Efficient Verification Of Timed Systems Using Backward Reachability Analysis

Jesper Møller

IT University Technical Report Series **TR-2002-11**

ISSN 1600–6100 **February 2002**

ISSN 1600–6100

ISBN 87-7949-015-8

Copies may be obtained by contacting:

IT University of Copenhagen
Glentevej 67
DK-2400 Copenhagen NV
Denmark

Telephone:	+45 38 16 88 88
Telefax:	+45 38 16 88 99
Web	`www.it.edu`

Efficient Verification Of Timed Systems Using Backward Reachability Analysis

Jesper Møller

Department of Innovation, The IT University of Copenhagen
Glentevej 67, 2400 Copenhagen NV, Denmark
Email: jm@it.edu

Abstract. In this paper we demonstrate that symbolic verification of real-time systems based on backward reachability analysis is generally more efficient than verification based on forward reachability analysis. Symbolic verification of real-time systems consists of computing the least fixpoint of a functional that given a set of states φ returns the states that are reachable from φ (forward reachability), or that can reach φ (backward reachability). The key observation is that the symbolic operations in the fixpoint computation can be simplified substantially when performing backward reachability analysis. In particular, we show that resetting clocks and making discrete state changes can be performed as substitutions instead of existential quantifications over reals and Booleans, respectively. Experimentally we find that backward reachability analysis is more efficient than forward reachability analysis.

1 Introduction

Formal verification based on reachability analysis of finite state systems [9] is today used extensively in the development of digital circuits and embedded software. 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 [8]. However, this symbolic technique cannot easily be adopted to models with real-valued variables such as timed systems. One problem is how to represent the infinite state space R of a timed system; another problem is how to perform the basic verification operations (resetting clocks, advancing time, etc.) on this representation to compute the reachable state space.

We demonstrate that real-time verification based on backward reachability (computing the set of states that can reach a state) is generally more efficient than forward reachability (computing the set of states that are reachable from a state). In forward reachability, which is used by many verification tools (e.g., [5, 27, 18, 29]), resetting a clock x in a set of states represented by the formula φ can be performed as $(\exists x.\varphi) \wedge x = 0$. When analysing the system backwards, resetting x can be performed as $\exists x.\big(\varphi \wedge (x = 0)\big) \equiv \varphi[0/x]$. 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 $\exists b.\big(\varphi \wedge (b \Leftrightarrow e)\big) \equiv \varphi[e/b]$. Williams et al. [31] 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 from **NP**-complete to **PSPACE**-complete.

We present a simple notation called δ-programs for modeling timed systems. A δ-program is a set of commands of the form $\delta\boldsymbol{v}.\varphi$, 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 Dijkstra's guarded commands [11] and Henzinger's timed guarded commands [13], except for two main differences. First, assignments are expressed as predicates over primed and unprimed variables. Second, we model time explicitly as a program variable (called z in this paper), which is interpreted as the common zero-point of all clocks. A command for updating time can be specified as:

$$\delta(z).\big(z' \leq z \wedge \forall z''(z' \leq z'' \leq z \Rightarrow inv[z''/z])\big).$$

where *inv* is a predicate expressing whether z is a legal zero-point. Introducing time explicitly in the notation makes the semantics of a δ-program 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. We show how to express these reachability operators as formulae in a first-order logic, and how to perform quantification by substitution in backward analysis. Next, we discuss the complexity of this logic, and briefly introduce a data structure for implementing it. We use this data structure to analyse Fischer's mutual exclusion protocol, and show that backward reachability analysis is more efficient than the forward ditto.

2 Modeling Timed Systems

δ-programs are a simple notation for modeling real-time systems similar to timed guarded commands [13] 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 [3] and timed Petri nets [5].

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:*

$$\varphi ::= b \mid x \sim d \mid x - y \sim d \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \exists b.\varphi \mid \exists x.\varphi\,,$$

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 $\varphi \in \Phi$ is an expression.

We use the tokens **false** and **true** to denote false and true expressions, respectively. The symbols $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (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 $\varphi[\boldsymbol{v}/\boldsymbol{v'}]$, as the syntactic substitution of all occurrences of v_i' in φ by v_i, for $i = 1, \ldots, n$. We also 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 $\varphi[\boldsymbol{v} := \boldsymbol{v'}] = (\exists \boldsymbol{v}.\varphi)[\boldsymbol{v}/\boldsymbol{v'}]$. 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 $s \models \varphi$, if φ evaluates to true in the state s, and we write $[\![\varphi]\!]$ for the set of states that satisfy φ. If any state satisfies φ, φ is a tautology and we write $\models \varphi$. 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 $\delta\boldsymbol{v}.\varphi$.*

A command $\delta(v_1, \ldots, v_n).\varphi$ 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. [1] 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 $s \models \varphi$.

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 $c \in C$ of the form $\delta\boldsymbol{v}.\varphi$ there is a transition $\xrightarrow{c}$ defined by the inference rule:*

$$\frac{s[\boldsymbol{v'} := \boldsymbol{r}\,] \models \varphi}{s \xrightarrow{c} s[\boldsymbol{v} := \boldsymbol{r}\,]}\,.$$

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:

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

The first command assigns to l_1, l_2, x_2 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 Encoding Timed Automata

We now show how to translate a timed automaton [3] 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. In [24] we showed that by making all clocks in a timed system relative to a variable z, it is possible to synchronously advance time in a set of states represented by an expression φ as an existential quantification over z in φ. We do not prove the correctness of the translation here, but refer the reader to [22].

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^z 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^z 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:

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

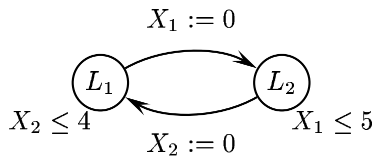
Finally, we add the following command for advancing time:

$$\delta(z).\Big(z' \le z \wedge \forall z''\big((z' \le z'' \le z) \Rightarrow \textstyle\bigwedge_{i=1}^{n}(l_i \Rightarrow inv_i^z[z''/z])\big)\Big),$$

[1] An alternative to this independent-choice strategy is a fixed-choice strategy. In a fixed-choice strategy, if two expressions φ_1 and φ_2 are equivalent then the values bound to the variables $\boldsymbol{u}$ and $\boldsymbol{v}$ in the commands $\delta\boldsymbol{u}.\varphi_1$ and $\delta\boldsymbol{v}.\varphi_1$ are identical. This fixed-choice operator is also known as Hilbert's ϵ-operator [14]. See [6] for a more detailed discussion on fixed-choice and independent-choice logics.

where n is the number of locations. 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^z 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 $c \in C$ is given by*

$$Post(S, c) = \{s' : \exists s \in S.\ s \xrightarrow{c} s'\}.$$

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

$$Post(S) = \textstyle\bigcup_{c \in C} Post(S, c).$$

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 [30], 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 $c \in C$ is given by*

$$Pre(S, c) = \{s : \exists s' \in S.\ s \xrightarrow{c} s'\}.$$

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

$$Pre(S) = \textstyle\bigcup_{c \in C} Pre(S, c).$$

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

We now describe how to verify properties of δ-programs using reachability analysis. Given a set of states S represented by a formula φ, we define how to construct a formula $\mathbf{post}(\varphi)$ that represents the set of states reachable from S by forward execution (i.e., $\mathit{Post}(S)$), and a formula $\mathbf{pre}(\varphi)$ that represents the set of states that can reach S by backward execution (i.e., $\mathit{Pre}(S)$). 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.

3.1 Forward Reachability Analysis

Given an expression φ representing a set of states $[\![\varphi]\!] \subseteq \mathcal{S}$, we now show how to determine an expression representing the set of states reachable from $[\![\varphi]\!]$.

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

$$\mathbf{post}(\varphi_0, \delta\boldsymbol{v}.\varphi) = \big(\exists\boldsymbol{v}.(\varphi \wedge \varphi_0)\big)[\boldsymbol{v}/\boldsymbol{v}'].$$

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

$$\mathbf{post}(\varphi_0) = \textstyle\bigvee_{c \in C} \mathbf{post}(\varphi_0, c).$$

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

$$\mathbf{post}^*(\varphi_0) = \mu X[\varphi_0 \vee \mathbf{post}(X)],$$

where $\mu X[\varphi_0 \vee \mathbf{post}(X)]$ is the least fixpoint of $\varphi_0 \vee \mathbf{post}(X)$.

Theorem 1 (Correctness of post). *Let φ_0 be an expression, and $P = (V, C)$ a program. Then $\mathit{Post}([\![\varphi_0]\!], c) = [\![\mathbf{post}(\varphi_0, c)]\!]$ for any $c \in C$.*

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 $\models \mathbf{post}^*(\varphi_0) \Rightarrow \varphi$.*

3.2 Backward Reachability Analysis

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

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

$$\mathbf{pre}(\varphi_0, \delta\boldsymbol{v}.\varphi) = \exists\boldsymbol{v}'.(\varphi \wedge \varphi_0[\boldsymbol{v}'/\boldsymbol{v}]).$$

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

$$\mathbf{pre}(\varphi_0) = \textstyle\bigvee_{c \in C} \mathbf{pre}(\varphi_0, c).$$

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

$$\mathbf{pre}^*(\varphi_0) = \mu X[\varphi_0 \vee \mathbf{pre}(X)].$$

Theorem 3 (Correctness of pre). *Let φ_0 be an expression, and $P = (V, C)$ a program. Then $\mathit{Pre}([\![\varphi_0]\!], c) = [\![\mathbf{pre}(\varphi_0, c)]\!]$ for any $c \in C$.*

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 $\not\models \mathbf{pre}^*(\neg\varphi) \wedge \varphi_0$.*

3.3 Comparison Between Forward and Backward Reachability

Consider the set of states reachable from $[\![\varphi_0]\!]$ by forward execution of the command $\delta(x, \boldsymbol{b}).(\varphi \wedge x' = z)$ is

$$\mathbf{post}\big(\varphi_0, \delta(x, \boldsymbol{b}).(\varphi \wedge x' = z)\big) = \big(\exists(x, \boldsymbol{b}).(\varphi \wedge x' = z \wedge \varphi_0)\big)[(x, \boldsymbol{b})/(x', \boldsymbol{b'})].$$

This expression cannot be simplified further without more information about φ and φ_0. In backward analysis, however, the set of states that can reach $[\![\varphi_0]\!]$ is

$$\begin{aligned}\mathbf{pre}\big(\varphi_0, \delta(x, \boldsymbol{b}).(\varphi \wedge x' = z)\big) &= \exists(x', \boldsymbol{b'}).\big(\varphi \wedge x' = z \wedge \varphi_0[(x', \boldsymbol{b'})/(x, \boldsymbol{b})]\big)\\ &= \exists \boldsymbol{b'}.\big(\varphi[z/x'] \wedge \varphi_0[(z, \boldsymbol{b'})/(x, \boldsymbol{b})]\big),\end{aligned}$$

using the equivalence $\exists x'.(\varphi \wedge x' = z) \equiv \varphi[z/x']$. 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 $\exists b'.\big(\varphi \wedge (b' \Leftrightarrow e)\big) \equiv \varphi[e/b']$. The computation of $\mathbf{pre}(\varphi_0, c)$ for a command c corresponding to a transition $(L_i, \boldsymbol{x}, G_i, L_j)$ in a timed automaton can be simplified so that all existential quantifications are performed as substitutions:

$$\begin{aligned}\mathbf{pre}(\varphi_0, c) &= \exists(l'_i, l'_j, \boldsymbol{x'}).\big(g_i^z \wedge l_i \wedge \neg l'_i \wedge l'_j \wedge \boldsymbol{x'} = \boldsymbol{z} \wedge \varphi_0[(l'_i, l'_j, \boldsymbol{x'})/(l_i, l_j, \boldsymbol{x})]\big)\\ &= g_i^z \wedge l_i \wedge \varphi_0[(\mathbf{false}, \mathbf{true}, \boldsymbol{z})/(l_i, l_j, \boldsymbol{x})].\end{aligned}$$

Performing quantification as substitution significantly simplifies the computational complexity of **pre**, and hence the fixpoint computation in backward reachability. As we shall see in the following section, satisfiability of quantifier-free expressions is **NP**-complete whereas satisfiability of quantified expressions is **PSPACE**-complete.

4 Algorithms and Data Structures

The core operation in verification of real-time systems is to determine whether an expression φ is satisfiable. This problem, which we call TIMED-SAT in the following, has only been studied by very few researchers; the primary focus has been on theories that are either more expressive (e.g., integers or reals with addition and order), or less expressive (e.g., quantified Boolean formulae), 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.

4.1 Complexity Overview

Let us first look at the satisfiability problem for two subsets of expressions. The first problem is to determine satisfiability of expressions of the form

$$\varphi ::= b \mid x \sim d \mid x - y \leq d \mid \varphi_1 \wedge \varphi_2 \,.$$

This problem is in the complexity class **P** and can be solved in time $O(n^3)$ where n is the number of variables using for example Floyd-Warshall's algorithm [10]. The second problem is to determine satisfiability of expressions of the form

$$\varphi ::= b \mid x \sim d \mid x - y \leq d \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \,.$$

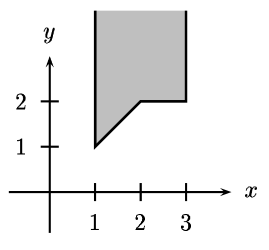
This problem is **NP**-complete, see for instance [13] for a proof. The TIMED-SAT problem is to determine satisfiability of expressions of the form

$$\varphi ::= b \mid x \sim d \mid x - y \leq d \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \exists b.\varphi \mid \exists x.\varphi \,.$$

Interestingly, it turns out that adding quantifiers "only" makes the problem **PSPACE**-complete [16]. Considering that we are working with variables over infinite domains, it is surprising that TIMED-SAT is no harder than satisfiability of quantified Boolean formulae [26]. TIMED-SAT is also easier than satisfiability of slightly more general theories, such as the first-order theories of reals with addition and order, which is **NEXP**-hard [15], and integers with addition and order, which is **2-NEXP**-hard [28].

4.2 Difference Decision Diagrams

Difference decision diagrams (DDDs) [23] are a data structure for representing and deciding satisfiability of expressions. Similar to how a BDD [7] represents the meaning of a Boolean formula implicitly, a DDD represents the meaning $[\![\varphi]\!]$ of an expression φ using a decision diagram. A DDD is a directed acyclic graph with two terminals, **0** and **1**, and a set of non-terminal vertices. Each non-terminal vertex corresponds to the if-then-else operator $\alpha \to \varphi_1, \varphi_0$, defined as $(\alpha \wedge \varphi_1) \vee (\neg\alpha \wedge \varphi_0)$, where the test expression α is a Boolean variable or a difference constraint of the form $x - y < d$ or $x - y \leq d$, and the high-branch φ_1 and low-branch φ_0 are DDD vertices. Each DDD vertex v denotes an expression φ^v given by $\varphi^v = \alpha(v) \to \varphi^{high(v)}, \varphi^{low(v)}$, where $\alpha(v)$ is the test expression of v, and *high*(v) and *low*(v) are the high- and low-branches, respectively. Figure 1 shows an example of a DDD.



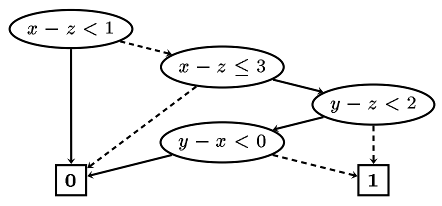


Fig. 1. The expression $\varphi = (1 \leq x - z \leq 3) \wedge \big((y - z \geq 2) \vee (y - x \geq 0)\big)$ as (a) an (x, y)-plot for $z = 0$, and (b) a difference decision diagram.

As shown in [23], DDDs can be ordered and path-reduced, yielding a semicanonical form, which makes it possible to check for validity and satisfiability in constant time (as for BDDs). The DDD data structure is not canonical, however, so equivalence checking must be performed as a validity check. The operations for constructing and manipulating DDDs according to the definition of expressions are easily defined recursively on the DDD data structure. The function APPLY(op, u, v) combines two ordered DDDs rooted at u and v with a Boolean operator *op* (e.g., the negation and conjunction operations in Definition 2). APPLY is a generalization of the version used for BDDs [7] 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 existentially quantify the variable x in a DDD rooted at u. The algorithm is an adoption of the Fourier-Motzkin quantifier-elimination method [12], removing all vertices reachable from u containing x, but keeping all implicit constraints induced by x among the other variables (e.g., $\exists x.(z - x < 1 \wedge x - y \leq 0)$ is equivalent to $z - y < 1$). EXISTS computes modified and additional constraints in polynomial time, but has an exponential worst-case running time since the resulting DDD must be ordered.

The function PATHREDUCE(u) removes all redundant vertices in a DDD rooted at u, making it semi-canonical (i.e., valid expressions are represented by **1**, unsatisfiable expressions by **0**). PATHREDUCE determines path feasibility using an incremental Bellman-Ford algorithm [25] with dynamic programming, but has exponential worst-case running time.

The DDD data structure is available as a C/C++ library [25], which has been implemented in the TEMPO model checker for event-recording automata [29], and in a model checker for timed Petri nets [1, 2]. Applications of DDDs in partial order reduction and local-time model checking are discussed in the PhD thesis of Minea [21].

5 Experimental Results

In this section we demonstrate that backward reachability is more efficient that forward reachability by analysing Fischer's mutual exclusion protocol [17] with a DDD-based model checker. Fischer's protocol consists of N processes competing for a shared resource. Each process can be in one of four states encoded using two Boolean variables:

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

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. 2):

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

The parameter k is a constant which determines how long a process waits until entering the critical state. We use $k = 10$ in the following.[2] The command for advancing time is:

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

The initial state is given by

$$\varphi_0 = \bigwedge_{i=1}^{N} \big(idle_i \wedge \neg id_i \wedge x_i = z\big).$$

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

$$\varphi = \bigwedge_{i=1}^{N} \bigwedge_{j \neq i} \neg\big(crit_i \wedge crit_j\big).$$

Fischer's protocol guarantees mutual exclusion if and only if $\models \mathbf{post}^*(\varphi_0) \Rightarrow \varphi$ (forward reachability), or $\not\models \mathbf{pre}^*(\neg\varphi) \wedge \varphi_0$ (backward reachability) . We have verified Fischer's protocol for increasing number N of processes using forward reachability, forward reachability with path reduction in the fixpoint computation, and backward reachability analysis.[3] The results are shown in Table 1. As expected, backward reachability analysis is faster and more space efficient than forward reachability analysis. The results for forward reachability are comparable with those obtained with for example UPPAAL [19] and KRONOS [32] using the default options.

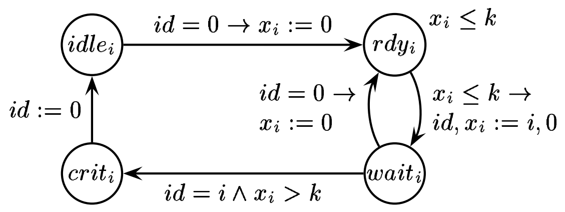


Fig. 2. Timed automaton for process i in Fischer's protocol.

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

[3] It turns out that path reduction does not gives space savings in backward analysis. One reason for this is that most redundant vertices are created by existential quantification, which are avoided in backward analysis.

	Forward		Forward, reduced		Backward	
N	CPU time	DDD size	CPU time	DDD size	CPU time	DDD size
1	0.9	9	0.9	9	0.8	1
2	1.0	42	1.1	36	1.0	16
3	1.2	189	1.1	122	1.1	103
4	1.4	1,082	1.2	405	1.1	259
5	2.3	7,673	2.1	1,346	1.2	521
6	13.2	61,438	10.6	4,478	1.3	979
7	148.1	515,823	122.0	14,903	1.7	1,813
8	4044.0	4,446,637	2138.0	50,291	2.3	3,383
9					3.5	6,409
10					6.3	12,331
11					12.4	24,029
12					27.8	47,263
13					65.1	93,553
14					589.2	185,939
15					2021.7	370,504
16					8521.5	739,399

Table 1. Experimental results for verification of Fischer's protocol with N processes using forward, forward with path reduction, and backward reachability analysis. The 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.

6 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.

We 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. We 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 also makes it easier to experiment with different models of time (e.g., strictly versus weakly monotonic time, or local time with several zero points), or even omitting time in the initial phase of the modeling.

We have briefly introduced a data structure (DDDs) for computing **post** and **pre**, and although DDDs do not work as efficiently for timed systems as BDDs do for untimed systems, they do enable us experiment and compare the two techniques for reachability analysis. The fixpoint characterizations of **post*** and **pre*** make it trivially simple to implement and validate a reachability analysis tool for δ-programs using the DDD-library. The experimental results for Fischer's protocol show that backward reachability is more efficient that forward reachability.

There are a number of ways to further improve the results in this paper. One avenue is to extend the reachability algorithms to support partial order reduction as described in [4, 20], and local-time models [21]. Another avenue is to improve the quantifier-elimination algorithm for DDDs.

References

1. Parosh Aziz Abdulla and Aletta Nylén. Better is better than well: On efficient verification of infinite-state systems. In *Proc. 15th IEEE Symposium on Logic in Computer Science (LICS)*, pages 132–140,

Santa Barbara, June 2000. IEEE Computer Society Press.

2. Parosh Aziz Abdulla and Aletta Nylén. Timed Petri nets and BQOs. In *Proc. 22nd International Conference on application and theory of Petri nets (ICATPN)*, volume 2075 of *LNCS*, pages 53–70, 2001.
3. R. Alur and D. Dill. The theory of timed automata. In *Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 28–73. Springer-Verlag, 1991.
4. Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial order reductions for timed systems. In *International Conference on Concurrency Theory*, pages 485–500, 1998.
5. 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.
6. Andreas Blass and Yuri Gurevich. Fixed-choice and independent-choice logics. Technical Report TR-369-98, University of Michigan, August 1998.
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 *Proceedings 5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, Pensylvania, 1990. IEEE Computer Society Press.
9. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag, 1981.
10. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1994.
11. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.
12. J.B.J. Fourier. Second extrait. In G. Darboux, editor, *Oeuvres*, pages 325–328, Paris, 1890. Gauthiers-Villars.
13. T. A. Henzinger, Z. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.
14. David Hilbert and Paul Bernays. *Grundlagen der Mathematik*, volume 2. Springer-Verlag, 1939.
15. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
16. Manolis Koubarakis. Complexity results for first-order theories of temporal constraints. In Jon Doyle, Erik Sandewall, and Pietro Torasso, editors, *Proc. of the Fourth Internatinal Conference on Principles of Knowledge Representation and Reasoning (KR'94)*, pages 379–390, San Francisco, California, 1994. Morgan Kaufmann.
17. Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.
18. K. G. Larsen, P. Pettersson, and W. Yi. Model-checking for real-time systems. In *Proc. of the 10th Int. Conference on Fundamentals of Computation Theory*, volume 965 of *LNCS*, pages 62–88, August 1995.
19. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1+2), 1997.
20. Marius Minea. Partial order reduction for model checking of timed automata. In *International Conference on Concurrency Theory*, pages 431–446, 1999.
21. Marius 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.
22. 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*, 2002. Special Issue on Model Checking (To appear).
23. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Proceedings 13th International Workshop on Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 111–125, Madrid, Spain, September 1999.
24. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Fully symbolic model checking of timed systems using difference decision diagrams. In *Proceedings First International Workshop on Symbolic Model Checking*, volume 23-2 of *Electronic Notes in Theoretical Computer Science*, pages 89–108, Trento, Italy, July 1999.
25. Jesper Møller and Jakob Lichtenberg. Difference decision diagrams. Master's thesis, Department of Information Technology, Technical University of Denmark, Building 344, DK-2800 Lyngby, Denmark, August 1998.

26. C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.
27. T. G. Rokicki. *Representing and Modeling Digital Circuits.* PhD thesis, Stanford University, 1993.
28. R. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 4(24):529–543, October 1977.
29. Maria Sorea. TEMPO: A model checker for event-recording automata. Technical Report SRI-CSL-01-04, SRI International, Computer Science Laboratory, 2001.
30. A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
31. P. F. Williams, A. Biere, E. M. Clarke, and A. Gupta. Combining decision diagrams and SAT procedures for efficient symbolic model checking. In *Proc. Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, Chicago, U.S.A., July 2000. Springer-Verlag.
32. S. Yovine. Kronos: A verification tool for real-time systems. *Springer Int. Journal of Software Tools for Technology Transfer*, 1(1/2), October 1997.