Optimal Clock Allocation for Synthesized Acyclic Timed Automata

Neda Saeedloei

Technical Report 16-02

Department of Computer Science
University of Minnesota Duluth
320 Heller Hall
1114 Kirby Drive
Duluth, MN 55812

Phone: (218) 726-7607

Email: cs@d.umn.edu

Web: www.d.umn.edu/cs/

June 27, 2016
Optimal Clock Allocation for Synthesized Acyclic Timed Automata

Neda Saeedloei
Department of Computer Science
University of Minnesota Duluth
Minnesota, USA
nsaeedlo@d.umn.edu

Abstract. Having recently proposed a new method for synthesizing timed automata from a set of scenarios, in this paper we propose a novel clock allocation algorithm that minimizes the number of clocks in the constructed timed automata. Scenarios, formally defined as Timed Event Sequences, together with mode graphs are used to describe partial behaviors of real-time systems. Given a set of scenarios in the form of Timed Event Sequences and a mode graph, our synthesis method constructs a minimal, deterministic, acyclic timed automaton, with a minimal number of clocks.

1 Introduction

Model-based design has been used as an effective approach for design, analysis, and verification of complex systems, including real-time systems. The process of designing and developing a complex system, especially a real-time system, can greatly benefit from building a formal model, i.e., one that is expressed in a formal language with well-defined semantics.

The starting point in building a model of a system is the requirements. While the use of model-based design is promising and rapidly growing, a major problem is the lack of good formal requirements models, as complete requirements of systems often do not exist or, if they do exist, are ambiguous or very low level. Formal models of real-time systems are especially hard to obtain given their intrinsic complexity introduced by time.

To address this problem, techniques have been proposed for automatic synthesis of formal models from scenarios [15, 4]. Recently, there has been also work on building formal models of human-centric decision systems from scenarios [10].

We recently proposed a new synthesis method for constructing formal models of real-time systems from scenarios [13]. A scenario can be viewed as a description of a partial behavior of a real-time system during a time interval: it describes not only the events that occur in the system, but also the timing relations among the events. A set of scenarios can capture the required time-related constraints on system behaviors.

We introduced Timed Event Sequences (TES), a formalism that allows practitioners to specify the required behavior of real-time systems in terms of scenarios
formally, precisely, and at a high level of abstraction. We also used mode graphs to specify the events that are legal in the various modes of a system. Given a set of scenarios described as TES and a mode graph, our synthesis method constructs a timed automaton [2] which is deterministic, acyclic, has a minimum number of states, and captures all those behaviors that are described by the scenarios [13].

In the current paper we focus on the problem of optimally allocating clocks to such a synthesized automaton. Minimizing the number of clocks is important, as it affects the complexity of the verification problem [1, 2]. It is well known that, given a timed automaton, the problem of finding whether there exists another timed automaton with fewer clocks accepting the same timed language is undecidable [6]. However, we consider only acyclic timed automata that arise naturally in the context of formal model synthesis [13] and propose a method for optimally allocating clocks to them.

Our clock allocation method starts with a deterministic, acyclic graph of the target automaton, where transitions have time annotations derived from scenarios: we call this a “time-annotated graph”. Intuitively, the time annotations are constraints among time variables, where a time variable stores the time at which an event in a scenario has occurred. We transform these time annotations to clock operations, i.e., clock resets and clock constraints, while keeping the number of clocks minimal. The method is based on liveness analysis of time variables: this allows assigning the same clocks to time variables with disjoint liveness ranges. Our clock allocation problem resembles the general register allocation problem in compilers [12, 3], but is simpler and allows more effective solutions.

2 Constructing Time-annotated Graphs from Scenarios and Mode Graphs

In this section, we present a brief overview of mode graphs and Timed Event Sequences, their use in formally representing scenarios, and our method for constructing states and transitions of the overall timed automaton from scenarios. More details can be found in our companion paper [13].

We assume the existence of a set $\Sigma$ of events (actions taken by both the system and its environment) and a set $V = \{t_1, t_2, ..., t_n\}$ of time variables.

**Mode Graphs.** A mode graph is a state machine whose states are called modes and whose transitions are triggered by various events [9]. Formally, a mode graph is a tuple $M = (M, m_0, \Sigma, T)$, where $M$ is a finite set of modes, $m_0$ is the initial mode, $\Sigma$ is a set of events, and $T : M \times \Sigma \rightarrow M$ is a transition relation (which is a function). A mode graph can also have a final mode $m_f \in M$; however, for most practical systems, the initial and final modes are the same. We define $source_{\text{mode}}(e) = \{m \mid (m, e, n) \in T, \text{for any mode } n \in M\}$. An example of a mode graph, corresponding to an ATM machine [13], is shown in the left hand side of Fig. 1. For simplicity of presentation, each mode is represented as $m_i$, $0 \leq i \leq 7$, in this mode graph. For instance, card-not-inserted is represented by $m_0$. 


Given two modes $m$ and $n$ in mode graph $M$, mode $m$ is a dominating mode of $n$, if all paths from the initial mode to $n$ pass through $m$ [11]. We define the dominance relation between modes in a mode graph by $DOM$: $m$ $DOM$ $n$ iff $m$ is a dominating mode of $n$. Note that $DOM$ is reflexive. We extend the definition of dominating modes to dominated events: an event $e$ of a mode graph is dominated by mode $m$, if $n \in$ source mode $(e)$ and $m$ $DOM$ $n$. For example, in the mode graph of Fig. 1, card-not-inserted, card-inserted, pin-entered, user-verified and waiting-for-bank are the dominating modes of waiting-for-bank. The event display-menu is dominated by all the modes that dominate waiting-for-bank.

To each mode $m_i$ in $M$ we assign a time variable $t_i \in V$, where $t_i$ is interpreted as the time of leaving $m_i$. If there is a cycle involving mode $m_i$ in the mode graph, then $t_i$ corresponds to the time of the first event occurrence at $m_i$. For example, in the mode graph of Fig. 1, $t_1$ associated with $m_1$ corresponds to the time of the first attempt to enter the PIN.

For a set $V$ of time variables, we define the set $\Phi(V)$ of time annotations of the form $W - t_j \sim a$, where $W$ is the time currently shown by the global wall clock, $t_j$ is a time variable in $V$, $\sim \in \{\leq, \geq, <, >, =\}$, and $a$ is a constant in the set of rational numbers, $\mathbb{Q}$.

**Timed Event Sequences.** A Timed Event Sequence $\xi$ is a tuple $(m^{initial}, \Psi, m^{final})$, where $m^{initial}$ and $m^{final}$ are in $M$, and $\Psi = [\psi_1, \psi_2, ..., \psi_n]$ is a non-empty sequence of timed events of the form $(e_i, \phi_i)$, where $e_i$ is an event in $\Sigma$, and $\phi_i \in 2^{\Phi(V)}$ is a (possibly empty) set of time annotations associated with $e_i$. Moreover, $\Psi$ is a set of consecutive transitions of the mode graph, such that for any two timed events $\psi_i = (e_i, \phi_i)$ and $\psi_{i+1} = (e_{i+1}, \phi_{i+1})$, $1 \leq i < n$, $(m, e_i, m') \in T$ and $(m', e_{i+1}, m'') \in T$, for some $m$, $m'$ and $m''$ in $M$. For
a time annotation \( W - t_j \sim a \in \phi_i \), \( W \) is the time shown by the wall clock when the event occurs, and \( t_j \) corresponds to the time of an event \( e_j \), occurring before \( e_i \) in \( \xi \), that originates at mode \( m_j \), where \( m_j \in \text{DOM} m_i \). This means that, in our scenarios, a time annotation accompanying an event \( e \) can refer only to the times of previous events which are originated in the dominating modes of \( e \). This fundamental assumption, which we call the dominance assumption, guarantees that all time variables are well-defined (i.e., each time variable used in a time annotation on a transition is being defined on every path that reaches the transition).

In the definition of Timed Event Sequences, \( m^{\text{initial}} \) and \( m^{\text{final}} \) are interpreted as the first and last mode of \( \xi \), respectively. For a scenario \( \xi \), specified as a Timed Event Sequence \((m^{\text{initial}}, \Psi, m^{\text{final}})\), we define initial_mode\((\xi) = m^{\text{initial}} \) and final_mode\((\xi) = m^{\text{final}} \). We also define modes\((\xi) \) to be the set of modes that can be reached by starting from initial_mode\((\xi) \) in \( \mathcal{M} \) and performing some non-empty contiguous sequence of events in \( \xi \).

The two TES shown in the right hand side of Fig. 1 specify the ATM’s initial behavior: \( S_1 \) describes the situation in which the user enters an incorrect PIN in the first attempt and the correct PIN in the second attempt. In \( S_2 \) the user enters the correct PIN in the first attempt.

Given a mode graph \( \mathcal{M} = (\mathcal{M}, m_0, \Sigma, T) \), a set of scenarios \( \Xi \) is complete if:

1. \( \forall \xi \in \Xi \), such that initial_mode\((\xi) = m_0 \).
2. \( \forall \xi \in \Xi \), either initial_mode\((\xi) = m_0 \), or \( \exists \xi_1, \xi_2, \ldots, \xi_j \in \Xi \), such that:
   - initial_mode\((\xi_1) = m_0 \),
   - for each \( \xi_k, 1 < k \leq j \), initial_mode\((\xi_k) \in \text{modes}(\xi_{k-1}) \),
   - initial_mode\((\xi_j) \in \text{modes}(\xi_{j}) \).
3. \( \exists \xi \in \Xi \), such that final_mode\((\xi) = m_f \).
4. \( \forall \xi \in \Xi \), either final_mode\((\xi) = m_f \), or \( \exists \xi_1, \xi_2, \ldots, \xi_j \in \Xi \), such that:
   - final_mode\((\xi_1) = m_f \),
   - for each \( \xi_k, 1 < k \leq j \), final_mode\((\xi_{k-1}) \in \text{modes}(\xi_k) \),
   - final_mode\((\xi_j) \in \text{modes}(\xi_j) \).
5. All scenarios are time consistent: If two scenarios contain the same transition (between the same two modes), then the time annotations corresponding to the transition must be identical in both scenarios.

Intuitively, criteria 2 and 4 ensure that the constructed timed automaton will have a unique initial state, and, moreover, there are no “dead ends”: states from which no transition can be taken. Criteria 1, 2 and 5 are required for constructing deterministic timed automata and criterion 3 guarantees existence of a final state. In the rest of the paper, we assume that the set of scenarios is complete.

Given a mode graph \( \mathcal{M} = (\mathcal{M}, m_0, \Sigma, T) \) and a complete set of scenarios, our synthesis [13] is performed in two phases: (i) constructing the states and transitions of the overall timed automaton, (ii) allocating clocks and adding clock operations, i.e., clock resets and clock constraints, to transitions. The first phase uses scenarios to construct a time-annotated graph \( G = (E, Q, q^0, q^f, R, L) \), where \( E \subseteq \Sigma \), \( Q \) is the set of states, \( q^0 \) and \( q^f \) are the initial and final states, \( R \) is the set of transitions, and \( L : Q \rightarrow M \) is a labeling function. This is an acyclic,
**deterministic** graph which contains the states and transitions of the overall timed automaton, where states are labelled with modes, and transitions are of the form $(s, q, e, \phi)$, where $s, q \in Q$, $e \in E$, and $\phi \in 2^{\Phi(V)}$. Each state $q$ of $G$ is labelled with mode $m_j$, i.e., $L(q) = m_j$, if there is a transition on event $e$ from state $s$ to state $q$ such that $L(s) = m_i$, and $(m_i, e, m_j) \in T$.

The time-annotated graph shown at the bottom of Fig. 1 is obtained from scenarios S1 and S2 in Fig. 1, and is generated by our construction algorithm described in our companion paper [13].

Note that if the mode graph does not contain cycles, then for each mode $m$ in $M$ there is only one state in $G$, such that $L(s) = m$; however, if the mode graph is cyclic, more than one state of $G$ might be labelled with the same mode. After the construction of the time-annotated graph is completed, a breadth-first walk will be performed on the graph to remove the duplicate modes associated with states: if two states $s$ and $q$ are associated with mode $m$, and $s$ is visited before $q$, the mode associated with $q$ will be removed ($L(q) = \text{null}$). This step makes it easier to ensure that when time variables will be replaced with clocks by our clock allocation algorithm, they will really refer to the first time of exiting the mode. Our method of construction ensures that the dominance assumption will hold after this step, hence all the time variables will be well-defined.

Having generated the states and transitions of the overall timed automaton, in the form of a time-annotated graph, we move to the second phase of our synthesis method: transforming the time-annotated graph to a timed automaton. Our approach is based on liveness analysis of time variables and aims at minimizing the number of clocks in the overall constructed timed automaton.

### 3 Constructing Timed Automata from Time-Annotated Graphs

For transforming a time-annotated graph $G$ to a final timed automaton, the time annotations of $G$ must be replaced by clock operations, i.e., clock resets and clock constraints. The transformation consists of: (i) determining the minimum number of clocks required, (ii) identifying transitions with which each clock must get reset and adding clock resets accordingly, (iii) rewriting the time annotations of $G$ as clock constraints. These tasks are performed in three steps:

- identifying the liveness ranges of time variables;
- assigning clocks to time variables (or to the ranges of time variables);
- generating clock constraints, and clock resets, using the ranges.

Our clock allocation problem is reminiscent of the general register allocation problem [12], but it is simpler and allows a more effective solution. It is simpler, because: (1) the graph is acyclic; (2) our graphs satisfy the dominance assumption; (3) the number of clocks is potentially unlimited; (4) the formalism does not allow the value of a clock to be copied to another clock.

---

1 We have a working prototype implementation of the first two steps. It can be made available upon request.
3.1 Liveness Analysis of Time Variables

The liveness analysis of time variables is performed by Algorithm 1. The algorithm determines the “liveness ranges” of all the time variables in the time-annotated graph. Informally, a liveness range for time variable $t_j$ captures the notion “$t_j$ is needed here or further on”. Given a particular path through the graph, we can easily identify the last use of $t_j$ (if any), and the point at which $t_j$ must be created. Identifying a range on the graph will take the form of annotating transitions with the number of the range. We do so in such a manner that on each possible path from the initial state to the final state any given range will be a continuous subsequence of transitions, and will indeed stretch from the creation of $t_j$ to its last use on that path.

Before describing the algorithm, we introduce a few auxiliary terms. Given a time-annotated graph $G = (E,Q,q^0,q^f,R,L)$, by $\text{out}(s)$, where $s \in Q$, we denote the set of transitions that originate in state $s$, i.e., for which $s$ is the source. For a time variable $t_j$ appearing in some time annotation $W - t_j \sim a$, the liveness ranges of $t_j$ will be referred to as range $j$.

Variable $t_j$ refers to the time of leaving a state $s$ such that $L(s) = m_j$. Note that there is only one such state. Range $j$ begins only on transitions in $\text{out}(s)$: not on all such transitions, but only on those that may begin a path on which $t_j$ is used.

Range $j$ may end in two situations: (i) at the last transition $r' \in R$ on a path originating at $s$, such that $r'$ is annotated with $W - t_j \sim a$, (ii) on a transition that “forks away” from the range.

The second situation is best understood on the example of Fig. 2 (a). In this graph, the range $i$ begins in a transition originating at state $s$, from which state $q$ is reachable. The state $q$ has two outgoing transitions. Assume that $t_i$ is never referred to in the subpath that begins in $r_1$. Then, the range of $i$ must end at transition $r_1$.

For a transition $r = (s,q,e,\phi) \in R$, we define $\text{time}_\text{ref}(r)$ to be the set $\{t_j \mid W - t_j \sim a \in \phi\}$. Intuitively, $\text{time}_\text{ref}(r)$ is the set of time variables which are referred to in the time annotations of transition $r$. We define $N$ to be the set $\{j \mid W - t_j \sim a \in \phi, \text{where } (s,q,e,\phi) \in R\}$. For a path $p = r_1...r_k$, we define $\text{transitions}(p) = \{r_1,...,r_k\}$. By a completed path we mean a path whose last transition ends at the final state, $q^f$. We also define the following functions:

- $\text{born} : R \rightarrow 2^N$ maps transition $r$ with source $s$ to the singleton set $\{j \mid L(s) = m_j\}$ if there exists a path $rr_1...r_k$, $k \geq 1$, such that $j \in \text{time}_\text{ref}(r_k)$; otherwise, $\text{born}(r) = \{}$.
- $\text{end} : N \rightarrow 2^R$: transition $r$ belongs to $\text{end}(j)$ if there is a path $r_0...r_k$, where $\text{born}(r_0) = \{j\}$, $k \geq 0$, and either of the following cases occurs:
  1. $j \in \text{time}_\text{ref}(r)$, but there is no completed path $rr'_0...r'_i$ such that $j \in \text{time}_\text{ref}(r'_i)$, for some $0 \leq i \leq l$.
  2. There is a path $r_0...r_kr''_0...r''_i$ such that $j \in \text{time}_\text{ref}(r''_i)$ for some $0 \leq i \leq l$, but $j \notin \text{time}_\text{ref}(r)$. (This is the situation illustrated in Fig. 2 (a)).
Fig. 2

- \textit{dying}: \mathbb{R} \rightarrow 2^\mathbb{N} \text{ maps transition } r \text{ to the set } \{ j \mid r \in \text{end}(j) \}.

- \textit{live \_range}: \mathbb{N} \rightarrow 2^\mathbb{R} \text{ maps range } j \text{ to the set } \{ r_i \mid r_i \in \text{transitions}(p), 0 \leq i < k, \text{ for any path } p = r_0 \ldots r_k, \text{ such that } \text{born}(r_0) = \{ j \} \text{ and } r_k \in \text{end}(j) \}.

- \textit{active}: \mathbb{R} \rightarrow 2^\mathbb{N} \text{ maps transition } r \text{ to the set } \{ j \mid r \in \text{live \_range}(j) \}. \text{ Notice that } \text{born}(r) \subseteq \text{active}(r), \text{ but } \text{dying}(r) \text{ is never included in } \text{active}(r).

- \textit{reachable}: \mathbb{R} \rightarrow 2^\mathbb{Q} \text{ maps transition } r \text{ to the set } \{ s \mid \text{there exists a (partial) path beginning with } r \text{ that ends at } s \}.

Algorithm 1 visits the transitions in the reverse of a topological ordering of transitions in \( G \). This ensures that a transition is visited after all the outgoing transitions of its target state. The algorithm marks each transition \( r \) of the graph with the following information:

- An identifier \( j \) that identifies a range that begins in \( r \) (\text{born}(r)). For \( r = (s,q,e,\phi) \in \mathbb{R}, \text{ with } L(s) = m_j, \text{born}(r) \) will be empty, if there is no transition reachable from \( r \), whose time annotation refers to \( t_j \).

- A set of numbers that identify ranges that end at \( r \) (\text{dying}(r)).

- A set of numbers that identify ranges to which \( r \) belongs (\text{active}(r)).

- A set of states reachable from \( r \) (\text{reachable}(r)).

Given a time-annotated graph \( G \), Algorithm 1 builds an “extended graph” which is identical to \( G \), except that each transition is of the form \( (r,\text{born}(r),\text{active}(r),\text{dying}(r),\text{reachable}(r)) \) for \( r \in \mathbb{R} \).

Given the graph of Fig. 2 (b) as an input, we show two transitions of the extended graph obtained by Algorithm 1. The transitions \( r_1 = (s_2, s_4, d, \emptyset) \) and \( r_2 = (s_4, s_5, f, \{ W - t_0 \geq 2 \}) \) will be extended to \( (r_1, \text{born} = \{ 2 \}, \text{dying} = \{ 1 \}, \text{active} = \{ 0, 2 \}, \text{reachable} = \{ s_4, s_5, s_6 \}) \) and \( (r_2, \text{born} = \emptyset, \text{dying} = \{ 0 \}, \text{active} = \{ 2 \}, \text{reachable} = \{ s_5, s_6 \}) \), respectively.

In the rest of the paper, we consider time-annotated graphs whose transitions are extended with such markings. So, we drop the word “extended”.

![Fig. 2](image-url)
Algorithm 1: Building the liveness ranges for time variables

Input: A time-annotated graph $G = (E, Q, q_0, q_f, R, L)$.
Output: An extended graph $G_e = (E, Q, q_0, q_f, R_e, L)$, where $R_e$ is the set of extended transitions.

1. $R_e := \emptyset$
2. For each transition $r = (s, q, e, \phi) \in R$, in the reverse of a topological ordering of transitions in $G$ do
   3. $\text{born}(r) := \emptyset$
   4. $\text{active}(r) := \emptyset$
   5. If $q = q_f$ then
      6. $\text{dying}(r) := \text{time}_\text{ref}(r)$
      7. $\text{reachable}(r) := \{q_f\}$
   Else
      8. $\text{dying}(r) := \emptyset$
      9. $\text{reachable}(r) := \emptyset$
   10. For each $r_o \in \text{out}(q)$ do
      11. $\text{active}(r) := \text{active}(r) \cup (\text{active}(r_o) \cup \text{dying}(r_o) \setminus \text{born}(r_o))$
      12. $\text{reachable}(r) := \text{reachable}(r) \cup \text{reachable}(r_o) \cup \{q\}$
      13. $\text{dying}(r) := \text{time}_\text{ref}(r) \setminus \text{active}(r)$
      14. If $L(s) = m_j$ and $j \in \text{active}(r)$ then
         15. $\text{born}(r) := \{j\}$
         16. For each $r_o \in \text{out}(q)$ do
            17. $\text{dying}(r) := \text{dying}(r_o) \cup (\text{active}(r) \setminus \text{dying}(r) \setminus \text{active}(r_o))$
            18. $\text{forking away}^\dagger$ from the range, see Fig. 2 (a)
            19. $R_e := R_e \cup \{(r, \text{born}(r), \text{active}(r), \text{dying}(r), \text{reachable}(r))\}$

3.2 Clock Assignments

After performing the liveness analysis and generating the “range markings” in our graph, the next step is to use these markings to allocate clocks. This step is performed by Algorithm 2 which assigns a clock to each liveness range in the time-annotated graph. The idea is that clocks cannot be reset within the ranges of their corresponding time variables. However, clocks that are assigned to time variables with disjoint ranges can be reassigned (and hence, be reset) and reused again.

The algorithm begins with an initial pool of available clocks, $P_0$, with $|R|$ number of clocks, where $R$ is the set of transitions in the graph (since at most one live range can be born in a transition, $P_0$ indeed has a sufficient number of clocks). Starting with an empty set of used clocks, $C$, Algorithm 2 performs a depth-first walk of the graph, beginning in the initial state. Every time it visits a transition where a range, e.g., $j$ begins, it initializes a clock, say $c$, and assigns it to time variable $t_j$. If $c$ has not been assigned to any live range before, it adds $c$ to $C$. While traversing the graph, every time the algorithm visits a transition where range $j$ ends, it returns $c$ to the pool of available clocks (but, a copy of $c$
remains in $C$). We assume that the clocks in $P_0$ are numbered and Algorithm 2 always allocates a clock with the smallest number. When the algorithm stops, set $C$ contains the clocks of the overall timed automaton. Notice that the number of clocks actually used (assigned) by Algorithm 2 can be significantly lower than the number of clocks in $P_0$.

We describe our clock allocation algorithm in two steps: in the first step, we present the algorithm for tree-shaped graphs, where the root of the tree is the initial state of the time-annotated graph. In the second step we extend the algorithm to allocate clocks to an arbitrary time-annotated graph.

First, we extend each state of the time-annotated graph with:

- the set of available clocks (i.e., clocks in the pool that not currently in use);
- the set of clock assignments of the form $(j, c)$, where $j$ is a range and $c$ is the clock assigned to $j$.

We define the following functions:

- $pool: Q \rightarrow 2^{P_0}$ maps a state $s$ to the set of clocks available at $s$;
- $clocks: Q \rightarrow 2^{\mathbb{N} \times P_0}$ maps a state $s$ to the set of clock assignments at $s$.

For the initial state, $s_0$, $clocks(s_0) = \emptyset$ and $pool(s_0) = P_0$. The algorithm ensures that every range $j$ and every clock $c$ appear at most once in $clocks(s)$, and a clock $c$ appears either in $clocks(s)$ or in $pool(s)$. These are indeed well-defined functions, i.e., they do not depend on the path through which we reached the state, as will be shown in Theorem 1 and Corollary 1. As Algorithm 2 visits the transitions of the tree it extends each state $s$ to $(s, pool(s), clocks(s))$.

This simple form of Algorithm 2 is not sufficient for the general case, because in general the graph contains “joins”, i.e., states that have more than one incoming transition. Whenever there is more than one incoming path to a state $q$, we must make sure that in every transition after $q$ a reference to a particular time variable $t_j$ is associated with the same clock, regardless of the detailed order in which our algorithm visited the various paths. We will now extend Algorithm 2 to take this into account.

Recall that we assign a range $j$ to a clock only when the range is born, i.e., only on the outgoing transitions of the state labelled $s$ with mode $m_j$. A problematic situation arises only if two or more transitions from $s$ can eventually lead to the same state (“join state”), say $q$, such that range $j$ is active in a transition in $out(q)$ (cf. Fig. 2 (c)). We call such a join state a problematic state. (Problematic states can be easily determined by looking at the intersection of states reachable from the outgoing transitions of $s$.) The algorithm must ensure the consistency of clock assignments to $j$ on all transitions in $out(s)$ that can lead to the same problematic state $q$.

We must therefore extend the actions performed in a state (during our walk of the graph), as follows: at each state $s$, labelled with, say, $m_j$, the outgoing transitions of $s$, $out(s)$, will be divided into two sets: (i) the set of “mother” transitions, i.e., the set of transitions $r$, where $born(r) \neq \emptyset$, and (ii) the set of “other” transitions, i.e., the set of transitions $r'$, where $born(r') = \emptyset$. The mother
Algorithm 2: Assigning clocks to a time-annotated tree-shaped graph

Input : An extended time-annotated tree $G_e = \langle E, Q, q^0, q^f, R_e, L \rangle$ and the initial pool of available clocks $P_0$.
Output: An extended time-annotated tree $G_e = \langle E, Q_e, q^0, q^f, R_e, L \rangle$, where $Q_e = Q \times 2^{P_0} \times 2^{N \times P_0}$ and the set $C \subseteq P_0$ of clocks.

1. $C := \emptyset$;
2. $Q_e := \emptyset$;
3. $pool(q^0) := P_0$;
4. $clocks(q^0) := \emptyset$;
5. foreach $r = ((s, q, e, s), born(r), active(r), dying(r), reachable(r)) \in R$ do
6. if $q$ is not extended yet then
7. $temp\_pool := pool(s)$;
8. $temp\_clocks := clocks(s)$;
9. foreach $j \in dying(r)$ do
10. $temp\_clocks := temp\_clocks \setminus \{j, c\}$, where $(j, c) \in clocks(s)$;
11. $temp\_pool := temp\_pool \cup \{c\}$;
12. if $born(r) \neq \emptyset$ then
13. $temp\_pool = temp\_pool \setminus \{d\}$, where $d$ is a clock in $temp\_pool$
14. which has the smallest number;
15. $temp\_clocks := temp\_clocks \cup \{j, d\}$, where $born(r) = \{j\}$;
16. $C := C \cup \{d\}$;
17. $pool(q) := temp\_pool$;
18. $clocks(q) := temp\_clocks$;
19. $Q_e := Q_e \cup (q, pool(q), clocks(q))$;

Transitions require special attention, while the other transitions are processed as before. The set of mother transitions will be divided into “families” of transitions, where two transitions $r_1$ and $r_2$ belong to the same family if paths originating at $r_1$ and $r_2$ can converge on the same problematic state. All transitions belonging to the same family must obtain the same clock assignment for range $j$. Notice that if some range, say $i$, dies/ends at all the transitions in the family, the clock corresponding to $i$ becomes available and can be assigned to $j$. The propagation of information (the pool of available clocks and the set of assigned clocks) to the target states of these transitions is done as before. It should be clear that such treatment suffices to handle the problematic situation depicted by Fig. 2 (c).

Figs. 3 (a) and (b) show two graphs with families of transitions leading to problematic states. In Fig. 3 (a), transitions $r_1$ and $r_3$, leading to the problematic state $q$, belong to the same family. Similarly, transitions $r_2$ and $r_4$, leading to the problematic state $n$, belong to a second family. The time variable $t_j$ must be assigned to the same clock, say $c$, on both $r_1$ and $r_3$. Likewise, $t_j$ must be assigned to the same clock (which can, but need not, be $c$) on both $r_2$ and $r_4$. Fig. 3 (b) illustrates the transitive nature of a family: $r_2$ and $r_4$ belong to the same family, because of the problematic state $q$. Similarly, $r_4$ and $r_5$ belong to
the same family on account of the problematic state $n$. Therefore $r_2$, $r_4$ and $r_5$ all belong to the same family, and must all obtain the same clock.

**Theorem 1** If there is more than one path to a state $s$, the set of clock assignments in $s$ will always be the same, regardless of the path taken by the algorithm to reach the state.

**Proof.** Assume $s$ with $L(s) = j$ is a state from which at least two paths, $p_1$ and $p_2$, converge to some state $p$. Assume $r_1$ and $r_2$ are the initial transitions of $p_1$ and $p_2$, respectively, and that $\text{clocks}(s) = \{(i_1,c_1), \ldots, (i_n,c_n)\}$. We consider three cases:

1. If a range $i_k$, $1 \leq k \leq n$, dies with transition $r$ along the path $p_1$, and is therefore removed from the set of clock assignments propagated from $s$, then $i_k$ is not active at any outgoing transition of $p$. (Since $p$ is reachable from $r$, if $i_k$ were active at a transition in $\text{out}(p)$, it would be also active at $r$). Now, either (i) $i_k$ is not active at $r_2$, or (ii) $i_k$ is active at $r_2$. In the first case, the clock assignment propagated from $s$ to $p$ along $p_2$ will not include an assignment for $i_k$. In the second case, the range for $i_k$ must die at some transition along path $p_2$, before reaching $p$, otherwise, $i_k$ would be active at a transition in $\text{out}(p)$. Therefore, $\text{clocks}(p)$ will not include $i_k$, regardless of which path is taken to reach $p$.

2. If a range $i_{k+1}$, $1 \leq k \leq n$, is born with a transition $r$ on path $p_1$, such that $r \neq r_1$, because of our dominance assumption, $i_{k+1}$ must die before reaching $p$, because the existence of $r_2$ implies that $p$ is not dominated by the state labelled with $m_{k+1}$. Similarly, any range that is born anywhere on path $p_2$ except on $r_2$ must die before reaching $p$. Therefore, ranges that are born along non-initial transitions of $p_1$ and $p_2$ will not be present in $\text{clocks}(p)$.

3. For a range $j$ that is born at $r_1$ and/or $r_2$, we consider two cases: (i) if $j$ is not active at an outgoing transition of $p$, then by an argument similar to that for case 1, $j$ will not be included in $\text{clocks}(p)$, regardless of the path which is taken to reach to $p$, (ii) if $j$ is active at a transition in $\text{out}(p)$, then
Corollary 1 If there is more than one path to a state \( s \), the pool of available clocks in \( s \) will always be the same, no matter which path is taken by the algorithm to reach \( s \).

Proof. This is a direct consequence of the theorem. The set of clocks is fixed, and each clock is either assigned to a range, or a member of the pool.

In the graph of Fig. 3 (c), assume that range \( j \) is mapped to clock \( c_1 \), on path \( p_1 \) and range \( k \) is mapped to clock \( c_2 \) on path \( p_2 \). At transition \( r_3 \), a new range \( i \) is born (while \( j \) and \( k \) are still active) and therefore, a new clock, i.e., different from \( c_1 \) and \( c_2 \), must be assigned to \( i \). So, the total number of allocated clocks is equal to the maximum number of active ranges on \( r_3 \) (indeed, this is an optimal allocation). Fig. 2 (b) shows that this is not always the case. Assume that Algorithm 2 assigns \( c_0 \) to \( t_0 \) and \( c_1 \) to \( t_1 \). Upon exit from \( s_2 \), \( c_0 \) becomes free on transition from \( s_2 \) to \( s_3 \) and \( c_1 \) becomes free on transition from \( s_2 \) to \( s_4 \). But \( s_5 \) is a problematic state, so on both transitions the algorithm must assign a third clock, say \( c_2 \), to \( t_2 \). Therefore, the number of allocated clocks is three, even though the number of active ranges on any transition is at most two.

Theorem 2 The clock allocation performed by Algorithms 2 is optimal, i.e., the final value of \(|C|\) cannot be decreased by performing another correct allocation.

Proof. The algorithm always tries to allocate the available clock that has the lowest number. All clocks in \( P_0 \setminus C \) have numbers that are higher than those of the clocks in \( C \). Therefore, \( C \) will never be increased if a clock from \( C \) can be used.

The set \( C \) is increased in one of two situations:

1. At some state \( s \), the set \( C \) does not contain an available clock, i.e., \( C \cap \text{pool}(s) = \emptyset \).
2. It is necessary to assign the same clock to a family of transitions. Members of the family release clocks in \( C \), but there is no clock in \( C \) that is available on all those transitions.

Observe that in case 1, the cardinality of \( C \) is equal to the number of ranges in some transition in \( \text{out}(s) \). This number does not depend on any particular clock allocation.

In case 2, there is no range that dies in all the members of the family. This fact cannot be changed by a different clock allocation.
Algorithm 3: Generating clocks, clock resets and clock constraints

Input: The set of clocks, \( C \), returned by Algorithm 2 and an extended time-annotated graph \( G_e = \langle E, Q_e, q^0_e, q^f_e, R_e, L \rangle \), where \( Q_e \) is the set of extended states and \( R_e \) is the set of extended transitions.

Output: A timed automaton \( M = \langle E, Q, q^0, q^f, C, \Theta \rangle \).

1. \( \Theta := \emptyset \);
2. foreach transition \( r = ((s, q, e, \phi), \text{born}(r), \text{active}(r), \text{dying}(r), \text{reachable}(r)) \) on a breadth-first walk of \( G_e \) do
3. \( \delta := \emptyset \);
4. foreach time annotation \( W - t_j \sim a \in \phi \) do
5. \( \delta = \delta \cup \{ c \sim a \} \), where \( (j, c) \in \text{clocks}(q) \);
6. if \( \text{born}(r) \neq \emptyset \) then
7. \( \lambda = \{ d \} \), where \( \text{born}(r) = \{ i \} \) and \( (i, d) \in \text{clocks}(q) \);
8. else
9. \( \lambda = \emptyset \);
10. \( r_a = (s, q, e, \lambda, \delta) \);
11. \( \Theta := \Theta \cup \{ r_a \} \);

3.3 Generating Clock Constraints and Clock Resets

The final step of our clock allocation method is to generate clock resets and clock constraints and assign them to the transitions. Algorithm 3 does this by (i) rewriting the time annotations of the graph to replace time variables by clocks, and (ii) assigning clock resets where ranges are born. For transforming the time annotation \( W - t_j \sim a \) of transition \( r = (s, q, e, \phi) \in R \), the set of clock assignments at state \( q \), i.e., \( \text{clocks}(q) \), will be used for obtaining the clock that is assigned to time variable \( t_j \). The time annotation \( W - t_j \sim a \) will be transformed into \( c \sim a \), if \( (j, c) \in \text{clocks}(q) \). Clock resets are generated by identifying transitions at which ranges are born. For a transition \( r = (s, q, e, \phi) \in R \), the clock reset \( d := 0 \) will be added to transition \( r \), if \( \text{born}(r) = \{ i \} \) and \( (i, d) \in \text{clocks}(q) \).

4 Related Work

To the best of our knowledge the problem of constructing timed automata from a set of scenarios has been addressed only by Somé et al [14]. However, the problem of minimizing the number of clocks is not addressed in that pioneering work. In fact, it is not clear how clocks, clock resets, and clock constraints are generated, and to which transitions the clock resets are assigned. Moreover, the use of contradictory scenarios may cause “unwanted non-determinism” in the constructed automata. We believe that this is undesirable, and that contradictory scenarios must be excluded from consideration before the synthesis process starts. The authors use the concept of “characteristic conditions” for generating states and identifying identical states, but the concept is not formally defined: in particular,
it is not obvious whether clock constraints are also considered as characteristic conditions. Moreover, arbitrary variables (e.g., a variable that counts the number of attempts for entering a PIN) are allowed in the constructed automata; this is unfaithful to both the syntax and the semantics of timed automata, which can feature only clock variables [2].

By contrast, in our approach we formally define a set of criteria for a complete set of scenarios from which a synthesis to a deterministic timed automaton is possible. We use modes, which are formally defined, as state labels and use these labels for identifying identical states. Our clock allocation algorithm precisely determines the minimum number of clocks required, the transitions along which each clock must get reset and the transitions where clock constraints must be added.

There has also been work on scenario-based synthesis of parametric timed automata [7], where scenarios with parametric timing constraints in the form of upper and lower time bounds are considered. Allowing parametric constraints in scenarios makes the corresponding synthesis methods not scalable.

The problem of reducing the number of clocks in timed automata has been addressed by constructing bisimilar timed automata [8, 5]. The approach used by Daws and Yovine [5] combines two methods for reducing the number of clocks. The first method is based on identifying the set of active clocks in each state of a timed automaton and applying a clock renaming to this set of active clocks locally, at each state, to obtain a bisimilar timed automaton. The second method is based on the notion of equality between clocks, where clocks that are equal in a state 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. Daws and Yovine consider a more general case than we do, i.e., cyclic timed automata. Their method will not always result in the minimum possible number of clocks, as argued by Guha et al. [8].

Guha et. al [8] 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. However, as a result of this the number of states of the constructed timed automaton may become exponential in the number of clocks of the original timed automaton. Working directly with zone graphs is a feasible approach for constructing bisimilar timed automata, but the algorithms are rather complicated.

In contrast to this work, our approach for reducing the number of clocks in a timed automaton is based on liveness analysis of time variables, and does not introduce new states. It should be noted that our method, while much simpler, is not as generally applicable.

Please note that with our approach there are no unused clocks and two clocks never have the same value, so the relevant techniques of Guha et al. [8] would not be applicable.
The time complexity of our clock allocation algorithm is bounded by the cost of the topological sort. In a graph with $|V|$ states and $|E|$ transitions, the time complexity of topological sort is $O(|V| + |E|)$. Therefore, in contrast to the algorithm of Guha et. al [8], the cost of our clock allocation algorithm is not exponential.

5 Conclusions

In our companion paper [13], we developed a new technique for specifying the required behaviors of real-time systems in terms of scenarios. We formulated the criteria for a set of scenarios to be a sufficient starting point for constructing a minimal, deterministic, acyclic timed automaton. We called such a set a complete set of scenarios and described a synthesis method for using it to construct a timed automaton. The clock allocation algorithm presented in [13] is somewhat naive: it does not minimize the number of clocks.

In the current paper, we proposed a novel approach for allocating clocks to the target timed automaton constructed from a complete set of scenarios. Our clock allocation method is based on performing liveness analysis of time variables appearing in scenarios. Our method is not applicable to arbitrary timed automata, because of the pivotal role of the dominance assumption; however, it is a general method that can be applied to minimize the number of clocks in any acyclic timed automaton whose graph satisfies that assumption. Such graphs arise naturally in the context of formal model synthesis, e.g., from our synthesis method for constructing timed automata from a set of scenarios [13].

Our synthesis method results in an optimal timed automaton, in the sense that both the number of states and the number of clocks are minimal.

Acknowledgement

The author is very grateful to Feliks Kluźniak for long hours of constructive discussions, valuable suggestions and help with editing the paper.

References


13. Saeedloei, N.: From scenarios to timed automata (Submitted)
