Language Inclusion Checking of Timed Automata with Non-Zenoness

Xinyu Wang, Jun Sun, Ting Wang, and Shengchao Qin

Abstract—Given a timed automaton $\mathcal{P}$ modeling an implementation and a timed automaton $\mathcal{S}$ as a specification, the problem of language inclusion checking is to decide whether the language of $\mathcal{P}$ is a subset of that of $\mathcal{S}$. It is known to be undecidable. The problem gets more complicated if non-Zenoness is taken into consideration. A run is Zeno if it permits infinitely many actions within finite time. Otherwise it is non-Zeno. Zeno runs might present in both $\mathcal{P}$ and $\mathcal{S}$. It is necessary to check whether a run is Zeno or not so as to avoid presenting Zeno runs as counterexamples of language inclusion checking. In this work, we propose a zone-based semi-algorithm for language inclusion checking with non-Zenoness. It is further improved with simulation reduction based on LU-simulation. Though our approach is not guaranteed to terminate, we show that it does in many cases through empirical study. Our approach has been incorporated into the PAT model checker, and applied to multiple systems to show its usefulness.

Index Terms—Timed Automata, Language Inclusion, Non-Zenoness.

1 Introduction

Timed automata, introduced by Alur and Dill in [3], 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 through the construction of region graphs [3]. Efficient methods based on zone abstraction for checking both safety and liveness properties have later been developed [25], [34]. Zone abstraction, which constructs zone graphs, is an effective technique for model checking timed automata and it has been employed by many tools including the popular UPPAAL [25]. Verification tools for timed automata based models have proven to be successful [10], [13], [25], [37].

Nonetheless, researchers have also identified several issues associated with timed automata based system verification [3], [14]. One of them is that the language inclusion checking problem is undecidable. In order to avoid this undecidability, a number of determinizable subclasses of timed automata have been identified, e.g., event-clock timed automata [4], [28], and integer resets timed automata [31]. In addition, it has been shown [26] the problem of checking whether the language of a timed automaton is a subset of that of a timed automaton with a single clock is decidable. Another issue is related to the notion of Zeno runs. An infinite run is non-Zeno if and only if it takes an unbounded amount of time; otherwise it is Zeno. Zeno runs are infeasible in reality and thus must be pruned during system analysis. That is, it is necessary to check whether a run is Zeno or not so as to avoid presenting Zeno runs as counterexamples. In particular, liveness properties are usually meaningless unless non-Zenoness is assumed; and safety properties cannot be trusted since Zeno runs may conceal deadlocks, etc. Furthermore, the reason why non-Zenoness checking is particularly challenging is that zone graphs are too abstract to directly infer time progress and hence non-Zenoness. For instance, given an infinite path in the zone graph, it is infeasible to tell whether there are infinitely many transitions that bound some clock x from above, but only finitely many transitions that reset x and, thus, the total time elapsed is bounded. A number of approaches have been proposed to solve the Zenoness checking problem in [22], [23], [34], [35]. The state-of-the-art approach is based on constructing guessing zone graphes which enrich zone graphs with additional information for Zenoness checking [22], [23].

In this work, we investigate the language inclusion checking problem of timed automata with non-Zenoness. That is, we define the language of a timed (safety) automaton to be the set of finite timed words obtained from runs which can be extended to an infinite non-Zeno run. That is, we consider a finite timed word non-Zeno if it can only occur as a part of Zeno infinite runs. We develop a zone-based approach which, given a timed automaton $\mathcal{P}$ modeling an implementation and a timed automaton $\mathcal{S}$ as a specification, checks whether the language of $\mathcal{P}$ is a subset of that of $\mathcal{S}$. Language inclusion checking can be often converted to a reachability problem on the product of $\mathcal{P}$ and the determinization of $\mathcal{S}$, where a state of the product is composed of a state of $\mathcal{P}$ and a set of states from $\mathcal{S}$. In our approach, we develop a semi-algorithm which determinizes $\mathcal{S}$ and constructs the product on-the-fly, where zones are used as a symbolic representation. Due to the complication of Zeno runs, the states of the product are enriched with additional labels so that only a part of a non-Zeno run in $\mathcal{P}$ (which has no corresponding non-Zeno run in $\mathcal{S}$) is output as a counterexample. Furthermore, simulation reduction, based on simulation relation between the product states obtained through LU-simulation [6], is incorporated, which often contributes to the termination of our semi-algorithm.

Our approach has been implemented in the PAT model checker [32]. It can be applied to arbitrary timed automata,

- *Xinyu Wang is with College of Computer Science, Zhejiang University, P.R. China. E-mail: wangxinyu@zju.edu.cn*
- *The corresponding author Ting Wang is with College of Computer Science, Zhejiang University of Technology, P.R. China. E-mail: wangting@zjut.edu.cn*
- *Jun Sun is with ISTD, Singapore University of Technology and Design, Singapore. E-mail: sunjun@sutd.edu.sg*
- *Shengchao Qin is with Teesside University, UK. E-mail: S.Qin@tees.ac.uk*

though it may not always terminate. To show that our approach can be useful in practice, we investigate when our approach is terminating empirically. Firstly, we prove that, if $\mathcal{S}$ satisfies a clock boundedness condition defined in [5], our approach is guaranteed to terminate. Secondly, using a large set of randomly generated timed automata, we show that our approach often terminates. Lastly, we apply our approach to multiple benchmark systems to investigate its scalability.

Related Work This work is related to the line of work on language inclusion checking for timed automata. The work in [3] is the first study on the problem. It shows that timed automata are not closed under complement, which is an obstacle in automatically comparing the languages of two timed automata. This conclusion leads to work on identifying determinizable subclasses of timed automata, with reduced expressiveness. Several subclasses of timed automata have been identified, e.g., event-clock timed automata [4], [28], timed automata with integer resets [31] and strongly non-Zeno timed automata [5]. In addition, it has been shown [26] that although timed automata with one clock are not determinizable, the problem of checking whether the language of a timed automaton is a subset of that of a timed automaton with a single clock is decidable.

Our work is inspired by [5], in which the authors present 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 [5] that an equivalent deterministic timed automaton can be constructed if the given timed automaton satisfies the clock boundedness condition. The work is closely related to [2], in which the authors proposed a zone-based approach for determinizing timed automata with one clock. Our work combines [2], [5] 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 [9], [24]. 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 [5]. In comparison, our approach could determinize timed automata which fail the boundedness condition in [5], and can cover the examples shown in [9]. The work is remotely related to work in [20]. 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 [20].

This work is closely related to our previous work in [38], which proposes a semi-algorithm for language inclusion checking of timed automata without non-Zenoness. We adopt the idea of constructing an abstract product of the implementation and specification in this work and extend it to investigate language inclusion checking with non-Zenoness. Though some of the used techniques are similar, we target two different problems. To the best of our knowledge, the language inclusion checking problem with non-Zenoness for timed automata has been rarely addressed. In [27], the authors showed that the refinement checking problem of safety MTL, a real-time extension of linear temporal logic, is decidable. Their approach is to translate a safety MTL formula into a single-clock timed automaton based on the non-Zeno semantics. The refinement checking problem of safety MTL is then reduced to the language inclusion checking problem between two single-clock timed automata.

This work is related to the line of work on non-Zenoness checking. In [34], it has been shown that every run in a timed automaton is non-Zeno if for each structural loop of the timed automaton (i.e., a loop in the timed automaton itself, not the underlying transition system), there exists a clock c such that c is reset during the loop and c is bounded from below in a guard of a transition during the loop. A weaker condition is identified in [12] (e.g., instead of checking all structural loops, only some loops are checked). Given a network of timed automata, it implies that every run is non-Zeno if the product automaton satisfies the condition. Sufficient conditions which guarantee absence of Zeno runs in a network without constructing the product have been identified [12]. Effectively, the work in [12] weakens the requirements imposed in [34].

The analysis in [12] is able to assert absence of Zeno runs for a larger class of specifications, but it assumes a simple timed automaton model. In [18], the authors show that the analysis is not sound when UPPAAL extensions such as non-Zero clock assignments and broadcast channels are considered, and that synchronisation can be better exploited to improve precision. Besides, they extend the analysis to cases where urgent and committed locations, urgent channels, parameters and selections are used in the model.

However, preventing Zeno runs altogether by construction would be too restrictive for users [18]. Rather, methods should be provided to check whether a run is Zeno or not and discard the Zeno runs in the process of verification. In [34], [35], the authors showed that every timed automaton can be transformed into a strongly non-Zeno one, for which, the emptiness problem can be solved easily. The price to pay is an extra clock. It has been shown that adding one clock may result in an exponentially larger zone graph [23]. The proposed remedy is the guessing-zone graph approach. In addition, this work is related to the work on non-Zeno real-time game strategy [15], which however is not based on zone abstraction.

Organization The remainders of the article are organized as follows. Section 2 reviews the relevant background. Section 3 shows how to reduce the language inclusion checking problem to a reachability problem in the concrete semantics. Section 4 then shows how to reduce the language inclusion checking problem to a reachability problem with zone abstraction. Section 5 then presents details on how the language inclusion problem is solved. Section 6 reports the experimental results. Section 7 concludes.

2 PRELIMINARY

In this section, we review some relevant background on labeled transition systems (LTS), timed automata and define our problem.

2.1 Labeled Transition Systems

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. $\mathcal{L}$ is deterministic if and only if $(s, e, s') \in T$ and $(s, e, s'') \in T$ imply $s' = s''$. 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 $(s_i, e_{i+1}, s_{i+1}) \in T$ 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 if and only if 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 the above run, the sequence $tr = \langle e_1, e_2, \cdots, e_n \rangle$ is called a trace. We say that s_n is reachable from s_0 via trace tr.

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_0 is simulated by s_1 with respect to F and F is not reachable from s_1, F is also not reachable from s_0, and hence s_0 can be skipped during system exploration if s_1 has been explored already. This is known as simulation reduction [16].

2.2 Timed Automata

Timed automata were originally introduced as finite-state timed Büchi automata [3], i.e., finite automata equipped with real-valued clock variables and Büchi accepting condition. The Büchi accepting condition is used to enforce progress, i.e., a run is accepting if and only if it visits some accepting state infinitely often. Later, timed safety automata were introduced in [21] 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 expressiveness of timed safety automata is strictly less than that of timed Büchi automata [30]. In the following, we focus on timed safety automata as they are supported by the popular tools like UPPAAL [25] and are often used in practice. Hereafter, they are referred to as timed automata following common practice.

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 as follows: $\delta := true | x \sim n | \delta_1 \wedge \delta_2 | \neg \delta_1$ where $\sim \in \{=, \leq, \geq, <, >\}$; x is a clock in C and n is an integer constant[1]. The set of downward constraints obtained with $\sim \in \{\leq, <\}$ and without negation $\neg$ 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 valuation v satisfies a clock constraint δ, written as $v \in \delta$, if and only if δ evaluates to be true using the clock values given by v. A clock constraint can be viewed as the set of clock valuations which satisfy the constraint. For $d \in \mathbb{R}^+$, let $v + d$ denote the clock valuation v' such that $v'(c) = v(c) + d$ for all $c \in C$. For $X \subseteq C$, let clock resetting notion $[X := 0]v$ 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.

Definition 1: A timed automaton is a tuple $(S, Init, \Sigma, C, L, T)$ where S is a finite set of locations; $Init \subseteq S$ is a set of initial locations; 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. An example timed automaton is shown at the top of Fig. 1.

Definition 2: Let $\mathcal{A} = (S, Init, \Sigma, C, L, T)$ be a timed automaton. The concrete semantics of $\mathcal{A}$ is an infinite-state LTS $\mathcal{C}(\mathcal{A}) = (S_c, Init_c, \mathbb{R}^+ \cup \Sigma, T_c)$ such that S_c is a set of concrete states of $\mathcal{A}$, each of which is a pair (s, v) where $s \in S$ is a location and v is a clock valuation; $Init_c = \{(s, C = 0) | s \in Init\}$ is a set of initial concrete states; and T_c is the smallest transition relation satisfying the following conditions:

- $((s, v), t, (s, v + t)) \in T_c$ if $v + t \in L(s)$;
- or, $((s, v), e, (s', [X := 0]v)) \in T_c$ if there is a transition $(s, e, g, X, s') \in T$ such that $v \in g$ and $[X := 0]v \in L(s')$.

A timed automaton $\mathcal{A}$ is deterministic if and only if $\mathcal{C}(\mathcal{A})$ is deterministic. Otherwise, $\mathcal{A}$ is non-deterministic. A (finite or infinite) run of $\mathcal{C}(\mathcal{A})$ is of the form

$$\pi = \langle (s_0, v_0), x_1, (s_1, v_1), x_2, \cdots, (s_i, v_i), x_i, \cdots \rangle$$

where $((s_i, v_i), x_i, (s_{i+1}, v_{i+1})) \in T_c$ for all i. Note that x_i is either an event in Σ or a number in $\mathbb{R}^+$. The duration of the run is the accumulated delay: $\Sigma_{d \in \{x_i | x_i \in \mathbb{R}^+\}} d$. An infinite run is non-Zeno if its duration is unbounded. A finite run is *non-Zeno* if and only if it is a prefix of some *non-Zeno* infinite run. That is, a finite run is Zeno if it cannot be extended to an infinite non-Zeno run. Given the above run π, we can obtain a timed word: $\langle (D_1, e_1), (D_2, e_2), \cdots, (D_i, e_i), \cdots \rangle$ such that e_1 is the first event in the sequence $\langle x_1, x_2, \cdots \rangle$ and D_1 is the accumulated delay before e_1 occurs; and e_2 is the second event in the sequence and D_2 is the accumulated delay before e_2 occurs and so on. We define $\mathcal{L}(\mathcal{A}, (s, v))$ to be the set of finite timed words obtained from the set of all non-Zeno finite runs starting with (s, v). The language of $\mathcal{A}$, written as $\mathcal{L}(\mathcal{A})$, is defined as $\{x | \exists s \in Init.\ x \in \mathcal{L}(\mathcal{A}, (s, C = 0))\}$. Two timed automata are equivalent if they define the same language.

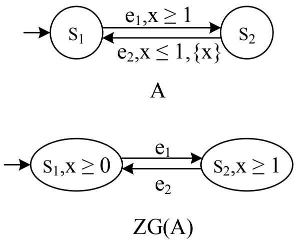
The language inclusion checking problem with non-Zenoness is the problem of checking whether $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{Q})$, given a timed automaton $\mathcal{P}$ and a timed automaton $\mathcal{Q}$. In the rest of the article, we fix two timed automata with the same alphabet $\mathcal{P} = (S_p, Init_p, \Sigma, C_p, L_p, T_p)$ and $\mathcal{Q} = (S_q, Init_q, \Sigma, C_q, L_q, T_q)$ such that S_p, S_q, C_p and C_q are pair-wise disjoint. We remark that in practice, $\mathcal{P}$ and $\mathcal{Q}$ are often composed of several timed automata executing in parallel, i.e., a network of timed automata. We skip the details on parallel composition of timed automata and remark our approach applies to networks of timed automata.

2.3 Zone Abstraction

Given a timed automaton $\mathcal{A}$, zone abstraction is a technique for building an abstraction of $\mathcal{C}(\mathcal{A})$ called a zone graph. Zone abstraction has been employed by many tools including UPPAAL [25]. A zone is the conjunction of multiple primitive constraints over a set of clocks. Technically speaking, a zone is the maximal set of clock valuations satisfying the constraint. A zone is empty if and only if the constraint is unsatisfiable. In the following, we use zones and clock constraints interchangeably as the latter is the syntactic representation of the former.

The following zone operations are relevant in this work. Given a zone δ, we use $\delta^{\uparrow}$ to denote the zone reached by delaying an

[1] We do not consider diagonal constraints in this work. Notice that due to the negation, a clock constraint may not be convex.



A

ZG(A)

Fig. 1. A sample zone graph

arbitrary amount of time from zone δ; we write $[X := 0]\delta$ to denote the zone obtained by resetting clocks in X to be 0. Given a set of clocks X, let $\delta[X]$ denote the projection of δ on X.

Definition 3: Let $\mathcal{A} = (S, Init, \Sigma, C, L, T)$ be a timed automaton. Its zone graph, denoted as $ZG(\mathcal{A})$, is an LTS $(S_z, Init_z, \Sigma, T_z)$ such that

- S_z is a set of nodes, each of which is a pair (s, δ) such that $s \in S$ is a location and δ is a zone;
- $Init_z = \{(init, (\bigwedge_{c \in C} c = 0)^{\uparrow} \wedge L(init)) | init \in Init\}$ is a set of initial nodes;
- T_z is the smallest labeled transition relation such that $((s_1, \delta_1), e, (s_2, \delta_2)) \in T_z$ if there exists a transition $(s_1, e, \delta, X, s_2) \in T$ and $\delta_1 \wedge \delta \neq false$ and $[X := 0](\delta_1 \wedge \delta) \wedge L(s_2) \neq false$ and $\delta_2 = ([X := 0](\delta_1 \wedge \delta) \wedge L(s_2))^{\uparrow} \wedge L(s_2)$.

An example zone graph is shown at the bottom of Fig. 1. The following establishes the relation between $ZG(\mathcal{A})$ and $\mathcal{C}(\mathcal{A})$.

Proposition 1 [39]: Let $\mathcal{A} = (S, Init, \Sigma, C, L, T)$ be a timed automaton and $ZG(\mathcal{A}) = (S_z, Init_z, \Sigma, T_z)$ be the zone graph. (s, v) is a reachable state of $\mathcal{C}(\mathcal{A})$ if and only if there exists a reachable state (s', δ) of $ZG(\mathcal{A})$ such that $s = s'$ and $v \in \delta$. □

The above proposition can be proved by an induction, i.e., it is true for any initial state of $\mathcal{C}(\mathcal{A})$ and it is preserved through every transition in $\mathcal{C}(\mathcal{A})$. It implies that zone abstraction preserves certain reachability properties (e.g., whether certain location is reachable or not). Unfortunately, zone abstraction is too abstract so that it is impossible to decide if a path in the zone graph $ZG(\mathcal{A})$ corresponds to a non-Zeno run of $\mathcal{A}$ [36]. The state-of-the-art approach for solving this problem is to construct a guessing zone graph, which enriches $ZG(\mathcal{A})$ with additional information for checking non-Zenoness [23]. We present the details of guessing zone graphes when they are relevant in Section 5.1.

We remark that the number of zones in $ZG(\mathcal{A})$ is in general infinite. A number of normalization operators have been defined [11], [29] to normalize the zones so that that the number of zones is finite. For instance, the idea of maximum ceiling normalization [29] is to transform zones that may contain arbitrarily large constants to a unique representation of a class of zones whose constants are bounded by certain fixed constant, e.g., the maximum clock ceiling in $\mathcal{A}$. In the following, we assume that zone normalization is not applied unless it is stated otherwise.

3 Checking with Concrete Semantics

In the following, we define an approach to solve the language inclusion checking problem in the concrete semantics. Note that it is only aimed to help understanding the approach in the next section as obviously the concrete semantics is too big an LTS and cannot be constructed explicitly.

The problem of checking whether $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{Q})$ can be converted to a reachability problem on the product of $\mathcal{C}(\mathcal{P})$ and the determinization of $\mathcal{C}(\mathcal{Q})$. In the following, we formally define these two operations: determinization and computing the product.

Definition 4: Given an LTS $\mathcal{L} = (S, Init, \Sigma, T)$, the determinization of $\mathcal{L}$ is an LTS $det(\mathcal{L}) = (S', \{Init\}, \Sigma, T')$ such that

- $S' = \mathbb{P}S$ is the set of all sets of states in S;
- T' is the smallest relation such that $(X, e, Y) \in T'$ if $Y = \{y \in S | \exists x \in X. \ (x, e, y) \in T\}$.

By definition, $det(\mathcal{L})$ is deterministic. It is easy to show that $\mathcal{L}$ and $det(\mathcal{L})$ have the same set of traces. We remark that $det(\mathcal{L})$ could have exponentially more states than $\mathcal{L}$.

Definition 5: Given two LTSs $\mathcal{L}_i = (S_i, Init_i, \Sigma_i, T_i)$ where $i \in \{1, 2\}$, the product of $\mathcal{L}_1$ and $\mathcal{L}_2$, denoted as $\mathcal{L}_1 \otimes \mathcal{L}_2$, is an LTS $(S, Init, \Sigma, T)$ such that $S = S_1 \times S_2$; $Init = Init_1 \times Init_2$; $\Sigma = \Sigma_1 \cup \Sigma_2$; and T is the smallest transition relation such that

- $((s_1, s_2), e, (s_1', s_2)) \in T$ if $(s_1, e, s_1') \in T_1$ and $e \notin \Sigma_2$;
- $((s_1, s_2), e, (s_1, s_2')) \in T$ if $(s_2, e, s_2') \in T_2$ and $e \notin \Sigma_1$;
- $((s_1, s_2), e, (s_1', s_2')) \in T$ if $(s_1, e, s_1') \in T_1$ and $(s_2, e, s_2') \in T_2$;

Next, we show how to check language inclusion with non-Zenoness based on $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$. Note that a state in $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$ is of the form (s, X) where s is a state in $\mathcal{C}(\mathcal{P})$ and X is a set of states in $\mathcal{C}(\mathcal{Q})$. Intuitively, s and X can be reached via the same timed word respectively in $\mathcal{C}(\mathcal{P})$ and $\mathcal{C}(\mathcal{Q})$. Without the assumption of non-Zenoness, it is sufficient to check whether there is a reachable state (s, X) in the product such that X is an empty set (i.e., there is a timed word which is possible in $\mathcal{P}$ but not in $\mathcal{Q}$). With the assumption of non-Zenoness, we need to check whether there is a non-Zeno run from s in $\mathcal{P}$ and there is a non-Zeno run from any state in X in $\mathcal{Q}$. For simplicity, we say that s is non-Zeno if and only if there is a non-Zeno run starting with s; and a set of states is non-Zeno if and only if it contains a non-Zeno state. Notice that by this definition, an empty set of states is Zeno.

Given a state (s, X) in $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$, we need to distinguish three cases in order to check language inclusion with non-Zenoness.

- Case 1: s is non-Zeno and X is non-Zeno, i.e., there exists a non-Zeno run starting with s in $\mathcal{P}$ and and a non-Zeno run starting with X in $\mathcal{Q}$. In this case, the language inclusion is not violated at this state.
- Case 2: s is non-Zeno and X is Zeno, i.e., there exists a non-Zeno run starting with s in $\mathcal{P}$ and there is no non-Zeno run starting with any state in X in $\mathcal{Q}$. Thus, the trace reaching s is a trace of $\mathcal{C}(\mathcal{P})$ but not a trace of $\mathcal{C}(\mathcal{Q})$ and the language inclusion may be violated.

- Case 3: s is Zeno, i.e., there is no non-Zeno run starting with s in $\mathcal{P}$. Thus, the runs starting with s are no longer relevant for the language inclusion checking.

Theorem 1: $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{Q})$ if and only if there is no reachable state (s, X) in $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$ with an incoming edge labeled with an event $e \in \Sigma$ and s is non-Zeno and X is Zeno.

Proof: By induction, we can show that: (1) for all (s, X) in $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$, if s is reachable from an initial state in $\mathcal{C}(\mathcal{P})$ through trace tr, X is the set of all states reachable through tr from any initial state in $\mathcal{C}(\mathcal{Q})$; and (2) if tr ends with an event $e \in \Sigma$, X is the set of all states reachable through a trace tr' such that tr and tr' result in the same timed word. Next, we prove the theorem in both directions by contradiction.
(if) We first show that if $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{Q})$ does not hold, there exists (s, X) which satisfies the condition. By definition, if $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{Q})$ does not hold, there is a timed word tw in $\mathcal{L}(\mathcal{P})$ but not in $\mathcal{L}(\mathcal{Q})$. By definition, there must be a trace tr of $\mathcal{C}(\mathcal{P})$ from which we can obtain tw. Assume that after trace tr, we reach state (s, X) in $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$. By definition, s must be non-Zeno. By (2) above, X contains all states reachable through a trace tr' such that tr and tr' result in tw. By assumption, tw is not in $\mathcal{L}(\mathcal{Q})$, therefore X must be Zeno.
(only if) We show that if there is a reachable configuration (s, X) satisfying the condition, $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{Q})$ does not hold. By assumption, (s, X) must be reachable via a trace tr ending with an event $e \in \Sigma$. Since s is non-Zeno, the timed word obtained from tr is in $\mathcal{L}(\mathcal{P})$. Since X is Zeno, the timed word obtained from tr is not in $\mathcal{L}(\mathcal{Q})$ by (2). Thus $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{Q})$ does not hold. □

In the following, we refer to a state (s, X) in $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$ with an incoming edge labeled with an event $e \in \Sigma$ where s is non-Zeno and X is Zeno as *a witness state*, as it is a witness of violation of language inclusion, by the above theorem. The problem of language inclusion checking is thus reduced to the problem checking whether a witness state is reachable.

4 Language Inclusion Checking with Zones

In this section, we present how to construct an abstraction of $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$ based on zone abstraction, based on which we solve the language inclusion checking problem. As discussed above, language inclusion checking requires constructing the product of $\mathcal{P}$ and the determinization of $\mathcal{Q}$. Determinizing timed automata in general is undecidable [3]. In the following, we describe an approach which can be applied to arbitrary timed automata. Though it is not guaranteed to terminate, we show that it is useful in the setting of language inclusion checking. The approach is presented in two parts in the following: 1) unfolding the original timed automaton into an infinite timed tree; and 2) building the product of $\mathcal{P}$ and the determinization of the timed tree generated from $\mathcal{Q}$ based on zones.

4.1 Removing State Invariant

In order to simplify the presentation in the remaining of the section, we first transform $\mathcal{P}$ and $\mathcal{Q}$ to a form without state invariants. The idea is to move the state invariants to transition guards. Given a timed automaton $\mathcal{A}$ and any state s with state invariant $L(s)$, we construct a timed automaton $\mathcal{A}'$ as follows. Firstly, any outgoing transition (s, e, δ, X, s') from s is changed to $(s, e, \delta \wedge L(s), X, s')$. Secondly, for any incoming transition (s', e, δ, X, s) of 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. 2(a), we construct the one in Fig. 2(b). The state invariant $x \leq 3$ of state s_3 is added to the transition from s_2 to s_3 and the transition from s_3 to s_1. By a simple induction, it can be shown that the set of finite timed words that can be obtained from $\mathcal{A}'$ is the same as that of $\mathcal{A}$.

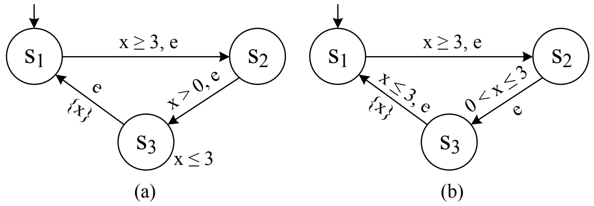


Fig. 2. Timed automata examples

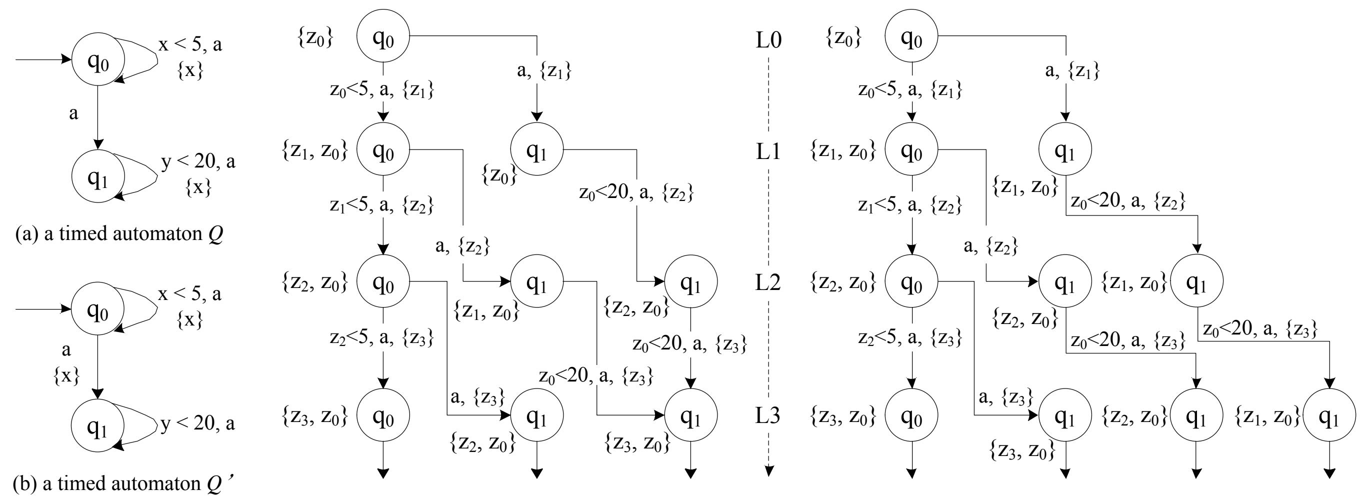
Notice that because $\mathcal{A}'$ has no state invariants (i.e., time can always elapse unboundedly at any state), every run of $\mathcal{A}'$ is non-Zeno by definition. For instance, for the example shown in Fig. 2, after the transformation, the timed automaton can idle at state s_3 forever. As a result, this transformation does not preserve $\mathcal{L}(\mathcal{A})$, i.e., all Zeno runs in $\mathcal{A}$ become non-Zeno in $\mathcal{A}'$ and a timed word which is obtained only from Zeno runs in $\mathcal{A}$ is in $\mathcal{L}(\mathcal{A}')$ but not in $\mathcal{L}(\mathcal{A})$. This is not an issue as non-Zenoness is always checked based on the original timed automata, as we show in Section 5. In the remainder of the section, we assume that all timed automata are without state invariants unless otherwise stated.

4.2 Unfolding

The idea of unfolding the original timed automaton into an infinite timed tree is adopted from [5]. The reason for the unfolding is that the infinite timed tree has the input-determinacy property, with which we can determinize this tree during the construction of the product.

In the following, we first show how to construct an unfolding of $\mathcal{Q}$ which is equivalent to $\mathcal{Q}$ through an example. Given the timed automaton $\mathcal{Q}$ in Fig. 3(a), Fig. 3(c) shows the infinite timed tree after unfolding $\mathcal{Q}$. A fresh clock is introduced at every level and used to replace the original clocks, i.e., x and y are replaced by clocks from a set $Z = \langle z_0, z_1, z_2, \cdots \rangle$. At level 0, we are at state q_0 and introduce a clock z_0. Since clock x and clock y start at the same time as z_0, we can use z_0 to replace x and y in the transition guard from q_0 at level 0 to the nodes at level 1. Because at level 0, the reading of clock z_0 is relevant to the future system behavior (i.e., there is a transition guard whose truth value depends on z_0), we say that z_0 is active at this node. In the tree, we label every node with a pair (q, A) where q is a location and A is a set of active clocks.

Two transitions from the level 0 node leads to the node of level 1, corresponding to the transitions from location q_0 to location q_0 and q_1 in Fig. 3(a). The clock constraint $x < 5$ is rewritten to $z_0 < 5$ using the active clock z_0 from the source node. A fresh clock z_1 is introduced along the transitions. Notice that the node with location q_0 at level 1 is labelled with a set of two active clocks. z_1 is active at location q_0 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



(a) a timed automaton Q

(b) a timed automaton Q'

(c) unfolding Q

(d) unfolding Q'

Fig. 3. Unfolding timed automata into infinite timed trees

along the transition. The set of active clocks of the node with location q_1 at level 1 is a singleton z_0 since both of the clocks x and y are not reset along the transition. Here z_1 is not active as its reading is irrelevant to future transitions from q_1. Following the same construction, we build the tree level by level.

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

- For any $q \in Init_q$, there is a level-0 node $n = (q, \{z_0\})$ in $Init_\infty$ with $level(n) = 0$, $f_n(c) = z_0$ for all $c \in C_q$.
- For each node $n = (q, A)$ at level i and for each transition $(q, e, \delta, X, q') \in T_q$, we add a node $n' = (q', A')$ at level $i+1$ where A' is the minimum set satisfying the following conditions: (1) if $c \in C_q \setminus X$, then $f_{n'}(c) = f_n(c)$ and $f_{n'}(c)$ is contained in A'; (2) if $c \in X$, $f_{n'}(c) = z_{i+1}$ and z_{i+1} is contained in A'; (3) $level(n') = i+1$. Afterwards, 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 exactly one clock. Given a node $n = (q, A)$ in the tree, observe that not every clock x in A is active. Hereafter, we assume that inactive clocks are always removed. The infinite timed trees constructed this way may or may not be finitely-branching. For instance, the tree in Fig. 3(c) is finitely-branching, as there are at most 4 branches from one level to the next. Furthermore, the number of clocks at each level is bounded by 3. Another example is shown in Fig. 3(d), which is obtained by unfolding the timed automaton shown in Fig. 3(b). More and more branches are generated during the level-by-level construction. As a result, the number of clocks becomes unbounded. These observations are closely linked to the termination of the language inclusion checking algorithm which is introduced later.

4.3 Constructing the Abstract Product

Following the result in [5], it can be shown that the set of finite timed words obtained from $\mathcal{Q}$ and from $\mathcal{Q}_\infty$ are equivalent. In the following, we define a zone graph for the product of $\mathcal{P}$ and determinization of $\mathcal{Q}_\infty$, referred to as the abstract product, based on which we solve the language inclusion checking problem.

The abstract product is an LTS $\mathcal{Z}_\infty = (S, Init, \Sigma, T)$. A state in $\mathcal{Z}_\infty$ is of the form (s_p, X_q, δ) such that $s_p \in S_p$; X_q is a set of nodes in $\mathcal{Q}_\infty$; and δ is a clock constraint constituted by clocks in C_p and $\mathcal{Q}_\infty$. Recall that a node of $\mathcal{S}_\infty$ is of the form (s_q, A) where $s_q \in S_q$ and A is a set of active clocks. Given a set of nodes X_q of $\mathcal{S}_\infty$, we write $Act(X_q)$ to denote the set of all active clocks, i.e., $\{c | \exists (s_q, A) \in X_q.\ c \in A\}$. δ constraints all clocks in $Act(X_q)$. We remark that a state in $\mathcal{Z}_\infty$ encodes a set of states in $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$. The set of initial states $Init$ is defined as the following: $\{(s_p, Init_\infty, (C_p \cup Act(Init_\infty) = 0)^\uparrow) | s_p \in Init_p\}$.

Next, we define T by showing how to generate successors of a given abstract configuration (s_p, X_q, δ), which is illustrated in Fig. 4. For every event $e \in \Sigma$, let $T_\infty(e, X_q)$ be the set of transitions in T_∞ which start with a state in X_q and are labeled with event e. Notice that the guard conditions of transitions in $T_\infty(e, X_q)$ may not be mutually exclusive. We define a set of constraints $Cons(e, X_q)$ such that each element in $Cons(e, X_q)$ is a constraint which conjuncts, for each transition in $T_\infty(e, X_q)$, either the transition guard or its negation. Notice that elements in $Cons(e, X_q)$ are by definition mutually exclusive. Given (s_p, X_q, δ) and a transition (s_p, e, g_p, X_p, s_p') in $\mathcal{P}$, for each $g \in Cons(e, X_q)$ we generate a successor (s_p', X_q', δ') such that the following conditions are satisfied.

- For any state $(s_q, A) \in X_q$ and any transition $((s_q, A), e, g_q, Y_q, (s_q', A')) \in T_\infty(e, X_q)$, $(s_q', A') \in X_q'$ if and only if $\delta \wedge g_p \wedge g \wedge g_q$ is not false.
- All states in X_q are at the same level and thus all transitions in $T_\infty(e, X_q)$ have the same resetting clock. Let y be that clock and $\delta' = ([\{y\} \cup X_p := 0](\delta \wedge g \wedge g_p))^\uparrow$.

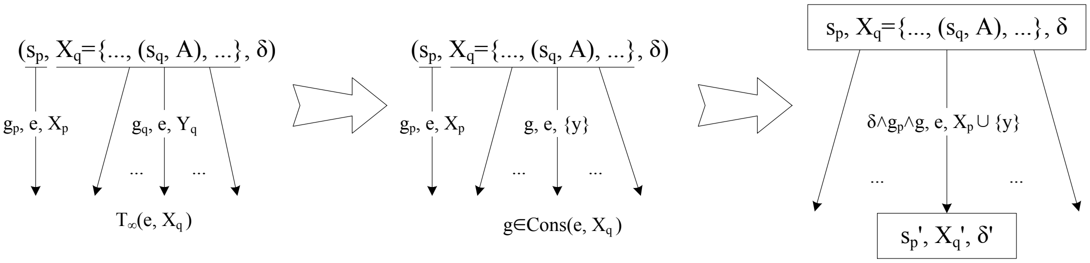

(s_p, X_q={..., (s_q, A), ...}, δ)
g_p, e, X_p
g_q, e, Y_q
... ...
T_∞(e, X_q)
(s_p, X_q={..., (s_q, A), ...}, δ)
g_p, e, X_p
g, e, {y}
... ...
g∈Cons(e, X_q)
s_p, X_q={..., (s_q, A), ...}, δ
δ∧g_p∧g, e, X_p∪ {y}
... ...
s_p', X_q', δ'

Fig. 4. Generating successors

- The transition from (s_p, X_q, δ) to (s'_p, X'_q, δ') is labeled with e.

We illustrate the above construction using the example in Fig. 5, where Fig. 5(a) is $\mathcal{P}$ and Fig. 5(b) is the infinite timed tree $\mathcal{Q}_\infty$. The only initial state of the abstract product is $(p_0, \{(q_0, \{z_0\})\}, 0 \leq x = z_0)$, as shown at the top of Fig. 5(c). As shown in Fig. 5(b), there are two transitions from node $(q_0, \{z_0\})$ labeled with event a, which are contained in $T_\infty(a, \{(q_0, \{z_0\})\})$. One of the transitions has the guard $z_0 < 5$ and the other one has the guard 'true'. The set $Cons(a, X_q)$ contains the following constraints: $z_0 < 5$ and $z_0 \geq 5$. Taking the transition form p_0 to p_0 in $\mathcal{P}$, we generate two potential successors for each of the constraints in $Cons(a, X_q)$, as shown above. Similarly, we can generate other states level by level. Note that for readability, we additionally label the edges in Fig. 5(c) with the corresponding guard condition and the resetting clocks.

The abstract product graph is always infinite-state as there are infinitely many clocks. In the following, we show how to reduce the number of states by reducing the number of clocks, which is inspired by [5]. Intuitively, given any abstract state (s_p, X_q, δ) in $\mathcal{Z}_\infty$, instead of always using a new clock in Z, we can reuse a clock which is not currently active, or equivalently not in $Act(X_q)$. We denote the zone graph after renaming as $\mathcal{Z}_r$. It is easy to show that $\mathcal{Z}_r$ and $\mathcal{Z}_\infty$ are equivalent [5].

The result of renaming Fig. 5(c) is shown in Fig. 6. For instance, given the configuration on the left of level 3 in Fig. 5(c), there are three active clocks z_0, z_2 and z_3. We can reuse z_1 and systematically rename z_3 to z_1 since z_1 is not active. Notice that after renaming, the number of clocks may still be infinite, e.g., the tree as shown in Fig. 3(d).

5 Language Inclusion with Non-Zenoness

In the following, we first show how to check whether the set of finite timed words of $\mathcal{P}$ is a subset of that of $\mathcal{Q}$ based on $\mathcal{Z}_r$ and then show how to solve the language inclusion checking problem with non-Zenoness.

5.1 Checking Language Inclusion with Non-Zenoness

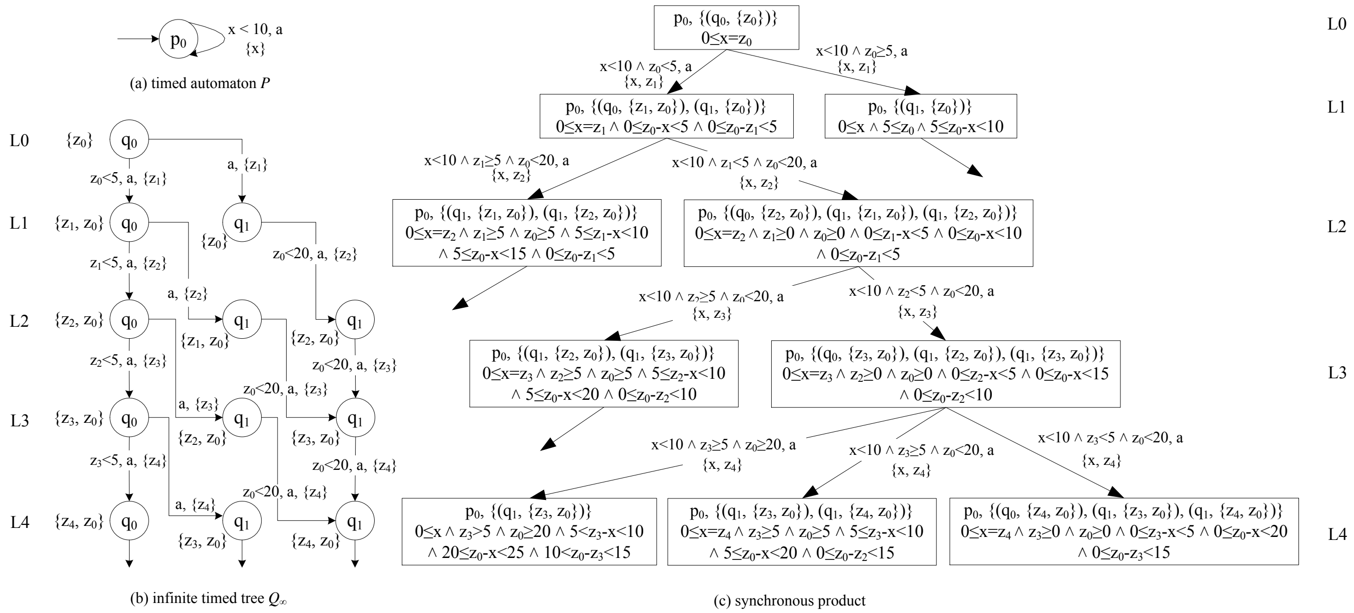
Given a state (s_p, X_q, δ) in $\mathcal{Z}_r$, we need to check that for any timed event that s_p can perform, whether X_q can perform the same timed event. Notice that one of the constraints in $Cons(e, X_q)$ conjuncts the negations of all guards of transitions in $T_\infty(e, X_q)$. Let us denote the constraint as neg. Given neg, assume the corresponding successor is (s'_p, X'_q, δ'). It is easy to see that X'_q must be empty. If δ' is not false, intuitively there exists a time point such that $\mathcal{P}$ can perform e whereas $\mathcal{Q}$ cannot. This is illustrated by the example shown in Fig. 6. Given the left-most abstract configuration of level 4 in the figure (named pq), the constraint neg in $Cons(a, X_q)$ is: $z_0 \geq 20$. Conjuncted with the guard condition $x < 10$ and the constraint in node pq, it is not false and there is a successor generated for neg as the node pq' on the left-bottom of the figure. Note that the X_q at this node is empty. As a result, the problem of checking whether the set of timed words of $\mathcal{P}$ is a subset of that of $\mathcal{Q}$ is reduced to checking whether $\mathcal{Z}_r$ contains a state (s_p, X_q, δ) where X_q is empty.

Language inclusion checking with non-Zenoness is more complicated because even if X_q is not empty, it might be that there are only Zeno runs from X_q in $\mathcal{Q}$ with the clock valuation satisfying δ. Recall that a state (s_p, X_q, δ) in $\mathcal{Z}_r$ encodes a set of states of $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$ (modula clock renaming). By Theorem 1, in order to check language inclusion, given a state (s_p, X_q, δ) in $\mathcal{Z}_r$, we need to check if the states encoded as (s_p, X_q, δ) include a witness state. In the following, we design a function $witness(s_p, X_q, \delta, \mathcal{P}, \mathcal{Q})$ which returns true if and only if the states encoded as (s_p, X_q, δ) include a witness state. For instance, given the state at the right of level 1 in Fig. 6, function $witness$ would return true since the set of states of $\mathcal{P}$ is non-Zeno, whereas the one of $\mathcal{Q}$ is Zeno because clock y is bounded from above but never reset afterwards (z_0 represents clocks x and y with $x = y$). As a result, we can immediately conclude that the language inclusion with non-Zenoness is violated.

In order to realize function $witness(s_p, X_q, \delta, \mathcal{P}, \mathcal{Q})$, we need to solve the underlying problem, i.e., given any timed automaton $\mathcal{A}$ and a set of states of $\mathcal{C}(\mathcal{A})$, how do we check whether there is non-Zeno run from one of the states in the set. We solve the problem through constructing a guessing zone graph [23] based on the original automaton $\mathcal{A}$ without removing the state invariants.

Definition 6: Let $\mathcal{A} = (S, Init, \Sigma, C, L, T)$ be a timed automaton and (s_0, δ_0) be an abstract state where $s_0 \in S$ and δ_0 is a clock constraint. The guessing zone graph with respect to (s_0, δ_0), denoted as $GZG(\mathcal{A}, s_0, \delta_0)$, is an LTS $(S_g, Init_g, \Sigma \times \Phi(C) \times 2^C, T_g)$ such that a state in S_g is of the form (s', δ', Y) where (s', δ') is an abstract state and $Y \subseteq C$; $Init_g = \{(s_0, \delta_0, C)\}$; and T_g is the smallest transition relation satisfying the following conditions.

- $((s_1, \delta_1, Y), e, \delta, X, (s_2, \delta_2, Y \cup X)) \in T_g$ if $t = (s_1, e, \delta, X, s_2)$ is a transition in $\mathcal{A}$ and there is a transition $((s_1, \delta_1), e, (s_2, \delta_2))$ in $ZG(\mathcal{A})$ and there are clock valuations $v \in \delta_1$, $v' \in \delta_2$ and $d \in \mathbb{R}^+$ such that $v + d \in \bigwedge_{x \in C - Y} x > 0$ and $v + d \in \delta$ and $[X := 0](v + d) = v'$.
- $((s, \delta, Y), \tau, true, \emptyset, (s, \delta, Y')) \in T_g$ where $Y' = \emptyset$ or $Y' = Y$ where τ is a special invisible event.



(a) timed automaton *P*

(b) infinite timed tree Q_∞

(c) synchronous product

Fig. 5. An example of product

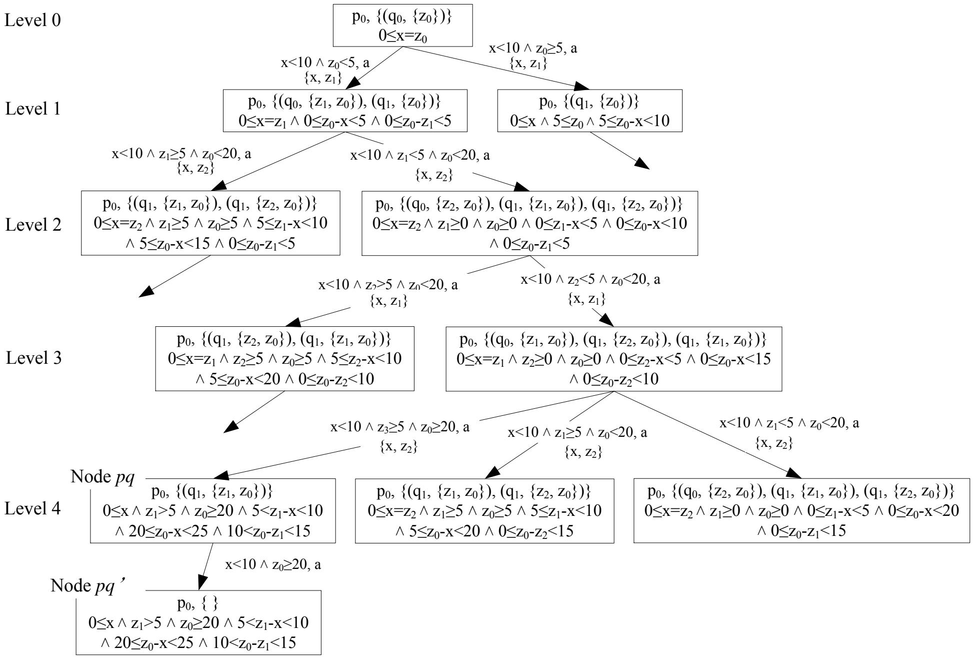

Level 0
p0, {(q0, {z0})}
0≤x=z0
x<10 ∧ z0<5, a
{x, z1}
x<10 ∧ z0≥5, a
{x, z1}
Level 1
p0, {(q0, {z1, z0}), (q1, {z0})}
0≤x=z1 ∧ 0≤z0-x<5 ∧ 0≤z0-z1<5
p0, {(q1, {z0})}
0≤x ∧ 5≤z0 ∧ 5≤z0-x<10
x<10 ∧ z1≥5 ∧ z0<20, a
{x, z2}
x<10 ∧ z1<5 ∧ z0<20, a
{x, z2}
Level 2
p0, {(q1, {z1, z0}), (q1, {z2, z0})}
0≤x=z2 ∧ z1≥5 ∧ z0≥5 ∧ 5≤z1-x<10
∧ 5≤z0-x<15 ∧ 0≤z0-z1<5
p0, {(q0, {z2, z0}), (q1, {z1, z0}), (q1, {z2, z0})}
0≤x=z2 ∧ z1≥0 ∧ z0≥0 ∧ 0≤z1-x<5 ∧ 0≤z0-x<10
∧ 0≤z0-z1<5
x<10 ∧ z2≥5 ∧ z0<20, a
{x, z1}
x<10 ∧ z2<5 ∧ z0<20, a
{x, z1}
Level 3
p0, {(q1, {z2, z0}), (q1, {z1, z0})}
0≤x=z1 ∧ z2≥5 ∧ z0≥5 ∧ 5≤z2-x<10
∧ 5≤z0-x<20 ∧ 0≤z0-z2<10
p0, {(q0, {z1, z0}), (q1, {z2, z0}), (q1, {z1, z0})}
0≤x=z1 ∧ z2≥0 ∧ z0≥0 ∧ 0≤z2-x<5 ∧ 0≤z0-x<15
∧ 0≤z0-z2<10
x<10 ∧ z3≥5 ∧ z0≥20, a
{x, z2}
x<10 ∧ z1≥5 ∧ z0<20, a
{x, z2}
x<10 ∧ z1<5 ∧ z0<20, a
{x, z2}
Node pq
Level 4
p0, {(q1, {z1, z0})}
0≤x ∧ z1>5 ∧ z0≥20 ∧ 5<z1-x<10
∧ 20≤z0-x<25 ∧ 10<z0-z1<15
p0, {(q1, {z1, z0}), (q1, {z2, z0})}
0≤x=z2 ∧ z1≥5 ∧ z0≥5 ∧ 5≤z1-x<10
∧ 5≤z0-x<20 ∧ 0≤z0-z2<15
p0, {(q0, {z2, z0}), (q1, {z1, z0}), (q1, {z2, z0})}
0≤x=z2 ∧ z1≥0 ∧ z0≥0 ∧ 0≤z1-x<5 ∧ 0≤z0-x<20
∧ 0≤z0-z1<15
x<10 ∧ z0≥20, a
Node pq'
p0, { }
0≤x ∧ z1>5 ∧ z0≥20 ∧ 5<z1-x<10
∧ 20≤z0-x<25 ∧ 10<z0-z1<15

Fig. 6. From language inclusion checking to reachability analysis

Compared to zone graphs, guessing zone graphes contain extra states and transition labels which are designed for detecting Zenoness. Intuitively, there are two reasons why a path in the zone graph disallows time from elapsing unboundedly. Either the path has infinitely many transitions that bound some clock x from above, but only finitely many transitions that reset x and, thus, the during of any run following the path is bounded. Or, the path contains transitions that reset x, and always subsequently transitions that requires $x = 0$ and thus time cannot elapse at all. The transitions which require $x = 0$ are called zero-checks. Guessing zone graphs are designed to handle these two cases explicitly. In particular, the Y component of a node (s, δ, Y) allows us to infer that the clocks not in Y are strictly positive. A node of the form $(s, \delta, \emptyset)$ is called *clear node*, from which every reachable zero-check is preceded by the reset of the clock that is checked, and hence nothing prevents time from elapsing in this node. An infinite path of the guessing zone graph is non-Zeno if all clocks bounded from above are reset infinitely often and the path contains a clear node [23]. A path is *blocked* if there is a clock that is bounded from above infinitely often and reset only finitely often by the transitions on the path. Otherwise the path is called *unblocked*.

Given the automaton A shown in Fig. 1, Fig. 7 shows the zone graph $ZG(A)$ along with the guessing zone graph $GZG(A, s_1, x \geq 0)$, where self-loop τ-transitions are removed for readability. An unblock path containing a clear node, i.e., a

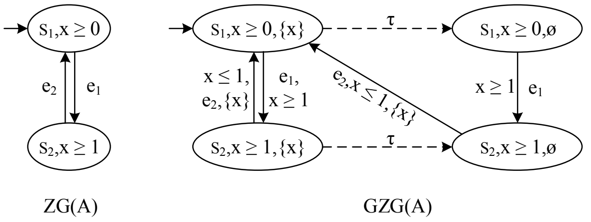


Fig. 7. A guessing zone graph with self-loop τ-transitions removed

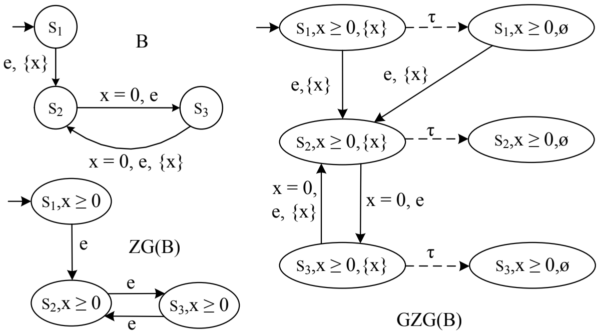


Fig. 8. A guessing zone graph with zero checks

non-Zeno path, can be found in this guessing zone graph. Fig. 8 shows a timed automaton with zero-checks, together with the corresponding zone graph and guessing zone graph (with self-loop τ-transitions removed). Obviously, there are no non-Zeno runs in automaton B. This is reflected in the guessing zone graph, where the infinite path in the graph (containing the two states labeled with $S_2, x \geq 0, \{x\}$ and $S_3, x \geq 0, \{x\}$ respectively) does not contain a clear node.

Theorem 2: A node in $GZG(\mathcal{A}, s, \delta)$ is non-Zeno if and only if it can reach a strongly connected component (SCC) in $GZG(\mathcal{A}, s, \delta)$ which contains a clear node and a path which visits every node and edge in the SCC is unblocking.

Proof: The proof follows the proof of Theorem 2 in [23].

Intuitively, if a node in $GZG(\mathcal{A}, s, \delta)$ can reach such an SCC, there must be an infinite run starting with the node such that neither reasons why the run disallows time from elapsing unboundedly is possible. Thus, function $witness(s_p, X_q, \delta, \mathcal{P}, \mathcal{Q})$ can be realized with the following steps. Firstly, we extract the set of states of the implementation as $(s_p, \delta[C_p])$ and build $GZG(\mathcal{P}, s_p, \delta[C_p])$. Secondly, by analyzing $GZG(\mathcal{P}, s_p, \delta[C_p])$ we answer whether $(s_p, \delta[C_p])$ is non-Zeno. Similarly, we answer whether $(s_q, \delta[A])$ is non-Zeno for every $(s_q, A) \in X_q$. If $(s_p, \delta[C_p])$ is non-Zeno and all $(s_q, \delta[A])$ are Zeno, we return true. The above method is computationally expensive as the worst case complexity of constructing the guessing zone graph and checking whether it contains one such SCC is $|ZG| \cdot (|C| + 1)^2$ where $|ZG|$ is the size of the zone graph and $|C|$ is the number of clocks. In practice, the algorithm can be improved by constructing only a small part of the guessing zone graph [23].

Furthermore, we can improve the above approach for implementing function $witness$ by caching the states which have been identified as Zeno or non-Zeno. To see how it works, let us first implement a function $nz(s, \delta, \mathcal{A}, NZSet, ZSet)$ which takes an abstract state (s, δ) and the corresponding timed automaton $\mathcal{A}$ as inputs and returns true if and only if (s, δ) is non-Zeno. The implementation of the function is shown as Algorithm 1. We maintain two variables $NZSet$ and $ZSet$ which are sets of abstract states which are known to be non-Zeno and Zeno respectively. Given a new abstract state (s, δ), if there is a non-Zeno state (s, δ') in $NZSet$ such that $\delta' \subseteq \delta$, (s, δ) is non-Zeno. Similarly, if there is a state (s, δ') in $ZSet$ such that $\delta \subseteq \delta'$, (s, δ) is Zeno. Otherwise, we construct $GZG(\mathcal{A}, s, \delta)$ and rely on Theorem 2 to answer whether (s, δ) is non-Zeno or not, while updating the two sets $NZSet$ and $ZSet$ accordingly. We further remark that we only need to keep the *maximal* elements in $NZSet$ and $ZSet$. That is, any state (s, δ) in $NZSet$ (or $ZSet$) can be omitted if there exists a state (s, δ') such that $\delta' \subseteq \delta$ (or $\delta \subseteq \delta'$). In other words, $NZSet$ and $ZSet$ are anti-chains [1], [40].

Algorithm 1: Function $nz(s, \delta, \mathcal{A}, NZSet, ZSet)$

```
Input: timed automaton $\mathcal{A}$ and an abstract state $(s, \delta)$
Output: true if and only if $(s, \delta)$ is non-Zeno
if there exists $(s, \delta')$ in $NZSet$ such that $\delta' \subseteq \delta$ then
    return true;
end if
if there exists $(s, \delta')$ in $ZSet$ such that $\delta \subseteq \delta'$ then
    return false;
end if
construct $GZG(\mathcal{A}, s, \delta)$;
for all $(s', \delta')$ in $GZG(\mathcal{A}, s, \delta)$ do
    if $(s', \delta')$ is non-Zeno by Theorem 2 then
        add $(s', \delta')$ into $NZSet$;
    else
        add $(s', \delta')$ into $ZSet$;
    end if
    return true iff it is non-Zeno by Theorem 2;
end for
```

The detailed implementation of $witness(s_p, X_q, \delta, \mathcal{P}, \mathcal{Q})$ is presented in Algorithm 2. It maintains four global variables: $NZSet_p$, $NZSet_p$, $NZSet_q$, $ZSet_q$. In particular, $NZSet_p$ (and $ZSet_p$) is a set of abstract states which are known to be non-Zeno (and Zeno) in $\mathcal{P}$. Similarly, $NZSet_q$ (and $ZSet_q$) is a set of abstract states which are known to be non-Zeno (and Zeno) in $\mathcal{Q}$. The following establishes the correctness of Algorithm 2.

Lemma 1: $witness(s_p, X_q, \delta, \mathcal{P}, \mathcal{Q})$ returns true only if (s_p, X_q, δ) contains a witness state of $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$.

Proof: Given a set of states (s, δ) of $\mathcal{C}(\mathcal{A})$, Algorithm 1 returns true if and only if (s, δ) contains a non-Zeno state By Theorem 2. Recall that (s_p, X_q, δ) encodes a set of states of the form (s, X) of $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$. Algorithm 2 returns true only at line 11. Because of line 5, (s_p, X_q, δ) must encode a state (s, X) such that s is non-Zeno in $\mathcal{C}(\mathcal{P})$, by the correctness of Algorithm 1. Because of line 6 to 10, by the correctness of Algorithm 1, states in X must be all Zeno.

Furthermore, because of line 1-3, either (s, X) has an incoming transition labeled with an event in Σ (since all transitions in $\mathcal{Z}_r$ are labeled with events) or there is a state (s', X') which does and (s, X) can be reached from (s', X') via time delay only. In the latter case, X' must be Zeno (because of line 6 to 10) and by the definition of non-Zenoness, s' must be non-Zeno. In either case, (s_p, X_q, δ) contains a witness state and

Algorithm 2: Function $witness(s_p, X_q, \delta, \mathcal{P}, \mathcal{Q})$

```
**Input**: $\mathcal{P}$ and $\mathcal{Q}$ and a state $(s_p, X_q, \delta)$ of $\mathcal{Z}_r$
**Output**: true iff $(s_p, X_q, \delta)$ contains a witness state
 1: if $(s_p, X_q, \delta)$ is the initial state of $\mathcal{Z}_r$ then
 2:     return false;
 3: end if
 4: let $NZSet_p$, $NZSet_p$, $NZSet_q$, $ZSet_q$ be empty sets;
 5: if $nz(s_p, \delta[C_p], \mathcal{P}, NZSet_p, ZSet_p)$ is true then
 6:     for all $(s_q, A) \in X_q$ do
 7:         if $nz(s_q, \delta[A], \mathcal{Q}, NZSet_q, ZSet_q)$ is true then
 8:             return false;
 9:         end if
10:     end for
11:     return true;
12: end if
13: return false;
```

thus, the lemma holds. □

Lemma 2: If there is a state (s_p, X_q, δ) in $\mathcal{Z}_r$ containing a witness state of $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$, there is a reachable state (s'_p, X'_q, δ') in $\mathcal{Z}_r$ such that $witness(s'_p, X'_q, \delta', \mathcal{P}, \mathcal{Q})$ returns true.

Proof: If (s_p, X_q, δ) contains a witness state (s, X) such that s is non-Zeno and X is Zeno, the condition at line 1 of $witness(s_p, X_q, \delta, \mathcal{P}, \mathcal{Q})$ must be false, since (s, X) must has an incoming transition labeled with an event by definition. By the correctness of Algorithm 1, the condition at line 5 must be true. Next, we analyze two cases. If all other states contained in (s_p, X_q, δ) are witness states, $witness(s_p, X_q, \delta, \mathcal{P}, \mathcal{Q})$ returns true at line 11. Otherwise, it returns false at line 8 because there is a state (s', X') in (s_p, X_q, δ) such that X' is not Zeno. In the former case, the lemma holds. In the following, we show that in the latter case, there must be state (s'_p, X'_q, δ') reachable from (s_p, X_q, δ) such that $witness(s'_p, X'_q, \delta', \mathcal{P}, \mathcal{Q})$ returns true.

Because X is Zeno and X' is not, there must be a state $(s_q, v_q) \in X$ which is Zeno and a state $(s_q, v'_q) \in X'$ which is not Zeno. Thus, there must be a non-Zeno run from (s_q, v'_q) which is infeasible from (s_q, v_q). By induction, we can show that there must be a transition which (s_q, v'_q) can fire after a number of steps whereas (s_q, v_q) cannot. Let tr be the first such transition. By the construction of $\mathcal{Z}_\infty$, there must be at least two states in $\mathcal{Z}_r$, one (say M) is guard by the transition guard of tr and the other guarded by its negation (say N). As a result, (s_q, v'_q) can only reach M whereas (s_q, v_q) can only reach N. Furthermore, any concrete state in N reachable from (s_q, v_q) must be Zeno (by definition). Since (s_q, v'_q) is an arbitrary state in X', eventually (s_p, X_q, δ) would reach a state which contains only witness states. Thus, the lemma holds. □

Theorem 3: $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{Q})$ if and only if there is no state (s_p, X_q, δ) in $\mathcal{Z}_r$ such that $witness(s_p, X_q, \delta, \mathcal{P}, \mathcal{Q})$ returns true.

Proof: (**if**) If $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{Q})$ does not hold, by Theorem 1, there is a witness state in $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$. By Proposition 1, there is a state in $\mathcal{Z}_r$ which contains a witness state. By Lemma 2, there is a state (s_p, X_q, δ) in $\mathcal{Z}_r$ such that $witness(s_p, X_q, \delta, \mathcal{P}, \mathcal{Q})$ returns true.

(**only if**) If there is a reachable state (s_p, X_q, δ) in $\mathcal{Z}_r$ such that $witness(s_p, X_q, \delta, \mathcal{P}, \mathcal{Q})$ returns true, by Lemma 1, (s_p, X_q, δ) contains a witness state. By Proposition 1, there is a witness state in $\mathcal{C}(\mathcal{P}) \otimes det(\mathcal{C}(\mathcal{Q}))$. By Theorem 1, $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{Q})$ does not hold. Thus, the theorem holds. □

5.2 Simulation Reduction

Let F_r be the set states of in $\mathcal{Z}_r$ such that function $witness$ returns true. We have so far reduced the language inclusion checking problem to the problem of checking whether a state in F_r is reachable. Next, we improve the reachability analysis by exploring simulation relation in $\mathcal{Z}_r$ with respects to F_r.

In the following, we first show that the lower-upper bounds (hereafter LU-bounds) simulation relation defined in [6] can be extended to a simulation relation in $\mathcal{Z}_r$ with respects to F_r. 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 $\mathcal{Z}_r$.

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 $U(s, c) < v(c) < v'(c)$. 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 3: Let (s, X, δ_i) where $i \in \{0, 1\}$ be two states of $\mathcal{Z}_r$. (s, X, δ_1) simulates (s, X, δ_0) w.r.t. F_r if $\delta_0 \preccurlyeq_{LU} \delta_1$.

Proof: First we show that $(s, X, \delta_0) \in F_r$ implies $(s, X, \delta_1) \in F_r$. (*) If $(s, \delta_0[C_p])$ is non-Zeno, $(s, \delta_1[C_p])$ is also non-Zeno because $\delta_0 \preccurlyeq_{LU} \delta_1$ implies $\delta_0 \subseteq \delta_1$. (**) By definition, $L(s_x, c)$ is always the same as $U(s_x, c)$ for any clock c in A, which can be seen as the classical maximal bounds discussed in [8]. Following the result in [8], we conclude that for all $(s_x, A) \in X$, if $\delta_0 \preccurlyeq_{LU} \delta_1$, $(s_x, \delta_0[A])$ is non-Zeno if and only if $(s_x, \delta_1[A])$ is. Thus if $witness(s, X, \delta_0, \mathcal{P}, \mathcal{Q})$ returns true at line 11, $witness(s, X, \delta_1, \mathcal{P}, \mathcal{Q})$ must return true by (*) and (**).

Next, we show that if (s, X, δ_0) transitions to (s', X', δ'_0) through an event e, there must be a corresponding transition from (s, X, δ_1) to a state (s', X', δ'_1) via e and $\delta'_0 \preccurlyeq_{LU} \delta'_1$. By the definition of $\mathcal{Z}_r$, if (s, X, δ_0) transitions to (s', X', δ'_0) through an event e, there must be a transition (s, e, g_p, X_p, s') in $\mathcal{P}$ and δ'_0 is $([Y \cup X_p := 0](\delta_0 \wedge g \wedge g_p))^{\uparrow}$, where $g \in Cons(e, X)$ and Y is a certain set of clocks. By Lemma 3 of [6], we can show that $\delta_0 \wedge g \wedge g_p$ is not false implies $\delta_1 \wedge g \wedge g_p$ is not false (since $\delta_0 \preccurlyeq_{LU} \delta_1$) and hence there must be a corresponding transition from (s, X, δ_1) to a state $(s', X'', [Y \cup X_p := 0](\delta_1 \wedge g \wedge g_p))$. Next, following [6], we can show that $[Y \cup X_p := 0](\delta_0 \wedge g \wedge g_p) \preccurlyeq_{LU} [Y \cup X_p := 0](\delta_1 \wedge g \wedge g_p)$. Lastly, we show that $X'' = X'$. By definition, X'' is a set of states of S_∞ such that for any $(s'', A'') \in X''$, there exists a state $(s, A) \in X$ and a transition $((s, A), e, g_s, Y, (s'', A'')) \in T_\infty$ such that $\delta_1 \wedge g_p \wedge g \wedge g_s$ is not false. X' is similarly defined. Since $\delta_0 \preccurlyeq_{LU} \delta_1$, $\delta_1 \wedge g_p \wedge g \wedge g_s$ is not false if $\delta_0 \wedge g_p \wedge g \wedge g_s$ is not false. Thus, $X' \subseteq X''$. Next, we show that if $\delta_0 \wedge g_p \wedge g \wedge g_s$ is false, then $\delta_1 \wedge g_p \wedge g \wedge g_s$ is false and hence every state not in X' is not in X'' and thus $X' = X''$. Thus the lemma holds. □

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 $\{v | \exists v' \in \delta \cdot v \preccurlyeq_{LU} v'\}$. Notice that we may not be able to represent this set as a convex time constraint and there are techniques to get around this problem [6]. 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 pq in $\mathcal{Z}_r^{LU}$ as $post(pq, \mathcal{Z}_r^{LU})$. By a simple argument, we can show that there is a reachable witness state in $\mathcal{Z}_r$ if and only if there is a reachable witness state in $\mathcal{Z}_r^{LU}$.

5.3 The Algorithm

Lastly, we present an on-the-fly algorithm for the language inclusion checking with non-Zenoness. Let $\mathcal{Z}_r^{LU}$ be the tuple $(S, Init, \Sigma, T)$ where $Init$ is a set $(init_p, Init_s, LU((C_p = 0 \wedge z_0 = 0)^\uparrow))$. Algorithm 3 constructs $\mathcal{Z}_r^{LU}$ on-the-fly while performing reachability analysis with simulation reduction and non-Zenoness checking. 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 $done$. If the state is a witness state, we return false at line 7. We generate successors of ps at line 9, and they are added into $working$ so that they are explored later. Lastly, we return true at line 15 after exploring all states. The following theorem states that the algorithm always produces correct results.

Theorem 4: Algorithm 3 returns true if and only if $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{S})$.

Proof Given a state ps in $\mathcal{Z}_r^{LU}$, we define distance $Dist(ps) \in \mathbb{N} \cup \{\infty\}$ as the length of the shortest trace for ps to reach a state in F_r. If no witness state can be reached from ps, $Dist(ps) = \infty$. $Dist(ps) = 0$ if and only if ps is a witness state. Next, we lift the function to a set of states. Given a set Y of states of $\mathcal{Z}_r^{LU}$, if $Y = \varnothing$ then $Dist(Y) = \infty$, otherwise $Dist(Y) = min_{ps \in Y} Dist(ps)$. It can be proven that the loop in Algorithm 3 preserves the following two invariants.

- There exists a witness state ps in $working \cup done$ if and only if ps is reachable from some state in $Init$.
- If there is a reachable witness state, $Dist(done) > Dist(working)$.

Algorithm 3 returns false only at line 7. In such a case, a state in F_r is reachable from $Init$ by the first invariant and thus $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{S})$ does not hold. Algorithm 3 returns true only when $working$ is empty, which implies that $Dist(done) > Dist(working)$ is not true. By the second invariant, $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{S})$ holds. Thus, the theorem holds. □

Algorithm 3: Language Inclusion Checking

```
1:  let $working := Init$;
2:  let $done := \emptyset$;
3:  while $working \neq \emptyset$ do
4:      take $ps = (s_p, X_s, \delta)$ out from $working$;
5:      add $ps$ into $done$;
6:      if $witness(s_p, X_s, \delta)$ returns true then
7:          return false;
8:      end if
9:      for all $(s_p', X_s', \delta') \in post(ps, \mathcal{Z}_r^{LU})$ do
10:         if $(s_p', X_s', \delta') \notin done$ then
11:             put $(s_p', X_s', \delta')$ into $working$;
12:         end if
13:     end for
14: end while
15: return true;
```

Next, we establish a sufficient condition for the termination of the algorithm.

Theorem 5: Let S be the set of states of $\mathcal{Z}_r^{LU}$. If S is finite, Algorithm 3 is terminating. □

The above theorem implies that our algorithm always terminates if the clock boundedness condition defined in [5] is satisfied. The clock boundedness condition is satisfied if the number of clocks in $\mathcal{Z}_r$ is bounded. It has been shown in [5] that if the boundedness condition is satisfied, $\mathcal{Z}_r$ and hence $\mathcal{Z}_r^{LU}$ have a bounded number of clocks and the set S is finite (assuming that maximum ceiling zone normalization is applied).

6 Evaluation

Our method has been implemented with 46K lines of C# code and integrated into the PAT model checker [32]. In our implementation, convex zones are encoded as difference bound matrix (DBM) and zone operations are casted to operations on DBM. Interested readers are referred to [7], [8] for details on DBM implementation. Note that in our setting, a zone may not be convex, e.g., $x > 2 \vee y < 3$ (due to negation used in constructing $\mathcal{Z}_r$), and thus cannot be represented as a single DBM. Such non-convex zones can be represented either as a difference bound logic formula, as shown in [4], or as a set of DBMs. In this work, the latter approach is adopted for efficiency. In the following, we evaluate our approach in order to show the feasibility of our algorithm. 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.

In the first empirical study, we model and verify benchmark timed systems using our semi-algorithm and evaluate its performance. The objective is to show our approach is reasonably scalable (i.e., terminates within a reasonable amount of time) in verifying these systems. The benchmark systems include Fischer's mutual exclusion protocol (Fischer for short), Lynch-Shavit's mutual exclusion protocol (Lynch), railway control system (Railway), and CSMA/CD protocol (CSMA). The results are shown in Table 1 where symbol '-' means either the verification time is more than 2 hours or out-of-memory exception happens. 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 timed automata (following the timed patterns documented in [17], [19]). Some of the properties contain one timed automaton with one clock, while some are networks of timed automata with more than one clock (i.e., one clock for each timed automaton). In the table, column '$|C_q|$' is the number of clocks in the specification. The systems in the same group, e.g., Lynch*4 and Lynch*5, have the same specification. Column 'Det' shows whether the specification is deterministic or not. The results of our semi-algorithm are shown in column 'Non-Zenoness with Reduction', which is compared with the algorithm without the assumption of non-Zenoness (column 'without Non-Zenoness'). In order to show the efficiency of the simulation reduction, we also present the results of the algorithm without simulation reduction, as shown in column 'Non-Zenoness without Reduction'. The number of transitions includes the transitions when constructing the product, and the transitions generated in Algorithm 2.

The results in Table 1 show that in all these cases, our semi-algorithm terminates, which is partly due to the fact that the properties people verify these systems against are often not very complicated. Compared to language inclusion checking without non-Zenoness, checking with non-Zenoness incurs quite some computational over-head, mainly due to the construction (for multiple times) of the guessing zone graphs. Lastly, it is evident that the simulation reduction is helpful in reducing the number of explored transitions and consequently the checking time.

In the second empirical study, we investigate how often our semi-algorithm terminates. We extend the approach on generating non-deterministic finite automata in [33] to automatically generate random timed automata, and then apply our semi-algorithm for language inclusion checking with non-Zenoness. 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. We generate k transitions (and hence the transition density for the events in the alphabet is $Dt = k/(2|S|)$) and distribute the transitions randomly among all $|S|$ states (the two events 0 and 1 are also distributed on the transitions where the number for event 0 is $num = k/2$ and the one for event 1 is $(k-num)$). For instance, if $|S|$ is set to be 6 and Dt is set to be 1.1, we randomly generate 13 transitions (round off to the nearest integer). 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 separate group of implementation specification pairs by generating an implementation first, and then randomly adding 20% transitions to obtain the specification.

The experimental results are shown in Table 2. The column $|S|$ and $|C|$ show the number of states and clocks. For each different combinations of $|S|$, $|C|$ and Dt, we compute three numbers, namely a, b and c respectively. a is the percentage of the specification satisfying the clock boundedness condition (and therefore being determinizable [5]). b is the percentage of cases in which Algorithm 3 without non-Zenoness checking terminates. c is the percentage of cases in which Algorithm 3 terminates. The gap between a and c thus shows the effectiveness of our approach on timed automata which may be non-determinizable. The gap between b and c show how often non-Zenoness checking helps to make language inclusion checking terminating. Additionally, it evidences that non-Zenoness is prevalent in language inclusion checking. We generate 500 random pairs to calculate each number.

It can be observed that in all cases $c \geq b > a$. b is always larger than a because even if the specification may not be deteminzable, since Algorithm 3 is on-the-fly, it may still terminate as long as a violation of the language inclusion is identified. c is no smaller than b because as soon as non-Zenoness checking helps to identify a state containing a witness state, we do not have to continue exploring from that state, which may contribute to the termination.

It can be observed that there is co-relation between transition density Dt and the terminating percentage, i.e., value of a, b and c. For instance, when $Dt \leq 0.8$, the terminating percentage is quite high, which implies that our semi-algorithm often terminates. In general, the lower the density is, the more likely that our approach terminates. In order to have some indication on whether our semi-algorithm would terminate verifying systems against common timed property patterns [17], [19] and the benchmark systems, we calculate the transition density of the common timed property patterns and the benchmark systems. We find that all the models have transition densities less than or equal to 1.0 except the absence pattern (refer to [17] for details on the pattern). Based on the results presented in Table 2, we conclude that in practice, our semi-algorithm has a relatively high probability (i.e., ≥ 0.6) of being terminating.

7 Conclusion

In summary, the contributions of this work are twofold. First, we develop a zone-based approach for language inclusion checking of timed automata with non-Zenoness, which is further combined with simulation reduction for better performance. Furthermore, we investigate when the semi-algorithm is terminating. Secondly, 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 language inclusion checking for timed automata, with non-Zenoness. As for the future work, we would like to investigate further optimization techniques so that our approach is more scalable.

Acknowledgement

This work was supported by National Natural Science Foundation of China (61602412, 61572426, U1509214, 61373033), and the research project from Singapore University of Technology and Design (T2MOE1303).

References

[1] P. A. Abdulla, Y.-F. Chen, L. Holk, R. Mayr, and T. Vojnar. When simulation meets antichains. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '10)*, pages 158–174, 2010.

[2] P. A. Abdulla, J. Ouaknine, K. Quaas, and J. Worrell. Zone-Based Universality Analysis for Single-Clock Timed Automata. In *Proceedings of International Symposium on Fundamentals of Software Engineering*, pages 98–112, 2007.

[3] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theory of Computer Science*, 126(2):183–235, 1994.

[4] R. Alur, L. Fix, and T. A. Henzinger. Event-clock Automata: A Determinizable Class of Timed Automata. *Theoretical Computer Science*, 211:253–273, 1999.

[5] C. Baier, N. Bertrand, P. Bouyer, and T. Brihaye. When Are Timed Automata Determinizable? In *Proceedings of 36th International Colloquium on Automata, Languages and Programming*, pages 43–54, 2009.

TABLE 1
Experiments on Language Inclusion Checking with Non-Zenoness

System	$\|C_q\|$	Det	without Non-Zenoness			Non-Zenoness without Reduction			Non-Zenoness with Reduction		
			states	transitions	time(s)	states	transitions	time(s)	states	transitions	time(s)
Fischer*5	1	Yes	998	2263	0.1	1603	61646	29.9	1603	45178	14.9
Fischer*6	1	Yes	4410	10144	0.9	7065	289698	138.3	7065	217303	73.3
Fischer*7	1	Yes	20044	47060	6.3	31307	1378184	881.9	31307	1056986	457.4
Fischer*4	2	No	1820	3917	0.4	3938	122934	75.1	3938	85900	49.1
Fischer*5	2	No	8456	20142	2.5	21604	795771	589.8	21604	530577	450.4
Lynch*4	1	Yes	903	2150	0.1	2225	136945	54.2	2225	123135	30.8
Lynch*5	1	Yes	3852	11725	0.8	16193	1293511	519.1	16193	1157550	370.2
CSMA*5	1	Yes	2384	5645	0.7	32	5557631	473.4	32	3004445	233.3
Railway*6	1	Yes	26731	39589	4.2	22005	4818205	1273.7	22005	2671729	686.5

TABLE 2
Experiments on Language Inclusion Checking with Non-Zenoness for Random Timed Automata

$\|S\|$	$\|C\|$	$Dt = 0.6$			$Dt = 0.8$			$Dt = 1.0$			$Dt = 1.1$			$Dt = 1.3$		
		a	b	c	a	b	c	a	b	c	a	b	c	a	b	c
4	1	0.97	0.99	0.99	0.76	0.95	0.96	0.65	0.89	0.89	0.48	0.70	0.70	0.11	0.22	0.27
4	2	0.94	0.99	0.99	0.70	0.88	0.90	0.56	0.80	0.81	0.39	0.56	0.57	0.18	0.23	0.31
4	3	0.96	0.99	0.99	0.73	0.86	0.87	0.57	0.74	0.75	0.41	0.55	0.62	0.10	0.17	0.27
6	1	0.98	1.00	1.00	0.89	0.97	0.98	0.49	0.72	0.72	0.37	0.53	0.61	0.18	0.25	0.32
6	2	0.98	0.99	0.99	0.88	0.96	0.97	0.43	0.61	0.66	0.30	0.43	0.54	0.11	0.17	0.33
6	3	0.98	0.99	0.99	0.90	0.96	0.97	0.41	0.58	0.61	0.29	0.40	0.50	0.14	0.18	0.36
8	1	0.99	1.00	1.00	0.75	0.91	0.91	0.40	0.55	0.63	0.25	0.38	0.49	0.06	0.07	0.25
8	2	0.99	0.99	1.00	0.75	0.89	0.90	0.36	0.52	0.61	0.25	0.36	0.51	0.03	0.05	0.24
8	3	0.99	1.00	1.00	0.73	0.89	0.88	0.35	0.49	0.60	0.20	0.31	0.52	0.03	0.05	0.15

[6] G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. Lower and Upper Bounds in Zone-based Abstractions of Timed Automata. *International Journal on Software Tools for Technology Transfer*, 8(3):204–215, 2004.

[7] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Y. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *Proceedings of 11th International Conference on Computer Aided Verification (CAV 99)*, page 341353, 1999.

[8] J. Bengtsson and Y. Wang. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124, 2004.

[9] N. Bertrand, A. Stainer, T. Jéron, and M. Krichen. A Game Approach to Determinize Timed Automata. In *Proceedings of the 14th International Conference on Foundations of Software Science and Computational Structures (FOSSACS '11)*, pages 245–259, 2011.

[10] D. Beyer, C. Lewerentz, and A. Noack. Rabbit: A Tool for BDD-based Verification of Real-Time Systems. In *CAV*, pages 122–125. Springer, 2003.

[11] P. Bouyer. Forward Analysis of Updatable Timed Automata. *Formal Methods in System Design*, 24(3):281–320, 2004.

[12] H. Bowman and R. Gómez. How to Stop Time Stopping. *Formal Aspects of Computing*, 18(4):459–493, 2006.

[13] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In *CAV*, volume 1427 of *LNCS*, pages 546–550. Springer, 1998.

[14] S. Cattani and M. Z. Kwiatkowska. A Refinement-based Process Algebra for Timed Automata. *Formal Aspects of Computing*, 17(2):138–159, 2005.

[15] K. Chatterjee and V. S. Prabhu. Synthesis of Memory-efficient Real-time Controllers for Safety Objectives. In *HSCC*, pages 221–230. ACM, 2011.

[16] D. L. Dill, A. J. Hu, and H. Wong-Toi. Checking for Language Inclusion Using Simulation Preorders. In *CAV*, pages 255–265, 1991.

[17] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-State Verification. In *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP '98)*, pages 7–15, 1998.

[18] R. Gómez and H. Bowman. Efficient Detection of Zeno Runs in Timed Automata. In *FORMATS*, volume 4763 of *LNCS*, pages 195–210. Springer, 2007.

[19] V. Gruhn and R. Laue. Patterns for Timed Property Specifications. *Electronic Notes in Theoretical Computer Science*, 153(2):117–133, 2006.

[20] T. A. Henzinger, Z. Manna, and A. Pnueli. What Good are Digital Clocks? In *Proceedings of 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, pages 545–558, 1992.

[21] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-time Systems. *Journal of Information and Computation*, 111(2):193–244, 1994.

[22] F. Herbreteau and B. Srivathsan. Efficient On-the-Fly Emptiness Check for Timed Büchi Automata. In *ATVA*, pages 218–232, 2010.

[23] F. Herbreteau, B. Srivathsan, and I. Walukiewicz. Efficient Emptiness Check for Timed Büchi Automata. *Formal Methods in System Design*, 40(2):122–146, 2012.

[24] M. Krichen and S. Tripakis. Conformance Testing for Real-Time Systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[25] K. G. Larsen, P. Petterson, and Y. Wang. UPPAAL in a Nutshell. *Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[26] J. Ouaknine and J. Worrell. On The Language Inclusion Problem for Timed Automata: Closing a Decidability Gap. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science*, pages 54–63, 2004.

[27] J. Ouaknine and J. Worrell. Safety metric temporal logic is fully decidable. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 411–425, 2006.

[28] J. Raskin and P. Schobbens. The Logic of Event Clocks - Decidability, Complexity and Expressiveness. *Journal of Automata, Languages, and Combinatories*, 4(3):247–286, 1999.

[29] T. G. Rokicki. *Representing and Modeling Digital Circuits*. PhD thesis, Stanford Uni., 1993.

[30] A. W. Roscoe. On the Expressive Power of CSP Refinement. *Formal Aspects of Computing*, 17(2):93–112, 2005.

[31] P. V. Suman, P. K. Pandya, S. N. Krishna, and L. Manasa. Timed Automata with Integer Resets: Language Inclusion and Expressiveness. In *Proceedings of 6th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 08)*, pages 78–92, 2008.

[32] J. Sun, Y. Liu, J. S. Dong, , and J. Pang. PAT: Towards flexible verification under fairness. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*, pages 709–714, 2009.

[33] D. Tabakov and M. Y. Vardi. Experimental evaluation of classical automata constructions. In *Proceedings of the 12th international conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '05)*, pages 396–411, 2005.

[34] S. Tripakis. Verifying progress in timed systems. In *Proceedings of the*

5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems (ARTS '99), pages 299–314, 1999.

[35] S. Tripakis, S. Yovine, and A. Bouajjani. Checking Timed Büchi Automata Emptiness Efficiently. *FMSD*, 26(3):267–292, 2005.

[36] Stavros Tripakis, Sergio Yovine, and Ahmed Bouajjani. Checking timed büchi automata emptiness efficiently. *Formal Methods in System Design*, 26(3):267–292, 2005.

[37] F. Wang. Symbolic Verification of Complex Real-Time Systems with Clock-Restriction Diagram. In *Formal Techniques for Networked and Distributed Systems*, pages 235–250. Springer, 2002.

[38] T. Wang, J. Sun, Y. Liu, X. Wang, and S. Li. Are timed automata bad for a specification language? language inclusion checking for timed automata. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 310–325, 2014.

[39] Y. Wang, P. Pettersson, and M. Daniels. Automatic Verification of Real-Time Communicating Systems by Constraint Solving. In *Proceedings of the 7th International Conference on Formal Description Techniques*, pages 223–238, 1994.

[40] M. D. Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A New Algorithm for Checking Universality of Finite Automata. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV '06)*, pages 17–30, 2006.