Symbolic Model Checking of Timed Guarded Commands using Difference Decision Diagrams ⋆

Jesper Møller, Henrik Hulgaard, Henrik Reif Andersen

Department of Innovation, The IT University of Copenhagen, Denmark

Abstract

We describe a novel methodology for analyzing timed systems symbolically. Given a formula representing a set of states, we describe how to determine a new formula that represents the set of states reachable by taking a discrete transition or by advancing time. The symbolic representations are given as formulae expressed in a simple first-order logic over difference constraints of the form $x - y \leq d$ which can be combined with Boolean operators and existentially quantified. We also show how to symbolically determine the set of states that can reach a given set of states (i.e., a backward step), thus making it possible to verify timed CTL-formulae symbolically. The main contribution is a way of advancing time symbolically essentially by quantifying out a special variable z which is used to represent the current zero point in time. We also describe a data structure called DDDs for representing difference constraint formulae, and we demonstrate the efficiency of the symbolic technique by analyzing two scheduling protocols using a DDD-based model checker.

Key words: real-time systems, timed automata, timed guarded commands, symbolic model checking, timed CTL, difference constraint systems, difference decision diagrams, quantifier elimination

1 Introduction

Model checking [15] is today used extensively for formal verification of finite state systems such as digital circuits and embedded software. The basic verification problem is to determine whether a given state of a system is reachable.

⋆ Financially supported by a grant from the Danish Technical Research Council. A preliminary version of this paper appeared in [43].

Email addresses: jm@it.edu (Jesper Møller), henrik@it.edu (Henrik Hulgaard), hra@it.edu (Henrik Reif Andersen).

The standard approach for solving this problem is to construct the set of reachable states R and then determine whether the given state is in R. 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 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 variables ranging over non-countable domains like for example real-time systems where time is modeled using continuous real variables and the behavior of a system is specified using constraints on these variables. To solve the reachability problem for a timed system, there are four key problems that have to be addressed:

(1) How to represent the infinite state space R of a timed system?
(2) How to tackle the state explosion problem for the discrete part of the state space?
(3) How to perform the basic verification operations (resetting clocks, advancing the time of clocks, etc.) on the representation to compute the reachable state space or to verify a temporal property of the system?
(4) How to determine whether two representations are equivalent?

A state in a timed system is a pair (s, v) where s is a discrete state (e.g., a marking of a Petri net, or a location in a timed automaton) and v is an assignment of values to the clocks in the system. Timed systems have an infinite number of states due to the dense domains of the clocks, so clock assignments are grouped into sets when analyzing timed systems. This allows the state space to be represented as a finite set of pairs (s, V) consisting of a discrete state s and the associated set of clock valuations V. The reachable state space R for a timed system can be determined by the generic algorithm in Fig. 1(a) where we view R as mapping a discrete state s to a set of convex clock valuations. The operator **post** fires all possible transitions and advances time from the set of states (s, V) such that each V_i is a convex set of clock valuations. The test in the line marked $(*)$ is performed by checking whether the clock valuation V_i is contained in any of the clock valuations used to represent $R[s_i]$.

1.1 Related Work

Model checking of timed systems (timed automata in particular; see [55] for a survey) has been extensively studied and a number of tools exist for verifying

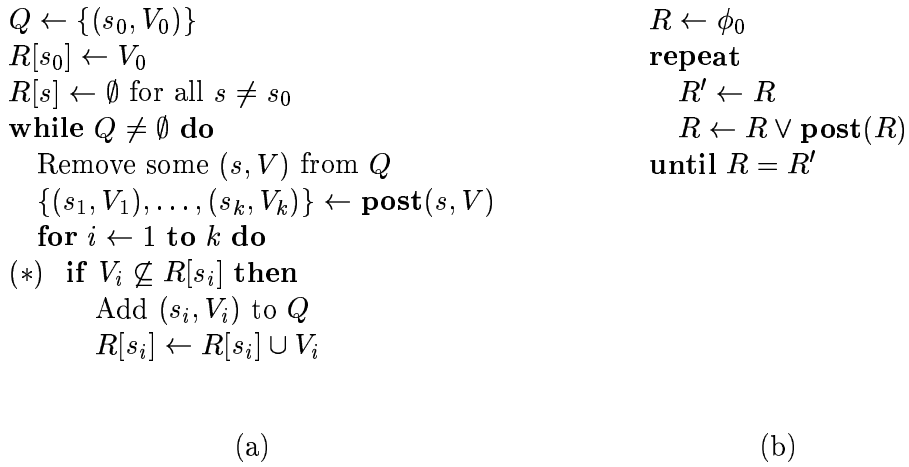


Fig. 1. Two approaches for constructing the set of reachable states R. (a) Outline of the algorithm used in current tools such as KRONOS and UPPAAL, and (b) a fully symbolic algorithm.

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 [14,10,12,25]. 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 [1] 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.

More recent timing analysis methods use difference bound matrices (DBMs) [24] for representing the timing information [8,38,47,54]. Each difference bound matrix can represent a convex set of clock assignments, thus to represent V, in general, a number of matrices are needed (i.e., representing V as a union of convex sets). Although DBMs provide a compact representation of a convex set of clock configurations, there are several problems with 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.
(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).

Several attempts have been made to remedy these shortcomings, for example by using partial order methods [48,52,6] or by using approximate meth-

ods [3,5,53]. Although these approaches do address the problem that the number of DBMs for representing the timing information can become very large, they are all inherently limited by the explicit enumeration of all discrete states.

1.2 Contributions

We present a methodology for analyzing timed systems symbolically, and describe a data structure for representing the state space of a timed system. We propose a simple notation called timed guarded commands for modeling a timed system. Timed guarded commands are similar to Dijkstra's guarded commands [23] extended with a finite number of clocks that can be tested in the guards and reset in the assignments. The notation is quite expressive: popular models of systems with time such as timed automata [1] and timed Petri nets [8] are easily encoded using timed guarded commands. Given a set of timed guarded commands, one may ask whether a given property is satisfied. We will focus on the basic question of whether a given combination of states is reachable, but also sketch how more general timing properties, expressed in a timed version of computation tree logic (CTL) [31], can be verified symbolically.

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 fixpoint iteration shown in Fig. 1(b). The core operation $\mathbf{post}(\phi)$ constructs the set of states reachable by taking any discrete transition, denoted by $\mathbf{post}_{\mathrm{d}}(\phi)$, or advancing time from a state satisfying ϕ, denoted by $\mathbf{post}_{\mathrm{t}}(\phi)$. Taking 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 [24] and also when solving systems of difference constraints [19].

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 by performing an existential quantification of z: Let P_{post} denote a predicate stating whether it is legal to advance time by changing the zero point from z to z'. Thus P_{post} will require that $z' \leq z$ since advancing time by some amount δ corresponds to decreasing the reference point z by δ. Typically, P_{post} will also include constraints expressing program invariants and urgency predicates. Now, a formula representing

the set of states reachable from ϕ by advancing time by δ is determined from:

$$\mathbf{post}_{\mathrm{t}}(\phi, \delta) = \Big(\exists z.(\phi \wedge P_{\mathrm{post}} \wedge z - z' = \delta)\Big)[z/z'] \,.$$

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

$$\mathbf{post}_{\mathrm{t}}(\phi) = \bigvee_{\delta \in \mathbb{R}} \mathbf{post}_{\mathrm{t}}(\phi, \delta) = \Big(\exists z.(\phi \wedge P_{\mathrm{post}})\Big)[z/z'] \,.$$

Another key contribution of this chapter is that we show that performing fully symbolic model checking of timed systems amounts to representing and deciding validity of difference constraint expressions which are first-order propositions of the form

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

where x and y are real-valued variables, and $d \in \mathbb{Q}$ is a constant. A practical model checking algorithm therefore requires a compact representation of difference constraint expressions, and an efficient decision procedure to determine validity of such expressions (including a procedure for quantifier elimination).

Henzinger et al. [31] 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 difference constraint expressions which have only one type of clock constraints ($x - y \leq d$) are 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). Second, 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.

1.3 Overview

This paper is organized as follows: In Sect. 2 we introduce a simple model of timed systems called timed guarded commands. Section 3 shows how to symbolically compute the set of reachable states and how to verify timed CTL-properties of timed guarded commands. Section 4 discusses various data structures and algorithms for implementing tools that can perform these symbolic analyses. In Sect. 5, we demonstrate the efficiency of the symbolic approach by analyzing Milner's scheduler and Fischer's protocol and comparing the results with the tools KRONOS and UPPAAL. Section 6 summarizes the contributions. Appendix A contains proofs of the theorems in Sect. 3.

2 Timed Guarded Commands

We present a simple notation called timed guarded commands [43] for modeling systems with time. Timed guarded commands are similar to Dijkstra's guarded commands [23] extended with a finite number of clocks that can be tested in the guards and reset in the assignments. The notation is quite expressive: popular models of systems with time such as timed automata [1] and timed Petri nets [8] are easily encoded using timed guarded commands. In this section we define the syntax and semantics of timed guarded commands, and in the following section we describe how to analyze timed guarded commands.

2.1 Syntax

We start with some basic definitions of the building blocks of timed guarded commands: variables, expressions, and commands.

Definition 1 (Variable) *Let $\mathcal{C}$ be a countable set of real-valued variables called clocks ranged over by x, and let $\mathcal{B}$ be a countable set of Boolean variables ranged over by b. The set of variables is $\mathcal{V} = \mathcal{B} \cup \mathcal{C}$.*

Next, we define a language for expressing propositions over Boolean variables and clocks:

Definition 2 (Expression) *Let Φ be the set of expressions of the form:*

$$\phi ::= x \sim d \mid x - y \sim d \mid b \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \,,$$

where $x, y \in \mathcal{C}$ are clocks, $b \in \mathcal{B}$ is a Boolean variable, $d \in \mathbb{Q}$ is a rational constant, $\sim \in \{\leq, <, =, \neq, >, \geq\}$ is a relational operator, and $\phi \in \Phi$ is an expression.

We use the tokens **false** and **true** to denote false and true expressions, respectively. The symbols $\neg$ (negation), $\wedge$ (conjunction), and $\vee$ (disjunction) have their usual meaning. The Boolean operators $\Rightarrow$ (implication) and $\Leftrightarrow$ (biimplication) are defined the standard way.

Definition 3 (Replacement) *Let $\phi \in \Phi$ be an expression, let $\vec{v} \in \mathcal{V}^n$ be an n-dimensional vector of variables, and let $\vec{r} \in (\mathbb{B} \cup \mathbb{Q})^n$ be an n-dimensional vector of values. Then the replacement $\phi[\vec{r}/\vec{v}\,]$ syntactically substitutes all occurrences of v_i by r_i, for $i = 1, \ldots, n$, in ϕ.*

Definition 4 (any-operator) *Let $\vec{v}, \vec{v'} \in \mathcal{V}^n$ be n-dimensional vectors of Boolean variables and clocks such that $v_i \neq v'_i$, for $i = 1, \ldots, n$, and let $\phi' \in \Phi$*

be an expression. Then the **any***-operator is written as:*

$$\vec{v} := \textbf{any}\ \vec{v'}.\phi'.$$

The **any**-operator is a nondeterministic assignment operator. Intuitively, the assignment

$$(v_1, \ldots, v_n) := \textbf{any}\ (v'_1, \ldots, v'_n).\phi'$$

has the following meaning: Assign to each clock or Boolean variable v_i any value v'_i, for $i = 1, \ldots, n$, such that the expression ϕ' is satisfied. If ϕ' is not satisfiable then the assignment has no effect (i.e., the variables $v_1, \ldots, v_n$ remain unchanged). Typically ϕ' is an expression over $v'_1, \ldots, v'_n$ and other variables. The choice of a value for a variable v' in an assignment is made nondeterministically and independently of choices for other variables v'' made in other assignments. [1]

We will often have to perform ordinary (deterministic) assignments, so we define the following shorthands:

$$\begin{array}{lcl} x := d & \equiv & x := \textbf{any}\ x'.x' = d \\ x := y + d & \equiv & x := \textbf{any}\ x'.x' - y = d \\ b := \phi & \equiv & b := \textbf{any}\ b'.b' \Leftrightarrow \phi \end{array}$$

Definition 5 (Command) *Let $\vec{v}, \vec{v'} \in \mathcal{V}^n$ be n-dimensional vectors of clocks and Boolean variables, and let $\phi' \in \Phi$ be an expression. Then a (timed guarded) command has the form*

$$\phi \rightarrow \vec{v} := \textbf{any}\ \vec{v'}.\phi'\ ,$$

where $\phi \in \Phi$ is called the guard. A command is said to be enabled if its guard evaluates to true.

A command specifies a conditional assignment: if the guard evaluates to true, then the command can be executed. Executing the command assigns a value to each variable on the left-hand side of the assignment.

Definition 6 (TGC program) *A timed guarded command (TGC) program P is a tuple (B, C, T, I), where $B \subseteq \mathcal{B}$ is a set of Boolean variables, $C \subseteq \mathcal{C}$ is a set of clocks, T is a set of commands over $B \cup C$, and $I \in \Phi$ is the program invariant.*

[1] An alternative to this independent-choice strategy is a fixed-choice strategy. In a fixed-choice strategy, if two expressions ϕ' and ϕ'' are equivalent then the two values bound to v' and v'' are identical. The fixed-choice **any** operator is also known as Hilbert's ϵ-operator [32]. The independent-choice **any** operator we use is identical to Blass and Gurevich's δ-operator [9].

The semantics of a TGC program is a transition system where the set of states are value assignments of the variables, and the transitions between states correspond to either executing commands in the program or advancing time by some amount.

Definition 7 (State) *A state of a* TGC *program* $P = (B, C, T, I)$ *is an interpretation of the Boolean variables and clocks. For a vector of variables* $\vec{v} \in (B \cup C)^n$, $s(\vec{v}) \in (\mathbb{B} \cup \mathbb{Q})^n$ *denotes the interpretation of* $\vec{v}$ *in the state* s. *A state* s *satisfies an expression* ϕ, *written* $s \models \phi$, *if* ϕ *evaluates to true in the state* s, *and we write* $[\![\phi]\!]$ *for the set of states that satisfy* ϕ.

Definition 8 (State update) *Let* $\vec{v}$ *be an* n*-dimensional vector of variables, and let* $\vec{r} \in (\mathbb{B} \cup \mathbb{Q})^n$ *be an* n*-dimensional vector of values. Then the state* $s' = s[\vec{v} := \vec{r}]$ *is equivalent to* s *except that* $s'(\vec{v}) = \vec{r}$.

We now define the transitions between states. 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 the variables according to the multi-assignment, and letting time pass uniformly increases the values of all clocks by some amount δ.

Definition 9 (Discrete transition) *Let* $P = (B, C, T, I)$ *be a* TGC *program. Then the discrete transition* $\xrightarrow{t}$ *for a timed guarded command* $t \in T$ *of form* $\phi \to \vec{v} := \textbf{any}\, \vec{v'}.\phi'$ *is defined by the following inference rule:*

$$\frac{s \models \phi \quad s[\vec{v'} := \vec{r}] \models \phi' \quad s[\vec{v} := \vec{r}] \models I}{s \xrightarrow{t} s[\vec{v} := \vec{r}]}.$$

Definition 10 (Timed transition) *Let* $P = (B, C, T, I)$ *be a* TGC *program. The timed transition* $\xrightarrow{\delta}$ *for advancing all clocks by* δ *is defined by the following inference rule:*

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

where $\delta, \delta' \in \mathbb{Q}$, $\vec{c}$ *denotes a vector of all clocks in* C, *and* $\vec{c} + \delta$ *denotes the vector where* δ *is added to each clock in* $\vec{c}$.

Definition 11 (Semantics) *The semantics of a* TGC *program is a transition system* $(\mathcal{S}, \to)$, *where* $\mathcal{S}$ *is the set of states of the program, and* $\to$ *is the transition relation as defined in Definitions 9 and 10.*

Given a transition system $(\mathcal{S}, \rightarrow)$ for a TGC program $P = (B, C, T, I)$ and a set of states $S \subseteq \mathcal{S}$, we now define various sets of states reachable from S by discrete or timed transitions:

Definition 12 (Discrete successor) *Let $(\mathcal{S}, \rightarrow)$ be the transition system for a* TGC *program $P = (B, C, T, I)$, and let $S \subseteq \mathcal{S}$ be a set of states. The set of states reachable from S by executing the command $t \in T$ is given by:*

$$Post_{\mathrm{d}}(S, t) = \{s' : \exists s \in S.\ s \xrightarrow{t} s'\}\,.$$

The set of states reachable from S by executing any timed guarded command in T is given by:

$$Post_{\mathrm{d}}(S) = \bigcup_{t \in T} Post_{\mathrm{d}}(S, t).$$

Definition 13 (Timed successor) *Let $(\mathcal{S}, \rightarrow)$ be the transition system for a* TGC *program $P = (B, C, T, I)$, and let $S \subseteq \mathcal{S}$ be a set of states. The set of states reachable from S by advancing time by δ is given by:*

$$Post_{\mathrm{t}}(S, \delta) = \{s' : \exists s \in S.\ s \xrightarrow{\delta} s'\}\,.$$

The set of states reachable from S by advancing time by an arbitrary amount is given by:

$$Post_{\mathrm{t}}(S) = \bigcup_{\delta \in \mathbb{R}} Post_{\mathrm{t}}(S, \delta).$$

Definition 14 (Reachable states) *Let $(\mathcal{S}, \rightarrow)$ be the transition system for a* TGC *program $P = (B, C, T, I)$, and let $S \subseteq \mathcal{S}$ be a set of states. The set of states reachable from S by executing any timed guarded command or advancing time by an arbitrary amount is given by:*

$$Post(S) = Post_{\mathrm{d}}(S) \cup Post_{\mathrm{t}}(S)\,.$$

The set of states reachable from S is given by:

$$Post^*(S) = \mu X[S \cup Post(X)],$$

where $\mu X[S \cup Post(X)]$ is the least fixpoint of $S \cup Post(X)$.

The least fixpoint $\mu X[f(X)]$ of a function f can be determined by computing a series of approximations $f(\emptyset), f(f(\emptyset)), \ldots$, until a fixpoint is reached [50], that is, until $f^i(\emptyset) \equiv f^{i+1}(\emptyset)$, for some i. There exists (contrived) timed systems where the fixpoint computation does not terminate, but as in the traditional analysis of timed automata, it is possible to determine subclasses of timed guarded commands for which termination is ensured.

3 Symbolic Model Checking

The previous section describes how to model timed systems as TGC programs. In this section we develop the necessary theory for verifying properties of a TGC program. Given a set of states represented by a formula, we determine a new formula that represents the set of states reachable by executing timed guarded commands according to the inference rule in Definition 9 or by advancing time according to the inference rule in Definition 10.

3.1 Difference Constraint Expressions

Recall the definition of difference constraint expressions from the introduction:

Definition 15 (Difference Constraint Expression) *Let Ψ be the set of difference constraint expressions of the form:*

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

where $x, y \in \mathcal{C}$ are clocks, $d \in \mathbb{Q}$ is a rational constant, and $\psi \in \Psi$ is a difference constraint expression.

The following procedure describes how to transform any expression $\phi \in \Phi$ generated by the grammar in Definition 2 to a difference constraint expression $\phi_z \in \Psi$ by introducing a new variable z (denoting "zero").

Definition 16 *Let $\phi \in \Phi$ be an expression. The corresponding difference constraint expression $\phi_z \in \Psi$ is obtained by performing the following three steps: (1) Replace each Boolean variable $b_i \in \mathcal{B}$ in ϕ by a difference constraint $x_i - x_i' \leq 0$, where $x_i, x_i' \in \mathcal{C}$ are clocks only used in the encoding of b_i.*[2] *(2) Replace each constraint of the form $x \sim d$ in ϕ by the difference constraint $x - z \sim d$. (3) Express each difference constraint of the form $x - y \sim d$ in ϕ in terms of the relational operator $\leq$.*

As we shall see in the following, eliminating constraints of the form $x \sim d$ from the grammar in Definition 2 makes it possible to add δ to all clocks simultaneously by decreasing the common reference-point z by δ (Theorem 24). Furthermore, advancing time by any value δ can be computed by an existential quantification of z (Theorem 25).

[2] It turns out that when using difference decision diagrams (see Sect. 4.4) with this apparently strange encoding of Boolean variables, the Boolean manipulations can be done as efficiently as when using BDDs.

We use $[\![\psi]\!]_z$ as a shorthand for $[\![\exists z.(\psi \wedge z = 0)]\!]$; that is, $[\![\psi]\!]_z$ is the set of states that satisfy ψ when z is equal to 0. It is easy to prove the following proposition:

Proposition 17 *Let $\phi \in \Phi$ be an expression generated from the grammar in Definition 2, and let ϕ_z be the corresponding difference constraint expression obtained as described above. Then $[\![\phi]\!] = [\![\phi_z]\!]_z$.*

We define two useful operators on difference constraint expressions: replacement and assignment.

Definition 18 (Replacement) *Replacement of a vector $\vec{v'} \in \mathcal{V}^n$ of variables by another vector $\vec{v} \in \mathcal{V}^n$ of variables plus a vector $\vec{d} \in \mathbb{Q}^n$ of constants, where $v_i \neq v'_i$ for each $i = 1, \ldots, n$, in an expression ψ is defined as follows:*

$$\psi[(\vec{v} + \vec{d})/\vec{v'}] = \exists \vec{v'}.(\psi \wedge \vec{v'} - \vec{v} = \vec{d}) \,,$$

where $\exists \vec{v'}.\psi$ is a shorthand for $\exists v'_1 \cdots \exists v'_n.\psi$, and $\vec{v'} - \vec{v} = \vec{d}$ is a shorthand for $v'_1 - v_1 = d_1 \wedge \cdots \wedge v'_n - v_n = d_n$.

Definition 19 (Assignment) *Assignment of a vector $\vec{v} \in \mathcal{V}^n$ of variables to a vector $\vec{v'} \in \mathcal{V}^n$ of variables such that the expression ψ' holds is defined as follows:*

$$\psi[\vec{v} := \textbf{any}\ \vec{v'}.\psi'] = \Big(\exists \vec{v}.(\psi \wedge \psi')\Big)[\vec{v}/\vec{v'}] \,.$$

3.2 Forward Analysis

Given a difference constraint expression $\psi \in \Psi$ representing a set of states $[\![\psi]\!]_z \subseteq \mathcal{S}$, we now show how to determine an expression representing the set of states reachable from $[\![\psi]\!]_z$.

Definition 20 ($\mathbf{post}_{\mathrm{d}}$-operator) *Let ψ be a difference constraint expression, and let $P = (B, C, T, I)$ be a* TGC *program. Then the $\mathbf{post}_{\mathrm{d}}$-operator for (forward) execution of the command $\phi \to \vec{v} := \textbf{any}\ \vec{v'}.\phi'$ is defined as*

$$\mathbf{post}_{\mathrm{d}}(\psi, \phi \to \vec{v} := \textbf{any}\ \vec{v'}.\phi') = (\psi \wedge \phi_z)[\vec{v} := \textbf{any}\ \vec{v'}.\phi'_z] \wedge I_z \,,$$

The $\mathbf{post}_{\mathrm{d}}$-operator for execution of any command in T is defined as

$$\mathbf{post}_{\mathrm{d}}(\psi) = \bigvee_{t \in T} \mathbf{post}_{\mathrm{d}}(\psi, t) \,.$$

The $\mathbf{post}_{\mathrm{d}}$-operator restricts ψ to the subset where the guard ϕ holds, performs the nondeterministic assignment defined in terms of the **any**-operator, and restricts the resulting set to the subset where the program invariant I holds.

Theorem 21 (Correctness of $\mathbf{post}_{\mathrm{d}}(\psi, t)$) *Let ψ be a difference constraint expression, and let $P = (B, C, T, I)$ be a* TGC *program. Then $Post_{\mathrm{d}}([\![\psi]\!]_z, t) = [\![\mathbf{post}_{\mathrm{d}}(\psi, t)]\!]_z$ for any command $t \in T$.*

Theorem 22 (Correctness of $\mathbf{post}_{\mathrm{d}}(\psi)$) *Let ψ be a difference constraint expression, and let $P = (B, C, T, I)$ be a* TGC *program. Then $Post_{\mathrm{d}}([\![\psi]\!]_z) = [\![\mathbf{post}_{\mathrm{d}}(\psi)]\!]_z$.*

Next, we define the operator $\mathbf{post}_{\mathrm{t}}$ for advancing time symbolically from a set of states $[\![\psi]\!]_z$. The key idea is to change 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. We require that the program invariant holds in z' and at all intermediate points in time.

Definition 23 ($\mathbf{post}_{\mathrm{t}}$-operator) *Let ψ be a difference constraint expression, and let $P = (B, C, T, I)$ be a* TGC *program. Then the $\mathbf{post}_{\mathrm{t}}$-operator for advancing time by δ in all states $[\![\psi]\!]_z$ is defined as*

$$\begin{aligned}\mathbf{post}_{\mathrm{t}}(\psi, \delta) &= (\psi \wedge P_{\mathrm{post}})[z := z - \delta] \\ &= \Big(\exists z.(\psi \wedge P_{\mathrm{post}} \wedge z - z' = \delta)\Big)[z/z']\end{aligned}$$

where the last equality follows from the definition of assignment, and where

$$P_{\mathrm{post}} = (z' \leq z) \wedge \forall z''.\Big((z' \leq z'' \leq z) \Rightarrow I_{z''}\Big).$$

The $\mathbf{post}_{\mathrm{t}}$-operator for advancing time by an arbitrary amount in all states $[\![\psi]\!]_z$ is defined as

$$\mathbf{post}_{\mathrm{t}}(\psi) = \bigvee_{\delta \in \mathbb{R}} \mathbf{post}_{\mathrm{t}}(\psi, \delta)$$

Theorem 24 (Correctness of $\mathbf{post}_{\mathrm{t}}(\psi, \delta)$) *Let ψ be a difference constraint expression, and let $P = (B, C, T, I)$ be a* TGC *program. Then $Post_{\mathrm{t}}([\![\psi]\!]_z, \delta) = [\![\mathbf{post}_{\mathrm{t}}(\psi, \delta)]\!]_z$ for any delay $\delta \in \mathbb{R}$.*

Theorem 25 (Correctness of $\mathbf{post}_{\mathrm{t}}(\psi)$) *Let ψ be a difference constraint expression, and let $P = (B, C, T, I)$ be a* TGC *program. Then $Post_{\mathrm{t}}([\![\psi]\!]_z) = [\![\mathbf{post}_{\mathrm{t}}(\psi)]\!]_z = [\![\exists z.(\psi \wedge P_{\mathrm{post}})[z/z']]\!]_z$.*

The $\mathbf{post}_{\mathrm{d}}$-operator and $\mathbf{post}_{\mathrm{t}}$-operator form the basis for constructing the set of reachable states symbolically. The operator $\mathbf{post}(\psi)$ determines the set of states which can be reached by taking either a discrete or a timed transition from a state in $[\![\psi]\!]_z$ and is defined as follows:

Definition 26 (post-operator) *Let ψ be a difference constraint expression, and let $P = (B, C, T, I)$ be a* TGC *program. Then the* **post***-operator for executing any command in T or advancing time by an arbitrary amount in all*

states $[\![\psi]\!]_z$ is defined as:

$$\mathbf{post}(\psi) = \mathbf{post}_{\mathrm{d}}(\psi) \vee \mathbf{post}_{\mathrm{t}}(\psi)\,.$$

The $\mathbf{post}^*$*-operator is defined as:*

$$\mathbf{post}^*(\psi) = \mu X[\psi \vee \mathbf{post}(X)]\,.$$

where $\mu X[\psi \vee \mathbf{post}(X)]$ is the least fixpoint of $\psi \vee \mathbf{post}(X)$.

The difference constraint expressions constructed by the **post**-operator and $\mathbf{post}^*$-operator in Definition 26 correspond exactly to the set of successors and reachable states, respectively, as defined by *Post* and *Post** in Definition 14:

Theorem 27 (Correctness of $\mathbf{post}(\psi)$ and $\mathbf{post}^*(\psi)$) *Let ψ be a difference constraint expression, and let $P = (B, C, T, I)$ be a* TGC *program. Then $Post([\![\psi]\!]_z) = [\![\mathbf{post}(\psi)]\!]_z$ and $Post^*([\![\psi]\!]_z) = [\![\mathbf{post}^*(\psi)]\!]_z$.*

3.3 Backward Analysis

Similarly to the **post** operators defined in the previous section we can define a number of **pre** operators for determining formulae for the set of states that can reach $[\![\psi]\!]_z$. These operators can for example be used to compute the set of states that satisfy a timed CTL formula.

Definition 28 ($\mathbf{pre}_{\mathrm{d}}$-operator) *Let ψ be a difference constraint expression, and let $P = (B, C, T, I)$ be a* TGC *program. Then the $\mathbf{pre}_{\mathrm{d}}$-operator for backward execution of the command $\phi \to \vec{v} := \mathbf{any}\, \vec{v'}.\phi'$ is defined as*

$$\mathbf{pre}_{\mathrm{d}}(\psi, \phi \to \vec{v} := \mathbf{any}\, \vec{v'}.\phi') = (\psi[\vec{v'}/\vec{v}] \wedge \phi_z)[\vec{v'} := \mathbf{any}\, \vec{v}.\phi'_z] \wedge I_z\,.$$

The $\mathbf{pre}_{\mathrm{d}}$-operator for backward execution of any command in T is defined as

$$\mathbf{pre}_{\mathrm{d}}(\psi) = \bigvee_{t \in T} \mathbf{pre}_{\mathrm{d}}(\psi, t)\,.$$

Similar to the $\mathbf{post}_{\mathrm{d}}$-operator, the $\mathbf{pre}_{\mathrm{d}}$-operator restricts ψ to the subset where the guard ϕ_z holds before the assignment, performs the nondeterministic assignment backward, and restricts the resulting set to the subset where the program invariant I holds. Effectively, this constructs the set of states that can reach $[\![\psi]\!]_z$ by backward execution of the command $\phi \to \vec{v} := \mathbf{any}\, \vec{v'}.\phi'$.

The set of states that can reach $[\![\psi]\!]_z$ by advancing time by δ is determined analogously to the forward case:

Definition 29 ($\mathbf{pre}_{\mathrm{t}}$-operator) *Let ψ be a difference constraint expression, and let $P = (B, C, T, I)$ be a* TGC *program. Then the $\mathbf{pre}_{\mathrm{t}}$-operator for decreasing time by δ in all states $[\![\psi]\!]_z$ is defined as*

$$\begin{aligned}\mathbf{pre}_{\mathrm{t}}(\psi, \delta) &= (\psi \wedge P_{\mathrm{pre}})[z := z + \delta] \\ &= \Big(\exists z.(\psi \wedge P_{\mathrm{pre}} \wedge z' - z = \delta)\Big)[z/z']\end{aligned}$$

where the last equality follows from the definition of assignment, and where P_{pre} is equivalent to P_{post} with z and z' exchanged:

$$P_{\mathrm{pre}} = (z \leq z') \wedge \forall z''.\Big((z \leq z'' \leq z') \Rightarrow I_{z''})\Big)\,.$$

The $\mathbf{pre}_{\mathrm{t}}$-operator for decreasing time by an arbitrary amount in all states $[\![\psi]\!]_z$ is defined as

$$\mathbf{pre}_{\mathrm{t}}(\psi) = \bigvee_{\delta \in \mathbb{R}} \mathbf{pre}_{\mathrm{t}}(\psi, \delta)$$

For completeness we also define the set of states that can reach $[\![\psi]\!]_z$ by executing commands or advancing time:

Definition 30 (pre-operator) *Let ψ be a difference constraint expression, and let $P = (B, C, T, I)$ be a* TGC *program. Then the* **pre***-operator for backward execution of any command in T or decreasing time by an arbitrary amount in all states $[\![\psi]\!]_z$ is defined as:*

$$\mathbf{pre}(\psi) = \mathbf{pre}_{\mathrm{d}}(\psi) \vee \mathbf{pre}_{\mathrm{t}}(\psi)\,.$$

The $\mathbf{pre}^*$-operator is defined as:

$$\mathbf{pre}^*(\psi) = \mu X[\psi \vee \mathbf{pre}(X)]\,.$$

where $\mu X[\psi \vee \mathbf{pre}(X)]$ is the least fixpoint of $\psi \vee \mathbf{pre}(X)$.

The correctness of the **pre** operators can easily be proved, analogously to the **post** operators.

3.4 Verification of Properties

The **post**- and **pre**-operators can be used to verify simple propositional properties of a TGC program. There are basically two techniques for verifying that a property is an invariant for a program:

Definition 31 (Forward reachability) *Let ψ be a difference constraint expression, and let $P = (B, C, T, I)$ be a* TGC *program with the initial state $\psi_0 \in \Psi$. Then ψ holds invariantly for P if $\mathbf{post}^*(\psi_0) \Rightarrow \psi$ is a tautology.*

Definition 32 (Backward reachability) *Let ψ be a difference constraint expression, and let $P = (B, C, T, I)$ be a* TGC *program with the initial state $\psi_0 \in \Psi$. Then ψ holds invariantly for P if $\neg(\psi_0 \wedge \mathbf{pre}^*(\neg\psi))$ is a tautology.*

The **pre** operators can also be used to perform symbolic model checking of timed CTL [31]. Timed CTL is obtained by extending CTL [15] with an auxiliary set of clocks called specification clocks. These clocks do not appear in the model but are used to express timing bounds on the temporal operators. A timed CTL formula has the following form:

$$\theta ::= \phi \mid \neg\theta \mid \theta \wedge \theta \mid u.\theta \mid \theta_1 \,\exists\mathcal{U}\, \theta_2 \mid \theta_1 \,\forall\mathcal{U}\, \theta_2 \,.$$

The atomic predicates ϕ of timed CTL are expressions, see Definition 2. A specification clock u can be bound and reset by a freeze quantifier $u.\psi$ [2]. The operators $\exists\mathcal{U}$ and $\forall\mathcal{U}$ are the existential and universal path quantifiers. Symbolically, we can find the set of states satisfying a given timed CTL formula ψ by a backward computation using a fixpoint iteration for the temporal operators. For instance, the set of states satisfying the expression "along some execution path, ψ_1 holds until ψ_2 holds", is computed symbolically as:

$$\psi_1 \,\exists\mathcal{U}\, \psi_2 = \mu X[\psi_2 \vee (\psi_1 \wedge \mathbf{pre}(X))].$$

The set of states satisfying the freeze quantifier is computed symbolically as:

$$u.\psi = \exists u.(\psi \wedge u - z = 0) = \psi[z/u],$$

assuming ψ uses z as zero point.

3.5 Urgent Commands

A command $t \in T$ is called urgent if it is required to execute instantaneously when the guard becomes enabled. This corresponds to restricting when time can advance. Given a set $T' \subseteq T$ of urgent timed guarded commands with guards $\phi_1, \ldots, \phi_m$, we define the urgency predicate as $U = \phi_1 \vee \cdots \vee \phi_m$. That is, the urgency predicate is an expression which specifies when it is illegal to advance time. We can use this predicate to ensure that these urgent commands will always be executed as soon as they become enabled—that is, time is not allowed to advance if one or more of these commands are enabled.

Consider a state $s \in [\![\psi]\!]_z$. We can only take a timed transition $s \xrightarrow{\delta} s'$ if there are no urgent commands enabled in s. Thus, we add an additional requirement to P_{post} ensuring that no urgent transitions are enabled while advancing time (except in the end point):

$$P^{\text{urg}}_{\text{post}} = P_{\text{post}} \wedge \forall z''.\Big((z' < z'' \leq z) \Rightarrow \neg U_{z''}\Big)\,.$$

In the following, we show how to simplify the syntax and semantics of a TGC program substantially. The key idea is that a timed transition essentially is the same as a discrete transition in the sense that a guard specifies when the transition can be taken, and an assignment updates the values of clocks. This leads to a semantics with only one type of transitions. Furthermore, we can embed program invariants, urgency predicates, and guards in the expression of an assignment. We start by defining a simplified TGC program, and then show how to translate a TGC program into a simplified TGC program.

Definition 33 (STGC syntax) *A simplified* TGC *(*STGC*) program P is a tuple (B, C, T), where $B \subseteq \mathcal{B}$ is a set of Boolean variables, $C \subseteq \mathcal{C}$ is a set of clocks, T is a set of commands of the form $\vec{v} := \textbf{any}\ \vec{v'}.\phi$, where $\vec{v}, \vec{v'} \in B \cup C$ are vectors of variables and $\phi \in \Phi$ is an expression.*

Definition 34 (STGC semantics) *The semantics of an* STGC *program $P = (B, C, T)$ is a transition system $(\mathcal{S}, \rightarrow)$ where the set of states $\mathcal{S}$ are value assignments of the variables as given in Definition 7. For each command $t \in T$ of the form $\vec{v} := \textbf{any}\ \vec{v'}.\phi$, the following inference rule defines the transition relation:*

$$\frac{s[\vec{v'} := \vec{r}] \models \phi}{s \xrightarrow{t} s[\vec{v} := \vec{r}]}$$

Analogous to Definition 20 we define the operator $\textbf{post}_{\text{STGC}}$ as follows:

$$\textbf{post}_{\text{STGC}}(\psi, \vec{v} := \textbf{any}\ \vec{v'}.\phi) = \psi[\vec{v} := \textbf{any}\ \vec{v'}.\phi_z].$$

Next, we show to translate a TGC program into an equivalent STGC program.

Definition 35 (Induced STGC) *A* TGC *program $P = (B, C, T, I)$ induces an* STGC *program $P' = (B, C, T')$, such that for each timed guarded command $\phi \rightarrow \vec{v} := \textbf{any}\ \vec{v'}.\phi'$ in T there is a command $\vec{v} := \textbf{any}\ \vec{v'}.\Big(\phi' \wedge \phi \wedge I[\vec{v'}/\vec{v}]\Big)$ in T'. Furthermore, T' contains the command $z := \textbf{any}\ z'.P_{\text{post}}$, where P_{post} is defined as in Definition 23.*

Theorem 36 *Let P be a* TGC *program, and let P' be the induced* STGC *program. Then P and P' define the same transition system.*

Using the symmetry between P_{pre} and P_{post}, it is easy to prove the following:

Theorem 37 *Let ψ be a difference constraint expression, and let P be a* TGC *program. Then the $\textbf{pre}_{\text{t}}$-operator for decreasing time in all states $[\![\psi]\!]_z$ can be defined as:*

$$\textbf{pre}_{\text{t}}(\psi) = \textbf{pre}_{\text{d}}(\psi, \textbf{true} \rightarrow z := \textbf{any}\ z'.P_{\text{post}})$$

4 Algorithms and Data Structures

The previous two sections define a notation called timed guarded commands for modeling timed systems and a technique called symbolic model checking for analyzing timed systems. To verify a property of a timed system, we start with some initial expression ψ_0 (either the initial state of the system in the forward analysis, or the negation of the property to verify in the backward analysis) and then compute a sequence of fixpoint approximations $\psi_0, \psi_1, \ldots$, until $\psi_i = \psi_{i+1}$ for some i. Two expressions ψ_i and ψ_{i+1} are equivalent if and only if the expression $\neg(\psi_i \Leftrightarrow \psi_{i+1})$ is not satisfiable.

The core operation in the fixpoint computation is thus to determine whether a difference constraint expression is satisfiable. We call this problem DCE-SAT. Interestingly, this problem has, to our best knowledge, only been studied by very few researchers; the primary focus has been on theories that are either more expressive, such as reals with addition and order, or less expressive, such as quantified Boolean formulae. An important aspect in verification of timed systems is the ability to deal with many discrete states together with the infinite nature of the real-valued variables. There are very few results on how to do this efficiently, and it remains an open problem to find algorithms and data structures that work just as well for timed systems as BDDs do for non-timed systems.

In general, the reachability problem for timed guarded commands is undecidable (i.e., the fixpoint computation might not terminate). It is straightforward to model a register-machine as a TGC program using a variable for each unbounded register, and a variable for the program counter. Test, increment, and decrement are easily expressed in assignments. An interesting task would be to identify conditions on TGC programs for which questions such as reachability are decidable. This has not been done yet.

In this section we give a survey of the available algorithms and data structures for solving the DCE-SAT problem. We start by giving an overview of some of the problems (both simpler and harder) which are related to DCE-SAT. Then we briefly describe a technique called quantifier elimination, and finally we describe two algorithmic approaches for solving DCE-SAT based on matrices and graphs data structures, respectively.

4.1 Complexity Overview

Let us first look at two subsets of difference constraint expressions and see how difficult it is to determine satisfiability for these simpler theories. The first problem is DCS-SAT: determine satisfiability of conjunctions of difference

inequalities (also called a difference constraint system), which is an expression of the form

$$\psi ::= x - y \leq d \mid \psi_1 \wedge \psi_2.$$

This problem can be solved time $O(n^3)$ where n is the number of variables using shortest-paths algorithms such as Bellman-Ford or Floyd-Warshall [19]. Hence, DCS-SAT is in the complexity class **P**.

The second problem is QFDCE-SAT: determine satisfiability of a quantifier-free difference constraint expression, which is an expression of the form

$$\psi ::= x - y \leq d \mid \psi_1 \wedge \psi_2 \mid \neg\psi \,.$$

Quantifier-free difference constraint expressions further allow negation, which, in combination with conjunction, also gives disjunction and strong inequality. Adding negation greatly increases the expressive power and, correspondingly, complicates the problem of determining satisfaction. The QFDCE-SAT problem is **NP**-complete, see for example [31] for a proof.

The main problem is DCE-SAT: determine satisfiability of a difference constraint expression as defined in Definition 15:

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

Interestingly, it turns out that adding quantifiers "only" makes the problem **PSPACE**-complete [36]. Considering that we are working with variables over infinite domains, it is surprising that DCE-SAT is no harder than QBF-SAT (satisfiability of quantified Boolean formulae) which is also **PSPACE**-complete [45].

It is also interesting to note that DCE-SAT is easier than satisfiability of slightly more general theories such as the first-order theory of reals with addition and order, which is **NEXP**-hard [33], and the theory of integers with addition and order (also called Presburger arithmetic [46]), which is **2-NEXP**-hard [18,28,49]. The theory of reals with addition and multiplication was shown to be decidable by Tarski [51], whereas the theory of integers with addition and multiplication (also called number theory) is undecidable—Gödel's famous Incompleteness Theorem [30].

It is of course always possible to solve each of the three satisfaction problems defined above using a decision procedure for a more general theory. For example, difference constraint systems can be decided using linear programming [35]; quantifier-free difference constraint expressions can be decided using disjunctive programming [4]; and difference constraint expressions can be decided by eliminating the quantifiers (as described in the following section) and then using the same method as for quantifier-free expressions.

Adding quantifiers to the vocabulary of a language greatly increases the expressive power, but also moves the satisfiability problem up in the hierarchy of complexity classes. For quantifier-free difference constraint expressions, the satisfiability problem goes from **NP**-complete to **PSPACE**-complete. The same holds for (quantifier-free) Boolean expressions, cf. [17,33]. For Presburger arithmetic the difference is even bigger: **NP**-complete for quantifier-free formulae [44], but **2-NEXP**-hard for formulae with quantifiers.

Since Tarski [51] showed that the theory of reals with addition and multiplication, which subsumes the theory of difference constraint expressions, admits quantifier elimination, there has been a substantial amount of research in developing efficient algorithms for eliminating quantifiers. Quantifier elimination consists of constructing a quantifier-free formula ψ' equivalent to a given quantified formula $\exists x.\psi$.

Tarski used quantifier elimination to obtain a decision procedure for the theory of reals with addition and order. The idea is to existentially quantify out all free variables, yielding a new, variable-free formula whose truth value can be evaluated. The original formula is satisfiable if and only if this new formula evaluates to true.

There exists a number of algorithms for eliminating quantifiers in each of the different theories mentioned above. The Fourier-Motzkin method [29] eliminates quantifiers from an existentially quantified linear program, essentially by isolating the quantified variable in each inequality (which gives a set of inequalities of the form $x \leq t_i$ and $t_j \leq x$) and then adding a new inequality for each possible combination (e.g., $t_j \leq t_i$).

Cooper's algorithm [18] eliminates quantifiers from Presburger formulae, and Ferrante and Rackoff [27] give an elimination procedure for the theory of reals with addition and order based on elimination sets. Ferrante and Geiser [26] study quantifier elimination in the theory of rationals with addition and order. And Collins [16] pioneered the development of cylindrical algebraic decomposition (CAD) techniques for quantifier elimination in the theory of reals with addition and multiplication. In [40], Loos and Weispfenning improve Ferrante and Rackoff's original algorithm by reducing the size of the elimination sets. Koubarakis [36] was the first researcher to study the complexity of quantifier elimination in the theory of difference constraint expressions.

Common for all of these quantifier elimination algorithms is that they are intended to work on the syntactic level of an expression (e.g., represented as a syntax tree). In the following we discuss two other approaches for solving DCE-SAT which are based on matrices and graphs, respectively.

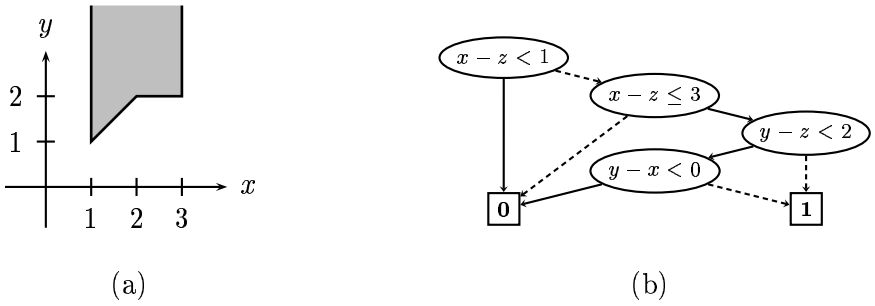
Any quantifier-free difference constraint expression can be written in disjunctive normal form where each disjunct is a difference constraint system. The key observation is now that a difference constraint system can be interpreted as a constraint graph: each variable x_i in the constraint system becomes a node in the graph, and for each difference constraint $x_i - x_j \leq d$ there is an edge from x_j to x_i with weight d [3]. The constraint graph can be represented as a so-called difference bound matrix (DBM) [24] M, where M_{ij} contain the least upper bound on $x_j - x_i$ or ∞ if there is no upper bound. It is easy to show that the difference constraint system for the graph is satisfiable if and only if the constraint graph has a cycle with negative weight, and again the Floyd-Warshall algorithm can be used to determine whether the graph has such a negative-weight cycle.

In other words, a quantifier-free difference constraint expression can be represented as a list of DBMs. Each DBM can represent a convex set of values, thus the expression is represented as a union of convex sets. Existential quantifiers can be distributed on each DBM and eliminated by running the Floyd-Warshall algorithm on the matrix and removing the rows and columns that correspond to the quantified variables. Negation and conjunction are more complicated to perform since they require that the expression is recast into disjunctive normal form. Difference bound matrices are used in many real-time verification tools, such as KRONOS [54] and UPPAAL [39], but they suffer from a number of problems as discussed in Sect. 1.1.

4.4 Difference Decision Diagrams

Difference decision diagrams (DDDs) [42] are another candidate for a data structure for representing difference constraint expressions. Similar to how a BDD [11] represents the meaning of a Boolean formula implicitly, a DDD represents the meaning $[\![\psi]\!]$ of a difference constraint expression ψ using a decision diagram in which the vertices contain difference constraints. A DDD is a directed acyclic graph (V, E) with two terminals **0** and **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

[3] If we also allow strong difference constraints of the form $x_i - x_j < d$, the weights in the graph become pairs of the form $(<, d)$ or $(\leq, d)$.



(a) (b)

Fig. 2. The expression $\psi = (1 \leq x - z \leq 3) \wedge \big((y - z \geq 2) \vee (y - x \geq 0)\big)$ as (a) an (x, y)-plot for $z = 0$, and (b) a difference decision diagram.

expression ψ^v given by:

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

where $\alpha(v)$ is the difference constraint of v, and $high(v)$ and $low(v)$ are the high- and low-branches, respectively. Figure 2 shows an example of a DDD.

As shown in [42], DDDs can be ordered and reduced, yielding a semicanonical form, which makes it possible to check for validity and satisfiability in constant time (as for BDDs). The DDD data structure is not canonical, however, so equivalence checking must be performed as a validity check. The operations for constructing and manipulating DDDs according to the syntactic constructions in Definition 15 are easily defined recursively on the DDD data structure, thus making it simple to specify and implement algorithms for these operations. The function APPLY(op, u, v) is used to combine two ordered, locally reduced DDDs rooted at u and v with a Boolean operator op, e.g., the negation and conjunction operations in Definition 15. APPLY is a generalization of the version used for BDDs [11] 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 quantifier-elimination method [29], 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 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 Definition 16 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 Milner's Scheduler and Fischer's Mutual Exclusion Protocol with a DDD-based model checker. We compare the runtimes with those obtained with the two tools, KRONOS [21,20,22] and UPPAAL [39,7].

5.1 Milner's Scheduler

Milner's scheduler [41] 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 clock y to ensure that a cycler passes the token on to the following cycler within the interval $[25, 200]$. We restrict the time a task can be executing by introducing a clock x_i that measures the execution time of each task t_i. The task t_i must terminate within $[80, 100]$ time units after it is started. The i^{th} cycler is described by two guarded commands and the task is modeled by a third guarded command:

$$\begin{array}{ll} c_i \wedge \neg t_i & \rightarrow y, x_i, t_i, c_i, h_i := 0, 0, \mathbf{true}, \mathbf{false}, \mathbf{true} \\ h_i \wedge y \geq 25 & \rightarrow c_{(i \bmod N)+1}, h_i := \mathbf{true}, \mathbf{false} \\ t_i \wedge x_i \geq 80 & \rightarrow t_i := \mathbf{false}\,. \end{array}$$

The initial state is given by

$$\phi_0 = c_1 \wedge \neg t_1 \wedge \neg h_1 \wedge \bigwedge_{i=2}^{N} \neg c_i \wedge \neg t_i \wedge \neg h_i.$$

The program invariant is given by

$$I = \bigwedge_{i=1}^{N} (h_i \Rightarrow y \leq 200) \wedge (t_i \Rightarrow x_i \leq 100)\,.$$

expressing that each cycler must pass on the token within 200 time units, and that each task must terminate 100 time units after it is started. Furthermore, the first guarded command is urgent, thus the urgency predicate is

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

We have computed the reachable state space $\mathbf{post}^*(\phi_0)$ for increasing number N of cyclers. The results are shown in Table 1(a) together with the runtimes obtained with KRONOS (version 2.2b) and UPPAAL (version 3.0.39) using the default options. This version of Milner's scheduler has exponentially many discrete states because a task can terminate independently of the other tasks.

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

(a) Milner's scheduler

N	KRONOS	UPPAAL	DDD
1	0.2	0.1	0.1
2	0.3	0.2	0.3
3	0.6	0.3	0.4
4	1.5	0.8	0.8
5	6.4	30.8	1.4
6	–	2986.1	4.1
7	–	–	18.0
8	–	–	179.6
9	–	–	–

(b) Fischer's protocol

Table 1
Experimental results. The CPU time in seconds for (a) building the reachable state space for Milner's scheduler, and (b) verifying mutual exclusion for Fischer's protocol. The results were obtained on a Pentium II PC with 128 MB of memory running Linux. A dash ('–') denotes that the analysis did not complete within an hour.

Thus, state-space exploration based on enumerating all discrete states as in the two other tools 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).

As for BDDs, the size of a DDD depends on the chosen variable ordering. In Milner's scheduler, experiments show that the Boolean variables should precede the clocks in the decision diagram. The Boolean variables are ordered as $t_1 \prec c_1 \prec h_1 \prec \cdots \prec t_N \prec c_N \prec h_N$. Pairs of clocks (x_i, x_j) are ordered reversed lexicographically using the ordering $z \prec y \prec x_1 \prec \cdots \prec x_N$. 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 [34] and conjoin each element $I_i = (h_i \Rightarrow y \leq 200) \wedge (t_i \Rightarrow x_i \leq 100)$ incrementally instead of building a DDD for I. This is possible because $\forall z''.(f \Rightarrow I_1 \wedge \cdots \wedge I_N)$ is equivalent to $\bigwedge_{i=1}^{N} \forall z''.(f \Rightarrow I_i)$.

5.2 Fischer's Mutual Exclusion Protocol

Fischer's mutual exclusion protocol [37] consists of N processes competing for a shared resource. Each process can be in one of four states modeled using two Boolean variables:

$$\begin{aligned} idle_i &= \neg b_i^1 \wedge \neg b_i^0 \\ rdy_i &= \neg b_i^1 \wedge b_i^0 \\ wait_i &= b_i^1 \wedge \neg b_i^0 \\ crit_i &= b_i^1 \wedge b_i^0 \end{aligned}$$

We use the variable s_i to represent the state of process i. We write $s_i = idle_i$ for the predicate $\neg b_i^1 \wedge \neg b_i^0$, and $s_i := idle_i$ for the assignment $b_i^1, b_i^0 :=$ **false**, **false**, etc. Furthermore, the processes use a shared variable id, which an integer in the range $[0; N]$, for controlling the access to the shared resource. Like the state variables, this variable can be encoding using $\lceil \log_2(N+1) \rceil$ Boolean variables. The timed guarded commands for a process are:

$$\begin{array}{ll} (s_i = idle_i \vee s_i = wait_i) \wedge id = 0 & \rightarrow s_i, x_i := rdy_i, 0 \\ s_i = rdy_i \wedge x \leq k & \rightarrow s_i, x_i, id := wait_i, 0, i \\ s_i = wait_i \wedge x > k \wedge id = i & \rightarrow s_i := crit_i \\ s_i = crit_i & \rightarrow s_i, id := idle_i, 0 \end{array}$$

The parameter k is a constant which determines how long a process waits until entering the critical state. We use $k = 10$ in the following. The initial state is given by

$$\phi_0 = (id = 0) \wedge \bigwedge_{i=1}^{N} (s_i = idle_i) \, .$$

The program invariant is given by

$$I = \bigwedge_{i=1}^{N} (s_i = rdy_i) \Rightarrow x \leq k \, .$$

The following property expresses that only one process is in the critical state:

$$M = \neg \bigvee_{i=1}^{N} \Big(s_i = crit_i \wedge \bigvee_{j \neq i} s_j = crit_j \Big)$$

Fischer's protocol guarantees mutual exclusion if and only if $\mathbf{pre}^*(\neg M) \wedge \phi_0$ is **false**. We have verified Fischer's protocol increasing number N of processes, and the results are shown in Table 1(b). Again the runtimes are faster for the symbolic DDD-based model checker.

6 Conclusion

Analyzing timed systems is extremely difficult. Very often, current timing verification tools cannot handle systems with a complexity that occur in practice. One reason for this is that current methods enumerate the discrete states of the system, and they are thus inherently limited by the number of states in the system.

We have shown how difference constraint expressions can be used to represent and verify concurrent timed systems in a fully symbolic manner. 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. The **any** operator makes is possible to add a timed guarded command for advancing time explicitly in a program, thus simplifying the definition of the semantics. Programs can be analyzed fully symbolically in a forward and backward manner using the **post** and **pre** operators which construct difference constraint expressions.

This result allows us to analyze timed systems without explicitly enumerating the discrete states of the system, thus removing a key limitation of current approaches. The complexity of performing timing analysis is reduced to deciding satisfiability of constraints the form $x - y \leq d$ combined with Boolean operators and existentially quantified.

We have shortly introduced a new data structure called DDDs for representing and deciding validity of such expressions. DDDs attempt to obtain the compactness of BDDs, but often fail in this respect. For some classes of timed systems, such as Milner's and Fischer's scheduling algorithms, DDDs work well because of the uniformity of these systems. For other classes of systems, such as asynchronous circuits, DDDs can only verify smaller instances (containing up to 10 clocks and 20 Boolean variables). Quantifier elimination is the core operation in real-time model checking, and a more efficient implementation of this operator on DDDs is the focus of current research.

A Proofs

A.1 Theorem 21

Let t be a timed guarded command of the form $\phi \rightarrow \vec{v} := \textbf{any}\, \vec{v'}.\phi'$. Using Definitions 9 and 12 we get:

$$
\begin{aligned}
Post_{\mathrm{d}}([\![\psi]\!]_z, t) &= \{s' : \exists s \in [\![\psi]\!]_z \wedge s \xrightarrow{t} s'\} \\
&= \{s[\vec{v} := \vec{r}] : s \in [\![\psi \wedge \phi_z]\!]_z \wedge s[\vec{v'} := \vec{r}] \in [\![\phi'_z]\!]_z \wedge \\
&\qquad\qquad s[\vec{v} := \vec{r}] \in [\![I_z]\!]_z\} \\
&= \{s[\vec{v} := \vec{r}] : s \in [\![\psi \wedge \phi_z]\!]_z \wedge s \in [\![\phi'_z[\vec{r}/\vec{v'}]]\!]_z \wedge \\
&\qquad\qquad s[\vec{v} := \vec{r}] \in [\![I_z]\!]_z\} \\
&= [\![(\psi \wedge \phi_z \wedge \phi'_z[\vec{r}/\vec{v'}])[\vec{v} := \textbf{any}\, \vec{v'}.\vec{v'} = \vec{r}] \wedge I_z]\!]_z \\
&= [\![(\psi \wedge \phi_z)[\vec{v} := \textbf{any}\, \vec{v'}.(\vec{v'} = \vec{r} \wedge \phi'_z[\vec{r}/\vec{v'}])] \wedge I_z]\!]_z \\
&= [\![(\psi \wedge \phi_z)[\vec{v} := \textbf{any}\, \vec{v'}.\phi'_z] \wedge I_z]\!]_z \\
&= [\![\textbf{post}_{\mathrm{d}}(\psi, t)]\!]_z
\end{aligned}
$$

A.2 Theorem 22

Immediate from the definitions of $Post_{\mathrm{d}}$ and $\mathbf{post}_{\mathrm{d}}$.

A.3 Theorem 24

From Definition 13 we have

$$Post_{\mathrm{t}}([\![\psi]\!]_z, \delta) = \{s' : s \in [\![\psi]\!]_z \wedge s \xrightarrow{\delta} s'\}\,,$$

where, by Definition 10, $s' = s[\vec{c} := \vec{c} + \delta]$, $\delta \geq 0$, and $\forall \delta'.0 \leq \delta' \leq \delta : s[\vec{c} := \vec{c} + \delta'] \models I$. That is:

$$Post_{\mathrm{t}}([\![\psi]\!]_z, \delta) = \{s' : s \in [\![\psi]\!]_z \wedge \delta \geq 0 \wedge \forall \delta'.0 \leq \delta' \leq \delta : s[\vec{c} := \vec{c} + \delta'] \models I\}\,.$$

We now introduce two new variables defined as $z' = z - \delta$ and $z'' = z - \delta'$. It is not difficult to see that with these definitions, $0 \leq \delta' < \delta$ is equivalent to $z' < z'' \leq z$. Furthermore, since

$$\begin{aligned} s[\vec{c} := \vec{c} + \delta'] \models I &\equiv s[\vec{c} := \vec{c} + \delta'] \in [\![I_z]\!]_z \\ &\equiv s \in [\![I_z[\vec{c} := \vec{c} - \delta']]\!]_z \\ &\equiv s \in [\![I_z[z := z + \delta']]\!]_z \\ &\equiv s \in [\![I_z[(z - \delta')/z]]\!]_z \\ &\equiv s \in [\![I_z[z''/z]]\!]_z \\ &\equiv s \in [\![I_{z''}]\!]_z\,, \end{aligned}$$

we can write $Post_{\mathrm{t}}([\![\psi]\!]_z, \delta)$ as:

$$\begin{aligned} Post_{\mathrm{t}}([\![\psi]\!]_z, \delta) &= \{s' : s \in [\![\psi]\!]_z \wedge \exists z'.\Big((\delta = z - z') \wedge (z - z' \geq 0) \wedge \\ &\qquad\qquad \forall z''.z' \leq z'' \leq z : s \in [\![I_{z''}]\!]_z\Big)\} \\ &= \{s' : s \in [\![\psi \wedge \exists z'.\Big((\delta = z - z') \wedge (z - z' \geq 0) \wedge \\ &\qquad\qquad \forall z''.(z' \leq z'' \leq z) \Rightarrow I_{z''}\Big)]\!]_z\} \end{aligned}$$

Using that $\{s[\vec{c} := \vec{c} + \delta] : s \in [\![\psi]\!]_z\}$ is equivalent to $[\![\psi[\vec{c} := \vec{c} + \delta]]\!]_z$, we obtain

$$Post_{\mathrm{t}}([\![\psi]\!]_z, \delta) = [\![\Big(\exists z'.(\psi \wedge (\delta = z - z') \wedge P_{\mathrm{post}})\Big)[\vec{c} := \vec{c} + \delta]]\!]_z\,.$$

Since $[\![\phi[\vec{c} := \vec{c} + \delta]]\!] = [\![\phi_z[z := z - \delta]]\!]_z = [\![\phi_z[(z + \delta)/z]]\!]_z$, it follows that

$$Post_{\mathrm{t}}([\![\psi]\!]_z, \delta) = [\![\Big(\exists z'.(\psi \wedge (\delta = z - z') \wedge P_{\mathrm{post}})\Big)[(z + \delta)/z]]\!]_z\,.$$

Using the definition of replacement, we get:

$$\begin{aligned} \mathit{Post}_{\mathrm{t}}([\![\psi]\!]_z, \delta) &= [\![(\psi \wedge P_{\mathrm{post}})[(z-\delta)/z'][(z+\delta)/z]]\!]_z \\ &= [\![(\psi \wedge P_{\mathrm{post}})[(z'+\delta)/z][z/z']]\!]_z \\ &= [\![\big(\exists z.(\psi \wedge (z - z' = \delta) \wedge P_{\mathrm{post}})\big)[z/z']]\!]_z \\ &= [\![\mathbf{post}_{\mathrm{t}}(\psi, \delta)]\!]_z \,. \end{aligned}$$

A.4 Theorem 25

The first equality in the theorem follows immediately from Definitions 13 and 23. The second equality holds because existential quantification distributes over disjunction:

$$\begin{aligned} \mathit{Post}_{\mathrm{t}}([\![\psi]\!]_z) &= \textstyle\bigcup_{\delta \in \mathbb{R}} \mathit{Post}_{\mathrm{t}}([\![\psi]\!]_z, \delta) \\ &= [\![\textstyle\bigvee_{\delta \in \mathbb{R}} \big(\exists z.(\psi \wedge (z - z' = \delta) \wedge P_{\mathrm{post}})\big)[z/z']]\!]_z \\ &= [\![\big(\exists z.(\psi \wedge \textstyle\bigvee_{\delta \in \mathbb{R}}(z - z' = \delta) \wedge P_{\mathrm{post}})\big)[z/z']]\!]_z \\ &= [\![\big(\exists z.(\psi \wedge P_{\mathrm{post}})\big)[z/z']]\!]_z \\ &= [\![\mathbf{post}_{\mathrm{t}}(\psi)]\!]_z \,. \end{aligned}$$

A.5 Theorem 27

Follows immediately from Definition 14 and Theorems 22 and 25.

A.6 Theorem 36

Using Definition 19 and Theorem 22 we first show that we can embed the guard and the program invariant in the expressions of the timed guarded commands in T:

$$\begin{aligned} \mathbf{post}_{\mathrm{d}}(\psi, \phi \to \vec{v} := \mathbf{any}\, \vec{v'}.\phi') &= (\psi \wedge \phi_z)[\vec{v} := \mathbf{any}\, \vec{v'}.\phi'_z] \wedge I_z \\ &= \exists \vec{v}.(\psi \wedge \phi'_z \wedge \phi_z)[\vec{v}/\vec{v'}\,] \wedge I_z \\ &= \exists \vec{v}.\big(\psi \wedge \phi'_z \wedge \phi_z \wedge I_z[\vec{v'}/\vec{v}]\big)[\vec{v}/\vec{v'}\,] \\ &= \mathbf{post}_{\mathrm{STGC}}\big(\psi, \vec{v} := \mathbf{any}\, \vec{v'}.(\phi' \wedge \phi \wedge I[\vec{v'}/\vec{v}])\big). \end{aligned}$$

Next, we show that we can advance time explicitly by adding a timed guarded command of the form $\mathbf{true} \to z := \mathbf{any}\, z'.P_{\mathrm{post}}$:

$$\begin{aligned} \mathbf{post}_{\mathrm{t}}(\psi) &= \exists z.(\psi \wedge P_{\mathrm{post}})[z/z'] \\ &= \psi[\mathbf{true} \to z := \mathbf{any}\, z'.P_{\mathrm{post}}] \\ &= \mathbf{post}_{\mathrm{STGC}}(\psi, \vec{z} := \mathbf{any}\, \vec{z'}.P_{\mathrm{post}}). \end{aligned}$$

References

[1] 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.

[2] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *IEEE Symposium on Foundations of Computer Science*, pages 164–169, 1989.

[3] F. Balarin. Approximate reachability analysis of timed automata. In *Proc. Real-Time Systems Symposium*, pages 52–61. IEEE, 1996.

[4] E. Balas. Disjunctive programming. *Annals of Discrete Mathematics*, 5:3–51, 1979.

[5] 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*, pages 88–100, April 1997.

[6] W. Belluomini and C. J. Myers. Verification of timed systems using POSETs. In *Proc. Computer Aided Verfication (CAV)*, pages 403–415, June 1998.

[7] 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.

[8] 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.

[9] Andreas Blass and Yuri Gurevich. Fixed-choice and independent-choice logics. Technical Report TR-369-98, University of Michigan, August 1998.

[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] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proc. Second GI Conference*, volume 33 of *LNCS*, pages 134–183. Springer-Verlag, 1975.

[17] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, Ohio, 1971.

[18] D. C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7:91–99, 1972.

[19] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1994.

[20] 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.

[21] 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.

[22] 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.

[23] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[24] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer-Verlag, 1989.

[25] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, M. Pnueli, and A. Rasse. Data structures for the verification of timed automata. In O. Maler, editor, *Int. Workshop on Hybrid and Real-Time Systems*, volume 1201 of *LNCS*, pages 346–360, Grenoble, France, 1997. Springer-Verlag.

[26] J. Ferrante and J. R. Geiser. An efficient decision procedure for the theory of rational order. *Theoretical Computer Science*, 4(2):227–233, 1977.

[27] Jeanne Ferrante and Charles Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM Journal of Computing*, 4(1):69–76, 1975.

[28] M. J. Fischer and M. O. Rabin. Super-exponential complexity of presburger arithmetic. In *Proceedings of the SIAM-AMS Symposium in Applied Mathematics*, volume 7, pages 27–41, 1974.

[29] J.B.J. Fourier. Second extrait. In G. Darboux, editor, *Oeuvres*, pages 325–328, Paris, 1890. Gauthiers-Villars.

[30] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter System (On formal undecidable theorems in Principia Mathematica and related systems). In *Monatshefte für Mathematik und Physik*, volume 38, pages 173–198, 1931.

[31] 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.

[32] David Hilbert and Paul Bernays. *Grundlagen der Mathematik*, volume 2. Springer-Verlag, 1939.

[33] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[34] Alan J. Hu and David L. Dill. Efficient verification with BDDs using implicitly conjoined invariants. In *Proc. Fifth International Conference on Computer Aided Verfication (CAV'93)*, volume 697 of *LNCS*, pages 3–14, 1993.

[35] N. K. Karmarkar. A new polynomial–time algorithm for linear programming. *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 302–311, 1984.

[36] Manolis Koubarakis. Complexity results for first-order theories of temporal constraints. In Jon Doyle, Erik Sandewall, and Pietro Torasso, editors, *Proc. of the Fourth Internatinal Conference on Principles of Knowledge Representation and Reasoning (KR'94)*, pages 379–390, San Francisco, California, 1994. Morgan Kaufmann.

[37] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[38] 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.

[39] 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.

[40] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36(5):450–462, 1993. Special issue on computational quantifier elimination.

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

[42] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Proceedings 13th International Conference on Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125, Madrid, Spain, September 1999.

[43] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Fully symbolic model checking of timed systems using difference decision diagrams. In *Proceedings First International Workshop on Symbolic Model Checking*, volume 23-2 of *Electronic Notes in Theoretical Computer Science*, pages 89–108, Trento, Italy, July 1999.

[44] D. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. *Journal of Computer and System Sciences*, 16:323–332, 1978.

[45] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[46] M. Presburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zählen, in welchem die addition als einzige operation hervortritt. *Comptes-Rendus du I Congres de Mathematiciens des pays Slaves*, pages 92–101, 1929.

[47] T. G. Rokicki. *Representing and Modeling Digital Circuits*. PhD thesis, Stanford University, 1993.

[48] 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.

[49] R. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 4(24):529–543, October 1977.

[50] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[51] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkeley, CA, 2nd edition, 1951.

[52] E. Verlind, G. de Jong, and B. Lin. Efficient partial enumeration for timing analysis of asynchronous systems. In *Proc. ACM/IEEE Design Automation Conference*, pages 55–58, June 1996.

[53] H. Wong-Toi and D.L. Dill. Approximations for verifying timing properties. In *Theories and Experiences for Real-Time Systems Development*, chapter 7, pages 177–204. World Scientific Publishing, 1994.

[54] S. Yovine. Kronos: A verification tool for real-time systems. *Springer Int. Journal of Software Tools for Technology Transfer*, 1(1/2), October 1997.

[55] S. Yovine. Model checking timed automata. In *Embedded Systems*, volume 1494 of *LNCS*, pages 114–152. Springer-Verlag, October 1998.