Timed Verification of Asynchronous Circuits*

Jesper Møller, Henrik Hulgaard, and Henrik Reif Andersen

The IT University of Copenhagen
Department of Innovation
{jm,henrik,hra}@it.edu

Abstract. We describe a methodology for analyzing timed systems symbolically. Given a formula representing a set of timed 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 constraints of the form $x - y \leq d$ which can be combined with Boolean operators and existentially quantified. 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 describe how to model asynchronous circuits using timed guarded commands and provide examples that demonstrate the potential of the symbolic analysis.

1 Introduction

Model checking [17] 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) [13]. 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 [15].

This chapter shows how to extend symbolic model checking to handle models that contain variables that range over non-countable domains. Specifically, we show how to analyze timed systems where timers are modeled using continuous variables and the behavior of a system is specified using constraints on these variables. The analysis of timed systems plays an important role in several areas:

- *Real-time systems* are a class of systems where correct behavior depends on the ability to react to external events in a predictable and timely manner. Real-time systems are found, for example, in transportation systems, aerospace, robotics, communication systems, defense systems, and industrial process control. Because real-time systems are used in time critical applications, failure to meet the required deadlines may have serious consequences as damage to property or humans. Thus, formal verification is necessary to ensure the correctness of a real-time system.

* Financially supported by a grant from the Danish Technical Research Council.

– *Protocols* may rely on timing properties. Some examples are Fischer's protocol for establishing mutual exclusion, Milner's protocol for scheduling tasks, and the FDDI communication protocol used in local area networks. Analyzing the correctness of such protocols requires formal arguments involving timing properties.
– *Asynchronous circuits* consist of components that synchronize by handshake signals rather than using a global clock. Although asynchronous circuits can be designed so that they work correctly for any delay of the components (so-called speed-independent and delay-insensitive circuits), to obtain high performance and reduce the amount of circuitry, the designers need to exploit knowledge of the actual delays of the gates. As a result, the different parts of the circuit make (timing) assumptions about their inputs and work correctly only as long as these assumptions are satisfied. For circuits of non-trivial size, the circuit designer needs automated analysis methods to ensure correct behavior of the circuit, for example that the circuit is hazard-free.

We propose a simple notation called *timed guarded commands* for modeling a timed system. Timed guarded commands are similar to Dijkstra's guarded commands [22] 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 [2] and timed Petri nets [9] are easily encoded using timed guarded commands.

Given a system defined by 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) [30], can be verified symbolically.

1.1 Current Analysis Approaches

The basic verification problem is to determine whether a given state s is reachable. The standard approach for solving this problem is to construct the set of reachable states R and then determine whether $s \in R$. To solve this 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 operations (resetting clocks, advancing the time of clocks, etc.) on the representation to compute the reachable part of the state space?
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, or the values on the wires of a circuit) and v is an assignment of values to the clocks in the system.

$Q \leftarrow \{(s_0, V_0)\}$
$R[s_0] \leftarrow V_0$
$R[s] \leftarrow \emptyset$ for all $s \neq s_0$
while $Q \neq \emptyset$ **do**
  Remove some (s, V) from Q
  $\{(s_1, V_1), \ldots, (s_k, V_k)\} \leftarrow$ **post**(s, V)
  for $i \leftarrow 1$ **to** k **do**
$(*)$ **if** $V_i \not\subseteq R[s_i]$ **then**
      Add (s_i, V_i) to Q
      $R[s_i] \leftarrow R[s_i] \cup V_i$
	$R \leftarrow \phi_0$
	repeat
	  $R' \leftarrow R$
	  $R \leftarrow R \vee$ **post**(R)
	until $R = R'$

(a)	(b)

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

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 and the associated group of clock valuations V. The reachable states space R for a timed system can be determined by the generic algorithm in Fig. 1(a) where we view R as a mapping from discrete states s to their associated group of clock valuations V. The operator **post** 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., [9,35,43, 50]) are based on representing clock assignments using a set of difference bound matrices (DBMs) [23]. 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 operator **post** 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 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 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.
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 [38] 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 [6] show a significant improvement in memory consumption, even though the discrete states are still enumerated explicitly.

In this chapter we address all three problems by constructing the set of reachable states R in a fully symbolic manner, 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 fixpoint iteration shown in Fig. 1(b).

A core operation when performing symbolic model checking is to determine the formula $\mathbf{post}(\phi)$ representing 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 [23] and also when solving systems of difference constraints [21].

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:

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

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

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

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).

1.3 Related Work

Model checking of timed systems (timed automata in particular; see [51] 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, 12, 14, 16]. 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) [23] for representing the timing information [9,35,43,50]. One can see the use of DBMs as expanding difference constraint expressions 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 [8,44,48] or by using approximate methods [4,7,49]. 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.

Henzinger et al. [30] 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.

1.4 Overview

This chapter is organized as follows: In Sect. 2 we introduce a simple model of timed systems called timed guarded commands. Section 3 shows how to analyze

timed guarded commands—for example, how to compute the set of reachable states symbolically, and how to formally verify properties (e.g., check for absence of hazards in a circuit). Section 4 discusses various data structures and algorithms for implementing tools that can perform these symbolic analyses. In Sect. 5 we look at some examples of timed systems and analyze them using a data structure called difference decision diagrams. Finally, Sect. 6 is a summary of the chapter.

2 Timed Guarded Commands

We present a simple notation called timed guarded commands [39] for modeling systems with time. This notation provides the basis for modeling asynchronous circuits, and is sufficient for encoding popular notations for systems with time such as timed automata [2] and timed Petri nets [9]. First we define the syntax and semantics of timed guarded commands, and then we describe how to model asynchronous circuits in this notation. Section 3 describes how to analyze a model of an asynchronous circuit specified using 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 (Variables). *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 (Expressions). *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. Other Boolean operators such as $\Rightarrow$ (implication), $\Leftrightarrow$ (bi-implication), and $\oplus$ (exclusive or) are defined the standard way.

Example 1. Let $x, y, z \in \mathcal{C}$ be clocks, and $a, b, c \in \mathcal{B}$ be Boolean variables. Then a valid expression over these variables is:

$$\phi_1 = \big(a \wedge b \wedge (x - y \neq 0) \wedge (x - z \leq 5)\big) \vee \big(c \Rightarrow (x = 0)\big) \,.$$

The expression

$$\phi_2 = (x + y = 0) \vee (x - 2y < 3)\,.$$

is *not* a well-formed expressions because $x + y = 0$ is not a valid atomic expression, and neither is $x - 2y < 3$.

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

Definition 4 (any-operator). *Let $\boldsymbol{v}, \boldsymbol{v'} \in \mathcal{V}^n$ be n-dimensional vectors of Boolean variables and clocks, and let $\phi' \in \Phi$ be an expression. Then the* **any***-operator is written as:*

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

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 ($i = 1, \ldots, n$) any value v'_i, $v'_i \neq v_i$, 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:

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

Example 2. The assignment

$$(x, y) := \textbf{any}\ (x', y').(0 \leq x' - x \leq 10) \wedge (0 \leq y' - x \leq 1)$$

has the following meaning: increment x by some value in the range $[0, 10]$, and, simultaneously, set y to some value in the range $[x, x + 1]$. For example, if $x = y = 1$ then after the assignment we have $x \in [1, 11]$ and $y \in [1, 2]$. Notice how the primed variables x' and y' in the two expressions are used to refer to the new values for x and y, respectively.

[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 identical then the two values bound to v' and v'' are also identical. The fixed-choice **any** operator is also known as Hilbert's ϵ-operator [31]. The independent-choice **any** operator we use is identical to Blass and Gurevich's δ-operator [10].

The ability to express nondeterministic assignments using the **any**-operator is used in Sect. 3 where we describe how to analyze timed guarded commands. In this section we only consider deterministic assignments.

Definition 5 (Commands). *Let $\boldsymbol{v}, \boldsymbol{v'} \in \mathcal{V}^n$ 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*

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

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, U), 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 $B \cup C$, $I \in \Phi$ is the program invariant, and $U \in \Phi$ is the urgency predicate.*

In the following section we give precise semantics of a TGC program and define what it means to run a TGC program. Informally speaking, to run a program we start in some initial state and continuously either execute an enabled command or advance time by increasing the values of all clocks by some amount. The program invariant is an expression which must always be fulfilled when running a program. The urgency predicate is an expression which specifies when it is illegal to advance time. We can use this predicate to ensure that certain 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 urgent commands are enabled. As we shall see in Sect. 2.5, the program invariant and the urgency predicate are used to ensure progress of a TGC program. See [45] for a more thorough treatment of urgency in timed systems.

Example 3. An example of a TGC program is $P = (\{b\}, \{x, y\}, \{t_1, t_2\}, I, U)$, where:

$$\begin{aligned} t_1 &: b \wedge (1 \leq x \leq 3) \rightarrow b \quad := \textbf{false} \\ t_2 &: b \wedge (7 \leq x \leq 9) \rightarrow b, y := \textbf{false}, 0 \, . \end{aligned}$$

The program invariant is given by the expression

$$I = \big(b \Rightarrow (x \leq 9)\big) \wedge \big(\neg b \Rightarrow (x \neq 5)\big) \, .$$

There are no urgent commands, so the urgency predicate is simply:

$$U = \textbf{false} \, .$$

2.2 Semantics

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.

Definition 7 (States). *A state of a* TGC *program* $P = (B, C, T, I, U)$ *is an interpretation (i.e., a value assignment) of the Boolean variables and clocks. For each variable* $v \in B \cup C$, $s(v) \in \mathbb{B} \cup \mathbb{Q}$ *denotes the interpretation of* 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.

We also use the notation $s(v_1, \ldots, v_n) = (r_1, \ldots, r_n)$, where $r_i \in \mathbb{B} \cup \mathbb{C}$ for $i = 1, \ldots, n$.

Example 4. Let us consider the TGC program from Example 3. The set of states $\mathcal{S}$ of this program are triples of the form (b, x, y), where b is either **true** or **false**, and x and y are real numbers. Consider the state $s(b, x, y) = (\mathbf{true}, 0, 2)$, and consider the expressions $\phi_1 = b \wedge x < 9$, and $\phi_2 = \neg b \vee x = y$. Clearly, $s \models \phi_1$, but $s \not\models \phi_2$.

Definition 8 (State update). *Let* $\boldsymbol{v} \in \mathcal{V}^n$ *be an* n*-dimensional vector of variables, and let* $\boldsymbol{r} \in (\mathbb{B} \cup \mathbb{Q})^n$ *be an* n*-dimensional vector of values. Then the state* $s' = s[\boldsymbol{v} := \boldsymbol{r}]$ *is equivalent to* s *except that* $s'(v_i) = r_i$ *for* $i = 1, \ldots, n$.

Example 5. Consider again the state s where $s(b, x, y) = (\mathbf{true}, 0, 2)$, and assume that $s' = s[(x, z) := (2, 3)]$. Then $s'(b, x, y, z) = (\mathbf{true}, 2, 2, 3)$.

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

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

Definition 9 says that if $\phi \to \boldsymbol{v} := \mathbf{any}\ \boldsymbol{v'}.\phi'$ is a command t in a TGC program P, then the transition system for P has a discrete transition from s to $s' = s[\boldsymbol{v} := \boldsymbol{r}]$ labeled t if the following conditions hold:

- the state s satisfies the guard ϕ,
- the state s updated with $\boldsymbol{v'} := \boldsymbol{r}$ satisfies the expression ϕ', and
- the state s' satisfies the state invariant I.

Example 6. Consider again the TGC program in Example 3, and let s be a state where $s(b, x, y) = (\mathbf{true}, 2, 2)$. Then, from Definition 9, the transition system for P contains a discrete transition $\xrightarrow{t_1}$ from s to s', where $s'(b, x, y) = (\mathbf{false}, 2, 2)$.

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

$$\frac{\delta \geq 0 \quad s[\boldsymbol{c} := \boldsymbol{c} + \delta] \models I \quad \forall \delta'. 0 \leq \delta' < \delta : s[\boldsymbol{c} := \boldsymbol{c} + \delta'] \models (I \wedge \neg U)}{s \xrightarrow{\delta} s[\boldsymbol{c} := \boldsymbol{c} + \delta]},$$

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

Definition 10 says when it is legal to advance time by some amount δ:

- the delay δ must be nonnegative,
- the state invariant I must hold if we advance time by δ', for any $\delta' \in [0, \delta]$, and
- the negated urgency predicate $\neg U$ must hold if we advance time by δ', for any $\delta' \in [0, \delta[$.

That is, we can advance time by δ if the state invariant holds continuously for all intermediate points in time (i.e., for $\delta' \in [0, \delta]$), and if the negation of the urgency predicate holds continuously for all intermediate points in time, except the end-point δ (i.e., for $\delta' \in [0, \delta[$).

Example 7. Consider again the TGC program in Example 3, and let s be a state where $s(b, x, y) = (\mathbf{true}, 2, 2)$. Then, from Definition 10, the transition system for P contains infinitely many timed transitions $\xrightarrow{\delta}$, where $\delta \in [0, 7]$, from s to s' with $s'(b, x, y) = (\mathbf{false}, 2 + \delta, 2 + \delta)$. However, there are no timed transitions $\xrightarrow{\delta}$ from s with $\delta > 7$, as this would violate the state invariant.

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

Given some initial state s_0, we can execute a program from s_0 by repeatedly either executing a command $t \in T$ if its guard is true (a discrete transition) or letting 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 δ. This is made precise in the following definition:

Definition 12 (Execution). *Let $P = (B, C, T, I, U)$ be a TGC program, and let $(\mathcal{S}, \rightarrow)$ be the corresponding transition system. An execution of P from the initial state s_0 is a (possibly) infinite sequence of state transitions*

$$s_0 \xrightarrow{\tau_0} s_1 \xrightarrow{\tau_1} s_2 \xrightarrow{\tau_2} \cdots$$

where $s_i \in \mathcal{S}$ and $s_i \xrightarrow{\tau_i} s_{i+1}$ is a discrete or timed transition of $(\mathcal{S}, \rightarrow)$.

Example 8. An example of an execution of the TGC program P in Example 3, with the initial state $s_0(b, x, y) = (\mathbf{true}, 0, 0)$, is:

$$(\mathbf{true}, 0, 0) \xrightarrow{2} (\mathbf{true}, 2, 2) \xrightarrow{6} (\mathbf{true}, 8, 8) \xrightarrow{t_2} (\mathbf{false}, 8, 0)\,.$$

2.3 Reachability

Given a transition system $(\mathcal{S}, \rightarrow)$ for a TGC program $P = (B, C, T, I, U)$ 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 13 (Discrete successor). *Let $(\mathcal{S}, \rightarrow)$ be the transition system for a* TGC *program $P = (B, C, T, I, U)$, 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:

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

Example 9. Let P be the TGC program from Example 3, and consider the states s_1 and s_2 where $s_1(b, x, y) = (\mathbf{true}, 0, 0)$ and $s_2(b, x, y) = (\mathbf{true}, 2, 2)$. Then

$$\begin{array}{l} Post_{\mathrm{d}}(\{s_1\}) = Post_{\mathrm{d}}(\{s_1\}, t_1) \cup Post_{\mathrm{d}}(\{s_1\}, t_2) = \emptyset \\ Post_{\mathrm{d}}(\{s_2\}) = Post_{\mathrm{d}}(\{s_2\}, t_1) \cup Post_{\mathrm{d}}(\{s_2\}, t_2) = \{(\mathbf{false}, 2, 2)\} \end{array}$$

Definition 14 (Timed successor). *Let $(\mathcal{S}, \rightarrow)$ be the transition system for a* TGC *program $P = (B, C, T, I, U)$, 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:

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

Example 10. Let P be the TGC program from Example 3, and consider the states s_1 and s_2 where $s_1(b, x, y) = (\mathbf{true}, 0, 0)$ and $s_2(b, x, y) = (\mathbf{true}, 2, 2)$. Then

$$\begin{array}{l} Post_{\mathrm{t}}(\{s_1\}) = \bigcup_{\delta \in \mathbb{R}} Post_{\mathrm{d}}(\{s_1\}, \delta) = \{(\mathbf{true}, \delta, \delta) \mid 0 \leq \delta \leq 9\} \\ Post_{\mathrm{t}}(\{s_2\}) = \bigcup_{\delta \in \mathbb{R}} Post_{\mathrm{d}}(\{s_2\}, \delta) = \{(\mathbf{true}, \delta, \delta) \mid 2 \leq \delta \leq 9\} \end{array}$$

Definition 15 (Reachable states). *Let $(\mathcal{S}, \rightarrow)$ be the transition system for a* TGC *program $P = (B, C, T, I, U)$, 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 is can be determined by computing a sequence of approximations

$$f(\emptyset), f(f(\emptyset)), \ldots$$

until a fixpoint is reached, that is, until $f^i(\emptyset) \equiv f^{i+1}(\emptyset)$, for some i. See also Fig. 1(b) in the introduction which shows an algorithm that computes the least fixpoint of ϕ_0. It is well known that there exists (contrived) timed systems where the computation of the fixpoint 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.

2.4 Modeling Timed Automata

TGC 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 transitions has a set of clocks $\{c_1, \ldots, c_n\}$ to be reset when the timed transition is fired:

$$l \xrightarrow{a,\ g,\ \{c_1,\ldots,c_n\}} l'\,.$$

A timed automaton can be encoded as a TGC program. Each location is encoded as a Boolean variable.[2] 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:

$$l \wedge e_a \wedge g \rightarrow (l, l', c_1, \ldots, c_n) := (\mathbf{false}, \mathbf{true}, 0, \ldots, 0)\,.$$

[2] This is sometimes referred to as a "one-hot" encoding of the locations. In practice, a logarithmic encoding may be more efficient.

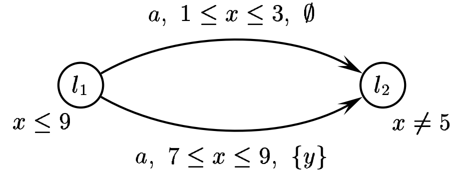

Fig. 2. The timed automaton in Example 11.

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 **false** to the source location l and **true** to the destination location l' of the timed transition and resets the relevant clocks.

Example 11. Figure 2 shows an automaton over the clocks $\{x, y\}$ with two locations and two timed transitions. Encoding this automaton as a TGC program yields the program P from Example 3 when ignoring the event a and encoding the two locations l_1 and l_2 logarithmically using a Boolean variable b.

2.5 Modeling Asynchronous Circuits

In this section we use timed guarded commands to define a simple gate-level model of an asynchronous circuit, inspired by the approach taken in [37].

Definition 16 (Gate). *A gate G is a six-tuple $(\boldsymbol{i}, o, g_\uparrow, g_\downarrow, [d_\uparrow, D_\uparrow], [d_\downarrow, D_\downarrow])$ where:*

- *$\boldsymbol{i} \in \mathcal{B}^n$ is a n-dimensional vector of inputs,*
- *$o \in \mathcal{B}$ is the output,*
- *$g_\uparrow : \mathcal{B}^n \to \mathcal{B}$ is an up-guard that specifies when the output o should become true,*
- *$g_\downarrow : \mathcal{B}^n \to \mathcal{B}$ is a down-guard that specifies when the output o should become false,*
- *$d_\uparrow$ and $D_\uparrow$ denote the minimum and maximum delays from when the up-guard $g_\uparrow$ becomes true to the output o becomes true.*
- *$d_\downarrow$ and $D_\downarrow$ denote the minimum and maximum delays from when the down-guard $g_\downarrow$ becomes true to the output o becomes false.*

Example 12 (AND-gate). An AND-gate with inputs inputs a, b has $g_\uparrow = a \wedge b$, and $g_\downarrow = \neg(a \wedge b)$.

Example 13 (Muller C-element). A Muller C-element with inputs a, b has $g_\uparrow = a \wedge b$, and $g_\downarrow = \neg a \wedge \neg b$.

Example 14 (Transistor). A transistor with gate i_1 and source i_2 has $g_\uparrow = i_1 \wedge i_2$, and $g_\downarrow = i_1 \wedge \neg i_2$.

To model the timing behavior of a gate, we introduce a clock x for measuring the time elapsed since the up-guard or down-guard has become true, and a Boolean variable u for modeling whether the gate is unstable: A gate is unstable when the up-guard is true, and the output has not yet changed to true, or, similarly, when the down-guard is true, and the output has not yet changed to false. With these definitions, we can model a gate as a TGC program as follows:

Definition 17 (Gate TGC program). *Let $(\boldsymbol{i}, o, g_\uparrow, g_\downarrow, [d_\uparrow, D_\uparrow], [d_\downarrow, D_\downarrow])$ be a gate with inputs $\boldsymbol{i} = (i_1, \ldots, i_n)$. Then the TGC program $P = (B, C, T, I, U)$ for G is defined as follows: $B = \{o, u, i_1, \ldots, i_n\}$, $C = \{x\}$, $T = \{t_1, t_2\}$ where:*

$$\begin{aligned} &t_1 : (\neg o \wedge \neg u \wedge g_\uparrow) \vee (o \wedge \neg u \wedge g_\downarrow) && \rightarrow u, x := \mathbf{true}, 0 \\ &t_2 : (\neg o \wedge u \wedge x \geq d_\uparrow) \vee (o \wedge u \wedge x \geq d_\downarrow) && \rightarrow u, o := \mathbf{false}, \neg o \end{aligned}$$

The program invariant is given by:

$$I = \big((o \wedge u) \Rightarrow x \leq D_\downarrow\big) \wedge \big((\neg o \wedge u) \Rightarrow x \leq D_\uparrow\big) .$$

Command t_1 is urgent, thus the urgency predicate is given by:

$$U = (\neg o \wedge \neg u \wedge g_\uparrow) \vee (o \wedge \neg u \wedge g_\downarrow) .$$

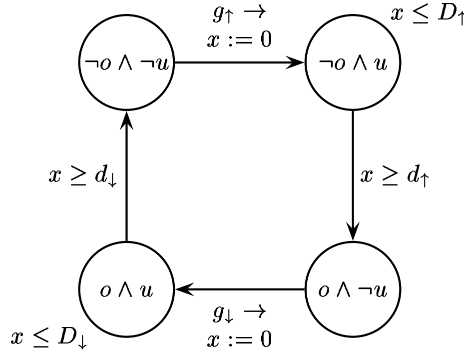

Fig. 3. Timed automaton for a gate as specified in Definition 17.

Let us explain the TGC program for a gate in a little more detail (see also Fig. 3 which shows a timed automaton for the program). Command t_1 models

that if the output is stable false (low) and the up-guard becomes true, then the output becomes unstable false (rising) and the clock x is reset. Similarly, if the output is stable true (high) and the down-guard becomes true, then the output becomes unstable true (falling). The urgency predicate ensures that t_1 is executed as soon as it becomes enabled (i.e., time cannot advance in our model if t_1 is enabled). The output then remains unstable false (rising) until $x \geq d_{\uparrow}$, and unstable true (falling) until $x \geq d_{\downarrow}$.

Command t_2 models that if the output is unstable false and $x \geq d_{\uparrow}$, then the output can (but does not have to) change to stable true. And similarly, if the output is unstable true and $x \geq d_{\downarrow}$, then the output can change to stable false. The state invariant ensures that the output must change to stable true before $D_{\uparrow}$, and, equally, must change to stable false before $D_{\downarrow}$.

Definition 17 specifies how to model a single gate as a TGC program. In the following definition we define how to model a circuit of N gates as a TGC program.

Definition 18 (Circuit TGC program). *Let $G_1, \ldots, G_N$ be the gates of a circuit, and let P_j for $j = 1, \ldots, N$ be the* TGC *program for G_j, where each output o_k of a gate G_k which is used as an input i_l for gate G_l is modeled using the same Boolean variable in P_k and P_l. Furthermore, the clocks of the* TGC *programs P_j must be distinct. Then the* TGC *program for the circuit is $P = (B, C, T, I, U)$, where $B = \bigcup_{j=1}^{N} B_j$, $C = \bigcup_{j=1}^{N} C_j$, $T = \bigcup_{j=1}^{N} T_j$, $I = \bigwedge_{j=1}^{N} I_j$, and $U = \bigvee_{j=1}^{N} U_j$.*

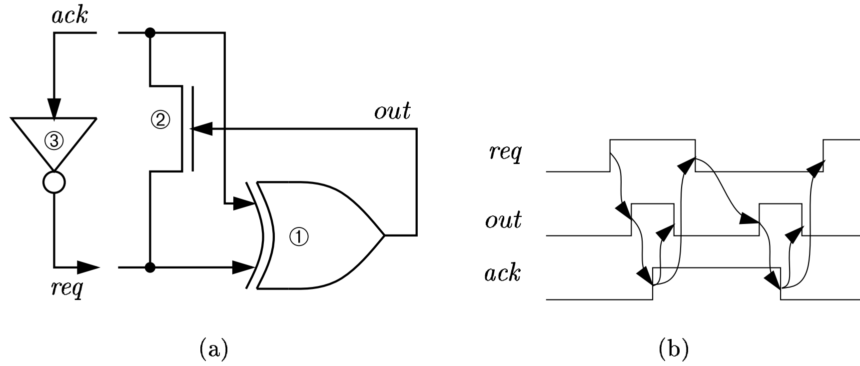

Fig. 4. (a) A pulse generator. (b) Timing diagram.

Example 15 (Pulse generator). Consider the asynchronous circuit depicted in Fig. 4(a). The circuit consists of an exclusive-or gate (labeled ①) and a transistor

(labeled ②). A pulse is generated on output *out* whenever the environment (here modeled with the inverter labeled ③) changes the request signal. Figure 4(b) is a timing diagram illustrating the behavior. The up- and down-guards for the three gates are:

$$\begin{aligned}
g_{out\uparrow} &= (ack \oplus req)\\
g_{out\downarrow} &= \neg(ack \oplus req)\\
g_{ack\uparrow} &= (out \wedge req)\\
g_{ack\downarrow} &= (out \wedge \neg req)\\
g_{req\uparrow} &= \neg ack\\
g_{req\downarrow} &= ack
\end{aligned}$$

Using Definition 18, the TGC program for the circuit is $P = (B, C, T, I, U)$, where the variables are $B = \{out, ack, req, out_u, ack_u, req_u\}$ (the variables out_u, ack_u, and req_u are used to model unstability of the gates) and $C = \{out_x, ack_x, req_x\}$. We assume that the delay intervals for the up- and down-guards are the same for each gate; that is, $[d_{out\uparrow}, D_{out\uparrow}] = [d_{out\downarrow}, D_{out\downarrow}] = [d_{out}, D_{out}]$, and similarly for *ack* and *req*. This gives us the following set of commands T (where we have simplified the guards):

$$\begin{aligned}
t_1 &: \neg out_u \wedge (out \oplus ack \oplus req) && \rightarrow out_u, out_x := \mathbf{true}, 0\\
t_2 &: out_u \wedge x \geq d_{out} && \rightarrow out_u, out := \mathbf{false}, \neg out\\
t_3 &: \neg ack_u \wedge out \wedge (ack \oplus req) && \rightarrow ack_u, ack_x := \mathbf{true}, 0\\
t_4 &: ack_u \wedge x \geq d_{ack} && \rightarrow ack_u, ack := \mathbf{false}, \neg ack\\
t_5 &: \neg req_u \wedge (ack \Leftrightarrow req) && \rightarrow req_u, req_x := \mathbf{true}, 0\\
t_6 &: req_u \wedge x \geq d_{req} && \rightarrow req_u, req := \mathbf{false}, \neg req
\end{aligned}$$

The program invariant is:

$$\begin{aligned}
I = {} & (out_u \Rightarrow out_x \leq D_{out})\\
& \wedge (ack_u \Rightarrow ack_x \leq D_{ack})\\
& \wedge (req_u \Rightarrow req_x \leq D_{req}) \,.
\end{aligned}$$

Commands t_1, t_3, t_5 are urgent, thus the urgency predicate becomes:

$$\begin{aligned}
U = {} & (\neg out_u \wedge (out \oplus ack \oplus req))\\
& \vee (\neg ack_u \wedge out \wedge (ack \oplus req))\\
& \vee (\neg req_u \wedge (ack \Leftrightarrow req)) \,.
\end{aligned}$$

In Sect. 3 we describe how to analyze the TGC program for a circuit, for example how to verify that the circuit is hazard-free:

Definition 19 (Hazard). *A gate G_i has a hazard if the up-guard becomes false when the output is unstable false, or the down-guard becomes false when the output is unstable true; that is, the hazard predicate for a gate is*

$$H_i = (\neg o \wedge u \wedge \neg g_{\uparrow}) \vee (o \wedge u \wedge \neg g_{\downarrow}) \,.$$

A circuit of N gates $G_1, \ldots, G_N$ has a hazard if one of the gates has a hazard; that is, the hazard predicate for a circuit is

$$H = \bigvee_{i=1}^{N} H_i \,,$$

where H_i is the hazard predicate for gate G_i.

The circuit is hazard-free if and only if $\neg H$ is an invariant for the TGC program for the circuit.

Example 16. Consider again the circuit in Fig. 4. The circuit has a hazard if the environment changes too fast. For example, consider the state where *req* and *out* are high and *ack* goes up. If the exclusive-or gate is slow, the environment may lower *req before* the output of the exclusive-or gate (*out*) has fallen and thus disabling the exclusive-or gate before it has lowered *out*.

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, for example that a circuit is hazard-free. More specifically, 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 20 (Difference Constraint Expressions). *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 21. *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.*[3]
2. *Replace each constraint of the form $x \sim d$ in ϕ by the difference constraint $x - z \sim d$.*

[3] 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. See also footnote 4 on page 27.

3. *Replace each difference constraint of the form $x - y \sim d$ in ϕ by a difference constraint with $\leq$ using the equivalences:*

$$\begin{aligned}
x - y > d &\equiv \neg(x - y \leq d)\,, \\
x - y \geq d &\equiv y - x \leq -d\,, \\
x - y < d &\equiv \neg(y - x \leq d)\,, \\
x - y = d &\equiv (x - y \leq d) \wedge (y - x \leq -d)\,, \\
x - y \neq d &\equiv \neg(x - y \leq d) \vee \neg(y - x \leq -d)\,.
\end{aligned}$$

Example 17. The expression $\phi = x \geq 0 \wedge y > 0$ corresponds to the difference constraint expression $\phi_z = z - x \leq 0 \wedge \neg(y - z \leq 0)$.

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 1. *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 22 (Replacement). *Replacement of a vector $\boldsymbol{v'} \in \mathcal{V}^n$ of variables by another vector $\boldsymbol{v} \in \mathcal{V}^n$ of variables plus a vector $\boldsymbol{d} \in \mathbb{Q}^n$ of constants, where $v_i \neq v'_i$ for each $i = 1, \ldots, n$, in an expression ψ is defined as follows:*

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

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

Definition 23 (Assignment). *Assignment of a vector $\boldsymbol{v} \in \mathcal{V}^n$ of variables to a vector $\boldsymbol{v'} \in \mathcal{V}^n$ of variables such that the expression ψ' holds is defined as follows:*

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

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:

$$[\![\phi[\boldsymbol{c} := \boldsymbol{c} + \delta]]\!] = [\![\phi_z[z := z - \delta]]\!]_z\,.$$

Furthermore, as we shall show 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 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 24 ($\mathbf{post}_{\mathrm{d}}$-operator). *Let ψ be a difference constraint expression, and let $P = (B, C, T, I, U)$ be a* TGC *program. Then the $\mathbf{post}_{\mathrm{d}}$-operator for (forward) execution of the command $\phi \to \boldsymbol{v} := \mathbf{any}\, \boldsymbol{v}'.\phi'$ is defined as*

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

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.

The following theorems state that the $\mathbf{post}_{\mathrm{d}}$ operators in Definition 24 correctly construct the set of discrete successors as defined by $Post_{\mathrm{d}}$ in Definition 13.

Theorem 1 (Correctness of $\mathbf{post}_{\mathrm{d}}(\psi, t)$). *Let ψ be a difference constraint expression, and let $P = (B, C, T, I, U)$ be a* TGC *program. Then $Post_{\mathrm{d}}([\![\psi]\!]_z, t) = [\![\mathbf{post}_{\mathrm{d}}(\psi, t)]\!]_z$ for any command $t \in T$.*

Proof. Let t be a timed guarded command of the form $\phi \to \boldsymbol{v} := \mathbf{any}\, \boldsymbol{v}'.\phi'$. Using Definitions 9 and 13 we get:

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

□

Theorem 2 (Correctness of $\mathbf{post}_{\mathrm{d}}(\psi)$). *Let ψ be a difference constraint expression, and let $P = (B, C, T, I, U)$ be a* TGC *program. Then $Post_{\mathrm{d}}([\![\psi]\!]_z) = [\![\mathbf{post}_{\mathrm{d}}(\psi)]\!]_z$.*

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

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. Also, the urgency predicate must hold at all intermediate points in time except in z'.

Definition 25 ($\mathbf{post}_\mathrm{t}$-operator). *Let ψ be a difference constraint expression, and let $P = (B, C, T, I, U)$ be a* TGC *program. Then the $\mathbf{post}_\mathrm{t}$-operator for advancing time by δ in all states $[\![\psi]\!]_z$ is defined as*

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

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

$$P_\mathrm{post} = (z' \leq z) \wedge I_{z'} \wedge \forall z''.\big((z' < z'' \leq z) \Rightarrow (I_{z''} \wedge \neg U_{z''})\big) .$$

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

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

The following theorems state that the $\mathbf{post}_\mathrm{t}$ operators in Definition 25 correctly construct the set of timed successors as defined by $Post_\mathrm{t}$ in Definition 14.

Theorem 3 (Correctness of $\mathbf{post}_\mathrm{t}(\psi, \delta)$). *Let ψ be a difference constraint expression, and let $P = (B, C, T, I, U)$ be a* TGC *program. Then $Post_\mathrm{t}([\![\psi]\!]_z, \delta) = [\![\mathbf{post}_\mathrm{t}(\psi, \delta)]\!]_z$ for any delay $\delta \in \mathbb{R}$.*

Proof. From Definition 14 we have

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

where, by Definition 10, $s' = s[\boldsymbol{c} := \boldsymbol{c} + \delta]$, $\delta \geq 0$, $s' \models I$, and $\forall \delta'.0 \leq \delta' < \delta : s[\boldsymbol{c} := \boldsymbol{c} + \delta'] \models (I \wedge \neg U)$. That is:

$$\begin{aligned}Post_\mathrm{t}([\![\psi]\!]_z, \delta) = \{s' : s \in [\![\psi]\!]_z \wedge \delta \geq 0 \wedge s[\boldsymbol{c} := \boldsymbol{c} + \delta] \models I \wedge \\ \forall \delta'.0 \leq \delta' < \delta : s[\boldsymbol{c} := \boldsymbol{c} + \delta'] \models (I \wedge \neg U)\} .\end{aligned}$$

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

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

and, similarly:

$$s[\boldsymbol{c} := \boldsymbol{c} + \delta'] \models (I \wedge \neg U) \equiv s \in [\![I_{z''} \wedge \neg U_{z''}]\!]_z \,,$$

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

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

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

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

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

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

Using the definition of replacement, we get:

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

□

Theorem 4 (Correctness of $\mathbf{post}_{\mathrm{t}}(\psi)$). *Let ψ be a difference constraint expression, and let $P = (B, C, T, I, U)$ be a* TGC *program. Then* $Post_{\mathrm{t}}([\![\psi]\!]_z) = [\![\mathbf{post}_{\mathrm{t}}(\psi)]\!]_z = [\![\exists z.(\psi \wedge P_{\mathrm{post}})[z/z']]\!]_z$.

Proof. The first equality in the theorem follows immediately from Definitions 14 and 25. The second equality holds because existential quantification distributes over disjunction:

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

□

Example 18. Consider a TGC program with the program invariant $I = x \neq 5$. Then the predicate P_{post} is given by:

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

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 $[\![\mathbf{post}_{\mathrm{t}}(\phi_z)]\!]_z$, where:

$$\mathbf{post}_{\mathrm{t}}(\phi_z) = \exists z.(\phi_z \wedge P_{\mathrm{post}})[z/z'] = (1 \leq x - z < 5) \vee (7 \leq x - z)\,.$$

That is, advancing time from ϕ gives $(1 \leq x < 5) \vee (7 \leq x)$.

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 $P = (B, C, T, I, U)$ 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 15:

Theorem 5 (Correctness of $\mathbf{post}(\psi)$ and $\mathbf{post}^*(\psi)$). *Let ψ be a difference constraint expression, and let $P = (B, C, T, I, U)$ be a* TGC *program. Then $Post([\![\psi]\!]_z) = [\![\mathbf{post}(\psi)]\!]_z$ and $Post^*([\![\psi]\!]_z) = [\![\mathbf{post}^*(\psi)]\!]_z$.*

Proof. Follows immediately from Definition 15 and Theorems 2 and 4. □

Example 19. Consider again the program from Example 3. The set of states reachable from $\phi = b \wedge (x = y = 0)$ is $[\![\mathbf{post}^*(\phi_z)]\!]_z$, where:

$$\begin{aligned}\mathbf{post}^*(\phi_z) = {} & (b \wedge x = y \wedge x - z \leq 9) \vee \\ & (\neg b \wedge [(x = y \wedge 1 \leq x - z < 5) \vee (7 \leq x - y \leq 9 \wedge 7 \leq x - z)])\,.\end{aligned}$$

3.3 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 27 ($\mathbf{pre}_\mathrm{d}$-operator). *Let ψ be a difference constraint expression, and let $P = (B, C, T, I, U)$ be a* TGC *program. Then the $\mathbf{pre}_\mathrm{d}$-operator for backward execution of the command $\phi \to \boldsymbol{v} := \mathbf{any}\, \boldsymbol{v}'.\phi'$ is defined as*

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

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 the timed guarded command $\phi \to \boldsymbol{v} := \mathbf{any}\, \boldsymbol{v}'.\phi'$.

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

Definition 28 ($\mathbf{pre}_\mathrm{t}$-operator). *Let ψ be a difference constraint expression, and let $P = (B, C, T, I, U)$ be a* TGC *program. Then the $\mathbf{pre}_\mathrm{t}$-operator for decreasing time by δ in all states $[\![\psi]\!]_z$ is defined as*

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

where the last equality follows from the definition of assignment, and where P_pre is equivalent to P_post with z and z' exchanged:

$$P_\mathrm{pre} = (z \leq z') \wedge I_z \wedge \forall z''.\big((z < z'' \leq z') \Rightarrow (I_{z''} \wedge \neg U_{z''})\big)\,.$$

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 29 (pre-operator). *Let ψ be a difference constraint expression, and let $P = (B, C, T, I, U)$ 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 how we proved the correctness of the corresponding **post** operators.

3.4 Verification of Simple 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 30 (Forward reachability). *Let ψ be a difference constraint expression, and let $P = (B, C, T, I, U)$ 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 31 (Backward reachability). *Let ψ be a difference constraint expression, and let $P = (B, C, T, I, U)$ 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.*

3.5 Verification of Timed CTL Properties

The **pre** operators can also be used to perform symbolic model checking of timed CTL [30]. Timed CTL is obtained by extending CTL [17] 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 *reset quantifier* $u.\psi$. 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 reset quantifier is computed symbolically as:

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

that is, the reset quantifier corresponds to restricting the value of u to zero and then remove it by existential quantification (assuming that ψ uses z as a zero point).

3.6 A Simpler Semantics

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 32 (STGC syntax). *A simplified timed guarded command (*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 $\boldsymbol{v} := \mathbf{any}\, \boldsymbol{v'}.\phi$, where $\boldsymbol{v}, \boldsymbol{v'} \in B \cup C$ are vectors of variables and $\phi \in \Phi$ is an expression.*

Definition 33 (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 $\boldsymbol{v} := \mathbf{any}\, \boldsymbol{v'}.\phi$, the following inference rule defines the transition relation:*

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

Analogous to Definition 24 we define the operator $\mathbf{post}_{\text{STGC}}$ as follows:

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

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

Definition 34 (Induced STGC). *A* TGC *program $P = (B, C, T, I, U)$ induces an* STGC *program $P' = (B, C, T')$, such that for each timed guarded command $\phi \rightarrow \boldsymbol{v} := \mathbf{any}\, \boldsymbol{v'}.\phi'$ in T there is a command $\boldsymbol{v} := \mathbf{any}\, \boldsymbol{v'}.(\phi' \wedge \phi \wedge I[\boldsymbol{v'}/\boldsymbol{v}])$ in T'. Furthermore, T' contains the command $z := \mathbf{any}\, z'.P_{\text{post}}$, where P_{post} is defined as in Definition 25.*

Theorem 6 (Equivalence between TGC and STGC). *Let P be a* TGC *program, and let P' be the induced* STGC *program. Then P and P' define the same transition system.*

Proof. Using Definition 23 and Theorem 2 we first show that we can embed the guard and the program invariant in the expressions of the timed guarded commands in T:

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

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

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

□

Using the symmetry between P_{pre} and P_{post}, it is not difficult to prove the following theorem:

Theorem 7. *Let ψ be a difference constraint expression, and let P be a* TGC *program. Then the* $\mathbf{pre}_{\mathrm{t}}$*-operator for decreasing time in all states* $[\![\psi]\!]_z$ *can be defined as:*

$$\mathbf{pre}_{\mathrm{t}}(\psi) = \mathbf{pre}_{\mathrm{d}}(\psi, \mathbf{true} \to z := \mathbf{any}\, z'.P_{\mathrm{post}})$$

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) [1,11]. 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 prob-

lem 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 [21]. 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 [30] for a proof.

The main problem is DCE-SAT: determine satisfiability of a difference constraint expression as defined in Definition 20:

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

Interestingly, it turns out that adding quantifiers "only" makes the problem **PSPACE**-complete [34]. 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 [41].[4]

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 [32], and the theory of integers with addition and order (also called Presburger arithmetic [42]), which is **2-NEXP**-hard [20,26,46]. The theory of reals with addition and multiplication was shown to be decidable by Tarski [47], whereas the theory of integers with addition and multiplication (also called number theory) is undecidable—Gödel's famous Incompleteness Theorem [28].

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 [33]; quantifier-free difference constraint expressions can be decided using disjunctive programming [5]; 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.

4.2 Quantifier Elimination

Adding quantifiers to the vocabulary of a language greatly increases the expressive power, but also moves the satisfiability problem up in the hierarchy

[4] This also justifies the encoding of Boolean variables as difference inequalities.

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. [19,32]. For Presburger arithmetic the difference is even bigger: **NP**-complete for quantifier-free formulae [40], but **2-NEXP**-hard for formulae with quantifiers.

Since Tarski [47] 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 [27] 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 [20] eliminates quantifiers from Presburger formulae, and Ferrante and Rackoff [25] give an elimination procedure for the theory of reals with addition and order. Ferrante and Geiser [24] study quantifier elimination in the theory of rationals with addition and order. Collins [18] pioneered the development of cylindrical algebraic decomposition (CAD) techniques for quantifier elimination in the theory of reals with addition and multiplication. Koubarakis [34] 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.

4.3 Difference Bound Matrices

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[5]. The constraint graph can be represented as a so-called

[5] 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 $(\leq, d)$.

difference bound matrix (DBM) [23] M, where M_{ij} contain the least upper bound on $x_j - x_i$ or ∞ if there is no upper bound.

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. 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. 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 [50] and UPPAAL [36], but they suffer from a number of problems as discussed in Sect. 1.1.

4.4 Difference Decision Diagrams

Difference decision diagrams (DDDs) [38] are another candidate for a data structure for representing difference constraint expressions. Similar to how a BDD [13] 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 $\alpha \to \psi_1, \psi_0$, 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:

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

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

Example 20. As an example of a DDD consider the following expression ψ over $x,\ y,\ z \in \mathbb{R}$:

$$\psi = 1 \le x - z \le 3 \ \wedge \ (y - z \ge 2 \ \vee \ y - x \ge 0) .$$

Figure 5 shows $[\![\psi]\!]_z$ as an (x, y)-plot and the corresponding DDD.

As shown in [38], DDDs can be ordered and reduced making 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 20 are easily defined

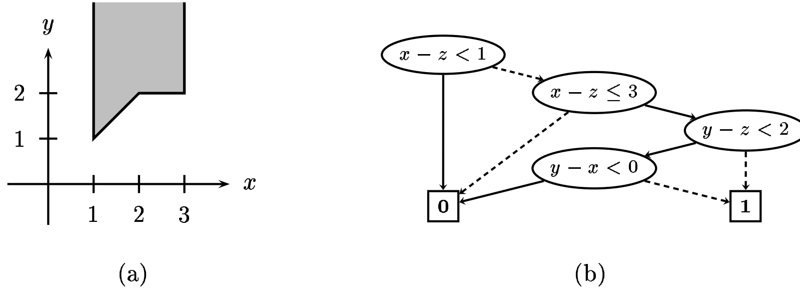

Fig. 5. The expression ψ in Example 20 as (a) an (x, y)-plot for $z = 0$, and (b) a difference decision diagram. High- and low-branches are drawn with solid and dashed lines, respectively.

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 20. APPLY is a generalization of the version used for BDDs 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 [27], 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 Definition 21 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 Examples

In this section we look at some examples: first we study the simple pulse generator from Sect. 2, and then we look at three (slightly) more realistic examples, namely Milner's Scheduler, Fischer's mutual exclusion protocol, and an RGD arbiter.

5.1 A Pulse Generator

Let us look at the asynchronous circuit from Example 15 again. We want to verify that the circuit is hazard-free using backward reachability analysis. Assume that the minimum and maximum delays are $[d_{out\uparrow}, D_{out\uparrow}] = [d_{out\downarrow}, D_{out\downarrow}] = [1, 2]$, $[d_{ack\uparrow}, D_{ack\uparrow}] = [d_{ack\downarrow}, D_{ack\downarrow}] = [1, 3]$ and $[d_{req\uparrow}, D_{req\uparrow}] = [d_{req\downarrow}, D_{req\downarrow}] = [3, 3]$. The initial state of the circuit is:

$$\psi_0 = \neg out \wedge \neg out_u \wedge \neg ack \wedge \neg ack_u \wedge \neg req \wedge \neg req_u\,.$$

The hazard predicate for the circuit simplifies to:

$$\begin{aligned} H = {} & \big(out_u \wedge (out \Leftrightarrow (ack \oplus req))\big) \\ & \vee \big(ack_u \wedge (out \Rightarrow (ack \Leftrightarrow req))\big) \\ & \vee \big(req_u \wedge (ack \oplus req)\big) \end{aligned}$$

The circuit is hazard-free if and only if $\neg H$ is an invariant. To verify this, we compute $\mathbf{pre}^*(H)$ and check that this formula does not intersect with ψ_0 using the DDD data structure. The fixpoint $\mathbf{pre}^*(H)$ is reached after 4 iterations and does not intersect with ψ_0. Thus, the circuit has no hazards. If we change the delays for the XOR-gate (out) from $[1, 2]$ to $[1, 3]$, the circuit has a hazard which is reached after 12 iterations.

5.2 Milner's Scheduler

Milner's scheduler 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 H_i 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 T_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 ith cycler is described by two guarded commands and the task is modeled by a third guarded command:

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

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

$$I = \bigwedge_{i=1}^{N} (h_i \Rightarrow H_i \leq 200) \wedge (t_i \Rightarrow T_i \leq 100)$$

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 $\mathbf{post}^*(\psi_0)$ for increasing number N of cyclers. This version of Milner's scheduler has exponentially many discrete states because a task can terminate independently of the other tasks. Thus, state-space exploration based on enumerating all discrete states only succeeds for small systems. In the symbolic approach, discrete states are represented implicitly and choosing a good ordering of the variables in the DDD gives polynomial runtimes and state space representations. A system with $N = 32$ schedulers can be verified in a few seconds.

5.3 Fischer's Mutual Exclusion Protocol

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

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

We use the variable s_i to represent the state of a process. We write $s_i = idle_i$ for the predicate $\neg t_i^1 \wedge \neg t_i^0$, and $s_i := idle_i$ for the assignment $t_i^1, t_i^0 := \mathbf{false}, \mathbf{false}$, 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:

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

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:

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

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 for increasing number N of processes, but unlike Milner's scheduler the runtimes increase more rapidly and we could only verify systems with up to $N = 8$ processes in a few seconds.

5.4 An Arbiter

An RGD arbiter is a device that receives *requests* from two clients and issues *grants* to them such that at most one *request* is granted at a time. That is, the arbiter guarantees mutual exclusion to some resource. We say that a client is granted when its receives a *grant* for a request. When a client no longer needs the resource, it issues a *done* event to the arbiter. A client must be granted to issue a *done* event, so we need only one *done* event.

Figure 6 shows an abstract model of an RGD arbiter, and a state–transition diagram for one client. A client can be in one of four states: *idle*, *req*, *pend* or *grant*. If the client is *idle*, it can issue a request to the arbiter and move to the *req* state. When the arbiter grants the request, the client can move to the *grant* state. From this state the client can either release the resource by sending the arbiter a done event and move to the *idle* state, or it can issue another request and move to the *pend* state. The arbiter must guarantee *mutual exclusion*; that is, at most one client should be granted at a time.

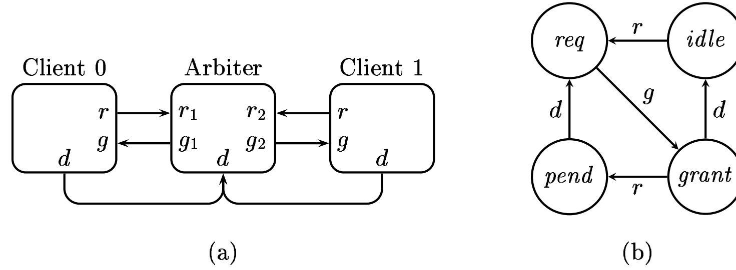

Fig. 6. RGD arbiter. (a) Abstract model. (b) State–transition diagram for one client.

In the abstract model of the arbiter we use events to model that two components (the arbiter and one of its clients) agree that something (a request, a grant, etc.) has happened, and which causes both components synchronously to perform some action. Events are good abstractions in the modeling phase of an

arbiter, but in a real design of an arbiter there are no such things as events, as things do not happen synchronously. In a more realistic model, a client changes the value of some wires to indicate that it is requesting; then after some time the arbiter acquires this information; and — when the resource becomes available — the arbiter will grant the client by changing the value of some other wires, which the client will notice, and so on. There are many ways to define exactly how this handshake mechanism has to work; in this example we will look at a protocol that uses *pulses* [29]. The idea is that a client sends a request (done) to the arbiter by changing the value of a wire r_i (d) from low to high, holding it high for some minimum time, and then changing it back to low again. We can think of lifting up a long rope lying on the ground and slapping it down, causing a wave to pulse from one end of the rope to the other. The arbiter issues grants similarly by firing back a pulse to the client using a wire g_i.

We can model this kind of pulse-communication in a TGC program as follows: when a client wants to request the resource, it sets a signal r high for at least R^h time units. Thereafter, the client may set r low again (but may also wait). The client must hold r low for at least R^l before it is allowed to issue a new request, see Fig. 7. Unlike a 2- or 4-phase handshake protocol, there is no ordering constraint on the falling edge of pulses. Instead, the component (a client or the arbiter) that generates a pulse must ensure that the pulse satisfies a minimum width constraint, and the component receiving the pulse must acquire it within that time. The arbiter grants requests in a similar way: When the arbiter has noticed that r is high, it will grant the request some time after (maybe infinitely long time after if the other client does not release the resource). The arbiter grants the request by setting g high for at least G^h time units. The client must acquire the grant if g is high for at least G^h time units. After G^h time units, the arbiter may set g low again (but may also wait). The arbiter must then hold g low for at least G^l time units before it is allowed to grant a new request.

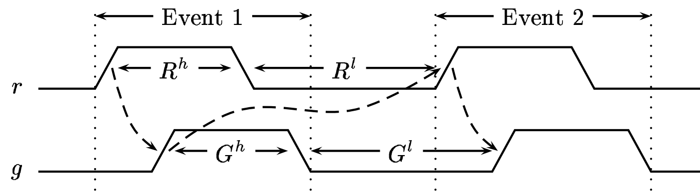

Fig. 7. Representing events as pulses.

To model a pulse as a set of guarded commands, we use a timer R_i to measure the time since the last r_i-transition from low-to-high or high-to-low. If r_i has been low for at least R^l time units, we can change r_i to high and reset R_i. Similarly, if r_i has been high for at least R^h time units, we can change r_i to low and reset

R_i. We can model this as two guarded commands:

$$\begin{array}{ll} \neg r_i \wedge R_i \geq R^l & \rightarrow r_i, R_i := \mathbf{true}, 0 \\ r_i \wedge R_i \geq R^h & \rightarrow r_i, R_i := \mathbf{false}, 0 \end{array}$$

As a first attempt to model the pulse arbiter, we can augment the state-transition diagram in Fig. 6(b) with the TGC program for a pulse. It turns out, however, that this it not sufficient to guarantee mutual exclusion. Consider the following scenario: A client makes a request and becomes granted; the g signal remains high and the client makes a pending request; the client sends the arbiter a done pulse, and because g still is high, the client thinks it has become granted again. Seeing the done pulse, the arbiter thinks that the first client is done with the resource, then grants the other client, and now both clients are granted at the same time. The problem is, that the clients and the arbiter need some internal variables to keep track of when they have acquired a pulse. To do so, we introduce five extra Boolean variables, r'_i, g'_i, d', for $i = 0, 1$, (initially all **false**). The idea is that a client only can move from *req* to *grant* when g_i is high and g'_i is low, and the client then sets g'_i high when taking the transition, indicating that it has acquired that grant. This gives us the following TGC program for the pulse arbiter ($i = 0, 1$):

$$\begin{array}{lll} c_i = \mathit{idle} \wedge \neg r_i \wedge R_i \geq R^l & \rightarrow c_i, r_i, R_i & := \mathit{req}, \mathbf{true}, 0 \\ c_i = \mathit{grant} \wedge \neg r_i \wedge R_i \geq R^l & \rightarrow c_i, r_i, R_i & := \mathit{pend}, \mathbf{true}, 0 \\ c_i = \mathit{req} \wedge g_i \wedge \neg g'_i & \rightarrow c_i, g'_i & := \mathit{grant}, \mathbf{true} \\ c_i = \mathit{grant} \wedge \neg d \wedge D \geq D^l & \rightarrow c_i, d, D & := \mathit{idle}, \mathbf{true}, 0 \\ c_i = \mathit{pend} \wedge \neg d \wedge D \geq D^l & \rightarrow c_i, d, D & := \mathit{req}, \mathbf{true}, 0 \\ (a_i = \mathit{idle} \vee a_i = \mathit{grant}) \wedge r_i \wedge \neg r'_i & \rightarrow a_i, r'_i & := \mathit{req}, \mathbf{true} \\ a_i = \mathit{req} \wedge \neg g_i \wedge G_i \geq G^l \wedge \neg f & \rightarrow a_i, g_i, G_i, f & := \mathit{grant}, \mathbf{true}, 0, \mathbf{true} \\ (a_i = \mathit{grant} \vee a_i = \mathit{pend}) \wedge d \wedge \neg d' & \rightarrow a_i, d', f & := \mathit{idle}, \mathbf{true}, \mathbf{false} \\ r_i \wedge R_i \geq R^h & \rightarrow r_i, r'_i, R_i & := \mathbf{false}, \mathbf{false}, 0 \\ g_i \wedge G_i \geq G^h & \rightarrow g_i, g'_i, G_i & := \mathbf{false}, \mathbf{false}, 0 \\ d \wedge D \geq D^h & \rightarrow d, d', D & := \mathbf{false}, \mathbf{false}, 0 \end{array}$$

The initial set of states ψ_0 is given by

$$\psi_0 \equiv \neg d \wedge \neg d' \wedge \neg f \wedge \bigwedge_{i=0,1} \neg r_i \wedge \neg g_i \wedge \neg r'_i \wedge \neg g'_i \wedge c_i = \mathit{idle} \wedge a_i = \mathit{idle}\,.$$

To verify that the TGC program for the pulse arbiter guarantees mutual exclusion we have used the DDD data structure to compute $\mathbf{pre}^*(\bigwedge_{i=0,1} \mathit{grant}_i \vee \mathit{pend}_i)$ and check that it does not intersect with ψ_0. Thus, we have proved that the model of the arbiter guarantees mutual exclusion.

6 Summary

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. 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 described 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. Quantifier elimination is the core operation in real-time model checking, and a more efficient implementation of this operator is the focus of current research.

References

1. R. Alur. *Techniques for Automatic Verication of Real-Time Systems.* PhD thesis, Department of Computer Science, Stanford University, 1991.
2. R. Alur and D. Dill. The theory of timed automata. In *Proc. Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 28–73. Springer-Verlag, 1991.
3. 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, *Proc. International Workshop on Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*, pages 346–360, Grenoble, France, 1997. Springer-Verlag.
4. F. Balarin. Approximate reachability analysis of timed automata. In *Proc. Real-Time Systems Symposium*, pages 52–61. IEEE Computer Society Press, 1996.
5. E. Balas. Disjunctive programming. *Annals of Discrete Mathematics*, 5:3–51, 1979.
6. G. Behrmann, K.G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. Technical Report 99/105, DoCS, Uppsala University, 1999.
7. W. Belluomini and C.J. Myers. Efficient timing analysis algorithms for timed state space exploration. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 88–100, April 1997.
8. W. Belluomini and C.J. Myers. Verification of timed systems using POSETs. In *Proc. Tenth International Conference on Computer-Aided Verification*, pages 403–415, June 1998.
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. A. Blass and Y. Gurevich. Fixed-choice and independent-choice logics. Technical Report TR-369-98, University of Michigan, August 1998.
11. P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Are timed automata updatable? In *Proc. 12th International Conference on Computer-Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 464–479. Springer-Verlag, July 2000.
12. M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In O. Grumberg, editor, *Proc. Ninth International Conference on Computer-Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 179–190, 1997.
13. R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
14. J.R. Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems.* PhD thesis, Carnegie Mellon, August 1992.
15. 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 *Proc. Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439. IEEE Computer Society Press, 1990.
16. S.V. Campos, E. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing quantitative characteristics of finite-state real-time systems. In *Proc. Real-Time Systems Symposium*, pages 266–70. IEEE Computer Society Press, December 1994.
17. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, May 1981.
18. G.E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In *Proc. Second GI Conference*, volume 33 of *Lecture Notes in Computer Science*, pages 134–183. Springer-Verlag, 1975.
19. S.A. Cook. The complexity of theorem-proving procedures. In *Proc. Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, Ohio, 1971.
20. D.C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence*, 7:91–99, 1972.
21. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* MIT Press, 1994.
22. E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.
23. D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1989.
24. J. Ferrante and J.R. Geiser. An efficient decision procedure for the theory of rational order. *Theoretical Computer Science*, 4(2):227–233, 1977.
25. J. Ferrante and C. Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM Journal of Computing*, 4(1):69–76, 1975.
26. M.J. Fischer and M.O. Rabin. Super-exponential complexity of presburger arithmetic. In *Proc. SIAM-AMS Symposium in Applied Mathematics*, volume 7, pages 27–41, 1974.
27. J.B.J. Fourier. Second extrait. In G. Darboux, editor, *Oeuvres*, pages 325–328, Paris, 1890. Gauthiers-Villars.
28. K. 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.

29. M.R. Greenstreet and T. Ono-Tesfaye. A fast, ASP*, RGD arbiter. In *Proc. Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 173–185, Barcelona, Spain, April 1999. IEEE.
30. T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.
31. D. Hilbert and P. Bernays. *Grundlagen der Mathematik*, volume 2. Springer-Verlag, 1939.
32. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
33. N.K. Karmarkar. A new polynomial–time algorithm for linear programming. *Proc. 16th Annual ACM Symposium on Theory of Computing*, pages 302–311, 1984.
34. M. Koubarakis. Complexity results for first-order theories of temporal constraints. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Proc. Fourth Internatinal Conference on Principles of Knowledge Representation and Reasoning*, pages 379–390, San Francisco, California, 1994. Morgan Kaufmann.
35. K.G. Larsen, P. Pettersson, and W. Yi. Model-checking for real-time systems. In *Proc. Tenth International Conference on Fundamentals of Computation Theory*, volume 965 of *Lecture Notes in Computer Science*, pages 62–88, August 1995.
36. K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.
37. O. Maler and A. Pnueli. Timing analysis of asynchronous circuits using timed automata. In P.E. Camurati and H. Eveking, editors, *Proc. Eighth International Conference on Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*, pages 189–205. Springer-Verlag, 1995.
38. J. Møller, J. Lichtenberg, H.R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Proc. 13th International Conference on Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125, Madrid, Spain, September 1999.
39. J. Møller, J. Lichtenberg, H.R. Andersen, and H. Hulgaard. Fully symbolic model checking of timed systems using difference decision diagrams. In *Proc. First International Workshop on Symbolic Model Checking*, volume 23-2 of *Electronic Notes in Theoretical Computer Science*, pages 89–108, Trento, Italy, July 1999.
40. 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.
41. C.H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.
42. 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.
43. T.G. Rokicki. *Representing and Modeling Digital Circuits.* PhD thesis, Stanford University, 1993.
44. 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*, volume 818 of *Lecture Notes in Computer Science*, pages 468–480, 1994.
45. S.Bornot and J.Sifakis. An algebraic framework for urgency. *Information and Computation*, 163(1):172–202, 2000.
46. R. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 4(24):529–543, October 1977.
47. A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, Berkeley, CA, 2nd edition, 1951.

48. 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.
49. 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.
50. S. Yovine. KRONOS: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2):123–133, October 1997.
51. S. Yovine. Model checking timed automata. In *Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*, pages 114–152. Springer-Verlag, October 1998.