Static Path Compression in Timed Systems *

Agata Janowska¹ and Wojciech Penczek²,³

¹ Institute of Informatics, Warsaw University, Banacha 2, 02-097 Warsaw, Poland
² Institute of Computer Science, PAS, Ordona 21, 01-237 Warsaw, Poland
³ Institute of Informatics, Podlasie Academy, 3 Maja 54, 08-110 Siedlce, Poland
janowska@mimuw.edu.pl, penczek@ipipan.waw.pl

Abstract. The paper presents a method of abstraction for timed systems. To extract an abstract model of a timed system we propose to use static analysis, namely a technique called path compression. The idea behind the path compression consists in identifying a path (or a set of paths) on which a process executes a sequence of transitions that do not influence a reachability property being verified, and replacing it with a single transition. The concrete model of an abstraction is proved to be in a stuttering bisimulation with the one of the original system. We present the technique for a timed system modeled in Intermediate Language of the verification tool Verics. The reduction is made at the very beginning of the verification process and this makes it beneficial and effective in handling the state explosion problem.

1 Introduction

Many verification methods of time-critical systems have been defined over the last ten years. Model checking seems to be one of the most important methods due to its practical applications. Essentially, the method consists in determining whether a temporal formula expressing a property of a system is valid on its model (the state space) representing all the possible executions. Model checking has an advantage of being automatic but suffers from the so-called state explosion problem, i.e., the number of states in the model of a system can grow exponentially in the size of the system [14]. Many methods have been recently suggested to reduce state spaces that are considered in the verification process [3, 6, 7, 13]. One of the solutions to cope with this problem is to construct an abstract model of the system [5]. Such a model is usually much smaller than the concrete one.

To extract an abstract model of a system we propose to use a static analysis, namely a technique called path compression. The idea behind the path compression consists in identifying a path (or a set of paths) on which a process executes a sequence of transitions that do not influence a property being verified, and replacing it with a single transition. In this paper, we present the technique for a timed system modeled in Intermediate Language of the verification tool Verics [8]. Our method is property driven since it depends on a formula in question. An

* The work is partly supported by the Polish grant No. 1566/T11/2005/29.
abstraction is *exact* with respect to all the properties preserved by a stuttering bisimulation.

Path compression as an abstraction method is known for untimed systems. In [15] a method called *path reduction* is used to reduce state spaces by compressing computation paths of parallel while programs. The authors of [12] use the concept of *path slicing* to reduce control flow automata. The technique determines which subset of transitions along a given path to some particular location is relevant to reachability of the location at the path.

There are only a few papers using static analysis in verification of timed systems. The closest one to our work concerns the concept of *influence information* [4]. The technique can be understood as slicing I/O Timed Components, i.e., timed automata [2, 10] extended with interfaces. Our first attempt in using static analysis for timed systems concerns program slicing [11]. The technique allows to eliminate data irrelevant to the property in question. In the approach presented in [11] a path compression is also possible, but only for paths along which the time cannot elapse. The current paper presents a general method of timed and untimed path compression. It is worth pointing out that the best results are achieved by combining the program slicing and the path compression.

The rest of the paper is structured as follows. Section 2 introduces the syntax and semantics of Intermediate Language. In Section 3 we present a motivating example of a timed system and give an informal overview of our method. Section 4 describes the path compression technique for Intermediate Language in detail. Some experimental results are given in Section 5. The last section shortly concludes the paper.

2 Model and its semantics

In this section we introduce syntax and semantics of Intermediate Language. To this aim, we first define arithmetic and boolean expressions.

**Expressions.** Let $V$ be a finite set of integer *variables* and $B$ be a finite set of *buffers*. A buffer is a possibly empty sequence of integer values. The set $\text{Expr}(V)$ of all the *arithmetic expressions* over $V$ is defined by the following grammar:

\[
\text{expr} ::= m \mid y \mid \text{expr} \oplus \text{expr} \mid \neg \text{expr} \mid (\text{expr})
\]

where $m \in \mathbb{Z}$, $y \in V$, and $\oplus \in \{-, +, *, /\}$. The set $\text{Bexpr}(V)$ of all the *boolean expressions* over $V$ and $B$ is defined inductively as follows:

\[
\phi ::= \text{true} \mid \text{expr} \sim \text{expr} \mid \text{get}(b, y) \mid \phi \land \phi \mid \phi \lor \phi \mid \neg \phi \mid (\phi)
\]

where $\text{expr} \in \text{Expr}(V)$, $b \in B$, $y \in V$, and $\sim \in \{=, \neq, <, >, \leq, \geq\}$. The set $\text{Act}(V, B)$ of all the *actions* over $V$ and $B$ is defined as follows:

\[
\alpha ::= \epsilon \mid y := \text{expr} \mid \text{get}(b, y) \mid \text{put}(b, \text{expr}) \mid \alpha; \alpha
\]

By $/$ we denote integer division and $\mathbb{Z}$ denotes the set of integer numbers, $\mathbb{N}$ — the set of natural numbers, and $\mathbb{R}_+$ — the set of non-negative real numbers.

---

4 By $/$ we denote integer division and $\mathbb{Z}$ denotes the set of integer numbers, $\mathbb{N}$ — the set of natural numbers, and $\mathbb{R}_+$ — the set of non-negative real numbers.
where $\varepsilon$ denotes the empty sequence, $y \in V, b \in B$, and $expr \in Expr(V)$.

**Definition 1.** A program in Intermediate Language is a tuple $P = (V, B, \{P_i\}_{i \in \{1, \ldots, n\}})$, where $V$ is a set of variables, $B$ is a set of buffers, $n \in \mathbb{N}$, and a process $P_i$ is a tuple $(id_i, Q_i, \Sigma_i, T_i)$, where $id_i$ is the process name, $Q_i$ is a set of locations, $\Sigma_i$ is the initial location, $\Sigma_i$ is a set of labels, and $T_i$ is a set of the transitions of the form $(q, g, d, u, l, a, q')$, where:

- $q \in Q_i$ is called the source,
- $q' \in Q_i$ is called the target,
- $g \in Bexpr(V, B)$ is called the guard,
- $d \in \{(d_1, d_2), (d_1, d_2), (d_1, d_2), (d_1, d_2) \mid d_1 \in \mathbb{N} \land d_2 \in \mathbb{N} \cup \{\infty\}\}$ is called the delay allowed,\(^5\)
- $u \in \{true, false\}$ is called the urgency attribute,
- $l \in \Sigma_i$ is called the label, and
- $a \in Act(V, B)$ is called the action.

We write $source(t), guard(t), delay(t), urgent(t), label(t), action(t)$, and $target(t)$ for $q, g, d, u, l, a$, and $q'$ of the transition $t$, respectively. If the $delay(t)$ is of the form $(d_1, d_2)$, then the transition must be executed (strictly) after $d_1$ and (strictly) before $d_2$ time units since the process has reached the location $source(t)$. The urgency attribute $urgent(t)$ says whether the transition has the priority over time progress. When the urgency attribute is equal to true the transition $t$ has to be executed as soon as it is enabled. We work under the assumption that for an urgent transition $t$ the delay allowed should be equal to $[0, \infty)$. Notice that $label(t)$ can be either local or synchronous. A local label is unique in a program and a synchronous one is shared among some processes.

**Variable and buffer valuation.** Let $\mathbb{Z}$ be the set of integer values and $\mathbb{Z}^*$ be a set of the sequences of integer values from $\mathbb{Z}$. We use $\Omega$ to denote $\mathbb{Z} \cup \mathbb{Z}^*$. A valuation is a total mapping $v : V \cup B \rightarrow \Omega$ such that $v(V) \subseteq \mathbb{Z}$ and $v(B) \subseteq \mathbb{Z}$. A function $v$ associates a value from $\mathbb{Z}$ with each variable and a sequence of values with each buffer. We extend this mapping to expressions in the usual way. The expression $get(b, y)$ evaluates to true if the buffer $b$ is nonempty and to false otherwise. A valuation $v^0$ such that $v^0(b) = \varepsilon$ for all $b \in B$ (all buffers are empty) is called an initial valuation.

Satisfiability of a boolean expression $g \in Bexpr(V, B)$ by a given valuation $v$ (we write $v \models g$) is defined inductively as follows: $v \models true$, $v \models e_1 \sim e_2$ iff $v(e_1) \sim v(e_2)$, $v \models get(b, y)$ iff $v(b) \neq \varepsilon$, $v \models g_1 \land g_2$ iff $v \models g_1$ and $v \models g_2$, $v \models g_1 \lor g_2$ iff $v \models g_1$ or $v \models g_2$, and $v \models \neg g$ iff $v \models \neq g$. Let $v \xleftarrow{a} v'$ denote the execution of an action $a \in Act(V, B)$, where $v$ and $v'$ are defined according to the action structure as follows ($v[u/y]$ denotes the valuation $v$ with the value $u$ assigned to the variable or buffer $y$):

\[
\begin{align*}
\text{if } d_2 = \infty, \text{ then } \models \varepsilon, \text{ otherwise } \models \neq \varepsilon.
\end{align*}
\]
\[ b \in B, e \in \text{Expr}(V), u \in \mathbb{Z}, w \in \mathbb{Z}^+, v(e) = u, v(b) = w \]
\[ v \xrightarrow{\text{put}(b,e)} v[w;u/b] \]

The effect of the operation \( \text{get}(b, y) \) is the elimination of the first element of the buffer \( b \) and the assignment to \( y \) the value of this element. The operation \( \text{put}(b, e) \) appends the value of the expression \( e \) to the end of the buffer \( b \).

**Delay valuation.** Let \( Q = \bigcup_{i=1}^{n} Q_i \) denote the set of all the locations of all the processes. We define a delay valuation to be a total mapping \( \tau : Q \to \mathbb{R}_+ \) which associates a non-negative real number with each location.

For \( \delta \in \mathbb{R}_+ \), \( \tau + \delta \) denotes the delay valuation \( \tau' \), such that \( \tau'(q) = \tau(q) + \delta \) for all \( q \in Q \). By \( \tau^0 \) we denote the initial delay valuation such that \( \tau^0(q) = 0 \) for all \( q \in Q \). For \( Y \subseteq Q \), \( \tau[Y := 0] \) denotes the delay valuation \( \tau' \), such that \( \tau'(q) = 0 \) for \( q \in Y \) and \( \tau'(q) = \tau(q) \) for \( q \in Q \setminus Y \).

**Synchronization.** In Intermediate Language a system is described by a set of processes running parallel and communicating with each other via shared variables or buffers. Processes perform transitions with shared labels synchronously. Local transitions are interleaved. There are various definitions of a parallel composition. We choose the one determining multi-synchronization which requires that the transitions with a shared label have to be executed in all the processes containing this label. By \( \Sigma(l) = \{ 1 \leq i \leq n \mid l \in \Sigma_i \} \) we denote the set of numbers of processes that contain the label \( l \).

**State space.** A state \( s \) of a program is a tuple of the form \( (q_1, \ldots, q_n, v, \tau) \), where \( q_i \in Q_i \) for \( 1 \leq i \leq n \), \( v \in \Omega^{V \cup B} \), and \( \tau \in \mathbb{R}_+^Q \). For a transition \( t_i \in T_i \) and a state \( s \) we define: \( \text{enabled}(t_i, s) \equiv (\text{source}(t_i) = q_i) \land v \models \text{guard}(t_i) \), and \( \text{fireable}(t_i, s) \equiv \text{enabled}(t_i, s) \land \tau(\text{source}(t_i)) \in \text{delay}(t_i) \).

By \( \{ a_j \}_{j \in Z} \) we denote the sequence of the actions \( a_j \), where \( j \in Z \subseteq \{ 1, \ldots, n \} \), such that the actions are arranged according to the numerical order of their indices.

**Definition 2.** The semantics of a program \( \mathcal{P} = (V, B, \{ P_i \}_{i \in \{ 1, \ldots, n \}} \) with \( P_i = (id_i, Q_i, q_i^0, \Sigma_i, T_i) \) for an initial valuation \( v^0 : V \cup B \to \Omega \) is a labeled transition system \( S = (S, s^0, \Sigma, \rightarrow) \) such that:

- \( S = Q_1 \times \cdots \times Q_n \times \Omega^{V \cup B} \times \mathbb{R}_+^Q \) is the set of states,
- \( s^0 = (q_1^0, \ldots, q_n^0, v^0, \tau^0) \in S \) is the initial state,
- \( \Sigma = \bigcup_{i=1}^{n} \Sigma_i \cup \mathbb{R}_+ \)
- \( \rightarrow \subseteq S \times \Sigma \times S \) is the smallest transition relation defined as follows:
  - let \( s = (q_1, \ldots, q_n, v, \tau) \) and \( s' = (q'_1, \ldots, q'_n, v', \tau') \), \( s \overset{\delta}{\rightarrow} s' \) iff for all \( i \in \Sigma(l) \) there exists a transition \( t_i \in T_i \) such that \( \text{fireable}(t_i, s) \), \( l = \text{label}(t_i) \), \( \tau'_i = \text{target}(t_i) \), \( v \xrightarrow{\text{action}(t_i)} v' \), \( \tau' = \tau[q'_i]_{i \in \Sigma(l)} := 0 \) and \( q'_j = q_j \) for all \( j \notin \{ 1, \ldots, n \} \setminus \Sigma(l) \),
  - let \( s = (q_1, \ldots, q_n, v, \tau) \) and \( s' = (q_1, \ldots, q_n, v, \tau + \delta) \), for \( \delta \in \mathbb{R}_+ \)
    \( s \overset{\delta}{\rightarrow} s' \) iff \( \text{enabled}(t, s) \land \text{urgent}(t, s) \lor (\text{fireable}(t, s) \land \neg \text{fireable}(t, s')) \) does not hold for any \( t \in \bigcup_{i=1}^{n} T_i \).
Initially, all the buffers are empty and all the variables have some initial values. At a state \( s = (q_1, \ldots, q_n, v, \tau) \) the system can:

- either execute a synchronous or a local fireable transition; the delay values
  of locations \( q_i \) for each \( i \)-th process executing the transition \( (i \in \Sigma(l)) \) are
  reset to zero, or
- let time \( \delta \) pass and move to the state \( (q_1, \ldots, q_n, v, \tau + \delta) \), unless there is an
  enabled transition with the urgency attribute set or the time passage
  would make any transition not fireable.

**Runs.** For \( (q_1, \ldots, q_n, v, \tau) \in S \), let \( (q_1, \ldots, q_n, v, \tau) + \delta \) denote \( (q_1, \ldots, q_n, v, \tau + \delta) \). An \( s-\)run of \( \mathcal{P} \) is a finite or infinite sequence of states: \( s_0 \overset{\delta_0}{\rightarrow} s_0 + \delta_0 \overset{a_0}{\rightarrow} s_1 \overset{\delta_1}{\rightarrow} s_1 + \delta_1 \overset{a_2}{\rightarrow} \ldots \), where \( s_0 = s \in S, a_i \in \Sigma \), and \( \delta_i \in \mathbb{R}_+ \), for each \( i \geq 0 \). A run
  is called *progressive* if \( \sum_{i \in \mathbb{N}} \delta_i \) is infinite. A program \( \mathcal{P} \) is *progressive* if all its
  \( s^0 \)-runs are progressive. For simplicity in our work we consider only progressive
  programs.

**Model.** Let \( S = (S, s^0, \Sigma, \rightarrow) \) be the labeled transition system of a program
  \( \mathcal{P} = (V, B, \{P_i\}_{i \in \{1, \ldots, n\}}) \) with \( P_i = (id_i, Q_i, q_i^0, \Sigma_i, T_i) \). A
  propositional variable
  is of the form: \( p_{i,q} \) for some \( 1 \leq i \leq n \), where \( q \in Q_i \), \( p_{\text{empty}}(b) \) for \( b \in B \),
  or \( p_{e_1} \sim e_2 \), where \( e_1, e_2 \in \text{Expr}(V) \) and \( \sim \) is a relational operator. We denote
  the set of all propositional variables by \( PV \). In order to reason about systems
  represented as a set of processes we define a *labeling* function \( \mathcal{V}: S \rightarrow 2^{PV} \). For
  \( s = (q_1, \ldots, q_n, v, \tau) \in S \), \( \mathcal{V}(s) \) is defined as follows: \( p_{e_1} \sim e_2 \in \mathcal{V}(s) \) if \( v \models e_1 \sim e_2 \),
  \( p_{\text{empty}}(b) \in \mathcal{V}(s) \) iff \( v(b) = \varepsilon \), and \( p_{i,q} \in \mathcal{V}(s) \) iff \( q_i = q \). The model
  is the pair \( \mathcal{M} = (S, \mathcal{V}) \).

Let \( PV_\varphi \subseteq PV \) be the set of propositional variables used in the formula
  \( \varphi \). We call a location \( q \in Q_i \) observable if it is initial or it appears in some
  proposition of \( PV_\varphi \), i.e., \( q = q_i^0 \) or \( p_{i,q} \in PV_\varphi \).

### 3 Motivating example

As the example we present a system that exploits the well known *Fischer’s mutual
  exclusion protocol* [1] to ensure mutual exclusion. The system consists of
  two processes using the same shared resource. Each of the processes is modeled
  as shown in Fig. 1. The global variable \( Z \) is used to schedule an access to the
  resource. The shared resource is represented by another global variable \( Y \). Be-
  sides competing for the resource the processes can perform some local actions.
  For the sake of clarity, we skip the labels of the transitions since all of them are
  local (i.e., non synchronous). The urgency attribute is not set for any transition.
  The delays of the form \([0, \infty) \) are omitted.

Before we show in detail how to reduce a system using the technique of path
  compression, we would like to give some intuitions and sketch our method. The
  method is property driven, that is it depends on a reachability property to be
  verified. For the system presented we want to verify reachability of the formula
\[ \varphi = \bigvee_{1 \leq i < j \leq n} (i.\text{critical}_i \land j.\text{critical}_j) \]

saying that the mutual exclusion property is violated. The important point to note here is that the validity of the formula in the model does not depend on values of the variables \( Y, k_1, \) and \( k_2. \)

**Sketch of the method.** The first step is to use a slicing algorithm to obtain a set of the *relevant* variables, i.e., the variables that can influence a property in question. The slicing algorithm for Intermediate Language was defined in \([11]\). After completing the slicing, the path compression algorithm is executed.

The slice of our example constructed according to the slicing algorithm is shown in Fig. 2. Comparing to the original one, the variables \( Y, k_1, \) and \( k_2 \) have been removed and so have been operations on them as they are irrelevant to the mutual exclusion property.

**Path compression.** The idea behind the path compression consists in replacing a path (or a set of paths) with a single transition. Such a path which can be replaced, called *compressible*, must satisfy certain conditions. First of all, all the locations on the path are not observable. Next, the path cannot contain a cycle, which means that none of its locations occurs twice. Finally, all the transitions of the path except for the last one have empty actions, guards equal to true and local labels. If there are many compressible paths between a pair of locations, then all of them can be replaced by one transition provided that the last transitions of the paths are equal or at least: have the same enabling...
conditions, perform the same operations on variables, and either have the same labels, or all their labels are local. In our example we have two compressible paths that fulfill all of the above conditions: \( \text{init}_i \rightarrow \text{reset}_i \rightarrow \text{idle}_i \rightarrow \text{trying}_i \) and \( \text{init}_i \rightarrow \text{inc}_i \rightarrow \text{idle}_i \rightarrow \text{trying}_i \). Both of them can be replaced by the transition \( \text{init}_i \rightarrow \text{trying}_i \) as shown in Fig. 3.

![Fig. 3. Process of Fig. 2 after path compression](image)

4 Path compression

In model checking applications we typically check if a given temporal logic formula \( \varphi \) holds in a system \( P \). What we would like to do is to generate a reduced system \( P' \) such that \( \varphi \) holds in \( P' \) if and only if it holds in \( P \). It is clear that \( P' \) needs to preserve behavior of these parts of \( P \) only, which can influence the validity of the formula \( \varphi \).

As before let \( P \) be a process, where \( Q \) and \( T \) are the sets of its locations and transitions, respectively. For \( q \in Q \) let \( \text{out}(q) \subseteq T \) be the set of its outgoing transitions and \( \text{in}(q) \subseteq T \) be the set of its incoming transitions. A location \( q \in Q \) is called an ending location, if \( \text{out}(q) = \emptyset \).

**Paths.** A path in the process \( P \) from a location \( q' \in Q \) to a location \( q'' \in Q \) is a sequence of locations and transitions of the form \( q_1t_2q_2\ldots t_mq_m \) such that \( q_1 = q', q_m = q'', m \geq 2, q_j \in Q, \) and \( t_j \in \text{out}(q_{j-1}) \cap \text{in}(q_j) \) for all \( 1 < j \leq m \). We say that a path \( q_1t_2q_2\ldots t_mq_m \) contains (or goes through) a location \( q \) if \( q_j = q \) for some \( 1 \leq j \leq m \) (similarly for transitions). A path contains a cycle if it contains one of its locations twice. We call a path maximal if it contains an ending location or a cycle. A path is simple if it does not contain a cycle. By \( \Pi(q',q'') \) we denote the set of all the simple paths from the location \( q' \) to the location \( q'' \). By \( \text{locs}(\Pi(q',q'')) \subseteq Q \) and \( \text{trans}(\Pi(q',q'')) \subseteq T \) we denote the set of all the locations and the set of all the transitions contained in the paths of \( \Pi(q',q'') \), respectively.

**Domination and post-domination.** We say that a location \( q \in Q \) dominates a location \( q' \in Q \) if every simple path from the initial location \( q^0 \) to \( q' \) goes through \( q \). Also, a location \( q' \) post-dominates a location \( q \) if every maximal path from \( q \) goes through \( q' \). Notice that the above definitions are not symmetrical due to possibly missing an ending location. In the process in Fig. 1 the location \( \text{init}_i \) dominates the location \( \text{reset}_i \), but the location \( \text{reset}_i \) does not dominate...
the location \textit{idle}_i. Also the location \textit{idle}_i post-dominates the location \textit{reset}_i, but the location \textit{reset}_i does not post-dominate the location \textit{init}_i.

We say also that a process has a reducible control flow, if its transitions can be divided into two disjoint groups: one forms an acyclic graph and the other consists of transitions whose sources dominate their targets. In such a process each cycle can be entered through exactly one location. In our work we consider only processes with reducible control flow. This is a minor simplification since non-reducible control flows rarely occur in practice. The transitions of all the processes presented in Section 3 can be divided into two required groups: one is composed of the single transition \textit{critical}_i \rightarrow \textit{init}_i and the other contains the rest of transitions. Note also that each cycle in the presented processes is entered only through the location \textit{init}_i.

\textbf{Path delays.} Let \textit{delay}(t) = [d_1, d_2],\(^6\) where \(d_1 \in \mathbb{N}\) and \(d_2 \in \mathbb{N} \cup \{\infty\}\), be the delay allowed of the transition \(t \in T\). We define the following attributes: \(lb(\textit{delay}(t)) = d_1\) and \(ub(\textit{delay}(t)) = d_2\). For a path \(\pi = q_1t_2q_2 \ldots t_mq_m\), we can define minimal and maximal delay on the path as follows:\(^7\) \(\text{min}_{\text{delay}}(\pi) = \sum_{j=2}^{m} lb(\textit{delay}(t_j))\) and \(\text{max}_{\text{delay}}(\pi) = \sum_{j=2}^{m} ub(\textit{delay}(t_j))\). Notice that in our example in Fig. 1 in Sect. 3 \(\text{min}_{\text{delay}}(\textit{init}_i \rightarrow \textit{reset}_i \rightarrow \textit{idle}_i) = 0\) and \(\text{max}_{\text{delay}}(\textit{init}_i \rightarrow \textit{reset}_i \rightarrow \textit{idle}_i) = d\).

\textbf{Definition 3.} Let \(q', q'' \in Q\) of \(P\). We call the set of paths \(\Pi(q', q'')\) compressible iff the following conditions are satisfied:

1. for each location \(q \in \text{locs}(\Pi(q', q'')) \setminus \{q''\}:
   (a) \(q\) is not observable,
   (b) \(q'\) dominates \(q\),
   (c) \(q''\) post-dominates \(q\),
   (d) \(\text{source}(t) \in \text{locs}(\Pi(q', q'')) \setminus \{q''\}\) for each transition \(t \in \text{in}(q)\),
2. for each transition \(t \in \text{trans}(\Pi(q', q'')) \setminus \text{in}(q'')\):
   (a) \(\text{label}(t)\) is local (non-synchronous),
   (b) \(\text{guard}(t)\) is equal to \(\text{true}\),
   (c) \(\text{action}(t)\) is equal to \(\epsilon\),
3. for each two transitions \(t, t' \in \text{in}(q'')\):
   (a) \(\text{label}(t) = \text{label}(t')\) or \(\text{label}(t)\) and \(\text{label}(t')\) are local,
   (b) \(\text{guard}(t) = \text{guard}(t')\),
   (c) \(\text{action}(t) = \text{action}(t')\),
   (d) \(\text{urgent}(t) = \text{urgent}(t') = \text{false}\),
4. for each \(d \in \mathbb{R}_+\) such that \(\min_{\pi \in \Pi(q', q'')} (\text{min}_{\text{delay}}(\pi)) \leq d\) and \(d < \max_{\pi \in \Pi(q', q'')} (\text{max}_{\text{delay}}(\pi))\), there exists a path \(\pi \in \Pi(q', q'')\) such that \(\text{min}_{\text{delay}}(\pi) \leq d < \text{max}_{\text{delay}}(\pi)\).

For two sets of compressible paths \(\Pi\) and \(\Pi'\), we say that \(\Pi \subseteq \Pi'\) if \(\text{locs}(\Pi) \subseteq \text{locs}(\Pi')\) and \(\text{trans}(\Pi) \subseteq \text{trans}(\Pi')\). A set \(\Pi\) of compressible paths is maximal if \(\Pi \not\subseteq \Pi'\) for any other set \(\Pi' \neq \Pi\) of compressible paths.

\(^6\) For sake of simplicity we only deal with one of the four possible forms of intervals.
\(^7\) \(d + \infty = \infty\) and \(\infty + d = \infty\) for an arbitrary \(d\).
Item 1(c) ensures that no cycle is contained in the compressible path set \( \Pi(q', q'') \) and no path goes “out” of \( \Pi(q', q'') \) by a location different than \( q'' \). The dual property, that no path enters \( \Pi(q', q'') \) through a location different than \( q' \), is guaranteed by 1(b) and 1(d) (the last item eliminates for example a situation where a transition from \( q'' \) goes to some location contained in \( \Pi(q', q'') \)). Note that we do not require a transition \( t \in \text{trans}(\Pi(q', q'')) \), which does not enter the location \( q'' \), to have the urgency attribute equal to \( \text{false} \). Since \( \text{guard}(t) \) is equal to \( \text{true} \), it follows that the transition \( t \) has to be executed as soon as the process enters the source location of \( t \). Therefore in such a case \( \text{urgent}(t) = \text{true} \) is equivalent to \( \text{delay}(t) = [0, 0] \). The last condition is necessary to keep the delay of a transition, which will replace the compressible path set, representable by a time interval of the form defined in Section 2.

Let us mention that from the above definition it follows that the sets of transitions of two maximal sets of compressible paths are disjoint.

**Definition 4.** The reduced process is obtained from the original one, where each maximal compressible set of path \( \Pi(q', q'') \) is replaced by a fresh transition \( t \). Let \( t' \) be an arbitrary transition entering \( q'' \). We define \( t \) as follows:

- source\((t) = q' \),
- target\((t) = q'' \),
- label\((t) = \text{label}(t') \),
- guard\((t) = \text{guard}(t') \),
- \( \text{delay}(t) = [\min_{\pi \in \Pi(q', q'')} (\text{min}_\pi(\pi)), \max_{\pi \in \Pi(q', q'')} (\text{max}_\pi(\pi))] \),
- \( \text{urgent}(t) = \text{false} \),
- \( \text{action}(t) = \text{action}(t') \),

For the label of the transition \( t \) by Def. 3.3(a) we can choose the label of an arbitrary \( t' \) entering \( q'' \) since the labels of all transitions entering \( q'' \) are equal or all of them are local. Similarly for the guard and the action of \( t \) (Def. 3.3(b)-(c)).

**Correctness.** We say that a location \( q \in Q_i \) is reducible to the location \( q' \) if there exists a location \( q'' \in Q_i \) such that \( \Pi(q', q'') \) is a maximal compressible path set and \( q \notin \text{locs}(\Pi(q', q'')) \setminus \{q''\} \). For \( q \in Q_i \) we define \( \text{red}(q) \) as follows:

if there exists a location \( q' \) such that \( q \) is reducible to \( q' \), then \( \text{red}(q) = q' \),

otherwise \( \text{red}(q) = q \) (uniqueness of the location \( q' \) follows from maximality of \( \Pi(q', q'') \) and Def. 3.1b).

Let \( S = (S, s^0, \Sigma, \rightarrow) \) be the labeled transition system for a program \( P \) and \( S' = (S', s'^0, \Sigma', \rightarrow') \) be the labeled transition system for the program \( P' \), where all the processes are reduced with respect to the set of propositional variables \( PV \). The labeling functions \( \mathcal{V} : S \rightarrow 2^{PV} \) and \( \mathcal{V}' : S' \rightarrow 2^{PV} \) are defined as in Section 2. Let \( M = (S, \mathcal{V}) \) and \( M' = (S', \mathcal{V}') \). Our aim is to show that \( M \) and \( M' \) are stuttering bisimilar according to the following definition.

**Definition 5.** [9] A relation \( \equiv_b \subseteq S \times S' \) is a stuttering simulation between \( M \) and \( M' \) if the following conditions hold: \( s_0 \equiv_b s'_0 \) and if \( s \equiv_b s' \), then \( \mathcal{V}(s) = \mathcal{V}'(s') \) and for every maximal path \( \sigma \) of \( M \) that starts at \( s \), there is a maximal
path \( \sigma' \) in \( M' \) that starts at \( s' \), a partition \( B_1, B_2, \ldots \) of \( \sigma \), and a partition \( B'_1, B'_2, \ldots \) of \( \sigma' \) such that for each \( j \geq 1 \), blocks \( B_j \) and \( B'_j \) are nonempty and finite, and every state in \( B_j \) is related by \( \cong_B \) to every state in \( B'_j \). A relation \( \cong_B \) is a stuttering bisimulation if both \( \cong_B \) and \( \cong_B^T \) (the transpose of \( \cong_B \)) are stuttering simulations.

**Definition 6.** Let \( \cong \subseteq S \times S' \) be the relation s.t. for each \( s = (q_1, \ldots, q_n, v, \tau) \in S \), and each \( s' = (q'_1, \ldots, q'_n, v', \tau') \in S' \) we have, \( s \cong s' \) iff the following holds:

1. \( q'_i = \text{red}(q_i) \), for each \( 1 \leq i \leq n \),
2. \( v' = v \), and
3. \( \tau'(q'_i) = \tau(\text{red}(q_i)) \) for each \( 1 \leq i \leq n \).

**Lemma 1.** The relation \( \cong \subseteq S \times S' \) is a stuttering bisimulation between the two models \( M = (S, \mathcal{V}) \) and \( M' = (S', \mathcal{V}') \).

**Proof.** (a sketch). Let \( s_k = (q_1^k, \ldots, q_n^k, v^k, \tau^k) \) and \( s'_k = (q'_1^k, \ldots, q'_n^k, v'^k, \tau'^k) \) for \( k, l \in \mathbb{N} \). It is easy to check that \( s_0 \cong s'_0 \). From Def. 6.1-2 and Def. 3.1(a), it follows that \( \mathcal{V}(s) = \mathcal{V}'(s') \) for \( s \cong s' \). (\( \Rightarrow \)) Consider a maximal path \( \sigma \) in \( M \). The construction of \( \sigma' \) is inductive. Let \( s'_i \) be the last state of an already constructed prefix of \( \sigma' \), \( s_k \) be a state of \( \sigma \) such that \( s_k \cong s'_i \) and \( s_k \xrightarrow{a_k} s_{k+1} \) be the next transition in \( \sigma \). There are two cases. Case 1: The state \( s_{k+1} \) belongs to the same block as \( s_k \) does iff a transition of the label \( a_k \) is contained in some mcps\(^8\) and does not enter its last location. We show that \( s_{k+1} \cong s'_i \). By Def. 3.2(a) \( a_k \) is a local transition\(^9\) of some process \( i \). Since \( q_i^k \) and \( q_i^{k+1} \) are contained in the same path of the \( i \)-th process, thus \( q_i^k = \text{red}(q_i^k) = \text{red}(q_i^{k+1}) \), which satisfies Item 1 of Def. 6. Item 2 is satisfied too, since by Def. 3.2(c) \( v^{k+1} = v^k \). Item 3 follows from \( \tau^{k+1} = \tau^k \). Case 2: For all other transitions \( s_{k+1} \) belongs to a new block. We show that there exists a transition \( a_i \) such that \( s'_i \xrightarrow{a_i} s'_{i+1} \) and \( s_{k+1} \cong s'_{i+1} \).

If \( a_k \) is an action transition which is not included in any mcps, then the proof is straightforward. If \( a_k \) is a local transition entering the last location of a path \( \pi \) of some mcps \( \mathcal{I} \) of the \( i \)-th process, then \( \text{red}(q_i^k) \) is the first location of \( \pi \). In this case let \( a_i \) be a transition constructed according to Def. 4 for \( \mathcal{I} \). We skip the proof that \( a_i \) is fireable at \( s'_i \). Since by Def. 3.1(c) \( q_i^{k+1} = \text{target}(t) = \text{target}(t') = q_i^{k+1} \), it follows that Item 1 of the Def. 6 holds for \( s_{k+1} \) and \( s'_{i+1} \). Next, by Def. 3.3(c) \( \text{action}(t) = \text{action}(t') \). Since \( v_i^k = v_i^{k+1} \), it follows that after the execution of the same sequence of operations \( v_i^{k+1} = v_i^{k+1} \), which satisfies Item 2. Item 3 is satisfied since \( \tau_i^{k+1} = \tau_i^{k+1} \). We leave it to the reader to check the other cases. (\( \Leftarrow \)) The main idea is the following: a transition in \( \sigma' \) corresponds to a sequence of transitions along a path of a suitable mcps interleaved with timed transitions (analogous to Case 1 above) or to a single transition (analogous to Case 2).

\(^8\) An abbreviation for maximal compressible path set.

\(^9\) We use \( a_k \) instead of “a transition of the label \( a_k \)”.
It is easy to show that if there exists a stuttering bisimulation between two models, then these models satisfy the same reachability properties and \( \text{LTL}_{\leq} \) formulas. Thus, a consequence of Lemma 1 is that the model of a system satisfies an \( \text{LTL}_{\leq} \) formula \( \varphi \) if and only if the model of its reduct with respect to the set of propositions \( PV_\varphi \) satisfies \( \varphi \).

5 Experimental results

We present experimental results for the example in Sect. 3 obtained with the verification tool Verics, on the machine equipped with the processor Intel Pentium 4 – 2.8 GHz, 2 GB of main memory, and the Linux Red Hat operating system. We have checked whether a state satisfying \( \varphi \) (see Sect. 3) is reachable in the following three systems: original (\( O \), Fig. 1), sliced (\( S \), Fig. 2), and sliced and then reduced according to the path compression technique (\( S+PC \), Fig. 3). The experiments have been performed for various numbers of processes (\( n \)) and values of parameters \( d \) and \( D \). In Fig. 4 we present the results for \( d = 2 \) and \( D = 1 \). We compare the sizes of the models and amounts of time needed to complete our tests, i.e. time of translation to a propositional formula by the BBMC module and time of satisfiability verification of the formula by ZChaff. The cases when memory have been exhausted are denoted by *.

![Fig. 4. Experimental results for the examples from Section 3](image)

The experimental results show that all the measured values are smaller for the sliced system than for the original. However, the most significant reduction is obtained when the slicing technique is combined with the path compression.

6 Conclusions

In the paper, a method of abstraction exploiting static structure of timed systems has been presented. The experiments confirm that the application of the method leads to significant reduction in the time and the memory consumed by the verification tool. The most important advantage of our approach is that it can be used prior to any existing tools analyzing timed systems including symbolic ones. It is also orthogonal to other abstraction methods and can be combined with them to yield a more powerful tool in terms of state space reduction.

---

10 Linear Time Temporal Logic without the next step operator.

11 The preservation of mutual exclusion is ensured for \( d < D \).
References