

Fully Symbolic Model Checking of Timed Systems using Difference Decision Diagrams¹

Jesper Møller, Jakob Lichtenberg, Henrik R. Andersen, and
Henrik Hulgaard

*The IT University in Copenhagen
Glentevej 67
DK-2400 Copenhagen NV, Denmark
E-mail: {jm, jl, hra, henrik}@itu.dk*

Abstract

Current approaches for analyzing timed systems are based on an explicit enumeration of the discrete states and thus these techniques are only capable of analyzing systems with a handful of timers and a few thousand states. We address this limitation by describing how to analyze a timed system fully symbolically, i.e., by representing sets of discrete states and their associated timing information *implicitly*. We demonstrate the efficiency of the symbolic technique by computing the set of reachable states for a non-trivial timed system and compare the results with the state-of-the-art tools KRONOS and UPPAAL. With an implementation based on difference decision diagrams, the runtimes are several orders of magnitudes better. The key operation in obtaining these results is the ability to advance time symbolically. We show how to do this efficiently by essentially quantifying out a special variable z which is used to represent the constant zero. The symbolic manipulations given in this paper are sufficient to verify TCTL-formulae fully symbolically.

1 Introduction

Model checking [15] is today used extensively for formal verification of finite state systems such as digital circuits and embedded software. The success of the technique is primarily due to the use of a *symbolic* representation of sets of states and relations between states as predicates over Boolean variables (using for instance binary decision diagrams (BDDs) [11]). By representing,

¹ This work was carried out while the authors were at the Department of Information Technology, Technical University of Denmark, and was financially supported by a grant from the Danish Technical Research Council.

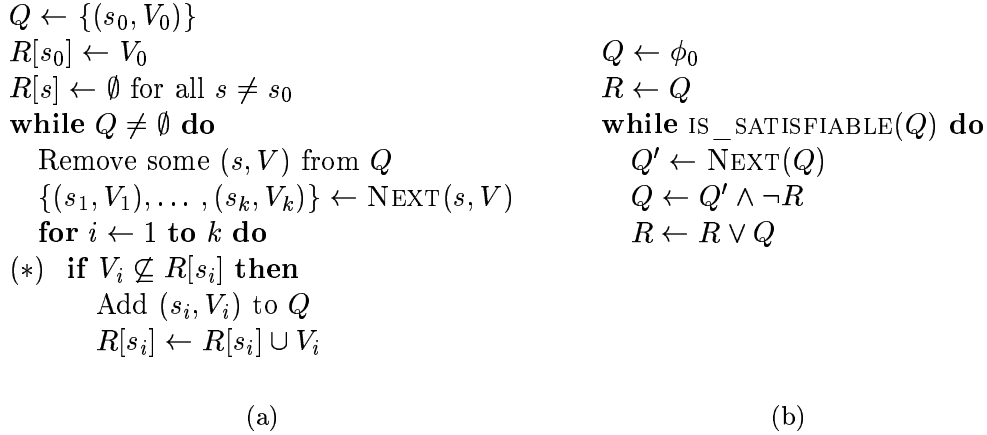


Fig. 1. Two different approaches for constructing the set of reachable states R . (a) Outline of the algorithm used in current tools such as KRONOS and UPPAAL, and (b) a fully symbolic algorithm.

for example, the set of reachable states as a predicate instead of explicitly enumerating the elements of the set, it is possible to verify systems with a very large number of states [13]. However, these symbolic methods do not easily generalize to models that contain continuous variables ranging over non-countable domains like for example real-time systems where time is modeled using continuous variables and the behavior of a system is specified using constraints on these variables. One problem is how to succinctly represent the usually infinite number of states of such systems; another problem is how to perform the basic verification operations (resetting clocks, advancing the time of clocks, etc.) symbolically on this representation in order to compute the reachable state space or to verify a temporal property of the system.

1.1 Current Approaches

A state in a timed system is a pair (s, v) where s is a discrete state (e.g., markings of Petri nets or locations in timed automata) and v is the associated timing information (i.e., a value assignment to the clocks in the system). To analyze timed systems (which have an infinite number of states due to the dense nature of the clocks), clock assignments are grouped into sets. This allows the state space of a timed system to be represented as a finite set of pairs (s, V) of discrete states and their associated group of clock valuations. The reachable states space R for a timed system can be determined by the generic algorithm in Fig. 1(a) (here we view R as a mapping from discrete states s to their associated group of clock valuations V). The function NEXT fires all possible transitions and advances time from the set of states (s, V) .

Current state-of-the-art techniques for verifying timed systems (e.g., [9, 24, 29, 33]) are based on representing each set of clock assignments using a set of difference bound matrices (DBMs) [20]. Each difference bound matrix can

represent a convex set of clock assignments, thus to represent V , in general, a number of matrices is needed (i.e., representing V as a union of convex sets). The function NEXT constructs the set of new states such that each V_i is a single DBM. The test in the line marked (*) is performed by checking whether the DBM V_i is not contained in any of the DBMs used to represent $R[s_i]$.

Although DBMs provide a compact representation of a convex set of clock configurations, there are several serious problems with the approaches based on DBMs: 1) the number of DBMs for representing the timing information V can become very large; 2) there is no sharing or reuse of DBMs among the different discrete states; and 3) each discrete state is represented explicitly, thus these approaches are limited by the number of reachable states of the system (the well-known state explosion problem).

1.2 A Symbolic Approach

The first two problems can be addressed by representing the set V as a propositional formula over inequalities of the form $x - y \leq d$ (x and y are clock variables and d is a constant). If we have a compact representation of such formulae and can decide valid implications for performing the check in the line marked with (*), we can use the algorithm in Fig. 1(a) immediately. Difference decision diagrams [27] are a candidate for such a data structure which furthermore allows reuse of sub-formula among the discrete states. Initial experiments with this approach implemented in UPPAAL [5] show a significant improvement in memory consumption, even though the discrete states still are enumerated explicitly.

In this paper, we address all three problems by constructing the set of reachable states R in a fully symbolic manner, i.e., without enumerating the discrete states and without representing the timing information as a set of DBMs. In our approach, both the discrete part of a state and the associated timing information are represented by a formula. That is, *sets* of states (s, V) are represented by a single formula ϕ , similar to how sets of discrete states are represented by a formula when performing traditional symbolic model checking of untimed systems. Using such a representation, the set of reachable states R can be computed using the standard fixed-point iteration shown in Fig. 1(b).

A core operation when performing symbolic model checking is to determine a formula representing the set of states reachable by firing any transition or advancing time from a state satisfying ϕ , i.e., the function NEXT(ϕ) in Fig. 1(b). Firing the transitions is straightforward, but advancing time is more involved. We introduce a variable z denoting “zero” or “current time” and express all constraints of the form $x \leq d$ as $x - z \leq d$. The use of a designated variable representing zero for eliminating absolute constraints is used both in DBMs [20] and also when solving systems of difference constraints [16]. A key contribution of this paper is that we show how the z -variable, in addition

to making the representation more uniform, also makes it possible to advance time in a set of states represented by a formula ϕ , essentially by performing an existential quantification of z : Let P_{next} denote a predicate stating whether it is legal to advance time by changing the reference point from z to z' . Thus P_{next} will require that $z' \leq z$ since advancing time by some amount δ corresponds to decreasing the reference point z by δ . Typically, P_{next} will also include constraints expressing state invariants and urgency predicates. Now, a formula representing the set of states reachable from ϕ by advancing time by δ is determined from:

$$(\exists z.(\phi \wedge P_{\text{next}} \wedge z - z' = \delta))[z/z'].$$

More generally, the set of states reachable from ϕ by advancing time *by an arbitrary amount* is determined from:

$$(\exists z.(\phi \wedge P_{\text{next}}))[z/z'].$$

Another key contribution of this paper is that we show that performing fully symbolic model checking of timed systems amounts to representing and deciding validity of formulae in a simple first-order propositional logic over inequalities of the form $x - y \leq d$:

$$\psi ::= x - y \leq d \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \exists x.\psi, \quad (1)$$

where x and y are real-valued variables and $d \in \mathbb{R}$ is a constant. A practical model checking algorithm therefore requires a compact representation of formulae of the form (1) and an efficient decision procedure to determine validity of such formulae.

In Section 2 we introduce a simple model of timed systems called timed guarded commands and sketch how it can represent timed automata. Section 3 shows how to symbolically compute the set of reachable states of such timed systems and sketch how to perform a fully symbolic model checking of TCTL formulae. Section 4 introduces a data structure called difference decision diagrams for representing and deciding validity of formulae of the form (1). In Section 5, we demonstrate the efficiency of the symbolic approach by computing the set of reachable states for a non-trivial timed system and compare the results with the state-of-the-art-tools KRONOS and UPPAAL. Finally, Section 6 summarizes the contributions.

1.3 Related Work

Model checking of timed systems (timed automata in particular; see [34] for a survey) has been extensively studied and a number of tools exist for verifying such systems. One approach is based on making the dense domains discrete by assuming that timers only can take integer or rational values. Such a discretization makes it possible to use BDDs for representing both the discrete

states and the associated timing information [3, 10, 12, 14]. However, this way of representing dense domains is often inefficient; the BDD representation is very sensitive to the granularity of the discretization and to the size of the delay ranges.

The unit-cube approach [2] models time as dense but represents the timing information using a finite number of equivalence classes. Again, the number of timed states is dependent on the size of the delay ranges and easily becomes unmanageable. As mentioned above, more recent timing analysis methods use difference bound matrices (DBMs) [20] for representing the timing information [9, 24, 29, 33]. One can see the use of DBMs as expanding formulae of the form (1) into disjunctive normal form and representing each conjunction of difference constraints using a difference bound matrix. Several attempts have been made to remedy the shortcoming of DBMs discussed above, for example by using partial order methods [7, 30, 31] or by using approximate methods [4, 6, 32]. Although these approaches do address the problem that the number of DBMs for representing the timing information can become very large, they still enumerate all discrete states.

Henzinger et al. [22] describe how to perform symbolic model checking of timed systems. Although apparently similar to our approach, there are a number of significant differences: First, we show that the simple first-order logic (1) with only one type of clock constraints ($x - y \leq d$) is sufficient for representing the set of states of a timed system. This allows us to represent sets of states efficiently using an implicit representation of formulae (e.g., difference decision diagrams). Secondly, we show how to perform all operations needed in symbolic model checking within this logic. A core operation is advancing time which we show can be performed within the logic by introducing a designated variable z and using existential quantification.

Based on the initial ideas of difference decision diagrams (DDD), Behrmann et al. [5] have implemented a minor variation of DDDs allowing a fanout of more than two (which they call CDDs). They have shown a significant improvement in memory consumptions in UPPAAL, even though the experiments in contrast to ours do not use a fully symbolic approach (the discrete states are enumerated explicitly). Thus, this approach will not be able to handle the larger instances of the timed system described in Section 5.1.

2 Modeling Timed Systems

Timed guarded commands [22] are a simple notation for modeling systems with time. The notation is sufficient for encoding popular notations for systems with time such as timed automata [2] and timed Petri nets [9].

2.1 Timed Guarded Commands

A *timed guarded command program* G is a tuple (B, C, T, I) , where B is a set of Boolean variables, C is a set of continuous variables called *clocks*, T is a set of *timed guarded commands*, and I is a *state invariant*. A timed guarded command $t \in T$ has the form $g \rightarrow \mathbf{v} := \mathbf{d}$, where g is a guard and $\mathbf{v} := \mathbf{d}$ is a multi-assignment of n constant values $\mathbf{d} \in (\mathbb{B} \cup \mathbb{R})^n$ to Boolean variables and clocks $\mathbf{v} \in (B \cup C)^n$. Guards and state invariants are expressions ϕ constructed from the following grammar:

$$\phi ::= \mathbf{F} \mid \mathbf{T} \mid x \sim d \mid x - y \sim d \mid b \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \exists b.\phi \mid \exists x.\phi, \quad (2)$$

where $x, y \in C$ are clocks, $b \in B$ is a Boolean variable, $d \in \mathbb{R}$ is a constant, and \sim is a relational operator from $\{\leq, <, =, \neq, >, \geq\}$. The symbols \mathbf{F} and \mathbf{T} denote false and true, respectively, and the symbols \neg (negation), \wedge (conjunction) and \exists (existential quantification) have their usual meaning.

Example 2.1 An example of a program is $G = (\{b\}, \{x, y\}, T, I)$, where T contains the two guarded commands

$$\begin{aligned} b \wedge (1 \leq x \leq 3) \rightarrow b &:= \mathbf{F} \\ b \wedge (7 \leq x \leq 9) \rightarrow b, y &:= \mathbf{F}, 0 \end{aligned}$$

and the state invariant is $I = (b \Rightarrow (x \leq 9)) \wedge (\neg b \Rightarrow (x \neq 5))$.

2.2 Transitional Semantics of Timed Guarded Commands

A *state* of the program $G = (B, C, T, I)$ is an interpretation (i.e., a value assignment) of the Boolean variables and the clocks. For each Boolean variable $b \in B$, $s(b) \in \mathbb{B}$ denotes the interpretation of b in the state s , and for each clock $x \in C$, $s(x) \in \mathbb{R}$ denotes the interpretation of x in the state s . We use the notation $s[x := y + d]$ to denote the state s' equivalent to s except that $s'(x) = s(y) + d$. A state (and sets of states) can be represented by an expression ϕ of the form (2). The state s *satisfies* an expression ϕ , written $s \models \phi$, if ϕ evaluates to true in the state s , and we write $\llbracket \phi \rrbracket$ for the set of states that satisfy ϕ .

The semantics of a timed guarded command program $G = (B, C, T, I)$ is a *transition system* $(\mathcal{S}, \rightarrow)$, where \mathcal{S} is the set of states of the program, and \rightarrow is the transition relation. In each state, the program can either execute a command $t \in T$ if its guard is true (a discrete transition) or let time pass δ time units (a timed transition). Executing a command changes the value of some or all of the variables (according to the multi-assignment), and letting time pass uniformly increases the values of all clocks by δ . We use the notation $s \xrightarrow{t} s'$ for a discrete transition from the state s to s' obtained by executing the command t , and the notation $s \xrightarrow{\delta} s'$ for a timed transition obtained by increasing all clocks by δ . The discrete transition \xrightarrow{t} for a timed command

$t \in T$ of form $g \rightarrow \mathbf{v} := \mathbf{d}$ is defined by the following rule:

$$\frac{s \models g \quad s[\mathbf{v} := \mathbf{d}] \models I}{s \xrightarrow{t} s[\mathbf{v} := \mathbf{d}]} . \quad (3)$$

The timed transition $\xrightarrow{\delta}$ for advancing all clocks by δ is defined by the following rule:

$$\frac{\delta \geq 0 \quad \forall \delta'. 0 \leq \delta' \leq \delta : s[\mathbf{c} := \mathbf{c} + \delta'] \models I}{s \xrightarrow{\delta} s[\mathbf{c} := \mathbf{c} + \delta]} , \quad (4)$$

where $\delta, \delta' \in \mathbb{R}$, \mathbf{c} denotes a vector of all clocks in C , and $\mathbf{c} + \delta$ denotes the vector where δ is added to the clocks in \mathbf{c} .

Example 2.2 Consider the timed guarded command program G from Example 2.1 and let s be a state satisfying $\neg b \wedge (x < 5)$. There are infinitely many timed transitions from s in the transition system for G , but none of these timed transitions leads to a state where $x \geq 5$ because the state invariant $\neg b \Rightarrow (x \neq 5)$ must hold continuously.²

Given a transition system $(\mathcal{S}, \rightarrow)$ for a timed guarded command program $G = (B, C, T, I)$ and a set of states $S \subseteq \mathcal{S}$, $Next_{\text{discrete}}(S)$ denotes the set of states reachable from S by executing any timed guarded command in T :

$$Next_{\text{discrete}}(S) = \bigcup_{t \in T} \{s' : \exists s \in S. s \xrightarrow{t} s'\} .$$

Similarly, the set of states reachable from S by advancing time by an arbitrary amount is given by:

$$Next_{\text{timed}}(S) = \bigcup_{\delta \in \mathbb{R}} \{s' : \exists s \in S. s \xrightarrow{\delta} s'\} .$$

The set of states reachable from S , denoted $Reachable(S)$, is defined as the least fixed point of the function $F(X) = S \cup Next(X)$, where

$$Next(X) = Next_{\text{discrete}}(X) \cup Next_{\text{timed}}(X) .$$

2.3 Encoding Timed Automata

Timed guarded command programs can be used to model popular notations for timed systems such as timed automata [2]. A timed automaton over a set of clocks consists of a set of locations, a set of events, and a set of timed transitions. Each location is associated with a location invariant over the clocks, and each timed transition from location l to location l' is labeled with an event a and has a guard g over the clocks. Furthermore, each of the timed

² This is an example of a program with a *time-blocked* state [1].

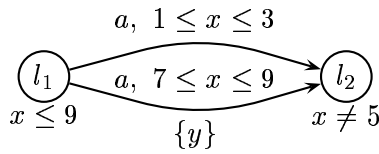


Fig. 2. The timed automaton in Example 2.3.

transitions has a set of clocks $\{c\}$ to be reset when the timed transition is fired:

$$l \xrightarrow[\{c\}]{a, g} l'.$$

A timed automaton can be encoded as a timed guarded command program. Each location is encoded as a Boolean variable.³ In a shared variable model as ours, the presence of an event from an alphabet Σ can be modeled by a global *event variable* e taking on any of the values in Σ . This variable can for instance be encoded using a logarithmic number of Boolean variables. Each timed transition in the automaton corresponds to a timed guarded command:

$$l \wedge e_a \wedge g \rightarrow l, l', c := F, T, \mathbf{0}.$$

The guard of the command is the guard of the timed transition g conjoined with the source location l of the timed transition and a condition e_a requiring the event variable e to have the value $a \in \Sigma$. The multi-assignment assigns F to the source location l and T to the destination location l' of the timed transition and resets the relevant clocks.

Example 2.3 Fig. 2 shows an automaton over the clocks $\{x, y\}$ with two locations and two timed transitions. Encoding this automaton as a timed guarded command program yields the program G from Example 2.1 when ignoring the event a and encoding the two locations l_1 and l_2 logarithmically using a Boolean variable b .

3 Analyzing Timed Guarded Commands

To verify properties of a timed guarded command program $G = (B, C, T, I)$, we symbolically analyze the corresponding transition system $(\mathcal{S}, \rightarrow)$. That is, given a set of states represented by a formula ψ , we determine a formula that represents the set of states reachable by executing timed guarded commands according to the inference rule (3) or by advancing time according to the inference rule (4). As we will show, this formula is obtained by manipulations entirely within the logic (1).

³ This is sometimes referred to as a “one-hot” encoding of the locations. In practice, a logarithmic encoding may be more efficient.

3.1 Difference Constraint Expressions

Any expression ϕ generated by the grammar (2) can be represented by a *difference constraint expression* ϕ_z of the form (1). The expression ϕ_z is obtained by introducing a new variable z (denoting “zero”) and performing the following three steps: First, encode each Boolean variable $b_i \in B$ in ϕ as a difference constraint $x_i - x'_i < 0$, where $x_i, x'_i \in C$ are clocks only used in the encoding of b_i .⁴ Second, replace each occurrence of a constraint of the form $x \sim d$ in ϕ with the difference constraint $x - z \sim d$. Third, express each difference constraint of the form $x - y \sim d$ in terms of the relational operator \leq .

We define two useful operators on difference constraint expressions: replacement and assignment. *Replacement* syntactically substitutes all occurrences of a variable x by a variable y plus a constant d in an expression ψ , denoted by $\psi[y + d/x]$. If x and y are different variables, the replacement $\psi[y + d/x]$ can be expressed in the grammar (1) as $\exists x.(\psi \wedge (x - y = d))$. Otherwise, $\psi[x + d/x]$ is defined as $\psi[t/x][x + d/t]$, where t is a variable different from x and not occurring in ψ . *Assignment* gives a variable x the value of a variable y plus a constant d , denoted by $\psi[x := y + d]$. If x and y are different variables, the assignment $\psi[x := y + d]$ is expressed in the grammar (1) as $(\exists x.\psi) \wedge (x - y = d)$. Otherwise, the assignment $\psi[x := x + d]$ is defined as $\psi[x - d/x]$ (which might seem counter-intuitive). Assignment and replacement of Boolean variables are defined in the standard way.

To formally express the symbolic manipulations, we introduce some useful shorthands. We use $\llbracket \psi \rrbracket_z$ as a shorthand for $\llbracket \exists z.(\psi \wedge z = 0) \rrbracket$, that is, $\llbracket \psi \rrbracket_z$ is the set of states that satisfy ψ when z is equal to 0. It is easy to prove that $\llbracket \phi \rrbracket = \llbracket \phi_z \rrbracket_z$, for any ϕ . Eliminating the constraints of the form $x \sim d$ from the grammar in (2) makes it possible to add δ to all clocks simultaneously by decreasing the common reference-point z by δ :

$$\llbracket \phi[\mathbf{c} := \mathbf{c} + \delta] \rrbracket = \llbracket \phi_z[z := z - \delta] \rrbracket_z. \quad (5)$$

Furthermore, as will be shown in the following, the set of states reachable by advancing time by any value δ can be computed by an existential quantification of z .

3.2 Reachability Analysis

Given an expression ψ of the form (1) representing a set of states $\llbracket \psi \rrbracket_z \subseteq \mathcal{S}$, we now show how to determine an expression representing the set of states reachable from $\llbracket \psi \rrbracket_z$. The set of states reachable by firing the timed guarded command t from any state in $\llbracket \psi \rrbracket_z$ is determined by the function $\text{NEXT}_{\text{discrete}}(\psi, t)$. The function restricts ψ to the subset where the guard g holds, performs the

⁴ It turns out that when using difference decision diagrams (see Section 4) with this apparently strange encoding of Boolean variables, the Boolean manipulations can be done as efficiently as when using BDDs.

assignment of the constants \mathbf{d} to the variables \mathbf{v} , and restricts the resulting set to the subset where the state invariant I holds:

$$\text{NEXT}_{\text{discrete}}(\psi, g \rightarrow \mathbf{v} := \mathbf{d}) = (\psi \wedge g_z)[\mathbf{v} := \mathbf{d}] \wedge I_z, \quad (6)$$

where the assignment $[\mathbf{v} := \mathbf{d}]$ is a shorthand for $c_i := z + d_i$ for each of the clocks c_i in \mathbf{v} and $b_i := d_i$ for each of Boolean variables b_i in \mathbf{v} . The set of states that can be reached from the set $\llbracket \psi \rrbracket_z$ by firing any timed guarded command in T is given by:

$$\text{NEXT}_{\text{discrete}}(\psi) = \bigvee_{t \in T} \text{NEXT}_{\text{discrete}}(\psi, t). \quad (7)$$

The z -variable plays a central rôle when determining the set of states that can be reached from $\llbracket \psi \rrbracket_z$ by firing a timed transition. We advance time by changing the reference-point from z to z' with $z' \leq z$ since decreasing the reference-point by δ corresponds to increasing the values of all clocks by δ . Often the system will restrict the valid choices for z' by requiring that the state invariant holds in z' and at all intermediate points in time. This is expressed by the predicate

$$P_{\text{next}} = (z' \leq z) \wedge I_{z'} \wedge \forall z''. ((z' < z'' \leq z) \Rightarrow I_{z''}).$$

If the state invariant I_z only expresses upper bounds on the clocks, the universal quantification is implied by $I_{z'}$ and can be omitted.

Now, to advance time by δ in all states $\llbracket \psi \rrbracket_z$, we simply decrease the reference-point z by δ : $\psi[z := z - \delta]$ which can also be written as $(\exists z'. (\psi \wedge z' = z - \delta))[z/z']$. The set of states reachable from $\llbracket \psi \rrbracket_z$ that also satisfy P_{next} is given by $\exists z'. (\psi \wedge (z - z' = \delta) \wedge P_{\text{next}})[z/z']$. Thus, the set of states reachable from $\llbracket \psi \rrbracket_z$ by advancing time by an *arbitrary* amount is given by

$$\begin{aligned} \text{NEXT}_{\text{timed}}(\psi) &= \bigvee_{\delta \in \mathbb{R}} \exists z'. (\psi \wedge (z - z' = \delta) \wedge P_{\text{next}})[z/z'] \\ &= \exists z'. (\psi \wedge P_{\text{next}})[z/z']. \end{aligned} \quad (8)$$

That is, we advance time in a set of states by performing a single existential quantification. The correctness of the next-state functions is proved in [28].

Example 3.1 If the state invariant is $x \neq 5$, the predicate P_{next} is given by:

$$\begin{aligned} P_{\text{next}} &= (z' \leq z) \wedge (x - z' \neq 5) \wedge \forall z''. ((z' < z'' \leq z) \Rightarrow (x - z'' \neq 5)) \\ &= (z' \leq z) \wedge ((x - z' < 5) \vee (x - z' > 5)). \end{aligned}$$

Consider the set of states satisfying $\phi = (1 \leq x \leq 3) \vee (7 \leq x \leq 9)$. The set of states obtained by advancing time from ϕ is thus given by $\llbracket \text{NEXT}_{\text{timed}}(\phi) \rrbracket_z$, where:

$$\text{NEXT}_{\text{timed}}(\phi) = \exists z'. (\phi_z \wedge P_{\text{next}})[z/z'] = (1 \leq x - z < 5) \vee (7 \leq x - z).$$

A timed guarded command $t \in T$ is called *urgent* if it is required to fire instantaneously whenever the guard becomes true. Modifying P_{next} to handle urgent commands is straightforward: Given a set $T' \subseteq T$ of urgent timed guarded commands, we let U denote the predicate:

$$U = \bigvee_{g \rightarrow v := \mathbf{d} \in T'} g.$$

Consider a state $s \in \llbracket \psi \rrbracket_z$. We can only fire a timed transition $s \xrightarrow{\delta} s'$ if there are no urgent transitions enabled in s . Thus, an additional requirement is added to P_{next} ensuring that no urgent transitions are enabled when advancing time (except in the endpoint), i.e., the revised P_{next} becomes:

$$P_{\text{next}} = (z' \leq z) \wedge I_{z'} \wedge \forall z'' . ((z' < z'' \leq z) \Rightarrow (I_{z''} \wedge \neg U_{z''})).$$

If the urgency predicate does not refer to z , P_{next} is simplified to

$$P_{\text{next}} = (z' \leq z) \wedge I_{z'} \wedge \forall z'' . ((z' < z'' \leq z) \Rightarrow I_{z''}) \wedge \neg U_z.$$

The functions defined in (7) and (8) form the basis for constructing the set of reachable states symbolically. Let $\text{NEXT}(\psi)$ be a function which determines the set of states which can be reached by firing either a discrete or a timed transition from a state in $\llbracket \psi \rrbracket_z$:

$$\text{NEXT}(\psi) = \text{NEXT}_{\text{discrete}}(\psi) \vee \text{NEXT}_{\text{timed}}(\psi).$$

The set of states reachable from $\llbracket \psi \rrbracket_z$, denoted $\text{REACHABLE}(\psi)$, is the least fixed point of the function $F(X) = \psi \vee \text{NEXT}(X)$, which can be determined using a standard fixed-point iteration. Detecting that a fixed point has been reached is done by checking that two successive approximations ψ_i and ψ_{i+1} are semantically equivalent (i.e., that $\psi_i \Leftrightarrow \psi_{i+1}$ is a tautology). It is well known that there exists (contrived) timed systems where the computation of the fixed point does not terminate, for example if the difference between two clocks increase ad infinitum. As in the traditional analysis of timed automata, it is possible to determine subclasses of timed guarded commands for which termination is ensured.

Example 3.2 Consider again the program from Example 2.1. The set of states reachable from $\phi = b \wedge (x = y = 0)$ is $\llbracket \text{REACHABLE}(\phi_z) \rrbracket_z$, where:

$$\begin{aligned} \text{REACHABLE}(\phi_z) = & (b \wedge x = y \wedge x - z \leq 9) \\ & \vee (\neg b \wedge ((x = y \wedge 1 \leq x - z < 5) \vee (7 \leq x - y \leq 9 \wedge 7 \leq x - z))). \end{aligned}$$

3.3 Symbolic Model Checking

To perform symbolic model checking, for example of a TCTL formula, the set of states that can reach a given set $\llbracket \psi \rrbracket_z$ needs to be determined. The set of

states that can reach $\llbracket \psi \rrbracket_z$ by firing any timed guarded command $g \rightarrow \mathbf{v} := \mathbf{d}$ in T is given by

$$\text{PREV}_{\text{discrete}}(\psi) = \bigvee_{g \rightarrow \mathbf{v} := \mathbf{d} \in T} (\exists \mathbf{v}. (\psi \wedge \mathbf{v} = \mathbf{d})) \wedge g_z \wedge I_z,$$

where the expression $\mathbf{v} = \mathbf{d}$ is a shorthand for $c_i - z = d_i$ for each of the clocks c_i in \mathbf{v} and $b_i \Leftrightarrow d_i$ for each of Boolean variables b_i in \mathbf{v} . The set of states that can reach $\llbracket \psi \rrbracket_z$ by advancing time is determined analogously to the forward case:

$$\text{PREV}_{\text{timed}}(\psi) = \exists z'. (\psi \wedge P_{\text{prev}})[z/z'],$$

where

$$P_{\text{prev}} = (z \leq z') \wedge I_{z'} \wedge \forall z''. ((z < z'' \leq z') \Rightarrow I_{z''}).$$

The set of states that can reach a state in $\llbracket \psi \rrbracket_z$ by firing either a discrete or a timed transitions is:

$$\text{PREV}(\psi) = \text{PREV}_{\text{discrete}}(\psi) \vee \text{PREV}_{\text{timed}}(\psi).$$

Thus, we can construct the set of states that can reach a state satisfying ψ as the least fixed-point of the function $B(X) = \psi \vee \text{PREV}(X)$. Moreover, PREV can be used to perform symbolic model checking of TCTL [22]. TCTL is a timed version of CTL [15] obtained by extending the logic with an auxiliary set of clocks called *specification clocks*. These clocks do not appear in the model and are used to express timing bounds on the temporal operators. The atomic predicates of TCTL are difference constraints over the clocks from the model and the specification clocks. Semantically, the specification clocks become part of the state, they proceed synchronously with the other clocks but are not changed by the model. A specification clock u can be bound and reset by a *reset quantifier* $u.\psi$.

Symbolically, we can find the set of states satisfying a given TCTL formula ψ by a backward computation using a fixed-point iteration for the temporal operators. For instance, the set of states satisfying the formula $\psi_1 EU \psi_2$ is computed symbolically as the least fixed point of the function $B(X) = \psi_2 \vee (\psi_1 \wedge \text{PREV}(X))$. The set of states satisfying $u.\psi$ is computed symbolically as $\exists u. (\psi \wedge u - z = 0)$, i.e., the reset quantifier corresponds to restricting the value of u to zero and then remove it by existential quantification. The atomic predicates and the Boolean connectives correspond precisely to the corresponding difference constraint expressions.

Above we have determined the set of states using a constrained image approach. To compose systems *synchronously*, as used for instance in timed automata, a timed guarded command program can be encoded using a *transition relation* R over “present-state” variables $V = B \cup C \cup \{z\}$ and the “next-state”

variables $V' = \{v' : v \in V\}$ (as traditionally done in symbolic model checking of discrete systems but including the reference points z and z'). The relation R is constructed by combining the transitions of each automaton using disjunctions and then combining the automata using conjunctions. Thus, the parallel composition of a set of timed automata can be analyzed fully symbolically, i.e., both symbolically with respect to the parallel composition *and* with respect to the representation of sets of clock valuations and discrete states (see [28]). Using a transition relation, we get the benefit that well-known and very useful tricks from the work on BDDs, such as early variable quantification and partitioned representation of the transition relation are immediately applicable.

4 Difference Decision Diagrams

The previous sections show that to perform symbolic analysis of timed systems we need a data structure for representing difference constraint expressions and a decision procedure to determine validity of such expressions. *Difference decision diagrams* (DDDs) [27] are a candidate for such a data structure. Similar to how a BDD represents the meaning of a Boolean formula implicitly, a DDD represents the meaning $\llbracket \psi \rrbracket$ of a difference constraint expression ψ of the form (1) using a decision diagram in which the vertices contain difference constraints.

A DDD is a directed acyclic graph (V, E) with two terminals $\mathbf{0}$ and $\mathbf{1}$ and a set of non-terminal vertices. Each non-terminal vertex corresponds to the if-then-else operator $\alpha \rightarrow \psi_1, \psi_0$, defined as $(\alpha \wedge \psi_1) \vee (\neg \alpha \wedge \psi_0)$, where the test expression α is a difference constraint and the high-branch ψ_1 and low-branch ψ_0 are other DDD vertices. Each vertex v in a DDD denotes a difference constraint expression ψ^v given by:

$$\psi^v = \alpha(v) \rightarrow \psi^{high(v)}, \psi^{low(v)}, \quad (9)$$

where $\alpha(v)$ is the difference constraint of v , and $high(v)$ and $low(v)$ are the high- and low-branches, respectively.

As an example of a DDD consider the following expression ψ over $x, y, z \in \mathbb{R}$:

$$\psi = 1 \leq x - z \leq 3 \wedge (y - z \geq 2 \vee y - x \geq 0). \quad (10)$$

Fig. 3 shows $\llbracket \psi \rrbracket_z$ as an (x, y) -plot and the corresponding DDD.

As shown in [27], DDDs can be ordered and reduced making it possible to check for validity and satisfiability in constant time. Furthermore, the operations for constructing and manipulating DDDs according to the syntactic constructions of (1) are easily defined recursively on the DDD data structure, thus making it simple to specify and implement algorithms for these operations. The function $\text{APPLY}(op, u, v)$ is used to combine two ordered, locally

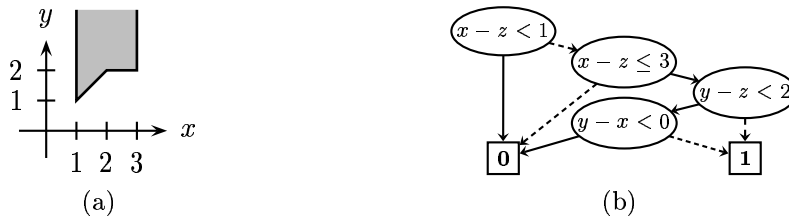


Fig. 3. The expression ψ in (10) as (a) an (x, y) -plot for $z = 0$, and (b) a difference decision diagram.

reduced DDDs rooted at u and v with a Boolean operator op , e.g., the negation and conjunction operations in (1). APPLY is a generalization of the version used for ROBDDs and has running time $O(|u||v|)$, where $|\cdot|$ denotes the number of vertices in a DDD. The function EXISTS(x, u) is used to quantify out the variable x in a DDD rooted at u . The algorithm is an adoption of the Fourier-Motzkin method [21], removing all vertices reachable from u containing x , but keeping all implicit constraints induced by x among the other variables (e.g., $\exists x.(z - x < 1 \wedge x - y \leq 0)$ is equivalent to $z - y < 1$). EXISTS computes the modified and additional constraints in polynomial time, but has an exponential worst-case running time since the resulting DDD must be ordered.

Recall that Boolean variables in (2) are encoded as $x_i - x'_i \leq 0$. This encoding allows us to represent and manipulate both real-valued and Boolean variables in a homogeneous manner. Furthermore, the encoding has the advantage that any Boolean expression will have a canonical DDD representation (because of the DDD reduction rules) and can be manipulated as efficiently as when represented by a BDD.

5 Experimental Results

We demonstrate the applicability and efficacy of the symbolic approach by analyzing two different versions of Milner’s scheduler with time. We compare the runtimes of the symbolic approach using DDDs with those obtained with two state-of-the-art tools, KRONOS [17–19] and UPPAAL [8, 25]. The two versions of Milner’s scheduler are simple, regular and highly concurrent systems, and they illustrate the advantages of our symbolic approach based on DDDs over state-of-the-art tools.

5.1 Milner’s Scheduler with One Clock

Milner’s scheduler [26] consists of N cyclers, connected in a ring, cooperating on controlling N tasks. We associate three Boolean variables c_i , h_i , and t_i with each cycler and use a global clock H to ensure that a cycler passes the token on to the following cycler within a bounded amount of time. The i^{th} cycler is described by two guarded commands and the task is modeled by a

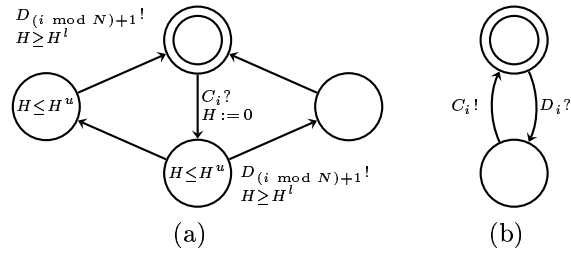


Fig. 4. The i^{th} cyclus of Milner’s scheduler with one clock. The cyclus is modeled with two timed automata synchronizing via CCS-like channels as in UPPAAL. Initial locations are drawn with double circles (for the first cyclus the initial location in (b) is the opposite location). The locations of the automaton in (a) represent the four possible combinations of the variables h_i and t_i , and the automaton in (b) represents the variable c_i .

third guarded command:

$$\begin{aligned}
 c_i \wedge \neg t_i &\rightarrow H, t_i, c_i, h_i && := 0, \top, \text{F}, \top \\
 h_i \wedge H^l \leq H &\rightarrow c_{(i \bmod N)+1}, h_i && := \top, \text{F} \\
 t_i &\rightarrow t_i && := \text{F}.
 \end{aligned}$$

The state invariant is given by

$$I = \bigwedge_{i=1}^N h_i \Rightarrow H \leq H^u,$$

expressing that cyclus i must pass on the token no later than time H^u . Thus, the amount of time a cyclus can keep the token is determined by the interval $[H^l, H^u]$. Furthermore, the first guarded command is urgent, thus the urgency predicate is

$$U = \bigvee_{i=1}^N c_i \wedge \neg t_i.$$

It is straightforward to model Milner’s scheduler with timed automata. Fig. 4 shows the i^{th} cyclus in the notation used by UPPAAL.

We have computed the reachable state space for increasing N using a fixed-point iteration with front sets. The results are shown in Table 1(a) together with the runtimes obtained with KRONOS (version 2.2b) and UPPAAL (version 2.17). This version of Milner’s scheduler has a number of discrete states which is exponential in N since a task can terminate independently of the other tasks. Thus, state space exploration based on enumerating all discrete states (as in UPPAAL and KRONOS) only succeeds for small systems. In the symbolic approach using DDDs, discrete states are represented implicitly (as when using BDDs for purely discrete systems) and choosing a good ordering of the variables gives polynomial runtimes (and state space representations).

Table 1

Experimental results for Milner's scheduler with (a) one clock using the bounds $[H^l, H^u] = [25, 200]$, and (b) one clock per task using the bounds $[H^l, H^u] = [25, 200]$ and $[T^l, T^u] = [80, 100]$. The first column shows the number of cyclers, and the following three columns show the CPU time (in seconds) to build the reachable state space using KRONOS, UPPAAL, and DDDs, respectively. The results were obtained on a Pentium II PC with 64 MB of memory running Linux. A '-' denotes that the analysis did not complete within an hour.

N	KRONOS	UPPAAL	DDD
4	0.2	0.1	0.1
5	0.7	0.2	0.1
6	22.6	0.6	0.1
7	339.2	2.3	0.1
8	—	9.0	0.2
9	—	35.0	0.2
10	—	138.4	0.2
11	—	529.8	0.2
12	—	2560.7	0.3
16	—	—	0.5
32	—	—	2.2
64	—	—	15.9
128	—	—	123.3
256	—	—	1104.8

(a)

N	KRONOS	UPPAAL	DDD
4	0.4	0.2	0.2
5	2.4	1.7	0.3
6	24.2	17.6	0.5
7	346.6	201.7	0.5
8	—	2460.2	0.6
16	—	—	1.5
32	—	—	5.7
64	—	—	31.7
128	—	—	217.3

(b)

5.2 Milner's Scheduler with a Clock per Task

We now restrict the time a task can be executing by introducing a clock T_i that measures the execution time of each task t_i . The task t_i must terminate within a certain bound $[T^l, T^u]$ after it is started. The resulting system potentially has $N + 1$ concurrently running clocks (one for each task plus one for the token), but the system will have fewer discrete states than the previous version since the bounded execution time of the tasks limits the number of reachable discrete states. The i^{th} cycler is now described by the guarded commands:

$$\begin{aligned}
 c_i \wedge \neg t_i &\rightarrow H, T_i, t_i, c_i, h_i := 0, 0, \top, \text{F}, \top \\
 h_i \wedge H^l \leq H &\rightarrow c_{(i \bmod N)+1}, h_i := \top, \text{F} \\
 t_i \wedge T^l \leq T_i &\rightarrow t_i \quad \quad \quad := \text{F}.
 \end{aligned}$$

Introducing the new clocks changes the state invariant to

$$I = \bigwedge_{i=1}^N (h_i \Rightarrow H \leq H^u) \wedge (t_i \Rightarrow T_i \leq T^u).$$

Fig. 5 shows the i^{th} cycler in the notation used by UPPAAL.

The runtimes for computing the reachable state space for increasing N are

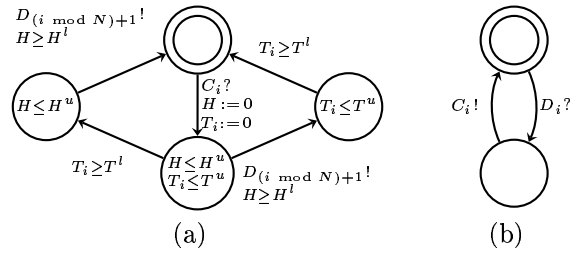


Fig. 5. The timed automata modeling the i^{th} cyclor of Milner’s scheduler with a clock for each task. Initial locations are drawn with double circles (for the first cyclor the initial location in (b) is the opposite location).

shown in Table 1(b). Again, the runtimes of KRONOS and UPPAAL are exponential in N , while the symbolic approach using DDDs results in polynomial runtimes. In this version of Milner’s scheduler, the problem for KRONOS and UPPAAL is the large number of clock variables. This is handled in the symbolic approach using DDDs by eliminating unused clocks from the representation (i.e., we quantify out T_i whenever the guarded command that sets t_i to false is fired in $\text{NEXT}_{\text{discrete}}$).

As for BDDs, the size of a DDD depends on the chosen variable ordering. In the two versions of Milner’s scheduler, experiments show that the Boolean variables should precede the clocks in the decision diagram. The Boolean variables are ordered as $t_1 \prec c_1 \prec h_1 \prec \dots \prec t_N \prec c_N \prec h_N$. Pairs of clocks (x_i, x_j) are ordered reversed lexicographically⁵ using the ordering $z \prec H \prec T_1 \prec \dots \prec T_N$. There are a number of techniques to avoid BDD-size blow-up that also apply to DDDs. For example, instead of building a DDD for I , we build a list of N implicitly conjoined DDDs as described in [23]. When we build the DDD for the set of discrete next states defined in (6) we conjoin each element $(h_i \Rightarrow H \leq H^u) \wedge (t_i \Rightarrow T_i \leq T^u)$ of the list to the expression $(\psi \wedge g_z)[\mathbf{v} := \mathbf{d}]$. We use the same technique when we build the DDD for the set of timed next states defined in (8), which is possible because I only expresses upper bounds on the clocks, and thus P_{next} is given by $(z' \leq z) \wedge I_{z'} \wedge \neg U_z$.

6 Conclusion

We have shown how difference constraint expressions can be used to fully symbolically represent and verify concurrent timed systems. A key idea is to avoid representing absolute constraints. Instead, these constraints are expressed relative to a special variable z , which allows us to advance all clocks synchronously by performing a single existential quantification.

Our results show that an efficient implementation of difference constraint expressions is highly desirable and we propose an implementation using difference decision diagrams. Difference decision diagrams (DDD) possess the same abilities as BDDs of providing a compact representation while admit-

⁵ That is, $(x_i, x_j) \prec (x'_i, x'_j)$ iff $x_j \prec x'_j$ or $x_j = x'_j \wedge x_i \prec x'_i$

ting an efficient validity check. Just as BDDs provide an implementation of quantified Boolean logic, which allows the symbolic verification of discrete systems [13], DDDs provide an efficient implementation of difference constraint expressions, which allows the symbolic verification of timed systems.

Continuing extending the power of the underlying Boolean logic, difference constraints could be replaced by the more powerful linear inequalities yielding Presburger formulae. An efficient representation of Presburger formulae would therefore, along the lines of this paper, immediately provide a symbolic verification of a guarded command language with Boolean combinations of linear inequalities as guards and linear expressions in assignments (including the extensions to automata and concurrent compositions).

References

- [1] M. Abadi and L. Lamport. An old-fashioned recipe for real-time. In *REX Workshop: Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 1–27. Springer-Verlag, 1991.
- [2] R. Alur and D. Dill. The theory of timed automata. In *Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 28–73. Springer-Verlag, 1991.
- [3] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data-structures for the verification of timed automata. In *Int. Workshop on Hybrid and Real-Time System*, March 1997.
- [4] F. Balarin. Approximate reachability analysis of timed automata. In *Proc. Real-Time Systems Symposium*, pages 52–61. IEEE, 1996.
- [5] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and Wang Yi. Efficient timed reachability analysis using clock difference diagrams. Technical Report DoCS Technical Report 99/105, Uppsala University, 1999.
- [6] W. Belluomini and C. J. Myers. Efficient timing analysis algorithms for timed state space exploration. In *Proc. Int. Symposium on Advanced Research in Asynchronous Circuits and Systems*, April 1997.
- [7] W. Belluomini and C. J. Myers. Verification of timed systems using POSETs. In *Proc. Computer Aided Verification (CAV)*, June 1998.
- [8] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL in 1995. In *Proc. International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055 of *LNCS*, pages 431–434. Springer-Verlag, March 1996.
- [9] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.

- [10] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In *Proc. Ninth International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *LNCS*, pages 179–190, 1997.
- [11] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
- [12] J. R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*. PhD thesis, Carnegie Mellon, August 1992.
- [13] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439. IEEE Computer Society Press, 1990.
- [14] S. V. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *Real-Time Systems Symposium*, pages 266–70. IEEE, December 1994.
- [15] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.
- [16] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1994.
- [17] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Proc. of Workshop on Verification and Control of Hybrid Systems*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, October 1995.
- [18] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Proc. 7th IFIP WG G.1 International Conference of Formal Description Techniques (FORTE)*, pages 227–242. Chapman & Hall, October 1994.
- [19] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proc. 16th IEEE Real-Time Systems Symposium (RTSS)*, pages 66–75. IEEE Computer Society Press, December 1995.
- [20] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer, 1989.
- [21] J.B.J. Fourier. Second extrait. In G. Darboux, editor, *Oeuvres*, pages 325–328, Paris, 1890. Gauthiers-Villars.
- [22] T. A. Henzinger, Z. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.
- [23] Alan J. Hu and David L. Dill. Efficient verification with BDDs using implicitly conjoined invariants. In *Proc. Fifth International Conference on Computer Aided Verification (CAV'93)*, volume 697 of *LNCS*, pages 3–14, 1993.

- [24] K. G. Larsen, P. Pettersson, and W. Yi. Model-checking for real-time systems. In *Proc. of the 10th Int. Conference on Fundamentals of Computation Theory*, volume 965 of *LNCS*, pages 62–88, August 1995.
- [25] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1+2), 1997.
- [26] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [27] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. Technical Report IT-TR-1999-023, Department of Information Technology, Technical University of Denmark, February 1999. A revised version will appear in proc. of Computer Science Logic (CSL'99), Madrid, 20–25 September 1999.
- [28] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. On the symbolic verification of timed systems. Technical Report IT-TR-1999-024, Department of Information Technology, Technical University of Denmark, February 1999.
- [29] T. G. Rokicki. *Representing and Modeling Digital Circuits*. PhD thesis, Stanford University, 1993.
- [30] T. G. Rokicki and C. J. Myers. Automatic verification of timed circuits. In D. L. Dill, editor, *Proc. Sixth International Conference on Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 468–480, 1994.
- [31] E. Verlind, G. de Jong, and B. Lin. Efficient partial enumeration for timing analysis of asynchronous systems. In *Proc. ACM/IEEE Design Automation Conference*, 1996.
- [32] H. Wong-Toi and D.L. Dill. Approximations for verifying timing properties. In *Theories and Experiences for Real-Time Systems Development*. World Scientific Publishing, 1994.
- [33] S. Yovine. Kronos: A verification tool for real-time systems. *Springer Int. Journal of Software Tools for Technology Transfer*, 1(1/2), October 1997.
- [34] S. Yovine. Model checking timed automata. In *Embedded Systems*, volume 1494 of *LNCS*. Springer-Verlag, 1998.