Optimal Clock Allocation for a Class of Timed Automata

Neda Saeedloei\(^1\) and Feliks Klużniak\(^2\)

1 Southern Illinois University, neda@cs.siu.edu
2 Logic Blox, feliks.kluzniak@logicblox.com

Abstract. We address the problem of allocating a minimal number of clocks to timed automata. To make the problem tractable we assume that all locations are reachable. We identify a fairly general class of timed automata, \(\mathit{TA}_{\mathit{DS}}\), for which there is an algorithm whose complexity is quadratic in the size of the graph.

1 Introduction

Minimizing the number of clocks in timed automata is important, as it affects the complexity of the verification problem [1, 2].

As is well known, it is in general undecidable whether the number of clocks in a given timed automaton \(\mathcal{A}\) can be reduced while preserving the language accepted by \(\mathcal{A}\) [5]. The best attempt at tackling the problem is based on constructing a timed automaton that is bisimilar to \(\mathcal{A}\) [6]. The constructed automaton may have more locations, but has the minimal number of clocks in the entire class of timed automata that are bisimilar to \(\mathcal{A}\). The growth in the number of locations can be exponential in the number of clocks in \(\mathcal{A}\). The computational complexity of that algorithm is 2-EXPTIME.

The current paper makes two principal contributions:

1. We use techniques known from compiler technology to tackle the problem directly, without changing the graph of the original automaton.
2. We identify a class of timed automata, \(\mathit{TA}_{\mathit{DS}}\), for which those techniques are particularly effective: an optimal allocation of clocks can be obtained at a cost that is quadratic in the size of the underlying graph.

Given a timed automaton \(\mathcal{A} \in \mathit{TA}_{\mathit{DS}}\), we abstract from the specific syntax of the constraints and consider only the clocks that are used in them. The reason for this is that in our analysis we do not want to concern ourselves with “pathological” cases in which the required number of clocks is strongly affected by some particular form of the constraints. An instance of such a case is an automaton with a clock \(c\) such that all the constraints on \(c\) are of the form \(c \geq 0\), which is always true. Clearly, in an optimal clock allocation \(c\) can be safely removed. Opportunities for optimisation arising out of the fact that the constraints have
some particular form should of course be pursued whenever it is practical to do so. Such optimisations are, however, outside the scope of the current paper. It would probably be best to perform them before clock optimisation.

The class of timed automata considered in this paper, $TA_{DS}$, is described in Sec. 3. The class is interesting primarily because it is amenable to efficient treatment, but it is not without practical value [10, 9].

Given a timed automaton in $TA_{DS}$, our clock allocation method replaces the clocks with new ones, minimizing their number, without changing the graph or the language of the original automaton. The method is based on liveness analysis of clocks: this allows us to use one clock to replace two clocks whose ranges are disjoint. (The terms “liveness” and “range” are borrowed from the field of compiler technology.)

2 Timed Automata

We now present a brief overview of timed automata [2].

For a set $C$ of clock variables, the set $\Phi(C)$ includes clock constraints of the form $c \sim a$, where $\sim \in \{\leq, \geq, <, >, =\}$, $c \in C$, and $a$ is a constant in the set of rational numbers, $\mathbb{Q}$.

A timed automaton is a tuple $A = \langle E, Q, Q_0, Q_f, C, R \rangle$, where

- $E$ is a finite alphabet;
- $Q$ is the (finite) set of locations;
- $Q_0 \subseteq Q$ is the set of initial locations;
- $Q_f \subseteq Q$ is the set of final locations;
- $C$ is a finite set of clock variables;
- $R \subseteq Q \times Q \times E \times 2^C \times 2^{\Phi(C)}$ is the set of transitions of the form $(q, q', e, \lambda, \phi)$, where the set $\lambda \subseteq C$ gives the clocks to be reset with this transition, and $\phi$ is a set of clock constraints over $C$.

A time sequence $\tau = \tau_1 \tau_2 ...$ is an infinite sequence of (time) values $\tau_i \in \mathbb{R}_{\geq 0}$, satisfying two requirements:

- Monotonicity: $\tau$ increases strictly monotonically, i.e., $\tau_i < \tau_{i+1}$ for all $i \geq 1$.
- Progress: For every $t \in \mathbb{R}$, there is some $i \geq 1$ such that $\tau_i > t$.

A timed word over an alphabet $E$ is a pair $(\sigma, \tau)$ where $\sigma = \sigma_1 \sigma_2 ...$ is an infinite word over $E$ and $\tau$ is a time sequence.

A clock interpretation for a set $C$ of clocks assigns a value in $\mathbb{R}_{\geq 0}$ to each clock; that is, it is a mapping from $C$ to $\mathbb{R}_{\geq 0}$. We say that a clock interpretation $\nu$ for $C$ satisfies a set of clock constraints $\phi$ over $C$ iff every clock constraint in $\phi$ evaluates to true after replacing each clock variable $c$ with $\nu(c)$.

For $\tau \in \mathbb{R}, \nu + \tau$ denotes the clock interpretation which maps every clock $c$ to the value $\nu(c) + \tau$. For $Y \subseteq C, [Y \mapsto \tau] \nu$ denotes the clock interpretation for $C$ which assigns $\tau$ to each $c \in Y$, and agrees with $\nu$ over the rest of the clocks.

---

3 We will follow the usual convention and use the word “clocks” instead of “clock variables”.
A run $\rho$ of a timed automaton over a timed word $(\sigma, \tau)$ is an infinite sequence of the form

$$\rho : \langle q_0, \nu_0 \rangle \xrightarrow{\sigma_1} \langle q_1, \nu_1 \rangle \xrightarrow{\sigma_2} \langle q_2, \nu_2 \rangle \xrightarrow{\sigma_3} \cdots$$

with $q_i \in Q$ and $\nu_i \in [C \mapsto \mathbb{R}^{\geq 0}]$, for all $i \geq 0$, satisfying two requirements:

- $q_0 \in Q_0$, and $\nu_0(c) = 0$ for all clocks $c \in C$.
- for every $i \geq 1$ there is a transition in $R$ of the form $(q_{i-1}, q_i, \sigma_i, \lambda_i, \phi_i)$, such that $(\nu_{i-1} + \tau_i - \tau_{i-1})$ satisfies $\phi_i$, and $\nu_i$ equals $[\lambda_i \mapsto 0][\nu_{i-1} + \tau_i - \tau_{i-1}]$.

The set $\text{inf}(\rho)$ consists of $q \in Q$ such that $q = q_i$ for infinitely many $i \geq 0$ in the run $\rho$.

Different notions of acceptance have been proposed. For example, a run $\rho$ of a timed Büchi automaton over a timed word $(\sigma, \tau)$ is an accepting run iff $\text{inf}(\rho) \cap Q_T \neq \emptyset$.

The language of $\mathcal{A}$, is the set $\{(\sigma, \tau) \mid \mathcal{A} \text{ has an accepting run over } (\sigma, \tau)\}$.

### 3 The Class $\mathcal{TA}_{DS}$

We now characterise $\mathcal{TA}_{DS}$, the class of timed automata considered in this paper.

Let $r = (q, q', e, \lambda, \phi) \in R$ be a transition. We define $\text{source}(r) = q$ and $\text{target}(r) = q'$. The outgoing/incoming transitions of a location $s \in Q$ are denoted by $\text{out}(s) = \{r \in R \mid s = \text{source}(r)\}$ and $\text{in}(s) = \{r \in R \mid s = \text{target}(r)\}$.

Let $\mathcal{A}$ be a timed automaton with a unique initial location. If $s$ and $q$ are locations in $\mathcal{A}$, then $s$ dominates $q$ if and only if all paths from the initial location to $q$ pass through $s$ [7]. We denote the dominance relation between locations in $\mathcal{A}$ by $\succeq$: $s \succeq q$ iff $s$ dominates $q$ (we also say that $q$ is dominated by $s$). We write $s \succeq q$ to denote that $s \succeq q$ and $s \neq q$. We extend the definition of dominated locations to dominated transitions: a transition $r$ is dominated by location $s$ iff $s \succeq \text{source}(r)$.

In the timed automaton of Fig. 1, $s_0, s_1, s_2, s_5$ and $s_6$ are all the locations that dominate $s_6$. Transition $r_7$ is dominated by $s_0, s_1, s_2$ and $s_5$.

In the rest of the paper we assume that every timed automaton $\mathcal{A}$ has an associated injective partial labelling function $L : Q \rightarrow N_L$, where $N_L \subset \{0, 1, 2, \ldots\}$ is the set of labels (values of $L$) used for $\mathcal{A}$.

To keep the presentation simple and make a distinction between the clocks in the original and target timed automata, we assume that the clocks in the original automaton belong to the set $V = \{t_0, t_1, t_2, \ldots\}$. Our clock allocation method will replace these with a set of clocks disjoint from $V$.

**Definition 1** A timed automaton belongs to the class $\mathcal{TA}_{DS}$ if and only if it satisfies the following three restrictions:

1. It has a unique initial location, $q^0$. Every location can be reached from $q^0$.
2. Clock $t_j$ is reset only on transitions in $\text{out}(s)$, where $L(s) = j$; moreover, $t_j$ is reset on all the transitions in $\text{out}(s)$.

\[\text{If a transition leads only to paths on which } t_j \text{ is not used, the reset (and the clock that is reset) will be eliminated by our algorithms.}\]
3. A clock constraint on a transition \( r \) can contain an occurrence of \( t_j \) only if \( j \in N_L \) and \( L^{-1}(j) > \text{source}(r) \).

Restriction 3, which we call the dominance assumption, means that if \( t_j \sim a \) is a clock constraint on a transition \( r \in \text{out}(s) \), then the value of \( t_j \) represents the amount of time that has elapsed since leaving a location \( q \), where \( q \succ s \) and \( L(q) = j \). This guarantees that all clocks are well-defined: a reference to \( t_j \) in a clock constraint on \( r \) is always preceded by a reset, on every path from \( q_0 \) to \( r \).

4 Finding an Optimal Allocation of Clocks

Our objective is to transform a timed automaton \( A \in \text{TAD}_S \) to an equivalent automaton \( A' \) with the same graph and an optimal (i.e., the smallest possible) number of clocks. This is done in four steps:

- **Liveness analysis** identifies the ranges of clocks in \( A \).
- The original clocks are replaced with new clocks in a way that takes ranges into account but minimizes the number of clocks.
- Old clock resets are deleted and new clock resets are generated.
- Existing clock constraints are rewritten in terms of the new clocks.

The first two steps will be described below (we also have a working prototype implementation). The last two steps are so simple that we will not go into the details.

The clock allocation problem is reminiscent of the general register allocation problem \([8, 3]\), but it is simpler and allows a more effective solution. It is simpler, because there is no limit on the number of clocks, and the value of a clock cannot be copied to another clock.
4.1 Liveness Analysis of Clocks

Liveness analysis of clocks is performed by Algorithm 1. The algorithm determines, in effect, the “ranges” of all the clocks in the original timed automaton. Informally, a range for clock $t_j$ is a set of transitions on which the value of $t_j$ must be available.

Before presenting the algorithm, we introduce some auxiliary definitions.

Assume $A = (E, Q, \{q^0\}, Q_f, V, R, L)$ is the original timed automaton.

For a transition $r = (s, q, e, \lambda, \phi) \in R$, we will use $\lambda_r$ and $\phi_r$ to denote the sets of clocks to be reset on $r$ and the clock constraints on $r$, respectively.

Let $N = \{j \mid t_j \sim a \in \phi\}$, where $(s, q, e, \lambda, \phi) \in R$. This is the set of clock numbers, i.e., of the indices of clocks that are referred to on all transitions in $R$. Notice that $N \subseteq N_L$.

Let $p = r_1...r_k$ be a path. We define $\text{transitions}(p) = \{r_1,...,r_k\}$.

We also define the following functions:

- $\text{clock_ref} : R \rightarrow 2^N$ maps transition $r$ to the set $\{j \mid t_j \sim a \in \phi_r\}$. Intuitively, $\text{clock_ref}(r)$ is the set of clocks which are referred to in the constraints on $r$.

- $\text{born} : R \rightarrow 2^N$ maps transition $r$ to the set $\{j \mid t_j \in \lambda_r\}$ and there exists a path $r^1...r_k$, $k \geq 1$, such that $j \in \text{clock_ref}(r_k)$ and $t_j \notin \lambda_r$, for $1 \leq i < k$. Intuitively, $\text{born}(r)$ identifies a clock that is reset on $r$, whose value can be used on some transition reachable from $r$.

- $\text{active} : R \rightarrow 2^N$ maps transition $r$ to the set $\{j \mid$ there is a path $r^1...r_k$, $k \geq 1$, such that $j \in \text{clock_ref}(r_k)$ and $t_j \notin \lambda_r$, for $1 \leq i < k\}$. Intuitively, $\text{active}(r)$ identifies clocks that are “alive” on $r$ (i.e., their values may be subsequently used). Notice that $\text{born}(r) \subseteq \text{active}(r)$.

- $\text{last_ref} : R \rightarrow 2^N$ maps transition $r$ to $\text{clock_ref}(r) \setminus \text{active}(r)$.

- $\text{needed} : R \rightarrow 2^N$ maps transition $r$ to $\text{active}(r) \cup \text{last_ref}(r)$.

The graph may contain cycles, so $r_k$ need not be different from $r$ in the definitions of $\text{born}$ and $\text{active}$.

Let $r$ be the last transition (on some path) that refers to the value of clock $t_j$. Then $j \in \text{last_ref}(r)$, and in the target timed automaton a (new) clock $c$ that has been assigned to the old clock $t_j$ can be reassigned and reset on $r$.

In the automaton of Fig. 1, $0$ is in both $\text{active}(r_2)$ and $\text{clock_ref}(r_2)$; however, on $r_6$, $0 \notin \text{clock_ref}(r_6)$, but $0 \notin \text{active}(r_6)$, as $0 \in \text{last_ref}(r_6)$.

We will say that a clock $t_j$ is $\text{needed}$ (or is $\text{active}$) on a transition $r$ if $j \in \text{needed}(r)$ (or $j \in \text{active}(r)$).

**Definition 2** A path $p = r_0...r_n$ is a path for clock $t_j$ iff $\text{born}(r_0) = \{j\}$ and $j \in \text{needed}(r_i)$ for $0 \leq i \leq n$.

In the automaton of Fig. 1, $r_1r_2r_4r_6$ is a path for clock $t_0$, as is $r_1r_2r_4,r_1r_2$ or just $r_1$. Similarly, $r_2r_3r_5$ is a path for clock $t_1$. Finally, $r_3r_5r_7$ and $r_4r_6r_7$ are paths for clock $t_2$.

**Definition 3** range : $N \times R \rightarrow 2^R$ maps $(j, r)$ to $\{r' \mid r' \in \text{transitions}(p)\}$, where $p$ is a path for clock $t_j$ that starts at $r$. Intuitively, range $(j, r)$, where $j \in \text{born}(r)$, is the set of transitions in all the paths for clock $t_j$ that begin at $r$. 

Optimal Clock Allocation for a Class of Timed Automata 5
Algorithm 1: Building the ranges for clocks

Input : A timed automaton $\mathcal{A} = \langle E, Q, \{q^0\}, Q_f, V, R, L \rangle$.

Output: An extended timed automaton $\mathcal{A}_e = \langle E, Q, \{q^0\}, Q_f, V, R_e, L \rangle$, where $R_e$ is the set of extended transitions.

$R_e := \emptyset$;

foreach transition $r = (s, q, e, \lambda, \phi) \in R$ in $\mathcal{A}$ do
  $\text{born}(r) := \text{active}(r) := \emptyset$;

repeat
  foreach transition $r = (s, q, e, \lambda, \phi) \in R$ in $\mathcal{A}$ do
    $R_e := R_e \cup \{(r, \text{born}(r), \text{active}(r))\}$;
  
  foreach $r_o \in \text{out}(q)$ do
    $\text{active}(r) := \text{active}(r) \cup ((\text{active}(r_o) \cup \text{clock}_\text{ref}(r_o)) \setminus \text{born}(r_o))$;

  if $L(s) = j$ and $j \in \text{active}(r)$ then
    $\text{born}(r) := \{j\}$;

  $R_e := R_e \cup \{(r, \text{born}(r), \text{active}(r))\}$;

until there were no changes;

We will use the term active range to denote that set of transitions in the range on which the clock in question is active.

Given an automaton $\mathcal{A}$, Algorithm 1 builds an “extended automaton” which is identical to $\mathcal{A}$, except that each transition is of the form $(r, \text{born}(r), \text{active}(r))$ for $r \in R$. Algorithm 1 is a standard flow-analysis algorithm. Its complexity is quadratic in the number of edges.\(^5\)

Given the automaton of Fig. 1 as input, we show two transitions of the extended automaton obtained by Algorithm 1. Transition $r_4$ will be extended to $(r_4, \text{born} = \{2\}, \text{active} = \{0, 2\})$ and $r_6$ to $(r_6, \text{born} = \emptyset, \text{active} = \{2\})$. Notice that both clocks $t_0$ and $t_2$ are needed at $r_4$ and $r_6$.

Given the above definitions we can formulate the following lemma.

Lemma 1 For a timed automaton $\mathcal{A}$ with its set of locations $Q$, we have

$$\forall q \in Q \forall r_i, r_k \in \text{in}(q) \quad \text{active}(r_i) = \text{active}(r_k).$$

Proof. Assume $j \in \text{active}(r_i)$. Then there is a transition $r$ in $\text{out}(q)$ such that $j \in \text{needed}(r)$ But $r$ can be reached from $r_k$, therefore $j \in \text{active}(r_k)$. The rest of the proof follows from symmetry. \(\square\)

In the rest of the paper we consider only timed automata whose transitions have been extended by Algorithm 1, so in the interest of brevity we will omit the adjective “extended”.

\(^5\) For all the algorithms in this paper we assume that set operations are performed in constant time: since the number of transitions, locations and original clocks is known, all sets can be represented by bit vectors.
4.2 Clock Allocation

After liveness analysis has generated the annotations in our automaton, the next step is to use these annotations to allocate new clocks. This is done by an algorithm that replaces the clocks of the original timed automaton with new ones, while minimizing their number. The general idea is that, once a clock is reset on a transition on which it is born, it should never be reset again within the active range of that original reset; however, the clock can be reused outside that active range, provided that certain consistency requirements are satisfied.

Before presenting the algorithm we formulate some additional notation.

Let \( A, B \) and \( C \) be sets and let \( r \subseteq A \times B \times C \). The relation \( r \) can be applied to an argument \( a \in A \) by using the left-associative operator \( "\cdot" \):

\[
 r.a = \{ (b, c) \mid (a, b, c) \in r \}. 
\]

Similarly, for \( b \in B \), \( r.a.b = \{ c \mid (b, c) \in r.a \} \).

If, for every \( (a, b) \in A \times B \), \( r.a.b \) is either a singleton or the empty set, then \( r \) is a function of two arguments: \( r : A \times B \to C \). If \( r.a.b \) is never the empty set, then the function is total, otherwise it is partial.

Next, we formally define clock allocations and their interesting properties.

We assume the existence of a set \( C \), disjoint from \( V \), with \(|R| \) clock variables (this is enough, since at most one clock is reset on any given transition).

**Definition 4** Given a timed automaton \( A \) with the set \( R \) of (extended) transitions and the set \( N \) of clock numbers, a **clock allocation** for \( A \) is a relation \( \text{alloc} \subseteq R \times C \times N \) such that \((r, c, j) \in \text{alloc} \Rightarrow j \in \text{active}(r)\).

Inclusion of \((r, c, j)\) in \( \text{alloc} \) represents the fact that on transition \( r \) clock \( c \) is associated with the old clock \( t_j \) (i.e., \( c \) will eventually replace \( t_j \) on \( r \)).

**Definition 5** A clock allocation \( \text{alloc} \) is **inconsistent** iff there exist two overlapping paths for some clock \( t_j \), \( p \) and \( p' \) (which need not be different), some \( c \in C \) and \( r_k, r_l \in \text{transitions}(p) \cup \text{transitions}(p') \) such that

\[
 j \in \text{active}(r_k) \land j \in \text{active}(r_l) \land (r_k, c, j) \in \text{alloc} \land (r_l, c, j) \notin \text{alloc}. 
\]

\( \text{alloc} \) is consistent iff it is not inconsistent.

**Definition 6** A clock allocation \( \text{alloc} \) is **correct** if:

- \( \text{alloc} \) is a function, i.e., \( \text{alloc} : R \times C \to N \);
- \( \text{alloc} \) is consistent.

Intuitively, the fact that \( \text{alloc} \) is a function means simply that, at any given transition, a clock \( c \) can be associated with at most one (old) clock \( t_j \). The fact that it is consistent means that \( t_j \) must be associated with the same clock \( c \) on an entire path where it is needed. Note that it is possible for a correct allocation to associate two or more (new) clocks with the same (old) clock on the same path. It is also possible for different clocks to be associated with the same (old) clock \( t_j \) on different paths for \( t_j \), as long as the paths are disjoint.

**Definition 7** A clock allocation is **lean** if it is an injective function.
Intuitively, a lean allocation does not associate a clock on a transition with more than one (new) clock.

**Definition 8** The clock allocation alloc is complete iff, for every transition \( r \in R \) and every \( j \in \text{active}(r) \), there is a clock \( c \in C \) such that \((r, c, j) \in \text{alloc}\).

**Observation 1** Let \( A \) be a timed automaton with the set of transitions \( R \), and let alloc be a complete, correct and lean clock allocation for \( A \). Then the following holds: \( \forall r \in R | \text{alloc}.r | = | \text{active}(r) | \).

**Definition 9** We define the number of clocks used in an allocation by:

\[
\text{cost}(\text{alloc}) = |\{ c \in C | \exists r \in R \exists j \in \mathbb{N} (r, c, j) \in \text{alloc} \}|.
\]

**Definition 10** Let \( A \) be a timed automaton and let alloc be a complete and correct clock allocation for \( A \). The allocation alloc is optimal if there is no complete correct allocation alloc’ for \( A \) such that \( \text{cost}(\text{alloc}') < \text{cost}(\text{alloc}) \).

**Theorem 1** Let \( A \) be a timed automaton with the set of locations \( Q \), and let alloc be a complete and correct clock allocation for \( A \). Then the following holds:

\[
\forall s \in Q \forall r_i, r_k \in \text{in}(s) \text{ alloc}.r_i = \text{alloc}.r_k.
\]

Proof. \( \text{alloc}.r_i = \text{alloc}.r_k \) is equivalent to

\[
\forall j \in \mathbb{N} \forall c \in C ((r_i, c, j) \in \text{alloc} \Leftrightarrow (r_k, c, j) \in \text{alloc}).
\]

Let \( j \) and \( c \) be such that \((r_i, c, j) \in \text{alloc}\). This implies that \( j \in \text{active}(r_i) \). But then, from Lemma 1, we have \( j \in \text{active}(r_k) \). Therefore there is some \( r \in \text{out}(s) \) such that both \( r_i r \) and \( r_k r \) belong to paths for clock \( t_j \) (because \( j \) is needed at \( r \)). Since the paths overlap at \( r \), from consistency of \( \text{alloc} \) we must have \((r_k, c, j) \in \text{alloc}\). The rest of the proof follows from symmetry. \(\square\)

In the timed automaton of Fig. 1, assume that \( c_0 \) is associated with clock \( t_0, c_1 \) with clock \( t_1 \) and \( c_2 \) with clock \( t_2 \). Then,

\[
\alpha = \{(r_1, c_0, 0), (r_2, c_0, 0), (r_2, c_1, 1), (r_3, c_1, 1), (r_3, c_2, 2), (r_4, c_0, 0), (r_4, c_2, 2), (r_5, c_2, 2), (r_6, c_2, 2)\}
\]

is a lean, correct and consistent allocation. Notice that \( 1 \in \text{last}_\text{ref}(r_5) \), so \( \text{active}(r_5) \) does not include clock number 1. Similarly, clock \( t_0 \) is not active on transition \( r_6 \). So \( \alpha.r_5 = \alpha.r_6 = \{(c_2, 2)\}, \) in accordance with Theorem 1.

The theorem has two important implications for an algorithm that computes a clock allocation:

- Information about the allocations on all the incoming transitions of a location can be stored in the location itself.
- The exact order in which the transitions of the automaton are traversed need not be important.

These points will become clear as we present our clock allocation algorithm.
4.3 The Clock Allocation Algorithm

We describe our algorithm in two steps: in the first step we present the algorithm for tree-shaped automata, where the root of the tree is the initial location; in the second step we extend the algorithm to allocate clocks for an arbitrary timed automaton in $\mathcal{T}_{\mathcal{DS}}$. We do so not because we think that tree-shaped automata are particularly important, but in the hope of helping the reader’s intuition.

The tree variant is presented as Algorithm 2, which begins with an initial pool of available clocks, $C$ and the set of used clocks, $\mathcal{U}$, which is initially empty. We assume that the clocks in $C$ are numbered: the algorithm always allocates that available clock which has the smallest number.

The algorithm performs a depth first walk of the automaton, beginning at the initial location and annotates each location $s$ with a set of available clocks, as well as with a set of clock assignments. Each assignment is a pair $(c,j)$, where $c$ is the clock that replaces the (old) clock $t_j$. Thanks to Theorem 1 this represents the clock allocation on the incoming transitions of $s$.

More precisely, we define the following functions:

- $\text{pool}: Q \to 2^C$ maps a location $s$ to the set of clocks available at $s$;
- $\text{assignments}: Q \to 2^{C \times N}$ maps $s$ to the set of clock assignments at $s$.

As the algorithm annotates each location $s$ with $\text{assignments}(s)$ and $\text{pool}(s)$, it ensures that every (old) clock $t_j$ and every (new) clock $c$ appear at most once in $\text{assignments}(s)$, and a clock $c$ appears either in $\text{assignments}(s)$ or in $\text{pool}(s)$.

Every time the algorithm visits a transition where a clock, e.g., $t_j$ is born, it associates a clock, say $c$, with $j$. Every time it visits a transition $r$ where $j \in \text{last}_{\mathcal{ref}}(r)$, it restores $c$ to the pool of available clocks. When the algorithm stops, $\mathcal{U}$ contains the clock variables that will be used in the target timed automaton: $|\mathcal{U}|$ can be significantly smaller than $|C|$.

Observe that the resulting allocation is obviously complete. Also, the following theorem holds (see Appendix A for the proof):

**Theorem 2** The allocation found by Algorithm 2 is correct and lean.

We now turn our attention to the general case, i.e., when the timed automaton $\mathcal{A}$ is not necessarily a tree. Algorithm 2 will trace out a spanning tree of $\mathcal{A}$, and the resulting allocation will still be an injective function. It might not, however, be consistent. So the algorithm must be augmented.

An inconsistency can arise in a situation similar to the one illustrated in Fig. 2. Two paths for a clock $t_j$ have different initial transitions: $r_1$ and $r_3$. On each of those initial transitions $j$ is associated with a different (new) clock. When the paths join at location $q$, the values of the allocation on the transitions belonging to in($q$) are different, so Theorem 1 does not hold, even though the allocation is complete and lean. We will call a location such as $q$ a problematic location. Note that $j$ is needed in the outgoing transition of $q$.

---

6 Breadth-first would also work. What is important is that a parent location be visited before its children.
Algorithm 2: Assigning clocks to a tree-shaped timed automaton

**Input**: An extended timed automaton $\mathcal{A}_e = \langle E, Q, \{q^0\}, Q_f, V, R_e, L \rangle$ and the initial pool of available clocks $\mathcal{C}$.

**Output**: An extended timed automaton $\mathcal{A}_e' = \langle E, Q_e, \{q^0\}, Q_f, V, R_e, L \rangle$, where $Q_e = Q \times 2^C \times 2^N$, and the set $U \subseteq \mathcal{C}$ of clocks used.

$U := Q_e := \emptyset$;

$\text{pool}(q^0) := \mathcal{C}$;

$\text{assignments}(q^0) := \emptyset$;

foreach $r = ((s, q, e, \phi), \text{born}(r), \text{active}(r)) \in R_e$, in a depth first order do

if $q$ has not been visited yet then

$\text{tmp} \_\text{pool} := \text{pool}(s)$;

$\text{tmp} \_\text{assignments} := \text{assignments}(s)$;

$\text{source} \_\text{active} := \{ j \mid (c, j) \in \text{assignments}(s), \text{for some } c \}$;

foreach $j \in \text{source} \_\text{active}(r) \setminus \text{active}(r)$ do

// $j \in \text{last} \_\text{ref}(r)$

$\text{tmp} \_\text{assignments} := \text{tmp} \_\text{assignments} \setminus \{(c, j)\}$, where

$(c, j) \in \text{assignments}(s)$;

$\text{tmp} \_\text{pool} := \text{tmp} \_\text{pool} \cup \{c\}$, where $(c, j) \in \text{assignments}(s)$;

if $\text{born}(r) \neq \emptyset$ then

$\text{tmp} \_\text{pool} := \text{tmp} \_\text{pool} \setminus \{d\}$, where $d$ is the clock in $\text{tmp} \_\text{pool}$

which has the smallest number;

$\text{tmp} \_\text{assignments} := \text{tmp} \_\text{assignments} \cup \{(d, j)\}$, where $\text{born}(r) = \{j\}$;

$U := U \cup \{d\}$;

$\text{pool}(q) := \text{tmp} \_\text{pool}$;

$\text{assignments}(q) := \text{tmp} \_\text{assignments}$;

$Q_e := Q_e \cup (q, \text{pool}(q), \text{assignments}(q))$;

end if

end if

end foreach

end foreach

end if

end foreach

end foreach

end foreach

end if

end if

end foreach

The algorithm must ensure that the same clock is assigned to $j$ on all transitions in $\text{out}(s)$ that can lead to the same problematic location $q$.

These considerations can be formalised as follows:

**Definition 11** Let $R_0 = \text{range}(j, r_0)$ and $R_1 = \text{range}(j, r_1)$ be two ranges of a clock $t_j$ on some transitions $r_0$ and $r_1$. $R_0$ is related to $R_1$ iff (1) $R_0$ and $R_1$ intersect, or (2) $R_1$ intersects with some range of clock $t_j$, say $R_2$, and $R_2$ is related to $R_0$.

**Definition 12** Let $S$ be a range of clock $t_j$. We use $\text{Rel}(j, S)$ to denote the union of all ranges of $t_j$ that are related to $S$.

Notice that a range is related to itself, and $\text{Rel}(j, S)$ can be a singleton. Moreover, the set of ranges of clock $t_j$ is partitioned by sets of the form $\text{Rel}(j, p)$: each such range belongs to exactly one such set.

**Definition 13** A family for clock $t_j$ is a set of all those transitions in some $\text{Rel}(j, S)$ on which $t_j$ is active.
Observe that each family for a clock $t_j$ must begin at the same location: location $s$, where $L(s) = j$. It is easy to determine whether two initial transitions for some clock $t_j$ belong to the same family: we must check whether they can lead to the same problematic location.

In the automaton of Fig. 2 we see two families for clock $t_j$. The first is generated by paths for $t_j$ that begin with $r_1$ and $r_2$: those transitions in these paths on which $t_j$ is active form a family, because of the problematic location $q$. The second family is generated by paths that begin with $r_2$ and $r_4$.

Fig. 3 illustrates the transitive nature of a family: $r_1$, $r_2$, and $r_3$ belong to the same family, because of the problematic location $n$. Similarly, $r_2$ and $r_4$ belong to the same family on account of the problematic location $q$. Therefore $r_1$, $r_2$, $r_3$, and $r_4$ all belong to the same family.

**Observation 2** Let $F$ be a family for some clock $t_j$, and let alloc be a complete correct allocation. Then there must exist a clock variable $c \in C$ such that $\text{alloc}.r.c = \{j\}$ for every transition $r \in F$ on which $t_j$ is active. Otherwise alloc would be inconsistent. We say that $c$ is allocated to $F$.

In the automaton of Fig. 2, $t_j$ must be associated with the same (new) clock, say $c$, on paths for clock $t_j$ that begin at transitions $r_1$ and $r_3$. Similarly, $t_j$ must be associated with the same clock (which can, but need not, be $c$) on paths that begin at $r_2$ and $r_4$, as their transitions belong to the same family.

**Observation 3** The number of families to which a transition $r$ belongs is $|\text{active}(r)|$.

**Proof.** Assume the number of families to which transition $r$ belongs is $n$. Therefore, there are exactly $n$ families for different clocks that share transition $r$ (two families for the same clock cannot share $r$, or they would be the same family). Since $t_j$ is active on all the transitions of a family for $t_j$, it follows that $\text{active}(r)$ contains exactly one element for each of the $n$ families. □

**Definition 14** Two families, $F_1$ and $F_2$, belong to the same cluster iff $F_1 \cap F_2 \neq \emptyset$. A cluster $cl$ is a maximal set of such families, i.e., every family outside $cl$ does not overlap with at least one of the families in $cl$.

Observe that the members of a cluster must be families for different clocks (otherwise they would not overlap). Observe also that a family can belong to more than one cluster.

We say transition $r$ belongs to cluster $\mathcal{F}$, if there is a family $F \in \mathcal{F}$, such that $r \in F$.

Notice that if there is no clock $t_j$ such that $j \in \text{active}(r)$, then $r$ does not belong to any family; therefore, it does not belong to any cluster.

Since each pair of families in a cluster shares some common transition, we immediately see the following:

**Observation 4** Every family in a cluster must be allocated a different clock.
In the automaton of Fig. 1, there is only one cluster, and it contains all the three families: \( \{r_1, r_2, r_4\} \) for clock \( t_0 \), \( \{r_2, r_3\} \) for clock \( t_1 \) and \( \{r_3, r_4, r_5, r_6\} \) for clock \( t_2 \). Obviously, a correct allocation for the automaton requires three clocks, even though no more than two clocks are needed on any particular transition.

**Definition 15** The size of a cluster is the cardinality of the set of families that form the cluster.

**Theorem 3** Let alloc be a complete correct allocation for a timed automaton \( \mathcal{A} \). Then \( \text{cost}(\text{alloc}) \) cannot be smaller than the size of the largest cluster in \( \mathcal{A} \).

**Proof.** This is a direct consequence of Observations 2 and 4. \( \square \)

We are now ready to return to a detailed description of our clock allocation algorithm.

To take into account the problematic locations, at each location \( s \) labelled with some \( j \), the outgoing transitions of \( s \) are divided into two sets: (i) the set of “mother” transitions, i.e., \( \{r \in \text{out}(s) \mid j \in \text{born}(r)\} \), and (ii) the set of “other” transition, i.e., \( \{r \in \text{out}(s) \mid \text{born}(r) = \emptyset\} \). Observe that the mother transitions are the initial transitions of all the families for clock \( t_j \). The mother transitions require special attention.

The set of mother transitions is divided into groups. Each group is the set of initial transitions of a family for clock \( t_j \). All transitions belonging to the same group must obtain the same clock assignment for clock number \( j \) (see Observation 2). Notice that if for some clock, say \( t_i \), \( i \in \text{last}_{\text{ref}}(r) \) for every transition \( r \) in the group, then the clock previously assigned to \( i \) becomes available and can be assigned to \( j \). (See procedure find-clock.)

It is a direct consequence of Theorem 1 that the choice of the path taken by the algorithm to reach location \( s \) has no effect on the values of assignments(\( s \)) and pool(\( s \)) (assuming that the algorithm produces a correct allocation).

We are now ready to return to a detailed description of the general algorithm.

Each location has a status that will advance through the following sequence: Unseen, Seen, and Visited. Whenever a location is visited for the first time, all its immediate successors are annotated, and thus become Seen. A location is visited only after it has been annotated (and its status is Seen). To start the process and establish the invariants, the algorithm begins by annotating the initial location.

The algorithm is described below as a collection of procedures. It is started by invoking compute-allocation(\( \mathcal{A}_e, C \)), where \( \mathcal{A}_e \) is the extended automaton obtained by Algorithm 1 and \( C \) is the initial pool of available clocks.

We assume the existence of the following two functions:

- \( \text{reachable} : Q \rightarrow 2^Q \) maps location \( q \) to the set of locations that are reachable from \( q \) by some non-empty path.
- \( \text{reachable_from} : Q \rightarrow 2^Q \) maps location \( q \) to the set of locations from which it can be reached by some non-empty path.
Procedure compute-allocation(timed automaton $A_e$, set of clocks $C$)

Input: An extended timed automaton $A_e = (E, Q, \{q^0\}, Q_f, V, R_e, L)$ and the initial pool of available clocks, $C$.

Output: An extended timed automaton $A'_e = (E, Q_e, \{q^0\}, Q_f, V, R_e, L)$, where $Q_e = Q \times 2^C \times 2^C \times N$.

foreach location $s \in Q_e$ do
  Set the status of $s$ to Unseen;
  annotate($q^0, C, \emptyset$);
  visit($q^0$);

Procedure annotate(location $q$, set of clocks $p$, set of assignments $a$)

// Invoked only when status of $q$ is Unseen.
pool(q) := p;
assignments(q) := a;
Set the status of $q$ to Seen;

Procedure visit(location $q$)

// Invoked only when the status of $q$ is Seen or Visited.
if status of $q$ is not Visited then
  Set the status of $q$ to Visited;
  annotate-immediate-successors-of($q$);
  foreach $r \in out(q)$ do
    visit(target($r$));

Procedure annotate-immediate-successors-of(location $q$)

Partition out($q$) into mothers and others;
foreach $r \in others$ do
  if status of target($r$) is Unseen then
    propagate($q, r, \emptyset$);
  // Otherwise target($r$) is already properly annotated: see Theorem 1.
if mothers $\neq \emptyset$ then
  Groups := partition-into-a-set-of-groups($q$, mothers);
  foreach group $\in$ Groups do
    c := find-clock($q$, group);
    foreach $r \in$ group do
      // The target of $r$ is Unseen (by the dominance assumption).
      propagate($q$, r, $\{c\}$);

After the algorithm has terminated, the value of alloc.$r$, for every transition $r$, is given by assignments(target($r$)). The information is easily used to generate new resets and rename clocks in the constraints.
Procedure find-clock(location $q$, set of transitions $group$)

// Find a clock for $L(q)$ on transitions in $group$.
live_on_entry := \{ j \mid (c,j) \in assignments(q) \};
dying_all := \bigcap_{r \in group}(live_on_entry \setminus active(r));
// The set of clocks whose ranges end in all the transitions in $group$:
released_all := \{ c \mid (c,j) \in assignments(q) \land j \in dying_all \};
available := released_all \cup pool(q);
Return the clock variable with the smallest number in available;

Procedure propagate(location $q$, transition $r$, set of clocks $sc$)

// Invoked only when the target of $r$ is Unseen.
// $q$ is the source of $r$. Propagate pool($q$) and assignments($q$) to target($r$),
// taking into account that some clock ranges may end on $r$. If $sc$ is not empty,
// it must be a singleton: in that case assign its member to clock number $L(q)$.
freed_assignments := \{ (d,j) \mid (d,j) \in assignments(q) \land j \notin active(r) \};
freed_clocks := \{ d \mid (d,j) \in freed_assignments \};
tmp_pool := pool(q) \cup freed_clocks;
tmp_assignments := assignments(q) \setminus freed_assignments;
if $sc \neq \emptyset$ then
    tmp_pool := tmp_pool \setminus sc;
    tmp_assignments := tmp_assignments \cup \{(c,L(q))\}, where c \in sc;
    annotate(target($r$), tmp_pool, tmp_assignments);

Procedure partition-into-a-set-of-groups(location $q$, set of transitions $mothers$)

mother_targets := \{ target(r) \mid r \in mothers \};
// Initially, each mother is in its own group.
Groups := \emptyset;
foreach $r \in mothers$ do
    Groups := Groups \cup \{r\};
PP := \emptyset; // potentially problematic locations
foreach $r \in mothers$ do
    foreach $s \in reachable(target(r))$ do
        if $L(q) \in active(r')$, where $r'$ is an arbitrary transition of in($s$) then
            PP := PP \cup \{s\};
// Those locations in $PP$ that can be reached from more than one mother are
// the problematic locations.
foreach $s \in PP$ do
    targets := reachable_from($s$) \cap mother_targets;
    Merge those members of $Groups$ that contain transitions whose target is in $targets$;
return $Groups$;
The time complexity is quadratic in the size of the graph. The most expensive computation occurs in partition-into-a-set-of-groups, where we may look at up to $|Q|$ nodes for each “mother” transition. But we look at each such transition only once, and there are at most $|R|$ of them.

The resulting allocation is obviously complete. Also, the algorithm satisfies the following theorem:

**Theorem 4** The computed allocation is correct and lean.

*Proof.* All the paths for a clock $t_j$ begin at the same location. The initial transitions of these paths (the “mother” transitions) are partitioned into groups. The members of a group are exactly the initial transitions of a family for $t_j$. The algorithm associates some clock with a group; the association is propagated to all the transitions of the paths for $t_j$ that begin in the group, therefore there is no pair of transitions that satisfies the definition of inconsistency.

When some clock $c$ is assigned to $j$ on the transitions of a group, $t_j$ is the only clock that is born, so $c$ is not, on those transitions, assigned to any $i$ such that $i \neq j$. Moreover, after $c$ is assigned to $j$, it is removed from the pool and returned only on transitions on which $t_j$ is not active. Therefore $c$ cannot be assigned to any other $i$ on any transition $r$ such that $j \in active(r)$. So the allocation is always an injective function, i.e., it is lean. \qed

**Lemma 2** Assume alloc is a complete, correct and lean allocation. Then, for any transition $r$, the number of clocks in alloc.$r$ is not greater than the size of the largest cluster to which $r$ belongs.

*Proof.* Assume cluster $F$ with size $n$ is the largest cluster to which $r$ belongs. Suppose that $|alloc.r| > n$. By Observation 1, $|active(r)| = n' > n$. By Observation 3, the number of families to which $r$ belongs is $n'$. This implies that $r$ belongs to a cluster $F'$ whose size is $n'$, which contradicts the assumption. \qed

**Theorem 5** The computed allocation is optimal.

*Proof.* This is a direct consequence of Lemma 2, Theorem 3, Theorem 4 and the fact that the algorithm always allocates the available clock with the smallest number, i.e., a new clock is added to the set of used clocks only when none of those already in the set will do. \qed

### 5 Related Work

The problem of reducing the number of clocks in timed automata has been addressed by constructing bisimilar timed automata [4, 6]. (Both these papers make use of standard liveness analysis; we claim no originality on this point.)

The approach used by Daws and Yovine [4] combines two methods for reducing the number of clocks. The first method is based on identifying the set of active clocks in each location of an automaton and applying a clock renaming to
this set of active clocks locally, at each location, to obtain a bisimilar timed automaton. The second method is based on the notion of equality between clocks: clocks that are equal in a location are identified, and only one of them is included in the target timed automaton. The authors use assignments of the form \( x := y \), where \( x \) and \( y \) are both clocks, which is an extension of the traditional formalism of timed automata. The method of Daws and Yovine will not always yield the minimum possible number of clocks, as argued by Guha et al. [6].

Guha et al. [6] propose another method for constructing bisimilar timed automata with a minimum number of clocks. Their method considers zone graphs, and uses them for identifying redundant transitions and implied constraints that can be eliminated. It also uses a technique of “splitting locations” for reducing the number of clocks. The number of locations of the constructed timed automaton may become exponential in the number of clocks of the original timed automaton. The computational complexity of their method is 2-EXPTIME.

6 Conclusions

We propose a new approach to optimal allocation of clocks for an interesting class of timed automata, \( TA_{DS} \) (see Definition 1 on p. 3).

In contrast to the work of Guha et al. [6], our method of reducing the number of clocks does not introduce new locations. It is not as generally applicable, but it is much simpler, and its cost is only quadratic in the size of the graph.

Its efficacy is based on our decision to assume, during analysis, that all locations are always reachable. Since this might sometimes not be the case, our clock allocation might not always be strictly optimal according to more general criteria, but we believe this caveat has little practical importance.

References

A Proof of Theorem 2

Proof. On any transition where a clock, say $t_j$, of the original timed automaton is born, one (new) clock, e.g., $c$, is assigned to $j$. Moreover, $c$ is not, on that transition, assigned to any other $i$ (corresponding to clock $t_i$). A clock is assigned to a clock number $i$ only when $t_i$ is born. So the allocation is always an injective function, i.e., it is lean.

To see that the allocation is consistent, take any path $p$ for some clock $t_j$. A clock $c$ is assigned to $j$ only on the first transition of $p$. At that point $c$ is removed from the pool and is not returned there before $p$ ends. It is therefore impossible for $c$ to be assigned to any $i$ on any transition $r$ of $p$ such that $j \in \text{active}(r)$.

Observe that if there is another path for $t_j$, say $p'$, that shares a transition with $p$, then $p$ and $p'$ will have a common initial transition, because the graph of the automaton is a tree. In that case $c$ will be assigned to $j$ on that transition, and will not be assigned to another clock number on any transition $r$ of $p$ or $p'$ such that $j \in \text{active}(r)$.