Electronic Notes in Theoretical Computer Science 65 No. 6 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 20 pages

Exact Acceleration of Real-Time Model Checking

Martijn Hendriks [1,2]

Department of Computer Science
University of Nijmegen
Nijmegen, The Netherlands

Kim G. Larsen [3,4]

BRICS, Department of Computer Science
University of Aalborg
Aalborg, Denmark

Abstract

Different time scales do often occur in real-time systems, e.g., a polling real-time system samples the environment many times per second, whereas the environment may only change a few times per second. When these systems are modeled as (networks of) timed automata, the validation using symbolic model checking techniques can significantly be slowed down by unnecessary *fragmentation* of the symbolic state space. This paper introduces a syntactical adjustment to a subset of timed automata that addresses this fragmentation problem and that can speed-up forward symbolic reachability analysis in a significant way. We prove that this syntactical adjustment does not alter reachability properties and that it indeed is effective. We illustrate our *exact acceleration* technique with run-time data obtained with the model checkers Uppaal and Kronos. Moreover, we demonstrate that automated application of our exact acceleration technique can significantly speed-up the verification of the run-time behavior of LEGO Mindstorms programs.

[1] We thank Oliver Möller for interesting discussions and we thank Jozef Hooman and the four anonymous reviewers for valuable comments on earlier versions of this paper.
[2] Email: martijnh@cs.kun.nl
[3] Part-time Professor at the Department of Computer Science, Twente University, The Netherlands
[4] Email: kgl@cs.auc.dk

1 Introduction

Efficiency is the most important aspect for practical applicability of model checking techniques for validation and verification of systems. Nowadays, the main problem that inhibits the large scale application of these techniques is the *state space explosion* problem. The fact that the state space of systems – in general – grows exponentially in the size of the components, renders the practical verification of realistic systems often impossible. The fundamental difficulty of this problem cannot be relieved in general, and practical solutions must be found by algorithms and data structures optimized for specific application areas.

Timed automata [4,2] are very well suited for modeling real-time systems and protocols in which time plays an important role. The development of symbolic techniques to represent the uncountable state space of timed automata has enabled the mechanical interpretation of logic, e.g., the temporal logic TCTL, over timed automata [3]. This has been implemented by, e.g., the model checkers UPPAAL and KRONOS, and these tools have successfully been applied to various case studies [16,19].

An important problem concerning symbolic model checking of timed automata, is encountered when the timed automata in a model use different *time scales*. This, for example, is often the case for models of reactive programs with their environment. Typically, the automata that model the reactive programs are based on microseconds whereas the automata of the environment function in the order of seconds. This difference can give rise to an unnecessary fragmentation of the symbolic state space. As a result, the time and memory consumption of the model check process increases.

The fragmentation problem has already been encountered and described by Hune and Iversen et al during the verification of LEGO Mindstorms programs using UPPAAL [14,15]. The symbolic state space is severely fragmented by the busy-waiting behavior of the control program automata. This problem can in general occur during the symbolic model checking of systems that are modeled by timed automata. Examples include the aforementioned reactive programs, and polling real-time systems, e.g., programmable logic controllers [11]. The validation of communication protocols will probably also suffer from the fragmentation problem when the context of the protocol is taken into account.

We propose an acceleration technique for a subset of timed automata, namely those that contain special cycles, that addresses the fragmentation problem. Our technique consists of a syntactical adjustment that can easily be computed from the timed automaton itself. We prove that this syntactical adjustment is *exact* with respect to reachability properties and that it can effectively speed-up forward symbolic reachability analysis. As a result, our approach is readily applicable using the existing model checkers. We demonstrate the acceleration by experimental results of the verification of a toy example. Moreover, we explain how we can automatically apply our technique to verify the run-time behavior of LEGO Mindstorms programs. Experimental results show that this automated application can be very profitable.

Related work. Closely related work has been done in the field of symbolic verification of systems that are modeled by a discrete control graph with unbounded integer variables [9]. Static analysis of the control graph is used to detect *interesting cycles*, of which the result of iterated execution can be computed by one single *meta transition*. These meta transitions are then added to the system and favored by the state space exploration algorithm, resulting in faster exploration of the state space.

Symbolic techniques using *queue-content decision diagrams*, or QDDs, for the analysis of communication protocols that are modeled by finite-state machines that communicate through unbounded FIFO-queues, also use meta transitions to accelerate the exploration of the state space [7,8]. Special cycles in the control-graph, e.g., the repeated receiving of messages from a channel, are associated with meta transitions that compute all states that are reachable by the iterative execution of the cycle. In these approaches only a limited class of cycles in the control graph can be accelerated due to the expressivity of QDDs. To overcome this problem, *constrained* QDDs have been introduced, that allow the acceleration of *any* cycle in a control graph [10].

Recently, acceleration techniques have been proposed in the setting of parameterized model checking [1,18]. The techniques, again, compute the effect of an unbounded number of actions to accelerate the forward exploration process.

Möller's "parking" approach to the sketched fragmentation problem is, like our approach, based on a syntactical adjustment of timed automata to speed-up the state space exploration [17]. The parking idea is more general than ours, but our method is *exact*, whereas parking is mostly an over-approximation. Möller applies his approach to an example somewhat larger than our examples, and measures speed-ups in the same order of magnitude as we do. We think that both methods show promises for handling the fragmentation problem.

In a sense, the syntactical adjustment of our approach also is a meta transition that computes the result of iterated execution of a cycle in the timed automaton. Using a breadth-first search order then guarantees that the exploration of this meta transition is not postponed. As far as we know this is the first application of acceleration techniques to timed automata.

Outline. In section 2 we briefly introduce the theory of timed automata and the semantics of reachability. We also discuss the forward symbolic reachability analysis, as performed by Uppaal. Moreover, using an example we illustrate why different time scales in models can increase the reachable symbolic state space. In section 3 we define our syntactic adjustment and we prove that it is exact with respect to reachability and that it indeed is effective. In section 4 we show experimental results obtained with Uppaal and Kronos for a toy example. Moreover, we explain the automated application and we show experimental results. Finally, in section 5 we discuss our technique and we state the possibilities for future work.

2 Timed automata

This section has been based on the work of Alur and Dill on the subject of timed automata [4,2]. In order to define finite automata that use real valued clocks, the set of clock constraints over a set of clock variables is defined. Let X be a set of clock variables, then the set $\Phi(X)$ of clock constraints ϕ is defined by the following grammar, where $x \in X$, $c \in \mathbb{N}$, and $\sim$ denotes one of the binary relations $<, \leq, =, \geq$ or $>$.

$$\phi := x \sim c \mid \phi_1 \wedge \phi_2$$

A *clock interpretation* ν for a set X is a mapping from X to $\mathbb{R}^+$, where $\mathbb{R}^+$ denotes the set of positive real numbers including zero. A clock interpretation ν for X satisfies a clock constraint ϕ over X, denoted by $\nu \models \phi$, if and only if ϕ evaluates to *true* with the values for the clocks given by ν. For $\delta \in \mathbb{R}^+$, $\nu + \delta$ denotes the clock interpretation which maps every clock x to the value $\nu(x) + \delta$. For a set $Y \subseteq X$, $\nu[Y := 0]$ denotes the clock interpretation for X which assigns 0 to each $x \in Y$ and agrees with ν over the rest of the clocks. We let $\Gamma(X)$ denote the set of all clock interpretations for X.

Definition 2.1 The tuple $(L, l^0, \Sigma, X, I, E)$ defines a timed automaton, where L is a finite set of locations, $l^0 \in L$ is the initial location, Σ is a finite set of labels, X is a finite set of clocks, I is a mapping that labels each location $l \in L$ with some clock constraint in $\Phi(X)$ and $E \subseteq L \times \Sigma \times \Phi(X) \times 2^X \times L$ is a set of edges. An edge $(l, a, \phi, \lambda, l')$ represents a transition from location l to location l' on the symbol a. The clock constraint ϕ specifies when the edge is enabled and the set $\lambda \subseteq X$ gives the clocks to be reset with this edge.

Example 2.2 As a running example throughout this and the next sections, we use the timed automaton depicted in figure 1 in the Uppaal notation. The locations are depicted as vertices labeled with the name of the location.

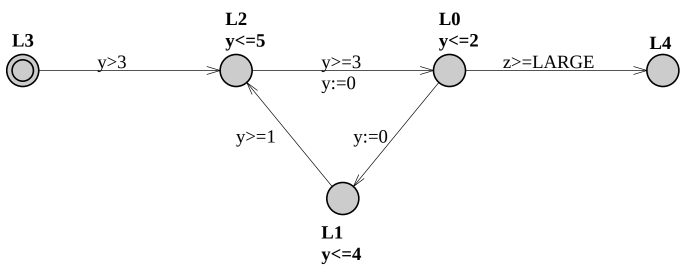

Fig. 1. Timed automaton P.

Location L3 is the initial location. The set of labels consists only of the *empty* label τ, that is not depicted. The clocks are y and z. The invariant mapping

I is given by the bold clock constraints depicted at the locations. If a location has no clock constraint associated with it, then we assume that the invariant for that location is **true**. The edges may be labeled with a clock constraint and a set of clocks. If an edge is not labeled with a clock constraint, then the guard of this edge is **true**. If an edge is not labeled with a set of clocks, then no clocks are reset on the edge, i.e. $\lambda = \emptyset$ for that edge. Finally, the object `LARGE` that appears in a clock guard on the edge from location `L0` to `L4` is a constant natural number.

The timed automaton of figure 1 offers a simplified modeling of a control program combined with an environment. The cycle `L0`, `L1`, `L2` corresponds to cyclic execution of a control program consisting of three atomic instructions with the invariants and guards on the clock `y` providing execution time information. Whenever the control cycle is in location `L0`, the environment (modeled by the clock `z`) is consulted potentially leading to an exit of the control cycle. The size of the threshold constant `LARGE` determines how slow the environment is relative to the execution time of control program instructions: the larger the constant the slower.

The semantics of a timed automaton A is defined by associating a transition system S_A with it. A state of S_A is a pair (l, ν), where l is a location of A and ν is a clock interpretation for X such that ν satisfies $I(l)$. There are two types of transitions in S_A:

- Let $\delta \in \mathbb{R}^+$. We say $((l, \nu), (l, \nu + \delta))$ is a δ-*delay transition*, iff $\nu + \delta' \models I(l)$ for all $0 \leq \delta' \leq \delta$.
- Let $a \in \Sigma$. We say $((l, \nu), (l', \nu'))$ is an a-*action transition*, iff an edge $(l, a, \phi, \lambda, l')$ exists such that $\nu \models \phi$, $\nu' = \nu[\lambda := 0]$ and $\nu' \models I(l')$.

Using this transition system, we can define the traces of a timed automaton.

Definition 2.3 Let $M = (L, l^0, \Sigma, X, I, E)$ be a timed automaton. We say that a finite or infinite sequence $((l_0, \nu_0), (l_1, \nu_1), ...)$ is a (l, ν)-*trace* of M, if

- $l_0 = l$ and $\nu_0 = \nu$, and
- $((l_i, \nu_i), (l_{i+1}, \nu_{i+1}))$ is an a-action transition for some $a \in \Sigma$ or a δ-delay transition for some $\delta \in \mathbb{R}^+$, for all $i \geq 0$ that appear in the sequence.

We call a (l, ν)-trace *compressed*, if that (l, ν)-trace starts with a delay transition, and it does not contain two consecutive action or delay transitions. Thus, after every action transition follows a delay transition and after every delay transition follows an action transition. Finally, we let $Tr(M)$ denote the set of all (l^0, ν_{init})-traces of a timed automaton M, where l^0 is the initial location of M and $\nu_{init}(x) = 0$ for all clocks x of M.

2.1 Model checking

In this paper we are concerned with reachability properties of timed automata. A reachability property ϕ of a timed automaton M is of the form $\exists\Diamond(P)$, where P is a *state* property of M. The state properties P are interpreted over the states of M, whereas the reachability properties ϕ are interpreted over the traces of M. For example, a reachability property for automaton P of example 2.2 is the following:

$$\exists\Diamond(\texttt{L2} \wedge (\texttt{y<=4} \vee \texttt{z>20}))$$

Informally this property means that a state (l, ν), such that $l = \texttt{L2}$, and $\nu(\texttt{y}) \leq 4$ or $\nu(\texttt{z}) > 20$, is reachable. This example also demonstrates the straightforward satisfaction relation, $\models$, that is used to interpret state formulas over the states of a timed automaton.

Using the traces of a timed automaton and the satisfaction relation for the state properties, we can define the satisfaction relation for the reachability properties.

Definition 2.4 For a timed automaton M and a property $\phi = \exists\Diamond(P)$, we say that M satisfies ϕ, denoted by $M \models \phi$, if a trace $((l_0, \nu_0), (l_1, \nu_1), ...) \in Tr(M)$ exists, such that $(l_i, \nu_i) \models P$ for some $i \geq 0$.

The reachability question and – more general – the timed computation tree logic TCTL, is decidable and various model checkers exist, e.g., UPPAAL and KRONOS [3,16,19]. In the next section we will explain forward symbolic reachability analysis that is used by UPPAAL and KRONOS.

2.2 Forward symbolic reachability analysis

The transition system defined by a timed automaton has uncountable many states. This renders the straightforward application of traditional finite-state model-checking algorithms impossible. However, Alur et al introduced the so-called regions as a finite-state symbolic technique for proving decidability of reachability as well as model-checking for TCTL [3].

Unfortunately, the number of regions grow exponentially in the size of the constants used in the model, and are therefore not particularly useful when constructing efficient model-checking tools; this applies in particular in or setting where large constants are to be expected due to the large difference in time-scale between the control program and the environment. Instead, real-time model-checkers such as UPPAAL and KRONOS are based on so-called zones, which provide a representation of convex sets of clock interpretations as constraints on (lower and upper) bounds on individual clocks and clock differences. As an example consider the set of clock interpretations for the two clocks x and y described by the following constraints:

$$0 \leq x \leq 5 \quad 7 \leq y \leq 12 \quad y - x = 7$$

Zones may efficiently be represented as *Difference Bounded Matrices* [6,12], which offers a canonical representation for constraint systems. Furthermore, the canonical form allows inclusion-check as well as the effect of action and delay transitions to be computed efficiently (i.e. time complexity mostly quadratic and in worst case cubic in the number of clocks)

The model-checking engine of UPPAAL performs a symbolic exploration of the reachable symbolic state space on-the-fly, starting with an initial state. For each unexplored symbolic state, its successors due to delay and action transitions are computed, and compared to already explored states. If a successor has already been seen in the past, it is discarded. On the other hand, if a successor has not yet been seen, it is added to the list of states waiting to be further explored.

To illustrate forward symbolic exploration consider the timed automaton of example 2.2. Depending on the value of `LARGE`, the cycle in automaton P must be executed a certain (large) number of times before the edge to location `L4` is enabled. In table 1 we show the symbolic states that result from one execution of the cycle starting in the initial state.

State #	Location	Zone		
1	`L3`	$y = 0$	$z = 0$	$z - y = 0$
2	`L2`	$3 < y \leq 5$	$3 < z \leq 5$	$z - y = 0$
3	`L0`	$0 \leq y \leq 2$	$3 < z \leq 7$	$3 < z - y \leq 5$
4	`L1`	$0 \leq y \leq 4$	$3 < z \leq 11$	$3 < z - y \leq 7$
5	`L2`	$1 \leq y \leq 5$	$4 < z \leq 12$	$3 < z - y \leq 7$
6	`L0`	$0 \leq y \leq 2$	$6 < z \leq 14$	$6 < z - y \leq 12$

Table 1
Simulation data of P.

This simulation data shows that the sixth symbolic state is not contained in the third, since the zone of the sixth symbolic state contains clock interpretations that are not in the zone of the third symbolic state, and therefore this state is explored further. In general, every new execution of the cycle gives rise to new symbolic states.

This example illustrates the symbolic state space fragmentation (and explosion) of busy-waiting on a slow environment. The cycle in automaton P is the control cycle of a control program and only if the environmental clock `z` passes some value, then the control program can undertake some action. It is clear that time and memory consumption of exploration of the complete reachable state space of P is very dependent on the value of `LARGE`. As we will see in the next section, this can be avoided.

3 Exact acceleration

The timed automaton of example 2.2 illustrates the fragmentation of the reachable symbolic state space. In this section we propose a technique that eliminates the fragmentation that is due to special cycles. In section 3.1 we give some basic definitions concerning cycles in timed automata. The subset of cycles that we can accelerate, is defined in section 3.2. Finally, in section

3.3 we define the syntactical adjustment of a subset of timed automata that accelerates the forward symbolic reachability analysis. We prove that this adjustment is exact with respect to reachability properties and that it accelerates the forward symbolic exploration of the reachable state space.

3.1 An introduction to cycles

We start this section with the definition of two functions to obtain the source and target locations of an edge of a timed automaton.

$$src((l, a, \phi, \lambda, l')) = l$$

$$trg((l, a, \phi, \lambda, l')) = l'$$

For a sequence of edges $E_c = (e_0, e_1, ..., e_{n-1}) \in E^n$ of a timed automaton, we let $Loc(E_c)$ denote the set of locations that appear in the edges:

$$Loc(E_c) \quad = \quad \{\, l \in L \mid \exists_{e \in E_c} [\, src(e) = l \ \vee \ trg(e) = l \,] \,\}$$

A cycle in a timed automaton is a sequence of edges, defined as follows:

Definition 3.1 Let $M = (L, l^0, \Sigma, X, I, E)$ be a timed automaton and let $n \geq 1$. We say that a sequence $(e_0, e_1, ..., e_{n-1}) \in E^n$ is a *cycle*, if the following holds:

- $trg(e_i) = src(e_{i+1})$ for all $0 \leq i < n - 1$, and $trg(e_{n-1}) = src(e_0)$, and
- $i \neq j \Rightarrow e_i \neq e_j$ for all $0 \leq i, j < n$.

The timed automaton of example 2.2 contains a cycle that, for example, is defined by the edges `L0` to `L1`, `L1` to `L2` and `L2` to `L0`. For a cycle, we can define the number of times that it is executable.

Definition 3.2 Let $E_c = (e_0, e_1, ..., e_{n-1})$ be a cycle in some timed automaton M and let $m > 0$. We say that the cycle is m-times executable, if a finite compressed trace in $Tr(M)$exists with a suffix, say of the form

$$((l_0, \nu_0), (l_0, \nu'_0), (l_1, \nu_1), ..., (l_{k-1}, \nu'_{k-1}), (l_k, \nu_k))$$

where $l_0 = src(e_0)$, such that the following holds:

- the i-th action transition $((l_i, \nu'_i), (l_{i+1}, \nu_{i+1}))$ corresponds to edge $e_{i \bmod n}$
- there are $m \cdot n$ action transitions

Example 3.3 The cycle in the timed automaton of example 2.2 is 1-time executable. This can be understood from the following suffix of a finite compressed trace, of which the first state obviously is reachable from the initial state (we denote the clock interpretation ν by a tuple that first contains the value $\nu(\mathtt{y})$ and second the value $\nu(\mathtt{z})$):

$$\Big((\mathtt{L0},(0,4)), (\mathtt{L0},(1,5)), (\mathtt{L1},(0,5)), (\mathtt{L1},(2,7)), (\mathtt{L2},(2,7)), (\mathtt{L2},(4,9)), (\mathtt{L0},(0,9))\Big)$$

Cycles in timed automata suffer in general from a certain delay due to invariants at locations and clock guards on edges. In many cases, this delay varies for every execution of the cycle. However, we will later see that there exist cycles with a "fixed" window of delay for each execution of the cycle. But first, we define this window as an interval containing all possible delays that can be accumulated by following the cycle exactly once.

Definition 3.4 Consider a timed automaton M and let $E_c = (e_0, ..., e_{n-1})$ be a cycle in M. We say that an interval $[a, b]$ is the *window* of E_c, if for all subsequences of compressed traces in $Tr(M)$, say of the form

$$((l_0, \nu_0), (l_0, \nu'_0), (l_1, \nu_1), ..., (l_{k-1}, \nu_{k-1}), (l_{k-1}, \nu'_{k-1}), (l_k, \nu_k))$$

such that $l_0 = l_k = src(e_0)$ and every action transition $((l_i, \nu'_i), (l_{i+1}, \nu_{i+1}))$ is due to edge e_i (this subsequence thus denotes exactly one execution of E_c), the following holds:

- the total amount of delay in this subsequence is an element of $[a, b]$, and
- for all $d \in [a, b]$ it holds that we can adjust the delays in the subsequence such that they accumulate to d, and there exists a trace in $Tr(M)$ of which it is a subsequence.

This window property is not trivial. There are cycles with a window, as we will see in lemma 3.8. Moreover, we can prove that not every cycle has a window by providing a counter example.

Lemma 3.5 *Not every cycle has a window.*

3.2 Acceleratable cycles

In this section we introduce a subset of interesting cycles in timed automata. These interesting cycles can use only one clock in the invariants, guards and resets. This clock can be used to specify lower and upper bounds on the edges of the cycle. This might seem like a strong restriction, but we argue that these kind of cycles occur often in control graphs of, e.g., polling real-time systems.

Definition 3.6 Let $M = (L, l^0, \Sigma, X, I, E)$ be a timed automaton, let $E_c = (e_0, ..., e_{n-1}) \in E^n$ and let $y \in X$. We say that the tuple (E_c, y) is an *acceleratable cycle*, if

- E_c is a cycle,
- $I(l)$ is empty or has the form $\{y \leq c\}$ for all $l \in Loc(E_c)$,
- if $(l, a, \phi, \lambda, l') \in E_c$, then either ϕ is empty or has the form $\{y \geq c\}$, and λ is empty or only contains y, and
- y is reset on all in-going edges to $src(e_0)$.

Clock y is called *the clock of the cycle* and location $src(e_0)$ is called *the reset location* from now on. The cycle in our example automaton is an acceleratable cycle, if we choose clock `y` as the clock of the cycle and `L0` as the reset location. The guards and invariants have the correct form for clock `y` and this clock is reset on the only incoming edge of `L0`.

To extract the constants from the clock guards and invariants, we define two partial functions cn_g and cn_I that map clock guards and invariants to natural numbers:

$$cn_g(\phi) = 0 \quad \text{if } \phi = \emptyset$$
$$cn_g(\phi) = c \quad \text{if } \phi = \{y \geq c\}$$

$$cn_I(\phi) = \infty \quad \text{if } \phi = \emptyset$$
$$cn_I(\phi) = c \quad \text{if } \phi = \{y \leq c\}$$

Acceleratable cycles have the property that if the cycle can be executed once, then it can be executed infinitely often. Consider, for example, the finite compressed trace of example 3.3. The first and the last state of this trace agree on the value of clock y. Since the guards and invariants in the cycle are solely concerned with this clock, the sequence action and delay transitions can be repeated an arbitrary number of times.

Lemma 3.7 (Cycle consecution) *Let (E_c, y) be an acceleratable cycle of some timed automaton M. If E_c is 1-time executable, then it is m-times executable, for all $m > 0$.*

Our acceleratable cycles have a window, that can be computed from the syntax of the timed automaton.

Lemma 3.8 (Window computation) *Each acceleratable cycle has a window.*

Proof (Sketch) Consider a timed automaton $M = (L, l^0, \Sigma, X, I, E)$ with an acceleratable cycle $((e_0, e_1, ..., e_{n-1}), y)$. We show how to compute the window from the syntax of the timed automaton.

Let $(l_i, a_i, \phi_i, \lambda_i, l_{i+1})$ denote edge e_i. We can find p natural numbers $0 \leq k_0 < k_1 < ... < k_{p-1}$ that exactly correspond to the indices of the edges on which clock y is reset. Since we know by definition that y is reset on edge e_{n-1}, p is at least one. Next, we compute the following numbers for $0 \leq j < p$ (we define $k_{-1} = -1$):

$$\begin{aligned} a_{k_j} &= max\,\{\, cn_g(\phi_i) \mid k_{j-1} < i \leq k_j \,\} \\ b_{k_j} &= cn_I(I(l_{k_j})) \end{aligned}$$

Since we consider the guards and invariants of an acceleratable cycle, all the numbers a_{k_j} and b_{k_j} are defined. We can show that the acceleratable cycle has a window of

$$\left[\sum_{j=0}^{p-1} a_{k_j} \,,\, \sum_{j=0}^{p-1} b_{k_j} \right]$$

□

Example 3.9 We saw that the timed automaton of example 2.2 has an acceleratable cycle starting in `L0`. Applying our technique of window computation, we first obtain that e_0 is the edge from `L0` to `L1`, e_1 is the edge from `L1` to `L2`, and e_2 is the edge from `L2` to `L0`. Since clock `y` is reset two times on these edges, we have $p = 2$ and $k_0 = 0$ and $k_1 = 2$. Thus, $a_0 = max\{0\} = 0$ and $a_1 = max\{1, 3\} = 3$. Moreover, $b_0 = cn_I(I(\mathtt{L0})) = 2$, and $b_1 = cn_I(I(\mathtt{L2})) = 5$. Therefore, the acceleratable cycle has a window of $[3, 7]$.

3.3 Acceleration

The motivation of this work is the acceleration of real-time model checking. We explained that the time that model checking of the property $\exists\Diamond(\mathtt{L4})$ for the timed automaton of example 2.2 takes, is very dependent on the value of the constant `LARGE`. This is due to the fact that many executions of the cycle must be explored to let the value of clock `z` grow large enough. The following definition appends an extra cycle, the meta transition, to automata with an acceleratable cycle. As we will see, this appended cycle computes the effect of the iterated execution of the acceleratable cycle.

Definition 3.10 Let $M = (L, l^0, \Sigma, X, I, E)$ be a timed automaton and let $A = ((e_0, ..., e_{n-1}), y)$ be an acceleratable cycle. Let $L = \{l_0, l_1, ..., l_m\}$, and let $e_i = (l_i, a_i, \phi_i, \lambda_i, l_{i+1})$. The *acceleration of* M is a new timed automaton $Acc(M, A) = (L_{new}, l^0, \Sigma, X, I_{new}, E_{new})$, where

$$L_{new} = \quad L \cup \{l'_1, l'_2, ..., l'_{n-1}\} \cup \{l'_0\} \cup \{l''_1, l''_2, ..., l''_{n-1}\}$$

$$\begin{aligned}
I_{new}(l_i) &= I(l_i) \text{ for all } 0 \leq i \leq m \\
I_{new}(l'_i) &= I(l_i) \text{ for all } 1 \leq i \leq n-1 \\
I_{new}(l'_0) &= \emptyset \\
I_{new}(l''_i) &= I(l_i) \text{ for all } 1 \leq i \leq n-1
\end{aligned}$$

$$\begin{aligned}
E_{new} = \quad & E \quad \cup \\
& \{\, (l_0, a_0, \phi_0, \lambda_0, l'_1), (l'_{n-1}, a_{n-1}, \phi_{n-1}, \lambda_{n-1}, l'_0) \,\} \quad \cup \\
& \{\, (l'_0, a_0, \phi_0, \lambda_0, l''_1), (l''_{n-1}, a_{n-1}, \phi_{n-1}, \lambda_{n-1}, l_0) \,\} \quad \cup \\
& \{\, (l'_i, a_i, \phi_i, \lambda_i, l'_{i+1}), (l''_i, a_i, \phi_i, \lambda_i, l''_{i+1}) \mid \text{for all } 1 \leq i < n-1 \}
\end{aligned}$$

Note that the definition of E_{new} does not cover the simple case where $n = 1$. Then, only two edges must be added: $(l_0, a, \phi, \lambda, l'_0)$ and $(l'_0, a, \phi, \lambda, l_0)$, if $e_0 = (l_0, a, \phi, \lambda, l_0)$.

Example 3.11 Since the timed automaton of example 2.2 has an acceleratable cycle, we can construct the acceleration, see figure 2. The key idea behind

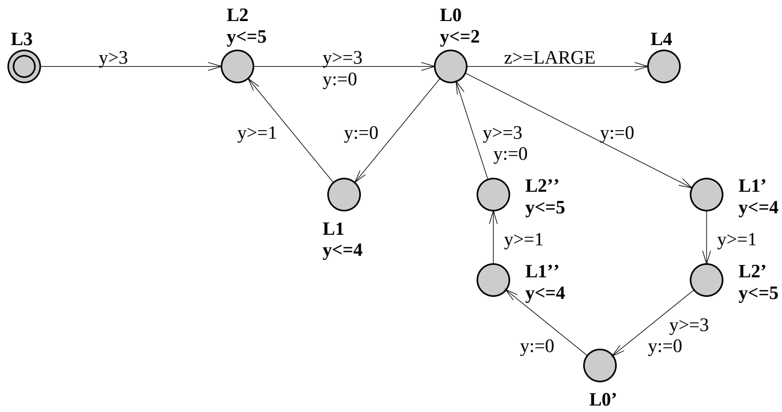

Fig. 2. Automaton P_A: the accelerated version of P.

the acceleration is that the new location `L0'`, that mimics location `L0`, has no invariant [17].

This acceleration is only interesting if we can use the accelerated version of some automaton to model check properties of the original automaton. The next theorem assures this, provided that the window of the acceleratable cycle is relatively wide enough.

Theorem 3.12 (Equivalence of reachability) *Let* $(L, l^0, \Sigma, X, I, E)$ *be a timed automaton* M*, let* A *be an acceleratable cycle of* M *with a window of* $[a, b]$*, and let* ϕ *be a reachability properties of* M*.*

$$3a \leq 2b \Rightarrow (M \models \phi \Leftrightarrow Acc(M, A) \models \phi)$$

If the precondition of this theorem is *not* satisfied, then the acceleration is still a safe over-approximation. Thus, if a state is unreachable in $Acc(M, A)$, then it is also unreachable in M. The fact that definition 3.10 unfolds the acceleratable cycle twice to form the appended cycle, is the direct cause of the necessary relative width of the window. It can be shown that the precondition can be generalized to $(i+1)a \leq ib$, where i is the number of unfoldings of the acceleratable cycle. This means that if a is strictly less than b, then we *can* accelerate. At this time we do not know how to handle the simple case where $a = b$.

It may seem that the addition of extra locations only increases the state space. However, we claim that if the clock of the acceleratable cycle is also reset on the first edge of the cycle, then the acceleration is guaranteed to work for breadth-first forward symbolic reachability analysis. Note that if a timed automaton has an acceleratable cycle, but does not satisfy the constraint described above, then we can introduce a dummy location to assure that this requirement is satisfied.

Example 3.13 Suppose that the timed automaton of example 2.2 does not have a reset of `y` on the edge from `L0` to `L1`. Then we add a dummy location between the locations `L2` and `L0` with invariant $y \leq 0$. Next, the edge from `L2` to `L0` is redirected to the dummy location. Finally, we add an extra edge from the dummy location to location `L0` that resets clock `y`.

The cycle that results from our trick is larger, but with the dummy location as reset location, it satisfies the requirement for the effectiveness of acceleration, as formalized in the next theorem.

Theorem 3.14 (Effectiveness of acceleration) *Let the timed automaton M be defined by $(L, l^0, \Sigma, X, I, E)$ and let $A = ((e_0, ..., e_{n-1}), y)$ be an acceleratable cycle. If y is reset on edge e_0, then all states reachable by more than one execution of the acceleratable cycle in M, are reachable by exactly one execution of the appended cycle in $Acc(M, A)$.*

This theorem guarantees us that breadth-first forward symbolic reachability analysis is accelerated. When using a breadth-first search-order, the appended cycle is "in parallel" explored with the first few executions of the acceleratable cycle. Our effectiveness theorem assures that the symbolic state that results from the execution of the appended cycle swallows the symbolic states that result from two or more executions of the acceleratable cycle. Therefore, the acceleratable cycle is only explored a small number of times. (The exact number of times depends on the implementation of the model check algorithm, but it will be independent of the difference in time scale.)

4 Experimental results

To demonstrate the effect of acceleration, we collected run-time data for the automaton of example 2.2 and for the automaton of example 3.11, which we manually accelerated. We used the model checkers UPPAAL and KRONOS – both with a breadth-first search order – to verify whether or not location `L4` is reachable.

UPPAAL 3.1.57					KRONOS 2.4.4		
	P		P_A				
	Mem	Time	Mem	Time		P	P_A
`LARGE`	[kB]	[s]	[kB]	[s]	`LARGE`	#	#
10^3	1084	0.05	1084	0.01	10^2	45	21
10^4	1488	2.98	1084	0.01	10^3	432	21
10^5	6312	374	1084	0.01	10^4	4290	21
10^6	–	†	1084	0.01	$1,5 \cdot 10^4$	6432	21

Table 2
Run-time data comparing P and its accelerated version P_A.

Table 2 shows the time and memory consumption of UPPAAL, and the number of states (in the table denoted by #) that Kronos explored, as functions of the value of LARGE. The † in the table means that we ran out of patience: the model checking takes more than 10 minutes. (Note that we do not want to compare Uppaal and Kronos. We only demonstrate the insensitivity of the accelerated version to the value of LARGE).

We can explain the constant time and memory consumption and the constant number of explored states of P_A by theorem 3.14. The breadth-first search order assures that the appended cycle is explored before the complete exploration of the acceleratable cycle. The resulting symbolic state swallows all the smaller symbolic states that result from execution of the acceleratable cycle.

A less theoretical example is provided by a compiler which translates Uppaal models to (i) executable LEGO Mindstorms code and (ii) another Uppaal model of the run-time behavior of the executable code. We constructed a very small example to illustrate how our acceleration technique can be used to speed-up the verification of the run-time behavior.

Consider the processes P_0 and P_1 in figures 3 and 4. These processes model a reactive program, called B, which controls two actuators and uses two sensors.

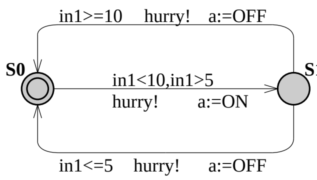

Fig. 3. Process P_0.

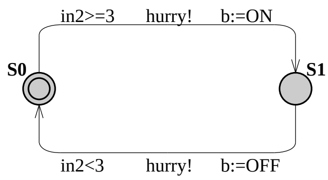

Fig. 4. Process P_1.

Process P_0 uses sensor 1 (whose value is modeled by the variable in1) and actuator A (whose mode is modeled by the variable a). Similarly, process P_1 uses sensor 2 and actuator B. Initially, both actuators are off. If the sensor value of sensor 1 becomes between 5 and 10, process P_0 switches actuator A on. If the sensor value leaves this region, then process P_0 switches actuator A off again. Process P_1 functions in a similar manner.

Figures 5 and 6 are the environmental processes. The hurry dummy provides an always enabled synchronization over the urgent channel hurry, which creates urgent edges. Note that all edges of P_0 and P_1 use this channel, with the result that they are taken as soon as possible. The environment periodically updates the sensor values with a “speed” expressed by the constant LARGE.

After compilation of the model, we obtain the symbolic byte code program of figure 7. There are three kinds of instructions present. First, there are assignments, e.g., v[0]:=0 and actmode[A]:=off. The first assignment manipulates the internal variable with index zero. The second assignment manipulates the mode of actuator A. Second, there are “test and branch far”

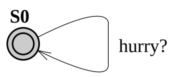

Fig. 5. The hurry dummy.

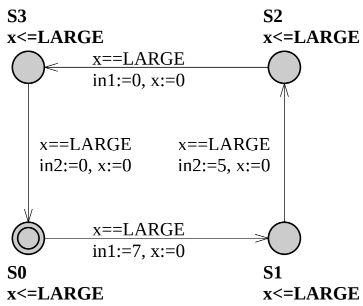

Fig. 6. The environment.

```
0000  v[0] := 0
0005  v[1] := 0
0010  actmode[A] := off
0012  actmode[B] := off
0014  tbf 0!=v[0], 51
0022  tbf 10<=snsval[0], 48
0030  tbf 5>=snsval[0], 48
0038  v[0] := 1
0043  actmode[A] := on
0045  baf 106
0048  baf 106
0051  tbf 10==snsval[0], 67
0059  tbf 10>=snsval[0], 77
0067  v[0] := 0
0072  actmode[A] := off
0074  baf 106
0077  tbf 5==snsval[0], 93
0085  tbf 5<=snsval[0], 103
0093  v[0] := 0
0098  actmode[A] := off
0100  baf 106
0103  baf 106
0106  tbf 0!=v[1], 143
0114  tbf 3==snsval[1], 130
0122  tbf 3>=snsval[1], 140
0130  v[1] := 1
0135  actmode[B] := on
0137  baf 164
0140  baf 164
0143  tbf 3<=snsval[1], 161
0151  v[1] := 0
0156  actmode[B] := off
0158  baf 164
0161  baf 164
0164  baf 14
```

Fig. 7. The executable byte code.

instructions, e.g., `tbf 0!=v[0], 51`. If the boolean expression `0!=v[0]` evaluates to *true*, then control is transferred to the instruction with address `51`. Otherwise, control is transferred to the next instruction. Finally, there are "branch always far" instructions, e.g., `baf 14`. This instruction transfers control to the instruction with address 14.

The byte code simulates one interleaving of P_0 and P_1. In an infinite while loop the processes execute action transitions in an alternating way. This loop starts with the instruction at address 14, and ends with the "baf" instruction at address 164. The "tbf" instructions inside the loop implement the alternation between P_0 and P_1 and the guards on the edges.

The second product of compilation is a model of the run-time behavior of the byte code program of figure 7. This model naturally contains the environmental processes of figures 5 and 6. The processes P_0 and P_1, however, are replaced by an exact model of the run-time behavior of the generated byte code, which is depicted in figure 8.

The "byte code process" of figure 8 is constructed by concatenation of models of the individual instructions of the executable byte code program [15]. Location `S`i models the $i+1$-th instruction. For example, location `S0`

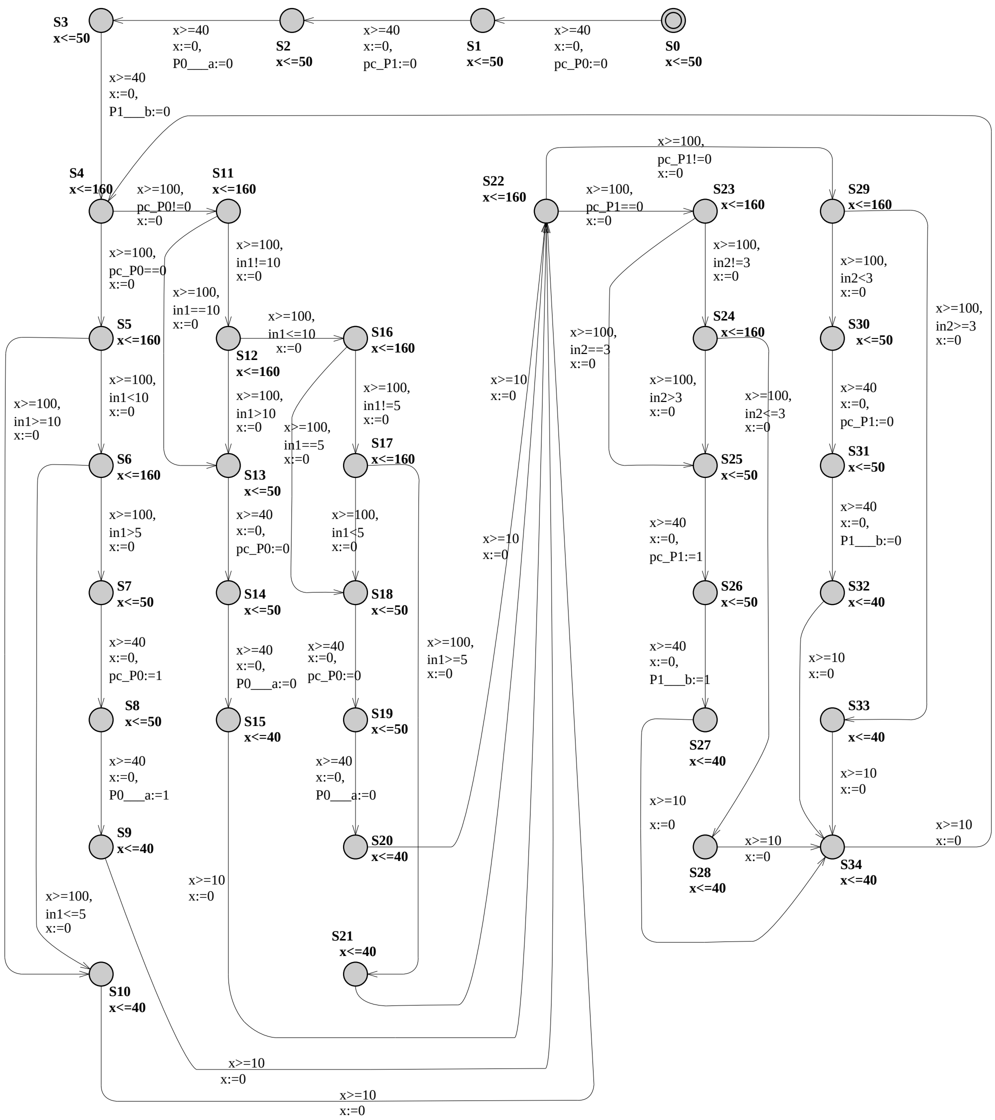

Fig. 8. The UPPAAL process of the byte code program.

models the first instruction, the assignment `v[0]:=0`. Clock `x` is used to model the duration of the instruction, which in this case is between 40 and 50 time units. The actual assignment is performed on exit of the location.

Note that every cycle in the byte code process is an acceleratable cycle. Due to the timing uncertainty of the instructions, many of these acceleratable cycles satisfy the window constraint $3a \leq 2b$ of theorem 3.12. However, our theory of exact acceleration has been developed for "simple" timed automata. UPPAAL models differ on three points:

- The presence of bounded integer variables.
- Parallel composition of processes. This, for example, introduces additional

location invariants on the locations of acceleratable cycles.

- The presence of urgent locations, committed locations and urgent channels.

Our theory can be adapted to overcome the first two differences. However, examples show that the presence of urgent channels can disturb the exactness of the acceleration. A solution to this is not yet known.

We implemented our theory of exact acceleration in the compiler to accelerate *idle cycles* of the byte code processes. These idle cycles occur when no transitions of the source processes of the byte code are enabled. In this situation, the byte code process tests all guards, but finds none satisfied. As a result, the byte code process displays useless busy-waiting behavior.

Our implementation uses a compressed version of the appended cycle as in definition 3.10. Moreover, we included the schemes to overcome the problems of bounded integer variables and parallel composition. Although we have not yet *proven* that this results in exact acceleration, we believe that it does when no urgent channels are used. Note that it *certainly* always is an over-approximation which can be used to check the truth of invariance and untruth of reachability properties.

To demonstrate the effect of this automatic application of exact acceleration, we checked two properties for the generated model of the run-time behavior of the byte code.

$$\forall\Box(\texttt{in1==0} \Rightarrow \texttt{in1==0}) \tag{1}$$

Property 1 obviously is always true. Consequently, the complete reachable state space will be explored.

$$\exists\Diamond(\texttt{Env.S3}) \tag{2}$$

Property 2 is used to explore part of the reachable state space. We can verify the truth of this property in the accelerated model under the assumption that our implementation is an exact acceleration.

We measured the time and memory consumption for these two properties as a function of the value of the constant `LARGE` for the unaccelerated model B and for the model B_A in which four idle cycles are accelerated. See table 3 and table 4 for the results.

	B		B_A	
`LARGE`	Mem [kB]	Time [s]	Mem [kB]	Time [s]
10^3	1084	0.05	1084	0.07
10^4	2412	0.38	2276	0.27
10^5	7492	16.20	4548	7.72
10^6	54676	2175.92	26704	1048.14

Table 3
Run-time data of for property 1.

	B		B_A	
LARGE	Mem [kB]	Time [s]	Mem [kB]	Time [s]
10^3	1084	0.02	1084	0.03
10^4	2212	0.18	1084	0.07
10^5	4876	7.65	2084	0.16
10^6	28716	880.37	3260	5.46

Table 4
Run-time data of for property 2.

The phenomenal speed-up of the theoretical example is not achieved in the more realistic example of the byte code. However, it seems that exploration of the complete reachable state space is approximately 2 times cheaper when LARGE becomes greater than 10^5 . Exploration of only a part of the reachable state space shows a larger improvement: it is approximately 43 times faster when LARGE equals 10^5, and approximately 160 times faster when LARGE equals 10^6. The data suggests that this difference will increase as LARGE increases.

5 Conclusion

In this paper we have presented an acceleration technique for forward symbolic reachability analysis of timed automata. Our technique is applicable to a subset of timed automata, namely those that contain *acceleratable* cycles. We append an extra cycle to the timed automaton that in *one* execution computes the result of the *iterated* execution of the acceleratable cycle in the original automaton. Whether or not a cycle is acceleratable, and the form of the appended cycle are easily computable from the syntax of the timed automaton.

We have proven that our syntactic adjustment is exact with respect to reachability properties and that it will speed up forward symbolic reachability analysis with a breadth-first search order. Using the model checkers UPPAAL and KRONOS, we have demonstrated that our technique can seriously reduce the time and memory consumption of the model check process. Moreover, we have shown how exact acceleration can automatically be applied, and that this gives a significant improvement.

Future work. It would be interesting to investigate the weakening of the constraints on acceleratable cycles, as used in this paper. We can probably permit upper bounds on the clock of the cycle and lower bounds on the other clocks on the edges of the cycle. As we already pointed out in section 4, the invariants of the cycle may also contain other clocks than the clock of the cycle. Another point of interest is to *replace* the acceleratable cycle by a linear structure. This could drop the dependency on the breadth-first search order.

Generalization of our technique to UPPAAL models is almost necessary for the practical applicability. As we already mentioned, parallel composition and bounded integer variables of UPPAAL probably pose no problems. The urgent channels, however, do, and additional research is needed to discover their exact nature in a setting of acceleration. Another important issue are the additional cycles that result from the parallel composition of automata. These cycles – that thus only exists in the parallel composition – could be significant for our technique. Yet, the technique in its present form cannot handle these cycles.

References

[1] Abdulla, P., A. Bouajjani, B. Jonsson and M. Nilsson, *Handling Global Conditions in Parameterized System Verification*, in: *11th International Conference on Computer Aided Verification*, number 1633 in LNCS (1999), pp. 134–145.

[2] Alur, R., *Timed Automata*, in: *11th International Conference on Computer Aided Verification*, number 1633 in LNCS (1999), pp. 8–22.

[3] Alur, R., C. Courcoubetis and D. Dill, *Model checking in dense real time*, Information and Computation **104** (1993), pp. 2–34.

[4] Alur, R. and D. Dill, *Automata for modeling real-time systems*, in: *17th International Colloquium on Automata, Languages, and Programming*, 1990, pp. 322–335.

[5] Behrmann, G., A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson and J. Romijn, *Efficient Guiding Towards Cost-Optimality in* UPPAAL, in: T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lecture Notes in Computer Science (2001), pp. 174–188.

[6] Bellman, R., "Dynamic Programming," Princeton University Press, 1957.

[7] Boigelot, B. and P. Godefroid, *Symbolic verification of communication protocols with infinite state spaces using QDDs*, in: *8th International Conference on Computer Aided Verification*, number 1102 in LNCS (1996), pp. 1–12.

[8] Boigelot, B., P. Godefroid, B. Willems and P. Wolper, *The power of QDDs*, in: *4th International Static Analysis Symposium*, LNCS (1997).

[9] Boigelot, B. and P. Wolper, *Symbolic verification with periodic sets*, in: *6th International Conference on Computer Aided Verification*, number 808 in LNCS (1994), pp. 55–67.

[10] Bouajjani, A. and P. Habermehl, *Symbolic reachability analysis of FIFO-channel systems with non-regular sets of configurations*, in: *24th International Colloquium on Automata, Languages, and Programming*, number 1256 in LNCS (1997).

[11] Dierks, H., "Specification and Verification of Polling Real-Time Systems," Ph.D. thesis, Carl von Ossietzky Universität Oldenburg (1999).

[12] Dill, D., *Timing Assumptions and Verification of Finite-State Concurrent Systems*, in: J. Sifakis, editor, *Proc. of Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science (1989), pp. 197–212.

[13] Hune, T., K. G. Larsen and P. Pettersson, *Guided Synthesis of Control Programs Using* UPPAAL, in: T. H. Lai, editor, *Proc. of the IEEE ICDCS International Workshop on Distributed Systems Verification and Validation* (2000), pp. E15–E22.

[14] Hune, T. S., *Modeling a language for embedded systems in timed automata*, Technical Report RS-00-17, BRICS, Basic Research in computer Science (2000), 26 pp. Earlier version entitled *Modelling a Real-Time Language* appeared in FMICS99, pages 259–282.

[15] Iversen, T. K., K. J. Kristoffersen, K. G. Larsen, M. Laursen, R. G. Madsen, S. K. Mortensen, P. Pettersson and C. B. Thomasen, *Model-Checking Real-Time Control Programs — Verifying LEGO Mindstorms Systems Using* UPPAAL, in: *IEEE Euromicro Conference on Real-Time Systems*, 2000, pp. 147–155.

[16] Larsen, K. G., P. Pettersson and W. Yi, UPPAAL *in a Nutshell*, in: *International Journal on Software Tools for Technology Transfer*, 1 (1998), pp. 134–152.

[17] Möller, M. O., *Parking Can Get You There Faster*, in: *Proceedings of the Workshop on Theory and Practice of Timed Systems*, ENTCS, 2002.

[18] Pnueli, A. and E. Shahar, *Liveness and Acceleration in Parameterized Verification*, in: *12th International Conference on Computer Aided Verification*, number 1855 in LNCS (2000), pp. 328–343.

[19] Yovine, S., KRONOS*: a verification tool for real-time systems*, in: *Software Tools for Technology Transfer*, 1997.