τ*. Process* A_1 *does not perform any action then. Other possibilities are when* A_1 *synchronizes with* CS_2 *by pairs of actions* $b!\|b?$ *and* $c?\|c!$*.*

Within CS_2 *three possible synchronizations are defined as well.* A_2 *can synchronize with* A_3 *by pair of actions* $a?\|a!$*. Processes* A_2 *and* A_3 *can also synchronize with the environment (which is* A_1 *in that case) by events* $b?$ *and* $c!$ *respectively.*

The communication system CS_1 *has defined one shared clock* x*. Because one of the modules of* CS_1 *is communication system* CS_2*, the clock* x *is shared also within this module.* CS_2 *defines also second shared clock* y*. This clock, however, is not seen and cannot be reset outside of* CS_2*.*

3.3.3 Summary

Both CS and automata networks are a way to model communicating systems without necessity of explicitly deriving a parallel composition which may lead to state explosion. The CS however allow to model nested and hierarchical systems while networks of $TIOA$ allow to define only a flat structure of communicating automata. On the other hand committed states introduced for networks of $TIOA$ enable modeling more complex processes that would be very difficult to model using even very advanced dynamic topologies.

4 Symbolic Analysis of Timed Automata

This chapter covers symbolic methods for analysis of timed automata. The Section 4.1 introduces the fundamentals of model checking. The symbolic approach for state representation and reachability analysis is covered by the Section 4.2. The last section introduces a Difference Bound Matrix – a standard data structure for symbolic state representation.

Contents

4.1 Model checking

Model checking is the most successful approach that has emerged for verifying requirements. Pioneering work in the model checking of temporal logic formulae was done in [42, 70]. A comprehensive review about model checking of timed and untimed systems is done in [14]. Model checking is a formal verification technique which allows for desired behavioral properties of a given system to be verified on the basis of a suitable model of the system through systematic inspection of all states of the model in a brute-force manner. In this

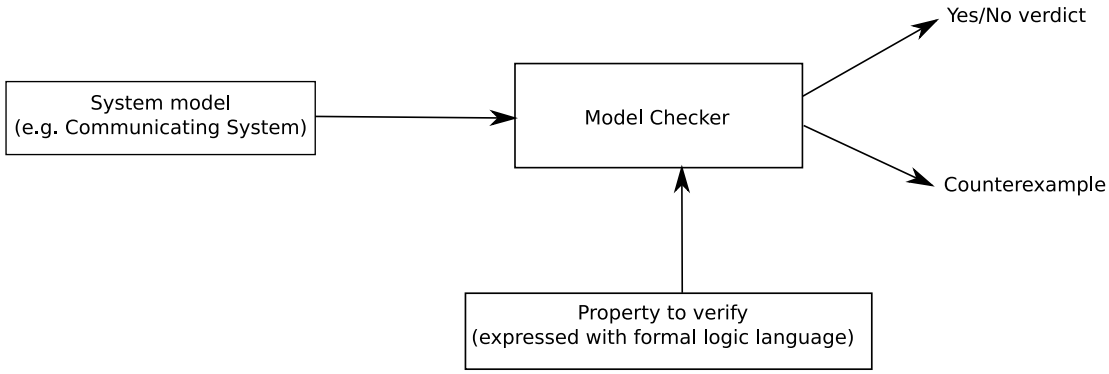


Figure 12: Model checking process

way, it can be shown that a system, represented by a given model truly satisfies a certain property:

$$system \models requirements$$

The model checking process is symbolically presented in the Figure 12. A model checking tool is provided with a formal system model (e.g. communicating system of timed automata) and a property to verify that is expressed by formal logic language (e.g. Timed Computation Tree Logic [41] or Timed Temporal Logic [27]). The model checker by analyzing entire reachability tree of the modelled system returns a verdict whether the property is satisfied and (optionally) a counterexample or a diagnostic trace.

The attractiveness of model checking comes from the fact that it is completely automatic, i.e. the learning curve for a user is very gentle and that it offers counterexamples in case a model fails to satisfy a property serving as indispensable debugging information. On top of this, the performance of model-checking tools has long since proved mature as witnessed by a large number of successful industrial applications (e.g. [81, 45]).

The main difficulty of the model-checking of real-time systems defined as timed automata is that uncountably many states have to be analyzed, since the semantics of the time automata is defined by an infinite transition system, say $TS(TA)$. A naive graph analysis in the state graph of $TS(TA)$ is therefore not feasible. Instead, the basic idea is to consider a finite quotient of this transition system, the so-called *region transition system*. The states in the region transition system are equivalence classes of states in $TS(TA)$ that all satisfy

the same atomic clock constraints, and from which "similar" time-divergent paths emanate. As the number of equivalence classes is finite, this provides a basis for model checking.

In practice, the equivalence classes are calculated "on-the-fly", during *symbolic reachability analysis*. Symbolic reachability analysis is a powerful paradigm for verification of infinite-state systems. Symbolic reachability analysis uses finite structures to represent infinite sets of configurations (see [39, 57]), 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.

4.2 Symbolic Path

4.2.1 Path

A path $\rho = [t_0, ..., t_n]$ of TA $A = (L, l_0, C, \Sigma, \rightarrow)$ is a sequence of transitions $t_i \in \rightarrow$ such that:

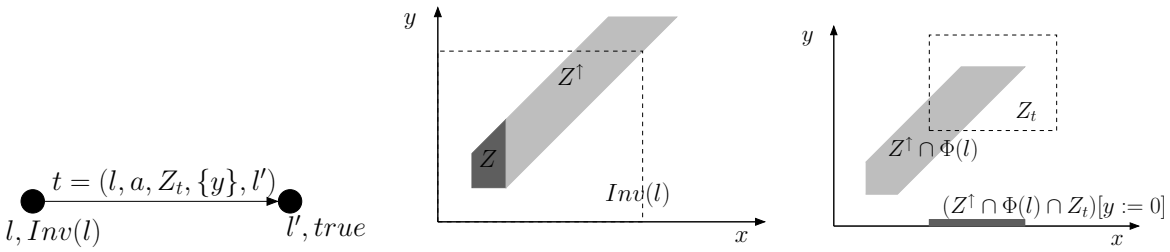
$$src(t_i) = \begin{cases} l_0 & \text{if } i = 0 \\ dest(t_{i-1}) & \text{otherwise} \end{cases}$$

4.2.2 Zones

A zone of TA $A = (L, l_0, \mathcal{C}, \Sigma, \rightarrow)$ is a pair (l, Z) where $l \in L$ is the system location and Z is a polyhedron containing possible valuations of system's clocks. For zones $H = (l, Z)$ and $H' = (l', Z')$ operations of inclusion and intersection are defined in the following way:

$$H \subseteq H' \;\Leftrightarrow\; l = l' \;\wedge\; Z \subseteq Z'$$

$$H \cap H' = \begin{cases} (l, Z \cap Z') & \text{, if } (l = l') \wedge (Z \cap Z' \neq \emptyset) \\ \emptyset & \text{, otherwise} \end{cases}$$



(a) automaton A (b) $Z^{\uparrow} \cap Inv(l)$ (c) $post((l, Z), t)$

Figure 13: The *post* operation

4.2.3 Symbolic operations on zones

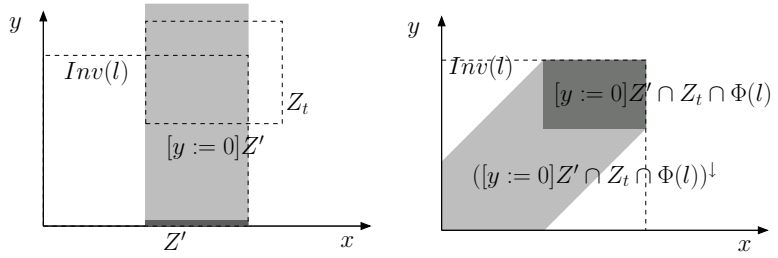
Let $H = (l, Z)$ be a zone and $t = (l, Z, a, r, l')$ be a transition of TA (with invariants) $A = (L, l_0, \mathcal{C}, \Sigma, Inv, \rightarrow)$.

Operation *post*()

The operation $post(H, t)$ returns a zone that contains all states that can be occupied by A after it performes transition t from zone H. Operation $post(H, t)$ is defined as follows:

$$post(H, t) = \Big(l', \Big(\big(Z^{\uparrow} \cap Inv(l)\big)[r := 0]\Big) \cap Inv(l')\Big)$$

Exemple 4.1. *Consider the automaton A from Figure 13(a) being in a zone (l, Z) (Z is depicted in 13(b)) performing transition $l \xrightarrow{Z_t, a, \{y\}} l'$. The polyhedron $Z^{\uparrow}$ illustrates all valuations that can be reached from Z by time elapse. However A can stay in the location l only when the valuations of it's clocks satisfy the invariant $Inv(l)$. The transition t may be performed only, if the clocks' valuations belongs to the polyhedron Z_t; this means that at the moment of executing t possible valuations are defined by intersection of all valuations that could be reached from Z in the location l $(Z^{\uparrow} \cap Inv(l))$ with the guard Z_t (see Figure 13(c)). At the moment of executing t the clock y is reset, what is symbolically represented by performing forward clock reset operation. Since the invariant for l' is expressed by* **true**, *the zone reached directly after performing t is defined by $(Z^{\uparrow} \cap Inv(l) \cap Z_t)[y := 0]$.*



(a) $[y := 0]Z' \cap Z_t \cap I(l)$ (b) $pred((l', Z'), t)$

Figure 14: The *pred* operation

Operation $pred()$

The operation $pred(H, t)$ is a reverse operation to $post()$. For an automaton that is in a zone $H = (l', Z')$ and a transition $t = (l, Z_t, a, r, l')$ the operation $pred(H, t)$ returns a zone that could be occupied before performing the transition t such that execution of t results in a state in H. The operation $pred(H, t)$ is defined as follows:

$$pred(H, t) = \big([r := 0]Z \cap Z_t \cap Inv(l)\big)^{\downarrow} \cap Inv(l)$$

Exemple 4.2. *Consider again the automaton A from Figure 13(a) this time in the zone $H' = (l', Z')$. The operation $[y := 0]Z'$ returned all valuations that could be occupied before reseting clock y (see Figure 14(a)). The valuations at the moment of executing t must have belonged to the guard polyhedron Z_t and to invariant of l, which is expressed by $[y := 0]Z' \cap Z_t \cap Inv(l)$. Applying backward time elapse operation on this polyhedron will return polyhedron containing all valuations from which $[y := 0]Z' \cap Z_t \cap Inv(l)$ could be reached by waiting some time (see Figure 14(a)). Finally the invariant of l must be used again to make the resulting polyhedron contain only valuations valid in l.*

post-pred stability

Let $H = (l, Z)$, H_1 and H_2 be three zones and $t_1 = (l_1, Z_1, a_1, r_1, l)$ and $t_2 = (l, Z_2, a_2, r_2, l_2)$ be two transitions of an TA A. Assume that $H = post(H_1, t_1)$ and $H = pred(H_2, t_2)$. Then:

- *pred stability of the post operation:* For all $q = (l, \nu) \in H$, there exist $q_1 = (l_1, \nu_1) \in H_1$, and $d_1 \geq 0$ such that $\nu_1 + d_1 \in Z_1$ and $\nu = (\nu_1 + d_1)[r_1 := 0]$. We say that H is *pred-stable* to H_1 by t_1. q_1 is called *predecessor state* of q.

- *post stability of the pred operation:* For all $q = (l, \nu) \in H$, there exist $q_2 = (l_2, \nu_2) \in H_2$, and $d_2 \geq 0$ such that $\nu_2 + d_2 \in Z_2$ and $(\nu + d_2)[r_2 := 0] = \nu_2$. We say that H is *post-stable* to H_2 by t_2. q_2 is called *successor state* of q.

Intuitively, post-pred stability means that for all states $(l, \nu) \in H$, there must be a state in H_1 that allows reaching (l, ν) by performing transition t_1. Also for any state $(l, \nu) \in H$ it is possible to reach a state in H_2 by performing a transition t_2.

4.2.4 Symbolic path analysis

Let $\rho = [t_1, \cdots, t_n]$ be a sequence of transitions of an automaton A such that $t_i = (l_{i-1}, Z_i, a_i, r_i, l_i)$, for all $i \in [1, n]$ and let $\mathcal{C}$ be a set of clocks of A.

Forward path analysis

For all $i \in [0, n]$:

$$\begin{cases} H_i = (l_0, \mathbf{zero}) & , \text{ if } i = 0 \\ H_i = post(H_{i-1}, t_i) & , \text{ otherwise} \end{cases}$$

Intuitively, the initial state of the system (H_0) is the initial location with all clocks set to 0. Then next zones H_i are obtained by $post()$ operation and are pred-stable to H_{i-1} by t_i.

Exemple 4.3. *Consider following path:*

$$\rho \ : \ l_0 \xrightarrow{x \leq 2, y:=0} l_1 \xrightarrow{y \leq 1, x:=0} l_2$$

By forward analysis of ρ we define:

$$\begin{cases} H_0 = (l_0, \mathbf{zero}) \\ H_1 = post(H_0, t_1) = (l_1, x \leq 2 \wedge y = 0) \\ H_2 = post(H_1, t_2) = (l_2, x = 0 \wedge y \leq 1) \end{cases}$$

The following corollary arises from the definition of the $post()$ operation:

Corollary 2. *The final location l_n of the path ρ is reachable if and only if the zone H_n is not empty.*

A symbolic path $S^+(\rho)$ associated with path ρ is a sequence of zones obtained by forward analysis of ρ:

$$S^+(\rho) \ : \ H_0 \xrightarrow{t_1} H_1 \cdots H_{n-1} \xrightarrow{t_n} H_n$$

Backward analysis

For all $i \in [0, n]$:

$$\begin{cases} H_i = \big(l_n, Inv(l_n)\big) & , \text{ if } \ i = n \\ H_i = pred(H_{i+1}, t_{i+1}) & , \text{ otherwise} \end{cases}$$

Exemple 4.4. *Consider following path:*

$$\rho \ : \ l_0 \xrightarrow{x \leq 2, y:=0} l_1 \xrightarrow{y \leq 1, x:=0} l_2$$

The zones obtained by backward analysis:

$$\begin{cases} H_2 = (l_0, \mathbf{true}) \\ H_1 = pred(H_2, t_2) = (l_1, x \geq 0 \wedge y \leq 1) \\ H_0 = pred(H_1, t_1) = (l_0, x \leq 2 \wedge y \geq 0) \end{cases}$$

Corollary 3. *The final location l_n of the path ρ is reachable if and only if $H_0 \cap (l_0, \mathbf{zero})$ is not empty.*

A symbolic path $S_-(\rho)$ associated with a path ρ is a sequence of zones obtained by backward analysis of ρ:

$$S_-(\rho) \ : \ H_0 \xrightarrow{t_1} H_1 \cdots H_{n-1} \xrightarrow{t_n} H_n$$

Forward-backward analysis

The pred-stability property of the $post()$ guarantee that each state $q \in H_i$ has a predecessor in H_{i-1}. On the other hand it does not guarantee that from all $q \in H_i$ the successor in H_{i+1} can be reached. The latter is guaranteed by the post-stability property of the $pred()$ operation. Therefore operations $post()$ and $pred()$ can be combined in order to formulate forward-backward analysis of the path ρ:

For all $i \in [0, n]$:

$$\begin{cases} H_i = (l_0, \mathbf{zero}) & , \text{ if } i = 0 \\ H_i = post(H_{i-1}, t_i) & , \text{ otherwise} \end{cases}$$

and:

$$\begin{cases} H'_i = H_i & , \text{ if } i = n \\ H'_i = H_i \cap pred(H'_{i+1}, t_{i+1}) & , \text{ otherwise} \end{cases}$$

By this, each H'_i verifies the post/pred stability property for all $i \in [0, n]$.

Exemple 4.5. *Consider following path:*

$$\rho \ : \ l_0 \xrightarrow{y:=0} l_1 \xrightarrow{y=1 \wedge x \leq 3} l_2$$

By forward analysis of ρ we get:

$$\begin{cases} H_0 = (l_0, \mathbf{zero}) \\ H_1 = post(H_0, t_1) = (l_1, x \geq 0 \wedge y = 0) \\ H_2 = post(H_1, t_2) = (l_2, x \leq 3 \wedge y = 1) \end{cases}$$

Notice that although the state $(l_1, [x = 3, y = 0])$ has a predecessor in H_0 it does not have a successor in H_2. Applying backward analysis to the path gives following result:

$$\begin{cases} H'_2 = H_2 = (l_2, x \leq 3 \wedge y = 1) \\ H'_1 = H_1 \cap pred(H'_2, t_2) = \big(l_1, (x \geq 0 \wedge y = 0) \cap (x \leq 3 \wedge y \leq 1 \wedge x - y \leq 2)\big) = \\ \quad = \big(l_1, (x \leq 2, y = 0)\big) \\ H'_0 = H_0 \cap pred(H_1, t_1) = (l_0, \mathbf{zero}) \end{cases}$$

Corollary 4. *For each state $(l, \nu) \in H'_i$ there exists a computation $r \ : \ (\bar{l}, \bar{\nu})$ over a timed sequence σ such that: $l_i = l$ and $\nu_i = \nu$.*

In other words, corollary 4 says that for each state in zone obtained by forward-backward analysis of path σ there exist a computation over σ that covers this state.

Finally a symbolic path $S^+_-(\rho)$ is defined as a sequence of zones obtained by forward-backward analysis of ρ:

$$S^+_-(\rho) \ : \ H'_0 \xrightarrow{t_1} H'_1 \cdots H'_{n-1} \xrightarrow{t_n} H'_n$$

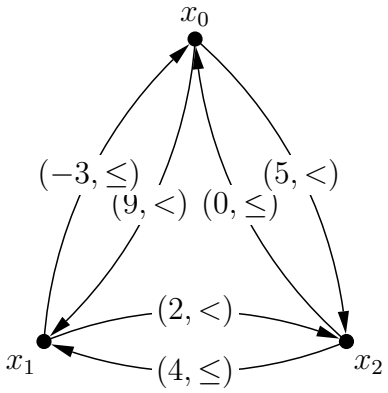
4.3 Difference Bounds Matrix

A Difference Bounds Matrix (DBM) is a data structure used for storing and processing polyhedra which are represented by a constraint graph.

Definition 12. *A Difference Bound Matrix (DBM) is a square matrix, where each row and each column is labelled with a clock $x_i \in \mathcal{C}$. One additional row and column represent the reference clock x_0 always equal to 0. Elements $M_{i,j}$ of DBM define bounds of the difference $x_i - x_j$ such that x_i is a clock labelling row i and x_j labells the column j.*

Later $row(i)$ (resp. $column(i)$) will denote the clock labelling ith row (resp. ith column) of a DBM. $length(M)$ will denote number of rows of DBM M. By convention DBMs are organized in the way that rows and columns with the same index are labelled with the same clock ($\forall i \in [1..length(M)] \ : \ row(i) = column(i)$). The first row and column is labelled with the reference clock x_0.

The Figure 15 shows the constraint graph from Example 2.4 and its corresponding DBM.



$$M = \left(\begin{array}{c|ccc} & x_0 & x_1 & x_2 \\ \hline x_0 & (0,\leq) & (-3,\leq) & (0,\leq) \\ x_1 & (9,\leq) & (0,\leq) & (4,\leq) \\ x_2 & (5,\leq) & (2,\leq) & (0,\leq) \end{array}\right)$$

Figure 15: Constraint graph and its corresponding DBM

4.3.1 Minimal DBMs

Minimal DBM is such a DBM that the constraint graph that it represents is minimal. The Algorithm 4.1 defines function $minimal(M)$ that for a DBM M returns DBM that is minimal and represents canonical form of a polyhedron represented by M. The algorithm is implementation of Floyd-Warschall shortest path algorithm.

Algorithm 4.1 $minimal(M)$

Arguments: DBM M

Returned value: DBM that is minimal

```
for all k ∈ [1..length(M)] do
  for all i ∈ [1..length(M)] do
    for all j ∈ [1..length(M)] do
      M_{i,j} = min(M_{i,j}, M_{i,k} + M_{k,j});
    end for
  end for
end for
return M;
```

Property 6. *A polyhedron represented by DBM M is not empty if and only if the diagonal of $minimal(M)$ contains only bounds not lower than $(0, \leq)$.*

The Property 6 is a consequence of the Property 4 about graphs with non-negative cycles.

The diagonal of DBM that is result of operation *minimal*() contains weights of loops of the constraint graph. Therefore if any element of diagonal is negative it means that the graph contains negative cycles.

4.3.2 Operations on DBM

The operations on DBM can be divided into two classes:

1. Property checking: this class includes operations that check emptiness of polyhedron represented by DBM, inclusion between two polyhedra and whether a polyhedron represented by the DBM satisfies a given constraint.
2. DBM transformation: this class contains operations transforming the polyhedron represented by the DBM. This includes intersection, time elapse and clock reset.

In following of this section we assume that the input matrices for the operations are minimal and they represent non-empty polyhedra. The algorithms for DBM processing are based on those described in [22].

Property checking

empty(M) The operation $empty(M)$ returns a boolean value indicating whether the polyhedron represented by M is empty. According to the Property 6 to check it, it is enough to check whether the diagonal contains bounds lower than $(0, \leq)$. The operation $empty(M)$ is implemented by the Algorithm 4.2.

includes$(\mathbf{M}, \mathbf{M'})$ The function $includes(M, M')$ checks inclusion relation for DBMs M and M'. It returns *true* if all valuations that belong to polyhedron represented by M belong also to the polyhedron represented by M'. In other words, if M represents polyhedron Z and M' represents Z', the function $includes(M, M')$ returns:

$$\begin{cases} \textbf{true} & \text{if} \quad Z \subseteq Z' \\ \textbf{false} & \text{otherwise} \end{cases}$$

If M and M' are in minimal form it is enough to check whether all bounds defining M are lower or equal than corresponding bounds defined by M'. This test is implemented by

Algorithm 4.2 $empty(M)$

Arguments: DBM M that is minimal

Returned value: a boolean value indicating whether M represents empty portion of space

```
for all i ∈ [1..length(M)] do
  if M_{i,i} < (0, ≤) then
    return true
  end if
end for
return false
```

Algorithm 4.3

Algorithm 4.3 $includes(M, M')$

Arguments: DBMs M and M' that are minimal

Returned value: a boolean value indicating whether polyhedron represented by M is included by polyhedron represented by M'.

```
for all i ∈ [1..length(M)] do
  for all j ∈ [1..length(M)] do
    if M_{i,j} > M'_{i,j} then
      return false;
    end if
  end for
end for
return true;
```

satisfies$(\mathbf{M}, \mathbf{x_i} - \mathbf{x_j} \prec \mathbf{m})$ This function checks whether a polyhedron defined by M satisfies a constraint $v_i - v_j \prec m$. In other words it checks, whether adding the constraint $v_i - v_j \prec m$ to the polyhedron defined by M will not result in an empty polyhedron. Note that adding the constraint $v_i - v_j \prec m$ to polyhedron represented by M will cause changing element $M_{i,j}$ to $min\big(M_{i,j}, (m, \prec)\big)$. Thus, to verify whether the resulting polyhedron will not be empty, it is enough to check whether $(0, \leq) \leq (m, \prec) + M_{j,i}$. The operation is

implemented by the Algorithm 4.4.

Algorithm 4.4 $satisfies(M, M')$

Arguments: Minimal DBM M and constraint $x_i - x_j \prec m$

Returned value: a boolean value indicating whether polyhedron represented by M after adding the constraint $x_i - x_j \prec m$ is not empty.

```
if (0, ≤) ≤ M_{j,i} + (m, ≺) then
  return true
else
  return false
end if
```

Transformations

and$(\mathbf{M}, \mathbf{x_i} - \mathbf{x_j} \prec \mathbf{m})$ Operation $and(M, x_i - x_j \prec m)$ represent adding the constraint $x_i - x_j \prec m$ to a polyhedron represented by M. The basic step for this operation is to check whether $(m, \prec) < M_{i,j}$ and if so, replacing $M_{i,j}$ with $(m, \prec)$. If the element $M_{i,j}$ has been changed, the matrix must be minimized again. It can be done using the $minimal()$ function, however it is possible to derive an algorithm that takes advantage of the fact that only one bound was altered and has $\mathcal{O}(n^2)$ complexity. The pseudocode is illustrated by the Algorithm 4.5.

To prove that the Algorithm 4.5 is correct it is more convenient to use constraints graphs. Adding constraint $x_i - x_j \prec m$ to the polyhedron represented by graph $G = (N, \omega, E)$ is equivalent to replacing weight of the edge $x_i \to x_j$ with the bound $(m, \prec)$. Note, that this means that the weight of each edge in the graph is now equal to the weight of the shortest from all paths between nodes n_x and n_y that traverse all nodes except n_i and n_j. This is analogical situation to Floyd-Warschall algorithm when the paths that traverse all nodes except x_i and x_j have been checked. Therefore it is enough to perform only two iterations of the outer loop of Floyd-Warschall algorithm to check paths traversing nodes of the altered edge.

Algorithm 4.5 $and(M, x_i - x_j \prec m)$

Arguments: DBMs M and that is minimal and a bound of $x_i - x_j$

Returned value: Minimal DBM M' that represents intersection of M and the bound.

```
if M_{j,i} + (m, ≺) < (0, ≤) then
  M_{0,0} = (−1, ≺) ;
else if (m, ≺) < M_{i,j} then
  M_{i,j} = (m, ≺)
  for all k ∈ {i, j} do
    for all x ∈ [1..length(M)] do
      for all y ∈ [1..length(M)] do
        M_{x,y} = min(M_{x,y}, M_{x,k} + M_{k,y})
      end for
    end for
  end for
end if
return M
```

intersection$(\mathbf{M}, \mathbf{M}')$ Intersecting two polyhedra is after minimization, the most often performed operation on DBMs. Intersecting of two DBMs is basically the same that intersecting each bound of the DBMs. Therefore to obtain a matrix that represent a polyhedron that is a result of intersection it is enough to build a matrix where each element is the lower element of two intersected matrices. This will not however preserve the canonical form, so the matrix needs to be minimized after that. For pseudocode for $intersection(M, M')$ see Algorithm 4.6.

Algorithm 4.6 $intersection(M, M')$

Arguments: DBMs M and M' that are minimal

Returned value: A DBM that represents polyhedron that is intersection of M and M'.

```
for all i ∈ [1..length(M)] do
  for all j ∈ [1..length(M)] do
    if M'_{i,j} < M_{i,j} then
      M_{i,j} = M'_{i,j};
    end if
  end for
end for
return M
```

Very often one of the intersected matrices is known and fixed during the analysis. For example all DBMs that represent transition guards or invariants are defined before the symbolic analysis and do not change. Then it is more efficient to offline (in advance to the analysis) extract the minimal constraint system for the polyhedron represented by the matrix (see Section 2.4.6) and perform *and*() operation for each non-redundant constraint.

future(M) For a DBM M representing polyhedron Z, the operation $future(M)$ returns a DBM that represents polyhedron $Z^{\uparrow}$, i.e. all valuations that can be reached by valuations in Z by delay.

Algorithmically $future(M)$ is computed by removing the upper bounds of all individual clocks, which is done by replacing all elements in the first column of M by $(\infty, <)$. the property that all clocks proceed with the same speed is ensured by keeping the constraints on the differences between clocks unchanged.

The operation preserves the minimal form of M. The pseudo-code for $future(M)$ is presented in Algorithm 4.7.

Algorithm 4.7 $future(M)$

Arguments: Minimal DBMs M that is representation of polyhedron Z

Returned value: Minimal DBM M' that represents $Z^{\uparrow}$.

```
for all i ∈ [1..length(M)] do
  M_{i,0} = (∞, <)
end for
return M
```

past(M) For a DBM M representing a polyhedron Z, the operation $past(M)$ results with a matrix M' representing a polyhedron $Z^{\downarrow}$, i.e. polyhedron containing all the valuations that can reach Z by delay.

Algorithmically, operation $past(M)$ can be done by assigning $(0, \leq)$ to all elements of the first column of M. This may result in a DBM that is not minimal. The pseudocode for $past(M)$ that returns a matrix that is minimal is presented by Algorithm 4.8.

Algorithm 4.8 $past(M)$

Arguments: Minimal DBMs M that is representation of polyhedron Z

Returned value: Minimal DBM that represents $Z^{\downarrow}$.

```
for all i ∈ [1..length(M)] do
  M_{0,i} = (0, ≤);
  for all j ∈ [1..length(M)] do
    if M_{j,i} < M_{0,i} then
      M_{0,i} = M_{j,i};
    end if
  end for
end for
return M
```

reset(M,X) For a matrix M that represents a polyhedron Z, operation $reset(M, X)$ returns a matrix that represent polyhedron $Z[X := 0]$. The reset operation can be simply applied by changing the elements $M_{i,0}$ and $M_{0,i}$ to $(0, \leq)$ for all i such that $row(i) \in X$ and remove (i.e. replace with $(\infty, <)$) all other bounds in row and column labelled with the reset clock. However, this will result in a DBM that is not minimal. Instead it is more efficient to replace rows and columns representing the reset clocks with row and column labelled with the reference clock (first row and column). The pseudocode for this operation is illustrated by the Algorithm 4.9.

Algorithm 4.9 $reset(M, X)$

Arguments: Minimal DBMs M representing Z and set of reset clocks X

Returned value: Minimal DBM that represents $Z[X := 0]$.

```
for all i ∈ [1..length(M)] do
  if row(i) ∈ X then
    for all j ∈ [1..length(M)] do
      M_{i,j} = M_{0,j};
      M_{j,i} = M_{j,0};
    end for
  end if
end for
return M
```

unreset(M,X) This is the most unintuitive operation. If the matrix M represents polyhedron Z, the operation $unreset(M, X)$ will return a matrix that represents $[X := 0]Z$ – a polyhedron that contains all valuations that after assigning 0 to clocks in X will end up as valuations of Z. This operation is simple, if M represents a polyhedron containing only such valuations ν that $\nu(x) = 0$ for all $x \in X$. In that case it is enough to replace all constraints on $x_i - x_j$ where $x_i \in X$ with $(\infty, <)$. This corresponds to filling the row i with $(\infty, <)$.

The situation is a bit more complicated if Z contains valuations ν, such that $\nu(x) > 0$ for some $x \in X$. Then, the polyhedron Z must be reduced to such a polyhedron that contains only those valuations ν for which $\nu(x) = 0$ for all $x \in X$. This is done by applying the

operation $and(M, x_i - x_0 \leq 0)$ for all x_i such that upper constraint on x_i is bigger than $(0, \leq)$. Note that if the lower bound on x_i is different than $(0, \leq)$ this operation will result in DBM representing empty polyhedron, which is rather intuitive.

The pseudocode for this operation is by the Algorithm 4.10.

Algorithm 4.10 $unreset(M, X)$

Arguments: Minimal DBMs M representing Z and set X of clocks to be unreset
Returned value: Minimal DBM that represents $[X := 0]Z$.

```
for all i ∈ [1..length(M)] do
  if row(i) ∈ X then
    if M_{i,0} ≠ (0, ≤) then
      and(M, x_i − x_0 ≤ 0);
    end if
    for all j ∈ [1..length(M)] do
      M_{i,j} = (∞, <);
    end for
  end if
end for
return M
```

5 Parameterized systems

This chapter is a state-of-the-art report on approaches for parametric analysis of real-time systems. The background is presented in the Section 5.1. Then, the Section 5.2 introduces Parametric Timed Automata (PTA) – an extension of TA for parametric modeling of real time systems. The section 5.3 discuss parametric DBM – the current framework used in symbolic parametric analysis of PTA. The last section summarizes the chapter.

Contents

5.1 Parametric reasoning

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. There are currently several tools that can do analysis of parameterized timed systems: HyTech [49], LPCM [75], TReX [9], TGSE [24] and an extension of UppAal [51, 7]. Some of them, like TReX use constrained Parametric DBM for symbolic state representation. Other tools, like TGSE, use external applications (e.g. lp_solve)

for solving parameterized linear constraint systems.

There are several works where existing tools for parametric verification have been compared. In [33] the tools mentioned above (without TGSE) have been confronted in realistic case study on IEEE 1394 protocol. [62] compares HYTECH, UPPAAL and TREX in a study on the PGM protocol.

This chapter explores current data structures that are used for modeling and analysis of parameterized timed systems. In the next chapter a new data structure that enhances and boosts possibilities of analysis and verification of parametric systems is presented.

5.2 Parametric Timed Automata

5.2.1 Preliminaries

Let $\mathcal{C}$ be a set of clocks, $\mathcal{P}$ be a set of parameters and let $AT(\mathcal{P})$ define set of algebraic expressions with parameters in $\mathcal{P}$. The set of all possible configurations of values of parameters will be denoted by $\mathcal{V}(\mathcal{P})$. The parameterized atomic constraint is an expression in the form:

$$x_i - x_j \bowtie t$$

such that $x_i, x_j \in \mathcal{C}$, $\bowtie \in (<, \leq, =, \geq, >)$ and $t \in AT(\mathcal{P})$.

A set of finite conjunction of parameterized constraints will be noted as $\Omega(\mathcal{C}, \mathcal{P})$. Elements of $\Omega(\mathcal{C}, \mathcal{P})$ are called *parameterized polyhedra*.

A parameterized bound is a pair $b = (t, \prec)$ that is used as a limit for a parameterized atomic constraint of a type $x_i - x_j \prec t$. The set of parameterized bounds is defined by:

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

In order to limit the set of values taken by parameters, the notion of a *constrained parameterized bound* was defined: $\widetilde{b} = (b, \varphi)$ where b is a parameterized bound and $\varphi \in \Phi(\mathcal{P})$ is a set of constraints that should be satisfied by the parameters. φ is a set of formulas over

P given by the grammar:

$$\varphi ::= t_1 \leq t_2 \mid \ \varphi \mid \varphi \wedge \varphi$$

with $t_1, t_2 \in AT(\mathcal{P})$.

The set of constrained parameterized bounds is defined by:

$$\widetilde{\mathcal{PB}} = \mathcal{PB} \times \Phi(\mathcal{P})$$

Inclusion relation

Let $\tilde{b_1} = (b_1,\ \varphi_1)$ and $\tilde{b_2} = (b_2,\ \varphi_2)$ be two constrained parameterized bounds.

$\widetilde{b_1} \subseteq \widetilde{b_2}$ if and only if for any possible values of parameters that satisfy constraints $\varphi_1 \wedge \varphi_2$ the relation $b_1 \leq b_2$ is true. The bound $(\infty, <)$ satisfies all $\widetilde{b} \in \widetilde{\mathcal{PB}}$ i.e. $\widetilde{b} \subseteq \big((\infty, <), true\big)$ for any $\widetilde{b} \in \widetilde{\mathcal{PB}}$.

The strict relation $\subset$ is similar. We say that $\widetilde{b_1} \subset \widetilde{b_2}$ if and only if for any possible values of parameters that satisfy constraints $\varphi_1 \wedge \varphi_2$ the relation $b_1 < b_2$ is true.

Bounds $\widetilde{b_1}$ and $\widetilde{b_1}$ are equal if and only if $\widetilde{b_1} \subseteq \widetilde{b_2} \wedge \widetilde{b_2} \subseteq \widetilde{b_1}$.

Operator $\oplus$

Operator $\oplus : \ \widetilde{\mathcal{PB}} \times \widetilde{\mathcal{PB}} \mapsto \widetilde{\mathcal{PB}}$ on parameterized bounds is defined as follows.

Let $\tilde{b_1} = \big((t_1, \prec_1),\ \varphi_1\big)$ and $\tilde{b_2} = \big((t_2, \prec_2)\ \varphi_2\big)$ be two constrained parameterized bounds. Then:

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

By definition for all $t \in AT(\mathcal{P})$:

$$t + \infty = \infty$$

$$t + (-\infty) = -\infty$$

$$\infty + \infty = \infty$$

$$\infty + (-\infty) = \infty$$

$$(-\infty) + (-\infty) = -\infty$$

The bound $(0, true)$ is an neutral element for operator $\oplus$:

$$\forall \widetilde{b} \in \widetilde{\mathcal{PB}} \ : \ \widetilde{b} \oplus (0, true) = \widetilde{b}$$

The bound $(\infty, false)$ is an absorbing element element for operator $\oplus$:

$$\forall \widetilde{b} \in \widetilde{\mathcal{PB}} \ : \ \widetilde{b} \oplus (\infty, false) = (\infty, false)$$

Operator $\otimes$

Before defining operator $\otimes$ it is necessary to define the minimum between two constrained bounds. For this following formulae need to be defined:

$$\Phi_{\leq} \equiv \exists p \in \mathcal{V}(\mathcal{P}) \mid \varphi_1 \wedge \varphi_2 \wedge t_1 \leq t_2$$

$$\Phi_{\geq} \equiv \exists p \in \mathcal{V}(\mathcal{P}) \mid \varphi_1 \wedge \varphi_2 \wedge t_1 \geq t_2$$

Intuitively, the formula $\Phi_{\leq}$ (resp. $\Phi_{\geq}$) means: there exists such a configuration of values of parameters from $\mathcal{P}$, that all the expression $\varphi_1 \wedge \varphi_2 \wedge t_1 \leq t_2$ (resp. $\varphi_1 \wedge \varphi_2 \wedge t_1 \geq t_2$) is true.

Operator $\otimes : \ \widetilde{\mathcal{PB}} \times \widetilde{\mathcal{PB}} \mapsto 2^{\widetilde{\mathcal{PB}}}$ is defined by: $\widetilde{b_1} \otimes \widetilde{b_2} = min(\widetilde{b_1}, \widetilde{b_2})$, where the function $min(\widetilde{b_1}, \widetilde{b_2})$ is defined in the following way:

$$min(\widetilde{b_1}, \widetilde{b_2}) = min_{\leq}(\widetilde{b_1}, \widetilde{b_2}, \Phi_{\leq}) \cup min_{\geq}(\widetilde{b_1}, \widetilde{b_2}, \Phi_{>})$$

with

$$min_{\leq}(\widetilde{b_1}, \widetilde{b_2}, \Phi_{\leq}) = \begin{cases} \big((t_1, \prec_1), \ \varphi_1 \wedge \varphi_2 \wedge (t_1 \leq t_2)\big) & \text{, if } \Phi_{\leq} \\ \emptyset & \text{, otherwise} \end{cases}$$

$$min_{\geq}(\widetilde{b_1}, \widetilde{b_2}, \Phi_{\geq}) = \begin{cases} \big((t_2, \prec_2), \ \varphi_1 \wedge \varphi_2 \wedge (t_1 \geq t_2)\big) & \text{, if } \Phi_{\geq} \\ \emptyset & \text{, otherwise} \end{cases}$$

The operation $\widetilde{b_1} \otimes \widetilde{b_2}$ may return one or two constrained parameterized bounds. The combination depends on satisfiability of $\Phi_{\leq}$ and $\Phi_{\geq}$. The neutral element for $\otimes$ is $\big((\infty, <), true\big)$: $\forall \widetilde{b} \in \widetilde{\mathcal{PB}} : \ \widetilde{b} \otimes \big((\infty, <), true\big) = \widetilde{b}$

Exemple 5.1. *Let* $\widetilde{b_1} = \big((p_1, <), p_1 \geq 2 \wedge p_1 \leq 6\big)$ *and* $\widetilde{b_2} = \big((p_1 + p_2, \leq), p_1 \geq 4 \wedge p_2 \leq 8)\big)$ *be two constrained parameterized bounds such that* $p_1, p_2 \in \mathcal{P}$*. The operations of* $\widetilde{b_1} \oplus \widetilde{b_2}$ *and* $\widetilde{b_1} \otimes \widetilde{b_2}$ *give following results:*

$$\widetilde{b_1} \oplus \widetilde{b_2} = \big((2p_1 + p_2, <),\ p_1 \geq 4 \wedge p_1 \leq 6 \wedge p_2 \leq 8\big)$$

The result of $\widetilde{b_1} \otimes \widetilde{b_2}$ *is an union of* $min_{\leq}(\widetilde{b_1}, \widetilde{b_2}, \Phi_{\leq})$*, and* $min_{\geq}(\widetilde{b_1}, \widetilde{b_2}, \Phi_{\geq})$*, where:*

$$min_{\leq}(\widetilde{b_1}, \widetilde{b_2}, \Phi_{\leq}) = \big((p_1, <),\ p_1 \geq 4 \wedge p_1 \leq 6 \wedge p_2 \leq 8 \wedge p_2 \geq 0\big)$$

$$min_{\geq}(\widetilde{b_1}, \widetilde{b_2}, \Phi_{\geq}) = \big((p_1 + p_2, \leq),\ p_1 \geq 4 \wedge p_1 \leq 6 \wedge p_2 \leq 0\big)$$

5.2.2 Definition of PTA

Parametric TA (PTA) is a TA that is extended with parameters. Transition guards and invariants of PTA may have form of conjunction of parameterized atomic constraints. The following definition of PTA was inspired by [8]:

Definition 13. Parametric Timed Automaton *A Parametric Timed Automaton (PTA) is a tuple* $\big(L, l_0, \Sigma, \mathcal{C}, \mathcal{P}, \varphi, Inv, \rightarrow\big)$*, where:*

- L *is a finite set of locations,*
- l_0 *is an initial location,*
- Σ *is an alphabet of events,*
- $\mathcal{C}$ *is a finite set of clocks,*
- $\mathcal{P}$ *is a finite set of parameters,*
- φ *is a conjunction of initial constraints on parameters,*
- $Inv\ :\ L \mapsto \Omega(\mathcal{C}, \mathcal{P})$ *is a function that assigns invariants to locations,*
- $\rightarrow \subseteq L \times \Sigma \times \Omega(\mathcal{C}, \mathcal{P}) \times 2^{\mathcal{C}} \times L$ *is a set of transitions in the form* (l, a, Z, r, l')*, where* l *and* l' *are source and initial locations respectively,* $a \in \Sigma_\tau$ *is an action associated with the transition,* $Z \in \Omega(\mathcal{C}, \mathcal{P})$ *is a transition guard and* $r \in 2^{\mathcal{C}}$ *is a set of clocks reset with the transition.*

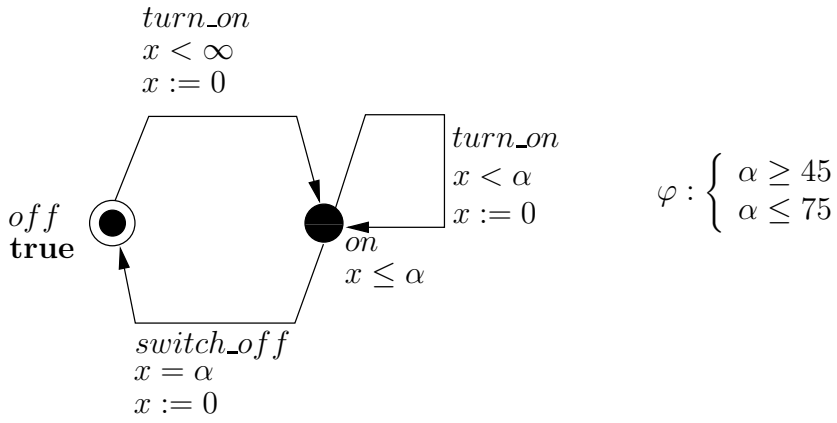


Figure 16: Example of Parametric Timed Automaton

Exemple 5.2. *Figure 16 shows an example of PTA. It is modification of the automaton from Example 3.1, so that some constants in the guards and invariants have been replaced with a parameter α. The parameter α is explicitly constrained so it can have any value between* 45 *and* 75*.*

5.3 Parametric DBM

It has been shown in [6] that the reachability problem for timed automata with parameters is undecidable. However, in [8] authors propose a semi-algorithmic approach that allows to deal with parametric timed systems. They define a new symbolic representation called Parametric DBMs (PDBMs) for use in reachability analysis, and provide a technique for computing representations of their sets of reachable configurations. In following the definition of parameterized DBM and processing methods for this structure are presented.

5.3.1 Definition of PDBM

PDBM is a symbolic data structure that is used to represent interpretation of variables during system analysis. PDBMs – in comparison to DBMs – were designed to work with parameters. PDBMs were successfully implemented in verification tool TReX [9].

Definition 14. *A Parametric Difference Bound Matrix is a square matrix of parameterized bounds where rows and columns are labelled with clocks, the first row and the first column are*

labelled with the reference clock always equal to 0. *Elements* $M_{i,j}$ *of PDBM are parametric bounds of the difference of clock labelling the row* i *and clock labelling the column* j.

$$\left(\begin{array}{c|ccc} & x_0 & x & y \\ \hline x_0 & (0,\leq) & (\alpha,<) & (0,\leq) \\ x & (\beta-\alpha+1,\leq) & (0,\leq) & (\gamma,<) \\ y & (3,\leq) & (-1,<) & (0,\leq) \end{array}\right)$$

Figure 17: An example $PDBM$

A $PDBM$ is a matrix that encodes constraints in form $x_i - x_j \prec t$ where x_i and x_j are clocks and t is an arithmetical expression with parameters. An example of $PDBM$ is presented in the Figure 17.

A Constrained $PDBM$ is a pair $\widetilde{M} = (M, \varphi)$, where M is a $PDBM$ and ϕ is conjunction of atomic constraints on parameters. The Figure 18 illustrates an example constrained $PDBM$.

$$\left(\begin{array}{c|ccc} & x_0 & x & y \\ \hline x_0 & (0,\leq) & (-\alpha,<) & (0,\leq) \\ x & (\beta-\alpha+1,\leq) & (0,\leq) & (\gamma,<) \\ y & (3,\leq) & (-1,<) & (0,\leq) \end{array}\right), \quad \varphi = \begin{cases} 0 \leq \alpha \\ \alpha \leq 9 \\ 6 \leq \beta \\ 2 \leq \gamma \\ \gamma \leq 3 \end{cases}$$

Figure 18: An example constrained $PDBM$

5.3.2 Operations on constrained PDBMs

The constrained $PDBM$s are used for symbolic state representation. Therefore it is necessary to define all operations required for symbolic reachability analysis to work with constrained $PDBM$s. This section covers definitions of the methods for property checking (inclusion, non emptiness), minimization and symbolic operations of forward and backward clock reset and time elapse.

Minimization

The canonical form for parametric DBMs is defined in the analogical way to standard (non-parameterized) DBMs. The parametric DBM is canonical if the graph it represents has a property that weight of each edge is lower than any path that connects the nodes connected by the edge. In case of PDBMs, however, since the weights of edges have form of parametric bounds it is not possible to decide the relation between weights using classical operators of sum and "less or equal" relation ($+$ and $\leq$) as it is in the case of standard DBMs. Instead the operators of $\oplus$ and $\subseteq$ are used respectively.

Formally, a constrained $PDBM$ $(\widetilde{M}, \Phi)$ is canonical if and only if following condition is satisfied:

$$\forall i, j, k \in [1, n] : \ (\widetilde{M}_{i,j}, \Phi) \subseteq (\widetilde{M}_{i,k}, \Phi) \oplus (\widetilde{M}_{k,j}, \Phi)$$

The minimization algorithm for parametric DBMs follows the same principles that the classical Floyd-Warschall algorithm for non-parametric case. 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. Because both variants are possible it may be necessary to consider both of them by splitting the DBM into two matrices, each for one variant. Since such a split may be required for each comparison operation, it is possible that the minimization algorithm will return n^3 matrices. The cost of minimization may be even $\mathcal{O}(2^{n^3})$, however usually it is not so high. For more details on the implementation issues for minimization algorithm see [51].

Exemple 5.3. *Consider the constrained PDBM $\widetilde{M}$ and its corresponding parameterized constraints graph from the Figure 19.*

The graph is not minimal, since the relations $(\widetilde{M}_{2,1},\ \phi) \subseteq (\widetilde{M}_{2,3},\ \phi) \oplus (\widetilde{M}_{3,1},\ \phi)$ and $(\widetilde{M}_{3,1},\ \phi) \subseteq (\widetilde{M}_{3,2},\ \phi) \oplus (\widetilde{M}_{2,1},\ \phi)$ are not true. Indeed, the relation $(\widetilde{M}_{2,1},\ \phi) \subseteq (\widetilde{M}_{2,3},\ \phi) \oplus (\widetilde{M}_{3,1},\ \phi)$ for $p_1 > p_2$ is false and relation $(\widetilde{M}_{3,1},\ \phi) \subseteq (\widetilde{M}_{3,2},\ \phi) \oplus (\widetilde{M}_{2,1},\ \phi)$ is false as well for $p_1 < p_2$. Therefore the relation between parameters p_1 and p_2 must be considered for constructing the canonical form of the PDBM. Consequently canonicalization process will lead to creating two PDBMs, each for one case, as it is seen in the Figure 20.

In order to check the consistency of each of the possible cases when computing the minimum

$$\widetilde{M} = \begin{pmatrix} & \mathbf{0} & \mathbf{x} & \mathbf{y} \\ \mathbf{0} & 0 & 0 & 0 \\ \mathbf{x} & p_1 & 0 & 0 \\ \mathbf{y} & p_2 & 0 & 0 \end{pmatrix}, \ \phi = \begin{cases} p_1 \geq 0 \\ p_2 \geq 0 \end{cases}$$

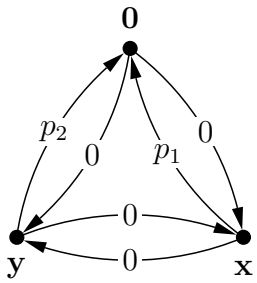


Figure 19: Constrained PDBM with corresponding parameterized constraints graph

$$\widetilde{M_1} = \begin{pmatrix} & \mathbf{0} & \mathbf{x} & \mathbf{y} \\ \mathbf{0} & (0,\leq) & (0,\leq) & (0,\leq) \\ \mathbf{x} & (p_1,\leq) & (0,\leq) & (0,\leq) \\ \mathbf{y} & (p_1,\leq) & (0,\leq) & (0,\leq) \end{pmatrix}, \ \phi = \begin{cases} 0 \leq p_2 \\ p_1 \leq p_2 \end{cases}$$

$$\widetilde{M_2} = \begin{pmatrix} & \mathbf{0} & \mathbf{x} & \mathbf{y} \\ \mathbf{0} & (0,\leq) & (0,\leq) & (0,\leq) \\ \mathbf{x} & (p_2,\leq) & (0,\leq) & (0,\leq) \\ \mathbf{y} & (p_2,\leq) & (0,\leq) & (0,\leq) \end{pmatrix}, \ \phi = \begin{cases} 0 \leq p_1 \\ p_2 \leq p_1 \end{cases}$$

Figure 20: Canonical constrained PDBMs

between two bounds, the satisfiability of formulas φ of the form

$$\phi \wedge t_1 \prec t_2$$

where $\prec \in \{<, \leq\}$ and ϕ is a conjunction of parameters constraints must be tested. If ϕ contains linear constraints or all parameters are real, then the test is decidable [8]. If ϕ is nonlinear formula mixing integer and real parameters, this test is undecidable. Nevertheless, it is possible to test the satisfiability of ϕ under the assumption that all parameters are reals. Further details are covered by [32].

Inclusion test

The inclusion test $\widetilde{M} \subseteq \widetilde{M'}$ for constrained $PDBMs$ $\widetilde{M} = (M,\ \varphi)$ and $\widetilde{M'}(M',\ \varphi')$ verifies whether all valuations that belong to polyhedron defined by $\widetilde{M}$ belong to polyhedron defined by $\widetilde{M'}$ for all possible parameters setting allowed by φ and φ'.

To decide the inclusion of $\widetilde{M}$ in $\widetilde{M}$ it is necessary to compare each pair of corresponding constraints in M and M', and to find out whether the bound defined by M is lower than the bound defined by M' for all parameters allowed by φ and φ'.

Formally:

$$\widetilde{M} \subseteq \widetilde{M'} \Leftrightarrow \forall i, j \in [1, n] :\ \left(M_{i,j},\ \varphi \wedge \varphi'\right) \subseteq \left(M'_{i,j},\ \varphi \wedge \varphi'\right)$$

where n is number of rows and columns of $\widetilde{M_1}$ and $\widetilde{M_2}$.

For inclusion test the compared PDBMs must be in the canonical form.

Intersection

The intersection for constrained parameterized bounds is already defined using operator $\otimes$. This method can be extended to matrices.

The intersection consists of computing the minimum for every i, j between two bounds $M_1(i, j)$ and $M_2(i, j)$ under the parameter constraints $\varphi_1 \wedge \varphi_2$. This is done by splitting and checking the consistency of each case, as in the construction of canonical representation. For every two bounds one or two constrained parameterized bounds may be obtained,

depending on the satisfiability of formulas $\Phi_{\leq}$ or $\Phi_{\geq}$. The result of intersection of two constrained PDBMs will be a set of constrained PDBMs as shown in the following example:

Exemple 5.4. *Let x be a clock and $\mathcal{P} = \{\alpha, \beta, \gamma, \delta\}$ be parameters. Let two control states of the transition graph be given by conjunction of constrained parameterized bounds that are represented by following PDBMs:*

$$\widetilde{M_1} = \begin{pmatrix} & \mathbf{0} & \mathbf{x} \\ \mathbf{0} & (0,\leq) & (\alpha,\leq) \\ \mathbf{x} & (\beta,\leq) & (0,\leq) \end{pmatrix}, \begin{Bmatrix} \alpha \geq 0 \\ \beta \geq 0 \end{Bmatrix} \quad \widetilde{M_1} = \begin{pmatrix} & \mathbf{0} & \mathbf{x} \\ \mathbf{0} & (0,\leq) & (\gamma,\leq) \\ \mathbf{x} & (\delta,\leq) & (0,\leq) \end{pmatrix}, \begin{Bmatrix} \gamma \geq 0 \\ \delta \geq 0 \end{Bmatrix}$$

The intersection $\widetilde{M_1} \otimes \widetilde{M_2}$ will result in four matrices:

$$\widetilde{M_1} \otimes \widetilde{M_2} = \left(\begin{pmatrix} & \mathbf{0} & \mathbf{x} \\ \mathbf{0} & (0,\leq) & (\alpha,\leq) \\ \mathbf{x} & (\beta,\leq) & (0,\leq) \end{pmatrix}, \begin{Bmatrix} \alpha \geq 0 \\ \beta \geq 0 \\ \gamma \geq 0 \\ \delta \geq 0 \\ \alpha \leq \gamma \\ \beta \leq \delta \end{Bmatrix} \right) \cup$$

$$\left(\begin{pmatrix} & \mathbf{0} & \mathbf{x} \\ \mathbf{0} & (0,\leq) & (\alpha,\leq) \\ \mathbf{x} & (\delta,\leq) & (0,\leq) \end{pmatrix}, \begin{Bmatrix} \alpha \geq 0 \\ \beta \geq 0 \\ \gamma \geq 0 \\ \delta \geq 0 \\ \alpha \leq \gamma \\ \delta \leq \beta \end{Bmatrix} \right) \cup \left(\begin{pmatrix} & \mathbf{0} & \mathbf{x} \\ \mathbf{0} & (0,\leq) & (\gamma,\leq) \\ \mathbf{x} & (\beta,\leq) & (0,\leq) \end{pmatrix}, \begin{Bmatrix} \alpha \geq 0 \\ \beta \geq 0 \\ \gamma \geq 0 \\ \delta \geq 0 \\ \gamma \leq \alpha \\ \beta \leq \delta \end{Bmatrix} \right) \cup$$

$$\left(\begin{pmatrix} & \mathbf{0} & \mathbf{x} \\ \mathbf{0} & (0,\leq) & (\gamma,\leq) \\ \mathbf{x} & (\delta,\leq) & (0,\leq) \end{pmatrix}, \begin{Bmatrix} \alpha \geq 0 \\ \beta \geq 0 \\ \gamma \geq 0 \\ \delta \geq 0 \\ \gamma \leq \alpha \\ \delta \leq \beta \end{Bmatrix} \right)$$

Intersection is based on comparing corresponding elements of two PDBMs. Whenever the result is unambiguos, the matrix is duplicated to consider two possible cases of comparison. It might be then assumed, that parameterized element in one of intersected matrices

will cause the duplication. Therefore, for simplicity it might be assumed that each single parameter used in PDBM doubles the size of the matrix. The memory consumption cost in this case is expressed by:

$$memory\_cost = 2^p \cdot n^2$$

where p is number of parameters in the system. Note, that this is optimistic scenario, because the same parameter can be used in more than one element of a PDBM.

The operational costs of intersecting two constrained PDBMs may be considered in two aspects.The first one is the number of comparisons that must be done for the operation. This value does not change comparing to intersecting two standard DBMs in the sense that all corresponding elements must be compared once (this results in n^2 comparisons). More important is the second aspect, however. This is the cost of all operations that come out from the fact that results may be unambiguous, i.e. cost of duplicating the matrix and the cost of verifying that the constraints of the parameters are solvable. Due to this fact, the operational cost of intersection may be expressed by:

$$operational\_cost = no\_of\_comparisons + no\_of\_copying + solve\_constraints$$

For the number of copying operations it might be assumed as above, that each parameter causes one duplication of the matrix. Therefore, the matrix is duplicated as many times as there are parameters in the intersected matrices. Finally, each duplication causes that there are separate constraints defined for each copy of duplicated matrices. This constraints have form of system of inequalities involving used parameters that must be solved in order to decide whether the copy of PDBM is consistent (i.e. it has solutions). Summing up all the costs, the operational cost of intersection is expressed by:

$$operational\_cost = n^2 \cdot comp + p \cdot n^2 \cdot copy + p \cdot solve\_constraints$$

where n^2 is the initial size of the PDBM, p is the number of parameters that cause ambiguous results of comparisons and $solve\_constraints$ is the average operational cost of solving single system of constraints of one copy of duplicated PDBM. $comp$ is a cost of single comparison operation and $copy$ is a cost of copying single element of PDBM.

Time elapse

The forward and backward time elapse is defined in the same way than for standard DBMs.

Elapsing time is applied by removing upper bounds of the clocks which corresponds to replacing the bounds in the first column of PDBM by $(\infty, <)$ (except the bound $M_{0,0}$ which is always equal to $(0, \leq)$). The set of constraints on parameters is not changed during the operation. As in case of normal DBM, the canonical form of PDBM is preserved.

The backward time elapse is done by replacing the bounds in the first row by $(0, \leq)$. Because the operation does not preserve the minimal form, the PDBM must be minimized again.

Clock reset

The application of the forward clock reset is done in the same way than applying forward clock reset to standard DBM – the row and column for the reset clock is replaced by the row and column for the variable v_0.

The most costly operation is the backward clock reset which requires applying intersection, minimization and finally replacing the bounds in the row representing the unreset clock by $(\infty, <)$.

5.4 Summary

The main drawback of PDBM is that parameters are constrained separately, outside of the main data structure used for constraining the allowed values of system clock. The consequence of this fact is that whenever constraints on parameters are ambiguous, the main structure must be duplicated in order to consider all cases that come out from the constraints. This drastically increases cost of the operations on a structure used for forward and backward analysis, especially minimization and intersection, where parameterized constraints must be frequently compared. Also, what has equally important meaning, the memory consumption of such a structure is unpredictable at the beginning of the simulation.

The next chapter introduces new data structure that can be used for parametric analysis

of real time systems. It was inspired by the observation that in symbolic representation, clocks and parameters have many common features. Most of all, the exact value of both are not precisely defined, instead it is constrained by set of inequalities. This means that if the format of constraints of clocks and parameters is unified, both can be constrained within the same data structure.

6 Extended Difference Bound Matrix

This chapter introduces Extended Difference Bounds Matrix (EDBM) – a new framework for processing parameterized automata. It allows to process constraints on clocks together with constraints on parameters.

The main innovation of the EDBM is that the elements of the matrix represent bounds of expressions in form $A - B$, such that A and B are defined by sums $\gamma_1 + \gamma_2 \cdots + \gamma_n$ where γ_i may be either clock or a parameter. All the elements of EDBM are numerical bounds – this allows much more economic processing than in case of PDBMs from the memory and time consumption point of view.

The EDBM as a data structure is introduced in the Section 6.1. The Section 6.2 discusses challenges and solution for finding a canonical form of a given EDBM. Other operations over EDBM that are required for symbolic analysis are covered by the Section 6.3. Finally, the section 6.4 shows how the traditional approach for symbolic forward and backward path analysis can be extended for EDBM.

Contents

6.1 Definition of Extended DBM

Let $A_i = \{\gamma_1, \cdots, \gamma_n\}$ such that $\gamma_i \in \mathcal{C} \cup \mathcal{P}$, i.e $A_i \subseteq \mathcal{C} \cup \mathcal{P}$, and let $\sum A_i = \gamma_1 + \cdots + \gamma_n$.

Definition 15. *And Extended Difference Bound Matrix (EDBM) is a square matrix, where rows columns are labeled by sets $A_i \subseteq \mathcal{C} \cup \mathcal{P}$. Elements $M_{i,j}$ of the matrix are numerical bounds $(m_{i,j}, \prec_{i,j})$ that define constraint in form $\sum A_i - \sum A_j \prec_{i,j} m_{i,j}$ such that row i and column j are labeled with sets A_i and A_j respectively.*

Notice, that EDBM defined as above can store parameterized constraints on clocks and difference of clocks together with numerical constraints on expressions with parameters. For example a constraint $x_1 - x_2 \leq \alpha$, where x_1 and x_2 are clocks and α is a parameter can be transformed to the form $x_1 - (x_2 + \alpha) \leq 0$ and stored in EDBM as element $M_{i,j} = (0, \leq)$ where i and j are indexes of row and column labelled with $\{x_1\}$ and $\{x_2, \alpha\}$ respectively.

EDBMs can be represented by *Extended Constraints Graphs* (ECG). Extended constraint graphs are extensions of classical constraints graphs, such that each node n_i is labelled by set $A_i \subseteq \mathcal{C} \cup \mathcal{P}$ and weights of edges $n_i \to n_j$ define bounds $(m, \prec)$ of expressions $\sum A_i - \sum A_j \prec m$.

By convention, rows and columns with the same index i are labeled with the same set A_i. The first row and column are labeled with $\emptyset$ (note that $\sum \emptyset = 0$), From now, $label(i)$ will denote the set labelling the row and column i. When referring to an extended constraint graph, $label(n_i)$ will denote the set labelling the node n_i. The function $expr(i, j)$ will return the expression that is bounded by element $M_{i,j}$ (i.e. $expr(i, j) = \sum label(i) - \sum label(j)$).

Exemple 6.1. *Let $x, y \in \mathcal{C}$ and $\alpha \in \mathcal{P}$. Then, let Z be a parameterized polyhedron defined by following set of constraints: $(1 \leq x \leq 5) \wedge (1 \leq y \leq 5) \wedge (x - y \leq \alpha - 4) \wedge (y - x \leq \alpha + 1)$ with $1 \leq \alpha \leq 6$. This polyhedron is represented by the constrained PDBM $\widetilde{M}$:*

$$\widetilde{M} = \left(\begin{array}{c|ccc} & \mathbf{0} & \mathbf{x} & \mathbf{y} \\ \hline \mathbf{0} & (0, \leq) & (-1, \leq) & (-1, \leq) \\ \mathbf{x} & (5, \leq) & (0, \leq) & (\alpha - 4, \leq) \\ \mathbf{y} & (5, \leq) & (\alpha + 1, \leq) & (0, \leq) \end{array} \right), \left\{ \begin{array}{lcl} \alpha & \geq & 1 \\ \alpha & \leq & 6 \end{array} \right.$$

The polyhedron for $\alpha = 2$ is depicted in the Figure 21.

Note that the constraint $x - y \leq \alpha - 4$ can be expressed as $x - (y + \alpha) \leq -4$. Also the

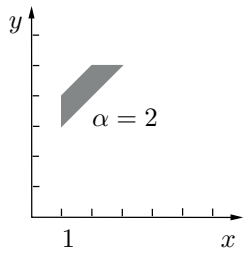


Figure 21: A parameterized polyhedron

constraint $y - x \leq \alpha + 1$ *can be expressed as* $y - (x + \alpha) \leq 1$. *Storing constraints defining* Z *in a EDBM will result in a matrix* M *presented below:*

$$M = \left(\begin{array}{c|cccccc} & \emptyset & \{\mathbf{x}\} & \{\mathbf{y}\} & \{\alpha\} & \{\mathbf{x},\alpha\} & \{\mathbf{y},\alpha\} \\ \hline \emptyset & (0,\leq) & (-1,\leq) & (-1,\leq) & (-1,\leq) & (\infty,<) & (\infty,<) \\ \{\mathbf{x}\} & (5,\leq) & (0,\leq) & (\infty,<) & (\infty,<) & (\infty,<) & (-4,\leq) \\ \{\mathbf{y}\} & (5,\leq) & (\infty,<) & (0,\leq) & (\infty,<) & (1,\leq) & (\infty,<) \\ \{\alpha\} & (6,\leq) & (\infty,<) & (\infty,<) & (0,\leq) & (\infty,<) & (\infty,<) \\ \{\mathbf{x},\alpha\} & (\infty,<) & (\infty,<) & (\infty,<) & (\infty,<) & (0,\leq) & (\infty,<) \\ \{\mathbf{y},\alpha\} & (\infty,<) & (\infty,<) & (\infty,<) & (\infty,<) & (\infty,<) & (0,\leq) \end{array}\right)$$

Z may be also represented by extended constraint graph G, presented in the Figure 22. For better view, only the edges with weight different than $(\infty,<)$ were labelled. The other edges are colored gray to keep the figure readable.

The polyhedron represented by EDBM M may be shown in three dimensions, as in the Figure 23. Intersecting the polyhedron with a plane parallel to the plane xy will return a 2 dimensional polyhedron containing values of x and y allowed for given value of α. For example intersecting this polyhedron with a plane parallel to xy that intersects the α-axis at the point $\alpha = 2$ will give the polyhedron from the Figure 21.

6.1.1 Equivalent elements and equivalence classes

It is possible that two elements of an EDBM are redundant in the sense that they define a bound of the same expression. For example in matrix M from example 6.1 both entries $M_{6,3}$ $(expr(6,3) = (y + \alpha) - y)$ and $M_{4,1}$ $(expr(4,1) = \alpha - 0)$ represent bound of α. Such

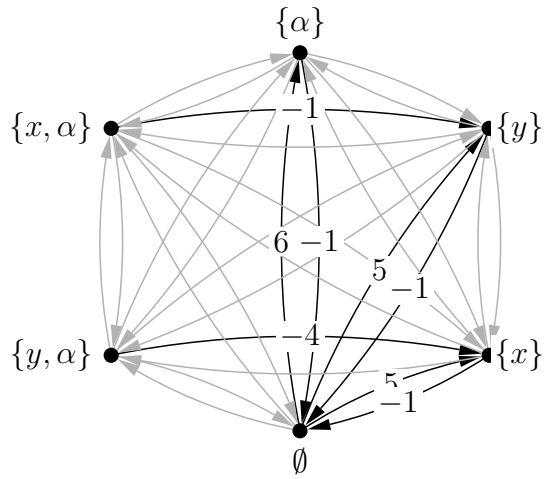


Figure 22: An extended constraint graph

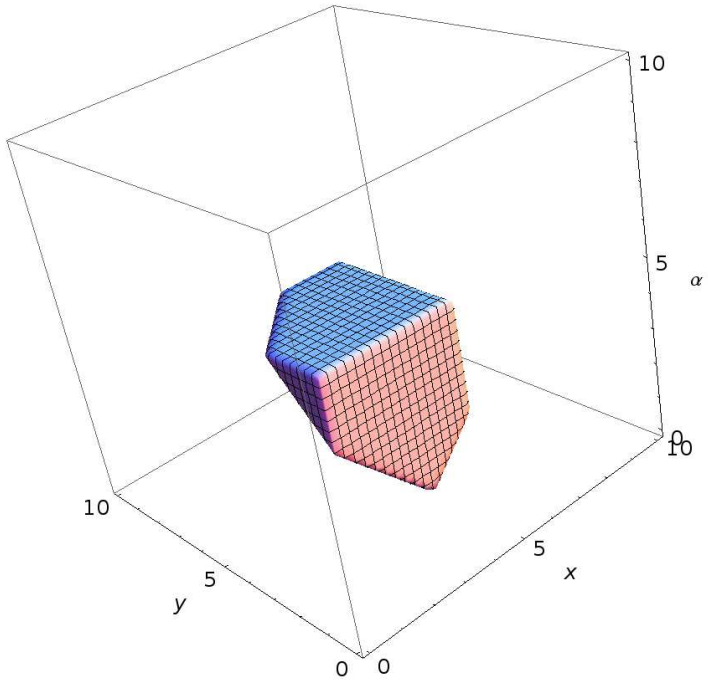


Figure 23: Polyhedron represented by M

elements will be called *equivalent*:

Definition 16. *Two elements $M_{i,j}$ and $M_{m,n}$ of an EDBM M are equivalent if and only if $expr(i, j) = expr(m, n)$.*

Set of equivalent elements will be called *equivalence class*. The function $equivalent(M_{i,j})$ will return the equivalence class to which the element $M_{i,j}$ belongs. The function $classes(M)$ will return all equivalence classes of the matrix M. The corresponding functions are also defined for extended constraint graphs.

A ECG corresponding to matrix M with edges belonging to the same equivalence class marked with distinct color is depicted in the Figure 24. For better view weights of the edges are not marked.

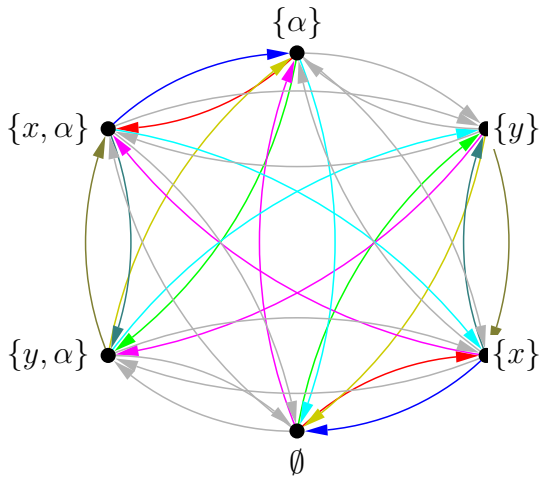


Figure 24: Graph with equivalent edges (grey edges are not equivalent with any other edges)

It is desirable that all the equivalent elements have the same value. A matrix, where all equivalent elements have the same value will be called *consistent*. To make the EDBM consistent without changing the content of polyhedron represented by the matrix, value of all elements within an equivalence class must be changed to the lowest value within the class. The Algorithm 6.1 define function $consistent(M)$ that brings the matrix M to the consistent form:

The consisten version of matrix M from example 6.1 has following form:

Algorithm 6.1 *consistent*(M): bringing EDBM to a consistent form

Arguments: EDBM M

Output: consistent version of M

```
for all ϵ ∈ classes(M) do
  min = (∞, <);
  for all M_{i,j} ∈ ϵ do
    if M_{i,j} < min then
      min = M_{i,j};
    end if
  end for
  for all M_{i,j} ∈ ϵ do
    M_{i,j} = min;
  end for
end for
return M
```

$$consistent(M) = \left(\begin{array}{c|cccccc} & \emptyset & \{\mathbf{x}\} & \{\mathbf{y}\} & \{\alpha\} & \{\mathbf{x},\alpha\} & \{\mathbf{y},\alpha\} \\ \hline \emptyset & (0,\leq) & (-1,\leq) & (-1,\leq) & (-1,\leq) & (\infty,<) & (\infty,<) \\ \{\mathbf{x}\} & (5,\leq) & (0,\leq) & (\infty,<) & (\infty,<) & (\infty,<) & (-4,\leq) \\ \{\mathbf{y}\} & (5,\leq) & (\infty,<) & (0,\leq) & (\infty,<) & (1,\leq) & (-1,\leq) \\ \{\alpha\} & (6,\leq) & (\infty,<) & (\infty,<) & (0,\leq) & (-1,\leq) & (-1,\leq) \\ \{\mathbf{x},\alpha\} & (\infty,<) & (6,\leq) & (\infty,<) & (5,\leq) & (0,\leq) & (\infty,<) \\ \{\mathbf{y},\alpha\} & (\infty,<) & (\infty,<) & (6,\leq) & (5,\leq) & (\infty,<) & (0,\leq) \end{array}\right)$$

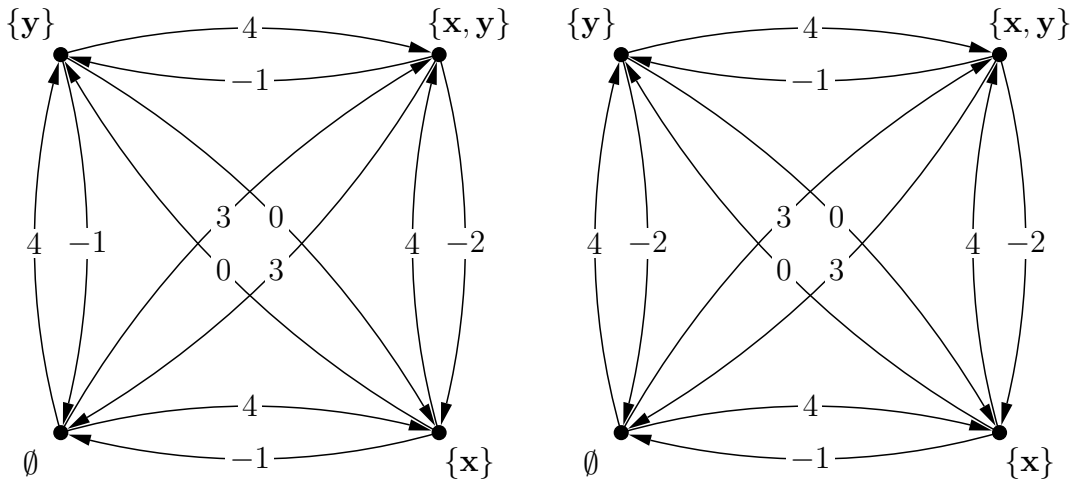
The operation *consistent*(M) does not necessarily preserve the minimal form of the matrix.

Note that because of existence of equivalent edges in extended constraints graphs it is possible that a positive graph that is not consistent may become negative when the graph is made consistent.

Corollary 5. *A Polyhedron Z represented by extended constraints graph G is not empty if and only if consistent form of G is positive.*

Exemple 6.2. *Consider the graph G from the Figure 25(a). The graph is positive, however*

it is not consistent, since the edges $\emptyset \to \{\mathbf{y}\}$ *and* $\{\mathbf{x}\} \to \{\mathbf{x}, \mathbf{y}\}$ *although equivalent have different weights. The consistent version of the graph is presented in the Figure 25(b). In this case the graph is not positive, because for example the weight of the cycle* $\{\mathbf{y}\} \to \emptyset \to \{\mathbf{x}, \mathbf{y}\} \to \{\mathbf{x}\} \to \{\mathbf{y}\}$ *is negative.*



(a) G (b) *consistent*(G)

Figure 25: Extended constraint graphs

6.2 Canonicalization of EDBM

Bringing EDBM to a canonical form is the most important and also most complicated operation on EDBMs. One of the biggest issues is that a minimal EDBM does not necessarily define a canonical form of a polyhedron. As an example, consider the matrix M from the Figure 26. Although M is minimal and consistent, it does not define canonical form of a polyhedron. Indeed, some constraints may still be tightened without changing the set of valuations defined by M. Note that the constraints $x - (y + \alpha) \leq -4$ and $y - (x + \alpha) \leq 1$ implicitly define constraint $-2\alpha \leq -3$. This means that the value of $M_{1,4}$ can be changed to -1.5 without consequences for the shape of Z.

The reason, why the EDBM M did not define canonical form of a polyhedron is that there is no path in the extended constraint graph corresponding to M that would consider the

$$M = \left(\begin{array}{c|cccccc} & \emptyset & \{\mathbf{x}\} & \{\mathbf{y}\} & \{\alpha\} & \{\mathbf{x},\alpha\} & \{\mathbf{y},\alpha\} \\ \hline \emptyset & (0,\leq) & (-1,\leq) & (-1,\leq) & (-1,\leq) & (-2,\leq) & (-5,\leq) \\ \{\mathbf{x}\} & (5,\leq) & (0,\leq) & (2,\leq) & (1,\leq) & (-1,\leq) & (-4,\leq) \\ \{\mathbf{y}\} & (5,\leq) & (4,\leq) & (0,\leq) & (4,\leq) & (1,\leq) & (-1,\leq) \\ \{\alpha\} & (6,\leq) & (5,\leq) & (5,\leq) & (0,\leq) & (-1,\leq) & (-1,\leq) \\ \{\mathbf{x},\alpha\} & (11,\leq) & (6,\leq) & (8,\leq) & (5,\leq) & (0,\leq) & (2,\leq) \\ \{\mathbf{y},\alpha\} & (11,\leq) & (10,\leq) & (6,\leq) & (5,\leq) & (4,\leq) & (0,\leq) \end{array}\right)$$

Figure 26: Example of minimal EDBM that does not define canonical form of a polyhedron

sum of bounds of $x-(y+\alpha)$ and $y-(x+\alpha)$. Therefore after minimization the bound of $-\alpha$ will not considered the sum of the bounds of $x-(y+\alpha)$ and $y-(x+\alpha)$.

6.2.1 Linear DBM

To calculate canonical form of a EDBM it is necessary to introduce a new structure: Linear DBM (LDBM). LDBM is generalization of EDBM such that rows and columns are labeled with multisets $A_i = \{\gamma_1, \cdots, \gamma_1, \cdots, \gamma_n, \cdots, \gamma_n\} = \{k_1\gamma_1, \cdots, k_n\gamma_n\}$ such that $k_i \in \mathbb{N}$ and $\gamma_i \in \mathcal{C} \cup \mathcal{P}$. Similarly to EDBMs, $\sum A_i = k_1\gamma_1 + \cdots + k_n\gamma_n$.

Definition 17. *Linear Difference Bound Matrix is a square matrix where ith row and ith column are labelled with multiset $A_i = \{k_1\gamma_1, \cdots, k_n\gamma_n\}$. The elements $M_{i,j}$ are numerical bounds of expressions $\sum A_i - \sum A_j$, such that A_i and A_j are multisets labelling row and column i and j respectively.*

LDBM allow to store bounds of expressions that can be any linear combination of clocks and parameters with integer coefficients. The graphical representation of LDBM is *linear constraint graph* (LCG).

Consistency of LDBM

Because elements of LDBM represent bounds of expressions where coefficient are not limited to -1 or 1, the definition of equivalence classes for LDBM must be extended to cover

new type of equivalence.

For example consider LDBM M with rows and columns labelled with $\emptyset$, $\{x\}$, $\{y\}$, $\{2x\}$ and $\{2y\}$:

$$M = \left(\begin{array}{c|ccccc} & \emptyset & \{x\} & \{y\} & \{2x\} & \{2y\} \\ \hline \emptyset & & & & & \\ \{x\} & & & M_{2,3} & & \\ \{y\} & & & & & \\ \{2x\} & & & & & M_{4,5} \\ \{2y\} & & & & & \end{array}\right)$$

The elements $M_{4,5}$ and $M_{2,3}$ represent bound of the same expression, but in case of $M_{4,5}$ multiplied by 2. The definition of consistency and equivalence for LDBMs must consider such cases. In general two elements of a LDBM are equivalent if they represent bound of the same expression multiplied with a positive constant.

To formally define equivalence for LDBM it is necessary to introduce operation $basis()$. This operation for linear expression $expr(i,j)$ returns linear expression $expr' = \frac{expr(i,j)}{\varphi}$ where $\varphi \in \mathbb{N}^+$ is the biggest natural such that coefficients of $expr'$ are integers. The value of φ will be called *equivalence factor* of $M_{i,j}$ noted by $\varphi(i,j)$. The matrix of equivalence factors for elements of the EDBM M above is following:

$$\varphi(M) = \left(\begin{array}{c|ccccc} & \emptyset & \{x\} & \{y\} & \{2x\} & \{2y\} \\ \hline \emptyset & 1 & 1 & 1 & 2 & 2 \\ \{x\} & 1 & 1 & 1 & 1 & 1 \\ \{y\} & 1 & 1 & 1 & 1 & 1 \\ \{2x\} & 2 & 1 & 1 & 1 & 2 \\ \{2y\} & 2 & 1 & 1 & 2 & 1 \end{array}\right)$$

With function $basis()$ the definition of equivalent elements of LDBM looks as follows:

Definition 18. *Two elements $M_{i,j}$ and $M_{k,l}$ of LDBM M are equivalent if and only if*

$$basis\big(expr(i,j)\big) = basis\big(expr(k,l)\big)$$

The basis of an equivalence class is the basis of its elements.

A LDBM is said to be consistent only if all elements that are in the same equivalence class have the same value with respect to their equivalence factors, i.e values of all elements in equivalence class divided by their equivalence factor are the same.

6.2.2 Closure of EDBM

To calculate a canonical form of a polyhedron Z represented by EDBM M it is necessary to construct such a LDBM M' that the LCG corresponding to M' will contain all paths that represent sum of constraints defined in M that can impact the canonical form of Z. In other words if two or more elements of M implicitly impact value of some other element of M they must be considered as a path in LCG corresponding to M'.

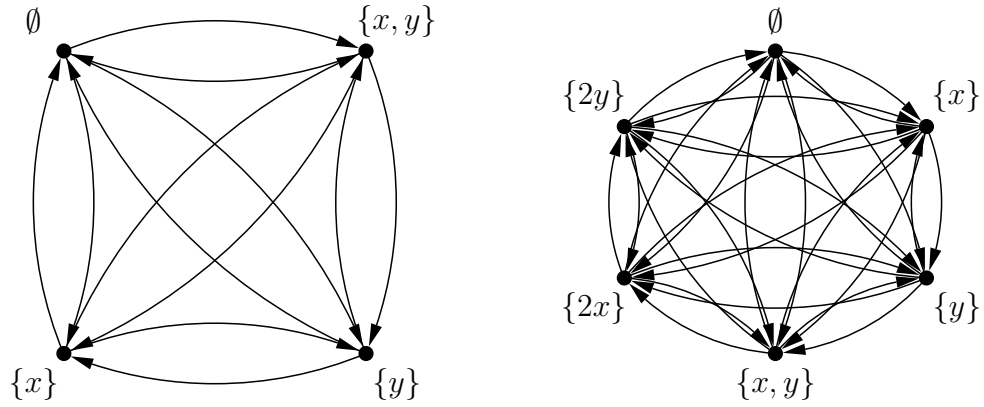
As an example consider again the matrix M from the figure 26:

$$M = \left(\begin{array}{c|cccccc} & \emptyset & \{\mathbf{x}\} & \{\mathbf{y}\} & \{\alpha\} & \{\mathbf{x},\alpha\} & \{\mathbf{y},\alpha\} \\ \hline \emptyset & (0,\leq) & (-1,\leq) & (-1,\leq) & (-1,\leq) & (-2,\leq) & (-5,\leq) \\ \{\mathbf{x}\} & (5,\leq) & (0,\leq) & (2,\leq) & (1,\leq) & (-1,\leq) & (-4,\leq) \\ \{\mathbf{y}\} & (5,\leq) & (4,\leq) & (0,\leq) & (4,\leq) & (1,\leq) & (-1,\leq) \\ \{\alpha\} & (6,\leq) & (5,\leq) & (5,\leq) & (0,\leq) & (-1,\leq) & (-1,\leq) \\ \{\mathbf{x},\alpha\} & (11,\leq) & (6,\leq) & (8,\leq) & (5,\leq) & (0,\leq) & (2,\leq) \\ \{\mathbf{y},\alpha\} & (11,\leq) & (10,\leq) & (6,\leq) & (5,\leq) & (4,\leq) & (0,\leq) \end{array}\right)$$

The elements $M_{2,6}$ and $M_{3,5}$ define constraints $x-(y+\alpha) \leq -4$ and $y-(x+\alpha) \leq 1$ respectively. Summing up those constraints will return the constraint $-2\alpha \leq 4 \equiv -\alpha \leq -1.5$, therefore the sum of elements $M_{2,6}$ and $M_{3,5}$ impact the canonical form of polyhedron represented by M. There is, however, no path in the ECG G corresponding to M that would consist of two edges representing bounds of expressions $x-(y+\alpha)$ and $y-(x+\alpha)$. Such a path can be enabled by adding a node labelled with $\{2\alpha, x\}$. Then the edges $\{2\alpha, x\} \rightarrow \{y, \alpha\}$ and $\{x, \alpha\} \rightarrow \{y\}$ would be equivalent. Since the edges $\{2\alpha, x\} \rightarrow \{x\}$ and $\{\alpha\} \rightarrow \emptyset$ are also equivalent, weight of the path $\{2\alpha, x\} \rightarrow \{y, \alpha\} \rightarrow \{x\}$ can be considered to calculate bound of $-\alpha$ during minimization.

A LDBM (or LCG) that can be used to calculate canonical form of polyhedron represented by EDBM M (or ECG G) will be called *closure* of M (or G).

Let ψ be a set of edges of ECG and $expr(\psi)$ return an expression obtained by adding all



(a) G (b) G'

Figure 27: An extended constraint graph and its closure

expressions bounded by edges of ψ. Formally the closure for extended constraint graphs is defined as follows:

Definition 19. *Let $G = (N, \omega, E)$ be an extended constraint graph and $G' = (N', \omega', E')$ be linear constraint graph. G' is said to be closure of G if and only if for any set of edges $\psi \in 2^E$ such that $\exists e \in E \mid basis\big(expr(\psi)\big) = expr(e)$ there is a path $p \in paths(G')$ that consists of edges equivalent to edges in ψ.*

Exemple 6.3. *The Figure 27 shows two graphs: extended constraint graph $G = (N, \omega, E)$ and linear constraint graph $G' = (N', \omega', E')$. In the power set 2^E there are four combinations of edges that can define bound of such an expression that its basis is represented by some edge in E:*

- $\psi_1 = \big\{\{y\} \to \{x\}, \emptyset \to \{x, y\}\big\}$ *that define bound of $2x$ (basis of $2x$ is represented by edge $\emptyset \to \{x\}$),*
- $\psi_2 = \big\{\{x\} \to \{y\}, \emptyset \to \{x, y\}\big\}$ *that define bound of $2y$ (basis of $2y$ is represented by edge $\emptyset \to y$),*
- $\psi_3 = \big\{\{y\} \to \{x\}, \{x, y\} \to \emptyset)\big\}$ *that define bound of $-2y$ (basis of $-2y$ is represented by edge $\{y\} \to \emptyset$) and*
- $\psi_4 = \big\{\{x\} \to \{y\}, \{x, y\} \to \emptyset)\big\}$ *that define bound of $-2x$ (basis of $-2x$ is represented by edge $\{x\} \to \emptyset$).*

Note that all of those dependencies may be represented by paths of G':

- $\emptyset \to \{x, y\} \to \{2x\}$ *represent the constraint defined by* $\{\{y\} \to \{x\}, \emptyset \to (x + y)\}$,
- $\emptyset \to \{x, y\} \to \{2y\}$ *represent the constraint defined by* $\{\{x\} \to \{y\}, \emptyset \to \{x, y\}\}$,
- $\{2x\} \to \{x, y\} \to \emptyset$ *represent the constraint defined by* $\{\{x\} \to \{y\}, \{x, y\} \to \emptyset\}$ *and*
- $\{2y\} \to \{x, y\} \to \emptyset$ *represent the constraint defined by* $\{\{y\} \to \{x\}, \{x, y\} \to \emptyset\}$.

Therefore G' *is closure of* G.

To obtain a LCG G' that is closure of ECG $G = (N, \omega, E)$ it is necessary to follow following steps:

1. Find all combinations of edges $\psi \in 2^E$ such that $expr(\psi)$ is an expression equivalent to expression bounded by some edge $e \in E$, i.e. $basis\big(expr(\psi)\big) = expr(e)$.
2. Define paths that represent identified combinations,
3. Add new nodes to G, so that construction of the paths identified above is possible,
4. Connect the nodes with the rest of the graph and make the resulting graph consistent.

Combinations of edges that can impact canonical form

In general to find all combinations of constraints represented by edges of $G = (N, \omega, E)$ that can determine the canonical form of polyhedron represented by G, all combinations $\psi \in 2^E$ need to be checked. For a fully connected graph with n nodes, there are $n(n-1)$ edges, so the set 2^E has $2^{n(n-1)}$ elements that must be considered.

Checking all $2^{n(n-1)}$ combinations from the set 2^E is very costly operation. The cost may be reduced by removing some combinations without checking. Those combinations are:

1. Combinations that contain edges that create a path. For example instead checking combination $\{\emptyset \to \{x, \alpha\}, \{\alpha\} \to \{y\}, \{y\} \to \{x\}\}$ it is enough to check combination $\{\emptyset \to \{x, \alpha\}, \{\alpha\} \to \{x\}\}$, because combinations $\big\{\{\alpha\} \to \{y\}, \{y\} \to \{x\}\big\}$ and $\big\{\{\alpha\} \to \{x\}\big\}$ define bound of the same expression.
2. Combinations that are union of two or more combinations that are already confirmed to potentially impact canonical form. For example if ψ_1 and ψ_2 are already selected combinations, the combination $\{\psi_1, \psi_2\}$ is already represented in the graph by edges representing $basis\big(expr(\psi_1)\big)$ and $basis\big(expr(\psi_2)\big)$. This in fact means that

after confirming that any combination ψ must be considered in the LCG, no other combinations that contain all edges of ψ need to be checked.

As an example consider combination of edges $\psi = \{\emptyset \to \{x, y\}, \{x\} \to \{y\}, \emptyset \to \{y, \alpha\}, \{y\} \to \{\alpha\}\}$. Assume that the constraint graph contains also an edge $\emptyset \to \{x, \alpha\}$. Theoretically the combination ψ potentially defines bound of $2x + 2\alpha$ which impact the weight of edge $\emptyset \to \{x, \alpha\}$, however the combinations $\{\emptyset \to \{x, y\}, \{x\} \to \{y\}\}$ and $\{\emptyset \to \{y, \alpha\}, \{y\} \to \{\alpha\}\}$ separately define bounds of $2x$ and 2α, so they impact weights of edges $\emptyset \to \{x\}$ and $\{x\} \to \{x, \alpha\}$ respectively, that create a path from $\emptyset$ to $\{x, \alpha\}$. Therefore the combination $\{\emptyset \to \{x, y\}, \{x\} \to \{y\}, \emptyset \to \{y, \alpha\}, \{y\} \to \{\alpha\}\}$ is already considered and it does not have to be checked explicitly.

3. Combinations that define bound on expression, such that its basis is represented by an edge in the combination. For example the combination $\{\{\emptyset \to \{x, y\}, \{y\} \to \{x\}\}, \emptyset \to \{x\}$ define bound of $3x$, however it contains already an edge $\emptyset \to \{x\}$ that defines bound of x, so it must not be considered.
4. Combinations that consist of edges opposite to edges of already checked combination.
5. Combinations that consist of edges equivalent to edges in already checked combination.

The function *combinations*(G) defined by the Algorithm 6.2 returns all set of combinations of constraints defined by edges of ECG G that may affect the canonical form of polyhedron represented by G.

The working of the Algorithm 6.2 is as follows. The input of the algorithm is the set E of edges of an extended constraint graph G. Initially the set S is defined by E and the set Ψ is empty. In each iteration of the **while** loop, the algorithm checks the combinations of $E \times S$ excluding those combinations whose subsets are already in Ψ or contain edges that create paths. If the basis of an expression represented by the combination is represented also by some other edge in E, the combination is added to the set Ψ that is returned at the end. Otherwise the combination is added to the set S' and will be used in the next iteration by combining with other edges of E. At the end of each iteration S' replaces S, so in the next step elements of $E \times S$ have incremented length. In the first iteration all possible combinations of two edges are checked, then combination of 3 edges and so on.

Exemple 6.4. *Let $G = (N, \omega, E)$ represent EDBM M from the Figure 26. The nodes of*

Algorithm 6.2 *combinations*(G)

Input: $G = (N, \omega, E)$ - Extended constraint graph that represent polyhedron Z

Output: Ψ - set of combinations of elements of E that can affect the canonical form of Z

$S = E$;
$\Psi = \emptyset$;
Restrict the set E such that it contains only one edge from each equivalence class;
while $S \neq \emptyset$ **do**
  $S' = \emptyset$;
  for all $e \in E$ **do**
    for all $s \in S$ **do**
      if $(e \in s) \vee \Big(\exists e' \in S \mid \big(dst(e) = src(e') \vee dst(e') = src(e)\big)\Big)$ **then**
        continue;
      end if
      $s = s \cup e$;
      if $(\exists \psi \in \Psi \mid \psi \in 2^s)$ **then**
        continue;
      end if
      if $\exists e \in E \mid basis\big(expr(s)\big) = expr(e)$ **then**
        $\Psi = \Psi \cup s$;
      else
        $S' = S' \cup s$;
      end if
    end for
  end for
  $S = S'$;
end while
return Ψ

G are labelled with: $\emptyset$, $\{x\}$, $\{y\}$, $\{\alpha\}$, $\{x,\alpha\}$ and $\{y,\alpha\}$. The set E of edges that is input for Algorithm 6.2 is defined by (some edges equivalent to those in E are omitted):

$E = \big\{\emptyset \to \{x\}, \{x\} \to \emptyset, \emptyset \to \{y\}, \{y\} \to \emptyset, \emptyset \to \{\alpha\}, \{\alpha\} \to \emptyset, \emptyset \to \{x,\alpha\}, \{x,\alpha\} \to \emptyset, \emptyset \to \{y,\alpha\}, \{y,\alpha\} \to \emptyset, \{x\} \to \{y\}, \{y\} \to \{x\}, \{x\} \to \{y,\alpha\}, \{y,\alpha\} \to \{x\}, \{y\} \to \{x,\alpha\}, \{x,\alpha\} \to \{y\}\big\}$

The set Ψ returned by function combinations(G) is presented in the Table 1.

$\psi \in combinations(G)$	$expr(\psi)$	$e \in E \mid expr(e) = basis\big(expr(\psi)\big)$
$\big\{\{\alpha\} \to \{x\}, \emptyset \to \{x,\alpha\}\big\}$	$2x$	$\emptyset \to \{x\}$
$\big\{\{\alpha\} \to \{y\}, \emptyset \to \{y,\alpha\}\big\}$	$2y$	$\emptyset \to \{y\}$
$\big\{\{x\} \to \{\alpha\}, \emptyset \to \{x,\alpha\}\big\}$	2α	$\emptyset \to \{\alpha\}$
$\big\{\{y\} \to \{\alpha\}, \emptyset \to \{y,\alpha\}\big\}$	2α	$\emptyset \to \{\alpha\}$
$\big\{\{y\} \to \{x,\alpha\}, \{x\} \to \{y,\alpha\}\big\}$	2α	$\emptyset \to \{\alpha\}$
$\big\{\emptyset \to \{x\}, \{y\} \to \{x,\alpha\}, \emptyset \to \{y,\alpha\}\big\}$	$2x + 2\alpha$	$\emptyset \to \{x,\alpha\}$
$\big\{\{y\} \to \{x\}, \{y\} \to \{\alpha\}, \emptyset \to \{x,\alpha\}\big\}$	$2x + 2\alpha - 2y$	$\{y\} \to \{x,\alpha\}$
$\big\{\emptyset \to \{y\}, \emptyset \to \{x,\alpha\}, \{x\} \to \{y,\alpha\}\big\}$	$2y + 2\alpha$	$\emptyset \to \{y,\alpha\}$
$\big\{\{x\} \to \{y\}, \{x\} \to \{\alpha\}, \emptyset \to \{y,\alpha\}\big\}$	$2y + 2\alpha - 2x$	$\{x\} \to \{y,\alpha\}$
$\big\{\{x\} \to \{\alpha\}, \{y\} \to \{x,\alpha\}, \emptyset \to \{y,\alpha\}\big\}$	3α	$\emptyset \to \{\alpha\}$
$\big\{\{y\} \to \{\alpha\}, \{x\} \to \{y,\alpha\}, \emptyset \to \{x,\alpha\}\big\}$	3α	$\emptyset \to \{\alpha\}$
All combinations containing inverse edges		

Table 1: Result of the function $\Psi = combinations(E)$

Identifying path for given combination of edges

To create a linear constraint graph that contains paths of edges equivalent to edges in combinations discovered by function *combinations*(G), it is necessary to add some additional nodes to the graph G, so that the paths can be constructed. For example, let $\psi = \big\{\{x\} \to \{\alpha\}, \{y\} \to \{x,\alpha\}, \emptyset \to \{y,\alpha\}\big\}$. Because $expr(\psi) = 3\alpha$, ψ can determine bound of α. To consider ψ in the graph, nodes representing for example $\{2\alpha, x\}$ and $\{3\alpha\}$ can be added to the graph G. Then the path $\emptyset \to \{y,\alpha\} \to \{2\alpha, x\} \to \{3\alpha\}$ consist of edges equivalent to edges in ψ and can determine weight of the edge $\emptyset \to \{3\alpha\}$ that is equivalent to edge $\emptyset \to \{\alpha\}$.

In general, the same combination of edges may be considered in the graph by adding different set of nodes. For example the following combination of edges: $\{\{x\} \to \{\alpha\}, \{y\} \to \{x, \alpha\}, \emptyset \to \{y, \alpha\}\}$ may be represented by adding nodes representing $\{2\alpha, x\}$ and $\{3\alpha\}$, but also nodes for $\{2\alpha, y\}$ and $\{3\alpha, x\}$. In the latter case, the edges in ψ will be represented by the path $\{x\} \to \{\alpha\} \to \{2\alpha, y\} \to \{3\alpha, x\}$.

The actual set of nodes that are added to the graph to represent given combination depends on the sequence of the edges in combination. Let $\overline{\psi} = [A_1 \to B_2, \cdots, A_n \to B_n]$ be a permutation of edges of $\psi \in \mathit{combinations}(G)$, such that A_i and B_i are sets labelling nodes of G. The Algorithm 6.3 defines function $\mathit{nodes}(\overline{\psi})$ that returns set of nodes necessary to construct a path of edges equivalent to edges in ψ in order given by $\overline{\psi}$.

Algorithm 6.3 *nodes(p)*

Input: $\overline{\psi} = [A_1 \to B_1, \cdots, A_n \to B_n]$ - sequence of edges that does not contain a path
Output: set of nodes in path that consists of edges equivalent to those in ψ

```
n_0 = A_1;
for all i ∈ [1..length(ψ̄)] do
  if A_i ⊈ n_{i-1} ⊎ B_i then
    STOP HERE - this permutation cannot be represented as a path;
    return
  else
    n_i = (n_{i-1} ⊎ B_i) \ A_i;
  end if
end for
return {n_0, ..., n_n};
```

Not every sequence can be represented by a path. As example consider the following sequence: $[\emptyset \to \{x\}, \{\alpha\} \to \{x, y\}, \emptyset \to \{y, \alpha\}]$. This sequence defines bound of $2x + 2y$ that potentially impacts the weight of the edge $\emptyset \to \{x, y\}$. It is not possible to construct a path that contains edges equivalent to edges in this sequence with given order. Instead, sequence $[\emptyset \to \{y, \alpha\}, \emptyset \to \{x\}, \{\alpha\} \to \{x, y\}]$ of the same set of edges may be represented as a path $\emptyset \to \{y, \alpha\} \to \{x, y, \alpha\} \to \{2x, 2y\}$.

Updating LCG with new nodes

In general it is desirable that a LCG contains minimal number of nodes, so that its minimization has the lowest possible complexity. Therefore when constructing an LCG that is closure to some ECG G the biggest possible number of nodes should be reused. To illustrate this, consider following sets of edges: $\{\emptyset \rightarrow \{\alpha, x\}, \{x\} \rightarrow \{\alpha\}\}$ and $\{\emptyset \rightarrow \{y, \alpha\}, \{y\} \rightarrow \{\alpha\}\}$. The both sets potentially define bound of 2α. The first set of edges may be represented by either of the following two paths:

1. $\emptyset \rightarrow \{x, \alpha\} \rightarrow \{2\alpha\}$
2. $\{x\} \rightarrow \{\alpha\} \rightarrow \{2\alpha, x\}$

while the second by any of those:

1. $\emptyset \rightarrow \{y, \alpha\} \rightarrow \{2\alpha\}$
2. $\{y\} \rightarrow \{\alpha\} \rightarrow \{2\alpha, y\}$

Adding node labelled with $\{2\alpha\}$ enables creating paths consisting of edges equivalent to both sets: $\{\emptyset \rightarrow \{\alpha, x\}, \{x\} \rightarrow \{\alpha\}\}$ and $\{\emptyset \rightarrow \{y, \alpha\}, \{y\} \rightarrow \{\alpha\}\}$. If instead, for the first set of edges the second path is chosen, it results with adding to the constraint graph node labelled with $\{2\alpha, x\}$. This node cannot be reused for creating a path that corresponds to the second set of edges – it is necessary to add another node labelled either with $\{2\alpha\}$ or $\{2\alpha, y\}$.

In general, the sequences of edges for creating paths should be chosen in the way that they:

1. reuse as many existing nodes as possible,
2. when they add nodes to the graph, the nodes should have biggest chance to be reused by other sequences.

The first condition can be fulfilled easily. It is enough for each set of edges to choose such permutation that adds the lowest number of nodes to the graph. It is more complicated however, to predict which of added nodes will be most usefull for the other sequences. In general, a node can be used by more sequences, if its label involves less variables. In the example above the node that involved only α was used by two sequences: $\emptyset \rightarrow \{x, \alpha\} \rightarrow \{2\alpha\}$ and $\emptyset \rightarrow \{y, \alpha\} \rightarrow \{2\alpha\}$, when nodes that involved two variables could be used for representing only one sequence.

The Algorithm 6.4 defines function *closure*() which returns a LCG that is closure of an ECG G provided as argument, such that LCG has minimal possible number of nodes. It uses the function *permutations*(ψ) which returns all possible sequences that are permutations of elements in ψ and the function *elements*(n) that returns number of distinct elements used in the label of the node n (the cardinality of the underlying set of elements of the label of n).

For each set of combinations of edges returned by function *combinations*(G) the algorithm calculates all possible sequences (permutations of edges in the combination). Each sequence defines set of nodes that must be used for its construction. The algorithm promotes those sequences that adds lowest possible number of nodes (by incrementing $score_1$ with every node that is reused) and those that add nodes that involve the least possible number of variables (by using $score_2$ that is number of variables in nodes' labells). The nodes added to the graph are connected with all other nodes. At the end the graph is made consistent.

Exemple 6.5. *Consider again the matrix M from Figure 26 and its corresponding ECG G. Result of the function combinations(G) was presented in the Table 1. The LCG obtained by function closure(G), has following set of nodes:*

$N' = \{\{0\}, \{x\}, \{y\}, \{\alpha\}, \{x, \alpha\}, \{y, \alpha\}, \{2x\}, \{2y\}, \{2\alpha\}, \{y, 2\alpha\}, \{x, y, \alpha\}, \{2x, 2\alpha\}, \{2y, 2\alpha\}, \{y, 3\alpha\}, \{3\alpha\}\}$

The Table 2 shows paths that can be used to consider each combination of edges from the Table1.

6.2.3 Minimization of LDBM

Minimization of LCG that is closure of ECG G defines constraints of the canonical form of polyhedron represented by G.

In case of classical constraint graphs, the canonical form of a represented polyhedron was obtained by minimization using the Floyd-Warschall algorithm. The canonical form of a polyhedron that is represented by ECG can be obtained by minimizing its closure. Due to existence of equivalent edges in LCG the classical Floyd-Warschall shortest path algorithm will not work. The reason for this is that bringing a LCG to a minimal form with Floyd-Warschall algorithm will not preserve its consistency. Then, making the graph consistent may not preserve the minimal form, as shown in the Example 6.2.

Algorithm 6.4 *closure*(G)

Input: $G = (N, \omega, E)$ extended constraint graph with nodes in N and edges in E

Output: LCG G' that is closure of G and has minimal possible set of nodes

$N' = N$;
$E' = E$;
$\Psi = combinations(G)$;
for all $\psi \in \Psi$ **do**
  $max\_score_1 = -\infty$;
  $min\_score_2 = \infty$;
  for all $\overline{\psi} \in permutations(\psi)$ **do**
    $Nodes = nodes(\overline{\psi})$;
    if $\overline{\psi}$ cannot be represented by a path **then**
      continue;
    end if
    $score_1 = 0$;
    $score_2 = 0$;
    for all $n \in Nodes$ **do**
      if $n \in N$ **then**
        $score_1 + +$;
      end if
      $score_2 += variables(n)$;
    end for
    if $(score_1 > max\_score_1) \vee \big((score_1 = max\_score_1) \wedge (score_2 < min\_score_2)\big)$ **then**
      $NewNodes = Nodes$;
    end if
  end for
  $N' = N' \cup NewNodes$;
  add edges to E' so that nodes in $NewNodes$ are connected with all other nodes in N';
end for
$consistent(G')$;

$\psi \in combinations(G)$	path in G' that represents ψ (consists of edges equivalent with those in ψ)
$\{\{\alpha\} \to \{x\}, \emptyset \to \{x, \alpha\}\}$	$\emptyset \to \{x, \alpha\} \to \{2x\}$
$\{\{\alpha\} \to \{y\}, \emptyset \to \{y, \alpha\}\}$	$\emptyset \to \{y, \alpha\} \to \{2y\}$
$\{\{x\} \to \{\alpha\}, \emptyset \to \{x, \alpha\}\}$	$\emptyset \to \{x, \alpha\} \to \{2\alpha\}$
$\{\{y\} \to \{\alpha\}, \emptyset \to \{y, \alpha\}\}$	$\emptyset \to \{y, \alpha\} \to \{2\alpha\}$
$\{\{y\} \to \{x, \alpha\}, \{x\} \to \{y, \alpha\}\}$	$\{y\} \to \{x, \alpha\} \to \{2\alpha, y\}$
$\{\emptyset \to \{x\}, \{y\} \to \{x, \alpha\}, \emptyset \to \{y, \alpha\}\}$	$\emptyset \to \{x\} \to \{x, y, \alpha\} \to \{2x, 2\alpha\}$
$\{\{y\} \to \{x\}, \{y\} \to \{\alpha\}, \emptyset \to \{x, \alpha\}\}$	$\{2y\} \to \{x, y\} \to \{x, \alpha\} \to \{2x, 2\alpha\}$
$\{\emptyset \to \{y\}, \emptyset \to \{x, \alpha\}, \{x\} \to \{y, \alpha\}\}$	$\emptyset \to \{y\} \to \{x, y, \alpha\} \to \{2y, 2\alpha\}$
$\{\{x\} \to \{y\}, \{x\} \to \{\alpha\}, \emptyset \to \{y, \alpha\}\}$	$\{2x\} \to \{x, y\} \to \{y\alpha\} \to \{2y, 2\alpha\}$
$\{\{x\} \to \{\alpha\}, \{y\} \to \{x, \alpha\}, \emptyset \to \{y, \alpha\}\}$	$\{y\} \to \{x, \alpha\} \to \{2\alpha\} \to \{3\alpha, y\}$
$\{\{y\} \to \{\alpha\}, \{x\} \to \{y, \alpha\}, \emptyset \to \{x, \alpha\}\}$	$\emptyset \to \{x, \alpha\} \to \{2\alpha, y\} \to \{3\alpha\}$
All inverse paths	

Table 2: Paths in $closure\{G\}$ that represent sum of expressions returned by $combinations\{G\}$

One solution to overcome this problem could be subsequently repeating operations of minimization and bringing the graph to consistent form unless it is minimal and consistent. However, checking consistency and minimality can be done only by applying the operation $consistent()$ and $minimal()$ to the LDBM representing the graph and then checking whether it was changed by the operation. This would result in a complexity overhead, since the last iteration from the computational point of view would be redundant.

The same result that is obtained by subsequently minimizing LDBM and making it consistent can be obtained by using function $minimal_L(G)$ presented by the Algorithm 6.5.

Let $G = (N, \omega, E)$ be a linear constraint graph corresponding to LDBM M and $G' = (N, \omega', E)$ be a LCG corresponding to the matrix that is obtained by the function $minimal_L(M)$. Then the following is true:

1. $\forall e \in E \ : \ \omega'(e) \leq min\big(\omega(p) \mid p \in path(e)\big)$.
2. G' is consistent.
3. In each equivalence class there exists at least one edge $e \in E$ such that $\omega'(e) =$

Algorithm 6.5 $minimal_L(G)$: minimization of LCG

Arguments: Consistent LDBM M

Output: minimal and consistent version of M

```
for all k ∈ [1..length(M)] do
  for all i ∈ [1..length(M)] do
    for all j ∈ [1..length(M)] do
      if M_{i,k} + M_{k,j} < M_{i,j} then
        for all M_{x,y} ∈ E(M_{i,j}) do
          M_{x,y} = (M_{i,k} + M_{k,j}) · φ_{x,y} / φ_{i,j};
        end for
      end if
    end for
  end for
end for
return M
```

$$min\Big(\omega\big(path(e)\big)\Big)$$

The first property means that each edge $e \in E'$ has weight not higher than any of the paths from $path(e)$ in graph G. This is the actual condition that the graph G' is minimal. Because the graph is also consistent, it implies that weight of each edge $e \in E'$ is not only lower or equal that weight of any path $p \in path(e)$ but is also lower or equal (with respect to the equivalence factor) that any path $p \in path(e')$ for all edges e' that are equivalent with e. The last property means that in each equivalence group there is at least one edge e whose weight was calculated using the weight of the shortest path from $path(e)$. This means that the constraint defined by e is the tightest constraint for expression represented by all edges in the equivalence group. This property implies that G' defines canonical form of a polyhedron represented by graphs for which G' is a closure.

Proof. The correctness of the Algorithm 6.5 can be proved by induction. Let $path_k(e)$ denote all paths that connect the same nodes that are connected by edge e and traverse nodes with index less than k. Now let Φ_k denote the property that after kth iteration of the outer loop all of conditions below are satisfied:

$\Phi_k^1 :\ \forall e \in E\ :\ \omega_{G'}(e) \leq \omega\Big(min\big(path_k(e)\big)\Big)$,

$\Phi_k^2 :\ G'$ is consistent,

$\Phi_k^3 :$ In each equivalence class of G' there exists an edge e such that $\omega'(e) = min\big(\omega(p)|p \in path_k(e)\big)$.

The initial assumption is that before start of the first loop the graph G is consistent. It means that weights of all edges in each equivalence class equals the weight of the lightest edge in the class with respect to the equivalence factors. In the first iteration a weight of each edge $n_i \rightarrow n_j$ is compared to weight of a path $n_i \rightarrow n_1 \rightarrow n_j$. If the latter is lower, weights of all edges $n_x \rightarrow n_y$ equivalent to $n_i \rightarrow n_j$ are changed to the weight of the path $n_i \rightarrow n_1 \rightarrow n_j$ with respect to equivalence factors $\varphi_{x,y}$ and $\varphi_{i,j}$. Because the graph was consistent before, it means that weights of all those edges are now lower than before. Therefore after entire iteration the property Φ_1^1 is satisfied. Since the weight of $n_i \rightarrow n_j$ is also changed, the property Φ_1^3 is satisfied as well. Also, because weight of all edges within equivalence class are always changed together, the graph is still consistent (property Φ_1^2).

In kth step, weights of edges $n_i \rightarrow n_k$ and $n_k \rightarrow n_j$ are lower or equal to any path from $path_k(n_i \rightarrow n_k)$ and $path_k(n_k \rightarrow n_j)$ respectively. Than, if $\omega(n_i \rightarrow n_k) + \omega(n_k \rightarrow n_j) < \omega(n_i \rightarrow n_j)$ the weights of all edges equivalent to $n_i \rightarrow n_j$ are changed. Because before the graph was consistent it means that the new weight is lower than weight of any edge in the equivalence class (property Φ_k^1). Naturally, the graph is consistent after kth iteration, because the same set of paths was compared for each edge within given equivalence class (Φ_k^2). Φ_k^3 is satisfied as well. This means that satisfaction of Φ_{k-1} implies satisfaction of Φ_k and the algorithm is correct.

□

The canonical form of the matrix M from the Figure 26 has following form:

$$M = \left(\begin{array}{c|cccccc} & \emptyset & \{x\} & \{y\} & \{\alpha\} & \{x,\alpha\} & \{y,\alpha\} \\ \hline \emptyset & (0,\leq) & (-1,\leq) & (-1,\leq) & (-1.5,\leq) & (-2.5,\leq) & (-5,\leq) \\ \{x\} & (5,\leq) & (0,\leq) & (2,\leq) & (1,\leq) & (-1.5,\leq) & (-4,\leq) \\ \{y\} & (5,\leq) & (4,\leq) & (0,\leq) & (3.5,\leq) & (1,\leq) & (-1.5,\leq) \\ \{\alpha\} & (6,\leq) & (5,\leq) & (5,\leq) & (0,\leq) & (-1,\leq) & (-1,\leq) \\ \{x,\alpha\} & (11,\leq) & (6,\leq) & (8,\leq) & (5,\leq) & (0,\leq) & (2,\leq) \\ \{y,\alpha\} & (11,\leq) & (10,\leq) & (6,\leq) & (5,\leq) & (4,\leq) & (0,\leq) \end{array}\right)$$

6.3 Operations on EDBM

Since the elements of a EDBM are numerical bounds, the property checking operations do not differ from corresponding operations on the standard DBM. On the other hand the transformation operations must be reconsidered, since the elements of EDBM define bounds of expressions involving clocks and parameters and the parameters do not change in time.

Below all the matrices that are subject of the operations are assumed to be in canonical form. $M_{A,B}$ will represent element of the EDBM in row labelled with A and column labelled with B, that define bound on expression $\sum A - \sum B$ such that $A = \{\gamma_{A1}, \cdots, \gamma_{An}\}$ and $B = \{\gamma_{B1}, \cdots, \gamma_{Bn}\}$. $M_{i,\bullet}$ and $M_{\bullet,i}$ will denote ith row and ith column of the matrix M respectively.

6.3.1 Property checking

$\mathbf{empty_E(M)}$ Testing whether a polyhedron represented by EDBM M is empty is done by checking whether the elements on diagonal are lower than $(0,\leq)$. The operation $empty_E(M)$ is implemented by the Algorithm 6.6.

$\mathbf{includes_E(M, M')}$ The function $includes_E(M, M')$ checks inclusion relation for EDBMs M and M'. It returns *true* if all valuations that belong to polyhedron represented by M belong also to the polyhedron represented by M'.

If M and M' are in minimal and consistent form it is enough to check whether all bounds

Algorithm 6.6 $empty_E(M)$

Arguments: canonical EDBM M

Returned value: a boolean value indicating whether M representing non-empty portion of space

```
for all i ∈ [1..length(M)] do
  if M_{i,i} < (0, ≤) then
    return false;
  end if
end for
return true;
```

defining M are lower or equal than corresponding bounds defined by M'. This test is implemented by Algorithm 6.7

Algorithm 6.7 $includes_E(M, M'))$

Arguments: canonical EDBMs M and M'

Returned value: a boolean value indicating whether polyhedron represented by M is included by polyhedron represented by M'.

```
for all i ∈ [1..length(M)] do
  for all j ∈ [1..length(M)] do
    if M'_{i,j} < M_{i,j} then
      return false
    end if
  end for
end for
return true
```

satisfies$(\mathbf{M}, (\sum \mathbf{A} - \sum \mathbf{B} \prec \mathbf{m}))$ This function verifies whether there are valuations in polyhedron represented by M that satisfy the constraint $\sum A - \sum B \prec m$, where A and B are sets represented by some row and column of M. As in case of standard DBM it is enough to find out whether $(0, \leq) \leq (m, \prec) + M_{B,A}$. Since the matrix M is assumed to be

consistent, there is no need to check elements equivalent to $M_{A,B}$.

6.3.2 Transformations

$\mathbf{and_E(M, \sum A - \sum B \prec m)}$ Operation $and_E(M, \sum A - \sum B \prec m)$ represent adding the constraint $\sum A - \sum B \prec m$ to the polyhedron represented by M. As in case of DBMs the basic step for this operation is to replace element $M_{A,B}$ and all equivalent elements, with the bound $(m, \prec)$ in case it is lower than current bound at $M_{A,B}$. To check whether the matrix is still positive, the sum $(m, \prec) + M_{B,A}$ must be positive. Because the EDBM is assumed to be consistent, respective sums of other elements in the equivalence classes are also positive and do not have to be checked.

Since the resulting matrix may be not minimal, it must be minimized using LDBM that is closure of the EDBM M. The cost of the minimization may be reduced, by taking advantage of the fact that only element $M_{A,B}$ and its equivalent elements were changed. For the ECG corresponding to M it means that weights of all edges in the graph are equal to weights of shortest paths that do not traverse nodes that are source or destination of altered edges. Therefore, only those paths that traverse those nodes must be checked in order to re-canonicalize the matrix. Those nodes in the matrix are represented by rows and columns that contain altered elements. This approach is implemented by the Algorithm 6.8.

The complexity of Algorithm 6.8 depends on the size of equivalence class of the altered element. In the best case, this size equals 1, which means that in fact the algorithm behaves exactly as the Algorithm 4.5 for standard DBMs and has complexity $\mathcal{O}(n^2)$. The size of biggest equivalence class depends on the number of rows and columns labelled with a set containing the same variable. In general, the order of magnitude of the size of equivalence classes equals n, so the entire operation $and_E()$ has maximum complexity $\mathcal{O}(n^3)$. Note, that n is the size of closure of the altered EDBM, so it can be much bigger than size of the EDBM.

Exemple 6.6. *Let Z be a polyhedron over the clock x and parameter α, depicted in the Figure 28. Z is described by EDBM M:*

Algorithm 6.8 $and_E(M, \sum A - \sum B \prec m))$

Arguments: Canonical EDBM M; constraint $\sum A - \sum B \prec m$

Returned value: Canonical EDBM that represents intersection of M and the constraint $\sum A - \sum B \prec m$.

if $includes(M, \sum A - \sum B \prec m)$ **then**
  $M_{0,0} = (-1, \prec)$;
  return M;
end if
for all i, j : $M_{i,j} \in equivalent(M_{A,B})$ **do**
  $M_{i,j} = (m, \prec) \cdot \frac{\varphi_{i,j}}{\varphi_{A,B}}$;
end for
$M' = closure(M)$;
for all k such that $M'_{k,\bullet}$ or $M'_{\bullet,k}$ contain element equivalent to $M'_{A,B}$ **do**
  for all $i \in [1 : length(M')$ **do**
    for all $j \in [1 : length(M')$ **do**
      $M'_{i,j} = min(M'_{i,j}, M'_{i,k} + (M'_{k,j}))$
    end for
  end for
end for
for all $i \in [1..length(M)]$ **do**
  for all $j \in [1..length(M)]$ **do**
    $M_{i,j} = M'_{i,j}$;
  end for
end for
return M

$$M = \left(\begin{array}{c|cccc} & \emptyset & \{x\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (0,\leq) & (-1,\leq) & (-2,\leq) \\ \{x\} & (3,\leq) & (0,\leq) & (1,\leq) & (-1,\leq) \\ \{\alpha\} & (4,\leq) & (3,\leq) & (0,\leq) & (0,\leq) \\ \{x,\alpha\} & (6,\leq) & (4,\leq) & (3,\leq) & (0,\leq) \end{array}\right)$$

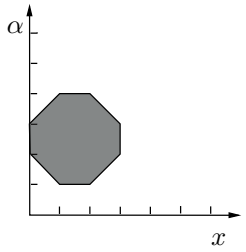


Figure 28: Parametric polyhedron

The example will show procedure for intersecting Z with the constraint $x \leq 1$, which is done by function and$_E(M, x \leq 1)$. First step is checking whether introducing this constraint will not make M negative. Because the constraint for x is represented by element $M_{2,1}$ it must be checked whether $(0,\leq) \leq (1,\leq) + M_{1,2}$. Since yes, the resulting matrix will be positive.

To calculate the canonical form of the new polyhedron it is necessary to use LDBM that is closure of M. It has two additional rows and columns, labelled with $\{2x\}$ and $\{2\alpha\}$. The minimal form of closure(M) has following form:

$$M' = \left(\begin{array}{c|cccccc} & \emptyset & \{x\} & \{\alpha\} & \{x,\alpha\} & \{2x\} & \{2\alpha\} \\ \hline \emptyset & (0,\leq) & (0,\leq) & (-1,\leq) & (-2,\leq) & (0,\leq) & (-2,\leq) \\ \{x\} & (3,\leq) & (0,\leq) & (1,\leq) & (-1,\leq) & (0,\leq) & (0,\leq) \\ \{\alpha\} & (4,\leq) & (3,\leq) & (0,\leq) & (0,\leq) & (3,\leq) & (-1,\leq) \\ \{x,\alpha\} & (6,\leq) & (4,\leq) & (3,\leq) & (0,\leq) & (3,\leq) & (1,\leq) \\ \{2x\} & (4,\leq) & (3,\leq) & (4,\leq) & (1,\leq) & (0,\leq) & (2,\leq) \\ \{2\alpha\} & (8,\leq) & (7,\leq) & (4,\leq) & (3,\leq) & (6,\leq) & (0,\leq) \end{array}\right)$$

Changing element $M_{2,1}$ must have consequences in changing all equivalent elements, in this case $M_{5,1}$ (with $\varphi_{5,1} = 2$) and $M_{5,2}$. Since the altered elements in the LCG corresponding to M' represent edges $\emptyset \to \{x\}$, $\emptyset \to \{2x\}$ and $\{x\} \to \{2x\}$, weights of all paths that traverse

nodes $\emptyset$, $\{x\}$ and $\{2x\}$ must be calculated again. This nodes are represented respectively by first, second and fifth row and column of M', so k in the Algorithm 6.8 will take values from the set $\{1, 2, 5\}$. The minimized LDBM M' has following form:

$$minimal(M') = \left(\begin{array}{c|cccccc} & \emptyset & \{x\} & \{\alpha\} & \{x,\alpha\} & \{2x\} & \{2\alpha\} \\ \hline \emptyset & (0,\leq) & (0,\leq) & (-1,\leq) & (-2,\leq) & (0,\leq) & (-2,\leq) \\ \{x\} & (1,\leq) & (0,\leq) & (0,\leq) & (-1,\leq) & (0,\leq) & (-1,\leq) \\ \{\alpha\} & (4,\leq) & (3,\leq) & (0,\leq) & (0,\leq) & (3,\leq) & (-1,\leq) \\ \{x,\alpha\} & (5,\leq) & (4,\leq) & (1,\leq) & (0,\leq) & (2,\leq) & (1,\leq) \\ \{2x\} & (2,\leq) & (1,\leq) & (1,\leq) & (0,\leq) & (0,\leq) & (0,\leq) \\ \{2\alpha\} & (8,\leq) & (7,\leq) & (4,\leq) & (3,\leq) & (6,\leq) & (0,\leq) \end{array}\right)$$

The polyhedron represented by EDBM that is result of operation $and_E(M, x \leq 1)$ is depicted in the Figure 29

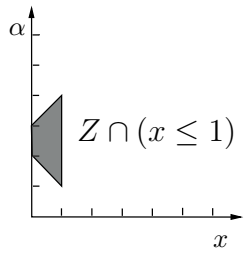


Figure 29: Polyhedron represented by result of operation $and_E(M, x \leq 1)$

intersection$_{\mathbf{E}}$ $(\mathbf{M}, \mathbf{M'})$ Intersecting two polyhedra represented by EDBMs M and M' can be done as in case of DBMs by taking for each element lower value from M or M'. The resulting EDBM is not canonical. To reduce the cost of re-canonicalization, the constraint graph representing M' may be reduced using Algorithm 2.3. Then the intersection may be calculated using the $and(M, \sum A - \sum B \prec m)$, for all non-redundant constraints of M'.

future$_{\mathbf{E}}$ $(\mathbf{M})$ In standard DBMs where all rows and columns represent clocks forward time elapse is implemented by removing upper constraints on all clocks, which is equivalent to replacing all elements of the first column with $(\infty, <)$ (except element $M_{0,0}$).

This operation is a little bit more complex in case of EDBM, since the rows and columns of EDBM represent not only clocks, but also parameters or sum of clocks and/or parameters.

The forward time elapse of valuations in polyhedron Z is represented by extending Z with all valuations that can be reached from Z in the future, assuming that all clocks proceed with the same tempo. In general, all valuations of Z satisfy set of constraints in form $\sum A - \sum B \prec m$. If A contains more clocks than B, the difference $\sum A - \sum B$ will grow with each second, going to infinity. Therefore implementation of operation $future_E()$ is done by assigning the bound $(\infty, <)$ to all elements of EDBM $M_{A,B}$ for which the set A contains more clocks than B.

The function $future_E(M)$ is implemented by the Algorithm 6.9. The canonical form of EDBM is preserved by this operation.

Algorithm 6.9 $future_E(M)$

Input: EDBM M in minimal and consistent form representing polyhedron Z

Output: EDBM representing $Z^{\uparrow}$

$M^{\uparrow} = M$;
for all $i \in [1..n]$ **do**
  if $|\big(label(i) \cap \mathcal{C}\big)| > 0$ **then**
    for all $j \in [1..n]$ **do**
      if $|\big(label(j) \cap \mathcal{C}\big)| < |\big(label(i) \cap \mathcal{C}\big)|$ **then**
        $M_{i,j} = (\infty, <)$;
      end if
    end for
  end if
end for
return $M^{\uparrow}$;

For matrix M from Example 6.6 the operation $future_E(M)$ will return following matrix:

$$future_E(M) = \left(\begin{array}{c|cccc} & \emptyset & \{x\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (0,\leq) & (-1,\leq) & (-2,\leq) \\ \{x\} & (\infty,<) & (0,\leq) & (\infty,<) & (-1,\leq) \\ \{\alpha\} & (4,\leq) & (3,\leq) & (0,\leq) & (0,\leq) \\ \{x,\alpha\} & (\infty,<) & (4,\leq) & (\infty,<) & (0,\leq) \end{array}\right)$$

The polyhedron represented by $future_E(M)$ is depicted in the Figure 30.

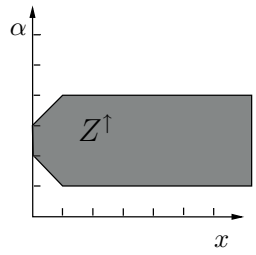


Figure 30: Polyhedron represented by $future_E(M)$

past$_{\mathbf{E}}$(M) The backward time elapse represents extending polyhedron Z with those valuations that can reach Z by forward time elapse. In DBMs this was implemented by replacing lower bounds of individual clocks by $(0,\leq)$.

Let expression bounded by element $M_{A,B}$ be written as $\sum A - \sum B = (\sum A_C - \sum B_C) + (\sum A_P - \sum B_P)$, such that sets A_C and B_C contain only clocks and sets A_P and B_P contain only parameters. When time runs backwards, the bound of the difference $\sum A - \sum B$ grows only, if A contains less clocks than B. Remind that values of individual clocks may not be lower than 0. This means that if B contains only clocks and A is empty, the element $M_{A,B}$ must be changed to $(0,\leq)$ (this concerns also lower bounds of all individual clocks, i.e. where $A = \emptyset$ and B is labelled with a single clock). Other elements $M_{A,B}$ such that $|B_C| > |A_C|$ and $|A| > 0$ must be changed to $(\infty,<)$. The pseudocode for backward time elapse is presented by the Algorithm 6.10.

Note that this operation does not preserve the canonical form of the matrix.

For matrix M from Example 6.6 the operation $past_E(M)$ will return following matrix:

Algorithm 6.10 $past_E(M)$

Input: EDBM M in minimal and consistent form representing a polyhedron Z

Output: EDBM representing $Z^{\downarrow}$

$M^{\downarrow} = M$;
for all $i \in [1..n]$ **do**
  for all $j \in [1..n]$ **do**
    if $|\big(label(i) \cap \mathcal{C}\big)| < |\big(label(j) \cap \mathcal{C}\big)|$ **then**
      if $|label(i)| > 0$ **then**
        $M_{i,j} = (0, \leq)$;
      else
        $M_{i,j} = (\infty, <)$;
      end if
    end if
  end for
end for
return $M^{\downarrow}$;

$$past_E(M) = \left(\begin{array}{c|cccc} & \emptyset & \{x\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (0,\leq) & (-1,\leq) & (-1,\leq) \\ \{x\} & (3,\leq) & (0,\leq) & (1,\leq) & (-1,\leq) \\ \{\alpha\} & (4,\leq) & (4,\leq) & (0,\leq) & (0,\leq) \\ \{x,\alpha\} & (6,\leq) & (4,\leq) & (3,\leq) & (0,\leq) \end{array}\right)$$

The polyhedron represented by $past_E(M)$ is depicted in the Figure 31.

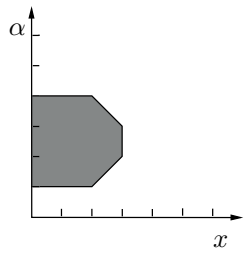


Figure 31: Polyhedron represented by $past_E(M)$

reset$_\mathbf{E}$(M, X) The clock reset may be implemented by setting the values of $M_{x,1}$ and $M_{1,x}$ to $(0,\leq)$ for all $x \in X$ and other elements in rows and columns where x is involved to $(\infty,<)$. The resulting matrix will not remain the canonical form then.

It is possible to define the operation $reset_E()$ in the way that the EDBM remains in its canonical form, however it depends on the content of the matrix. Let $M_{A+x,B}$ denote an element representing bound of the expression $\sum A + x - \sum B$, such that $x \subseteq X$. Assigning 0 to all clocks in x means that the bound of $\sum A + x - \sum B$ becomes in fact the bound of the expression $\sum A - \sum B$, which is represented by the element $M_{A,B}$. Thus, entire row $M_{A+x,\bullet}$ can be replaced by the row $M_{A,\bullet}$. For the columns the situation is analogical: reseting clocks in x means that the bound of $\sum A - (\sum B + x)$ becomes equivalent to bound of $\sum A - \sum B$, what means that the column $M_{\bullet,B+x}$ must be replaced with the column $M_{\bullet,B}$. It may happen, that the EDBM M does not contain row and column representing A and B. In that case the all elements in row $M_{A+x,\bullet}$ and column $M_{\bullet,B+x}$ must be replaced with $(\infty,<)$ and the matrix M must be recanonicalized.

The operation $reset_E(M, X)$ is implemented by the Algorithm 6.11.

The result of operation $reset_E(M, x)$ for matrix M from Example 6.6 is following:

Algorithm 6.11 $reset_E(M, X)$

Arguments: canonical EDBM M, representing polyhedron Z; Set of reset clocks X

Output: EDBM representing $Z[X := 0]$

```
canonical = true;
for all i ∈ [1..n] do
  x = label(i) ∩ X
  if x ≠ ∅ then
    if ∃k | label(k) = label(i) \ x then
      for all j ∈ [1..n] do
        M_{i,j} = M_{k,j};
        M_{j,i} = M_{j,k};
      end for
    else
      for all j ∈ [1..n] do
        M_{i,j} = (∞, <);
        M_{j,i} = (∞, <);
        canonical = false
      end for
    end if
  end if
end for
if canonical = false then
  M = canonical(M);
end if
return M;
```

$$reset_E(M,x) = \left(\begin{array}{c|cccc} & \emptyset & \{x\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (0,\leq) & (-1,\leq) & (-1,\leq) \\ \{x\} & (0,\leq) & (0,\leq) & (-1,\leq) & (-1,\leq) \\ \{\alpha\} & (4,\leq) & (4,\leq) & (0,\leq) & (0,\leq) \\ \{x,\alpha\} & (4,\leq) & (4,\leq) & (0,\leq) & (0,\leq) \end{array}\right)$$

The polyhedron represented by $reset_E(M,x)$ is depicted in the Figure 32.

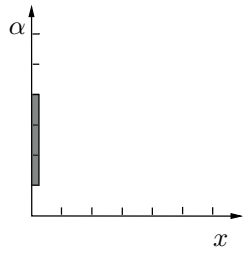


Figure 32: Polyhedron represented by $reset_E(M,x)$

unreset$_\mathbf{E}$(M, X) As in case of standard DBM the first step for operation $unreset_E(M,X)$ is to restrict polyhedron represented by M to those valuations that can be result of reset clocks in X. This is done by operation $and_E(M, x \leq 0)$ for all clocks $x \in X$. Then the constraints on clocks in X are released.

Releasing constraints on clocks in X is analogical to the operation $future_E(M)$ where only elements of X are considered as clocks. All the parameters and clocks not in X do not change their values. Therefore, to implement it, it is enough to replace all elements $M_{A,B}$, such that A contains more elements of X than B, with $(\infty, <)$.

The operation $unreset_E(M,X)$ is implemented by Algorithm 6.12.

Since operations $and_E()$ and $future_E()$ return canonical matrices, the result of $unreset_E(M,X)$ is also canonical.

Result of $unreset_E(M,x)$ for matrix M from Example 6.6 is following:

Algorithm 6.12 $unreset(M, X)$

Arguments:EDBM M representing polyhedron Z; set of unreset clocks X

Output: EDBM M' representing $[X := 0]Z$

```
for all x ∈ X do
  M = and(M, x ≤ 0);
end for
if ¬empty(M) then
  for all i ∈ [1..n] do
    for all j ∈ [1..n] do
      if |label(i) ∩ X| > |label(j) ∩ X| then
        M_{i,j} = (∞, <);
      end if
    end for
  end for
end if
return M
```

$$unreset_E(M,x) = \left(\begin{array}{c|cccc} & \emptyset & \{x\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (0,\leq) & (-2,\leq) & (-2,\leq) \\ \{x\} & (\infty,<) & (0,\leq) & (\infty,<) & (-2,\leq) \\ \{\alpha\} & (3,\leq) & (3,\leq) & (0,\leq) & (0,\leq) \\ \{x,\alpha\} & (\infty,<) & (3,\leq) & (\infty,<) & (0,\leq) \end{array}\right)$$

The resulting polyhedron is depicted in the Figure 33.

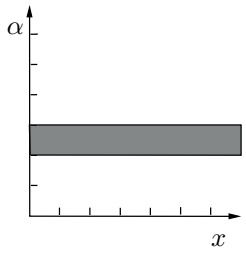


Figure 33: Polyhedron represented by $unreset_E(M,x)$

6.4 Symbolic analysis using EDBM

Symbolic forward and backward analysis of parameterized timed automata can be done by applying EDBM operations to the definitions of $post()$ and $pred()$ operations.

Recall that for zones $H = (l, Z)$ and $H' = (l', Z')$ and transition $t = (l, a, Z_G, r, l')$ the $post()$ and $pred()$ operations are defined by:

$$post(H,t) = (l', ((Z^{\uparrow} \cap Inv(l))[r := 0])Inv(l'))$$

$$pred(H',t) = ([r := 0]Z \cap Z_t \cap Inv(l))^{\downarrow} \cap Inv(l)$$

Let Z, $Inv(l)$, Z_G and $Inv(l')$ be represented by matrices M, $M_{Inv(l)}$, M_G and $M_{Inv(l')}$. The operations $post()$ and $pred()$ can be implemented as:

- $post(l, Z) = (l', Z')$
- $pred(l', Z') = (l, Z'')$

Algorithm 6.13 $post_E(M, t)$

$M = future_E(M)$;
$M = intersect(M, M_{Inv(l)})$;
$M = canonical_E(M)$;
$M = reset_E(M, r)$;
$M = intersect\big(M, M_{Inv(l')}\big)$;
$M = canonical_E(M)$;
return M;

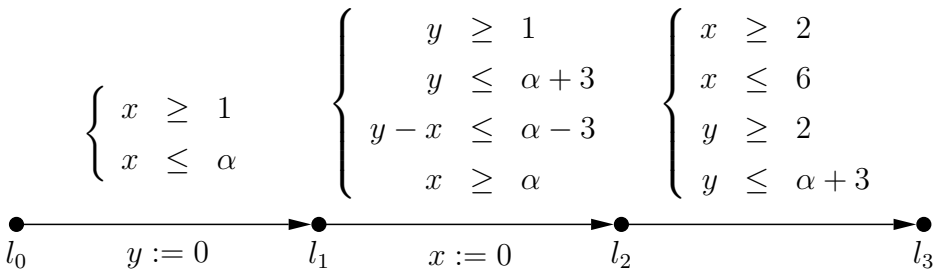
Algorithm 6.14 $pred_E(M, t)$

$M = unreset_E(M, r)$;
$M = intersect(M, M_{Inv(l)})$;
$M = intersect(M, M_G)$;
$M = canonical_E(M)$;
$M = past_E(M)$;
$M = intersect\big(M, M_{Inv(l)}\big)$;
$M = canonical_E(M)$;
return M;

such that Z' and Z'' are represented by EDBMs M' and M'' obtained in the way described by Algorithms 6.13 and 6.14 respectively.

Due to the fact that an EDBM defines values of clocks and parameters, the EDBM representing an initial state must define the values of clocks to 0 and the values of parameters according to initial constraints defined for the analyzed system.

Exemple 6.7. *Consider the following path, where x and y are clocks and α is parameter defined by the constraint $0 \leq \alpha \leq 10$:*



Since constraints in the transition guards are defined for x, y, $x - y$ and $x + \alpha$, the EDBM used for analysis of the path need to define rows and columns labelled with $\emptyset$, $\{x\}$, $\{y\}$, $\{\alpha\}$ and $\{x, \alpha\}$. Additionally, the LDBM used for canonicalization must have rows and columns labelled with $\{2x\}$, $\{2y\}$, $\{2\alpha\}$, $\{y, \alpha\}$ and $\{2x, 2\alpha\}$. The initial state of the symbolic path is defined by $H_0 = (l_0, Z_0)$ where Z_0 is described by the EDBM M_0:

$$M_0 = \left(\begin{array}{c|ccccc} & \emptyset & \{x\} & \{y\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (0,\leq) & (0,\leq) & (0,\leq) & (0,\leq) \\ \{x\} & (0,\leq) & (0,\leq) & (0,\leq) & (0,\leq) & (0,\leq) \\ \{y\} & (0,\leq) & (0,\leq) & (0,\leq) & (0,\leq) & (0,\leq) \\ \{\alpha\} & (10,\leq) & (10,\leq) & (10,\leq) & (0,\leq) & (0,\leq) \\ \{x,\alpha\} & (10,\leq) & (10,\leq) & (10,\leq) & (0,\leq) & (0,\leq) \end{array}\right)$$

The first step is applying operation $future_E(M_0)$, what gives following result:

$$future(M_0) = \left(\begin{array}{c|ccccc} & \emptyset & \{x\} & \{y\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (0,\leq) & (0,\leq) & (0,\leq) & (0,\leq) \\ \{x\} & (\infty,<) & (0,\leq) & (0,\leq) & (\infty,<) & (0,\leq) \\ \{y\} & (\infty,<) & (0,\leq) & (0,\leq) & (\infty,<) & (0,\leq) \\ \{\alpha\} & (10,\leq) & (10,\leq) & (10,\leq) & (0,\leq) & (0,\leq) \\ \{x,\alpha\} & (\infty,<) & (10,\leq) & (10,\leq) & (\infty,<) & (0,\leq) \end{array}\right)$$

The resulting matrix must be intersected with constraints $x \geq 1$ and $x \leq \alpha$ what is done by subsequent applying operations $and(M, -x \leq 1)$ and $and(M, x - \alpha \leq 0)$. The result is presented below:

$$and\Big(and\big(future(M_0), -x \leq -1\big), x - \alpha \leq 0\Big) =$$

$$\left(\begin{array}{c|ccccc} & \emptyset & \{x\} & \{y\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (-1,\leq) & (-1,\leq) & (-1,\leq) & (-2,\leq) \\ \{x\} & (10,\leq) & (0,\leq) & (0,\leq) & (0,\leq) & (-1,\leq) \\ \{y\} & (10,\leq) & (0,\leq) & (0,\leq) & (0,\leq) & (-1,\leq) \\ \{\alpha\} & (10,\leq) & (9,\leq) & (9,\leq) & (0,\leq) & (-1,\leq) \\ \{x,\alpha\} & (20,<) & (10,\leq) & (10,\leq) & (10,\leq) & (0,\leq) \end{array}\right)$$

Finally, to complete the operation $post_E(Z_0, t_0 \rightarrow t_1)$ the operation $reset_E(M, x)$ must be applied:

$$M_1 = post(M_0, t_1 \rightarrow t_2) = \left(\begin{array}{c|ccccc} & \emptyset & \{x\} & \{y\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (-1,\leq) & (0,\leq) & (-1,\leq) & (-2,\leq) \\ \{x\} & (10,\leq) & (0,\leq) & (10,\leq) & (0,\leq) & (-1,\leq) \\ \{y\} & (0,\leq) & (-1,\leq) & (0,\leq) & (-1,\leq) & (-2,\leq) \\ \{\alpha\} & (10,\leq) & (9,\leq) & (10,\leq) & (0,\leq) & (-1,\leq) \\ \{x,\alpha\} & (20,<) & (10,\leq) & (20,\leq) & (10,\leq) & (0,\leq) \end{array}\right)$$

The remaining matrices that represent polyhedra that are results of forward analysis done by applying post() operations are as follows:

$$M_2 = post(M_1, l_1 \rightarrow l_2) = \left(\begin{array}{c|ccccc} & \emptyset & \{x\} & \{y\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (0,\leq) & (-1,\leq) & (-2,\leq) & (-2,\leq) \\ \{x\} & (0,\leq) & (0,\leq) & (-1,\leq) & (-2,\leq) & (-2,\leq) \\ \{y\} & (9,\leq) & (9,\leq) & (0,\leq) & (-1,\leq) & (-1,\leq) \\ \{\alpha\} & (10,\leq) & (10,\leq) & (9,\leq) & (0,\leq) & (0,\leq) \\ \{x,\alpha\} & (10,\leq) & (10,\leq) & (9,\leq) & (0,\leq) & (0,\leq) \end{array}\right)$$

$$M_3 = post(M_2, l_2 \rightarrow l_3) = \left(\begin{array}{c|ccccc} & \emptyset & \{x\} & \{y\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (-2,\leq) & (-3,\leq) & (-2,\leq) & (-4,\leq) \\ \{x\} & (6,\leq) & (0,\leq) & (-1,\leq) & (2,\leq) & (-2,\leq) \\ \{y\} & (13,\leq) & (9,\leq) & (0,\leq) & (3,\leq) & (-1,\leq) \\ \{\alpha\} & (10,\leq) & (8,\leq) & (7,\leq) & (0,\leq) & (-2,\leq) \\ \{x,\alpha\} & (16,\leq) & (10,\leq) & (9,\leq) & (6,\leq) & (0,\leq) \end{array}\right)$$

The EDBMs obtained by backward analysis have following form:

$$M'_3 = M_3$$

$$M'_2 = intersect\big(M_2, pred(M'_3, t_2 \rightarrow t_3)\big) =$$

$$\left(\begin{array}{c|ccccc} & \emptyset & \{x\} & \{y\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (0,\leq) & (-1,\leq) & (-2,\leq) & (-2,\leq) \\ \{x\} & (0,\leq) & (0,\leq) & (-1,\leq) & (-2,\leq) & (-2,\leq) \\ \{y\} & (9,\leq) & (9,\leq) & (0,\leq) & (-1,\leq) & (-1,\leq) \\ \{\alpha\} & (10,\leq) & (10,\leq) & (9,\leq) & (0,\leq) & (0,\leq) \\ \{x,\alpha\} & (10,\leq) & (10,\leq) & (9,\leq) & (0,\leq) & (0,\leq) \end{array}\right)$$

,

$$M'_1 = intersect\big(M_1, pred(M'_2, t_1 \rightarrow t_2)\big) =$$

$$\left(\begin{array}{c|ccccc} & \emptyset & \{x\} & \{y\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (-1,\leq) & (0,\leq) & (-2,\leq) & (-3,\leq) \\ \{x\} & (9,\leq) & (0,\leq) & (9,\leq) & (0,\leq) & (-2,\leq) \\ \{y\} & (0,\leq) & (-1,\leq) & (0,\leq) & (-2,\leq) & (-3,\leq) \\ \{\alpha\} & (10,\leq) & (9,\leq) & (10,\leq) & (0,\leq) & (-1,\leq) \\ \{x,\alpha\} & (19,<) & (10,\leq) & (19,\leq) & (9,\leq) & (0,\leq) \end{array}\right)$$

,

$$M'_0 = intersect\big(M_0, pred(M'_1, l_0 \rightarrow l_1)\big) =$$

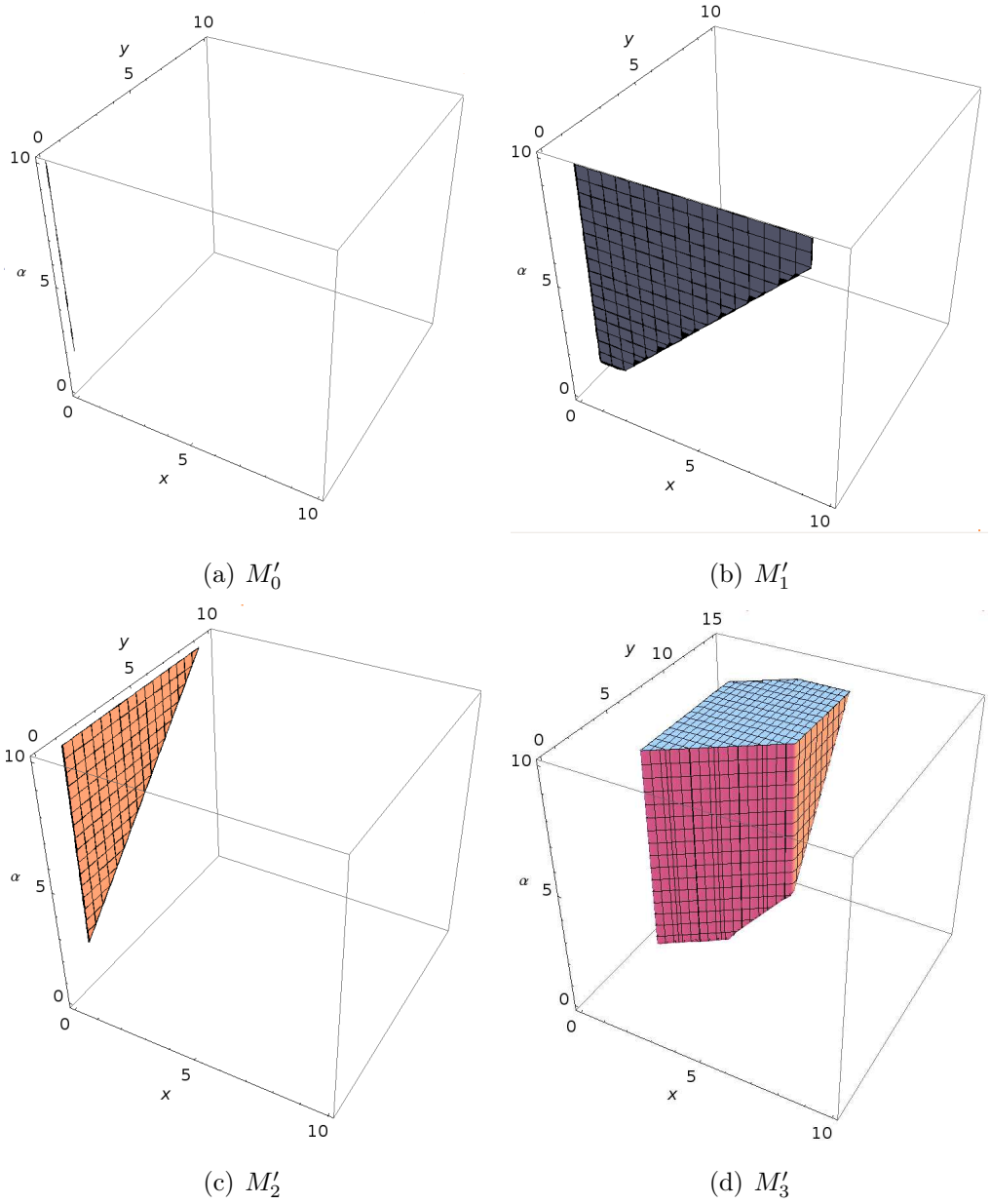
$$\left(\begin{array}{c|ccccc} & \emptyset & \{x\} & \{y\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (0,\leq) & (0,\leq) & (-2,\leq) & (-2,\leq) \\ \{x\} & (0,\leq) & (0,\leq) & (0,\leq) & (-2,\leq) & (-2,\leq) \\ \{y\} & (0,\leq) & (0,\leq) & (0,\leq) & (-2,\leq) & (-2,\leq) \\ \{\alpha\} & (10,\leq) & (10,\leq) & (10,\leq) & (0,\leq) & (0,\leq) \\ \{x,\alpha\} & (10,\leq) & (10,\leq) & (10,\leq) & (0,\leq) & (0,\leq) \end{array}\right)$$

The polyhedra represented by EDBMs obtained by forward backward analysis are depicted in the Figure 34. Note that despite that the allowed values of the parameter α were in the range $[0, 10]$ the path is executable only, when the value of α is in the range $[2, 10]$.

6.5 Summary

EDBM presented in this chapter is a compact data structure that allows symbolic representation of both clocks and paraeters within one matrix. Although the size of EDBM is bigger than initial size of constrained PDBM, the latter may grow during the analysis to unpredictable and unmanagable sizes.

To understand benefits of using EDBM let assume that adding one parameter to a specification of system with n clocks will result in one ambiguous result of comparison operation (e.g. during intersection of two matrices). Note that this is rather optimistic assumption. Such duplication will cause that the memory consumed by PDBM will grow from n^2 to $2n^2$ (the growth in this case is n^2). Considering the same parameter in EDBM will result in growing the matrix from n^2 to $(n+1)^2$ (in this case the net growth equals to $2n+1$).



(a) M'_0 (b) M'_1

(c) M'_2 (d) M'_3

Figure 34: Polyhedra representing symbolic states obtained by forward and backward analysis

Using EDBM is beneficial then, if $n^2 > 2n + 1$ what is true for all $n \geq 3$.

Situation gets more complicated if constraints in the specification requires EDBM to have rows and columns labelled with sets containing more than one element. In this case the size of LDBM that is a closure of given EDBM may drastically grow. Experiments with implementation of the Algorithm 6.4 showed that each row that is labeled with set of two

elements causes that the length of EDBM's closure roughly doubles. However, as shown later in the Section 7.1 efficient implementation causes that the memory consumption in this case does not to be so high as it comes from the size of the closure matrix.

7 Implementation and experiments

The implementation of the framework proposed in the previous chapter is documented here. The Section 7.1 shows some tips for efficient implementation of the EDBM data structure. The Section 7.2 shows how implementation of EDBM was embedded in a toy-tool for analysis of real time systems modeled as communicating systems of TA. The format of input files of this tool is documented by the Section 7.3. The Section 7.4 describes a few test generation algorithms that have been implemented on the top of the tool, finally the Section 7.5 reports experiments with generating test cases for the FlexRay MAC process.

Contents

7.1 Implementation of EDBM and LDBM

7.1.1 Implementation of bound

`Bound` is a class that defines elements of an EDBM. The class defines two attributes:

```
double value;
bool closure;
```

The two attributes define value of the bound and its closure. The `value` attribute is a real value whereas the `closure` is boolean indicating whether the bounding element is "$\leq$" (**true**) or "$<$" (**false)**.

The class defines binary operators that allow summing two bounds and multiplying a bound and real value. Apart from that, all possible comparison operators are also defined. All public methods of the `Bound` class are presented below:

```
Bound(const double, const bool);
Bound(const Bound&);
void operator+(const Bound&);
void operator*(const double);
bool operator<(const Bound&);
bool operator<=(const Bound&);
bool operator==(const Bound&);
bool operator!=(const Bound&);
bool operator>(const Bound&);
bool operator>=(const Bound&);
```

7.1.2 EDBM class implementation

Implementing EDBM (and LDBM) as a matrix of bounds would be inefficient, due to existence of equivalent elements. It is more convenient and sensible to implement the structure using an associative container where the bounds are associated with expressions that they limit. For example an element in a row labelled with $\{x, y\}$ and a column labelled with $\{y\}$ would be represented by the same bound that element of a row labelled with $\{x\}$ and a

column labelled with $\{\emptyset\}$. Apart from reducing memory consumption such approach facilitates operations like intersection or canonicalization, since the EDBM and LDBM stored in this way are always consistent.

EDBM has been implemented in C++ as `EDBM` class, using elements of standard library (STL) and boost library [1].

Attributes of the EDBM class

The attributes of the `EDBM` class are declared as follows:

```
std::vector<std::string> variables_defs;
std::vector<bool> clocks;
RowDefinition edbm_row_definition;
RowDefinition ldbm_row_definition;
std::map<Expression, Bound> values;
```

where `RowDefinition` and `Expression` are types defined by:

```
typedef boost::numeric::ublas::vector<int> Expression;
typedef std::vector<Expression> RowDefinition;
```

The attribute `variables_defs` is a vector that contains labels of clocks and parameters that are used for labelling rows and columns of the EDBM matrix. The attribute `clocks` is a boolean vector that indicate which of the elements of the vector `variables_defs` are clocks. The next two attributes, namely `edbm_row_definition` and `ldbm_row_definition`, define how each row and column of EDBM and its closure are labelled (it is assumed that rows and columns are labelled in the same order, i.e. row i is labelled with the same set that column i). Technically these attributes are vectors of boolean vectors. The attribute `values` is a table that contains all values of the EDBM and corresponding closure (note that EDBM is entirely contained in its closure). It is implemented as a map, where key is a vector defining bounded expression and the value is a bound of this expression. The `values` store only bounds for expression for which the equivalence factor equals 1 – other values can be derived from them.

Exemple 7.1. *Consider following EDBM:*

$$M = \left(\begin{array}{c|cccc} & \emptyset & \{x\} & \{\alpha\} & \{x,\alpha\} \\ \hline \emptyset & (0,\leq) & (0,\leq) & (-1,\leq) & (-2,\leq) \\ \{x\} & (3,\leq) & (0,\leq) & (1,\leq) & (-1,\leq) \\ \{\alpha\} & (4,\leq) & (3,\leq) & (0,\leq) & (0,\leq) \\ \{x,\alpha\} & (6,\leq) & (4,\leq) & (3,\leq) & (0,\leq) \end{array}\right)$$

Its corresponding LDBM has following form:

$$M' = \left(\begin{array}{c|cccccc} & \emptyset & \{x\} & \{\alpha\} & \{x,\alpha\} & \{2x\} & \{2\alpha\} \\ \hline \emptyset & (0,\leq) & (0,\leq) & (-1,\leq) & (-2,\leq) & (0,\leq) & (-2,\leq) \\ \{x\} & (3,\leq) & (0,\leq) & (1,\leq) & (-1,\leq) & (0,\leq) & (0,\leq) \\ \{\alpha\} & (4,\leq) & (3,\leq) & (0,\leq) & (0,\leq) & (3,\leq) & (-1,\leq) \\ \{x,\alpha\} & (6,\leq) & (4,\leq) & (3,\leq) & (0,\leq) & (3,\leq) & (1,\leq) \\ \{2x\} & (4,\leq) & (3,\leq) & (4,\leq) & (1,\leq) & (0,\leq) & (2,\leq) \\ \{2\alpha\} & (8,\leq) & (7,\leq) & (4,\leq) & (3,\leq) & (6,\leq) & (0,\leq) \end{array}\right)$$

Rows and column of those matrices are labelled with expressions involving clock x and parameter α. Thus, the vector `variables_defs` *will have form:*

$$\texttt{variables\_defs} = [x, \alpha]$$

Since only x is a clock, the `clocks` *vector will look in the following way:*

$$\texttt{clocks} = [\mathbf{true}, \mathbf{false}]$$

The labels of rows and columns of EDBM and LDBM will be defined using vectors of two integers. For example the first row, labelled with $\emptyset$ will be represented by vector $[0,0]$. The row labelled with $\{x,\alpha\}$ will be labelled by $[1,1]$. The entire definitions of edbm and ldbm rows labels are as follows:

$$\texttt{edbm\_row\_definition} = \big[[0,0],[1,0],[0,1],[1,1]\big]$$

$$\texttt{ldbm\_row\_definition} = \big[[0,0],[1,0],[0,1],[1,1],[2,0],[0,2]\big]$$

The attribute `values` *stores bounds for all expressions bounded by LDBM. Those expressions are represented by vectors of integers that are obtained by subtracting vector labelling column of LDBM from vector that labells row of LDBM. In this example, the attribute* `values` *will have following form:*

$$values = \begin{pmatrix} [0,0] & \rightarrow & (0,\leq) \\ [-1,0] & \rightarrow & (0,\leq) \\ [0,-1] & \rightarrow & (-1,\leq) \\ [-1,-1] & \rightarrow & (-2,\leq) \\ [1,0] & \rightarrow & (1,\leq) \\ [1,-1] & \rightarrow & (1,\leq) \\ [1,-2] & \rightarrow & (0,\leq) \\ [0,1] & \rightarrow & (4,\leq) \\ [-1,1] & \rightarrow & (3,\leq) \\ [-2,1] & \rightarrow & (3,\leq) \\ [1,1] & \rightarrow & (6,\leq) \\ [2,-1] & \rightarrow & (0,\leq) \\ [-1,2] & \rightarrow & (7,\leq) \end{pmatrix}$$

Thus, with this implementation approach, instead storing 36 values of LDBM matrix it is enough to store only 13.

Methods of the EDBM class

Constructors The EDBM class is equipped with set of constructors that may be used depending on how many information about the constructed EDBM is available. The basic form of the constructor takes three arguments that are a vector that contains labels of clocks and parameters used for labelling rows and labels of EDBM, a vector of booleans indicating which of the elements of the first argument are clocks and a definition of EDBM's row. Its declaration has following form:

```
EDBM(const std::vector<std::string>, const std::vector<bool>,
     const RowDefinition);
```

The function calculates form of LDBM that is a closure of this EDBM and fills it with $(0, \leq)$ bounds.

Since calculating form of LDBM that is closure of given EDBM is very time consuming process, the class EDBM provides additional constructor that uses LDBM provided by user, so once calculated closure can be used many times. This constructor has following declaration:

```
EDBM(const std::vector<std::string>, const std::vector<bool>,
     const RowDefinition, const RowDefinition);
```

Note, that the correctness of the closure is not verified, so the results of system analysis may be misleading when wrong closure is provided.

Apart from those two constructors, the class provides standard copy constructor:

```
EDBM(const EDBM&);
```

Getter and setter methods The class EDBM provides two getter methods for its elements:

```
Bound getBound(const unsigned int row, const unsigned int column) const;
Bound getBound(const Expression expr) const;
```

The first one takes indexes of row and column of the required bound. The second as argument takes the expression that is bounded by required bound. This expression is represented by vector of integers, where each one determines multiplicity of subsequent variables in the expression (like in `RowDefninition`). The methods access the `values` table and provide the required bound already multiplied with the equivalence factor. Note that the getter method may return all bounds of the LDBM defined as closure of the EDBM that the class represents.

The setter methods are analogical to getter methods, and are declared by:

```
Bound setBound(const unsigned int row, const unsigned int column);
Bound setBound(const Expression expr);
```

Those methods update the `values` table considering the equivalence factor of updated bound. Oppositely to getter methods it is only possible to set bounds that belong to EDBM. Requiring change of the bounds that are elements of LDBM and are not in EDBM results in run-time error. Setting values of bounds is not followed by canonicalization of the EDBM.

Apart from the methods described above the class provides diagnostic methods to get the size and form of EDBM and LDBM matrices:

```
unsigned int getEdbmSize() const;
RowDefinition getEdbmRowDefinition() const;
unsigned int getLdbmSize() const;
RowDefinition getLdbmRowDefinition() const;
```

Property checking methods The property checking methods implement algorithms defined in 6.3.1. They are declared in the following way:

```
bool empty() const;
bool includes(const EDBM&) const;
bool satisfies(const Expr&, const Bound&) const;
bool satisfies(const int, const int, const Bound&) const;
```

The operation `empty()` checks whether values on a diagonal (that are represented by a single bound in the `values` table) are positive. The operation `includes(EDBM)` checks whether all bounds in EDBM provided as the argument are lower than corresponding bounds in the `values` table. The operation `satisfies` test satisfiability of individual bounds. There are two overloaded versions of this function: one takes requested expression as argument while second takes indexes of the tested bound in the matrix.

Transformation methods The transformation methods implement operation of canonicalization and operations from section 6.3.2. Canonicalization is implemented by the method declared as:

```
void canonicalize();
```

and is done by minimizing the LDBM. Due to the internal structure of the `EDBM` class the cost of this operation is $\mathcal{O}(n^3)$, where n is the length of the LDBM matrix (number of its rows).

Intersection and the $and_E(M, \sum A - \sum B \prec m)$ operations are implemented by following methods:

```
bool intersect(const EDBM);
bool and(const Expression, const Bound);
bool and(const unsigned int, const unsigned int, const Bound);
```

All those functions perform re-canonicalization at the end and returns an indicator whether the resulting matrix is not empty. The two versions of the `and()` function a arguments take either the expression that is to be restricted or indexes of the restricted element.

Forward and backward time elapse is implemented by following functions:

```
void future();
void past();
```

They are implemented according to Algorithm 6.9 and Algorithm 6.10 respectively. Their complexity is linear with respect to a size of the `values` table.

The forward and backward clock resets are implemented by methods:

```
void reset(const std::vector<bool>);
void unreset(const std::vector<bool>);
```

The argument of those function is a vector that indicates positions of reset or unreset clocks in the `variables_defs` vector. These vectors may point only those elements of `variables_defs` that are clocks. Otherwise the method will generate run-time error. The complexity of `reset()` is linear in terms of size of `ldbm_row_definition`. Unreset is implemented with $\mathcal{O}(n^2)$ cost, where n is length of the LDBM (because of involved $and()$ operation).

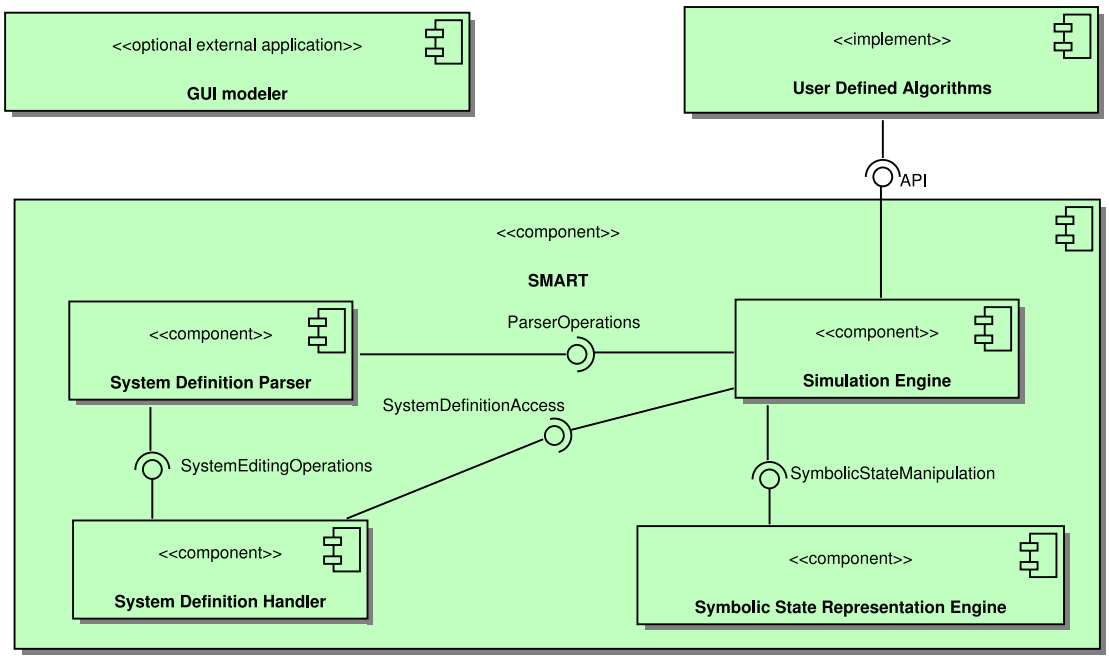


Figure 35: SMART architecture

7.2 The SMART tool

SMART (Simulation Modeling and Analysis of Real Time systems) is an open platform implemented in C++ for modeling and analysis of real time systems using Communicating Systems of TIOA extended with variables and parameters. It has layered and modular structure, what provides freedom in implementing methods for system manipulation.

The overall structure of SMART is presented in the Figure 35. The tool is implemented as a library that provides an API for definition of algorithms for model checking, test generation etc. Each of the module of the tool may be replaced by user defined module as long as it provides required interface.

The tool provides a parser in the module *System Definition Parser* that reads systems definition provided as a set of text files. This text file may be created manually, or obtained using some graphical editing tool e.g. CALIFE [76]. CALIFE is an open source platform that can be used to interface tools working on automata. The CALIFE platform works on several automata models (transition systems, timed automata, counter automata,...) and allows to define new models and interface new tools. A timed-automata system modelled under the CALIFE System Editor can be exported to a XML format that can be parsed

by SMART. The parsing operations are available via the *ParserOperations* interface.

However, since the CALIFE cannot be used to model nested communicating systems, it has been decided to create new system description language, that allows to model nested structures of automata. Text files with description of communicating system are parsed by the System Definition Parser component of the SMART platform.

During an analysis, a system definition is stored by the *System Definition* component. This component provides definition of classes that represent modules of CS (automata and other CSs) and their communication rules (topologies). It provides interface for the System Definition Parser, so the system definition can be initialized at the beginning of the analysis. It also provides an interface for the Simulation Engine component, that allows accessing a system definition when needed.

The *Simulation Engine* component is the heart of the SMART platform. It is accessed by the user via API. The API provides operations that can be used for definition of user's algorithm for system analysis. This includes checking currently available transitions, performing forward or backward synchronizations by *post*() and *pred*() operations or taking system's state snapshot. The API allows the system to be returned to any previous state (by storing an execution tree) or to move to any arbitrary state by providing a system state snapshot. The simulation engine accesses *SystemDefinitionAccess* methods provided by the System Definition component. It also uses the *SymbolicStateManipulation* interface provided by the component Symbolic State Representation Engine.

The next component is the *Symbolic State Representation Engine*. This is collection of classes that store the actual system's symbolic state. The main part of it is the implementation of the EDBM framework. Apart from it, the component stores the vector of occupied locations by the processes and values of variables. The component provides the SymbolicStateManipulation interface to the Simulation Engine that defines methods for symbolic operations on polyhedra ($future$, $past$, $reset$, and etc.). It also provides methods for snapshoting the system's state and setting it to any value.

The last component that is part of the platform is *Global Definitions* component. It is set of classes, macros and global definitions that are used by the entire tool.

The *User Defined Algorithms* component is not an integral part of the SMART platform, but set of algorithms implemented by the user using the SMART's API. The algorithms

may be compiled either to executable files to a library that is used for other algorithms of higher level.

7.2.1 Global Definitions

This package contains definitions of classes and types that are used by all other modules in the tool. Two most important of these classes are Variable class and AlgebraicExpression. Apart from those two, the Global Definitions package contains also definitions of classes that store information about system that is exchanged between other packages.

Variable

There are five types of variables that can be used in SMART. Two of them represent primitive types: Int and Double, two of them represent associative containers: Vector and Stack and finally one type represents user defined structure: Struct, that is collection of variables of all of the types (but not other structures). Variables in SMART are implemented by `Variable` class. This is an abstract class, from which classes for each of five types inherit. The inheritance diagram for the class `Variable` is depicted in the Figure 36.

The base class defines function that allows checking the variable's type. It also declares the abstract functions for duplicate and getting the value of variable.

The primitive types `Double` and `Int` represent real and integer variables respectively. Their classes define basic binary operations: sum, difference, multiplication and division plus all comparison operators.

`Vector` is a type that allows storing an array of variables of the same type (Int or Double). Access to vector's elements is done via providing index of requested variable in square brackets. A vector can be extended with new elements by using the `push()` operation and providing a value of the new element. The new element will be added to the end of the vector. By operation `pop()` it is possible to remove last element of the vector. The operation `size()` returns the length of the vector. Finally `clear()` removes all vector's elements.

The `Stack` class provides access only to the element that was recently put on it. Initially it is empty. Putting a new element on a stack is done by `push()` operator. An element on

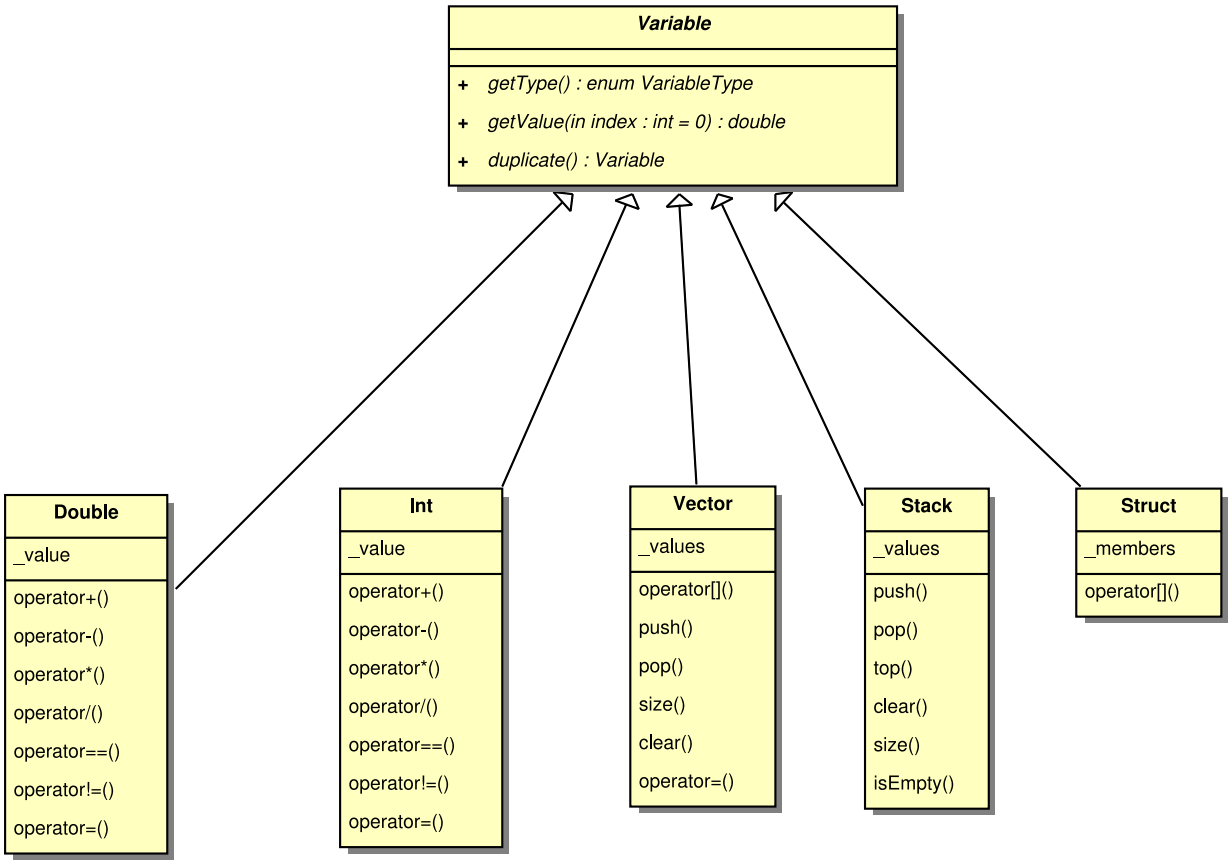


Figure 36: Inheritance diagram of the `Variable` class

a top of a stack is removed by operator `pop()`. The operation `top()` returns reference to the element on the top (without removing it). `clear()` and `size()` work analogously to the operations defined for Vector. The operation `isEmpty()` returns **true** if the size of the stack is 0.

Finally, the `Struct` class is implemented as a dictionary, where the key is member label and the value is a object of Variable type. The access to members of a struct is done by providing the string representing member's label in square brackets.

AlgebraicExpression

Algebraic expressions are mostly used to define invariants and guards in the system locations and transitions. Sometimes they may also be used to define an index of an element of a vector that is updated during a transition. The SMART tool can recognize and store any expression composed with labels and constants separated by one of operators for sum, difference, multiplication and division. Elements of expression may be grouped with parentheses. The strings `Inf` represents infinity, so it may not be used as a label of a variable in an expression. The example below can be considered as algebraic expression for SMART:

```
2 * x + (1 - x * x) + y
```

however the following cannot:

```
2x + 1
```

Expressions are read from right to left, without prioritizing operators, so for example the evaluation of this expression:

```
3 * 2 + 2
```

would be 12 instead 8. This drawback can be overtaken by using parentheses, for example the expression

```
(3 * 2) + 2
```

will be evaluated to 8.

The minus sign is recognized by SMART according to context, so it is possible to write:

```
-(1 + x) * y
```

instead

```
(-1 - x) * y
```

Expressions may be defined using elements of vectors. In this case the index of the referred element may be expressed as an expression as well, for example:

```
(2 * v[2]) + v[x + 2]
```

Finally, it is possible to use container size and the top element of a Stack variable as element of expression:

```
vec.size() + stack.top()
```

Algebraic expressions are implemented in SMART as a recursive binary tree of objects of `AlgebraicExpression` class. Main elements of the class declaration are presented below:

```
class AlgebraicExpression{
 public:
  AlgebraicExpression* operator+(const AlgebraicExpression*);
  AlgebraicExpression* operator-(const AlgebraicExpression*);
  AlgebraicExpression* operator*(const AlgebraicExpression*);
  AlgebraicExpression* operator/(const AlgebraicExpression*);
  bool operator==(const AlgebraicExpression*);
  bool operator!=(const AlgebraicExpression*);
  AlgebraicExpression reduce();
  ...

 protected:
  std::string main_element;
  AlgebraicExpression* index;
  AlgebraicExpression* child_left;
  AlgebraicExpression* child_right;
}
```

Leafs of trees representing expressions are objects representing numerical value or a label. In this case the `main_element` stores the value or the label as a string. In case when the label represents element of a vector variable, the index pointed by `index`. For more complex expressions, nodes that are not leafs contain a symbol of binary operator as `main_element` and the `left_child` and `right_child` pointers point to objects representing expressions

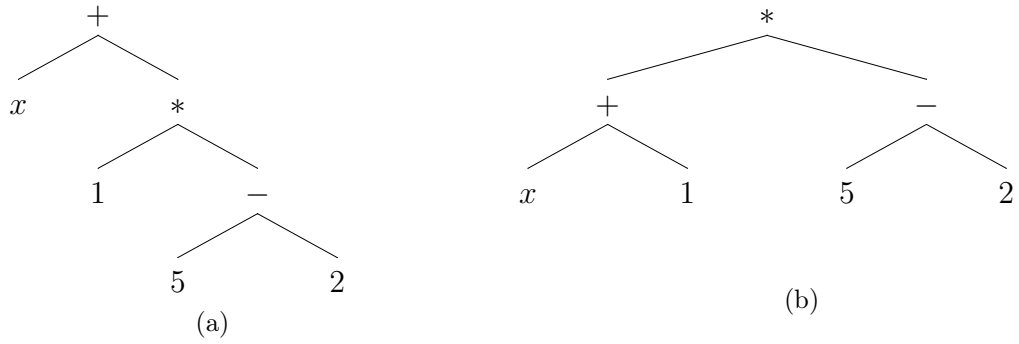


Figure 37: Trees representing algebraic expressions

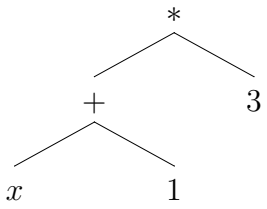


Figure 38: Reduced expression tree from the Figure 37(b)

at left and right side of the operand.

For example, following expression:

```
x + 1 * 5 - 2
```

would be represented by the tree in the Figure 37(a). The modified version of this expression:

```
(x + 1) * (5 - 2)
```

would be represented by the tree from the Figure 37(b).

The class AlgebraicExpression implements algebraic operators for sum, difference, multiplication and division that allow to merge two expressions.

An useful function of AlgebraicExpression class is *reduce*(), that allows reducing expressions by evaluating its numerical parts. For example, the expression `(x + 1) * (5 - 2)` represented by the tree in the Figure 37(b) can be reduced to `(x+1) * 3` that corresponds to the tree in the Figure 38.

The reduction of expression tree is done in three phases:

1. **Numerical reduction**: All sub-expressions that contain only operations on numeric values are resolved. The result replaces the node that previously represented the operation.
2. **Tree reduction**: This phase reduces all numeric leafs of expression tree that are connected by the same type of operations (additive or multiplicative). For example, the expression `(1 + (a + 1))` can be reduced to `(2 + a)`. Actually this phase reduces the expression to `(2 + (a + 0))`. The result of reducing operation is stored in left operand and right is replaced by 0 (for additive operations) or 1 (for multiplicative operations). So for another example, expression `(16 * (a / 4))` will be replaced by `(4 * (a / 1))`.
3. **Tree restructuring**: This phase restructures the expression tree by removing neutral operations (0 for additive and 1 for multiplicative operations). For example the operation `(1 * (a + b))` would be replaced by `(a + b)`. For operations '−' and '/' the tree is reduced only if the neutral operand is "right standing", e.g. the expression `(0 - (a + b))` will not be reduced.

The procedure is recursive, so it proceeds until no sub-tree can be reduced any more.

7.2.2 Parser

The parser of SMART is implemented by `Parser` class using the lex/yacc tandem, or rather their GNU Open Source versions *flex* and *bison*.

flex is a tool for generating scanners. A scanner is a program which recognizes lexical patterns in text. The flex program reads the given input files for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. flex generates as output a C source file, lex.yy.c by default, which defines a routine *yylex*(). This file can be compiled and linked with the flex runtime library to produce an executable. When the executable is run, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code [78].

GNU bison is a parser generator that is part of the GNU project. Bison converts a grammar description for a context-free grammar into a C or C++ program which can parse a sequence of tokens that conforms to that grammar (a LALR parser). It can also produce "Generalized Left-to-right Rightmost" (GLR) parsers for ambiguous grammars [2].

The Parser module reads the set of input text files with system definition and using the set of constructors of System Definition module sets the appropriate system definition. The module provides an interface to the Simulation Engine that allows requesting parsing the system.

This interface contains one public method: `parseSystem()`. The method takes one argument that is filename of the file containing the top-level system description, i.e. the description of the CS at the highest topological level. Default name for such a file is "system.sys". The method reads the input file and builds the system definition using the interface provided by the System Definition Handler component.

7.2.3 System Definition

The system definition module stores the definition of system and its processes. It provides interface for Simulation Engine module that allows accessing system and its components definition.

The basic class of the system hierarchy is `Module`. This is an abstract class that can be instantiated as `System` or `Automaton`. The simplified inheritance class diagram for the `Module` class is depicted in the Figure 39. Each instantiation of `Module` class contains list of module clocks, variables and parameters. In case when the module is instantiated as object of System class, those elements are global to all its submodules. The class provides also getter methods to access the lists. Another method of the class is a function that returns type of the module.

The instantiation of `System` class contains list of modules that constitute the system and definition of topology of communication of the system's modules.

The top-level module of a system must be instantiation of `System` class, so even if a system contains only a single process, it must be encapsulated within System. The methods `getForwardSync()` and `getBackwardSync()` return list of potentially possible synchronizations of the system's modules that can be performed forward or backward respectively. The argument of those functions is a vector of locations occupied by system's modules. The returned list of possible synchronizations contains all transitions that can be performed forward or backward basing only on topological rules and not on satisfying guard conditions.

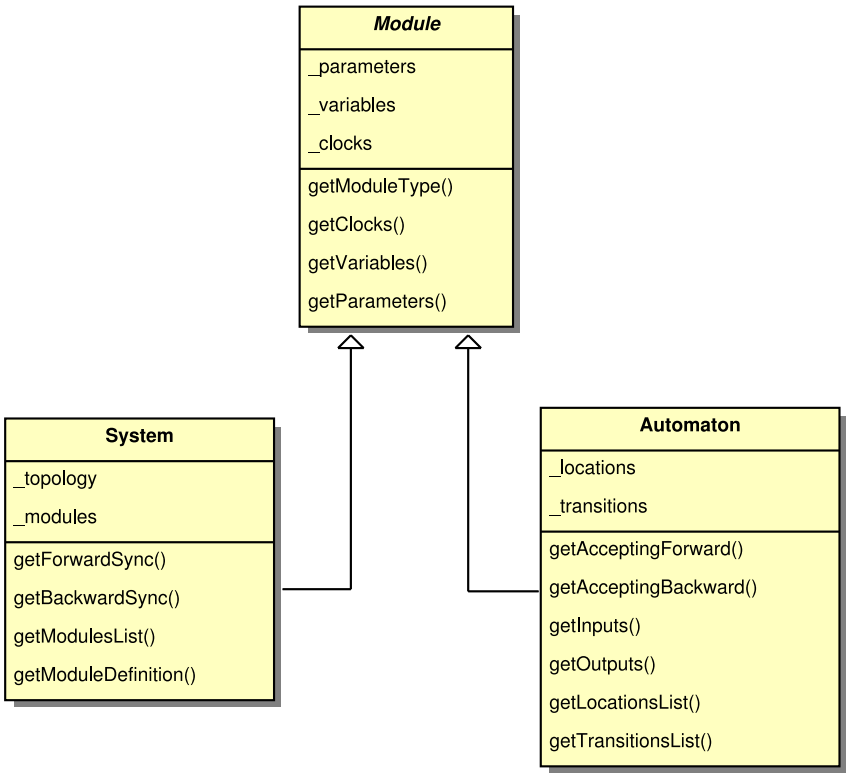


Figure 39: Inheritance diagram of the `Module` class

The `System` class defines also a getter function to its modules. The returned value of this method is implemented as a map where a key is defined by a label of system's submodule and a value is a pointer to the definition of its type. The last public method provided by System class is `getModuleDefinition()`. This function takes label of the module type as argument and return the pointer to the module definition.

The `Automaton` class contains lists of locations and transitions of defined automaton plus getter mathods for these lists. The functions `getInputs()` and `getOutputs()` provide lists of input and output events respectively. The functions `getIncomingTransitions()` and `getOutgoingTransitions()` return lists of transitions that enter or leave the location, which id has been provided as argument. Finally, the functions `getAcceptingForward()` and `getAcceptingBackward()` as arguments take location id and event label and return list of transition that respectively leave or enter given location and are associated with given event.

For System Parser, the classes provide set of constructors that can be used to instantiate system definition.

7.2.4 Simulation Engine

The Simulation Engine is the module directly accessed by the user via API. The API is a set of functions that allow user to perform forward and backward system analysis, simulation etc. It also provides useful functions that allow moving the system to any state (without checking whether this state is really reachable). The system evaluation history is permanently stored and available to user, so the system may be turned back to any state that was occupied in the past at any moment.

The top class of the Simulation Engine package is the `SystemEngine` class. This class has following public interface:

```
class SystemEngine{
 public:
  SystemEngine(const std::string);
  const System& getSystemDefinition() const;
  const SystemState& getCurrentState() const;
  const std::list<Sync> getPossibleForwardSync() const;
```

```
  const std::list<Sync> getPossibleBackwardSync() const;
  const SystemState& synchronizeForward(const Sync&);
  const SystemState& synchronizeBackward(const Sync&);
  const SystemState& reset(const unsigned int);
  const SystemState& reset();
  const SystemState& stepBack(const unsigned int);
  void goToState(const SystemState&);
  SystemHistory& getSystemHistory();
 protected:
  SystemState current_state;
  SystemHistory history;
  ...
}
```

Its constructor method as the argument takes a string that is a link to the file containing definition of the top-level system module. The system is then instantiated and initialized to its initial state, where all clocks are at 0, all variables have their initial value and allowed ranges for all parameters are defined in the input file.

The class contains attribute `current_state` that contains actual location of all the system's modules, actual values of all variables and an EDBM defining relations between all clocks and parameters. There is also getter method defined for the attribute. The attribute `history` is implemented as a structure that contains system's initial state and a stack of performed operations and subsequently occupied states. The class `SystemEngine` defines a getter method for this attribute as well.

The method `getSystemDefinition()` returns a reference to definition of the top-level system module. Using the interfaces of Module, System and Automaton classes, the user can fully explore the system definition. The method `getCurrentState()` returns the actual system state, that is actual location of all the modules, actual values of all variables and an EDBM defining relations between all clocks and parameters.

The methods `getPossibleForwardSync()` and `getPossibleBackwardSync()` returns lists of enabled forward and backward synchronization definitions according to currently occupied state. The synchronization definition has form of a map, where a key is defined by unique id identifying each of system's module and the value defines transition that the

module performs for this synchronization.

The functions `synchronizeForward()` and `synchronizeBackward()` as arguments take elements of a list returned by function `getPossibleForwardSync()` or `getPossibleBackwardSync()`. It changes the value of `current_state` and logs this operation in the `history` variable.

There are two functions `reset()` defined. First one as the argument takes id of a system's module (it can be either system or a process) and performs reset of this module. If the reset module is a system then all of its sub-modules are reset. This operation is logged in the system's history. The other function `reset()` does not take any argument. It brings the system back to its initial state and clears the history.

The function `stepBack()` as argument takes an integer n that denotes a number of performed synchronizations that will be cancelled. The operation brings system to a state that was occupied n synchronizations before. The top n synchronizations are removed from the `history` attribute. The function `goToState()` moves the system to a state provided in argument without checking whether the requested state is reachable. It is logged in the history as a special transition.

7.2.5 Symbolic State Handler

The module is implemented by SymbolicState class. It handles the symbolic state and implements all operations that are performed for symbolic system analysis. It contains instantiation of EDBM that stores relations between all systems clocks and parameters and implements operations for forward and backward analysis.

Apart from instantiation of EDBM it contains a list of all system variables with current values and list of currently occupied location by all of the modules in the system. Both of these lists may be accessed by SimulationEngine via getter and setter methods.

7.3 SMART input files

In general there are two types of files used for system description: files describing timed automata, as basic component of the communicating system and files describing communicating systems itself - either as a top level structure of the system or as a nested component of another system.

7.3.1 Automata description

Files that contain description of a single automaton have following structure:

```
AUTOMATON label{
//the order of three first sections is not strict; they are also optional

  VARIABLES:
    ...  //list of local variables
  CLOCKS:
    ... //list of local clocks
  PARAMETERS
    ... //list of local parameters

  STATES:
    ... //list of locations of automaton
  TRANSITIONS:
    ... //list of transitions
}
```

The file must begin with a keyword `AUTOMATON` followed by a label of the automaton. The label of automaton must be unique within whole system. It is used in the system description file as a type. So if a system contains many processes of the same type, the type must be defined only once. The label is a string that contains letters, digits and the underscore sign ("_"). It must not start with a digit.

The actual definition of the automaton is contained within brackets. It consists of five sections of which three are optional. These three optional sections define local clocks,

variables and parameters. The order of them is not strict, however they must appear before last two sections which define locations and transitions of automaton. These sections are obligatory and must appear after definition of clocks, variables and parameters. The section with states must appear before the section with transitions.

Clocks

The section with clocks declarations begins with a keyword `CLOCKS` followed by a colon. Then labels of all local clocks of the automaton are listed using a comma as a separator. The last clock is followed by a semicolon, for example:

```
CLOCKS: clock1, clock2, clock3;
```

The labels of clocks do not have to be unique in the system, however it is not recommended to use labels that have been used for global clock that may be accessed by instantiation of the automaton. In this case the label will refer to the local clock, so accessing the global clock with the same name will not be possible.

Variables

The section with variables definitions begins with keyword `VARIABLES` and a colon. To define a variable of a basic type (Int or Double) it is necessary to declare the type (with a keyword `Int` or `Double`) and then define a label or labels separated with comma. Last label must be followed by a semicolon. After each label, the initial value must be provided in parentheses, for example:

```
Int int1(0), int2(5);
Double double1(0), double2(-3.14);
Double double3(2.71);
```

To declare a vector it is necessary to use a type keyword `Vector` and a label that is followed with a type of elements of the vector in angle brackets, then initial size of the vector in square brackets andfinally the initial value of vector elements in parentheses, for example:

```
Vector v1<Int>[5](0), v2<Double>[3](2.71);
Vector v3>Int>[2](1);
```

Stack is declared by keyword `Stack`, the label of the stack and the type of stack variable in angle brackets. Since the initial size of a stack is 0 it is not necessary to initialize it, for example:

```
Stack s1<Int>, s2<Double>;
Stack s3<Double>;
```

Finally, structures are defined with keyword `Struct`. The label of the struct is followed by its definition in brackets. The definition of a structure has the same grammar than definition of a variables section. The labels of structure members may collide with names of other variables. That is because structures define their own namespaces. For example:

```
Struct str1{Int i(0); Double d(23.1);};
Struct str2{Int i1(0), i2(1);}, str3{Double d(1); Int i1(0); Int i2(4);};
Struct str3{Vector v<Int>[3](0); Int i2(1);};
```

Parameters

Section containing parameters begin with a keyword `PARAMETERS` and a colon. After that, all parameters are listed with the allowed range in square brackets. Declaring a parameter's type is not necessary, since parameters may be only real. Parameters are separated by commas and the last parameter is followed by semicolon, for example:

```
PARAMETERS: max_delay[1,50], window_size[5,20];
```

Definition of locations

Each location is declared in separate line. A declaration of a location is contained within angle bracket. It has following grammar:

```
<location_id, label, {invariant}, {attributes}>
```

The only obligatory element is `location_id`. All other elements are optional. They can be skipped by replacing them with `#` or by not listing them. Leaving empty brackets for definition of invariant or attributes semantically also means skipping this element.

The `location_id` is a integer number, unique among all locations of this automaton. The location with id 0 is the initial location of the automaton. Label of a location is a string defined in the same way that other labels (clocks, variables etc.). Because locations are identified by their ids, labels do not have to be unique. It is also not necessary to define labels for each location – putting `#` in the place of label means that the location is not labelled.

Invariants are defined by set of constraints on clocks separated by commas. Each constraint has following form:

```
expression lower_bound_limit min_value , max_value upper_bound_limit
```

The `expression` is defined by a difference of two sums of clocks, parameters and real values. It is possible to use parentheses to define `expression`. The bounds limits can be chosen from "`[`" or "`(`" for lower limit and "`]`" or "`)`" for upper limit. Square brackets mean closed bounds and parentheses mean open bound. `min_value` and `max_value` define bounds of the range defining the invariant. They can be any algebraic expressions defined with variables and real numbers (plus `Inf` and `-Inf` denoting plus and minus infinity). Expressions used as lower and upper limits must be evaluable to a value at any time of the simulation, so they may contain neither clocks nor parameters.

There are special methods for accessing non-primitive types of variables, i.e. vectors stacks and structures.

The elements of vector type may be accessed by giving the position in square parentheses. It is also possible to refer to a vector element by an expression, for example: `vector_label[1]` or `vector_label[integer_label1+(2*integer_label2)]`. It is also possible to get size of vector by using method `size()`, for example `vector_label.size()`. The top element of a stack may be called using *top*() function, e.g. `stack_label.top()`. Also, it is possible to use size of a stack as element of algebraic expression (e.g. `stack_label.size()`). Elements of structures are called by using the label of the structure followed by label of requested member in square brackets (e.g. `struct_label[member_label]`).

Finally, list of attributes is just a list of labels assigned to the location. Two reserved keywords for attributes are `COMMITTED` and `URGENT` that define committed and urgent locations (see Section 3.2). Other attributes do not semantically impact the automaton, however they may be used for algorithms definition, etc.

An example state definition may look as shown below:

```
< 1, state1,{x[0,20], (x+y)-par_1[-Inf,10]}, {COMMITTED, dont_cover} >
```

Definition of transitions

Transitions, similarly to locations are defined one per line. The definition is also contained within angle brackets. In general a grammar for transition definition looks as follows:

```
< src_id, dest_id, event, {guard}, {updates}, {attributes} >
```

The `src_id` and `dest_id` define ids of source and destination locations of the transition respectively. `event` is a label of event associated with the transition. Output and input events begin with `!` and `?` respectively. The internal events are noted by $\sim$.

Transition guards are defined as list of constraints separated by commas. A constraint have general form as it is shown below:

```
expression lower_bound_limit min_value , max_value upper_bound_limit
```

There are two allowed forms of `expression`. It may be difference of two sums of clocks, parameters and real numbers, as in case of location invariants, or it may be any algebraic expression defined with variables and reals. `lower_bound_limit`, `min_value`, `max_value` and `upper_bound_limit` have the same meaning that for specifying invariants.

`updates` is the list of updates performed during execution of the transition separated by commas. Each update has a form:

```
updated_element (new_value)
```

`updated_element` refers to the element that is updated. It may be a label of a clock or variable, or it may be specification of action performed on container (vector or stack). If `updated_element` is a label of a clock, the `new_value` in parentheses may be only 0 (reseting is the only operation allowed on clocks). If it is a variable, then `new_value` may be any algebraic expression evaluated to a value (containing neither parameters nor clocks). When `updated_element` is an element of a vector, its index must follow vectors label in square brackets (index may be expressed as any evaluable algebraic expression),

e.g. `label[1]` or `label[label.size() - 1]`. When it is an element of struct, the label of the element must follow the label of struct in square brackets, e.g. `struct[element]`.

Additionally there are possible actions defined for containers like vector or stack. It is possible to add new value to end of a vector or top of stack (`label.push(expr)`, where `expr` is evaluable algebraic expression), remove the top value from stack or vector (`label.pop()`) or remove all values from a stack `label.clear()`.

An example update fragment for a transition may look as follows:

```
{clock1(0), var1(var1+1), vect.push(stack1.top()), stack1.pop()}
```

Finally, the list of attributes is defined by list of labels that may not be words `URGENT` or `COMMITTED`. Labels do not change semantics of the transition and may freely used by user.

A simple transition may have following form:

```
<0, 1, !begin, {x[0,5], x-(y+a)(2, 2+var1)}, {x(0), var1(var1-1)}, {BLUE}>
```

7.3.2 System description

The grammar of the description of the system of the highest level does not differ for description of its sub-systems. Thanks to this it is possible to analyze stand-alone communicating system and then to analyze the same system in some wider context.

The general structure of file containing system description is following:

```
SYSTEM{
   GLOBAL_CLOCKS:
     ... //optional list of global clocks
   GLOBAL_VARIABLES:
     ...  //optional list of global variables
   GLOBAL_PARAMETERS
     ... //optional list of global parameters
   TYPES:
     ... //declarations of types of instantiated modules
   MODULES:
```

```
    ... //list instantiated modules of the types declared above
  TOPOLOGY:
   ...   //definition of communication rules for system's modules
}
```

The first three sections define clocks, variables and parameters that are global for the automaton. They are optional and the order of them is not strict. The grammar of this sections is exactly the same as grammar for analogical sections in automata description. Labels of global clocks, variables or parameters defined at this level may be covered by the same label defined for one of its modules. For example, if a system defines global clock x and one of it subsystems defines a clock with the same name, any operation on the clock x by given subsystem will concern its local clock.

Declaration of modules' types

The section TYPES contains paths to files with definitions of types of system's modules. Each path is followed by a semicolon, e.g.:

```
TYPES:
  ./subsystem1.sys;
  ./subsystem2.sys;
  ./automaton1.aut;
```

Definition of modules

Section MODULES defines instantiations of system's modules. Modules may be of a type defined in one of files provided in the section TYPES. A module is instantiated by providing a label of its type followed by a colon and listing labels of the instantiations separated by commas. The last label is followed by a semicolon:

```
MODULES:
  System1: sys1, sys2;
  AutomatonType1: aut1, aut2;
  AutomatonType1: aut3, aut4;
```

It is possible to define a concrete value of a parameter of an instantiated module by providing this value in parentheses after a label of the module. In this case the parameter has fixed value and can be used as variables in evaluable expressions. It is possible to concretize parameters of only some instantiations of the same type, leaving the rest undefined, for example:

```
MODULES:
  System1: sys1(par1 = 5), sys2;
  AutomatonType1: aut1(par2 = 1, par3 = 0), aut2(par2 = 2);
  AutomatonType1: aut3, aut4;
```

Definition of communication topology

In the current version of SMART it is possible to define only static topologies. A topology may be defined manually or automatically. To manually define communication topology, all communication channels must be listed in the file with system description. Automatic topology is constructed by the system parser using labels of events associated with transitions of system's modules.

Manual topology definition To define a topology manually, the keyword TOPOLOGY must be followed by a keyword MANUAL and a colon. Then all communication channels must be defined. Channels are defined by associating output events of system modules with input events of other modules. The general structure of manual topology definition is as follows:

```
TOPOLOGY MANUAL:
  module_label_1:
    !output_event->input_events->visibility;
  ...
  module_label_n:
    !output_event->input_events->visibility;
```

The `module_label_i` declares a label of the module for which the rules are defined. `!output_event` is a label of output event of this module. `input_events` is a list of input events of other modules separated with commas, associated with given output event of module labelled with `module_label_i`. Such an input event is defined by label of module, a question mark and label of the input event. `visibility` is an optional label of event (input or output) that denotes how the synchronization is seen from outside of this system. Visible transitions create system's interface and are subject of topological rules of systems of higher levels, unless this system is on the top-level of the defined hierarchy.

Additionally there are rules created for non-synchronizing transitions of single modules. The visibility of such transitions is intuitive. Transitions associated with internal event τ are not seen outside the system. Transitions associated with interface event are seen under label of given event.

A simple manual communication topology definition could look as follows:

```
TOPOLOGY MANUAL:
  module_1:
    !event_1->module_2.?event_1->!event_1;
    !event_1->module_2.?event_1, module_3.event_1->!event_1;
    !event_2->module_2.?event_2,module_3.?something_else;
   module_2:
    !event_3->module_1.?event_3;
   module_3:
    !some_event->module1.?some_event->?some_other_event;
```

Such topology would define following synchronization rules:

- module_1 performs transition associated with !*event*_1, while module_2 performs transition associated with ?*event*_1. This synchronization is seen outside the system as output !*event*_1.
- module_1 performs transition associated with !*event*_1, while module_2 andmodule_3 perform transitions associated with ?*event*_1. Outside the system this is seen as output event !*event*_1.
- module_1 performs transition associated with !*event*_2, while module_2 performs transition associated with ?*event*_2 and module_3 performs transition associated

with event ?*something_else*. This synchronization is not seen from outside of the system.

- module_2 performs transition associated with event *event_3* while module_2 performs transition associated with ?*event_3*. This synchronization is not seen outside the system.
- module_3 performs transition associated with event !*some_event* while module_2 performs transition associated with ?*some_event*. Outside the system this is seen as input event ?*some_other_event*

Automatic topology definition There are two possible options for automatic topology definition: unicast or broadcast.

An unicast topology means that there will be separate channel defined for each pair of input and output events with the same label. For example if there is one module (say A) that has transition associated with event $!e$ and four modules (say $B1$, $B2$, $B3$ and $B4$) with transitions associated with event $?e$, there will be four communication channels defined, corresponding to following definition:

```
A.!e->B1.?e;
A.!e->B2.?e;
A.!e->B3.?e;
A.!e->B4.?e;
```

When a module A performs transition associated with $!e$, only one of the four other modules will synchronize with it by executing transition associated with $?e$.

This is different in case of broadcast topology. This time only one channel will be created: when module A performs transition associated with $!e$, the four other modules will synchronize with it by executing their transitions associated with $?e$. However, this synchronization may be performed only when it is enabled for all modules, i.e. there is a transition leaving current locations of all modules associated with $?e$ and the guard conditions of those transitions are satisfied. This corresponds to following definition:

```
A.!e->B1.?e, B2.?e, B3.?e, B4.?e;
```

In both cases, synchronization transition are not seen from outside of the system, i.e. there is no event associated with synchronization that is emitted to or accepted from outside of the system.

An automatically created communication topology is defined by putting in the system description files what follows:

```
TOPOLOGY AUTOMATIC: UNICAST
```

in case of unicast topology, or

```
TOPOLOGY AUTOMATIC: BROADCAST
```

for broadcast topology.

7.4 Generating test cases with SMART

7.4.1 Test selection using coloring coverage criterion

One way to deal with the explosion of the number of test cases is to consider test purposes which are properties to be checked in the tested system [77, 61, 56]. In other approach, a tester may also adopt a strategy to derive test cases according to a defined coverage criteria, what is described in details in [67]. This approach increase the user confidence in the system being tested. A coverage criterion may deal with locations, transitions, variables and paths. The choose of one of these criteria has a direct consequence on the corresponding generation algorithm. That implies that a testing tool offering, for example, locations and variables coverage implements one algorithm for each criterion which increases the complexity of the tool and its suitability for the final user. In [25], a new formalism, namely coloring coverage criterion and its corresponding generation methodology was introduced.

Coloring coverage criterion is defined using *coloring function*. This is a function that assigns to states and transitions of automata a pair (c, n) that consists of label named *color* and a natural number. The *friend* function for each used color associates a group of other used colors that are *friendly* for given color. A test suite TS is said to satisfy the coverage criterion for color c if:

- each element of system that is labelled with c is covered by a executable test case of TS,
- all colors covered by a single test case are friendly to each other
- if an element labelled with a pair (c_1, n_1) appears in a test case before an element labelled with (c_2, n_2) then $n_1 \leq n_2$.

Using coloring coverage criterion it is possible to express other kinds of coverage criteria, like location, transition or variable coverage. Also it is possible to define more sophisticated criteria, like *definition-use coverage* [71].

The coloring coverage criterion allows to extend the existing criteria and offers an unified model for coverage, allowing the reuse of algorithm for different coverage criteria.

7.4.2 Test generation algorithms

As a proof of concept three algorithms for generating test cases according to coloring coverage criterion have been implemented on the top of SMART platform, using its API. They use different search strategies for reachability space exploration and selection of execution trace to follow.

The jump algorithm

The *jump* algorithm explores the reachability tree to some user defined depth. Then choose from the tree a path that covers the biggest number of unvisited elements (locations or transitions) marked with desired color. If more than one path covers the same number of unvisited desired elements it chooses the one that "hits" a such an element earlier. Desired elements are those covered with a color that the test suite is to cover and having weight bigger or equal than the biggest weight in the set of already covered elements. If no new desired element can be hit within defined depth algorithm chooses one path according to user defined strategy. This strategy can be RANDOM (the random path is chosen) or FRESH (the path which transitions have been traversed the least number of times). After the path is chosen, the algorithm performs the jump - go through entire path at once, and marks all the new elements it covers as visited.

The walk algorithm

The algorithm *walk* is very similar to jump. It also chooses the optimal path with the same criteria then jump, however after the path is chosen it does not jump, but makes only the first step from the optimal path and marks a new desired element as visited if it was hit. Then the search operation is repeated. In theory generation of test suite according to this algorithm should take more time, however generated test suite should be shorter than suites generated by the previous algorithm (assuming the same depth).

The hit-or-jump algorithm

The Hit-Or-Jump algorithm was proposed in [28]. In this case the reachability tree is searched as long as the first uncovered desired element is found. If so the algorithm moves system to that state and mark element as hit. If the tree is explored (within the defined depth) without hit, the algorithm chooses a path with defined length (length equals the depth) according to the defined strategy (RANDOM or FRESH) and jumps over the path. The algorithm used here should create test suites relatively within the shortest period of time, but they should be theoretically the longest from test suites generated by all of the algorithms presented above.

7.5 Experiments

All of the algorithms described above have been compiled as a single executable file *testgen*. The optional arguments are (in this strict order): path to a file with top-level system description (default path is `./system.sys`) and path to a file with configuration parameters (by default `./testgen.cfg`). The configuration file defines following parameters:

- `ALGORITHM` – `WALK`, `JUMP` or `HOJ` (for Hit-Or-Jump).
- `DEPTH` – The depth of reachability tree constructed during the test case generation.
- `MAX_WITHOUT_HIT` – The number of system transitions that does not cover new desired element after which system is reset.
- `MAX_RESET` – Number of system resets after which the test generation is stopped, regardless whether the coverage criterion was satisfied.

- `STRATEGY` – `RANDOM` or `FRESH`

The case study was done on the Media Access Control (MAC) process of the FlexRay protocol [34]. FlexRay is a modern, high speed, data link layer protocol for in-car communication systems. It has been developed by the FlexRay consortium in the year 2004. The FlexRay MAC process combines static time division multiplexing with dynamic bandwidth allocation. The original specification of FlexRay is available as SDL charts, but with a few modifications it has been modeled as TA, such that its functionality is mapped 1:1 with original specification. The model of the process contains 29 locations, 55 transitions, 8 clocks, 9 parameters used in clock constraints and 3 integer variables. An EDBM constructed to handle this example had 18 rows and columns, its closuring LDBM had 39 rows and columns. The most complicated guard had form:

```
tDynSegOffset + gdMinislotActPointOffset - gdActPointOffset == 0
```

where `tDynSegOffset` was a clock and `gdMinislotActPointOffset` and `gdActPointOffset` were parameters. The rest of constraints had form $x \prec p$, such that x was a clock and p was a single parameter. Because of the clock constraints that have been defined using sum of a clock and a parameter, this example perfectly can illustrate the power of EDBM for processing parameterized systems.

As an experiment test suites covering all system's locations and transitions have been defined using all the test generation algorithms defined above. Additionally, they used different path selection strategies (FRESH or RANDOM) and search depth (1, 5 or 8). The parameters `MAX_RESET` and `MAX_WITHOUT_HIT` were defined to resp. 5 and 100. For all configurations required test suites have been produced 10 times to get average results. The Table 3 shows average results obtained for test generation.

The *algorithm* columns contains the algorithms that were used to generate tests. *depth* is the depth of reachability tree build at each step. *# test cases* is the number of test cases in the generated test suite. *total length* is the total number of steps (executed transitions) in all test cases in the generated test suite. *gen. time [s]* is the time that was spent for generating the test case in seconds. For the last three columns two results are given: obtained for RANDOM and FRESH strategy (respectively columns RND and FR). Finally, the last columns shows the obtained coverage of the test suite. In all the cases the algorithms were able to generate test suites covering entire specification.

algorithm	**depth**	**# test cases**		**total length**		**gen. time [s]**		**coverage [%]**
		RND	**FR**	**RND**	**FR**	**RND**	**FR**	
WALK	1	6	4.5	612.6	564	9.33	9	100
WALK	5	2	2	229.7	226.2	26	29.25	100
WALK	8	2	2	158	164.2	256	271.25	100
JUMP	1	5	5.33	561.6	537.6	9.66	12	100
JUMP	5	2.25	2.66	370	260.6	10	9	100
JUMP	8	2	2	226	231.7	29.75	25	100
HOJ	1	4.5	4.5	455.5	444.5	9	5.5	100
HOJ	5	2	2	383.5	340.5	6	4.5	100
HOJ	8	2.25	2	272	297.2	7.75	11.25	100

Table 3: Results of experiments with SMART and FlexRay protocol

As it could be expected the fastest algorithm is Hit-Or-Jump, however it generates the longest test suites. On the other hand, test suites generated by "walk" algorithm are even 50% shorter, but only when bigger search depth is considered. In this cases test generation may take even 27 times more than generation of test suites with Hit-Or-Jump algorithm using the same depth. The algorithm "jump" seems to be good compromise: it generates test cases 3–4 times longer than Hit-Or-Jump with the same search depth, however they are considerably shorter (up to 40%). It is noticeable that path selection strategy (RANDOM or FRESH) has no significant impact on the performance and length of generated tests.

8 Conclusions and future work

8.1 Conclusions

The aim of the thesis was to explore common data structures used for parametric verification of real timed systems modeled as timed automata and to propose new structure that facilitates symbolic verification of parameterized automata and increases expressiviness of operations over transitions of such automata. The proposed solution – Extended Difference Bound Matrix fulfills both of those requirements.

What distinguish EDBM from other structures used for parametric verification is that all the bounds stored in the structure are purely numerical. This feature highly facilitates all the operations that requires comparing two elements of the matrix. In the case of parameterized bounds that are always present in competitive solutions the result of comparison may be ambiguous what has consequences in a necessity of duplicating the structure to consider all possible cases, what results with high costs of operations in terms of time and memory consumption. Since elements of EDBM are always numerical bounds, results of the comparison are always unambiguous. As result, the size of the matrix is constant during entire analysis what facilitates and boosts all operations over the structure.

Another advantage of EDBM is that structurally it does not differ very much from the solutions that are currently implemented in model checking tools. In fact, migration from tools that use DBM (like UppAal) to EDBM requires only changing the way how the bounds are stored and slight modification of the operations over the structure. Since the bound representation of EDBM and standard DBM is the same, most of the code may remain unchanged. Also the interface of the EDBM class is derived from the standard interface of DBM, what can simplify making the tool engine work with the new structure.

The final advantage over other data structures is extended expressiviness in terms of forms

of expressions that can be bounded by the structure. Note, that only those kind of expressions may be used as transition guards in the automata modeling analyzed systems. The standard approach – Parametric DBM allows to store bounds of expressions that represent a single clock or difference of two clocks. The Parametric Hypercubes, a solution proposed in [63] extend set of expressions that can be bounded to sum of arbitrary number of clocks. The solution proposed in this thesis allows to bound even wider set of expressions: all expressions that are difference of two sums of clocks may be handled be EDBM.

Of course everything must have its price. Bounding more complicated expressions has also negative consequences. One of them is that form of all expressions that are used in transition guards or invariants of the analyzed automata must be known a priori to the analysis. It means that before the analysis may start, all the model must be scanned in order to find out what expressions are used in it. Other approaches do not require this.

Second drawback, more serious, is the necessity of calculating the closure of the EDBM used for the system analysis. The cost of calculating closure depends exponentially on the number of clocks and parameters and also on the complexity of used expressions. This complexity is expressed in a size of sets labelling rows and columns of EDBM, what is corresponding to number of clocks and parameters in expression guarding transitions of the automata. Fortunately, in practice, those expressions are not two complicated and usually have form of $x \prec p$, where x is a clock and p is a parameter. In such cases, there is no necessity of calculating the closure.

8.2 Future perspectives

Although in theory the superiority of EDBM over alternative solutions has been demonstrated, no practical comparison between EDBM and other structures has been done. This is mainly because due to lack of time and resources no mature implementation of a tool using EDBM has been built. The implementation of SMART was intended more as a proof-of-concept than as a tool with a practical merit. It is sure that the code of the tool could be much optimized and then compared to its competitive alternatives. Therefore a plan for the nearest future is to build an optimized and technically mature implementation that can fully use the advantages of EDBM over other structures.

Other improvement that can be done to the EDBM is speeding up the process of calculating

closure of EDBM. The solution presented in the thesis is based by brute force checking of all combinations of the expressions represented by EDBM elements. The number of those combinations grows exponentially with the number of clocks and parameters used in the system and with complexity of the bounded expressions. Therefore, for complex systems, calculating closure may take too long what makes using EDBM inconvenient. Therefore finding analytical way for determining EDBM's closure is a target for a research work in the future.

Bibliography

[1] www.boost.org.

[2] *GNU Bison manual*, 2.3 edition, 2006.

[3] Specification & Description Language - Real Time v. 2.2, 2006.

[4] Omar Alfandi, Patryk Chamuczyński, Henrik Brosenne, Constantin Werner, and Dieter Hogrefe. Performance evaluation of PANA pre-authentication and PANA context transfer. *International Conference on Wireless and Mobile Communications*, 0:43–48, 2008.

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

[6] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *ACM Symposium on Theory of Computing*, pages 592–601, 1993.

[7] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D'argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Moller, Paul Pettersson, and Carsten Weise. Uppaal-now, next and future. In *In Proc. MOVEP 00, LNCS 2067*, pages 99–124. Springer, 2001.

[8] Aurore Annichini, Eugene Asarin, and Ahmed Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 419–434, London, UK, 2000. Springer-Verlag.

[9] Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. Trex: A tool for reachability analysis of complex systems. In *Computer Aided Verification*, pages 368–372, 2001.

[10] André Arnold. Topological characterizations of infinite behaviours of transition systems. In *ICALP*, pages 28–38, 1983.

[11] Ismail Assayad, Philippe Gerner, Sergio Yovine, and Valerie Bertin. Modelling, analysis and parallel implementation of an on-line video encoder. In *DFMA '05: Proceedings of the First International Conference on Distributed Frameworks for Multimedia Applications*, pages 295–302, Washington, DC, USA, 2005. IEEE Computer Society.

[12] Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 359–369, New York, NY, USA, 1991. ACM.

[13] C. Baier, N. Bertrand, P. Bouyer, Th. Brihaye, and M. Groesser. Probabilistic and topological semantics for timed automata. In V. Arvind and Sanjiva Prasad, editors, *Proceedings of the 27th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'07)*, volume 4855 of *Lecture Notes in Computer Science*, New Delhi, India, December 2007. Springer.

[14] Christel Baier and Joost P. Katoen. *Principles of Model Checking - The MIT Press.*

[15] M. Balser, O. Coltell, J. van Croonenborg, C. Duelli, F. van Harmelen, A. Jovell, P. Lucas, M. Marcos, S. Miksch, W. Reif, K. Rosenbrand, A. Seyfang, and A. ten Teije. Protocure: Supporting the development of medical protocols through formal methods. In *Proceedings of the workshop on Computerised Protocols and Guidelines*, page xxx. IOS Press, 2004.

[16] Miryam Barad. Timed petri nets as a verification tool. In *WSC '98: Proceedings of the 30th conference on Winter simulation*, pages 547–554, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[17] Tomás Barros, Rabéa Boulifa, and Eric Madelaine. Parameterized models for distributed java objects. In *FORTE*, pages 43–60, 2004.

[18] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL implementation secrets. In *Proc. of 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002.

[19] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of*

Computer, Communication, and Software Systems, SFM-RT 2004, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[20] Richard Ernest Bellman. *Dynamic Programming*. Dover Publications, Incorporated, 2003.

[21] Johan Bengtsson, W. O. David Griffioen, Kåre J. Kristoffersen, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Verification of an audio protocol with bus collision using uppaal. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 244–256, London, UK, 1996. Springer-Verlag.

[22] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In W. Reisig and G. Rozenberg, editors, *In Lecture Notes on Concurrency and Petri Nets*, Lecture Notes in Computer Science vol 3098. Springer–Verlag, 2004.

[23] Ismaïl Berrada. *Modélisation, Analyse et Test des Systèmes Communicants à Contraintes Temporelles: Vers une Approche Ouverte du Test*. PhD thesis, University of Bordeaux I, 2005.

[24] Ismaïl Berrada, Richard Castanet, and Patrick Félix. Testing communicating systems: a model, a methodology and a tool. In *17th IFIP International Conference on Testing of Communicating Systems*, Lecture Notes in Computer Science, Montreal, Canada, may 2005. Elsevier.

[25] Ismail Berrada, Patryk Chamuczyński, and Richard Castanet. An efficient test selection algorithm for real time systems. In *NOTERE 2007*, Marrakech, Marocco, June 2007.

[26] BMW. byteflight protocol specification.

[27] A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly symbolic model checking for real-time systems. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, page 25, Washington, DC, USA, 1997. IEEE Computer Society.

[28] Ana R. Cavalli, David Lee, Christian Rinderknecht, and Fatiha Zaïdi. Hit-or-jump: An algorithm for embedded testing with applications to in services. In *FORTE XII / PSTV XIX '99: Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication*

Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), pages 41–56, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[29] Patryk Chamuczyński. Timed functional modeling of time-triggered protocol algorithms. Master's thesis, Wroclaw University of Technology, 2004.

[30] Patryk Chamuczyński, Omar Alfandi, Henrik Brosenne, Constantin Werner, and Dieter Hogrefe. Enabling pervasiveness by seamless inter-domain handover: Performance study of pana pre-authentication. In *PERCOM '08: Proceedings of the 2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications*, pages 372–376, Washington, DC, USA, 2008. IEEE Computer Society.

[31] Patryk Chamuczyński, Omar Alfandi, Constantin Werner, Henrik Brosene, and Dieter Hogrefe. Performance study of pana pre-authentication for interdomain handover. In *ICNS '08: Proceedings of the Fourth International Conference on Networking and Services (icns 2008)*, pages 79–83, Washington, DC, USA, 2008. IEEE Computer Society.

[32] Aurore Collomb. *Vérification d'automates étendus : algorithmes d'analyse symbolique et mise en oeuvre*. PhD thesis, University Grenoble II, 2001.

[33] Collomb-Annichini and Sighireanu. Parameterized reachability analysis of the IEEE 1394 root contention protocol using TReX.

[34] FlexRay consortium. Flexray communication systems protocol specification version 2.0, 2005.

[35] Judith Crow and Ben Di Vito. Formalizing space shuttle software requirements: four case studies. *ACM Trans. Softw. Eng. Methodol.*, 7(3):296–332, 1998.

[36] P. D'Argenio, J. Katoen, and E. Brinksma. A stochastic automata model and its algebraic approach, 1995.

[37] Alexandre David and Wang Yi. Hierarchical timed automata for uppaal, 1999.

[38] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool kronos. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 208–219, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[39] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the international workshop on Automatic verification methods for finite*

state systems, pages 197–212, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[40] Michal Pawel Dlubek. *Optical performance monitoring for amplified spontaneous emission related issues in transparent optical network*. PhD thesis, University of Newcastle, 2008.

[41] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[42] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 169–181, London, UK, 1980. Springer-Verlag.

[43] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[44] D. Forsberg et al. Protocol for carrying authentication for network access (PANA). IETF draft, sep 2007.

[45] David P. Gilliarn, John C. Kelly, and Matt Bishop. Reducing software security risk through an integrated approach. In *WETICE '00: Proceedings of the 9th IEEE International Workshops on Enabling Technologies*, pages 141–146, Washington, DC, USA, 2000. IEEE Computer Society.

[46] Susanne Graf, Ileana Ober, and Iulian Ober. A real-time profile for uml. *Int. J. Softw. Tools Technol. Transf.*, 8(2):113–127, 2006.

[47] T. A. Henzinger. The theory of hybrid automata. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 278, Washington, DC, USA, 1996. IEEE Computer Society.

[48] T. A. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. 600:226–??, 1992.

[49] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.

[50] Gerard J. Holzmann. The spin model-checker. *In Proceeding FORTE 1999*, 28:481–497, 1997.

[51] Thomas Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. Linear parametric model checking of timed automata. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 189–203, 2001.

[52] ISO/IEC9646. *OSI Conformance Testing Methodology and Framework*. ISO, Geneva, Switzerland.

[53] ISO/IEC9646. *Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*. ISO, Geneva, Switzerland, 2003.

[54] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Va. The theory of timed i/o automata. Technical report, 2004.

[55] R. M. Keller. Formal verification of parallel programs. *Communications of the ACM*, (19(7)):371–384, 1976.

[56] James D. Kindrick, John A. Sauter, and Robert S. Matthews. Improving conformance and interoperability testing. *StandardView*, 4(1):61–68, 1996.

[57] K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: compact data structure and state-space reduction. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, page 14, Washington, DC, USA, 1997. IEEE Computer Society.

[58] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[59] Fredrik Larsson. Efficient implementation of model-checkers for networks of timed automata. Technical report, 2000.

[60] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gear controller. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 281–297, London, UK, 1998. Springer-Verlag.

[61] Patricia D. L. Machado, Daniel A. Silva, and Alexandre C. Mota. Towards property oriented testing. *Electron. Notes Theor. Comput. Sci.*, 184:3–19, 2007.

[62] Petr Matoušek. Tools for parametric verification. a comparison on a case study. *Journal of Universal Computer Science*, 10(10):1469–1495, 2004.

[63] Petr Matoušek. *Symbolic Data Structures for Parametric Verification.* PhD thesis, 2005.

[64] Mercedes Merayo, nez Manuel Nú and Ismael Rodríguez. Extending efsms to specify and test timed systems with action durations and time-outs. *IEEE Trans. Comput.*, 57(6):835–844, 2008.

[65] Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Compiling real-time specifications into extended automata. *IEEE Transactions on Software Engineering*, 18:794–804, 1992.

[66] Lawrence C Paulson. Isabelle: A generic theorem prover. *Journal of Automated Reasoning*, 5, 1994.

[67] Doron A. Peled, David Gries, and Fred B. Schneider, editors. *Software reliability methods.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[68] Paul Peterson. *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice.* PhD thesis, 1999.

[69] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[70] J. P. Queille and J. Sifakis. A temporal logic to deal with fairness in transition systems. In *SFCS '82: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 217–225, Washington, DC, USA, 1982. IEEE Computer Society.

[71] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, 1985.

[72] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theor. Comput. Sci.*, 58(1-3):249–261, 1988.

[73] Theo C. Ruys and Rom Langerak. Validation of bosch mobile communication network architecture with SPIN. In *In Proceedings of SPIN97, the Third International Workshop on SPIN, University of Twente*, 1997.

[74] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003.

[75] R. F. Lutje Spelberg, Hans Toetenel, and Marcel Ammerlaan. Partition refinement in real-time model checking. In *FTRTFT '98: Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 143–157, London, UK, 1998. Springer-Verlag.

[76] Bertrand Tavernier. Calife: A generic graphical user interface for automata tools. *Electr. Notes Theor. Comput. Sci.*, 110:169–172, 2004.

[77] Jüri Vain, Kullo Raiend, Andres Kull, and Juhan P. Ernits. Synthesis of test purpose directed reactive planning tester for nondeterministic systems. In *ASE*, pages 363–372, 2007.

[78] Will Estes Vern Paxson and John Millaway. *The flex Manual*, 2.5.35 edition, 2007.

[79] B. L. Di Vito and L. W. Roberts. Using formal methods to assist in the requirements analysis of the space shuttle gps change request. Technical report, 1996.

[80] Yingxu Wang. The real-time process algebra (rtpa). *Ann. Softw. Eng.*, 14(1-4):235–274, 2002.

[81] Libor Waszniowski and Zdeněk Hanzálek. Formal verification of multitasking applications based on timed automata model. *Real-Time Syst.*, 38(1):39–65, 2008.

[82] Henri B. Weinberg and Lenore D. Zuck. Timed ethernet: Real-time formal specification of ethernet. In *CONCUR '92: Proceedings of the Third International Conference on Concurrency Theory*, pages 370–385, London, UK, 1992. Springer-Verlag.

[83] Freek Wiedijk. Comparing mathematical provers. In *Proc. MKM 03, volume 2594 of LNCS*, pages 188–202. Springer, 2003.

[84] Wai Wong. Recording and checking HOL proofs. In Phillip J. Windley E. Thomas Shubert and James Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications: 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 353–368. Springer-Verlag, 1995.

[85] Wai Wong. A proof checker for HOL. Technical Report 389, University of Cambridge Computer Laboratory, March 1996.

[86] Wang Yi. Ccs + time = an interleaving model for real time systems. In *Proceedings of the 18th international colloquium on Automata, languages and programming*, pages 217–228, New York, NY, USA, 1991. Springer-Verlag New York, Inc.