Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems

School of Information Systems

4-2014

Are timed automata bad for a specification language? Language inclusion checking for timed automata

Ting WANG

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Yang LIU

Xinyu WANG

Shanping LI

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Software Engineering Commons, and the Theory and Algorithms Commons

Citation

WANG, Ting; SUN, Jun; LIU, Yang; WANG, Xinyu; and LI, Shanping. Are timed automata bad for a specification language? Language inclusion checking for timed automata. (2014). *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Grenoble, France, April 5-13*. 310-325. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/4957

This Conference Proceeding Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.

Are Timed Automata Bad for a Specification Language? Language Inclusion Checking for Timed Automata*

Ting Wang[1], Jun Sun[2], Yang Liu[3], Xinyu Wang[1], and Shanping Li[1]

[1] College of Computer Science and Technology, Zhejiang University, China
[2] ISTD, Singapore University of Technology and Design, Singapore
[3] School of Computer Engineering, Nanyang Technological University, Singapore

Abstract. Given a timed automaton $\mathcal{P}$ modeling an implementation and a timed automaton $\mathcal{S}$ as a specification, language inclusion checking is to decide whether the language of $\mathcal{P}$ is a subset of that of $\mathcal{S}$. It is known that this problem is undecidable and "this result is an obstacle in using timed automata as a specification language" [2]. This undecidability result, however, does not imply that all timed automata are bad for specification. In this work, we propose a zone-based semi-algorithm for language inclusion checking, which implements simulation reduction based on Anti-Chain and LU-simulation. Though it is not guaranteed to terminate, we show that it does in many cases through both theoretical and empirical analysis. The semi-algorithm has been incorporated into the PAT model checker, and applied to multiple systems to show its usefulness and scalability.

1 Introduction

Timed automata, introduced by Alur and Dill in [2], have emerged as one of the most popular models to specify and analyze real-time systems. It has been shown that the reachability problem for timed automata is decidable using the construction of region graphs [2]. Efficient zone-based methods for checking both safety and liveness properties have later been developed [14,21]. In [2], it has also been shown that timed automata in general cannot be determinized, and the language inclusion problem is undecidable, which "is an obstacle in using timed automata as a specification language".

In order to avoid undecidability, a number of subclasses of timed automata which are determinizable (and perhaps serve as a good specification language) have been identified, e.g., event-clock timed automata [3,17], timed automata restricted to at most one clock [16] and integer resets timed automata [18]. Recently, Baier *et al.* [4] described a method for determinizing arbitrary timed automaton, which under a boundedness condition, yields an equivalent deterministic timed automaton in finite time. Furthermore, they show that the boundedness condition is satisfied by several subclasses of timed automata which are known to be determinizable. However, the method is based on region graphs and it is well-known that region graphs are inefficient and lead to state space explosion. Compared to region graphs, zone graphs are often used in existing tools for real-time system verification, such as UPPAAL [14] and KRONOS [23]. Zone-based approaches have also been used to solve problems which are related to the language

* This research is sponsored in part by NSFC Program (No.61103032) of China.

E. Ábrahám and K. Havelund (Eds.): TACAS 2014, LNCS 8413, pp. 310–325, 2014.
© Springer-Verlag Berlin Heidelberg 2014

inclusion problem, like the universality problem (which asks whether a timed automaton accepts all timed words) for timed automata with one-clock only [1]. However, to the best of our knowledge, there has not been any zone-based method proposed for language inclusion checking for arbitrary timed automata.

In this work we develop a zone-based method to solve the language inclusion problem. Formally, given an implementation timed automaton $\mathcal{P}$ and a specification timed automaton $\mathcal{S}$, the language inclusion problem is to decide whether the language of $\mathcal{P}$ is a subset of that of $\mathcal{S}$. It is known that the problem can be converted to a reachability problem on the synchronous product of $\mathcal{P}$ and determinization of $\mathcal{S}$ [16]. Inspired by [1,4], the main contribution of this work is that we present a semi-algorithm with a transformation that determinizes $\mathcal{S}$ and constructs the product on-the-fly, where zones are used as a symbolic representation. Furthermore, simulation relations between the product states are used, which can be obtained through LU-simulation [5] and Anti-Chain [22]. With the simulation relations, many product states may be skipped, which often contributes to the termination of our semi-algorithm.

Our semi-algorithm can be applied to arbitrary timed automata, though it may not terminate sometimes. To argue that timed automata can nonetheless serve as a specification language, we investigate when our approach is terminating, both theoretically and empirically. Firstly, we prove that, with the clock boundedness condition [4], we are able to construct a suitable well-quasi-order on the product state space to ensure termination. It thus implies that our semi-algorithm is always terminating for subclasses of timed automata which are known to be determinizable. Furthermore, we prove that for some classes of timed automata which may violate the boundedness condition, our semi-algorithm is always terminating as long as there is a well-quasi-order on the abstract state space explored. Secondly, using randomly generated timed automata, we show that our approach terminates for many timed automata which are not determinizable (and violating the boundedness condition) because of the simulation reduction. Thirdly, we collect a set of commonly used patterns for specifying timed properties [8,12] and show that our approach is always terminating for all of those properties. Lastly, our semi-algorithm has been implemented in the PAT [19] framework, and applied to a number of benchmark systems to demonstrate its effectiveness and scalability.

The remainders of the paper is organized as follows. Section 2 reviews the notions of timed automata. Section 3 shows how to reduce language inclusion checking to a reachability problem, which is then solved using a zone-based approach. Section 4 reports the experimental results. Section 5 reviews related work. Section 6 concludes.

2 Background

In this section, we review the relevant background and define the language inclusion problem. We start with defining labeled transition systems (LTS). An LTS is a tuple $\mathcal{L} = (S, Init, \Sigma, T)$, where S is a set of states; $Init \subseteq S$ is a set of initial states; Σ is an alphabet; and $T \subseteq S \times \Sigma \times S$ is a labeled transition relation. A run of $\mathcal{L}$ is a finite sequence of alternating states and events $\langle s_0, e_1, s_1, e_2, \cdots, e_n, s_n \rangle$ such that $(s_i, e_i, s_{i+1}) \in T$ for all $0 \leq i \leq n-1$. We say the run starts with s_0 and ends with s_n. A state s' is reachable from s iff there is a run starting with s and ending with s'. A state

is always reachable from itself. A run is rooted if it starts with a state in $Init$. A state is reachable if there is a rooted run which ends at the state. Given a state $s \in S$ and an event $e \in \Sigma$, we write $post(s, e, \mathcal{L})$ to denote $\{s' | (s, e, s') \in T\}$. We write $post(s, \mathcal{L})$ to denote $\{s' | \exists e \in \Sigma \cdot (s, e, s') \in T\}$, i.e., the set of successors of s.

Let $F \subseteq S$ be a set of target states. Given two states s_0 and s_1 in S, we say that s_0 is simulated by s_1 with respect to F if $s_0 \in F$ implies that $s_1 \in F$; and for any $e \in \Sigma$, $(s_0, e, s'_0) \in T$ implies there exists $(s_1, e, s'_1) \in T$ such that s'_0 is simulated by s'_1. In order to check whether a state in F is reachable, if we know that s is simulated by s', then s can be skipped during system exploration if s' has been explored already. This is known as simulation reduction [7].

The original definition of timed automata is finite-state timed Büchi automata [2] equipped with real-valued clock variables and Büchi accepting condition (to enforce progress). Later, timed safety automata were introduced in [11] which adopt an intuitive notion of progress. That is, instead of having accepting states, each state in timed safety automata is associated with a local timing constraint called a *state invariant*. An automaton can stay at a state as long as the valuation of the clocks satisfies the state invariant. The reader can refer to [9] for the expressiveness of timed safety automata. In the following, we focus on timed safety automata as they are supported in the state-of-art model checker UPPAAL [14] and are often used in practice. Hereafter, they are simply referred to as timed automata.

Let $\mathbb{R}^+$ be the set of non-negative real numbers. Given a set of clocks C, we define $\Phi(C)$ as the set of clock constraints. Each clock constraint is inductively defined by: $\delta := true | x \sim n | \delta_1 \wedge \delta_2 | \neg \delta_1$ where $\sim \in \{=, \leq, \geq, <, >\}$; x is a clock in C and $n \in \mathbb{R}^+$ is a constant. Without loss of generality, we assume that n is an integer constant. The set of downward constraints obtained with $\sim \in \{\leq, <\}$ is denoted as $\Phi_{\leq,<}(C)$. A clock valuation v for a set of clocks C is a function which assigns a real value to each clock. A clock constraint can be viewed as the set of clock valuations which satisfy the constraint. A clock valuation v satisfies a clock constraint δ, written as $v \in \delta$, iff δ evaluates to be true using the clock values given by v. For $d \in \mathbb{R}^+$, let $v + d$ denote the clock valuation v' s.t. $v'(c) = v(c) + d$ for all $c \in C$. For $X \subseteq C$, let clock resetting notion $[X \mapsto 0]v$ denote the valuation v' such that $v'(c) = v(c)$ for all $c \in C \wedge c \notin X$, and $v'(x) = 0$ for all $x \in X$. We write $C = 0$ to be the clock valuation where each clock $c \in C$ reads 0.

Formally, a timed automaton is a tuple $\mathcal{A} = (S, Init, \Sigma, C, L, T)$ where S is a finite set of states; $Init \subseteq S$ is a set of initial states; Σ is an alphabet; C is a finite set of clocks; $L : S \rightarrow \Phi_{\leq,<}(C)$ labels each state with an invariant; $T \subseteq S \times \Sigma \times \Phi(C) \times 2^C \times S$ is a labeled transition relation. Intuitively, a transition $(s, e, \delta, X, s') \in T$ can be fired if δ is satisfied. After event e occurs, clocks in X are set to zero. The (concrete) semantics of $\mathcal{A}$ is an infinite-state LTS, denoted as $\mathcal{C}(\mathcal{A}) = (S_c, Init_c, \mathbb{R}^+ \times \Sigma, T_c)$ such that S_c is a set of configurations of $\mathcal{A}$, each of which is a pair (s, v) where $s \in S$ is a state and v is a clock valuation; $Init_c = \{(s, C = 0) | s \in Init\}$ is a set of initial configurations; and T_c is a set of concrete transitions of the form $((s, v), (d, e), (s', v'))$ such that there exists a transition $(s, e, \delta, X, s') \in T$; $v + d \in \delta$; $v + d \in L(s)$; $[X \mapsto 0](v + d) = v'$; and $v' \in L(s')$. Intuitively, the system idles for d time units at state s and then take the transition (generating event e) to reach state s'. An example timed automaton is shown in Fig. 1(a). The initial state is p_0. The automaton has a state

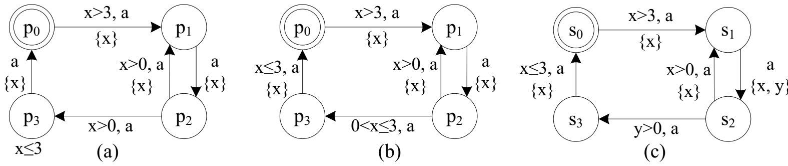


Fig. 1. Timed automata examples

invariant $x \leq 3$ on state p_3 which implies that if the control is at p_3, it must transit to the next state before the value of clock x is larger than 3.

A timed automaton $\mathcal{A}$ is deterministic iff $Init$ contains only one state and for any two transitions $(s_0, e_0, \delta_0, X_0, s_0') \in T$ and $(s_1, e_1, \delta_1, X_1, s_1') \in T$, if $s_0 = s_1$ and $e_0 = e_1$, then δ_0 and δ_1 are mutually exclusive. Otherwise, $\mathcal{A}$ is non-deterministic. For instance, The timed automaton in Fig. 1(c) is non-deterministic as the two transitions from state s_2 are both labeled with a and the guards are not mutually exclusive.

Given $\langle (s_0, v_0), (d_1, e_1), (s_1, v_1), (d_2, e_2), \cdots (s_n, v_n) \rangle$ as a run of $\mathcal{C}(\mathcal{A})$, we can obtain a timed word: $\langle (D_1, e_1), (D_2, e_2), \cdots, (D_n, e_n) \rangle$ so that $D_i = \sum_{j=1}^{i} d_j$ for all $1 \leq i \leq n$. We define the $\mathcal{L}(\mathcal{A}, (s, v))$ to be the set of timed words obtained from the set of all runs starting with (s, v). The language of $\mathcal{A}$, written as $\mathcal{L}(\mathcal{A})$, is defined as the language obtained from any rooted run of $\mathcal{A}$. Two timed automata are equivalent if they define the same language. In practice, a system model is often composed of several automata executing in parallel. We skip the details on parallel composition of timed automata and remark our approach in this work applies to networks of timed automata. The language inclusion checking problem is then defined as follows. Given a timed automaton $\mathcal{P}$ and a timed automaton $\mathcal{S}$, how do we check whether $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{S})$?

In order to simplify the presentation in later sections, we first transform a given timed automaton to an equivalent one without state invariants, which will not affect our approach. The idea is to move the state invariants to transition guards. Given a timed automaton $\mathcal{A}$ and a state s with state invariant $L(s)$, we construct a timed automaton $\mathcal{A}'$ with the following two steps. Firstly, if (s, e, δ, X, s') is a transition from s, change it to $(s, e, \delta \wedge L(s), X, s')$. Secondly, if (s', e, δ, X, s) is a transition leading to s, for any clock constraint of the form $x \sim n$ where $\sim \in \{\leq, <\}$ in $L(s)$, if $x \notin X$, conjunct δ with $x \sim n$. For instance, given the timed automaton in Fig. 1(a), we construct the timed automaton in Fig. 1(b). The state invariant $x \leq 3$ of state p_3 is added to the transition from p_2 to p_3 and the transition from p_3 to p_0. By a simple argument, it can be shown that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$. Notice that this transformation is not sound if the language of a timed automaton is defined differently, e.g., with a non-Zenoness assumption. In the following, we assume that all timed automata are without state invariants.

3 Language Inclusion Checking

In this section, we present our method on solving the language inclusion checking problem. We fix $\mathcal{P} = (S_p, Init_p, \Sigma_p, C_p, L_p, T_p)$ and $\mathcal{S} = (S_s, Init_s, \Sigma_s, C_s, L_s, T_s)$ to be the two timed automata such that S_p and S_s are disjoint as well as C_p and C_s.[1]

[1] The proofs in this section can be found at
`http://www.comp.nus.edu.sg/~pat/refine_ta/paper.pdf`

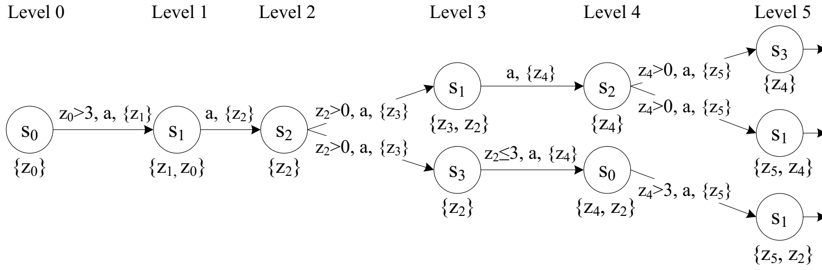


Fig. 2. Unfolding a timed automaton into an infinite timed tree

3.1 Unfolding Specification

In our method, we construct on-the-fly an unfolding of $\mathcal{S}$ in the form of an infinite timed tree which is equivalent to $\mathcal{S}$. The idea is adopted from the approach in [4]. Before we present the formal definition, we illustrate the unfolding using an example. Fig. 2 shows the infinite timed tree after unfolding the automaton shown in Fig. 1(c). The idea is to introduce a fresh clock at every level and use the newly introduced clocks to replace ordinary clocks, i.e., x and y in this example. The benefit of doing this becomes clear later. At level 0, we are at state s_0 and introduce a clock z_0. Now since clock x and clock y are started at the same time as z_0 and the clocks will not be reset before the transition from s_0 takes place, we can use z_0 to replace x and y in the transition guard from s_0 at level 0 to s_1 at level 1. Because at level 0, the reading of clock z_0 is relevant to the future system behavior, we say that z_0 is active. In the tree, we label every node with a pair (s, A) where s is a state and A is a set of active clocks. Notice that not all clocks are active. For instance, clock z_0 and clock z_1 are no longer active at level 2.

One transition from the level 0 node leads to the node of level 1, corresponding to the transition from state s_0 to state s_1 in Fig. 1(c). The clock constraint $x > 3$ is rewritten to $z_0 > 3$ using only active clocks from the source node. A fresh clock z_1 is introduced along the transition. Notice that the node at level 1 is labelled with a set of two active clocks. z_1 is active at state s_1 at level 1 since it can be used to replace clock x which is reset along the transition, whereas z_0 is active because it is used to replace clock y which is not reset along the transition. The set of active clocks of the node at level 2 is a singleton z_2 since both of the clocks x and y are reset along the transition. z_0 and z_1 are not active as their reading is irrelevant to future transitions from s_2. Following the same construction, we build the tree level by level.

In the following, we define the unfolding of $\mathcal{S}$. Let $Z = \langle z_0, z_1, z_2, \cdots \rangle$ be an infinite sequence of clocks. The unfolding $\mathcal{S}$ is an infinite timed tree, which can be viewed as a timed automaton $\mathcal{S}_\infty = (St_\infty, Init_\infty, \Sigma_\infty, Z, T_\infty)$ with infinitely many states. Furthermore, we assume that $\mathcal{S}_\infty$ is associated with function $level$ such that $level(n)$ is the level of node n in the tree for all $n \in St_\infty$. A state n in St_∞ is in the form of (s, A) where $s \in S_s$ and A is a set of clocks. Given any state n, we define a function $f_n : C_s \mapsto Z$ which maps ordinary clocks in C_s to active clocks in Z. In an abuse of notations, given a clock constraint δ on C_s, we write $f_n(\delta)$ to denote the clock constraint obtained by replacing clocks in C_s with those in Z according to f_n. Given any state

$n = (s, A)$, we define A to be $\{f_n(c)|c \in C_s\}$. The initial states $Init_\infty$ and transition relation T_∞ are unfolded as follows.

- For any $s \in Init_s$, there is a level-0 node $n = (s, \{z_0\})$ in St_∞ with $level(n) = 0$, $f_n(c) = z_0$ for all $c \in C_s$.
- For each node $n = (s, A)$ at level i and for each transition $(s, e, \delta, X, s') \in T_s$, we add a node $n' = (s', A')$ at level $i + 1$ such that $f_{n'}(c) = f_n(c)$ if $c \in C_s \setminus X$, $f_{n'}(c) = z_{i+1}$ if $c \in X$; $level(n') = i + 1$. We add a transition $(n, e, f_n(\delta), \{z_{i+1}\}, n')$ to T_∞.

Note that transitions at the same level have the same set of resetting clocks, which contains one clock. Given a node $n = (s, A)$ in the tree, observe that not every clock x in A is active as the clock may never be used to guard any transition from s. Hereafter, we assume that inactive clocks are always removed.

3.2 Zone Abstraction for Language Inclusion Checking

It can be shown that $\mathcal{S}$ and $\mathcal{S}_\infty$ are equivalent [4]. Intuitively, $\mathcal{S}$ and $\mathcal{S}_\infty$ have the same language, thus the language inclusion problem can be converted to the language inclusion problem between $\mathcal{P}$ and $\mathcal{S}_\infty$. To solve the problem, we have to deal with two sources of infinity. One is that there are infinitely many clocks and the other is there are infinitely many clock valuations for each clock. In the following, we tackle the latter with zone abstraction [14].

In this work, we define a zone (which may or may not be convex) as the maximum set of clock valuations satisfying a clock constraint. Given a clock constraint δ, let $\delta^\uparrow$ denote the zone reached by delaying an arbitrary amount of time. For $X \subseteq C$, let $[X \mapsto 0]\delta$ denote the zone obtained by setting clocks in X to 0; and let $\delta[X]$ denote the projection of δ on X.

We define an LTS $\mathcal{Z}_\infty = (S, Init, \Sigma, T)$, which is a zone graph generated from the synchronous product of $\mathcal{P}$ and the determinization of $\mathcal{S}_\infty$. A state in S is an abstract configuration of the form (s_p, X_s, δ) such that $s_p \in S_p$; X_s is a set of nodes in $\mathcal{S}_\infty$ as defined in Section 3.1; and δ is a clock constraint. Recall that a state of $\mathcal{S}_\infty$ is of the form (s_s, A) where $s_s \in S_s$ and A is a set of active clocks. Given a set of states X_s of $\mathcal{S}_\infty$, we write $Act(X_s)$ to denote the set of all active clocks, i.e., $\{c|\exists(s_s, A) \in X_s \cdot c \in A\}$. δ constraints all clocks in $Act(X_s)$.

The $Init$ of the zone graph is defined as: $\{(s_p, Init_\infty, (Act(Init_\infty) = 0)^\uparrow)|s_p \in Init_p\}$. Σ equals to Σ_p. Next, we define T by showing how to generate successors of a given abstract configuration (s_p, X_s, δ). For every state $(s_s, A) \in X_s$, let $T_\infty(e, X_s)$ be the set of transitions in T_∞ which start with a state in X_s and are labeled with event e. Notice that the guard conditions of transitions in $T_\infty(e, X_s)$ may not be mutually exclusive. We define a set of constraints $Cons(e, X_s)$ such that each element in $Cons(e, X_s)$ is a constraint which conjuncts, for each transition in $T_\infty(e, X_s)$, either the transition guard or its negation. Notice that elements in $Cons(e, X_s)$ are by definition mutually exclusive. Given (s_p, X_s, δ) and an outgoing transition (s_p, e, g_p, X_p, s_p') from s_p in $\mathcal{P}$, for each $g \in Cons(e, X_s)$ we generate a successor (s_p', X_s', δ') as follows.

- For any state $(s_s, A) \in X_s$ and any transition $((s_s, A), e, g_s, Y, (s_s', A')) \in T_\infty$, if $\delta \wedge g_p \wedge g \wedge g_s$ is not false, then $(s_s', A') \in X_s'$.

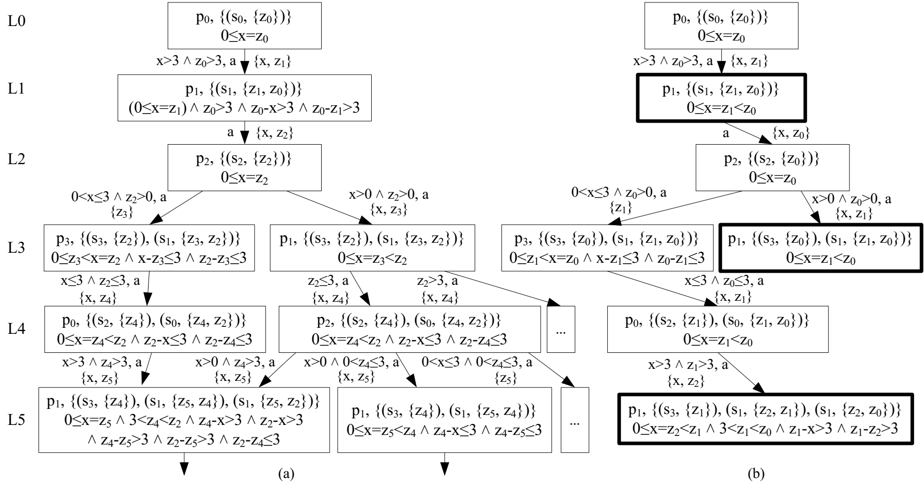


Fig. 3. Zone graphs: (a) $\mathcal{Z}_\infty$ and (b) $\mathcal{Z}_r^{LU}$

– All states in X_s are at the same level and thus all transitions in $T_\infty(e, X_s)$ have the same resetting clock. Let Y be that clock and $\delta' = ([Y \cup X_p \mapsto 0](\delta \wedge g \wedge g_p))^\uparrow$.
– The transition from (s_p, X_s, δ) to (s'_p, X'_s, δ') is labeled with the tuple $(e, g_p \wedge g, X_p \cup Y)$.

We illustrate the above using the example in Fig. 3(a). Given the abstract configuration in level 4, which is $(p_2, \{(s_2, \{z_4\}), (s_0, \{z_4, z_2\})\}, 0 \leq x = z_4 < z_2 \wedge z_2 - x \leq 3 \wedge z_2 - z_4 \leq 3)$. As shown in Fig. 2, there are two transitions from state $(s_2, \{z_4\})$ which are labeled with event a and one from state $(s_0, \{z_4, z_2\})$, which makes up $T_s(a, \{(s_2, \{z_4\}), (s_0, \{z_4, z_2\})\})$. The two transitions from $(s_2, \{z_4\})$ have the same guard $z_4 > 0$ and the one from $(s_0, \{z_4, z_2\})$ has the guard $z_4 > 3$. The set $Cons(e, X_s)$ contains the following constraints: $z_4 > 0 \wedge z_4 > 3$, $z_4 \leq 0 \wedge z_4 > 3$, $z_4 > 0 \wedge z_4 \leq 3$, and $z_4 \leq 0 \wedge z_4 \leq 3$. Taking the transition form p_2 to p_1 as an example, we generate four potential successors for each of constraints in $Cons(e, X_s)$, as shown above. Two of them are infeasible as the resultant constraints are false. The rest two are shown in Fig. 3(a) (the first two from left at level 5). Since z_2 is no longer active for the second successor, the clock constraint of the second successor is modified to $0 \leq x = z_5 < z_4 \wedge z_4 - x \leq 3 \wedge z_4 - z_5 \leq 3$ so as to remove constraints on z_2. Similarly, we can generate other configurations in Fig. 3(a).

In the following, we reduce the language inclusion checking problem to a reachability problem in $\mathcal{Z}_\infty$. Notice that one of constraints in $Cons(e, X_s)$ conjuncts the negations of all guards of transitions in $T_\infty(e, X_s)$. Let us denote the constraint as neg. For instance, given the same abstract state in the middle of level 4 in Fig. 3(a), the constraint neg in $Cons(e, X_s)$ is: $z_4 \leq 0 \wedge z_4 \leq 3$, which is equivalent to $z_4 \leq 0$. Conjuncted with the guard condition $x > 0$ and the initial constraint $0 \leq x = z_4 < z_2 \wedge z_2 - x \leq 3 \wedge z_2 - z_4 \leq 3$, it becomes false and hence no successor is generated for neg. Given neg, assume the corresponding successor is (s'_p, X'_s, δ'). It is easy to see

that X'_s is empty. If δ' is not false, intuitively there exists a time point such that $\mathcal{P}$ can perform e whereas $\mathcal{S}$ cannot, which implies language inclusion is not true. Thus, we have the following theorem.

Theorem 1. *$\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{S})$ iff there is no reachable state $(s_p, \emptyset, \delta)$ in $\mathcal{Z}_\infty$.* □

Theorem 1 therefore reduces our language inclusion problem to a reachability problem on $\mathcal{Z}_\infty$. If a state in the form of $(s_p, \emptyset, \delta)$ is reachable, then we can conclude that the language inclusion is false. The remaining problem is that there may be infinitely many clocks. In the following, we show how to reduce the number of clocks, which is inspired by [4]. Intuitively, given any abstract state (s_p, X_s, δ) in the zone graph $\mathcal{Z}_\infty$, instead of always using a new clock in Z, we can reuse a clock which is not currently active, or equivalently not in $Act(X_s)$. For instance, given the state on level 1 in Fig. 3(a), there are two active clocks z_0 and z_1. For the successor of this state on level 2, instead of using z_2, we can reuse z_0 and systematically rename z_2 to z_0 afterwards. The result of renaming is shown partially in Fig. 3(b) (notice that some zones in Fig. 3(b) are different from the ones in Fig. 3(a) and some states have been removed, because of simulation reduction shown next). We denote the zone graph after renaming as $\mathcal{Z}_r$. We also denote the successors of an abstract state ps in $\mathcal{Z}_r$ as $post(ps, \mathcal{Z}_r)$. By a simple argument, it can be shown that there is a reachable state $(s_p, \emptyset, \delta)$ in $\mathcal{Z}_\infty$ iff there is a reachable state $(s'_p, \emptyset, \delta')$ in $\mathcal{Z}_r$.

3.3 Simulation Reduction

We have so far reduced the language inclusion checking problem to a reachability problem in the potentially infinite-state LTS $\mathcal{Z}_r$. Next, we reduce the size of $\mathcal{Z}_r$ by exploring simulation relation between states in $\mathcal{Z}_r$. We first extend the lower-upper bounds (hereafter LU-bounds) simulation relation defined in [5] to language inclusion checking.

We define two functions L and U. Given a state s in $\mathcal{Z}_r$ and a clock $x \in C_p \cup Z$, we perform a depth-first-search to collect all transitions reachable from s without going through a transition which resets x. Next, we set $L(s, x)$ (resp. $U(s, x)$) to be the maximal constant k such that there exists a constraint $x > k$ or $x \geq k$ (resp. $x < k$ or $x \leq k$) in a guard of those transitions. If such a constant does not exist, we set $L(s, x)$ (resp. $U(s, x)$) to $-\infty$. We remark that $L(s, x)$ is always the same as $U(s, x)$ for a clock in Z because both guard conditions and their negations are used in constructing $\mathcal{Z}_r$. For instance, if we denote the state at level 0 in Fig. 3(b) as s_0, which can be seen as the initial state in $\mathcal{Z}_r$, the function L is then defined such that $L(s_0, x) = 3$, $L(s_0, z_0) = 3$.

Next, we define a relation between two zones using the LU-bounds and show that the relation constitutes a simulation relation. Given two clock valuations v and v' at a state s and the two functions L and U, we write $v \preccurlyeq_{LU} v'$ if for each clock c, either $v'(c) = v(c)$ or $L(s, c) < v'(c) < v(c)$ or $U(s, c) < v(c) < v'(c)$. Next, given two zones δ_1 and δ_2, we write $\delta_1 \preccurlyeq_{LU} \delta_2$ to denote that for all $v_1 \in \delta_1$, there is a $v_2 \in \delta_2$ such that $v_1 \preccurlyeq_{LU} v_2$. The following shows that $\preccurlyeq_{LU}$ constitutes a simulation relation.

Lemma 1. *Let (s, X, δ_i) where $i \in \{0, 1\}$ be two states of $\mathcal{Z}_r$ and F be the set of states $\{(s', \emptyset, \delta')\}$ in $\mathcal{Z}_r$. (s, X, δ_1) simulates (s, X, δ_0) w.r.t. F if $\delta_0 \preccurlyeq_{LU} \delta_1$.* □

With the above lemma, given an abstract state (s, X, δ) of $\mathcal{Z}_r$, we can enlarge the time constraint δ so as to include all clock valuations which are simulated by some valuations in δ without changing the result of reachability analysis. In the following, we write $LU(\delta)$ to denote the set $\{v | \exists v' \in \delta \cdot v \preccurlyeq_{LU} v'\}$[2]. We construct an LTS, denoted as $\mathcal{Z}_r^{LU}$ which replaces each state (s, X, δ) in $\mathcal{Z}_r$ with $(s, X, LU(\delta))$. We denote the successors of a state ps in $\mathcal{Z}_r^{LU}$ as $post(ps, \mathcal{Z}_r^{LU})$. By a simple argument, we can show that there is a reachable state $(s, \emptyset, \delta)$ in $\mathcal{Z}_r$ iff there is a reachable state $(s', \emptyset, \delta')$ in $\mathcal{Z}_r^{LU}$. For instance, given the $\mathcal{Z}_r$ after renaming $\mathcal{Z}_\infty$ shown in Fig. 3(a), Fig. 3(b) shows the corresponding $\mathcal{Z}_r^{LU}$.

Next, we incorporate another simulation relation in our work which is inspired by the Anti-Chain algorithm [22]. The idea is that given two abstract states (s, X, δ) and (s', X', δ') of $\mathcal{Z}_r^{LU}$, we can infer a simulation relation by comparing X and X'. One problem is that states in X and X' may have different sets of active clocks. The exact names of the clocks however do not matter semantically. In order to compare X and X' (and compare δ and δ'), we define clock mappings. A mapping from $Act(X')$ to $Act(X)$ is a injection function $f : Act(X') \rightarrow Act(X)$ which maps every clock in $Act(X')$ to one in $Act(X)$. We write $X' \subseteq_f X$ if there exists a mapping f such that for all $(s'_s, A') \in X'$, there exists $(s_s, A) \in X$ such that $s_s = s'_s$ and for all $x \in A'$, $f(x) \in A$. Notice that there might be clocks in $Act(X)$ which are not mapped to. We write $range(f)$ to denote the set of clocks which are mapped to in $Act(X)$. With an abuse of notations, given a constraint δ' constituted by clocks in $Act(X')$, we write $f(\delta')$ to denote the constraint obtained by renaming the clocks accordingly to f. We write $\delta \subseteq_f \delta'$ if $\delta[range(f)] \subseteq f(\delta')$, i.e., the clock valuations which satisfy the constraint $\delta[range(f)]$ (obtained by projecting δ onto clocks in $Act(X')$) satisfy δ' after clock renaming. Next, we define a relation between two abstract configurations. We write $(s, X, \delta) \sqsubseteq (s', X', \delta')$ iff the following are satisfied: $s = s'$ and there exists a mapping f such that $X' \subseteq_f X$ and $\delta \subseteq_f \delta'$. The next lemma establishes that $\sqsubseteq$ is a simulation relation.

Lemma 2. *Let (s, X, δ) and (s', X', δ') be states in $\mathcal{Z}_r^{LU}$. Let $F = \{(s, \emptyset, \delta_0)\}$ be the set of target states. (s', X', δ') simulates (s, X, δ) w.r.t. F if $(s, X, \delta) \sqsubseteq (s', X', \delta')$.* □

For example, let ps_0 denote the state at level 1 in Fig. 3(a). Let ps_1 denote the bold-lined state at level 1 and ps_2 denote the one at level 5 in Fig. 3(b). With the LU simulation relation, ps_0 can be replaced by ps_1. A renaming function f can be defined from clocks in ps_1 to clocks in ps_2, i.e., $f(z_0) = z_1$ and $f(z_1) = z_2$. After renaming, ps_1 becomes $(p_1, \{(s_1, \{z_2, z_1\})\}, 0 \leq x = z_2 < z_1)$. Therefore, $ps_2 \sqsubseteq ps_1$ and hence we do not need to explore from ps_2. Similarly, we do not need to explore from the bold-lined state at level 3 in Fig. 3(b), namely ps_3. Notice that without the LU simulation reduction $ps_3 \sqsubseteq ps_1$ cannot hold, and the successors of ps_3 must be explored.

3.4 Algorithm

In the following, we present our semi-algorithm. Let $\mathcal{Z}_r^{LU}$ be the tuple $(S, Init, \Sigma, T)$ where $Init$ is a set $(init_p, Init_s, LU((C_p = 0 \land z_0 = 0)^{\uparrow}))$. Algorithm 1 constructs

[2] Notice that we may not be able to represent this set as a convex time constraint [5].

Algorithm 1. Language inclusion checking

```
1: let working := Init;
2: let done := ∅;
3: while working ≠ ∅ do
4:     remove ps = (s_p, X_s, δ) from working;
5:     add ps into done and remove all ps' ∈ done s.t. ps' ⊑ ps;
6:     for all (s'_p, X'_s, δ') ∈ post(ps, Z_r^LU) do
7:         if X'_s = ∅ then
8:             return false;
9:         end if
10:        if ∄ ps' ∈ done such that (s'_p, X'_s, δ') ⊑ ps' then
11:            put (s'_p, X'_s, δ') into working;
12:        end if
13:    end for
14: end while
15: return true;
```

$\mathcal{Z}_r^{LU}$ on-the-fly while performing reachability analysis with simulation reduction. It maintains two data structures. One is a set $working$ which stores states in S which are yet to be explored. The other is a set $done$ which contains states which have already been explored. Initially, $working$ is set to be $Init$ and $done$ is empty. During the loop from line 3 to line 14, each time a state is removed from $working$ and added to $done$. Notice that in order to keep $done$ small, whenever a state ps is added into $done$, all states which are simulated by ps are removed. We generate successors of ps at line 6. For each successor, if it is a target state, we return false at line 8. If it is simulated by a state in $done$, it is ignored. Otherwise, it is added into $working$ so that it will be explored later. Lastly, we return true at line 15 after exploring all states. We remark that $done$ is an *Anti-Chain* [22] as any pair of states in $done$ is incomparable. The following theorem states that the semi-algorithm always produces correct results.

Theorem 2. *Algorithm 1 returns true iff* $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{S})$. □

Next, we establish sufficient conditions for the termination of semi-algorithm with the theorems of *well quasi-order* (WQO [15]). A *quasi-order* (QO) on a set $\mathcal{A}$ is a pair $(\mathcal{A}, \preccurlyeq)$ where $\preccurlyeq$ is a reflective and transitive binary relation in $\mathcal{A} \times \mathcal{A}$. A QO is a WQO if for each infinite sequence $\langle a_0, a_1, a_2, \ldots\rangle$ composed of the elements in $\mathcal{A}$, there exists $i < j$ such that $a_j \preccurlyeq a_i$. Therefore if a WQO can be found among states in $\mathcal{Z}_r^{LU}$ with the simulation relation $\sqsubseteq$, our semi-algorithm terminates, as stated in the following theorem.

Theorem 3. *Let S be the set of states of $\mathcal{Z}_r^{LU}$. If $(S, \sqsubseteq)$ is a WQO, Algorithm 1 is terminating.* □

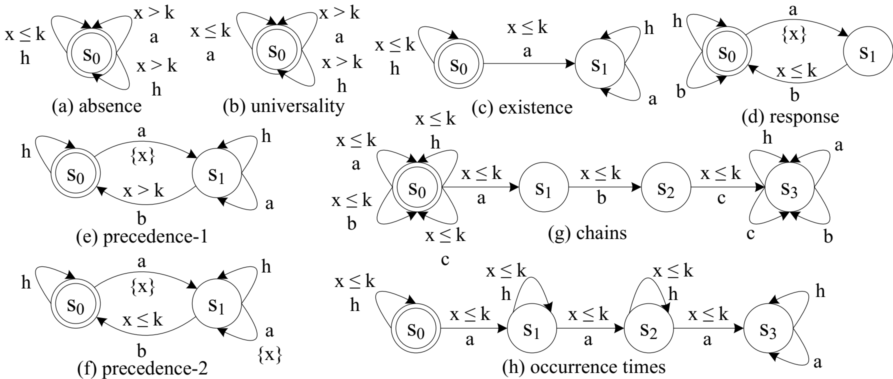
The above theorem implies that our semi-algorithm always terminates given the subclass of timed automata satisfying the clock boundedness condition [4], including strongly non-Zeno timed automata, event-clock timed automata and timed automata with integer resets. That is, if the boundedness condition is satisfied, $\mathcal{Z}_r^{LU}$ has a bounded number

of clocks and if the number of clocks are bounded, obviously the set S is finite (with maximum ceiling zone normalization). Since $(S, =)$ is a WQO if S is finite by a property of WQO, and '=' implies '$\sqsubseteq$', $(S, \sqsubseteq)$ is also a WQO for this special case. Furthermore, the theorem also shows that the semi-algorithm is terminating for all single-clock timed automata, as a WQO has been shown in [1], which may not satisfy the boundedness condition.

4 Evaluation

Our method has been implemented with 46K lines of C# code and integrated into the PAT model checker [19][3]. We remark that in our setting, a zone may not be convex (for instance, due to negation used in constructing $\mathcal{Z}_r$) and thus cannot be represented as a single difference bound matrix (DBM). Rather it can be represented either as a difference bound logic formula, as shown in [3], or as a set of DBMs. In this work, the latter approach is adopted for the efficiency reason. In the following, we evaluate our approach in order to answer three research questions. All experiment data are obtained using a PC with Intel(R) Core(TM) i7-2600 CPU at 3.40 GHz and 8.0 GB RAM.

The first question is: are timed automata good to specify commonly used timed properties? That is, if timed automata are used to model the properties, will our semi-algorithm terminate? In [8,12], the authors summarized a set of commonly used patterns for real-time properties. Some of the patterns are shown below where a, b, c are events; x is a clock and h denotes all the other events. Most of the patterns are self-explanatory and therefore we refer to the readers to [8,12] for details. We remark that although the patterns below are all single-clock timed automata, a specification may be the parallel composition of multiple patterns and hence have multiple clocks. Observe that all timed automata below are deterministic except (g). A simple investigation shows that (g) satisfies the clock boundedness condition and hence our semi-algorithm terminates for all the properties below.



(a) absence (b) universality (c) existence (d) response (e) precedence-1 (g) chains (f) precedence-2 (h) occurrence times

The second question is: is the semi-algorithm useful in practice? That is, given a real-world system, is it scalable? In the following, we model and verify benchmark

[3] PAT and the experiment details can be found at http://www.comp.nus.edu.sg/~pat/refine_ta

Table 1. Experiments on Language Inclusion Checking for Timed Systems

System	$\lvert C_s \rvert$	Det	$\sqsubseteq +LU$			LU			$\sqsubseteq$		
			stored	total	time	stored	total	time	stored	total	time
Fischer*8	1	Yes	91563	224208	28.3	138657	300384	516.7	-	-	-
Fischer*6	6	No	38603	78332	537.0	-	-	-	-	-	-
Fischer*6	2	No	27393	58531	6.8	36218	70348	30.3	-	-	-
Fischer*7	2	No	121782	271895	42.9	159631	326772	661.7	-	-	-
Railway*8	1	Yes	796154	1124950	142.1	-	-	-	-	-	-
Railway*6	6	No	23265	33427	7.2	27903	39638	20.4	-	-	-
Railway*7	7	No	180034	260199	66.7	222806	318698	1352.8	-	-	-
Lynch*5	1	Yes	3852	11725	0.6	16193	48165	6.0	45488	421582	377.2
Lynch*7	1	Yes	79531	400105	34.9	-	-	-	-	-	-
Lynch*5	2	No	8091	29686	2.4	63623	208607	151.3	56135	324899	290.1
Lynch*6	2	No	35407	162923	16.7	477930	1828668	5751.1	-	-	-
FDDI*7	7	Yes	1198	1590	7.4	8064	9592	36.4	8452	11836	125.5
CSMA*7	1	Yes	9840	36255	4.5	-	-	-	-	-	-

timed systems using our semi-algorithm and evaluate its performance. The benchmark systems include Fischer's mutual exclusion protocol (Fischer for short, similarly hereinafter), Lynch-Shavit's mutual exclusion protocol (Lynch), railway control system (Railway), fiber distributed data interface (FDDI), and CSMA/CD protocol (CSMA). The results are shown in Table 1. The systems are all built as networks of timed automata, and the number of processes is shown in column 'System'. The verified properties are requirements on the systems specified using the timed patterns. Some of the properties contain one timed automaton with one clock, while the rest are networks of timed automata with more than one clock (one clock for each timed automaton). In the table, column '$\lvert C_s \rvert$' is the number of clocks (processes) in the specification. The systems in the same group, e.g., Fischer*6 and Fischer*7 both with $\lvert C_s \rvert = 2$, have the same specification. Notice that the number of processes in a system and the one in the specification can be different because we can 'hide' events in the systems and use h in the specifications as shown in the patterns. Column 'Det' shows whether the specification is deterministic or not. The results of our semi-algorithm are shown in column '$\sqsubseteq +LU$'. In order to show the effectiveness of simulation reduction, we show the results without $\sqsubseteq$ reduction in column LU and the results without LU-reduction in column $\sqsubseteq$. For each algorithm, column 'stored' denotes the number of stored states; column 'total' denotes the total number of generated states; column 'time' denotes the verification time in seconds. Symbol '-' means either the verification time is more than 2 hours or out-of-memory exception happens. Notice that our semi-algorithm terminates in all cases and all verification results are true. Comparing $stored$ and $total$, we can see that many states are skipped due to simulation reduction. From the verification time we can see that both simulation relations are helpful in reducing the state space. To the best of our knowledge, there is no existing tool supporting language inclusion checking of these models.

Table 2. Experiments on Random Timed Automata

$\|S\|$	$\|C\|$	$Dt = 0.6$	$Dt = 0.8$	$Dt = 1.0$	$Dt = 1.1$	$Dt = 1.3$
4	1	1.00\0.99\0.98	0.99\0.93\0.74	0.99\0.82\0.59	0.99\0.63\0.39	0.89\0.18\0.09
4	2	0.99\0.98\0.94	0.98\0.87\0.68	0.94\0.72\0.51	0.85\0.49\0.33	0.45\0.12\0.06
4	3	0.99\0.98\0.93	0.95\0.82\0.65	0.89\0.67\0.52	0.75\0.42\0.28	0.31\0.10\0.06
6	1	1.00\0.99\0.98	0.99\0.97\0.90	0.99\0.61\0.41	0.97\0.43\0.29	0.83\0.13\0.08
6	2	0.99\0.99\0.98	0.99\0.96\0.88	0.88\0.49\0.32	0.79\0.34\0.22	0.44\0.09\0.05
6	3	0.99\0.99\0.98	0.99\0.94\0.85	0.78\0.44\0.29	0.69\0.31\0.21	0.34\0.11\0.07
8	1	1.00\0.99\0.99	0.99\0.92\0.83	0.96\0.53\0.40	0.94\0.37\0.31	0.55\0.08\0.07
8	2	0.99\0.99\0.99	0.99\0.91\0.84	0.84\0.48\0.37	0.73\0.32\0.25	0.25\0.10\0.09
8	3	0.99\0.99\0.99	0.98\0.91\0.83	0.78\0.47\0.38	0.70\0.40\0.32	0.20\0.08\0.07

The last question is: how good are timed automata as a specification language? We consider a timed automaton specification is 'good' if given an implementation model, our semi-algorithm answers conclusively on the language inclusion problem. To answer this question, we extend the approach on generating non-deterministic finite automata in [20] to automatically generate random timed automata, and then apply our semi-algorithm for language inclusion checking. Without loss of generality, a generated timed automaton has always one initial state and the alphabet is $\{0, 1\}$. In addition, the following parameters are used to control the random generation process: the number of state $|S|$, the number of clocks $|C|$, a parameter Dt for transition density and a clock ceiling. For each event in the alphabet, we generate k transitions (and hence the transition density for the event is $Dt = k/|S|$) and distribute the transitions randomly among all $|S|$ states. For each transition, the clock constraint and the resetting clocks are generated randomly according to the clock ceiling. We remark that if both implementation and specification models are generated randomly, language inclusion almost always fails. Thus, in order to have cases where language inclusion does hold, we generate a group of implementation specification pairs by generating an implementation first, and then adding transitions to the implementation to get the specification.

The experimental results are shown in Table 2[4]. For each different combinations of $|S|$, $|C|$ and Dt, we compute three numbers shown in the form of $a \setminus b \setminus c$. a is the percentage of cases in which our semi-algorithm terminates; c is the percentage of the cases satisfying the boundedness condition (and therefore being determinizable [4]). The gap between a and c thus shows the effectiveness of our approach on timed automata which may be non-determinizable. In order to show the effectiveness of simulation reduction, b is the percentage of cases in which our semi-algorithm terminates without simulation reduction (and with maximum ceiling zone normalization). We generate 1000 random pairs to calculate each number. In all cases $a > b$ and $b > c$, e.g., a is much larger than

[4] Notice that there are cases where there is only one clock in the specification and yet our semi-algorithm is not terminating. This is because of using a set of DBMs to represent zones. That is, because there is no efficient procedure to check whether a zone z is a subset of another (which is represented as the union of multiple DBMs), the LU-simulation that we discover is partial and we may unnecessarily explore more states, infinitely more in some cases.

b and c when $Dt \geq 1.0$. This result implies that our semi-algorithm terminates even if the specification may not be 'determinizable', which we credit to simulation reduction and the fact that the semi-algorithm is on-the-fly (so that language inclusion checking can be done without complete determinization). When transition density increases, the gap between a and b increases (e.g., when $Dt \geq 1.0$, b is always much smaller that a), which evidences the effectiveness of our simulation reduction. In general, the lower the density is, the more likely it is that the semi-algorithm terminates. We calculate the transition density of the timed property patterns and the benchmark systems. We find that all the events have transition densities less than or equal to 1.0 except the absence pattern. Based on the results presented in Table 2, we conclude that in practice, our semi-algorithm has a high probability of terminating. This perhaps supports the view that timed automata could serve as a good specification language.

5 Related Work

The work in [2] is the first study on the language inclusion checking problem for timed automata. The work shows that timed automata are not closed under complement, which is an obstacle in automatically comparing the languages of two timed automata. Naturally, this conclusion leads to work on identifying determinizable subclasses of timed automata, with reduced expressiveness. Several subclasses of timed automata have been identified, i.e., event-clock timed automata [3,17], timed automata with integer resets [18] or with one clock [16] and strongly non-Zeno timed automata [4].

Our work is inspired by the work in [4] which presents an approach for deciding when a timed automaton is determinizable. The idea is to check whether the timed automaton satisfies a clock boundedness condition. The authors show that the condition is satisfied by event-clock timed automata, timed automata with integer resets and strongly non-Zeno timed automata. Using region construction, it is shown in [4] that an equivalent deterministic timed automaton can be constructed if the given timed automaton satisfies the boundedness condition. The work is closely related to [1], in which the authors proposed a zone-based approach for determinizing timed automata with one clock. Our work combines [1,4] and extends them with simulation reduction so as to provide an approach which could be useful for arbitrary timed automata in practice.

In addition, a game-based approach for determinizing timed automata has been proposed in [6,13]. This approach produces an equivalent deterministic timed automaton or a deterministic over-approximation, which allows one to enlarge the set of timed automata that can be automatically determinized compared to the one in [4]. In comparison, our approach could determinize timed automata which fail the boundedness condition in [4], and can cover the examples shown in [6]. The work is remotely related to work in [10]. In particular, it has been shown that under digitization with the definition of weakly monotonic timed words, whether the language of a closed timed automaton is included in the language of an open timed automaton is decidable [10].

6 Conclusion

In summary, the contributions of this work are threefold. First, we develop a zone-based approach for language inclusion checking of timed automata, which is further combined

with simulation reduction for better performance. Second, we investigate, both theoretical and empirically, when the semi-algorithm is terminating. Lastly, we implement the semi-algorithm in the PAT framework and apply it to benchmark systems. As far as the authors know, our implementation is the first tool which supports using arbitrary timed automata as a specification language. More importantly, with the proposed semi-algorithm and the empirical results, we would like to argue that timed automata do serve a specification language in practice. As for the future work, we would like to investigate the language inclusion checking problem with the assumption of non-Zenoness.

References

1. Abdulla, P.A., Ouaknine, J., Quaas, K., Worrell, J.B.: Zone-Based Universality Analysis for Single-Clock Timed Automata. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 98–112. Springer, Heidelberg (2007)
2. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theory of Computer Science 126(2), 183–235 (1994)
3. Alur, R., Fix, L., Henzinger, T.A.: Event-clock Automata: A Determinizable Class of Timed Automata. Theoretical Computer Science 211, 253–273 (1999)
4. Baier, C., Bertrand, N., Bouyer, P., Brihaye, T.: When Are Timed Automata Determinizable? In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part II. LNCS, vol. 5556, pp. 43–54. Springer, Heidelberg (2009)
5. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and Upper Bounds in Zone-based Abstractions of Timed Automata. International Journal on Software Tools for Technology Transfer 8(3), 204–215 (2004)
6. Bertrand, N., Stainer, A., Jéron, T., Krichen, M.: A Game Approach to Determinize Timed Automata. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 245–259. Springer, Heidelberg (2011)
7. Dill, D.L., Hu, A.J., Wong-Toi, H.: Checking for Language Inclusion Using Simulation Preorders. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 255–265. Springer, Heidelberg (1992)
8. Gruhn, V., Laue, R.: Patterns for Timed Property Specifications. Electronic Notes in Theoretical Computer Science 153(2), 117–133 (2006)
9. Henzinger, T.A., Kopke, P.W., Wong-Toi, H.: The Expressive Power of Clocks. In: Fülöp, Z. (ed.) ICALP 1995. LNCS, vol. 944, pp. 417–428. Springer, Heidelberg (1995)
10. Henzinger, T.A., Manna, Z., Pnueli, A.: What Good are Digital Clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
11. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic Model Checking for Real-time Systems. Journal of Information and Computation 111(2), 193–244 (1994)
12. Konrad, S., Cheng, B.H.C.: Real-time Specification Patterns. In: ICSE, pp. 372–381 (2005)
13. Krichen, M., Tripakis, S.: Conformance Testing for Real-Time Systems. Formal Methods in System Design 34(3), 238–304 (2009)
14. Larsen, K.G., Petterson, P., Wang, Y.: UPPAAL in a Nutshell. Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
15. Marcone, A.: Foundations of BQO Theory. Transactions of the American Mathematical Society 345(2), 641–660 (1994)
16. Ouaknine, J., Worrell, J.: On The Language Inclusion Problem for Timed Automata: Closing a Decidability Gap. In: LICS, pp. 54–63 (2004)
17. Raskin, J., Schobbens, P.: The Logic of Event Clocks - Decidability, Complexity and Expressiveness. Journal of Automata, Languages, and Combinatories 4(3), 247–286 (1999)

18. Suman, P.V., Pandya, P.K., Krishna, S.N., Manasa, L.: Timed Automata with Integer Resets: Language Inclusion and Expressiveness. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 78–92. Springer, Heidelberg (2008)
19. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
20. Tabakov, D., Vardi, M.Y.: Experimental Evaluation of Classical Automata Constructions. In: Sutcliffe, G., Voronkov, A. (eds.) LPAR 2005. LNCS (LNAI), vol. 3835, pp. 396–411. Springer, Heidelberg (2005)
21. Tripakis, S.: Verifying progress in timed systems. In: Katoen, J.-P. (ed.) ARTS 1999. LNCS, vol. 1601, pp. 299–314. Springer, Heidelberg (1999)
22. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: A New Algorithm for Checking Universality of Finite Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)
23. Yovine, S.: Kronos: a Verification Tool for Real-time Systems. Journal on Software Tools for Technology Transfer 1(1-2), 123–133 (1997)