Simplifying Fixpoint Computations in Verification of Real-Time Systems

Jesper B. Møller

Department of Innovation, IT University of Copenhagen
jm@it.edu

Abstract. Symbolic verification of real-time systems consists of computing the least fixpoint of a functional which given a set of states ϕ returns the states that are reachable from ϕ (in forward reachability), or that can reach ϕ (in backward reachability). This paper presents two techniques for simplifying the fixpoint computation: First, I demonstrate that in backwards reachability, clock resets and discrete state changes can be performed as substitutions instead of existential quantifications over reals and Booleans, respectively. Second, I introduce a local-time model for real-time systems which allows clocks to advance asynchronously, thus resulting in an over-approximation of the least fixpoint, but which in some cases is sufficient for verifying a temporal property.

1 Introduction

Symbolic model checking is an efficient technique for verifying the correctness of finite state systems [8]. The basic idea is to construct the set of reachable states R, and then determine whether a given state is in R. By representing sets of states symbolically as predicates over Boolean variables using for instance binary decision diagrams (BDDs) [7], it is possible to verify systems with a very large number of states. Henzinger et al. [11] have shown how to extend symbolic model checking to real-time systems, and many others have contributed to this growing research field which can be divided into five areas:

- Notations for modeling systems (e.g., timed automata [2], timed Petri nets [4], and timed guarded commands [12])
- Logics for specifying requirements (e.g., timed computation tree logic [1], and metric interval temporal logic [3])
- Data structures and algorithms for representing and manipulating logics (e.g., difference bound matrices [9], difference decision diagrams [22], clock difference diagrams [15], and quantifier elimination [13])
- Techniques for formal analysis (e.g., partial order techniques [10, 19], and compositional and on-the-fly techniques [16, 5])
- Tools for automatic verification (e.g., KRONOS [29], UPPAAL [18], RED [27], and TEMPO [25])

The core operation in reachability analysis of real-time system is the computation of the least fixpoint of a functional of the form $\mu X[\phi_0 \vee f(X)]$, where f is a

function that given a set of states, represented by a formula ϕ, returns the set of states that are reachable from ϕ (denoted $\mathbf{post}(\phi)$) in forward reachability, or that can reach ϕ (denoted $\mathbf{pre}(\phi)$) in backward reachability.

The main contributions of this paper are two techniques for simplifying the fixpoint computation. First, I show that in backwards reachability, we can perform clock resets and discrete state changes as substitutions instead of existential quantifications over reals and Booleans, respectively. In forward reachability, which is used by many verification tools (e.g., [4, 24, 17, 25]), resetting a clock x in a set of states represented by a formula ϕ can be performed as $(\exists x.\phi) \wedge x = 0$. When analysing the system backwards, resetting x can be performed as $\exists x.\big(\phi \wedge (x = 0)\big) \equiv \phi[0/x]$. Similarly, a discrete state change that corresponds to the assignment $b := e$, where b is a Boolean variable and e is an expression, can be performed backwards as $\exists b.\big(\phi \wedge (b \Leftrightarrow e)\big) \equiv \phi[e/b]$. Williams et al. [28] use a similar technique for performing quantification by substitution in the analysis of (untimed) circuits. Existential quantification is an expensive operation which complicates the satisfiability problem of the underlying logic (both in the timed and untimed case) from **NP**-complete to **PSPACE**-complete.

Second, I introduce a simple notation for modeling timed systems called δ-programs. A δ-program is a set of commands of the form $\delta\boldsymbol{v}.\phi$, where $\boldsymbol{v}$ is a vector of variables to update, and ϕ is a transition relation over primed and unprimed variables. The notation is similar to timed guarded commands [12], except that assignments are expressed as predicates over primed and unprimed variables, and time is modeled explicitly as a program variable z which is interpreted as the common zero point of all clocks. A command for advancing time can be specified as $\delta z.(z' \leq z \wedge P)$, where P is a predicate expressing whether it is legal to change the zero point from z to z'. Introducing time explicitly in the notation makes the semantics of δ-programs simple, consisting of only one inference rule. And, consequently, it is easy to define the symbolic reachability operators **post** and **pre**, and prove correctness of them. I present a local-time model for real-time systems in which clocks advance time asynchronously. This is accomplished by introducing a zero point z_i for each clock x_i in a δ-program, and then advancing time by using a command of the form $\delta\boldsymbol{z}.(\bigwedge_{i=1}^{n} z_i' \leq z_i \wedge P_i)$, where P_i specifies whether it is legal to change the zero point from z_i to z_i'. The local-time model results in an over-approximation of the least fixpoint, but in some cases it is still possible to use this over-approximation for verifying a temporal property.

I have implemented the two techniques using a data structure called difference decision diagrams [22], and evaluated the efficiency of the techniques on some real-time case studies. Experiments show that backwards reachability performs better than forward reachability with respect to both time and space consumptions. In conjunction with the local-time model, it is furthermore possible to verify mutual exclusion for Fischer's protocol in linear time, and to compute the exact reachable state space for the alpha and beta examples defined in [6], also in linear time.

The rest of the paper is organized as follows: Section 2 introduces δ-programs for modeling real-time systems, and Section 3 describes the two techniques for

simplifying the fixpoint computation. Section 4 reports experimental results for the two techniques using an implementation of difference decision diagrams [23], and Section 5 concludes with a discussion of the advantages and limitations of the presented ideas, and suggestions for future research.

2 Modeling Timed Systems

δ-programs are a simple notation for modeling real-time systems similar to timed guarded commands [12] but allowing nondeterministic, independent-choice assignments of real variables. The notation is expressive enough to encode popular models of systems with time such as timed automata [2] and timed Petri nets [4].

2.1 δ-Programs

The basic components of δ-programs are variables, expressions, and commands.

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

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

$$\phi ::= b \mid x \sim d \mid x - y \sim d \mid \neg\phi \mid \phi_1 \land \phi_2 \mid \phi_1 \lor \phi_2 \mid \exists b.\phi \mid \exists x.\phi \,,$$

where $b \in \mathcal{B}$ is a Boolean variable, $x, y \in \mathcal{C}$ are clocks, $d \in \mathbb{Q}$ is a rational constant, $\sim \in \{\leq, <, =, >, \geq\}$ 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), $\land$ (conjunction), $\lor$ (disjunction), and $\exists$ (existential quantification) have their usual meaning. The operators $\Rightarrow$ (implication), $\Leftrightarrow$ (biimplication) and $\forall$ (universal quantification) are defined the standard way. We define replacement of a vector $\boldsymbol{v'} \in \mathcal{V}^n$ of variables by another vector $\boldsymbol{v} \in \mathcal{V}^n$ of variables in an expression ϕ, denoted by $\phi[\boldsymbol{v}/\boldsymbol{v'}]$, as the syntactic substitution of all occurrences of v'_i in ϕ by v_i, for $i = 1, \ldots, n$. We define assignment of a vector $\boldsymbol{v} \in \mathcal{V}^n$ of variables to a vector $\boldsymbol{v'} \in \mathcal{V}^n$ of variables in an expression ϕ as $\phi[\boldsymbol{v} := \boldsymbol{v'}] = (\exists \boldsymbol{v}.\phi)[\boldsymbol{v}/\boldsymbol{v'}]$. The meaning of an expression over $\boldsymbol{v}$ is defined as the standard interpretation of the variables $\boldsymbol{v}$.

Definition 3 (State). *A state s is an interpretation of the variables in $\mathcal{V}$. For a vector of variables $\boldsymbol{v} \in \mathcal{V}^n$, $s(\boldsymbol{v}) \in (\mathbb{B} \cup \mathbb{R})^n$ denotes the interpretation of $\boldsymbol{v}$ 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. If any state satisfies ϕ, ϕ is a tautology and we write $\models \phi$. Let $\boldsymbol{v}$ and $\boldsymbol{v'}$ be n-dimensional vectors of variables, and let $\boldsymbol{r} \in (\mathbb{B} \cup \mathbb{R})^n$ be an n-dimensional vector of values. Then the state $s' = s[\boldsymbol{v} := \boldsymbol{v'} + \boldsymbol{r}\,]$ is equivalent to s except that $s'(\boldsymbol{v}) = s(\boldsymbol{v'}) + \boldsymbol{r}$.*

Definition 4 (Command). *Let $\boldsymbol{v} \in \mathcal{V}^n$ be an n-dimensional vector of variables, and ϕ an expression over $\boldsymbol{v}$ and $\boldsymbol{v'}$. Then a command has the form $\delta\boldsymbol{v}.\phi$.*

A command $\delta(v_1, \ldots, v_n).\phi$ specifies a guarded, nondeterministic, independent-choice assignment: Assign to each variable v_i any value v_i', for $i = 1, \ldots, n$, such that the expression ϕ is satisfied. The choice of a value for a variable v_i in one command is made nondeterministically and independently of choices for other variables in other commands. The expression ϕ is used to express both the guard of the command (when it is enabled to execute) as a predicate over current-state variables $\boldsymbol{v}$, and the assignment of the command (the effect of executing it) as a predicate over the current-state variables $\boldsymbol{v}$ and next-state variables $\boldsymbol{v'}$. A command is said to be enabled in a state s if $s \models \phi$.

Definition 5 (δ-program). *A δ-program P is a tuple (V, C), where $V \subseteq \mathcal{V}$ is a set of variables, and C is a set of commands over V and V'. The semantics of a 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. For each command $\delta\boldsymbol{v}.\phi \in C$ there is a transition $\xrightarrow{\delta\boldsymbol{v}.\phi}$ defined by the inference rule:*

$$\frac{s[\boldsymbol{v'} := \boldsymbol{r}\,] \models \phi}{s \xrightarrow{\delta\boldsymbol{v}.\phi} s[\boldsymbol{v} := \boldsymbol{r}\,]}\;.$$

Example 1. Let us consider a simple example. Let $P = (V, C)$ be a program with variables $l_1, l_2 \in \mathcal{B}$ and $x_1, x_2, z \in \mathcal{C}$, and the following commands:

$$\begin{aligned}
&\delta(l_1, l_2, x_1).(l_1 \wedge \neg l_1' \wedge l_2' \wedge x_1' = z)\\
&\delta(l_1, l_2, x_2).(l_2 \wedge \neg l_2' \wedge l_1' \wedge x_2' = z)\\
&\delta(z).\big(z' \le z \wedge \forall z''(z' \le z'' \le z \Rightarrow (l_2 \Rightarrow x_1 - z'' \le 5))\\
&\qquad\qquad \wedge \forall z''(z' \le z'' \le z \Rightarrow (l_1 \Rightarrow x_2 - z'' \le 4))\big)
\end{aligned}$$

The first command assigns to l_1, l_2, x_1 any values l_1', l_2', x_1' such that the expression $l_1 \wedge \neg l_1' \wedge l_2' \wedge x_1' = z$ is satisfied. Intuitively this means, that if l_1 is true, then l_1 is set to false, l_2 is set to true, and x_1 is set to z. Similarly for the second command. The third command assigns to z any value z' for which $z' \le z$, and $l_1 \Rightarrow x_2 - z'' \le 4$ and $l_2 \Rightarrow x_1 - z'' \le 5$ are satisfied for all $z'' \in [z', z]$.

2.2 Timed Automata

It is straightforward to translate a timed automaton [2] into an equivalent δ-program. The key idea is to introduce a special variable z which is interpreted as the common zero point of all clocks in the automaton. As shown in [20] it is then possible to synchronously advance time in a set of states represented by an expression ϕ as an existential quantification over z in ϕ.

A timed automaton A is a tuple $(\mathcal{L}, \mathcal{X}, \mathcal{I}, \mathcal{T})$: $\mathcal{L}$ is a set of locations, $\mathcal{X}$ is a set of clocks, $\mathcal{I}$ is a set of location invariants, and $\mathcal{T}$ is a set of transitions of the form (L_i, Y, G_i, L_j), where $L_i \in \mathcal{L}$ is the source location, $L_j \in \mathcal{L}$ is the destination location, G_i is a guard over the clocks, and $Y \subseteq \mathcal{X}$ is a set of clocks to reset

when the transition is taken. Guards and location invariants are conjunctions of clock constraints of the form $X_1 \sim d$ and $X_1 - X_2 \sim d$, where $X_1, X_2 \in \mathcal{X}$ are clocks.

We translate a timed automaton into a δ-program as follows: For simplicity, we encode each location $L_i \in \mathcal{L}$ as a Boolean variable l_i (a logarithmic encoding may be more efficient in practice). We encode each clock $X_i \in \mathcal{X}$ as a real-valued variable x_i. For each guard G_i we construct a modified guard g_i where we replace each constraint of the form $X_j \sim d$ by $x_j - z \sim d$, and replace each constraint of the form $X_j - X_k \sim d$ by $x_j - x_k \sim d$. We translate location invariants $I_i \in \mathcal{I}$ into inv_i in a similar way. For each transition of the form $(L_i, \{X_1, \ldots, X_m\}, G_i, L_j)$, we add the command:

$$\delta(l_i, l_j, x_1, \ldots, x_m).\big(g_i \wedge l_i \wedge \neg l_i' \wedge l_j' \wedge \textstyle\bigwedge_{k=1}^{m}(x_k' = z)\big).$$

Finally, we add the following command for advancing time:

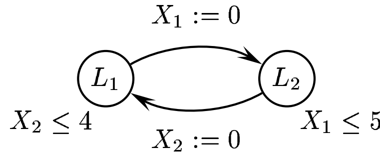
$$\delta(z).\big(z' \leq z \wedge \textstyle\bigwedge_{i=1}^{n} P_i\big),$$

where n is the number of locations, and

$$P_i = \forall z''.\big((z' \leq z'' \leq z) \Rightarrow (l_i \Rightarrow inv_i[z''/z])\big).$$

Here, we change the zero point z of all clocks to some new value z' such that $z' \leq z$, since advancing time by some amount $\delta \geq 0$ corresponds to decreasing the zero point z by δ. Furthermore, each location invariant inv_i must hold for all intermediate zero points z'' between z' and z.

Example 2. Consider the following timed automaton:



Translating this timed automaton gives the δ-program in Example 1.

2.3 Reachability

Given a transition system $(\mathcal{S}, \rightarrow)$ for a program $P = (V, C)$ and a set of states $S \subseteq \mathcal{S}$, we now define the set of states reachable from S by forward execution of a single command $c \in C$, forward execution of any command in C, and repeated forward execution of commands in C.

Definition 6 (*Post*)**.** *Let $(\mathcal{S}, \rightarrow)$ be the transition system for a program $P = (V, C)$, and let $S \subseteq \mathcal{S}$ be a set of states. The set of states reachable from S by forward execution of a command $\delta\boldsymbol{v}.\phi \in C$ is given by*

$$Post(S, \delta\boldsymbol{v}.\phi) = \{s' : \exists s \in S.\ s \xrightarrow{\delta\boldsymbol{v}.\phi} s'\}.$$

The set of states reachable from S by forward execution of any command in C is given by

$$Post(S) = \bigcup_{\delta \boldsymbol{v}.\phi \in C} Post(S, \delta \boldsymbol{v}.\phi).$$

The set of states reachable from S by repeated forward execution of any command in C is given by

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

Here, $\mu X[S \cup Post(X)]$ denotes the least fixpoint of $S \cup Post(X)$. The least fixpoint $\mu X[f(X)]$ of a functional f can be determined by computing a series of approximations $f(\emptyset), f(f(\emptyset)), \ldots$, until a fixpoint is reached [26], that is, until $f^i(\emptyset) \equiv f^{i+1}(\emptyset)$, for some i.

Analogously to *Post*, we define the corresponding sets of states that can reach S by backward execution of a single command $c \in C$, backward execution of any command in C, and repeated backward execution of commands in C.

Definition 7 (*Pre*). *Let $(\mathcal{S}, \rightarrow)$ be the transition system for a program $P = (V, C)$, and let $S \subseteq \mathcal{S}$ be a set of states. The set of states that can reach S by backward execution of a command $\delta \boldsymbol{v}.\phi \in C$ is given by*

$$Pre(S, \delta \boldsymbol{v}.\phi) = \{s : \exists s' \in S.\ s \xrightarrow{\delta \boldsymbol{v}.\phi} s'\}.$$

The set of states that can reach S by backward execution of any command in C is given by

$$Pre(S) = \bigcup_{\delta \boldsymbol{v}.\phi \in C} Pre(S, \delta \boldsymbol{v}.\phi).$$

The set of states that can reach S by repeated backward execution of any command in C is given by

$$Pre^*(S) = \mu X[S \cup Pre(X)].$$

3 Verification of Timed Systems

Given a set of states S represented by a formula ϕ, we shall now see how to construct a formula $\mathbf{post}(\phi)$ that represents the set of states reachable from S by forward execution (i.e., $Post(S)$), and a formula $\mathbf{pre}(\phi)$ that represents the set of states that can reach S by backward execution (i.e., $Pre(S)$). The simple semantics of δ-programs with only one inference rule makes it easy to specify **post** and **pre** using Boolean operations in the logic of expressions. The correctness of these symbolic reachability operators are proved in [20].

3.1 Symbolic Reachability Operators

Given an expression ϕ representing a set of states $[\![\phi]\!] \subseteq \mathcal{S}$, we construct an expression representing the set of states reachable from $[\![\phi]\!]$.

Definition 8 (post). *Let ϕ_0 be an expression, and $P = (V, C)$ a program. The* **post**-*operator for forward execution of a command $\delta \boldsymbol{v}.\phi \in C$ is given by*

$$\mathbf{post}(\phi_0, \delta \boldsymbol{v}.\phi) = \big(\exists \boldsymbol{v}.(\phi \wedge \phi_0)\big)[\boldsymbol{v}/\boldsymbol{v'}].$$

The **post**-*operator for forward execution of any command in C is defined as*

$$\mathbf{post}(\phi_0) = \textstyle\bigvee_{\delta \boldsymbol{v}.\phi \in C} \mathbf{post}(\phi_0, \delta \boldsymbol{v}.\phi).$$

The **post***-*operator for repeated forward execution of any command in C is defined as*

$$\mathbf{post}^*(\phi_0) = \mu X[\phi_0 \vee \mathbf{post}(X)],$$

where $\mu X[\phi_0 \vee \mathbf{post}(X)]$ is the least fixpoint of $\phi_0 \vee \mathbf{post}(X)$.

Theorem 1 (Correctness of post). *Let ϕ_0 be an expression, and $P = (V, C)$ a program. Then $Post([\![\phi_0]\!], \delta \boldsymbol{v}.\phi) = [\![\mathbf{post}(\phi_0, \delta \boldsymbol{v}.\phi)]\!]$ for any $\delta \boldsymbol{v}.\phi \in C$.*

Theorem 2 (Forward reachability). *Let ϕ be an expression, and $P = (V, C)$ a program with initial state ϕ_0. Then ϕ holds invariantly for P if and only if $\models \mathbf{post}^*(\phi_0) \Rightarrow \phi$.*

Similarly to the **post**-operators we define **pre**-operators for determining formulae representing the set of states that can reach $[\![\phi]\!]$.

Definition 9 (pre). *Let ϕ_0 be an expression, and $P = (V, C)$ a program. The* **pre**-*operator for backward execution of a command $\delta \boldsymbol{v}.\phi \in C$ is given by*

$$\mathbf{pre}(\phi_0, \delta \boldsymbol{v}.\phi) = \exists \boldsymbol{v'}.(\phi \wedge \phi_0[\boldsymbol{v'}/\boldsymbol{v}]).$$

The **pre**-*operator for backward execution of any command in C is defined as*

$$\mathbf{pre}(\phi_0) = \textstyle\bigvee_{\delta \boldsymbol{v}.\phi \in C} \mathbf{pre}(\phi_0, \delta \boldsymbol{v}.\phi).$$

The **pre***-*operator for repeated backward execution of any command in C is defined as*

$$\mathbf{pre}^*(\phi_0) = \mu X[\phi_0 \vee \mathbf{pre}(X)].$$

Theorem 3 (Correctness of pre). *Let ϕ_0 be an expression, and $P = (V, C)$ a program. Then $Pre([\![\phi_0]\!], \delta \boldsymbol{v}.\phi) = [\![\mathbf{pre}(\phi_0, \delta \boldsymbol{v}.\phi)]\!]$ for any $\delta \boldsymbol{v}.\phi \in C$.*

Theorem 4 (Backward reachability). *Let ϕ be an expression, and $P = (V, C)$ a program with initial state ϕ_0. Then ϕ holds invariantly for P if and only if $\not\models \mathbf{pre}^*(\neg\phi) \wedge \phi_0$.*

Consider the set of states that can reach $[\![\phi_0]\!]$ by backward execution of the command $\delta(x, \boldsymbol{b}).(\phi \wedge x' = z)$:

$$\begin{aligned}\mathbf{pre}\big(\phi_0, \delta(x, \boldsymbol{b}).(\phi \wedge x' = z)\big) &= \exists (x', \boldsymbol{b'}).\big(\phi \wedge x' = z \wedge \phi_0[(x', \boldsymbol{b'})/(x, \boldsymbol{b})]\big) \\ &= \exists \boldsymbol{b'}.\big(\phi[z/x'] \wedge \phi_0[(z, \boldsymbol{b'})/(x, \boldsymbol{b})]\big),\end{aligned}$$

using the equivalence $\exists x'.(\phi \wedge x' = z) \equiv \phi[z/x']$. Thus, resetting a clock in backward reachability analysis can be performed symbolically as a substitution instead of an existential quantification. A similar simplification can be performed for discrete state changes using the equivalence $\exists b'.\big(\phi \wedge (b' \Leftrightarrow e)\big) \equiv \phi[e/b']$. Performing quantification as substitution significantly simplifies the computational complexity of **pre**, and hence the fixpoint computation in backward reachability. Satisfiability of quantifier-free expressions is **NP**-complete [12] whereas satisfiability of quantified expressions is **PSPACE**-complete [13].

3.2 Local-Time Reachability Analysis

In this section I present a local-time model of δ-programs. The idea is to introduce a zero point z_i for each clock x_i in a δ-program, and then advance clocks asynchronously. This gives an over-approximation of the reachable state space, but in some cases it is sufficient for proving that a δ-program satisfies a given property. We do not need to change the semantics of δ-programs to advance clocks asynchronously; we simply change the command for advancing time. Without loss of generality, we can assume that the original command for advancing time has the form $\delta z.\big(z' \leq z \wedge \bigwedge_{i=1}^{n} P_i\big)$, with $P_i = \forall z''.\big((z' \leq z'' \leq z) \Rightarrow \phi_i\big)$, where ϕ_i is an expression over clock x_i and other variables, but not other clocks except x_i. Now, to make all clocks advance asynchronously, we introduce a zero point z_i for each clock x_i, $i = 1, \ldots, n$, and then perform the following substitutions of sub-expressions in commands: replace $x_i - z \sim d$ by $x_i - z_i \sim d$, replace $z' - z'' \sim d$ by $z'_i - z''_i \sim d$, and replace $z'' - z \sim d$ by $z''_i - z_i \sim d$.

Example 3. Consider again the δ-program from Example 1, and let us introduce a zero point for each of the two clocks:

$$
\begin{aligned}
&\delta(l_1, l_2, x_1).(l_1 \wedge \neg l'_1 \wedge l'_2 \wedge x'_1 = z_1) \\
&\delta(l_1, l_2, x_2).(l_2 \wedge \neg l'_2 \wedge l'_1 \wedge x'_2 = z_2) \\
&\delta(z_1, z_2).\big(z'_1 \leq z_1 \wedge \forall z''_1(z'_1 \leq z''_1 \leq z_1 \Rightarrow (l_2 \Rightarrow x_1 - z''_1 \leq 5)) \wedge \\
&\qquad\qquad z'_2 \leq z_2 \wedge \forall z''_2(z'_2 \leq z''_2 \leq z_2 \Rightarrow (l_1 \Rightarrow x_2 - z''_2 \leq 4))\big)
\end{aligned}
$$

The two first commands are modified so that x_1 and x_2 are relative to z_1 and z_2, respectively. The third command is modified so that x_1 advances time by changing the zero point z_1, and x_2 advances time by changing the zero point z_2.

The following theorem shows that when advancing clocks asynchronously in a δ-program, the **post**-operator gives an over-approximation of *Post*. A similar result holds for the **pre**-operator and *Pre*. To simplify the exposition, we introduce the following notation: Let ϕ be an expression over variables $v_1, \ldots, v_n$. Then ϕ defines an n-ary predicate $\phi(v_1, \ldots, v_n)$ with parameters $(v_1, \ldots, v_n)$. We write $\phi(\boldsymbol{v_1}, \ldots, \boldsymbol{v_n})$ for the predicate $\phi(v_{11}, \ldots, v_{1l}, \ldots, v_{n1}, \ldots, v_{nm})$, where $\boldsymbol{v_i}$ are vectors.

Theorem 5 (Local-time reachability). *Let $\phi(\boldsymbol{z}, \boldsymbol{z'}, \boldsymbol{u})$ be a predicate with $\boldsymbol{z} = (z_1, \ldots, z_n)$ and $\boldsymbol{z'} = (z'_1, \ldots, z'_n)$. Let $\phi_0(\boldsymbol{z}, \boldsymbol{v})$ be a predicate, and let $\boldsymbol{w} =$*

$(w, \ldots, w)$ *and* $\boldsymbol{w'} = (w', \ldots, w')$ *be* n*-dimensional vectors. Let*

$$\phi_{\text{global}} = \mathbf{post}(\phi_0(\boldsymbol{w}, \boldsymbol{v}), \delta w.\phi(\boldsymbol{w}, \boldsymbol{w'}, \boldsymbol{u})),$$
$$\phi_{\text{local}} = \mathbf{post}(\phi_0(\boldsymbol{z}, \boldsymbol{v}), \delta \boldsymbol{z}.\phi(\boldsymbol{z}, \boldsymbol{z'}, \boldsymbol{u}))[\boldsymbol{w}/\boldsymbol{z}].$$

Then $\models \phi_{\text{global}} \Rightarrow \phi_{\text{local}}$.

Proof. From the definition of **post** we get:

$$\begin{aligned}\phi_{\text{global}} &= \Big(\exists w.\big(\phi(\boldsymbol{w}, \boldsymbol{w'}, \boldsymbol{u}) \wedge \phi_0(\boldsymbol{w}, \boldsymbol{v})\big)\Big)[w/w'] \\ &= \exists w'.\big(\phi(\boldsymbol{w'}, \boldsymbol{w}, \boldsymbol{u}) \wedge \phi_0(\boldsymbol{w'}, \boldsymbol{v})\big),\end{aligned}$$

and

$$\begin{aligned}\phi_{\text{local}} &= \Big(\exists \boldsymbol{z}.\big(\phi(\boldsymbol{z}, \boldsymbol{z'}, \boldsymbol{u}) \wedge \phi_0(\boldsymbol{z}, \boldsymbol{v})\big)\Big)[\boldsymbol{z}/\boldsymbol{z'}][\boldsymbol{w}/\boldsymbol{z}] \\ &= \Big(\exists \boldsymbol{z}.\big(\phi(\boldsymbol{z}, \boldsymbol{z'}, \boldsymbol{u}) \wedge \phi_0(\boldsymbol{z}, \boldsymbol{v})\big)\Big)[\boldsymbol{w}/\boldsymbol{z'}] \\ &= \exists \boldsymbol{z}.\big(\phi(\boldsymbol{z}, \boldsymbol{w}, \boldsymbol{u}) \wedge \phi_0(\boldsymbol{z}, \boldsymbol{v})\big).\end{aligned}$$

Clearly, if there exists a w' such that $\phi(\boldsymbol{w'}, \boldsymbol{w}, \boldsymbol{u}) \wedge \phi_0(\boldsymbol{w'}, \boldsymbol{v})$ is true, then there also exists a $\boldsymbol{z}$, namely $\boldsymbol{z} = \boldsymbol{w'}$, such that $\phi(\boldsymbol{z}, \boldsymbol{w}, \boldsymbol{u}) \wedge \phi_0(\boldsymbol{z}, \boldsymbol{v})$ is true. Thus, ϕ_{local} follows logically from ϕ_{global}. □

That is, ϕ_{global} represents the set of states reachable from $\phi_0(\boldsymbol{z}, \boldsymbol{v})$ by the command $\delta w.\phi(\boldsymbol{z}, \boldsymbol{z'}, \boldsymbol{u})$, where we replace each occurrence of z_i in ϕ_0 and ϕ by w before the call to **post**. Thus, ϕ_{global} corresponds to advancing all clocks synchronously using the zero point w. Analogously, ϕ_{local} represents the set of states reachable from $\phi_0(\boldsymbol{z}, \boldsymbol{v})$ by the command $\delta \boldsymbol{z}.\phi(\boldsymbol{z}, \boldsymbol{z'}, \boldsymbol{u})$, where we replace each occurrence of z_i by w after the call to **post**. Thus, ϕ_{local} corresponds to first advancing all clocks asynchronously using n zero points $z_1, \ldots, z_n$, and then unifying these zero points to w.

4 Experimental Results

I have developed a symbolic model checker for δ-programs using an implementation of difference decision diagrams (DDDs) [22] called DDDLIB [23]. This section reports experimental results for verification of some real-time case studies. The reported CPU times are in seconds, and the DDD sizes are in vertices (each 28 bytes). The results were obtained on a 1 GHz Pentium III with 2 GB of memory running Linux.

4.1 Fischer's Mutual Exclusion Protocol

Fischer's mutual exclusion protocol [14] consists of N processes competing for a shared resource. Each process can be in one of four states encoded using two Boolean variables:

$$idle_i = \neg b_i^1 \wedge \neg b_i^2, \quad rdy_i = \neg b_i^1 \wedge b_i^2, \quad wait_i = b_i^1 \wedge \neg b_i^2, \quad crit_i = b_i^1 \wedge b_i^2.$$

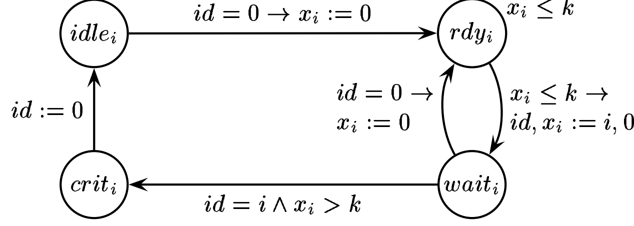


Fig. 1. Timed automaton for process i in Fischer's protocol.

The processes use a shared integer variable in the range $[0, N]$ for controlling the access to the shared resource. For simplicity, we encode this variable using N Boolean variables $id_1, \ldots, id_N$. The commands for process i are (see also Fig. 1):

$$\begin{aligned}
&\delta(b_i^1, b_i^2, x_i).\big((idle_i \vee wait_i) \wedge rdy_i' \wedge x_i' = z \wedge \textstyle\bigwedge_{j=1}^{N} \neg id_j\big)\\
&\delta(b_i^1, b_i^2, x_i, id_i).\big(rdy_i \wedge wait_i' \wedge x_i - z \le k \wedge x_i' = z \wedge id_i'\big)\\
&\delta(b_i^1, b_i^2).\big(wait_i \wedge crit_i' \wedge x_i - z > k \wedge id_i \wedge \textstyle\bigwedge_{j \ne i} \neg id_j\big)\\
&\delta(b_i^1, b_i^2, id_i).\big(crit_i \wedge idle_i' \wedge \neg id_i'\big)
\end{aligned}$$

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.[1] The command for advancing time is:

$$\delta(z).\Big(z' \le z \wedge \textstyle\bigwedge_{i=1}^{N} \big(\forall z''(z' \le z'' \le z) \Rightarrow (rdy_i \Rightarrow 0 \le x_i - z'' \le k)\big)\Big).$$

The initial state is given by $\phi_0 = \bigwedge_{i=1}^{N} \big(idle_i \wedge \neg id_i \wedge x_i = z\big)$, and the property that there is only one process in the critical state at a time can be expressed as $\phi = \bigwedge_{i=1}^{N} \bigwedge_{j \ne i} \neg\big(crit_i \wedge crit_j\big)$.

Fischer's protocol guarantees mutual exclusion if and only if $\models \mathbf{post}^*(\phi_0) \Rightarrow \phi$ (forward reachability), or $\not\models \mathbf{pre}^*(\neg\phi) \wedge \phi_0$ (backward reachability). I have verified Fischer's protocol for increasing number of processes N using forward, backward, and local-time backward reachability analysis, see Table 1. The results for forward reachability are comparable with those obtained with UPPAAL [18] and KRONOS [29] using the default options. As expected, backward reachability analysis is faster and more space efficient than forward reachability analysis. For local-time backward reachability, it turns out that it is possible to prove mutual exclusion for Fischer's protocol using the over-approximation. The runtimes for local-time backward reachability are linear in N.

4.2 Alpha and Beta Examples

The alpha and beta examples [6] are two very simple real-time systems which, despite their simplicity, are hard to analyze. The alpha example consists of N timed automata, each containing one state and one transition. Each transition

[1] The runtimes of Fischer's protocol are not affected by the size of k when using DDDs.

	Forward		Backward		Local-time backward	
N	CPU time	DDD size	CPU time	DDD size	CPU time	DDD size
2	1.1	36	1.0	16	1.0	16
4	1.2	405	1.1	259	1.1	139
6	10.6	4,478	1.3	979	1.2	275
8	2138.0	50,291	2.3	3,383	1.4	411
10			6.3	12,331	1.5	547
12			27.8	47,263	1.6	683
14			589.2	185,939	1.7	819
16			8521.5	739,399	2.2	955
32					6.3	2,043
64					27.0	4,219
128					161.5	8,571
256					1318.0	17,275
512					5602.0	34,683

Table 1. Experimental results for verification of Fischer's protocol with N processes using forward, backward, and local-time backward reachability analysis.

resets a clock x_i, and must be taken in the interval $[l, u]$ since the clock was last reset. The alpha example can be modeled as a δ-program with N commands of the form:

$$\delta(b_i, x_i).(b_i \wedge b'_i \wedge x_i - z \geq l \wedge x'_i - z = 0)$$

and a command for advancing time of the form:

$$\delta(z).\Big(z' \leq z \wedge \bigwedge_{i=1}^{N} \left(\forall z''(z' \leq z'' \leq z) \Rightarrow (b_i \Rightarrow 0 \leq x_i - z'' \leq u)\right)\Big).$$

The initial state is given by $\phi_0 = \bigwedge_{i=1}^{N} \left(b_i \wedge x_i = z\right)$.

The beta example consists of N timed automata, each containing two states and two transitions similar to those of the alpha example. The beta example can be modeled as N commands of the form:

$$\delta(b_i, x_i).(\neg(b_i \Leftrightarrow b'_i) \wedge x_i - z \geq l \wedge x'_i - z = 0)$$

and a command for advancing time of the form:

$$\delta(z).\Big(z' \leq z \wedge \bigwedge_{i=1}^{N} \left(\forall z''(z' \leq z'' \leq z) \Rightarrow (0 \leq x_i - z'' \leq u)\right)\Big).$$

I have computed the reachable set of states for the alpha and beta examples using forward reachability analysis, and local-time forward reachability analysis. The results are shown in Table 2. Using forward reachability analysis it is only possible to compute the reachable state spaces for the two examples with up to $N = 4$ within one hour when using difference decision diagrams. This should be compared with the results of Bozga et al. [6] which were able to compute the reachable state spaces for systems with up to $N = 18$ timed automata using an implementation based on numerical decision diagrams.

	Alpha		Beta	
N	CPU time	DDD size	CPU time	DDD size
16	0.1	50	0.1	34
32	0.2	98	0.2	66
64	0.4	194	0.6	130
128	1.9	386	2.8	258
256	9.8	770	16.3	514
512	49.1	1538	74.4	1026
1024	212.0	3074	317.1	2050

Table 2. Experimental results for computation of the reachable set of states for the alpha and beta examples with N processes using local-time forward reachability analysis.

Using local-time forward reachability, it turns out that the computed reachable state spaces for the alpha and beta examples are exact, and not over-approximations. The reachable state space is $\bigwedge_{i=1}^{N} (b_i \wedge 0 \leq x_i \leq u)$ for alpha, and $\bigwedge_{i=1}^{N} (0 \leq x_i \leq u)$ for beta. As for Fischer's protocol, the runtimes are linear in N.

5 Conclusion

Analysis of timed systems is extremely difficult, and most current verification tools can only handle moderate-sized systems with up to a few tens of clocks and a few hundred thousand discrete states. Some of the problems are how to efficiently perform the basic verification operations, **post** and **pre**, to compute the reachable states in a forward or backward manner; and how to efficiently represent the infinite state space of a timed system, including how to avoid the state explosion problem for the discrete part.

I have introduced δ-programs as a uniform notation for modeling timed systems. Time is modeled explicitly as a variable in the program which makes it easy to define a transitional semantics and corresponding symbolic forward (**post**) and backward (**pre**) reachability operators. I have showed that the **pre**-operator can be simplified so that resetting of clocks and discrete state changes are performed as substitutions instead of quantifications. Introducing time on the syntactic and not semantic level makes it easy to experiment with different models of time (e.g., strictly versus weakly monotonic time, or local time with several zero points as demonstrated in this paper), or even omitting time in the initial phase of the modeling.

I have presented a local-time model for δ-programs which gives an over-approximation of the correct set of reachable state. The initial experiments with this approach look promising: Using this local-time model makes it possible to verify Fischer's protocol in linear time, and to compute in linear time the exact reachable state space for the alpha and beta examples defined in [6]. However,

there is no guarantee that the proposed local-time model will always work as well as in the studied examples. An over-approximation might become too large and hence not useful. On a simple model of a pulse-generating circuit described in [21], the local-time model described in this paper gives an over-approximation which is too large to prove correctness of the circuit. The reason is that the correctness of this circuit relies heavily on the correct timing between its sub-components, and these timing issues are to some extent ignored in the given local-time model. A solution might be to compute a better over-approximation of the state space by letting clocks of related components share a common zero point. Another direction for future research is to develop more efficient data structures and algorithms for representing and manipulating formulae in the fixpoint computation.

References

1. R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.
2. R. Alur and D. Dill. The theory of timed automata. In *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 28–73. Springer-Verlag, 1991.
3. R. Alur, T. Feder, and T.A. Henzinger. The benefits of relaxing punctuality. In *Proc. 10th Annual Symposium on Principles of Distributed Computing*, pages 139–152. ACM Press, 1991.
4. 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.
5. A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly symbolic model-checking for real-time systems. In *Proc. 18th IEEE Real-Time Systems Symposium*, pages 25–35, San Francisco, USA, December 1997. IEEE Computer Scienty Press.
6. 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.
7. R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
8. 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.
9. 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.
10. P. Godefroid. Using partial orders to improve automatic verification methods. In E. M. Clarke and R. P. Kurshan, editors, *Proc. Second International Conference on Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185. Springer-Verlag, 1990.
11. T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In *Proc. Seventh IEEE Symposium of Logics in Computer Science*, pages 394–406, Santa-Cruz, California, 1992. IEEE Computer Scienty Press.

12. 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.
13. 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.
14. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.
15. K.G. Larsen, J. Pearson, C. Weise, and W. Yi. Clock difference diagrams. *Nordic Journal of Computing*, 6(3):271–298, 1999.
16. K.G. Larsen, P. Pettersson, and W. Yi. Compositional and symbolic model-checking of real-time systems. In *Proc. 16th Annual Real-time Systems Symposium*, pages 76–89. IEEE Computer Society Press, 1995.
17. 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.
18. 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.
19. M. Minea. *Partial Order Reduction for Verification of Timed Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, December 1999. CMU-CS-00-102.
20. J. Møller, H. Hulgaard, and H.R. Andersen. Symbolic model checking of timed guarded commands using difference decision diagrams. *Journal of Logic and Algebraic Programming*, 52(1–2), 2002.
21. J. Møller, H. Hulgaard, and H.R. Andersen. Timed verification of asynchronous circuits. In G. Rozenberg J. Cortadella, A. Yakovlev, editor, *Advances in Concurrency and Hardware Design*. Springer-Verlag, 2002.
22. 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.
23. J.B. Møller. DDDLIB: A library for solving quantified difference inequalities. In A. Voronkov, editor, *Proc. 18th International Conference on Automated Deduction*, Lecture Notes in Computer Science, Copenhagen, Denmark, 27–30 July 2002. Springer-Verlag.
24. T.G. Rokicki. *Representing and Modeling Digital Circuits.* PhD thesis, Stanford University, 1993.
25. M. Sorea. TEMPO: A model-checker for event-recording automata. In *Proc. Workshop on Real-Time Tools*, August 2001. Also as SRI Technical Report CSL-01-04.
26. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
27. F. Wang. Symbolic verification of complex real-time systems with clock-restriction diagram. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Proc. International Conference on Formal Techniques for Networked and Distributed Systems*, volume 197 of *IFIP Conference Proceedings*, pages 235–250. Kluwer, August 2001.
28. P.F. Williams, A. Biere, E.M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proc. 12th International Conference on Computer-Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2000.
29. 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.