Timed Automata: Semantics, Algorithms and Tools

Johan Bengtsson and Wang Yi

Uppsala University
Email: {johanb,yi}@it.uu.se

Abstract. This chapter is to provide a tutorial and pointers to results and related work on timed automata with a focus on semantical and algorithmic aspects of verification tools. We present the concrete and abstract semantics of timed automata (based on transition rules, regions and zones), decision problems, and algorithms for verification. A detailed description on DBM (Difference Bound Matrices) is included, which is the central data structure behind several verification tools for timed systems. As an example, we give a brief introduction to the tool UPPAAL.

1 Introduction

Timed automata is a theory for modeling and verification of real time systems. Examples of other formalisms with the same purpose, are timed Petri Nets, timed process algebras, and real time logics [BD91,RR88,Yi91,NS94,AH94,Cha99]. Following the work of Alur and Dill [AD90,AD94], several model checkers have been developed with timed automata being the core of their input languages *e.g.* [Yov97,LPY97]. It is fair to say that they have been the driving force for the application and development of the theory. The goal of this chapter is to provide a tutorial on timed automata with a focus on the semantics and algorithms based on which these tools are developed.

In the original theory of timed automata [AD90,AD94], 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. A simplified version, namely *Timed Safety Automata* is introduced in [HNSY94] to specify progress properties using local invariant conditions. Due to its simplicity, Timed Safety Automata has been adopted in several verification tools for timed automata *e.g.* UPPAAL [LPY97] and Kronos [Yov97]. In this presentation, we shall focus on Timed Safety Automata, and following the literature, refer them as *Timed Automata* or simply automata when it is understood from the context.

The rest of the chapter is organized as follows: In the next section, we describe the syntax and operational semantics of timed automata. The section also addresses decision problems relevant to automatic verification. In the literature, the decidability and undecidability of such problems are often considered to be the fundamental properties of a computation model. Section 3 presents the abstract version of the operational semantics based on regions and zones. Section 4 describes the data structure DBM (Difference

Bound Matrices) for the efficient representation and manipulation of zones, and operations on zones, needed for symbolic verification. Section 5 gives a brief introduction to the verification tool UPPAAL. Finally, as an appendix, we list the pseudo-code for the presented DBM algorithms.

2 Timed Automata

A timed automaton is essentially a finite automaton (that is a graph containing a finite set of nodes or locations and a finite set of labeled edges) extended with real-valued variables. Such an automaton may be considered as an abstract model of a timed system. The variables model the logical clocks in the system, that are initialized with zero when the system is started, and then increase synchronously with the same rate. Clock constraints *i.e.* guards on edges are used to restrict the behavior of the automaton. A transition represented by an edge can be taken when the clocks values satisfy the guard labeled on the edge. Clocks may be reset to zero when a transition is taken.

The first example Fig. 1(a) is an example timed automaton. The timing behavior of the automaton is controlled by two clocks x and y. The clock x is used to control the self-loop in the location **loop**. The single transition of the loop may occur when $x = 1$. Clock y controls the execution of the entire automaton. The automaton may leave **start** at any time point when y is in the interval between 10 and 20; it can go from **loop** to **end** when y is between 40 and 50, *etc.*

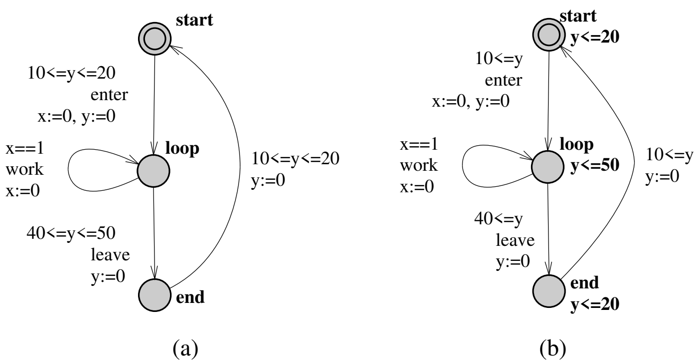


Fig. 1. Timed Automata and Location Invariants

Timed Büchi Automata A guard on an edge of an automaton is only an enabling condition of the transition represented by the edge; but it can not force the transition to be taken. For instance, the example automaton may stay forever in any location, just idling. In the initial work by Alur and Dill [AD90], the problem is solved by introducing Büchi-acceptance conditions; a subset of the locations in the automaton are marked as accepting, and only those executions passing through an accepting location infinitely often are considered valid behaviors of the automaton. As an example, consider again the automaton in Fig. 1(a) and assume that **end** is marked as accepting. This implies that all executions of the automaton must visit **end** infinitely many times. This imposes implicit conditions on **start** and **loop**. The location **start** must be left when the value of y is at most 20, otherwise the automaton will get stuck in **start** and never be able to enter **end**. Likewise, the automaton must leave **loop** when y is at most 50 to be able to enter **end**.

Timed Safety Automata A more intuitive notion of progress is introduced in *timed safety automata* [HNSY94]. Instead of accepting conditions, in timed safety automata, locations may be put local timing constraints called *location invariants*. An automaton may remain in a location as long as the clocks values satisfy the invariant condition of the location. For example, consider the timed safety automaton in Fig. 1(b), which corresponds to the Büchi automaton in Fig. 1(a) with **end** marked as an accepting location. The invariant specifies a local condition that **start** and **end** must be left when y is at most 20 and **loop** must be left when y is at most 50. This gives a local view of the timing behavior of the automaton in each location.

In the rest of this chapter, we shall focus on timed safety automata and refer such automata as *Timed Automata* or simply automata without confusion.

2.1 Formal Syntax

Assume a finite set of real-valued variables $\mathcal{C}$ ranged over by x, y *etc.*standing for clocks and a finite alphabet Σ ranged over by a, b *etc.*standing for actions.

Clock Constraints A clock constraint is a conjunctive formula of atomic constraints of the form $x \sim n$ or $x - y \sim n$ for $x, y \in \mathcal{C}, \sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. Clock constraints will be used as guards for timed automata. We use $\mathcal{B}(\mathcal{C})$ to denote the set of clock constraints, ranged over by g and also by D later.

Definition 1 (Timed Automaton) *A timed automaton $\mathcal{A}$ is a tuple $\langle N, l_0, E, I \rangle$ where*

- *N is a finite set of locations (or nodes),*
- *$l_0 \in N$ is the initial location,*
- *$E \subseteq N \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times N$ is the set of edges and*
- *$I : N \longrightarrow \mathcal{B}(\mathcal{C})$ assigns invariants to locations*

We shall write $l \xrightarrow{g,a,r} l'$ when $\langle l, g, a, r, l' \rangle \in E$.

As in verification tools e.g. UPPAAL *[LPY97], we restrict location invariants to constraints that are downwards closed, in the form: $x \leq n$ or $x < n$ where n is a natural number.*

Concurrency and Communication To model concurrent systems, timed automata can be extended with parallel composition. In process algebras, various parallel composition operators have been proposed to model different aspects of concurrency (see *e.g.* CCS and CSP [Mil89,Hoa78]). These algebraic operators can be adopted in timed automata. In the UPPAAL modeling language [LPY97], the CCS parallel composition operator [Mil89] is used, which allows interleaving of actions as well as hand-shake synchronization. The precise definition of this operator is given in Section 5.

Essentially the parallel composition of a set of automata is the product of the automata. Building the product automaton is an entirely syntactical but computationally expensive operation. In UPPAAL, the product automaton is computed on-the-fly during verification.

2.2 Operational Semantics

The semantics of a timed automaton is defined as a transition system where a state or configuration consists of the current location and the current values of clocks. There are two types of transitions between states. The automaton may either delay for some time (a delay transition), or follow an enabled edge (an action transition).

To keep track of the changes of clock values, we use functions known as *clock assignments* mapping $\mathcal{C}$ to the non-negative reals $\mathbb{R}_+$. Let u, v denote such functions, and use $u \in g$ to mean that the clock values denoted by u satisfy the guard g. For $d \in \mathbb{R}_+$, let $u + d$ denote the clock assignment that maps all $x \in \mathcal{C}$ to $u(x) + d$, and for $r \subseteq \mathcal{C}$, let $[r \mapsto 0]u$ denote the clock assignment that maps all clocks in r to 0 and agree with u for the other clocks in $\mathcal{C} \setminus r$.

Definition 2 (Operational Semantics) *The semantics of a timed automaton is a transition system (also known as a timed transition system) where states are pairs $\langle l, u\rangle$, and transitions are defined by the rules:*

- $\langle l, u\rangle \xrightarrow{d} \langle l, u + d\rangle$ *if* $u \in I(l)$ *and* $(u + d) \in I(l)$ *for a non-negative real* $d \in \mathbb{R}_+$
- $\langle l, u\rangle \xrightarrow{a} \langle l', u'\rangle$ *if* $l \xrightarrow{g,a,r} l', u \in g, u' = [r \mapsto 0]u$ *and* $u' \in I(l')$

2.3 Verification Problems

The operational semantics is the basis for verification of timed automata. In the following, we formalize decision problems in timed automata based on transition systems.

Language Inclusion A *timed action* is a pair (t, a), where $a \in \Sigma$ is an action taken by an automaton $\mathcal{A}$ after $t \in \mathbb{R}_+$ time units since $\mathcal{A}$ has been started. The absolute time t is called a *time-stamp* of the action a. A *timed trace* is a (possibly infinite) sequence of timed actions $\xi = (t_1, a_1)(t_2, a_2)...(t_i, a_i)...$ where $t_i \leq t_{i+1}$ for all $i \geq 1$.

Definition 3 *A* run *of a timed automaton $\mathcal{A} = \langle N, l_0, E, I\rangle$ with initial state $\langle l_0, u_0\rangle$ over a timed trace $\xi = (t_1, a_1)(t_2, a_2)(t_3, a_3)...$ is a sequence of transitions:*

$$\langle l_0, u_0\rangle \xrightarrow{d_1}\xrightarrow{a_1} \langle l_1, u_1\rangle \xrightarrow{d_2}\xrightarrow{a_2} \langle l_2, u_2\rangle \xrightarrow{d_3}\xrightarrow{a_3} \langle l_3, u_3\rangle \ldots$$

satisfying the condition $t_i = t_{i-1} + d_i$ for all $i \geq 1$.

The timed language $L(\mathcal{A})$ is the set of all timed traces ξ for which there exists a run of $\mathcal{A}$ over ξ.

Undecidability The negative result on timed automata as a computation model is that the language inclusion checking problem *i.e.* to check $L(\mathcal{A}) \subseteq L(\mathcal{B})$ is undecidable [AD94,ACH94]. Unlike finite state automata, timed automata is not determinizable in general. Timed automata can not be complemented either, that is, the complement of the timed language of a timed automaton may not be described as a timed automaton.

The inclusion checking problem will be decidable if $\mathcal{B}$ in checking $L(\mathcal{A}) \subseteq L(\mathcal{B})$ is restricted to the deterministic class of timed automata. Research effort has been made to characterize interesting classes of determinizable timed systems *e.g.* event-clock automata [AFH99] and timed communicating sequential processes [YJ94]. Essentially, the undecidability of language inclusion problem is due to the arbitrary clock reset. If all the edges labeled with the same action symbol in a timed automaton, are also labeled with the same set of clocks to reset, the automaton will be determinizable. This covers the class of event-clock automata [AFH99].

We may abstract away from the time-stamps appearing in timed traces and define the *untimed language* $L_{untimed}(\mathcal{A})$ as the set of all traces in the form $a_1 a_2 a_3 \ldots$ for which there exists a timed trace $\xi = (t_1, a_1)(t_2, a_2)(t_3, a_3)...$ in the timed language of $\mathcal{A}$.

The inclusion checking problem for untimed languages is decidable. This is one of the classic results for timed automata [AD94].

Bisimulation Another classic result on timed systems is the decidability of timed bisimulation [Cer92]. Timed bisimulation is introduced for timed process algebras [Yi91]. However, it can be easily extended to timed automata.

Definition 4 *A bisimulation R over the states of timed transition systems and the alphabet $\Sigma \cup \mathbb{R}_+$, is a symmetrical binary relation satisfying the following condition:*

for all $(s_1, s_2) \in R$, if $s_1 \xrightarrow{\sigma} s_1'$ for some $\sigma \in \Sigma \cup \mathbb{R}_+$ and s_1', then $s_2 \xrightarrow{\sigma} s_2'$ and $(s_1', s_2') \in R$ for some s_2'.

Two automata are timed bisimilar iff there is a bisimulation containing the initial states of the automata.

Intuitively, two automata are timed bisimilar iff they perform the same action transition at the same time and reach bisimilar states. In [Cer92], it is shown that timed bisimulation is decidable.

We may abstract away from timing information to establish bisimulation between automata based actions performed only. This is captured by the notion of untimed bisimulation. We define $s \xrightarrow{\epsilon} s'$ if $s \xrightarrow{d} s'$ for some real number d. Untimed bisimulation is defined by by replacing the alphabet with $\Sigma \cup \{\epsilon\}$ in Definition 4. As timed bisimulation, untimed bisimulation is decidable [LW97].

Reachability Analysis Perhaps, the most useful question to ask about a timed automaton is the reachability of a given final state or a set of final states. Such final states may be used to characterize safety properties of a system.

Definition 5 *We shall write $\langle l, u\rangle \rightarrow \langle l', u'\rangle$ if $\langle l, u\rangle \xrightarrow{\sigma} \langle l', u'\rangle$ for some $\sigma \in \Sigma \cup \mathbb{R}_+$. For an automaton with initial state $\langle l_0, u_0\rangle$, $\langle l, u\rangle$, is reachable iff $\langle l_0, u_0\rangle \rightarrow^* \langle l, u\rangle$. More generally, given a constraint $\phi \in \mathcal{B}(\mathcal{C})$ we say that the configuration $\langle l, \phi\rangle$ is reachable if $\langle l, u\rangle$ is reachable for some u satisfying ϕ.*

The notion of reachability is more expressive than it appears to be. We may specify invariant properties using the negation of reachability properties, and bounded liveness properties using clock constraints in combination with local properties on locations [LPY01] (see Section 5 for an example).

The reachability problem is decidable. In fact, one of the major advances in verification of timed systems is the symbolic technique [Dil89,YL93,HNSY94,YPD94,LPY95], developed in connection with verification tools. It adopts the idea from symbolic model checking for untimed systems, which uses boolean formulas to represent sets of states and operations on formulas to represent sets of state transitions. It is proven that the infinite state-space of timed automata can be finitely partitioned into symbolic states using clock constraints known as *zones* [Bel57,Dil89]. A detailed description on this is given in Section 3 and 4.

3 Symbolic Semantics and Verification

As clocks are real-valued, the transition system of a timed automaton is infinite, which is not an adequate model for automated verification.

3.1 Regions, Zones and Symbolic Semantics

The foundation for the decidability results in timed automata is based on the notion of *region equivalence* over clock assignments [AD94,ACD93].

Definition 6 (Region Equivalence) *Let k be a function, called a* clock ceiling*, mapping each clock $x \in \mathcal{C}$ to a natural number $k(x)$ (*i.e. *the ceiling of x). For a real number d, let $\{d\}$ denote the fractional part of d, and $\lfloor d \rfloor$ denote its integer part. Two clock assignments u, v are* region-equivalent*, denoted $u \dot{\sim}_k v$, iff*

1. *for all x, either $\lfloor u(x) \rfloor = \lfloor v(x) \rfloor$ or both $u(x) > k(x)$ and $v(x) > k(x)$,*
2. *for all x, if $u(x) \leq k(x)$ then $\{u(x)\} = 0$ iff $\{v(x)\} = 0$ and*
3. *for all x, y if $u(x) \leq k(x)$ and $u(y) \leq k(y)$ then $\{u(x)\} \leq \{u(y)\}$ iff $\{v(x)\} \leq \{v(y)\}$*

Note that the region equivalence is indexed with a clock ceiling k. When the clock ceiling is given by the maximal clock constants of a timed automaton under consideration, we shall omit the index and write $\dot{\sim}$ instead. An equivalence class $[u]$ induced by $\dot{\sim}$ is called a *region*, where $[u]$ denotes the set of clock assignments region-equivalent to u. The basis for a finite partitioning of the state-space of a timed automaton is the following facts. First, for a fixed number of clocks each of which has a maximal constant, the number of regions is finite. Second, $u \dot{\sim} v$ implies (l, u) and (l, v) are bisimilar w.r.t. the untimed bisimulation for any locaton l of a timed automaton. We use the equivalence classes induced by the untimed bisimulation as symbolic (or abstract) states to construct a finite-state model called the *region graph* or *region automaton* of the original timed automaton. The transition relation between symbolic states is defined as follows:

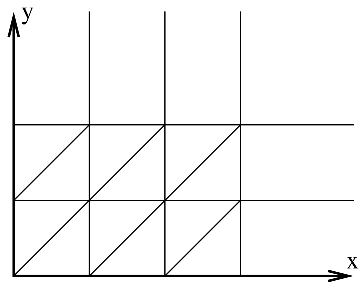


Fig. 2. Regions for a System with Two Clocks

– $\langle l, [u] \rangle \Rightarrow \langle l, [v] \rangle$ if $\langle l, u \rangle \xrightarrow{d} \langle l, v \rangle$ for a positive real number d and
– $\langle l, [u] \rangle \Rightarrow \langle l', [v] \rangle$ if $\langle l, u \rangle \xrightarrow{a} \langle l', v \rangle$ for an action a.

Note that the transition relation $\Rightarrow$ is finite. Thus the region graph for a timed automaton is finite. Several verification problems such as reachability analysis, untimed language inclusion, language emptiness [AD94] as well as timed bisimulation [Cer92] can be solved by techniques based on the region construction.

However, the problem with region graphs is the potential explosion in the number of regions. In fact, it is exponential in the number of clocks as well as the maximal constants appearing in the guards of an automaton. As an example, consider Fig. 2. The figure shows the possible regions in each location of an automaton with two clocks x and y. The largest number compared to x is 3, and the largest number compared to y is 2. In the figure, all corner points (intersections), line segments, and open areas are regions. Thus, the number of possible regions in each location of this example is 60.

A more efficient representation of the state-space for timed automata is based on the notion of *zone* and *zone-graphs* [Dil89,HNSY92,YL93,YPD94,HNSY94]. In a zone graph, instead of regions, zones are used to denote symbolic states. This in practice gives a coarser and thus more compact representation of the state-space. The basic operations and algorithms for zones to construct zone-graphs are described in Section 4. As an example, a timed automaton and the corresponding zone graph (or reachability graph) is shown in Fig. 3. We note that for this automaton the zone graph has only 8 states. The region-graph for the same example has over 50 states.

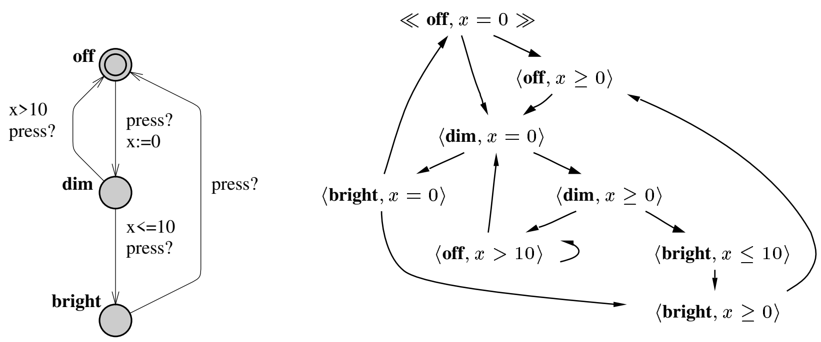


Fig. 3. A Timed Automaton and its Zone Graph

A zone is a clock constraint. Strictly speaking, a *zone* is the solution set of a clock constraint, that is the maximal set of clock assignments satisfying the constraint. It is well-known that such sets can be efficiently represented and stored in memory as DBMs (*Difference Bound Matrices*) [Bel57]. For a clock constraint D, let $[D]$ denote the maximal set of clock assignments satisfying D. In the following, to save notation, we shall use D to stand for $[D]$ without confusion. Then $\mathcal{B}(\mathcal{C})$ denotes the set of zones.

A symbolic state of a timed automaton is a pair $\langle l, D\rangle$ representing a set of states of the automaton, where l is a location and D is a zone. A symbolic transition describes all the possible concrete transitions from the set of states.

Definition 7 *Let D be a zone and r a set of clocks. We define $D^{\uparrow} = \{u+d \,|\, u \in D, d \in \mathbb{R}_+\}$ and $r(D) = \{[r \mapsto 0]u \mid u \in D\}$. Let $\rightsquigarrow$ denote the symbolic transition relation over symbolic states defined by the following rules:*

– $\langle l, D\rangle \rightsquigarrow \langle l, D^{\uparrow} \wedge I(l)\rangle$

– $\langle l, D\rangle \rightsquigarrow \langle l', r(D \wedge g) \wedge I(l')\rangle$ *if* $l \xrightarrow{g,a,r} l'$

We shall study these operations in details in Section 4 where $D^{\uparrow}$ is written as $\mathtt{up}(D)$ and $r(D)$ as $\mathtt{reset}(D, r := 0)$. It will be shown that the set of zones $\mathcal{B}(\mathcal{C})$ is closed under these operations, in the sense that the result of the operations is also a zone. Another important property of zones is that a zone has a canonical form. A zone D is *closed under entailment* or just closed for short, if no constraint in D can be strengthened without reducing the solution set. The canonicity of zones means that for each zone $D \in \mathcal{B}(\mathcal{C})$, there is a unique zone $D' \in \mathcal{B}(\mathcal{C})$ such that D and D' have exactly the same solution set and D' is closed under entailment. Section 4 describes how to compute and represent the canonical form of a zone. It is the key structure for the efficient implementation of state-space exploration using the symbolic semantics.

The symbolic semantics corresponds closely to the operational semantics in the sense that $\langle l, D\rangle \rightsquigarrow \langle l', D'\rangle$ implies for all $u' \in D'$, $\langle l, u\rangle \rightarrow \langle l', u'\rangle$ for some $u \in D$. More generally, the symbolic semantics is a correct and full characterization of the operational semantics given in Definition 2.

Theorem 1 *Assume a timed automaton with initial state* $\langle l_0, u_0\rangle$.

1. *(soundness)* $\langle l_0, \{u_0\}\rangle \rightsquigarrow^* \langle l_f, D_f\rangle$ *implies* $\langle l_0, u_0\rangle \rightarrow^* \langle l_f, u_f\rangle$ *for all* $u_f \in D_f$.
2. *(Completeness)* $\langle l_0, u_0\rangle \rightarrow^* \langle l_f, u_f\rangle$ *implies* $\langle l_0, \{u_0\}\rangle \rightsquigarrow^* \langle l_f, D_f\rangle$ *for some* D_f *such that* $u_f \in D_f$

The soundness means that if the initial symbolic state $\langle l_0, \{u_0\}\rangle$ may lead to a set of final states $\langle l_f, D_f\rangle$ according to $\rightsquigarrow$, all the final states should be reachable according to the concrete operational semantics. The completeness means that if a state is reachable according to the concrete operational semantics, it should be possible to conclude this using the symbolic transition relation.

Unfortunately, the relation $\rightsquigarrow$ is infinite, and thus the zone-graph of a timed automaton may be infinite, which can be a problem to guarantee termination in a verification procedure. As an example, consider the automaton in Fig. 4. The value of clock y drifts away unboundedly, inducing an infinite zone-graph.

The solution is to transform (*i.e.* normalize) zones that may contain arbitrarily large constants to their representatives in a class of zones whose constants are bounded by fixed constants *e.g.* the maximal clock constants appearing in the automaton, using an abstraction technique similar to the widening operation [Hal93]. The intuition is that once the value of a clock is larger than the maximal constant in the automaton, it is no longer important to know the precise value of the clock, but only the fact that it is above the constant.

3.2 Zone-Normalization for Automata without Difference Constraints

In the original theory of timed automata [AD94], difference constraints are not allowed to appear in the guards. Such automata (whose guards contain only atomic constraints in

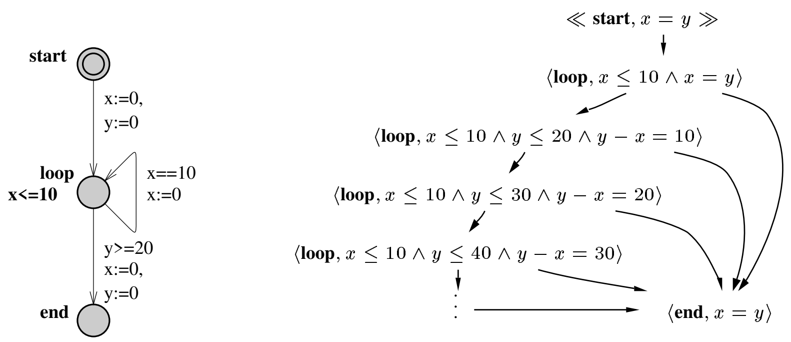


Fig. 4. A Timed Automaton with an Infinite Zone-Graph

the form $x \sim n$) are known as diagonal-free automata in the literature in [BDGP98]. For diagonal-free automata, a well-studied zone-normalization procedure is the so-called *k-normalization* operation on zones [Rok93,Pet99]. It is implemented in several verification tools for timed automata *e.g.* UPPAAL to guarantee termination.

Definition 8 **(k-Normalization)** *Let D be a zone and k a clock ceiling. The semantics of the k-normalization operation on zones is defined as follows:*

$$\mathit{norm}_k(D) = \{u | u \dot{\sim}_k v, v \in D\}$$

Note that the normalization operation is indexed by a clock ceiling k. According to [Rok93,Pet99], $\mathsf{norm}_k(D)$ can be computed from the canonical representation of D by

1. removing all constraints of the form $x < m$, $x \leq m$, $x - y < m$ and $x - y \leq m$ where $m > k(x)$,
2. replacing all constraints of the form $x > m$, $x \geq m$, $x - y > m$ and $x - y \geq m$ where $m > k(x)$ with $x > k(x)$ and $x - y > k(x)$ respectively.

Let $[D]_k$ denote the resulted zone by the above transformation. This is exactly the normalized zone as defined in Definition 8, that is, $[D]_k = \{u | u \dot{\sim}_k v, v \in D\}$

As an example, the normalized zone-graph of the automaton in Fig. 4 is shown in Fig. 5 where the clock ceiling is given by the maximal clock constants appearing in the automaton.

Note that for a fixed number of clocks with a clock ceiling k, there can be only finitely many normalized zones. The intuition is that if the constants allowed to use are bounded, one can write down only finitely many clock constraints. This gives rise to a finite characterization for $\rightarrow$.

Definition 9 $\langle l, D\rangle \rightsquigarrow_k \langle l', \mathit{norm}_k(D')\rangle$ *if* $\langle l, D\rangle \rightsquigarrow \langle l', D'\rangle$.

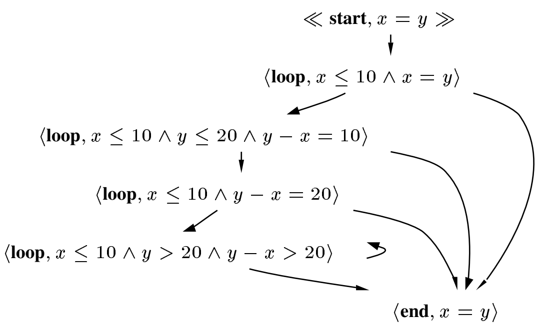


Fig. 5. Normalized Zone Graph for the Automaton in Fig. 4

For the class of diagonal-free timed automata $\rightsquigarrow_k$ is sound, complete and finite in the following sense.

Theorem 2 *Assume a timed automaton with initial state $\langle l_0, u_0 \rangle$, whose maximal clock constants are bounded by a clock ceiling k. Assume that the automaton has no guards containing difference constraints in the form of $x - y \sim n$.*

1. *(soundness) $\langle l_0, \{u_0\} \rangle \rightsquigarrow_k^* \langle l_f, D_f \rangle$ implies $\langle l_0, u_0 \rangle \rightarrow^* \langle l_f, u_f \rangle$ for all $u_f \in D_f$ such that $u_f(x) \leq k(x)$ for all x.*
2. *(Completeness) $\langle l_0, u_0 \rangle \rightarrow^* \langle l_f, u_f \rangle$ with $u_f(x) \leq k(x)$ for all x, implies $\langle l_0, \{u_0\} \rangle \rightsquigarrow_k^* \langle l_f, D_f \rangle$ for some D_f such that $u_f \in D_f$*
3. *(Finiteness) The transition relation $\rightsquigarrow_k$ is finite.*

Unfortunately the soundness will not hold for timed automata whose guards contain difference constraints. We demonstrate this by an example. Consider the automaton shown in Fig. 6. The final location of the automaton is not reachable according to the operational semantics. This is because in location S_2, the clock zone is ($x - y > 2$ and $x > 2$) and the guard on the outgoing edge is $x < z + 1 \wedge z < y + 1$ which is equivalent to $x - z < 1 \wedge z - y < 1 \wedge x - y < 2$. Obviously the zone at S_2 can never enable the guard, and thus the last transition will never be possible. However, because the maximal constants for clock x is 1 (and 2 for y), the zone in location S_2: $x - y > 2 \wedge x > 2$ will be normalized to $x - y > 1 \wedge x > 1$ by the maximal constant 1 for x, which enables the guard $x - z < 1 \wedge z - y < 1$ and thus the symbolic reachability analysis based on the above normalization algorithm would incorrectly conclude that the last location is reachable.

The zones in canonical forms, generated in exploring the state-space of the counter example are given in Fig. 7. The implicit constraints that all clocks are non-negative are not shown.

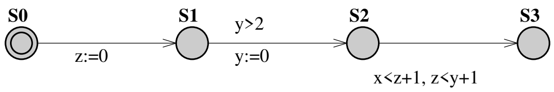


Fig. 6. A counter example

$$S_0 : \begin{cases} x - y = 0 \\ y - z = 0 \\ z - x = 0 \end{cases} \quad S_1 : \begin{cases} x - y = 0 \\ z - x \leq 0 \\ z - y \leq 0 \end{cases} \quad S_2 : \begin{cases} y - x < -2 \\ y - z \leq 0 \\ z - x \leq 0 \\ \mathbf{0} - x < -2 \end{cases}$$

(a) Zones without normalization

$$S_0 : \begin{cases} x - y = 0 \\ y - z = 0 \\ z - x = 0 \end{cases} \quad S_1 : \begin{cases} x - y = 0 \\ z - x \leq 0 \\ z - y \leq 0 \end{cases} \quad S_2 : \begin{cases} y - x < -1 \\ y - z \leq 0 \\ z - x \leq 0 \\ \mathbf{0} - x < -1 \end{cases} \quad S_3 : \begin{cases} y - x < -1 \\ y - z < 0 \\ z - x < 0 \\ \mathbf{0} - x < -1 \\ \mathbf{0} - z < 0 \\ x - z < 1 \end{cases}$$

(b) Zones normalized with k-normalization

Fig. 7. Zones for the counter example in Fig. 6

Note that at S_0 and S_1, the normalized and un-normalized zones are identical. The problem is at S_2 where the intersection of the guard (on the only outgoing edge) with the un-normalized zone is empty and non-empty with the normalized zone.

3.3 Zone-Normalization for Automata with Difference Constraints

Our definition of timed automata (Definition 1) allows any clock constraint to appear in a guard, which may be a difference constraint in the form of $x - y \sim n$. Such automata are indeed needed in many applications *e.g.* to model scheduling problems [FPY02]. It is shown that an automaton containing difference constraints can be transformed to an equivalent diagonal-free automaton [BDGP98]. However, the transformation is not applicable since it is based on the region construction. Besides, it is impractical to implement such an approach in a tool. Since the transformation modifies the model before analysis, it is difficult to trace debugging information provided by the tool back to the original model.

In [Ben02,BY03], a refined normalization algorithm is presented for automata that may have guards containing difference constraints. The algorithm transforms DBMs according to not only the maximal constants of clocks but also difference constraints appearing in the automaton under consideration. Note that the difference constraints correspond to the diagonal lines which split the entire space of clock assignments. A finer partitioning is needed.

We present the semantical characterization for the refined normalization operation based on a refined version of the region equivalence from Definition 6.

Definition 10 (Normalization Using Difference Constraints) *Let $\mathcal{G}$ stand for a finite set of difference constraints of the form $x-y \sim n$ for $x, y \in \mathcal{C}$, $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$, and k for a clock ceiling. Two clock assignments u, v are equivalent, denoted $u \dot{\sim}_{k,\mathcal{G}} v$ if the following holds:*

- *$u \dot{\sim}_k v$ and*
- *for all $g \in \mathcal{G}$, $u \in g$ iff $v \in g$.*

The semantics of the refined k-normalization operation on zones is defined as follows:

$$\mathsf{norm}_{k,\mathcal{G}}(D) = \{u | u \ \dot{\sim}_{k,\mathcal{G}} \ v, v \in D\}$$

Note that the refined region equivalence is indexed by both a clock ceiling k and a finite set of difference constraints $\mathcal{G}$, and so is the normalization operation.

Since the number of regions induced by $\dot{\sim}_k$ is finite and there are only finitely many constraints in $\mathcal{G}$, $\dot{\sim}_{k,\mathcal{G}}$ induces finitely many equivalence classes. Thus for any given zone D, $\mathsf{norm}_{k,\mathcal{G}}(D)$ is well-defined in the sense that it contains only a finite set of equivalence classes though the set may not be a convex zone, and it can be computed effectively according to the refined regions. In general, $\mathsf{norm}_{k,\mathcal{G}}(D)$ is a non-convex zone, which can be implemented as the union of a finite list of convex zones. The next section will show how to compute this efficiently.

The refined zone-normalization gives rise to a finite characterization for $\rightarrow$.

Definition 11 $\langle l, D\rangle \leadsto_{k,\mathcal{G}} \langle l', \mathsf{norm}_{k,\mathcal{G}}(D')\rangle$ *if* $\langle l, D\rangle \leadsto \langle l', D'\rangle$.

The following states the correctness and finiteness of $\leadsto_{k,\mathcal{G}}$.

Theorem 3 *Assume a timed automaton with initial state $\langle l_0, u_0\rangle$, whose maximal clock constants are bounded by a clock ceiling k, and whose guards contain only a finite set of difference constraints denoted $\mathcal{G}$.*

1. *(soundness) $\langle l_0, \{u_0\}\rangle \ (\leadsto_{k,\mathcal{G}})^* \ \langle l_f, D_f\rangle$ implies $\langle l_0, u_0\rangle \rightarrow^* \langle l_f, u_f\rangle$ for all $u_f \in D_f$ such that $u_f(x) \leq k(x)$ for all x.*
2. *(Completeness) $\langle l_0, u_0\rangle \rightarrow^* \langle l_f, u_f\rangle$ with $u_f(x) \leq k(x)$ for all x implies $\langle l_0, \{u_0\}\rangle \ (\leadsto_{k,\mathcal{G}})^* \ \langle l_f, D_f\rangle$ for some D_f such that $u_f \in D_f$*
3. *(Finiteness) The transition relation $\leadsto_{k,\mathcal{G}}$ is finite.*

3.4 Symbolic Reachability Analysis

Model-checking concerns two types of properties *liveness* and *safety*. The essential algorithm of checking liveness properties is loop detection, which is computationally expensive. The main effort on verification of timed systems has been put on safety properties that can be checked using reachability analysis by traversing the state-space of timed automata.

Algorithm 1 Reachability analysis

```
PASSED = ∅, WAIT = {⟨l_0, D_0⟩}
while WAIT ≠ ∅ do
  take ⟨l, D⟩ from WAIT
  if l = l_f ∧ D ∩ φ_f ≠ ∅ then return "YES"
  if D ⊈ D' for all ⟨l, D'⟩ ∈ PASSED then
    add ⟨l, D⟩ to PASSED
    for all ⟨l', D'⟩ such that ⟨l, D⟩ ⇝ k, G⟨l', D'⟩ do
      add ⟨l', D'⟩ to WAIT
    end for
  end if
end while
return "NO"
```

Reachability analysis can be used to check properties on states. It consists of two basic steps, computing the state-space of an automaton under consideration, and searching for states that satisfy or contradict given properties. The first step can either be performed prior to the search, or done *on-the-fly* during the search process. Computing the state-space on-the-fly has an obvious advantage over pre-computing, in that only the part of the state-space needed to prove the property is generated. It should be noted though, that even on-the-fly methods will generate the entire state-space to prove certain properties, *e.g.* invariant properties.

Several model-checkers for timed systems are designed and optimized for reachability analysis based on the symbolic semantics and the zone-representation (see Section 4). As an example, we present the core of the verification engine of UPPAAL (see Algorithm 1).

Assume a timed automaton $\mathcal{A}$ with a set of initial states and a set of final states (*e.g.* the bad states) characterized as $\langle l_0, D_0\rangle$ and $\langle l_f, \phi_f\rangle$ respectively. Assume that k is the clock ceiling defined by the maximal constants appearing in $\mathcal{A}$ and ϕ_f, and $\mathcal{G}$ denotes the set of difference constraints appearing in $\mathcal{A}$ and ϕ_f. Algorithm 1 can be used to check if the initial states may evolve to any state whose location is l_f and whose clock assignment satisfies ϕ_f. It computes the normalized zone-graph of the automaton on-the-fly, in search for symbolic states containing location l_f and zones intersecting with ϕ_f.

The algorithm computes the transitive closure of $\rightsquigarrow_{k,\mathcal{G}}$ step by step, and at each step, checks if the reached zones intersect with ϕ_f. From Theorem 2, it follows that the algorithm will return with a correct answer. It is also guaranteed to terminate because $\rightsquigarrow_{k,\mathcal{G}}$ is finite. As mentioned earlier, for a given timed automaton with a fixed set of clocks whose maximal constants are bounded by a clock ceiling k, and a fixed set of diagonal constraints contained in the guards, the number of all possible normalized zones is bounded because a normalized zone can not contain arbitrarily large or arbitrarily small constants. In fact the smallest possible zones are the refined regions. Thus the whole state-space of a timed automaton can only be partitioned into finitely many symbolic states and the worst case is the size of the region graph of the automaton, induced by the refined region equivalence. Therefore, the algorithm is working on a finite structure and it will terminate.

Algorithm 1 also highlights some of the issues in developing a model-checker for timed automata. Firstly, the representation and manipulation of states, primarily zones, is crucial to the performance of a model-checker. Note that in addition to the operations to compute the successors of a zone according to $\rightsquigarrow_{k,\mathcal{G}}$, the algorithm uses two more operations to check the emptiness of a zone as well as the inclusion between two zones. Thus, designing efficient algorithms and data-structures for zones is a major issue in developing a verification tool for timed automata, which is addressed in Section 4. Secondly, PASSED holds all encountered states and its size puts a limit on the size of systems we can verify. This raises the research challenges *e.g.* state compression [Ben02], state-space reduction [BJLY98] and approximate techniques [Bal96].

4 DBM: Algorithms and Data Structures

The preceding section presents the key elements needed in symbolic reachability analysis. Recall that the operations on zones are all defined in terms of sets of clock assignments. It is not clear how to compute the result of such an operation. In this section, we describe how to represent zones, compute the operations and check properties on zones. Pseudo code for the operations is given in the appendix.

4.1 DBM basics

Recall that a clock constraint over $\mathcal{C}$ is a conjunction of atomic constraints of the form $x \sim m$ and $x - y \sim n$ where $x, y \in \mathcal{C}$, $\sim \in \{\leq, <, =, >, \geq\}$ and $m, n \in \mathbb{N}$. A zone denoted by a clock constraint D is the maximal set of clock assignments satisfying D. The most important property of zones is that they can can be represented as matrices *i.e.* DBMs (Difference Bound Matrices) [Bel57,Dil89], which have a canonical representation. In the following, we describe the basic structure and properties of DBMs.

To have a unified form for clock constraints we introduce a reference clock $\mathbf{0}$ with the constant value 0. Let $\mathcal{C}_0 = \mathcal{C} \cup \{\mathbf{0}\}$. Then any zone $D \in \mathcal{B}(\mathcal{C})$ can be rewritten as a conjunction of constraints of the form $x - y \preceq n$ for $x, y \in \mathcal{C}_0$, $\preceq \in \{<, \leq\}$ and $n \in \mathbb{Z}$.

Naturally, if the rewritten zone has two constraints on the same pair of variables only the intersection of the two is significant. Thus, a zone can be represented using at most $|\mathcal{C}_0|^2$ atomic constraints of the form $x-y \preceq n$ such that each pair of clocks $x-y$ is mentioned only once. We can then store zones using $|\mathcal{C}_0| \times |\mathcal{C}_0|$ matrices where each element in the matrix corresponds to an atomic constraint. Since each element in such a matrix represents a bound on the difference between two clocks, they are named *Difference Bound Matrices* (DBMs). In the following presentation, we use D_{ij} to denote element (i, j) in the DBM representing the zone D.

To construct the DBM representation for a zone D, we start by numbering all clocks in $\mathcal{C}_0$ as $0, \ldots, n$ and the index for $\mathbf{0}$ is 0. Let each clock be denoted by one row in the matrix. The row is used for storing lower bounds on the difference between the clock and all other clocks, and thus the corresponding column is used for upper bounds. The elements in the matrix are then computed in three steps.

- For each constraint $x_i - x_j \preceq n$ of D, let $D_{ij} = (n, \preceq)$.
- For each clock difference $x_i - x_j$ that is unbounded in D, let $D_{ij} = \infty$. Where ∞ is a special value denoting that no bound is present.
- Finally add the implicit constraints that all clocks are positive, *i.e.* $\mathbf{0} - x_i \leq 0$, and that the difference between a clock and itself is always 0, *i.e.* $x_i - x_i \leq 0$.

As an example, consider the zone $D = x-\mathbf{0} < 20 \wedge y-\mathbf{0} \leq 20 \wedge y-x \leq 10 \wedge x-y \leq -10 \wedge \mathbf{0} - z < 5$. To construct the matrix representation of D, we number the clocks in the order $\mathbf{0}, x, y, z$. The resulting matrix representation is shown below:

$$M(D) = \begin{pmatrix} (0, \leq) & (0, \leq) & (0, \leq) & (5, <) \\ (20, <) & (0, \leq) & (-10, \leq) & \infty \\ (20, \leq) & (10, \leq) & (0, \leq) & \infty \\ \infty & \infty & \infty & (0, \leq) \end{pmatrix}$$

To manipulate DBMs efficiently we need two operations on bounds: comparison and addition. We define that $(n, \preceq) < \infty$, $(n_1, \preceq_1) < (n_2, \preceq_2)$ if $n_1 < n_2$ and $(n, <) < (n, \leq)$. Further we define addition as $b_1 + \infty = \infty$, $(m, \leq) + (n, \leq) = (m+n, \leq)$ and $(m, <) + (n, \preceq) = (m+n, <)$.

Canonical DBMs Usually there are an infinite number of zones sharing the same solution set. However, for each family of zones with the same solution set there is a unique constraint where no atomic constraint can be strengthened without losing solutions.

To compute the canonical form of a given zone we need to derive the tightest constraint on each clock difference. To solve this problem, we use a graph-interpretation of zones. A zone may be transformed to a weighted graph where the clocks in $\mathcal{C}_0$ are nodes and the atomic constraints are edges labeled with bounds. A constraint in the form of $x - y \preceq n$ will be converted to an edge from node y to node x labeled with $(n, \preceq)$, namely the distance from y to x is bounded by n.

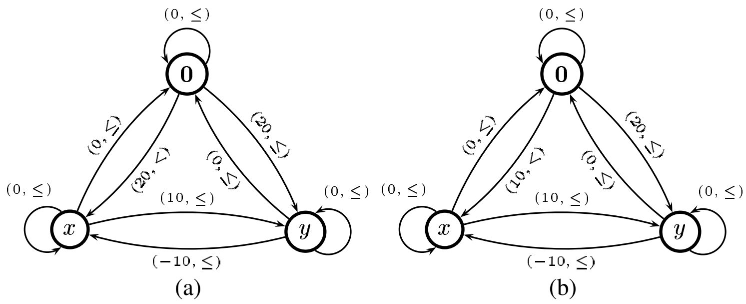


Fig. 8. Graph interpretation of the example zone and its closed form

As an example, consider the zone $x - \mathbf{0} < 20 \wedge y - \mathbf{0} \leq 20 \wedge y - x \leq 10 \wedge$ $x - y \leq -10$. By combining the atomic constraints $y - \mathbf{0} \leq 20$ and $x - y \leq -10$ we derive that $x - \mathbf{0} \leq 10$, *i.e.* the bound on $x - \mathbf{0}$ can be strengthened. Consider the graph interpretation of this zone, presented in Fig. 8(a). The tighter bound on $x - \mathbf{0}$ can be derived from the graph, using the path $\mathbf{0} \to y \to x$, giving the graph in Fig. 8(b). Thus, deriving the tightest constraint on a pair of clocks in a zone is equivalent to finding the shortest path between their nodes in the graph interpretation of the zone. The conclusion is that a canonical, *i.e.* closed, version of a zone can be computed using a shortest path algorithm. Floyd-Warshall algorithm [Flo62] (Algorithm 2) is often used to transform zones to canonical form. However, since this algorithm is quite expensive (cubic in the number of clocks), it is desirable to make all frequently used operations preserve the canonical form, *i.e.* the result of performing an operation on a canonical zone should also be canonical.

Minimal Constraint Systems A zone may contain redundant constraints. For example, a zone contains constraints $x-y < 2, y-z < 5$ and $x-z < 7$. The constraint $x-z < 7$ is obviously redundant because it may be derived from the first two. It is desirable to remove such constraints to store only the minimal number of constraints. Consider, for instance, the zone $x - y \leq 0 \wedge y - z \leq 0 \wedge z - x \leq 0 \wedge 2 \leq x - \mathbf{0} \leq 3$. This is a zone in a minimal form containing only five constraints. The closed form of this zone contains more than 12 constraints. It is known, *e.g.* from [LLPY97], that for each zone there is a minimal constraint system with the same solution set. By computing this minimal form for all zones and storing them in memory using a sparse representation we might reduce the memory consumption for state-space exploration. This problem has been thoroughly investigated in [LLPY97,Pet99,Lar00].

The following is a summary of the published work on the minimal representation of zones. We present an algorithm that computes the minimal form of a closed DBM. Closing a DBM corresponds to computing the shortest path between all clocks. Our goal is to compute the minimal set of bounds for a given shortest path closure. For clarity, the algorithm is presented in terms of directed weighted graphs. However, the results are directly applicable to the graph interpretation of DBMs.

First we introduce some notation: we say that a cycle in a graph is a *zero cycle* if the sum of weights along the cycle is 0, and an edge $x_i \xrightarrow{n_{ij}} x_j$ is *redundant* if there is another path between x_i and x_j where the sum of weights is no larger than n_{ij}.

In graphs without zero cycles we can remove all redundant edges without affecting the shortest path closure [LLPY97]. Further, if the input graph is in shortest path form (as for closed DBMs) all redundant edges can be located by considering alternative paths of length two.

As an example, consider Fig. 9. The figure shows the shortest path closure for a zero-cycle free graph (a) and its minimal form (b). In the graph we find that $x_1 \xrightarrow{9} x_2$ is made redundant by the path $x_1 \xrightarrow{7} x_4 \xrightarrow{2} x_2$ and can thus be removed. Further, the edge $x_3 \xrightarrow{15} x_2$ is redundant due to $x_3 \xrightarrow{5} x_1 \xrightarrow{9} x_2$. Note that we consider edges marked as redundant when searching for new redundant edges. The reason is that we let the redundant edges represent the path making them redundant, thus allowing all redundant edges to be located using only alternative paths of length two. This procedure is repeated until no more redundant edges can be found.

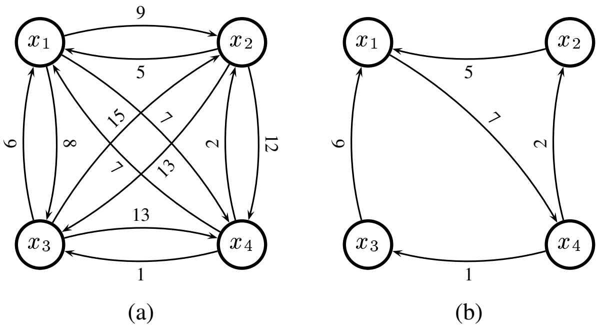


Fig. 9. A zero cycle free graph and its minimal form

This gives the $O(n^3)$ procedure for removing redundant edges presented in Algorithm 3. The algorithm can be directly applied to zero-cycle free DBMs to compute the minimal number of constraints needed to represent a given zone.

However, this algorithm will not work if there are zero-cycles in the graph. The reason is that the set of redundant edges in a graph with zero-cycles is not unique. As an example, consider the graph in Fig. 10(a). Applying the above reasoning on this graph would remove $x_1 \xrightarrow{3} x_3$ based on the path $x_1 \xrightarrow{-2} x_2 \xrightarrow{5} x_3$. It would also remove the edge $x_2 \xrightarrow{5} x_3$ based on the path $x_2 \xrightarrow{2} x_1 \xrightarrow{3} x_3$. But if both these edges are removed it is no longer possible to construct paths leading into x_3. In this example there is a dependence between the edges $x_1 \xrightarrow{3} x_3$ and $x_2 \xrightarrow{5} x_3$ such that only one of them can be considered redundant.

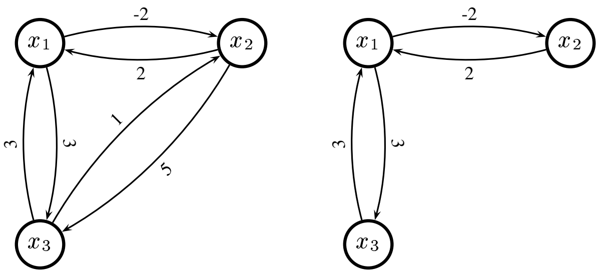


Fig. 10. A graph with a zero-cycle and its minimal form

The solution to this problem is to partition the nodes according to zero-cycles and build a super-graph where each node is a partition. The graph from Fig. 10(a) has two partitions, one containing x_1 and x_2 and the other containing x_3. To compute the edges in the super-graph we pick one representative for each partition and let the edges between the partitions inherit the weights from edges between the representatives. In our example, we choose x_1 and x_3 as representatives for their equivalence classes. The edges in the graph are then $\{x_1, x_2\} \xrightarrow{3} \{x_3\}$ and $\{x_3\} \xrightarrow{3} \{x_1, x_2\}$. The super-graph is clearly zero-cycle free and can be reduced using Algorithm 3. This small graph can not be reduced further. The relation between the nodes within a partition is uniquely defined by the zero-cycle and all other edges may be removed. In our example all edges within each equivalence class are part of the zero-cycle and thus none of them can be removed. Finally the reduced super-graph is connected to the reduced partitions. In our example we end up with the graph in Fig. 10(b). Pseudo-code for the reduction-procedure is shown in Algorithm 4.

Now we have an algorithm for computing the minimal number of edges to represent a given shortest path closure that can be used to compute the minimal number of constraints needed to represent a given zone.

4.2 Basic Operations on DBMs

This subsection presents all the basic operations on DBMs except the ones for zone-normalization, needed in symbolic state space exploration of timed automata, both for forwards and backwards analysis. The two operations for zone-normalization are presented in the next subsection.

First note that even if a verification tool only explores the state space in one direction all operations are still useful for other purposes, *e.g.* for generating diagnostic traces. The operations are illustrated graphically in Fig. 11.

To simplify the presentation we assume that the input zones are consistent and in canonical form. The basic operations on DBMs can be divided into two classes:

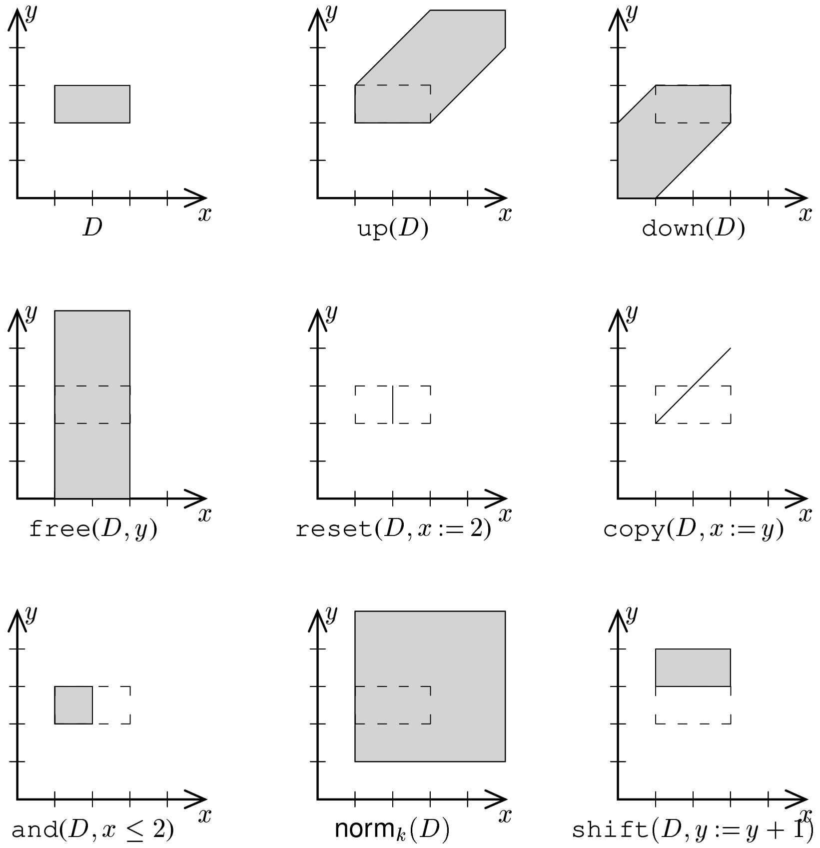


Fig. 11. DBM operations applied to the same zone where for $\mathsf{norm}_k(D)$, k is defined by $k(x) = 2$ and $k(y) = 1$

1. **Property-Checking:** This class includes operations to check the consistency of a DBM, the inclusion between zones, and whether a zone satisfies a given atomic constraint.
2. **Transformation:** This class includes operations to compute the strongest postcondition and weakest precondition of a zone according to conjunction with guards, time delay and clock reset.

Property-Checking

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

For a zone to be inconsistent there must be at least one pair of clocks where the upper bound on their difference is smaller than the lower bound. For DBMs this can be checked by searching for negative cycles in the graph interpretation. However, the most efficient way to implement a consistency check is to detect when an upper bound is set to lower value than the corresponding lower bound and mark the zone as inconsistent by setting D_{00} to a negative value. For a zone in canonical form this test can be performed locally. To check if a zone is inconsistent it will then be enough to check whether D_{00} is negative.

`relation`(D, D') Another key operation in state space exploration is inclusion checking for the solution sets of two zones. For DBMs in canonical form, the condition that $D_{ij} \leq D'_{ij}$ for all clocks $i, j \in \mathcal{C}_0$ is necessary and sufficient to conclude that $D \subseteq D'$. Naturally the opposite condition applies to checking if $D' \subseteq D$. This allows for the combined inclusion check described in Algorithm 5.

`satisfied`$(D, x_i - x_j \preceq m)$ Sometimes it is desirable to non-destructively check if a zone satisfies a constraint, *i.e.* to check if the zone $D \wedge x_i - x_j \preceq m$ is consistent without altering D. From the definition of the `consistent`-operation we know that a zone is consistent if it has no negative cycles. Thus, checking if $D \wedge x_i - x_j \preceq m$ is non-empty can be done by checking if adding the guard to the zone would introduce a negative cycle. For a DBM on canonical form this test can be performed locally by checking if $(m, \preceq) + D_{ji}$ is negative.

Transformations

up(D) The $\mathtt{up}$ operation computes the strongest postcondition of a zone with respect to delay, *i.e.* $\mathtt{up}(D)$ contains the clock assignments that can be reached from D by delay. Formally, this operation is defined as $\mathtt{up}(D) = \{u + d \mid u \in D, d \in \mathbb{R}_+\}$.

Algorithmically, $\mathtt{up}$ is computed by removing the upper bounds on all individual clocks (In a DBM all elements D_{i0} are set to ∞). This is the same as saying that any clock assignment in a given zone may delay an arbitrary amount of time. The property that all clocks proceed at the same speed is ensured by the fact that constraints on the differences between clocks are not altered by the operation.

This operation preserves the canonical form, *i.e.* applying $\mathtt{up}$ to a canonical DBM will result in a new canonical DBM. The $\mathtt{up}$ operation is also presented in Algorithm 6.

down(D) This operation computes the weakest precondition of D with respect to delay. Formally $\mathtt{down}(D) = \{u \mid u + d \in D, d \in \mathbb{R}_+\}$, *i.e.* the set of clock assignments that can reach D by some delay d. Algorithmically, $\mathtt{down}$ is computed by setting the lower bound on all individual clocks to $(0, \leq)$. However due to constraints on clock differences this algorithm may produce non-canonical DBMs. As an example, consider the zone in Fig. 12(a). When $\mathtt{down}$ is applied to this zone (Fig. 12(b)), the lower bound on x is 1 and not 0, due to constraints on clock differences. Thus, to obtain an algorithm that produce canonical DBMs the difference constraints have to be taken into account when computing the new lower bounds.

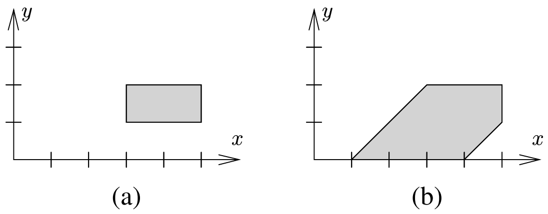


Fig. 12. Applying $\mathtt{down}$ to a zone.

To compute the lower bound for a clock x, start by assuming that all other clocks y_i have the value 0. Then examine all difference constraints $y_i - x$ and compute a new lower bound for x under this assumption. The new bound on $\mathbf{0} - x$ will be the minimum bound on $y_i - x$ found in the DBM. Pseudo-code for $\mathtt{down}$ is presented in Algorithm 7.

and($D, x_i - y_j \preceq b$) The most used operation in state-space exploration is conjunction, *i.e.* adding a constraint to a zone. The basic step of the $\mathtt{and}$ operation is to check if $(b, \preceq) < D_{ij}$ and in this case set the bound D_{ij} to $(b, \preceq)$. If the bound has been altered, *i.e.* if adding the guard affected the solution set, the DBM has to be put back on canonical form. One way to do this would be to use the generic shortest path algorithm, however for this particular case it is possible to derive a specialization of the algorithm allowing re-canonicalization in $O(n^2)$ instead of $O(n^3)$.

The specialized algorithm takes the advantage that D_{ij} is the only bound that has been changed. Since the Floyd-Warshall algorithm is insensitive to how the nodes in the graph are ordered, we may decide to treat x_i and x_j last. The outer loop of Algorithm 2 will then only affect the DBM twice, for $k = i$ and $k = j$. This allows the canonicalisation algorithm to be reduced to checking, for all pairs of clocks in the DBM, if the path via either x_i or x_j is shorter than the direct connection. The pseudo code for this is presented in Algorithm 8.

free(D, x) The `free` operation removes all constraints on a given clock, *i.e.* the clock may take any positive value. Formally this is expressed as $\texttt{free}(D, x) = \{u[x = d] \mid u \in D, d \in \mathbb{R}_+\}$. In state-space exploration this operation is used in combination with conjunction, to implement reset operations on clocks. It can be used in both forwards and backwards exploration, but since forwards exploration allows other more efficient implementations of reset, `free` is only used when exploring the state-space backwards.

A simple algorithm for this operation is to remove all bounds on x in the DBM and set $D_{0x} = (0, \leq)$. However, the result may not be on canonical form. To obtain an algorithm preserving the canonical form, we change how new difference constraints regarding x are derived. We note that the constraint $\mathbf{0} - x \leq 0$ can be combined with constraints of the form $y - \mathbf{0} \preceq m$ to compute new bounds for $y - x$. For instance, if $y - \mathbf{0} \leq 5$ we can derive that $y - x \leq 5$. To obtain a DBM on canonical form we derive bounds for D_{yx} based on D_{y0} instead of setting $D_{yx} = \infty$.In Algorithm 9 this is presented as pseudo code.

reset($D, x := m$) In forwards exploration this operation is used to set clocks to specific values, *i.e.* $\texttt{reset}(D, x := m) = \{u[x = m] \mid u \in D\}$. Without the requirement that output should be on canonical form, `reset` can be implemented by setting $D_{x0} = (m, \leq)$, $D_{0x} = (-m, \leq)$ and remove all other bounds on x. However, if we instead of removing the difference constraints compute new values using constraints on the other clocks, like in the implementation of `free`, we obtain an implementation that preserve the canonical form. Such an implementation is presented in Algorithm 10.

copy($D, x := y$) This is another operation used in forwards state-space exploration. It copies the value of one clock to another. Formally, we define $\texttt{copy}(D, x := y)$ as $\{u[x = u(y)] \mid u \in D\}$. As `reset`, `copy` can be implemented by assigning $D_{xy} = (0, \leq)$, $D_{yx} = (0, \leq)$, removing all other bounds on x and re-canonicalize the zone. However, a more efficient implementation is obtained by assigning $D_{xy} = (0, \leq)$, $D_{yx} = (0, \leq)$ and then copy the rest of the bounds on x from y. For pseudo code, see Algorithm 11

$\texttt{shift}(D, x := x + m)$ The last reset operation is shifting a clock, *i.e.* adding or subtracting a clock with an integer value, *i.e.* $\texttt{shift}(D, x := x+m) = \{u[x = u(x)+m] \mid u \in D\}$. The definition gives a hint on how to implement the operation. We can view the shift operation as a substitution of $x - m$ for x in the zone. With this reasoning m is added to the upper and lower bounds of x. However, since lower bounds on x are represented by constraints on $y - x$, m is subtracted from all those bounds. This operation is presented in pseudo-code in Algorithm 12

4.3 Zone-Normalization

The operations for zone-normalization are to transform zones which may contain arbitrarily large constants to zones containing only bounded constants in order to obtain a finite zone-graph.

$\mathsf{norm}_k(D)$ For a timed automaton and a safety property to be checked, that contain no difference constraints, the k-normalization $\mathsf{norm}_k(D)$ is needed, and it can be computed based on the canonical form of D (see Section 3). It is to remove all upper bounds higher than the maximal constants and lower all lower bounds higher than the maximal constants down to the maximal constants. The result of $\mathsf{norm}_k(D)$ is illustrated graphically in Fig. 11.

In the canonical DBM representation of a zone, the operation consists of two steps: first, remove all bounds $x - y \preceq m$ such that $(m, \preceq) > (k(x), \leq)$ and second, set all bounds $x - y \preceq m$ such that $(m, \preceq) < (-k(y), <)$ to $(-k(y), <)$. Pseudo-code for k-normalization is given in Algorithm 13 where k_i denotes $k(x_i)$.

The k-normalization will not preserve the canonical form of a DBM, and the best way to put the result back on canonical form is to use Algorithm 2.

$\mathsf{norm}_{k,\mathcal{G}}(D)$ For automata containing difference constraints in the guards, it is more complicated and expensive to compute the normalized zones. Assume an automataon $\mathcal{A}$ containing the set of difference constraints $\mathcal{G}$ and the maximal clock constants bounded by a clock ceiling k. Assume a zone D of $\mathcal{A}$ to be normalized. According to the semantical characterization for $\mathsf{norm}_{k,\mathcal{G}}(D)$ in Definition 10 we know that if a difference constraint is not satisfied by any assignment in the zone D, it should not be satisfied by any assignment in the normalized one, $\mathsf{norm}_{k,\mathcal{G}}(D)$, and if all assignments in D satisfy a difference constraint then so should all assignments in $\mathsf{norm}_{k,\mathcal{G}}(D)$. This leads to a core normalization algorithm consisting of three steps:

1. Collect all difference constraints g used as guards in $\mathcal{A}$ such that
   (a) $g \wedge D$ is empty. This is the case when g is outside of D.
   (b) $\neg g \wedge D$ is empty. That is the case when g contains D completely.

   Let $\mathcal{G}_{\text{unsat}} = \{g | g \wedge D = \emptyset\} \cup \{\neg g | \neg g \wedge D = \emptyset\}$

2. Compute $\mathsf{norm}_k(D)$, that is, to run the k-normalization without considering the difference constraints.
3. Subtract (or cut) the k-normalized zone of D by all difference constraints in $\mathcal{G}_{\mathrm{unsat}}$, that is to compute $\mathsf{norm}_k(D) \wedge \neg\mathcal{G}_{\mathrm{unsat}}$.

The last step is to make sure that none of the collected difference constraints are satisfied after the k-normalization. In Algorithm 14, the core normalization is given as pseudo code. The set $\mathcal{G}_{\mathrm{d}}$ used in the algorithm is the set of difference constraints appearing in the automaton under consideration with the maximal clock constants bounded by a given clock ceiling k as input.

It appears to be the case that $\mathsf{norm}_k(D) \wedge \neg\mathcal{G}_{\mathrm{unsat}}$ is the normalized zone we are looking for. Unfortunately this is not. The core normalization does not handle the third case when a difference constraint splits the zone D to be normalized. That is, there is a guard g such that $g \wedge D \neq \emptyset$ and $\neg g \wedge D \neq \emptyset$. In this case, we need to split D by g using Algorithm 15, and then apply the core normalization algorithm to the parts of D separately, which are the sub-zones of D resulted from the splitting. Each of the sub-zone D_i should satisfy either $D_i \wedge g = \emptyset$ or $D_i \wedge g = D_i$ for all $g \in \mathcal{G}$. The union of the normalized sub-zones using the above core normalization algorithm is what we are looking for, that is $\mathsf{norm}_{k,\mathcal{G}}(D)$.

The complete normalization procedure is presented in Algorithm 16. The splitting, denoted by split in the description, is used as a preprocessing step and then the core normalization algorithm (*i.e.* Algorithm 14) is applied to all the resulted sub-zones resulted from the splitting.

Finally, the symbolic transition relation $\rightsquigarrow_{k,\mathcal{G}}$ can be computed as follows: If $\langle l, D\rangle \rightsquigarrow \langle l', D'\rangle$, $\langle l, D\rangle \rightsquigarrow_{k,\mathcal{G}} \langle l', D''\rangle$ for all $D'' \in Q$ used in Algorithm 16, *i.e.* the algorithm for $\mathsf{norm}_{k,\mathcal{G}}(D')$.

To demonstrate the normalization procedure we apply it to the zone for location S_2 in our counter example. The difference constraints in the example are $g_1 \equiv x - z < 1$ and $g_2 \equiv z - y < 1$. The zone contains both clock assignments satisfying g_1 and assignments satisfying its negation, and thus we have to split the zone with respect to this constraint prior to normalization, giving the zones below.

$$\begin{cases} y - x < -2 \\ y - z < -1 \\ z - x \leq 0 \\ \mathbf{0} - x < -2 \\ \mathbf{0} - z < -1 \\ x - z < 1 \end{cases} \qquad \begin{cases} y - x < -2 \\ y - z \leq 0 \\ \mathbf{0} - x < -2 \\ z - x \leq -1 \end{cases}$$

(a) satisfying g_1 (b) satisfying $\neg g_1$

Zone (a) above does not contain any clock assignments satisfying g_2 and thus it will not be split further. Zone (b) however needs to be split into two parts satisfying g_2 and $\neg g_2$. This gives us the following zones to normalize by the core normalization procedure.

$$\begin{cases} y - x < -2 \\ y - z < -1 \\ z - x \leq 0 \\ \mathbf{0} - x < -2 \\ \mathbf{0} - z < -1 \\ x - z < 1 \end{cases} \quad \begin{cases} y - x < -2 \\ y - z \leq 0 \\ \mathbf{0} - x < -2 \\ z - x \leq -1 \end{cases} \quad \begin{cases} y - x < -2 \\ y - z \leq -1 \\ z - x \leq -1 \\ \mathbf{0} - x < -2 \\ \mathbf{0} - z \leq -1 \end{cases}$$

(a) g_1 and $\neg g_2$ (b) $\neg g_1$ and g_2 (c) $\neg g_1$ and $\neg g_2$

The sets of difference constraints not satisfied by the zones (a), (b) and (c) shown above are: $\mathcal{G}^{(a)}_{\mathrm{unsat}} = \{\neg g_1, g_2\}$, $\mathcal{G}^{(b)}_{\mathrm{unsat}} = \{g_1, \neg g_2\}$, $\mathcal{G}^{(c)}_{\mathrm{unsat}} = \{g_1, g_2\}$ respectively. We apply k-normalization to each of them, giving:

$$\begin{cases} y - x < -1 \\ y - z < -1 \\ z - x \leq 0 \\ \mathbf{0} - x < -1 \\ \mathbf{0} - z < -1 \\ x - z < 1 \end{cases} \quad \begin{cases} y - x < -1 \\ y - z \leq 0 \\ \mathbf{0} - x < -1 \\ x - z \geq 1 \end{cases} \quad \begin{cases} y - x < -1 \\ y - z \leq -1 \\ z - x \leq -1 \\ \mathbf{0} - x < -1 \\ \mathbf{0} - z \leq -1 \end{cases}$$

(A) g_1 and $\neg g_2$ (B) $\neg g_1$ and g_2 (C) $\neg g_1$ and $\neg g_2$

Since the k-normalized zones (A), (B) and (C) shown above do not enable any constraint in $\mathcal{G}_{\mathrm{unsat}}$, we need not to subtract the corresponding difference constraints from the zones. Finally, we note that, as the un-normalized zones (a), (b) and (c), none of the normalized zones (A), (B) and (C) intersects with $g_1 \wedge g_2$; the transition from S_2 to S_3 is not enabled by the normalization procedure.

4.4 Zones in Memory

This section describes a number of techniques for storing zones in computer memory. The section starts by describing how to map DBM elements on machine words. It continues by discussing how to place two-dimensional DBMs in linear memory and ends by describing how to store zones using a sparse representation.

Storing DBM Elements To store a DBM element in memory we need to keep track of the integer limit and whether it is strict or not. The range of the integer limit is typically much lower than the maximum value of a machine word and the strictness can be stored using just one bit. Thus, it is possible to store both the limit and the strictness in different parts of the same machine word. Since comparing and adding DBM elements are frequently used operations it is crucial for the performance of a DBM package that they can be efficiently implemented for the chosen encoding. Fortunately, it is possible to construct an encoding of bounds in machine words, where checking if b_1 is less than b_2 can be performed by checking if the encoded b_1 is smaller than the encoded b_2.

The encoding we propose is to use the least significant bit (LSB) of the machine word to store whether the bound is strict or not. Since strict bounds are smaller than non-strict we let a set (1) bit denote that the bound is non-strict while an unset (0) bit denote that the bound is strict. The rest of the bits in the machine word are used to store the integer bound. To encode ∞ we use the largest positive number that fit in a machine word (denoted `MAX_INT`).

For good performance we also need an efficient implementation of addition of bounds. For the proposed encoding Algorithm 17 adds two encoded bounds b_1 and b_2. The symbols & and | in the algorithm are used to denote bitwise-and and bitwise-or, respectively.

Placing DBMs in Memory Another issue is how to store two-dimensional DBMs in linear memory. In this section we present two different techniques and give a brief comparison between them. The natural way to put matrices in linear memory is to store the elements by row (or by column), *i.e.* each row of the matrix is stored consequently in memory. This layout has one big advantage, its good performance. This advantage is mainly due to the simple function for computing the location of a given element in the matrix: $loc(x, y) = x*(n+1)+y$. This function can (on most computers) be computed in only two instructions. This is important since all accesses to DBM elements use this function. How the different DBM elements are place in memory with this layout if presented in Fig. 13(a).

The second way to store a DBM in linear memory is based on a layered model where each layer consists of the bounds between a clock and the clocks with lower index in the DBM. In this representation it is cheap to implement local clocks, since all information about the local clocks are localised at the end of the DBM. The drawback with this layout is the more complicated function from DBM indices to memory locations. For this layout we have:

$$loc(x, y) = \begin{cases} y * (y+1) + x & \text{if } x \leq y \\ x * x + y & \text{otherwise} \end{cases}$$

This adds at least two instructions (one comparison and one conditional jump) to the transformation. This may not seem such a huge overhead, but it is clearly noticeable. The cache performance is also worse when using this layout than when storing the DBMs row-wise. This layout is illustrated in Fig. 13(b).

The conclusion is that unless the tool under construction supports adding and removing clocks dynamically the row-wise mapping should be used. On the other hand, if the tool supports local clocks the layered mapping may be preferable since no reordering of the DBM is needed when entering or leaving a clock scope.

Storing Sparse Zones In most verification tools, the majority of the zones are kept in the set of states already visited during verification. They are used as a reference to ensure termination by preventing states from being explored more than once. For the

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix} \quad \begin{pmatrix} 0 & 2 & 6 & 12 \\ 1 & 3 & 7 & 13 \\ 4 & 5 & 8 & 14 \\ 9 & 10 & 11 & 15 \end{pmatrix}$$

(a) Row wise (b) Layered

Fig. 13. Different layouts of DBMs in memory

states in this set we may benefit from storing only the minimal number of constraints using a sparse representation.

A straight forward implementation is to store a sparse zone as a vector of constraints of the form $\langle x, y, b \rangle$. We may save additional memory by omitting implicit constraints, such as $\mathbf{0} - x \leq 0$. A downside with using sparse zones is that each constraint require twice the amount of memory needed for a constraint in a full DBM, since the sparse representation must store clock indices explicitly. Thus, unless half of the constraints in a DBM are redundant we do not gain from using sparse zones.

A nice feature of the sparse representation is that checking whether a zone D_f represented as a full DBM is included in a sparse zone D_s may be implemented without computing the full DBM for D_s. It suffices to check for all constraints in D_s that the corresponding bound in D_f is tighter. However, to check if $D_s \subseteq D_f$ we have to compute the full DBM for D_s.

5 UPPAAL

In the last decade, there have been a number of tools developed based on timed automata to model and verify real time systems, notably Kronos [Yov97] and UPPAAL [LPY97]. As an example, we give a brief introduction to the UPPAAL tool (www.uppaal.com).

UPPAAL is a tool box for modeling, simulation and verification of timed automata, based on the algorithms and data-structures presented in previous sections. The tool was released for the first time in 1995, and since then it has been developed and maintained in collaboration between Uppsala University and Aalborg University.

5.1 Modeling with UPPAAL

The core of the UPPAAL modeling language is networks of timed automata. In addition, the language has been further extended with features to ease the modeling task and to guide the verifier in state space exploration. The most important of these are shared integer variables, urgent channels and committed locations.

Networks of Timed Automata A *network of timed automata* is the parallel composition $A_1|\cdots|A_n$ of a set of timed automata $A_1,\ldots,A_n$, called processes, combined into a single system by the CCS parallel composition operator with all external actions hidden. Synchronous communication between the processes is by hand-shake synchronization using input and output actions; asynchronous communication is by shared variables as described later. To model hand-shake synchronization, the action alphabet Σ is assumed to consist of symbols for input actions denoted $a?$, output actions denoted $a!$, and internal actions represented by the distinct symbol τ.

An example system composed of two timed automata is shown in Fig. 14. The network models a time-dependent light-switch (to the left) and its user (to the right). The user and the switch communicate using the label *press*. The user can press the switch (press!) and the switch waits to be pressed (press?). The product automaton, *i.e.* the automaton describing the combined system is shown in Fig. 15.

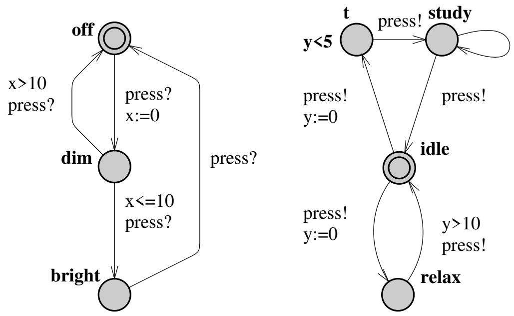


Fig. 14. Network of Timed Automata

The semantics of networks is given as for single timed automata in terms of transition systems. A state of a network is a pair $\langle l, u\rangle$ where l denotes a vector of current locations of the network, one for each process, and u is as usual a clock assignment remembering the current values of the clocks in the system. A network may perform two types of transitions, delay transitions and discrete transitions. The rule for delay transitions is similar to the case of single timed automata where the invariant of a location vector is the conjunction of the location invariants of the processes. There are two rules for discrete transitions defining local actions where one of the processes makes a move on its own, and synchronizing actions where two processes synchronize on a channel and move simultaneously.

Let l_i stand for the ith element of a location vector l and $l[l'_i/l_i]$ for the vector l with l_i being substituted with l'_i. The transition rules are as follows:

- $\langle l, u\rangle \xrightarrow{d} \langle l, u + t\rangle$ if $u \in I(l)$ and $(u + d) \in I(l)$, where $I(l) = \bigwedge I(l_i)$
- $\langle l, u\rangle \xrightarrow{\tau} \langle l[l'_i/l_i], u'\rangle$ if $l_i \xrightarrow{g,\tau,r} l'_i, u \in g, u' = [r \mapsto 0]u, u' \in I(l[l'_i/l_i])$

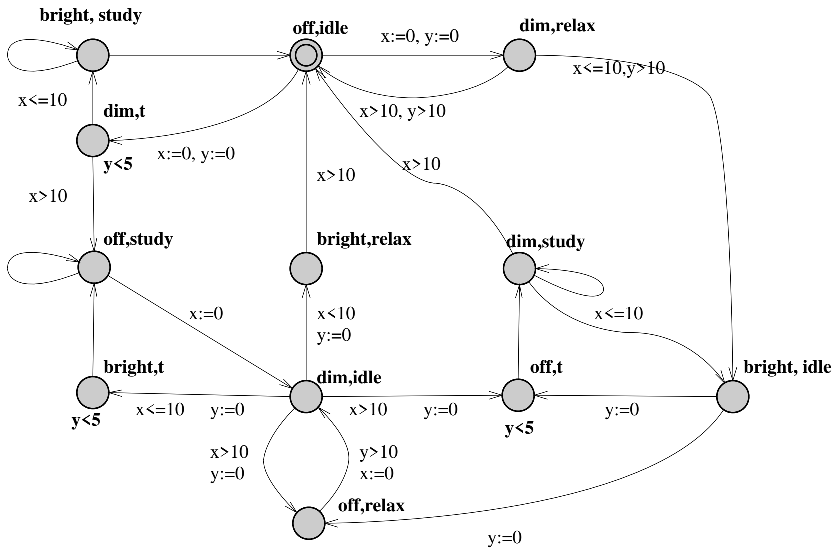


Fig. 15. Product Automaton for the Network in Fig. 14

– $\langle l, u\rangle \xrightarrow{\tau} \langle l[l'_i/l_i][l'_j/l_j], u'\rangle$ if there exist $i \neq j$ such that

1. $l_i \xrightarrow{g_i, a?, r_i} l'_i$, $l_j \xrightarrow{g_j, a!, r_j} l'_j$ and $u \in g_i \wedge g_j$, and
2. $u' = [r_i \cup r_j \mapsto 0]u$ and $u' \in I(l[l'_i/l_i][l'_j/l_j])$.

Note that a network is a closed system which may not perform any external action. In fact, the CCS hiding operator is embedded in the above rules.

Shared Integer Variables Clocks may be considered as typed variables with type *clock*. In UPPAAL, one may also use integer variables and arrays of integers, each with a bounded domain and an initial value. Predicates over the integer variables can be used as guards on the edges of an automaton process and the integer variables may be updated using resets on the edges. In the current version of UPPAAL, the syntax related to integer variables resembles the standard C syntax. Both integer guards and integer resets are standard C expressions with the restriction that guards can not have side-effects.

The semantics of networks can be defined accordingly. The clock assignment u in the state configuration $\langle l, u\rangle$ can be extended to store the values of integer variables in addition to clocks. Since delay does not affect the integer variables, the delay transitions are the same as for networks without integer variables. The action transitions are extended in the natural way, *i.e.* for an action transition to be enabled the extended clock assignment must also satisfy all integer guards on the corresponding edges and when a transition is taken the assignment is updated according to the integer and clock resets.

There is a problem with variable updating in a synchronizing transition where one of the processes participating in the transition updates a variable used by the other. In UPPAAL, for a synchronization transition, the resets on the edge with an output-label is performed before the resets on the edge with an input-label. This destroys the symmetry of input and output actions. But it gives a natural and clear semantics for variable updating. The transition rule for synchronization is modified accordingly:

– $\langle l, u\rangle \xrightarrow{\tau} \langle l[l_i'/l_i][l_j'/l_j], u'\rangle$ if there exist $i \neq j$ such that

  1. $l_i \xrightarrow{g_i, a?, r_i} l_i'$, $l_j \xrightarrow{g_j, a!, r_j} l_j'$ and $u \in g_i \wedge g_j$, and
  2. $u' = [r_i \mapsto 0]([r_j \mapsto 0]u)$ and $u' \in I(l[l_i'/l_i][l_j'/l_j])$

Urgent Channels To model urgent synchronizing transitions, which should be taken as soon as they are enabled, UPPAAL supports a notion of urgent channels. An urgent channel works much like an ordinary channel, but with the exception that if a synchronization on an urgent channel is possible the system may not delay. Interleaving with other enabled action transitions, however, is still allowed. In order to keep clock constraints representable using convex zones, clock guards are not allowed on edges synchronizing on urgent channels.

To illustrate why this restriction is necessary, consider the network shown in Fig. 16. Both processes may independently go from their first state to their second state. In the second state, the first process must delay for at least 10 time units before it is allowed to synchronize on the urgent channel u. In the second state, the other process must delay for at least 5 time units before it is allowed to synchronize on the urgent channel u. As soon as both processes have spent the minimal time periods required in their second state, they should synchronize and move to their third state. The problem is in [**S2,T2**] where the zone may be for example $(x \geq 10 \wedge y = 5) \vee (y \geq 5 \wedge x = 10)$ which is a non-convex zone.

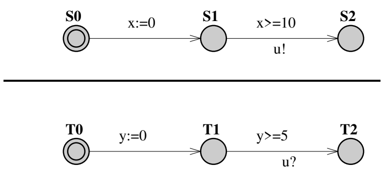


Fig. 16. An example of a network with non convex timing regions.

For this example, the problem can be solved by splitting the non-convex zone into two convex ones. But in general, the splitting is a computationally expensive operation. In UPPAAL, we decided to avoid such operations for the sake of efficiency. So only integer guards are allowed on edges involving synchronizations on urgent channels.

Committed Locations To model atomic sequences of actions, *e.g.* atomic broadcast or multicast, UPPAAL supports a notion of *committed locations*. A committed location is a location where no delay is allowed. In a network, if any process is in a committed location then only action transitions starting from such a committed location are allowed. Thus, processes in committed locations may be interleaved only with processes in a committed location.

Syntactically, each process A_i in a network may have a subset $N_i^C \subseteq N_i$ of locations marked as committed locations. Let $C(l)$ denote the set of locations in l, that are committed. For the same reason as in the case of urgent channels, as a syntactical restriction, no clock constraints but predicates over integer variables are allowed to appear in a guard on an outgoing edge from a committed location.

The transition rules are given in the following, where $\rightarrow_c$ denotes the transition relation for a network with committed locations and $\rightarrow$ denotes the transition relation for the same network without considering the committed locations.

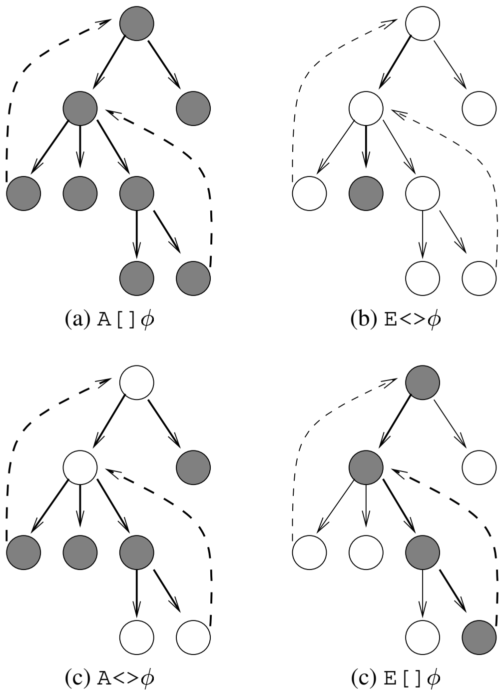
- $\langle l, u\rangle \xrightarrow{d}_c \langle l, u+d\rangle$ if $\langle l, u\rangle \xrightarrow{d} \langle l, u+d\rangle$ and $C(l) = \emptyset$
- $\langle l, u\rangle \xrightarrow{\tau}_c \langle l[l_i'/l_i], u'\rangle$ if
  1. $\langle l, u\rangle \xrightarrow{\tau} \langle l[l_i'/l_i], u'\rangle$, and
  2. Either $l_i \in C(l)$ or $C(l) = \emptyset$
- $\langle l, u\rangle \xrightarrow{\tau}_c \left\langle l[l_i'/l_i][l_j'/l_j], u'\right\rangle$ if
  1. $\langle l, u\rangle \xrightarrow{\tau} \left\langle l[l_i'/l_i][l_j'/l_j], u'\right\rangle$, and
  2. Either $l_i \in C(l)$, $l_j \in C(l)$ or $C(l) = \emptyset$

5.2 Verifying with UPPAAL

The model checking engine of UPPAAL is designed to check a subset of TCTL formula [ACD90] for networks of timed automata. The formulas should be one of the following forms:

- `A[]` ϕ — Invariantly ϕ.
- `E<>` ϕ — Possibly ϕ.
- `A<>` ϕ — Always Eventually ϕ.
- `E[]` ϕ — Potentially Always ϕ.
- ϕ `-->` ψ — ϕ always leads to ψ. This is the shorthand for `A[]` ($\phi \Rightarrow$ `A<>`ψ).

where ϕ, ψ are local properties that can be checked locally on a state, *i.e.* boolean expressions over predicates on locations and integer variables, and clock constraints in $\mathcal{B}(\mathcal{C})$.



(a) `A[]`ϕ (b) `E<>`ϕ

(c) `A<>`ϕ (c) `E[]`ϕ

Fig. 17. (T)CTL-formulae

The transition system of a network may be unfolded into an infinite tree containing states and transitions. The semantics of the formulas are defined over such a tree. The letters A and E are used to quantify over paths. A is used to denote that the given property should hold for all paths of the tree while E denotes that there should be at least one path of the tree where the property holds. The symbols `[]` and <> are used to quantify over states within a path. `[]` denotes that all states on the path should satisfy the property, while <> denotes that at least one state in the execution satisfies the property. In Fig. 17 the four basic property types are illustrated using execution trees, where the dashed arrows are used to denote repetitions in the trees. The states satisfying ϕ are denoted by filled nodes and edges corresponding to the paths are highlighted using bold arrows.

The two types of properties most commonly used in verification of timed systems are `E<>`ϕ and `A[]`ψ. They are dual in the sense that `E<>`ϕ is satisfied if and only if `A[]`$\neg\phi$ is not satisfied. This type of properties are often classified as safety properties, *i.e.* meaning that the system is safe in the sense that a specified hazard can not occur. It is also possible to transform so called bounded liveness properties, *i.e.* properties stating that some desired state will be reached within a given time, into safety properties using observer automata [ABL98] or by annotating the model [LPY98]. For example, to check if an automaton will surely reach a location l within 10 time units, we use one clock x (set to 0 initially) and introduce a boolean variable l_b (set to false initially). For each incoming edge to l in the automaton, set l_b to true. Then if the automaton satisfies the invariant property "$x \leq 10 \vee l_b$", it will surely reach l within 10 time units provided that the automaton contains no zeno loops which stop time to progress.

The other three types of properties are commonly classified as unbounded liveness properties, *i.e.* they are used to express and check for global progress. These properties are not commonly used in UPPAAL case-studies. It seems to be the case that bounded liveness properties are more important for timed systems.

5.3 The UPPAAL Architecture

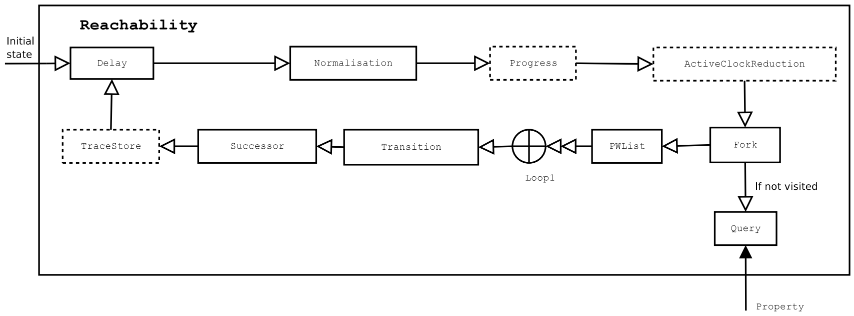


Fig. 18. Schematic view of the reachability pipeline in UPPAAL.

To provide a system that is both efficient, easy to use and portable, UPPAAL is split into two components, a graphical user interface written in Java and a verification engine written in C++. The engine and the GUI communicate using a protocol, allowing the verification to be performed either on the local workstation or on a powerful server in a network.

To implement the reachability analysis algorithm 1, the UPPAAL verification engine is organized as a pipeline that incarnates the natural data flow in the algorithm. A sketch of this pipeline is shown in Fig. 18. This architecture simplifies both activating and deactivating optimizations at runtime by inserting and removing stages dynamically, and introducing new optimizations and features in the tool by implementing new or changing existing stages.

In addition to the zone-manipulation algorithms described in Section 4 and the pipeline architecture, in UPPAAL a number of optimizations have been implemented:

- Minimal constraint systems [LLPY97] and CDDs [LPWY99,BLP+99], to reduce memory consumption by removing redundant information in zones before storing them.
- Selective storing of states in PASSED [LLPY97], where static analysis is used to detect states that can be omitted safely from PASSED without losing termination.

- Compression [Ben01] and sharing [BDLY03,DBLY03] of state data, to reduce the memory consumption of PASSED and WAIT.
- Active clock reduction [DY96], that use live-range analysis to determine when the value of a clock is irrelevant. This does not only reduce the size of individual states but also the perceived state-space.
- Supertrace [Hol91] and Hash Compaction [WL93,SD95] where already visited states are stored only as hash signatures, and Convex-hull approximation [Bal96] where convex hulls are used to approximate unions of zones, for reducing memory consumption at a risk of inconclusive results.

Acknowledgements:

We would like to thank Pavel Krcal for pointing out an error in an early version of this document.

Appendix: Pseudo-Code for Operations on DBMs

Algorithm 2 `close`(D): Floyds algorithm for computing shortest path

for $k := 0$ **to** n **do**
  for $i := 0$ **to** n **do**
    for $j := 0$ **to** n **do**
      $D_{ij} := \min(D_{ij}, D_{ik} + D_{kj})$
    end for
  end for
end for

Algorithm 3 Reduction of Zero-Cycle Free Graph G with n nodes

for $i := 1$ **to** n **do**
  for $j := 1$ **to** n **do**
    for $k := 1$ **to** n **do**
      if $G_{ik} + G_{kj} \leq G_{ij}$ **then**
        Mark edge $i \rightarrow j$ as redundant
      end if
    end for
  end for
end for
Remove all edges marked as redundant.

Algorithm 4 Reduction of negative-cycle free graph G with n nodes

```
for i := 1 to n do
  if Node_i is not in a partition then
    Eq_i = ∅
    for j := i to n do
      if G_ij + G_ji = 0 then
        Eq_i := Eq_i ∪ {Node_i}
      end if
    end for
  end if
end for
Let G' be a graph without nodes.
for each Eq_i do
  Pick one representative Node_i ∈ Eq_i
  Add Node_i to G'
  Connect Node_i to all nodes in G' using weights from G.
end for
Reduce G'
for each Eq_i do
  Add one zero cycle containing all nodes in Eq_i to G'
end for
```

Algorithm 5 relation(D, D')

```
f_{D⊆D'} := tt
f_{D⊇D'} := tt
for i := 0 to n do
  for j := 0 to n do
    f_{D⊆D'} := f_{D⊆D'} ∧ (D_ij ≤ D'_ij)
    f_{D⊇D'} := f_{D⊇D'} ∧ (D_ij ≥ D'_ij)
  end for
end for
return ⟨f_{D⊆D'}, f_{D⊇D'}⟩
```

Algorithm 6 up(D)

```
for i := 1 to n do
  D_i0 := ∞
end for
```

Algorithm 7 $\texttt{down}(D)$

for $i := 1$ **to** n **do**
  $D_{0i} = (0, \leq)$
  for $j := 1$ **to** n **do**
    if $D_{ji} < D_{0i}$ **then**
      $D_{0i} = Dji$
    end if
  end for
end for

Algorithm 8 $\texttt{and}(D, g)$

if $D_{yx} + (m, \preceq) < 0$ **then**
  $D_{00} = (-1, \leq)$
else if $(m, \preceq) < D_{xy}$ **then**
  $D_{xy} = (m, \preceq)$
  for $i := 0$ **to** n **do**
    for $j := 0$ to n **do**
      if $D_{ix} + D_{xj} < D_{ij}$ **then**
        $D_{ij} = D_{ix} + D_{xj}$
      end if
      if $D_{iy} + D_{yj} < D_{ij}$ **then**
        $D_{ij} = D_{iy} + D_{yj}$
      end if
    end for
  end for
end if

Algorithm 9 $\texttt{free}(D, x)$

for $i := 0$ **to** n **do**
  if $i \neq x$ **then**
    $D_{xi} = \infty$
    $D_{ix} = D_{i0}$
  end if
end for

Algorithm 10 $\texttt{reset}(D, x := m$

for $i := 0$ **to** n **do**
  $D_{xi} := (m, \leq) + D_{0i}$
  $D_{ix} := D_{i0} + (-m, \leq)$
end for

Algorithm 11 $\texttt{copy}(D, x := y)$

for $i := 0$ **to** n **do**
  if $i \neq x$ **then**
    $D_{xi} := D_{yi}$
    $D_{ix} := D_{iy}$
  end if
end for
$D_{xy} := (0, \leq)$
$D_{yx} := (0, \leq)$

Algorithm 12 $\texttt{shift}(D, x := x + m)$

for $i := 0$ **to** n **do**
  if $i \neq x$ **then**
    $D_{xi} := D_{xi} + (m, \leq)$
    $D_{ix} := D_{ix} + (-m, \leq)$
  end if
end for
$D_{x0} := max(D_{x0}, (0, \leq))$
$D_{0x} := min(D_{0x}, (0, \leq))$

Algorithm 13 $\mathsf{norm}_k(D))$

for $i := 0$ to n **do**
  for $j := 0$ to n **do**
    if $D_{ij} \neq \infty$ and $D_{ij} > (k_i), \leq)$ **then**
      $D_{ij} = \infty$
    else if $D_{ij} \neq \infty$ and $D_{ij} < (-k_j, <)$ **then**
      $D_{ij} = (-k_j, <)$
    end if
  end for
end for
$\texttt{close}(D)$

Algorithm 14 Core normalization: $\mathsf{Core\text{-}Norm}_k(D)$

$\mathcal{G}_{\mathrm{unsat}} := \emptyset$
for all $g \in \mathcal{G}_{\mathrm{d}}$ **do**
  if $D \wedge g = \emptyset$ **then**
    $\mathcal{G}_{\mathrm{unsat}} := \mathcal{G}_{\mathrm{unsat}} \cup \{g\}$
  end if
  if $D \wedge \neg g = \emptyset$ **then**
    $\mathcal{G}_{\mathrm{unsat}} := \mathcal{G}_{\mathrm{unsat}} \cup \{\neg g\}$
  end if
end for
$D := \mathsf{norm}_k(D)$
for all $g \in \mathcal{G}_{\mathrm{unsat}}$ **do**
  $D := D \wedge \neg g$
end for
return D

Algorithm 15 Zone splitting: $\mathsf{split}(D)$

$Q := \{D\}, Q' := \emptyset$
for all $g \in \mathcal{G}_{\mathrm{d}}$ **do**
  for all $D' \in Q$ **do**
    if $D' \wedge g$ and $D' \wedge \neg g$ **then**
      $Q' := Q' \cup \{D' \wedge g, D' \wedge \neg g\}$
    else
      $Q' := Q' \cup \{D'\}$
    end if
  end for
  $Q := Q', Q' := \emptyset$
end for
return Q

Algorithm 16 Normalization: $\mathsf{norm}_{k,\mathcal{G}}(D)$

$Q := \emptyset$
for all $D' \in \mathsf{split}(D)$ **do**
  $Q := Q \cup \{\mathsf{Core\text{-}Norm}_k(D')\}$
end for
return Q

Algorithm 17 Algorithm for adding encoded bounds

if $b_1 = \texttt{MAX\_INT}$ or $b_2 = \texttt{MAX\_INT}$ **then**
  return `MAX_INT`
else
  return $b_1 + b_2 - ((b_1\&1)|(b2\&1))$
end if

References

[ABL98] Luca Aceto, Augusto Bergueno, and Kim G. Larsen. Model checking via reachability testing for timed automata. In *Proceedings, Fourth Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[ACD90] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 414–425. IEEE Computer Society Press, 1990.

[ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Journal of Information and Computation*, 104(1):2–34, 1993.

[ACH94] Rajeev Alur, Costas Courcoubetis, and Thomas A. Henzinger. The observational power of clocks. In *International Conference on Concurrency Theory*, pages 162–177, 1994.

[AD90] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Proceedings, Seventeenth International Colloquium on Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer-Verlag, 1990.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Journal of Theoretical Computer Science*, 126(2):183–235, 1994.

[AFH99] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata: a determinizable class of timed automata. *Theoretical Computer Science*, 211(1–2):253–273, 1999.

[AH94] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994.

[Bal96] Felice Balarin. Approximate reachability analysis of timed automata. In *Proceedings, 17*th *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1996.

[BD91] Bernard Berthomieu and Michel Diaz. Modeling and verification of timed dependent systems using timed petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.

[BDGP98] Beatrice Bérard, Volker Diekert, Paul Gastin, and Antoine Petit. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae*, 36:145–182, 1998.

[BDLY03] Gerd Behrmann, Alexandre David, Kim G. Larsen, and Wang Yi. Unification & sharing in timed automata verification. In *Proceedings, Tenth International SPIN Workshop*, volume 2648 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Ben01] Johan Bengtsson. Reducing memory usage in symbolic state-space exploration for timed systems. Technical Report 2001-009, Department of Information Technology, Uppsala University, 2001.

[Ben02] Johan Bengtsson. Clocks, dbms and states in timed systems. Ph.D. Thesis, ACTA Universitatis Upsaliensis 39, Uppsala University, 2002.

[BJLY98] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial order reductions for timed systems. In *Proceedings, Ninth International Conference on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 485–500. Springer-Verlag, 1998.

[BLP+99] Gerd Behrmann, Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. In *Proceedings, Eleventh International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 341–353. Springer-Verlag, 1999.

[BY03] Johan Bentsson and Wang Yi. On clock difference constraints and termination in reachability analysis of timed automata. In *Formal Methods and Software Engineering, 5th International Conference on Formal Engineering Methods, ICFEM 2003, Singapore, November 5-7, 2003, Proceedings*, volume 2885 of *Lecture Notes in Computer Science*. Springer, 2003.

[Cer92] Karlis Cerans. Decidability of bisimulation equivalences for parallel timer processes. In *Computer Aided Verification*, volume 663 of *LNCS*. Springer, 1992.

[Cha99] Zhou Chaochen. Duration calculus, a logical approach to real-time systems. *Lecture Notes in Computer Science*, 1548:1–7, 1999.

[DBLY03] Alexandre David, Gerd Behrmann, Kim G. Larsen, and Wang Yi. A tool architecture for the next genreation of UPPAAL. Technical Report 2003-011, Department of Information Technology, Uppsala University, 2003.

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings, Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1989.

[DY96] Conrado Daws and Sergio Yovine. Reducing the number of clock variables of timed automata. In *Proceedings, 17*th *IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, 1996.

[Flo62] Robert W. Floyd. Acm algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, June 1962.

[FPY02] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In J.-P. Katoen and P. Stevens, editors, *Proc. of the 8*th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2280 in Lecture Notes in Computer Science, pages 67–82. Springer–Verlag, 2002.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. In *Fifth Conference on Computer-Aided Verification*, Elounda (Greece), July 1993. LNCS 697, Springer Verlag.

[HNSY92] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 394–406, 1992.

[HNSY94] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Journal of Information and Computation*, 111(2):193–244, 1994.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–676, August 1978.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[Lar00] Fredrik Larsson. Efficient implementation of model-checkers for networks of timed automata. Licentiate Thesis 2000-003, Department of Information Technology, Uppsala University, 2000.

[LLPY97] Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: Compact data structure and state space reduction. In *Proceedings, 18*th *IEEE Real-Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, 1997.

[LPWY99] Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Clock difference diagrams. *Nordic Journal of Computing*, 1999.

[LPY95] Kim G. Larsen, Paul Pettersson, and Wang Yi. Compositional and Symbolic Model-Checking of Real-Time Systems. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 76–87. IEEE Computer Society Press, December 1995.

[LPY97] Kim G. Larsen, Paul Petterson, and Wang Yi. UPPAAL in a nutshell. *Journal on Software Tools for Technology Transfer*, 1997.

[LPY98] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gear-Box Controller. In *Proceedings, Fourth Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, number 1384 in Lecture Notes in Computer Science, pages 281–297. Springer-Verlag, 1998.

[LPY01] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal Design and Analysis of a Gearbox Controller. *Springer International Journal of Software Tools for Technology Transfer (STTT)*, 3(3):353–368, 2001.

[LW97] Kim Guldstrand Larsen and Yi Wang. Time-abstracted bisimulation: Implicit specifications and decidability. *Information and Computation*, 134(2):75–101, 1997.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[NS94] Xavier Nicollin and Joseph Sifakis. The algebra of timed processes, ATP: Theory and application. *Journal of Information and Computation*, 114(1):131–178, 1994.

[Pet99] Paul Pettersson. *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice*. PhD thesis, Uppsala University, 1999.

[Rok93] Tomas Gerhard Rokicki. *Representing and Modeling Digital Circuits*. PhD thesis, Stanford University, 1993.

[RR88] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58(1-3):249–261, 1988.

[SD95] Ulrich Stern and David L. Dill. Improved probabilistic verification by hash compaction. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[WL93] Pierre Wolper and Dennis Leroy. Reliable hashing without collision detection. In *Proceedings, Fifth International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer-Verlag, 1993.

[Yi91] Wang Yi. CCS + time = an interleaving model for real time systems. In *Proceedings, Eighteenth International Colloquium on Automata, Languages and Programming*, volume 510 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[YJ94] Wang Yi and B. Jonsson. Decidability of timed language-inclusion for networks of real-time communicating sequen tial processes. In *Proc.* 14^{th} *Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 880 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[YL93] Mihalis Yannakakis and David Lee. An efficient algorithm for minimizing real-time transition systems. In *Proceedings, Fifth International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 210–224. Springer-Verlag, 1993.

[Yov97] Sergio Yovine. Kronos: a verification tool for real-time systems. *Journal on Software Tools for Technology Transfer*, 1, October 1997.

[YPD94] Wang Yi, Paul Petterson, and Mats Daniels. Automatic verification of real-time communicating systems by constraint-solving. In *Proceedings, Seventh International Conference on Formal Description Techniques*, pages 223–238, 1994.