# A Systematic Study on Explicit-state Non-Zenoness Checking for Timed Automata

Ting Wang, Jun Sun, Xinyu Wang, Yang Liu, Yuanjie Si, Jin Song Dong, Xiaohu Yang, Xiaohong Li

**Abstract**—Zeno runs, where infinitely many actions occur within finite time, may arise in Timed Automata models. Zeno runs are not feasible in reality and must be pruned during system verification. Thus it is necessary to check whether a run is Zeno or not so as to avoid presenting Zeno runs as counterexamples during model checking. Existing approaches on non-Zenoness checking include either introducing an additional clock in the Timed Automata models or additional accepting states in the zone graphs. In addition, there are approaches proposed for alternative timed modeling languages, which could be generalized to Timed Automata. In this work, we investigate the problem of non-Zenoness checking in the context of model checking LTL properties, not only evaluating and comparing existing approaches but also proposing a new method. To have a systematic evaluation, we develop a software toolkit to support multiple non-Zenoness checking algorithms. The experimental results show the effectiveness of our newly proposed algorithm, and demonstrate the strengths and weaknesses of different approaches.

**Index Terms**—Timed Automata; non-Zenoness; Model Checking; Verification Tool

## 1 Introduction

Timed Automata [1], [2] are popular for real-time system modeling and verification. They allow modeling of real-time systems through explicit manipulation of clock variables. Real-time behavior is captured by clock constraints on system transitions, setting or resetting clocks, etc. Verification tools for Timed Automata based models have proven to be successful [3], [4], [5], [6]. Nonetheless, researchers have also identified various limitations for Timed Automata based system modeling and verification [1], [7]. For instance, Timed Automata are not determinizable [8], [9]; modeling hierarchical systems in Timed Automata is non-trivial [7], [10], [11], etc.

In this work, we focus on the emptiness checking problem, i.e., the problem of checking whether a Timed Automaton accepts at least one non-Zeno run. 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 verification. That is, it is necessary to check whether a run is Zeno or not so as to avoid presenting Zeno runs as counterexamples. For instance, 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 interesting is that it is infeasible with zone abstraction [12]. Zone abstraction, which constructs zone graphs, is an effective technique for model checking Timed Automata and it has been employed by many tools including Uppaal [3]. Zone graphs are however too abstract to directly infer time progress and hence non-Zenoness. There have been existing approaches on solving the problem of combining non-Zenoness checking and zone abstraction. The basic idea is to enhance the zone graphs with additional information which facilitates non-Zenoness checking. The proposed approaches include introducing one clock in the Timed Automaton model [13], [14] or adding additional accepting states in the zone graph [12], [15].

Despite the existing approaches, there are a number of issues yet to be investigated. Firstly, the state-of-the-art emptiness checking algorithm [15] has a complexity of $(|C|+1)^2 \cdot |ZG|$ where $|C|$ is the number of clocks and $|ZG|$ is the size of the zone graph. In other words, it incurs significant overhead in checking non-Zenoness (as compared to constructing the zone graph only). On the other hand, there are methods proposed for alternative real-time system modeling languages which incur much less overhead. For instance, in [16], the authors show that non-Zenoness checking for Stateful Timed CSP can be achieved based on zone graphs without adding clocks or accepting states, i.e., with a complexity of $|ZG|$. A closer look shows that their proof relies on the fact that clocks in Stateful Timed CSP have constant upper bounds. Given that Stateful Timed CSP and Timed Automata have similar expressiveness [16], [17], their algorithm can be potentially extended to, at the least, a subset of Timed Automata. The question is then whether we extend their algorithm to arbitrary Timed Automata by transforming them into a


- *Ting Wang, Xinyu Wang, Yuanjie Si and Xiaohu Yang are with the College of Computer Science, Zhejiang University, P.R. China. E-mail: {qdw,wangxinyu,siyuanjie,yangxh}@zju.edu.cn*
- *Jun Sun is with ISTD, Singapore University of Technology and Design, Singapore. E-mail: sunjun@sutd.edu.sg*
- *Yang Liu is with the School of Computer Engineering, Nanyang Technological University, Singapore. E-mail: yangliu@ntu.edu.sg*
- *Jin Song Dong is with the School of Computing, National University of Singapore, Singapore. E-mail: dongjs@comp.nus.edu.sg*
- *Xiaohong Li is with the School of Computer Science and Technology, Tianjin University, P.R. China. E-mail: xiaohongli@tju.edu.cn*
- *The corresponding author Xinyu Wang is with the College of Computer Science, Zhejiang University, Zheda Road 38, Hangzhou, Zhejiang Province, P.R. China, 310000. E-mail: wangxinyu@zju.edu.cn. Phone: +8613867468299.*

form satisfying similar syntactical conditions and whether the transformation is beneficial.

Secondly, the existing approaches [14], [15] are mostly developed for Timed Büchi Automata, rather than Timed Safety Automata [2] which are supported by popular tools like UPPAAL and are often used in practice. The question is then whether they work effectively despite the difference between Timed Büchi Automata and Timed Safety Automata. For instance, given a network of Timed Safety Automata, the approach proposed in [13] adds one state for each accepting state and thus would double the number of states and potentially lead to state space explosion. Thirdly, there are limited evaluation and comparison on the existing algorithms, within a practical and fair environment. For instance, in practice, checking whether a system contains a non-Zeno run has limited usage by itself. Instead, it is more useful to check whether there is a non-Zeno counterexample given a property (for instance, a temporal logic formula). Therefore, it is necessary to develop a model checker which makes use of the proposed algorithms and compare them in the same setting. Lastly, though different approaches have been proposed, to the best of our knowledge, few model checkers provide satisfactory support to deal with the non-Zenoness problem. UPPAAL [3] provides some form of non-Zenoness detection but it is sufficient-only. KRONOS [4] allows a sufficient and necessary condition for non-Zenoness checking, but it is computationally expensive. It is thus important to provide a tool which offers the best of all the approaches.

In this work, we make the following new technical contributions. Firstly, we generalize the approach in [16] so as to solve the non-Zenoness checking problem for Timed Automata. The rationale is that common timed behavior patterns like *delay*, *timeout* and *deadline* can be captured using clocks which have constant upper bounds. For instance, if a clock $c$ is used to model a 'timeout' at time $d$, $c$ is associated with an upper bound $d$ which remains constant. Based on the results shown in [16], we define a subclass of Timed Safety Automata, called CUB-TA, and develop an efficient non-Zenoness checking algorithm for CUB-TA with a complexity $|ZG|$ (i.e., linear in the size of the zone graph) without adding any extra clocks or states. To be precise, CUB-TA are Timed Safety Automata whose clocks have non-decreasing upper bound along any path before they are reset. Furthermore, we develop an algorithm which automatically transforms an arbitrary Timed Automaton into an equivalent CUB-TA by paying the price of potential extra states. Though introducing extra states may incur computational overhead, we show that a number of benchmark Timed Automata models either are CUB-TA or can be transformed into equivalent CUB-TA by adding only few states. As a result, non-Zenoness can be checked more efficiently than previous approaches. Secondly, we conduct a systematic comparison between existing approaches by model checking a number of systems modeled using a network of Timed Safety Automata, against temporal logic properties, with the assumption of non-Zenoness. The comparison results allow us to identify the best algorithm in different settings. Lastly, we develop a software toolkit (in the PAT framework [18]) which combines the existing approaches and heuristically selects the 'right' algorithm based on the input model. To the best of our knowledge, our implementation in PAT is the only model checker which directly supports model checking LTL with the non-Zenoness assumption.

**Organization** The remainders of the paper are organized as follows. Section 2 defines the Timed Safety Automata and the non-Zenoness problem. Section 3 presents the existing non-Zenoness checking algorithms. Section 4 proposes a new approach. Section 5 presents the implementations and evaluation results. Section 6 concludes the paper and discusses the related work.

## 2 BACKGROUND

In this section, we present the definitions of Timed Automata and zone abstraction.

### 2.1 Timed Automata

Let $\mathbb{R}^+$ denote the set of non-negative real numbers. Given a set of clocks $C$, the set of clock constraints is defined inductively based on a primitive constraint $\delta := true \mid x \sim n \mid x - y \sim n \mid \delta_1 \wedge \delta_2$ where $\sim \in \{=, <, \leq, >, \geq\}$; $x, y$ are clocks in $C$ and $n \in \mathbb{R}^+$ is a constant. Without loss of generality (Lemma 4.1 of [1]), we assume that $n$ is a non-negative integer constant. A constraint of the form $x - y \sim n$ is called diagonal. We write $\Phi(C)$ to denote the set of all diagonal-free clock constraints over $C$. The set of downward constraints in $\Phi(C)$ 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 valuation $v$ satisfies a clock constraint $\delta$, written as $v \models \delta$, if and only if $\delta$ evaluates to true using the clock values given by $v$. 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 $[X \mapsto 0]v$ denote the valuation $v'$ such that $v'(c) = v(c)$ for all $c \in C \setminus X$ and $v'(x) = 0$ for all $x \in X$.

**Definition 1:** A Timed Büchi Automaton (TBA) is a tuple $\mathcal{A}_b = (S, Init, \Sigma, C, F, T)$ where $S$ is a finite set of control locations; $Init \subseteq S$ is a set of initial locations; $\Sigma$ is an alphabet; $C$ is a finite set of clocks; $F \subseteq S$ is a set of accepting locations; $T \subseteq S \times \Sigma \times \Phi(C) \times 2^C \times S$ is a labeled transition relation.

A configuration of a TBA $\mathcal{A}_b$ is a pair $(s, v)$ such that $s \in S$ is a location and $v$ is a clock valuation. A run of $\mathcal{A}_b$ is an infinite sequence $\pi = \langle (s_0, v_0), (d_0, e_0), (s_1, v_1), (d_1, e_1), \cdots \rangle$ such that $s_0 \in Init$; $v_0$ assigns 0 to every clock; and for all $i \geq 0$, there exists a transition $(s_i, e_i, \delta, X, s_{i+1}) \in T$ such that $v_i + d_i \vDash \delta$ and $v_{i+1} = [X \mapsto 0](v_i + d_i)$. $\pi$ is non-Zeno if and only if the sum of all $d_i$ is unbounded. Given $\pi$, we can obtain a timed word $\langle (d_0, e_0), (d_1, e_1), \cdots, (d_i, e_i), \cdots \rangle$.

To ensure progress, a subset of the locations $F$ are marked as accepting and a run is accepting if it visits any accepting

location infinitely often. With the accepting conditions, the automaton cannot idly stay in a location forever, and only accepting runs are considered as valid behaviors of the automaton. We define the language of a TBA to be the set of timed words which can be obtained from the set of accepting non-Zeno runs. A TBA is non-empty if and only if its language is not an empty set. Two TBA are equivalent if they define the same language.

For example, Fig. 1(a) models a Train Process in Railway Control System [19] using TBA, where *Safe* is the initial location and *Cross* is an accepting location. Since *Cross* is the only accepting location, all accepting runs of the automaton must visit *Cross* infinitely often. It implies that the location *Appr* must be left when the value of clock $c$ is at most 20, otherwise the automaton will get stuck at location *Appr* and never be able to enter *Cross*. Likewise, the automaton must leave *Start* when $c$ is at most 15 to be able to enter *Cross*.

Later, Timed Safety Automaton (TSA) was introduced in [2] which adopts an intuitive notion of progress. Instead of having accepting locations, each location in TSA is associated with a local timing constraint called a *location invariant*. An automaton can stay at a location as long as the valuation of the clocks satisfy the invariant. The timed expressiveness of Timed Safety Automata is strictly less than that of Timed Büchi Automata [20].

**Definition 2:** A Timed Safety Automaton is a tuple $\mathcal{A} = (S, Init, \Sigma, C, L, T)$ where $S$ is a finite set of control locations; $Init \subseteq S$ is a set of initial locations; $\Sigma$ is an alphabet; $C$ is a finite set of clocks; $L : S \to \Phi_{\leq,<}(C)$ labels each location with an invariant; $T \subseteq S \times \Sigma \times \Phi(C) \times 2^C \times S$ is a labeled transition relation.

Similar to TBA, a configuration of a TSA $\mathcal{A}$ is a pair $(s, v)$ such that $s \in S$ is a location and $v$ is a clock valuation with $v \vDash L(s)$. A run of $\mathcal{A}$ is an infinite sequence $\pi = \langle (s_0, v_0), (d_0, e_0), (s_1, v_1), (d_1, e_1), \cdots \rangle$ such that $s_0 \in Init$; $v_0$ assigns every clock to 0; and for all $i \geq 0$, there is a transition $(s_i, e_i, \delta, X, s_{i+1}) \in T$ such that $v_i + d_i \vDash L(s_i)$ and $v_i + d_i \vDash \delta$ and $v_{i+1} = [X \mapsto 0](v_i + d_i)$ and $v_{i+1} \vDash L(s_{i+1})$. $\pi$ is non-Zeno if and only if the sum of all $d_i$ is unbounded. Given $\pi$, we can obtain a timed word $\langle (d_0, e_0), (d_1, e_1), \cdots, (d_i, e_i), \cdots \rangle$. We define the language of a TSA to be the set of timed words which can be obtained from the set of non-Zeno runs. Two TSA are equivalent if they define the same language.

The main difference between TBA and TSA is that for TBA, accepting locations are used to guarantee time progress, while in TSA, location invariants are used. A TBA and a TSA are equivalent if they define the same language. For example, consider the TSA in Fig. 1(b), which is equivalent to the TBA in Fig. 1(a). The invariant specifies a local condition that location *Cross* must be left when $c$ is at most 5, *Appr* must be left when $c$ is at most 20, and *Start* must be left when $c$ is at most 15. This gives a local view of the timing behavior of the automaton at each location. In the rest of the paper, we focus on TSA as they are supported by popular tools like UPPAAL and are often used for system modeling[1]. For simplicity, they are referred simply as Timed Automata unless otherwise stated.

In many practical applications, a system is composed of many components running in parallel. Each of these components can be modeled as a Timed Automaton, and a composition operator can be used to define a network of communicating Timed Automata. In the following, we define the parallel composition of two Timed Automata, which can be readily extended to multiple Timed Automata.

**Definition 3:** Let $\mathcal{A}_i = (S_i, Init_i, \Sigma_i, C_i, L_i, T_i)$ where $i \in \{1, 2\}$ be two Timed Automata. Parallel composition of $\mathcal{A}_1$ and $\mathcal{A}_2$, written as $\mathcal{A}_1 \parallel \mathcal{A}_2$, is a Timed Automaton $(S, Init, \Sigma, C, L, T)$ such that $S = S_1 \times S_2$; $Init = Init_1 \times Init_2$; $\Sigma = \Sigma_1 \cup \Sigma_2$; $C = C_1 \cup C_2$; for all $(s_1, s_2) \in S$, $L((s_1, s_2)) = L_1(s_1) \wedge L_2(s_2)$; $T$ is the smallest transition relation satisfying the following rules.

- $((s_1, s_2), a, \delta, X, (s_1', s_2)) \in T$ for all $s_2 \in S_2$ if $(s_1, a, \delta, X, s_1') \in T_1$ and $a \notin \Sigma_2$, .
- $((s_1, s_2), a, \delta, X, (s_1, s_2')) \in T$ for all $s_1 \in S_1$ if $(s_2, a, \delta, X, s_2') \in T_2 \wedge a \notin \Sigma_1$.
- if $(s_1, a, \delta, X, s_1') \in T_1$ and $(s_2, a, \delta', X', s_2') \in T_2$ and $a \in \Sigma_1 \cap \Sigma_2$, $((s_1, s_2), a, \delta \wedge \delta', X \cup X', (s_1', s_2')) \in T$.

The three rules above state that a transition of the composition is either a local transition of either $\mathcal{A}_1$ or $\mathcal{A}_2$, or a synchronization on a common event of $\mathcal{A}_1$ and $\mathcal{A}_2$. Parallel composition is commutative and associative.

## 2.2 Zone Abstraction

Zone abstraction is an effective technique for model checking Timed Automata, which has been employed by many tools including UPPAAL [3]. The result of zone abstraction is a zone graph, which is subject to model checking. It is also known that zone graphs are too abstract to directly infer time progress and hence non-Zenoness [12]. 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.

It is well known that a zone can be equivalently represented as a DBM (short for Difference Bound Matrices [21], [22]). Let $t_1, t_2, \cdots, t_n$ denote $n$ clocks and $t_0$ denote a dummy clock whose value is always 0. A DBM representing a constraint on the clocks is a $(n+1) \times (n+1)$ matrix. Entry $(i, j)$ in the matrix is a pair $(\sim_j^i, d_j^i)$ where $\sim_j^i \in \{<, \leq\}$ and $d_j^i$ is an integer, representing the upper bound on difference between clock $t_i$ and $t_j$, i.e., $t_i - t_j \sim_j^i d_j^i$. A DBM thus represents the conjunction of clock constraints $t_i - t_j \sim_j^i d_j^i$ for all clocks $t_i$ and $t_j$ such that $0 \leq i \leq n$ and $0 \leq j \leq n$. Interested readers are referred to [21], [22] for more details on zone operations and its corresponding

1. TBA are often used to model properties.

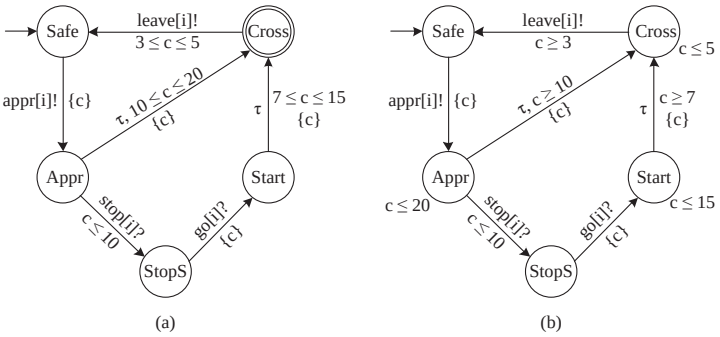



Fig. 1. TBA and TSA

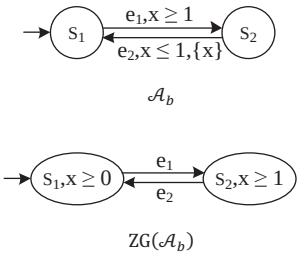



Fig. 2. A sample zone graph

DBM implementation. The following zone operations are relevant in this work. Given a zone $\delta$, we use $\delta^{\uparrow}$ to denote the zone reached by delaying an arbitrary amount of time from zone $\delta$. Given a set of clocks $C$, we use $\delta \backslash C$ to denote the zone obtained by removing the constraints on clocks in $C$, i.e., projection onto clocks not in $C$.

Before giving the formal definition of zone graphs, it is necessary to introduce the *zone normalization* function, denoted as $\mathcal{N}$, to ensure that the number of zones in a zone graph is finite [23], [24]. The idea of normalization 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 $\mathcal{A}$. The intuition is that once the value of a clock is larger than the ceiling, its precise value is no longer relevant, but only the fact that it is larger than the ceiling.

**Definition 4:** Let $\mathcal{A} = (S, Init, \Sigma, C, L, T)$ be a timed automaton. Its zone graph, denoted as $ZG(\mathcal{A})$, is a tuple $(S_z, Init_z, \Sigma, T_z)$ such that

- $S_z$ is a set of nodes, each of which is a pair $(s, \delta)$ such that $s \in S$ is a location and $\delta$ is a clock constraint;
- $Init_z = \{(init, \mathcal{N}((\bigwedge_{c \in C} c = 0)^{\uparrow} \wedge L(init))) \mid init \in Init\}$ is a set of initial nodes;
- $T_z : S_z \times \Sigma \times S_z$ is a transition relation such that $((s_1, \delta_1), e, (s_2, \delta_2)) \in T_z$ if and only if there exists a transition $(s_1, e, \delta, X, s_2) \in T$ and $\delta_1 \wedge \delta \neq \mathit{false}$ and $[X \mapsto 0](\delta_1 \wedge \delta) \wedge L(s_2) \neq \mathit{false}$ and $\delta_2 = (\mathcal{N}([X \mapsto 0](\delta_1 \wedge \delta) \wedge L(s_2)))^{\uparrow}$.

We remark that the initial zones of the nodes in $Init_z$ are always normalized, and the transition relation $T_z$ ensures that all the zones in $S_z$ are normalized. Fig. 2 shows a simple automaton $\mathcal{A}_b$ and its corresponding zone graph $ZG(\mathcal{A}_b)$.

A *path* of $ZG(\mathcal{A})$ is an infinite or finite sequence of transitions: $\pi_z = \langle (s_0, \delta_0), e_0, (s_1, \delta_1), e_1, (s_2, \delta_2), e_2, \cdots \rangle$ where $(s_0, \delta_0) \in Init_z$ and $((s_i, \delta_i), e_i, (s_{i+1}, \delta_{i+1})) \in T_z$ for all $i \geq 0$. A run $\langle (s_0, v_0), (d_0, e_0), (s_1, v_1), (d_0, e_0), \cdots \rangle$ of $\mathcal{A}$ is an *instance* of $\pi_z$ if $v_i \models \delta_i$ for all $i \geq 0$. The path is called an *abstraction* of the run. It can be shown that every path in $ZG(\mathcal{A})$ is an abstraction of a run of $\mathcal{A}$, and conversely, every run of $\mathcal{A}$ is an instance of a path in $ZG(\mathcal{A})$ [25]. As a result, zone graph preserves reachability and linear properties. Similarly, we can define the zone graph and related concepts for TBA and a path of the zone graph for a TBA is accepting if and only if it visits some accepting node infinitely often.

## 3 ALGORITHMS

In this section, we present existing approaches for non-Zenoness checking.

### 3.1 Algorithm 1: Strongly Non-Zeno

One solution to the non-Zenoness problem is to transform a TBA into a *strongly non-Zeno automaton* (SNZ). An SNZ is a TBA satisfying a syntactic condition such that all accepting runs starting at the initial location are non-Zeno. SNZ was defined in [13] and has been implemented in the tool Profounder [14].

Formally, given a TBA $\mathcal{A}_b = (S, Init, \Sigma, C, F, T)$ and a clock $t$ not in $C$, the corresponding SNZ, denoted as $SNZ(\mathcal{A}_b)$, is constructed using the following procedure.

1) Add a fresh clock $t$ into $C$;
2) For each location $s$ in $F$, add a new accepting location $s'$;
3) For each transition $(s_i, e, \delta, X, s) \in T$, where $s \in F$ and $s_i \in S$, add a new transition $(s_i, e, \delta \wedge t \geq 1, X \cup \{t\}, s')$;
4) For each location $s$ in $F$, replace $s$ with $s'$ in $F$;
5) For each location $s'$ in $F$ and the corresponding location $s$, add a transition $(s', \tau, \mathit{true}, \varnothing, s)$;

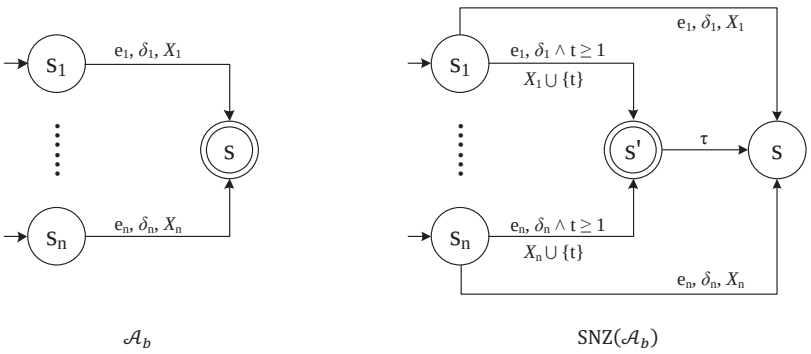



Fig. 3. Strongly non-Zeno timed automaton [14]

The construction is illustrated in Fig. 3. Informally, to transform an arbitrary Timed Automaton $\mathcal{A}_b$ into an equivalent SNZ, one needs to add one new clock $t$ to monitor time progress. Furthermore, every accepting location is duplicated such that one copy of the location is as before but is no longer accepting, while the other copy is accepting, but only can be reached when $t \geq 1$. Moreover when the second copy is reached, $t$ is reset to 0. Intuitively, this construction ensures that at least one unit of time has passed between two visits to an accepting location. Though proposed for TBA, this approach can be easily applied to TSA. Given a TSA $\mathcal{A}$, because all locations in $\mathcal{A}$ are 'accepting', to construct $SNZ(\mathcal{A})$, every location in $\mathcal{A}$ has to be duplicated. For example, Fig. 4 shows the constructed SNZ for the TSA in Fig. 1(b). Notice that every location is copied and each newly added location has an incoming transition with a guard condition $t \geq 1$.

**Lemma 1 [15]:** If a path in $ZG(\mathcal{A}_b)$ visits infinitely often both a transition that bounds some clock $t$ from below and a transition that resets the same clock $t$, then all its instances are non-Zeno. □

The converse is also true. If $\mathcal{A}_b$ has a non-Zeno accepting run, at least 1 time unit elapses infinitely often. Hence, the guard $t \geq 1$ is satisfied infinitely often and there exists an accepting run of $SNZ(\mathcal{A}_b)$ and one can find an accepting path in the zone graph of $SNZ(\mathcal{A}_b)$. The following theorem shows that every accepting run of $SNZ(\mathcal{A}_b)$ is non-Zeno.

**Theorem 1 [13]:** Let $\mathcal{A}_b$ be a TBA. $SNZ(\mathcal{A}_b)$ is strongly non-Zeno. □

Furthermore, it is easy to show that if $\mathcal{A}_b$ and $\mathcal{A}'_b$ are strongly non-Zeno, so is $\mathcal{A}_b \parallel \mathcal{A}'_b$. Thus, given a network of TBA, we can convert each TBA to an SNZ and guarantee that the composition is strongly non-Zeno. After constructing $SNZ(\mathcal{A}_b)$, the problem of checking whether there is an accepting non-Zeno run is reduced to checking whether it visits any accepting location infinitely often (i.e., Büchi acceptance), which has been studied extensively

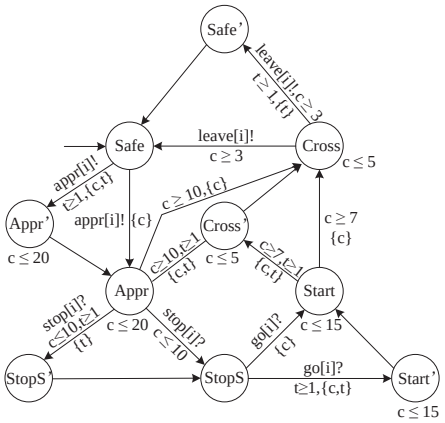



Fig. 4. The train model in SNZ

(see [26] for a survey). For instance, Tarjan's SCC (Strongly Connected Components, i.e., maximal strongly connected subgraphs) algorithm can be used to find an accepting cycle in the zone graph.

This simple approach of adding one clock, however, may lead to an exponential blowup in the size of the zone graph, as shown in [12]. Consider the TBA $\mathcal{B}_b$ in Fig. 5 which yields an exponentially larger zone graph with the addition of one clock. For simplicity, we omit the event labeling if the event is $\tau$. Due to the inherent nondeterminism, one obtains $k!$ zones at location $a$ that describe an ordering of the form $x_{i_1} \leq x_{i_2} \leq \cdots \leq x_{i_k}$ with $\{i_1, i_2, \cdots, i_k\} = \{1, 2, \cdots, k\}$. Automaton $\mathcal{B}_b$ has a Zeno cycle between location $a$ and $b$, with $b$ being the only accepting location. As a result, any algorithm that searches for a non-Zeno accepting cycle has to explore the entire zone graph. Consequently, an algorithm that first transforms $\mathcal{B}_b$ into a strongly non-Zeno automaton needs to explore at least $k!\,(d-1)^k$ zones. This is because the modified automaton $SNZ(\mathcal{B}_b)$ yields a zone graph that includes zones describing distances.

### 3.2 Algorithm 2: Guessing Zone Graph

In order to address the problem of exponential blowup, an alternative solution has been proposed in [12], which

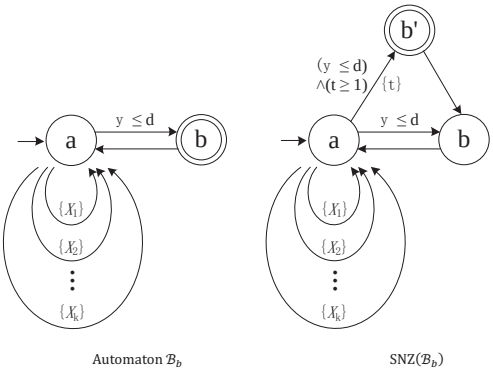



Fig. 5. Exponential blow up in the zone graph [12]

avoids adding extra clocks. The proposed method is to construct a *guessing zone graph* which introduces extra states into the zone graph. Notice that this approach was also proposed for TBA and can be readily extended to networks of TSA. The intuition behind this approach is that there are two kinds of Zeno paths in the zone graph. 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 total time elapsed is bounded. Or, the path contains a transition that resets $x$, and subsequently a transition that requires $x = 0$. Thus time cannot elapse at all. This is called zero-checks. Guessing zone graphs are designed to handle these two cases explicitly.

**Definition 5:** Given a TBA $\mathcal{A}_b = (S, Init, \Sigma, C, F, T)$ and its zone graph $ZG(\mathcal{A}_b) = (S_z, Init_z, \Sigma, T_z)$, $GZG(\mathcal{A}_b)$ is a labeled transition system $(S_g, Init_g, \Sigma, T_g)$ such that a state in $S_g$ is of the form $(s, \delta, Y)$ where $(s, \delta)$ is a node in $ZG(\mathcal{A}_b)$ and $Y \subseteq C$; $Init_g = \{(s_0, \delta_0, C) \mid (s_0, \delta_0) \in Init_z\}$; and $T_g : S_g \times \Sigma \times \Phi(C) \times 2^C \times S_g$ is the least transition relation which satisfies the following two 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 $\mathcal{A}_b$ and there is a transition $((s_1, \delta_1), e, (s_2, \delta_2))$ in $ZG(\mathcal{A}_b)$ and there are clock valuations $v \in \delta_1$, $v' \in \delta_2$ and $d \in \mathbb{R}^+$ such that $v + d \vDash \bigwedge_{x \in C-Y} x > 0$ and $v + d \vDash \delta$ and $[X \mapsto 0](v + d) = v'$.
- $((s, \delta, Y), \tau, true, \varnothing, (s, \delta, Y')) \in T_g$ where $Y' = \varnothing$ or $Y' = Y$ where $\tau$ is a special invisible event. □

Intuitively, the $Y$ component of a node $(s, \delta, Y)$ allows us to infer that the clocks not in $Y$ are strictly positive. A node of the form $(s, \delta, \varnothing)$ 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 elapse in this node. A node is *accepting* if it contains an accepting location $s$. A path of the guessing zone graph is non-Zeno if all clocks bounded from above are reset infinitely often during the run and the run visits a clear node [12]. 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*. Fig. 6 shows a simple zone graph $ZG(\mathcal{A}_b)$ along with its guessing zone graph $GZG(\mathcal{A}_b)$, corresponding to the automaton shown in Fig. 2.

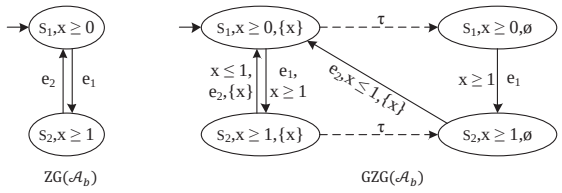



Fig. 6. A zone graph and its guessing zone graph

**Algorithm 1:** GZG emptiness check

```
Input: a zone graph ZG
Output: true if and only if the model is non-empty
1  while there are unvisited states in ZG do
2      find an accepting SCC scc;
3      if scc contains blocking clocks then
4          remove all blocking transitions from scc;
5          apply Algorithm 1 to scc;
6      end
7      else if scc is nonblocking with zero-checks then
8          construct GZG gzg for scc;
9          if gzg contains an SCC which contains a clear
           node then
10             return true;
11         end
12     end
13     else return true;
14 end
15 return false;
```

**Theorem 2 [12]:** A TBA $\mathcal{A}_b$ has a non-Zeno accepting run if and only if there exists an unblocked path in $GZG(\mathcal{A}_b)$ visiting both an accepting node and a clear node infinitely often. □

This theorem reduces the problem of non-Zenoness checking to a Büchi acceptance checking plus a simple check on whether a clock bounded from above is always reset. Thus, the non-Zenoness checking problem can be solved based on algorithms like Tarjan's SCC algorithm, with a complexity linear in the size of $GZG(\mathcal{A}_b)$. Nonetheless, since $GZG(\mathcal{A}_b)$ is $|C| + 1$ times larger than $ZG(\mathcal{A}_b)$, it is important to avoid constructing the full $GZG(\mathcal{A}_b)$ if possible. Guessing zone graphs are designed to detect nodes where time elapse is not prohibited by future zero-checks. However, in the absence of zero-checks, this construction is not necessary, i.e., it is sufficient to construct the part of $GZG(\mathcal{A}_b)$ corresponding to SCCs in $ZG(\mathcal{A}_b)$ with zero-checks. An on-the-fly algorithm was proposed in [15], shown as Algorithm 1, which detects zero-checks and constructs $GZG(\mathcal{A}_b)$ only if necessary.

Given a zone graph (not guess zone graph), Algorithm 1

returns true if it contains a non-Zeno accepting run. For simplicity, transitions in the zone graph are marked with two additional labels. One is a set of resetting clocks and the other is a set of clocks that are bounded from above. Algorithm 1 applies Tarjan's algorithm to identify SCCs. Once an SCC *scc* which contains at least one accepting state in the zone graph is found (line 2), we check whether any clock is blocked at line 3 by comparing the union of resetting clocks and the union of bounded clocks of the transitions in *scc*. If some clocks are blocked (i.e., condition at line 3 is satisfied), we remove blocking transitions at line 4 (i.e., transitions which put an upper bound on a clock which is never reset in *scc*). Intuitively, the blocking transitions are the reasons why time can not go unbounded and therefore loops formed by the remaining transitions in *scc* may form non-Zeno paths. At line 5, we apply Algorithm 1 recursively so as to check whether there are non-Zeno accepting runs in the remaining of *scc*. If no clock is blocked in *scc*, we check whether *scc* contains zero-checks at line 7. If it does, the corresponding GZG is constructed and Algorithm 1 returns true only if the corresponding GZG contains an SCC which contains a clear node. If there is no zero-check, the loop formed by the states and transitions in *scc* is non-Zeno and therefore Algorithm 1 returns true at line 13.

The worst case complexity of the algorithm 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 often performs better by constructing only a small part of $GZG(\mathcal{A}_b)$. In the following, we illustrate how Algorithm 1 works with a simple example.

Consider the TBA $\mathcal{A}_b$ in Fig. 7, where location 2 is accepting. For readability, a node of $ZG(\mathcal{A}_b)$ is written as $([m](z), n)$ where $m$ is a location in $\mathcal{A}_b$, $z$ is a zone and $n$ is an identifier. Algorithm 1 firstly identifies the SCC formed by the nodes with identifiers 2, 3, 4, 5 and 6, which are connected by the double edges in $ZG(\mathcal{A}_b)$. Next, it finds that the SCC is blocking as clock $y$ is bounded from above on the transition from node 5 to 2 and is never reset. Hence, the blocking transition from node 5 to 2 is removed and Algorithm 1 is applied to the remaining states and transitions in the SCC. The SCC containing nodes 4, 5 and 6 is now identified, which is both unblocked and accepting. Next, Algorithm 1 finds that the SCC contains zero-checks (on the transition from node 4 to node 5) and the corresponding part of $GZG(\mathcal{A}_b)$ is constructed, shown on the right of Fig. 7. Notice that node 4 of $ZG(\mathcal{A}_b)$ becomes node 1 of the part of $GZG(\mathcal{A}_b)$. Following Definition 5, in $GZG(\mathcal{A}_b)$, we construct node 2 which is clear node. Notice that node 2 in $GZG(\mathcal{A}_b)$ does not have any successor because the set of clear clocks is empty, and hence the outgoing transition with $x == 0$ is omitted. Next, Algorithm 1 finds an SCC in $GZG(\mathcal{A}_b)$ which contains a clear node, i.e., the SCC formed by node 3, 6, 7, 8 and 9 where 7 and 8 are clear nodes.

### 3.3 Alternative Approaches

Besides the two approaches presented above, there are alternative approaches which can be used to solve the problem of model checking with non-Zenoness in the literature. In the following, we review them briefly.

An approach based on a *simulation graph* has been proposed in [14] to check emptiness of a TBA. The proposed algorithm is both symbolic and on-the-fly. The simulation graph is a graph whose nodes are non-empty symbolic states of a TBA and edges represent operations of generating successors of symbolic states. The authors define a subclass of TBA with persistent acceptance conditions, i.e., every outgoing transition of accepting location targets to an accepting location. That is, once a TBA enters accepting locations it never exits. Then the authors distinguish four cases and proposed an algorithm for each case, depending on whether the automaton is strongly non-Zeno or not (a TBA is called strongly non-Zeno if all accepting runs starting at the initial state are non-Zeno), and whether it has persistent acceptance conditions or not. Table 1 lists the different cases and their algorithms. The algorithms have been implemented in a tool called Profounder. However, the nodes of the simulation graph are unions of regions which are non-convex, thus, not efficiently representable. In [25], the authors extended the work of [14] to show that the main result of [14] carries over to a zone-based simulation graph. The idea of [25] is to use simulation graph over-approximations to preserve convexity based on the results in [24] and then extend the algorithms in [14] to zone-closed simulation graph. This approach has been implemented in Open-Kronos for emptiness checking of SNZ.

There have been a series of work on symbolic model checking algorithms for Timed Automata using BDD-like data structures [27]–[30]. It has been shown that non-Zenoness can be supported using an auxiliary clock variable, combined with a greatest fix-point calculation. Their algorithms have been implemented in the tool named RED. While related, symbolic model checking is a different paradigm compared to explicit-state model checking as we discuss in this work. Furthermore, it is known that performance of BDD-based symbolic model checking varies significantly with a variety of factors like variable ordering, encoding techniques, etc. Therefore, in this work, we choose to focus on studying explicit-state algorithms for non-Zenoness checking and comparing them in the same implementation.

In addition, the authors in [31] deal with the Zeno problem in concurrent two-player timed games with safety objectives. In their setting, Timed Automata are viewed as infinite-state timed game structures. In each round of the timed game, both players simultaneously propose moves, with each move consisting of an action and a time delay after which the player wants the proposed action to take place. Of the two proposed moves, the move with the shorter time delay wins the round and determines the next state of the game. To prevent a player from winning by

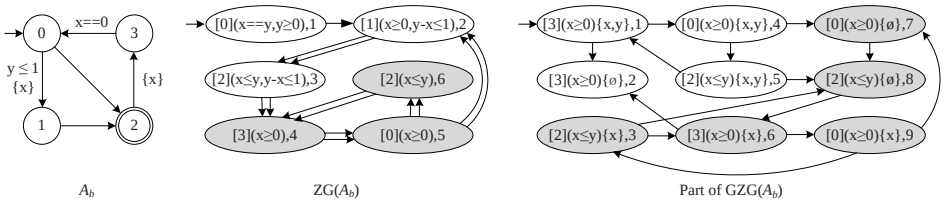



Fig. 7. An example using GZG approach

TABLE 1
The algorithms used in simulation graph

| | Persistent acceptance conditions | Non-persistent acceptance conditions |
|---|---|---|
| Strongly non-Zeno | find simple accepting cycle using simple DFS | find simple accepting cycle using double DFS or SCCs |
| Non strongly non-Zeno | find simple accepting progressive cycle using full DFS or transform to SNZ | find accepting progressive cycle using incomplete search or transform to SNZ |

blocking time, each player is restricted to strategies that ensure the player cannot be responsible for causing a Zeno run. In their approach, non-Zenoness is inferred from the history of certain predicates of the system clocks, rather than from an extra clock that is kept in memory. The authors construct the winning strategies for the controller which requires access to a linear number of memory bits, significantly improving the previous known exponential bound. However this game theoretic approach is presented based on region graphs and it is not clear whether it works with zone graphs. In comparison, the focus of this work is on solving the problem of combining non-Zenoness checking and zone abstraction.

## 4 Emptiness Check for CUB-TA

As shown above, the state-of-the-art emptiness checking algorithm [15] has a complexity of $(|C|+1)^2 \cdot |ZG|$. In other words, it still incurs significant overhead in checking non-Zenoness. On the other hand, there are methods proposed for alternative real-time system modeling languages which incur less overhead. In [16], the authors showed that non-Zenoness checking for Stateful Timed CSP can be solved based on zone graphs without adding extra clocks or states. The reason is that in Stateful Timed CSP, clocks have constant upper bounds and hence, intuitively, there will not be unforeseen zero-checks. In the following, we generalize this result to Timed Automata with non-decreasing upper bounds, referred to as CUB-TA, and propose an alternative approach for non-Zenoness. That is, we propose to transform an arbitrary Timed Automaton into an equivalent CUB-TA, and then use an efficient algorithm to check non-Zenoness. We leave the question on whether this transformation is beneficial to the next section.

### 4.1 Automata with Non-decreasing Upper Bounds

A clock upper bound is either $\infty$ or a pair $(n,\sim)$ where $\sim$ is either $<$ or $\leq$. We write $(n_1,\sim_1) = (n_2,\sim_2)$ to denote $n_1 = n_2$ and $\sim_1 = \sim_2$; $(n_1,\sim_1) \leq (n_2,\sim_2)$ to denote $n_2 > n_1$, or if $n_2 = n_1$, then either $\sim_2$ is $\leq$ or both $\sim_1$ and $\sim_2$ are $<$. Further, we write $(n,\sim) > d$ where $d$ is a constant to denote $n > d$. We define $min((n,\sim_1),(m,\sim_2))$ to be $(n,\sim_1)$ if $(n,\sim_1) \leq (m,\sim_2)$; otherwise, $min((n,\sim_1),(m,\sim_2)) = (m,\sim_2)$. Given a clock $c$ and a clock constraint $\delta$, we write $ub(\delta,c)$ to denote the upper bound of $c$ given $\delta$. Formally,

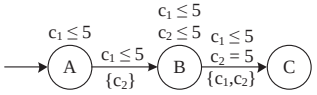
$$ub(\delta,c) = \begin{cases} (n,\sim) & -\text{ if } \delta \text{ is } c \sim n \text{ and } \sim \in \{\leq,<\} \\ \infty & -\text{ if } \delta \text{ is } c > n \text{ or } c \geq n \\ \infty & -\text{ if } \delta \text{ is } x \sim n \text{ and } x \neq c \\ \infty & -\text{ if } \delta \text{ is } true \\ min(ub(\delta_1,c), ub(\delta_2,c)) & -\text{ if } \delta \text{ is } \delta_1 \wedge \delta_2 \end{cases}$$

We fix a TSA $\mathcal{A} = (S, Init, \Sigma, C, L, T)$ and its zone graph $ZG(\mathcal{A}) = (S_z, Init_z, \Sigma, T_z)$ in the following. A path of $\mathcal{A}$ from $s_0$ is a sequence $\pi = \langle s_0, e_0, \delta_0, X_0, s_1, e_1, \delta_1, X_1, \cdots \rangle$ such that $(s_i, e_i, \delta_i, X_i, s_{i+1}) \in T$ for all $i \geq 0$. We write $paths(s_0)$ to denote all paths starting from $s_0$. Given a path $\pi$, let $reset(\pi,c)$ be the number of transitions before $c$ is first reset, i.e., $reset(\pi,c) = k$ if there exists $k$ such that $c \in X_k$ and $c \notin X_j$ for all $0 \leq j < k$; otherwise $reset(\pi,c) = \infty$.

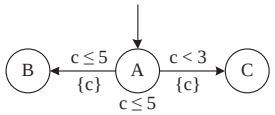
**Definition 6:** A TSA $\mathcal{A}$ is a CUB-TA if and only if for all clock $c \in C$; $s_0 \in S$; $\pi = \langle s_0, e_0, \delta_0, X_0, s_1, e_1, \delta_1, X_1, \cdots \rangle \in paths(s_0)$ and for all $i$ such that $0 \leq i < reset(\pi,c)$, $ub(L(s_i),c) \leq ub(\delta_i,c) \leq ub(L(s_{i+1}),c)$. □

Intuitively, every clock in a CUB-TA has a non-decreasing upper bound along any path before it is reset. For instance, Fig. 8(a) shows a CUB-TA, and Fig. 8(b) shows a TSA which is not, since $c$'s upper bound at location $A$ is $(5,\leq)$ and it is $(3,<)$ for the transition from location $A$ to $C$.

In order to solve the emptiness problem, we extend zone graph $ZG(\mathcal{A})$ with two transition labels. One is a set of resetting clocks, i.e., if a transition in $ZG(\mathcal{A})$ is generated by a transition $(s_1, e, \delta, X, s_2)$ in $\mathcal{A}$, the transition is labeled with $X$. The other label is a Boolean flag $b$ which is true if and only if the transition can *potentially* be delayed. The value of the flag can be determined as follows. Let $c_0$ be a clock such that $c_0 \notin C$. Let




(a) CUB-TA




(b) non-CUB-TA

Fig. 8. CUB-TA examples

$((s_1,\delta_1),e,(s_2,\delta_2))$ be a transition in $ZG$. We set $c_0 = 0$ at node $(s_1,\delta_1)$ so that it becomes $(s_1,\delta_1 \wedge c_0 = 0)$. If $[X \mapsto 0]((\delta_1 \wedge c_0 = 0)^{\uparrow} \wedge \delta) \wedge L(s_2)$ implies $c_0 = 0$ where $\delta$ and $X$ are the clock guard and resetting clocks corresponding to the transition, then the flag $b$ is false. The idea is to have a clock $c_0$ starting at 0 for every state so that by looking at the value of $c_0$ after a transition, we can infer whether the transition is required to occur immediately. *We say that a transition can be delayed* locally *(for some non-zero amount of time) if $c_0 = 0$ is not implied (i.e., flag b is true).* Notice that given a system run, a transition which can be *locally* delayed may in fact be constrained to occur immediately *globally*. For instance, given the CUB-TA in Fig. 8(a), the transition from location $A$ to $B$ can be delayed locally. Nonetheless, given the run starting with location $A$ and ending with $C$, it is implied that the transition from $A$ to $B$ must occur immediately (as the transition from $B$ to $C$ must be delayed for 5 time units). We remark that introducing the clock $c_0$ here is different from the approach of introducing an extra clock for non-Zenoness detection [13] as $c_0$ is 0 in all nodes of the zone graph and therefore no extra state is introduced.

**Theorem 3:** Let $\pi = \langle (s_0,\delta_0),(e_0,X_0,b_0),(s_1,\delta_1),(e_1,X_1,b_1),\cdots\rangle$ be an (infinite) run of $ZG(\mathcal{A})$ where $(X_i,b_i)$ are the newly added transition labels. $\pi$ is non-Zeno if and only if

- $*$ there exist infinitely many $k$ such that $b_k = true$; and
- $\star$ for all $c \in C$, for all $i \geq 0$, if $ub(L(s_i),c) \neq \infty$, there exists $j$ such that $j \geq i$ and $c \in X_j$ or $ub(L(s_j),c) = \infty$.

***Proof*** (***only-if***) If $\pi$ is non-Zeno, $*$ is trivially true. If a clock $c$ is bounded from above (i.e., with an upper bound other than $\infty$), it must be reset later or the upper bound becomes infinity since by definition its value goes unbounded along the run; otherwise, we have an empty zone and thus an infeasible run. Hence, $\star$ is true.
(***if***) In the following, we show that if $*$ and $\star$ are true, $\pi$ is non-Zeno. Let the following be a segment of $\pi$ according to $*$ and $\star$.

$$\langle (s_i,\delta_i),(X_i,b_i),(s_{i+1},\delta_{i+1}),\cdots,(s_j,\delta_j),\cdots,(s_k,\delta_k)\rangle$$

where $i \leq j \leq k$ and $b_j = true$. Furthermore, for all $c \in C$, if $ub(L(s_j),c) \neq \infty$, there exists $m,n$ such that $i-1 \leq m < j \leq n \leq k-1$ such that $c \in X_m$ (or $m = -1$ such that $c$ is 'reset' before the initial state) and, $c \in X_n$ or $ub(L(s_n),c) = \infty$. That is, the segment contains a transition which can be delayed locally. Furthermore, the segment covers the 'life-span' (between two resets) of all clocks in $L(s_j)$ which have an upper bound other than $\infty$.

In general, the infinite run $\pi$ is progressive if and only if it takes an unbounded amount of time. Since there are infinitely many segments as above in $\pi$, if any such segment can take a positive amount of time, then the run $\pi$ is progressive and thus non-Zeno. Next, we show that the segment can take a positive amount of time.

Let $y_x$ denote the number of time units that can elapse from state $s_x$ to $s_{x+1}$ where $i \leq x \leq k-1$. In the following, we obtain *all* constraints on upper bounds of $y_j$. For each clock $c \in C$, we have a set of constraints of the following form: $y_m + y_{m+1} + \cdots + y_t \sim ub(L(s_t),c)$ where $\sim \in \{\leq, <\}$ and $j \leq t \leq n$. The constraints put upper bounds on the total time of a part of the segment, i.e., from the moment $c$ is previously reset to the moment of entering state $s_t$. Notice that constraints on lower bounds are ignored as they are irrelevant. Because $b_j$ is true, it is implied that $ub(L(s_t),c) > 0$ for all $m \leq t \leq n$ (by assumption $b_j = true$). In the following, we analyze two cases.

- If $\sim$ is $\leq$, the constraints are satisfied with $y_j = ub_{min}$ where $ub_{min}$ is the minimum of $ub(L(s_t),c)$ for all $c \in C$ and for all $m \leq t \leq n$ and the rest of the variables equal to 0.
- If $\sim$ is $<$, the constraints are satisfied with $y_j = \frac{ub_{min}}{2}$.

In both cases, the segment is progressive and, therefore, we conclude that $\pi$ is non-Zeno. Furthermore, because $y_x$ where $x \neq j$ is subject to other constraints, it may be that $y_x$ must be strictly positive and as a result the above constraints are satisfiable only when $y_j$ is 0. In such a case, the segment is progressive because $y_x$ is strictly positive and, therefore, we conclude that $\pi$ is non-Zeno. With the arguments above, we conclude that the theorem holds. □

Intuitively, the theorem states that, if there is a transition which can be delayed locally, either it can be delayed globally or some other transition can be delayed globally. The proof does not work for arbitrary Timed Automaton as the upper bound of a clock could be decreasing, e.g., a zero-check may be encountered later and $y_j$ is constrained to be zero.

### 4.2 CUB-TA Emptiness Check

In the following, we present an algorithm to solve the emptiness problem based on the theorem below, which reduces the problem to an SCC searching problem.

**Theorem 4:** A CUB-TA $\mathcal{A}$ contains a non-Zeno run if and only if $ZG(\mathcal{A})$ contains a reachable (maximum) strongly connected component (SCC) such that

- † it contains a transition $((s,\delta),(e,X,b),(s',\delta'))$ such that $b = true$; and
- ‡ for every clock $c$ in $C$, if $ub(L(s),c) \neq \infty$ for some state $(s,\delta)$ in the SCC, there exists a transition in the SCC with label $(X,b)$ such that $c \in X$.

***Proof*** (***only-if***) Assume that the model is non-empty, there must be a non-Zeno run, say $\pi$. Since $ZG$ is finite-state, $\pi$ must visit a set of states and transitions, denoted as $Inf$, infinitely often. There must be an SCC, say *scc*, which contains $Inf$. $Inf$ must contain a transition with a label $b$ being true (by contradiction) and therefore † is trivially true. Next, we prove ‡ by contradiction. Assume there is a state $s$ in *scc* where a clock $c$ has an upper bound $d$ which is not $\infty$ and there is no transition in *scc* which resets $c$. Because the upper bound of $c$ never decreases (by definition of CUB-TA), the upper bound of $c$ at every state in *scc* must be $d$. Since *scc* contains $Inf$, this implies that $\pi$ is Zeno as $c$ is always bounded from above and never reset, which contradicts our assumption that $\pi$ is non-Zeno. Thus, *scc* must satisfy ‡.
(***if***) Assume there is an SCC satisfying † and ‡. Let $\pi$ be a run which visits every state/transition in the SCC infinitely often. It is easy to see that $\pi$ satisfies $*$ of Theorem 3 because of †. By ‡, we conclude that every clock which has an upper bound other than $\infty$ at a state is reset later. Therefore, $\pi$ is non-Zeno by Theorem 3 and therefore $\mathcal{A}$ contains a non-Zeno run.
Therefore, we conclude that the theorem holds. □

The above theorem implies that in order to solve the emptiness problem, we need to test each SCC against two conditions: whether it contains a transition which can be locally delayed; and whether every clock which has an upper bound other than $\infty$ at some state is reset along some transition in the SCC. Notice that both checks have a complexity linear in the size of the SCC. This leads to Algorithm 2, which can be extended to model check temporal logic properties (e.g., LTL) with the assumption of non-Zenoness. It takes a CUB-TA as input, and constructs $ZG$ on-the-fly while applying Tarjan's algorithm to identify SCCs. Once an SCC is found, we check whether it satisfies † and ‡. If so, it returns true at line 5. After checking all SCCs, it returns false.

The correctness of the algorithm can be established based on Theorem 4. The algorithm is terminating as $ZG$ (with zone normalization) is finite-state. The complexity of the algorithm is linear in time $|ZG|$ (which is due to Tarjan's algorithm for identifying SCC). The overhead of checking † and ‡ is minor.

### 4.3 Transform Arbitrary Automaton to CUB-TA

Algorithm 2 works only for CUB-TA. In the following, we develop an approach for non-Zenoness checking by transforming an arbitrary Timed Automaton to an equivalent CUB-TA. Recall that in Section 2.1 the language of a TSA is defined as the set of timed words which can be obtained from the set of non-Zeno runs. Hereafter we say that a timed automaton and the corresponding CUB-TA are equivalent if and only if they define the same language. Intuitively, a Timed Automaton is not a CUB-TA if there exists a 'problematic' transition $(s, e, \delta, X, s')$ and a clock $c$ such that $ub(L(s), c)$ is larger than $ub(\delta, c)$; or $c \notin X$ and $ub(L(s), c)$ is larger than $ub(L(s'), c)$. In both cases, we can 'strengthen' the invariant of $s$ by adopting the respective clock upper bounds in $\delta$ or $L(s')$ without affecting the system taking this transition. It may however affect other transitions, e.g., strengthening $L(s)$ might disable other transitions. Hence, to obtain an equivalent CUB-TA, we can split location $s$ into multiple ones, each of which is labeled with a different upper bound for taking a different transition. Afterwards, we can remove the 'problematic' transitions from the original location so that the resultant Timed Automaton is a CUB-TA.

**Algorithm 2:** CUB-TA emptiness check

```
Input: a zone graph ZG(A)
Output: true if and only if A contains at least one
        non-Zeno run
1 while there are un-visited states in the zone graph do
2 |   find a new SCC scc;
3 |   mark all states in scc as visited;
4 |   if scc satisfies † and ‡ then
5 |   |   return true;
6 |   end
7 end
8 return false;
```

Given a TSA $\mathcal{A} = (S, Init, \Sigma, C, L, T)$, the transformation is shown in Algorithm 3. Given two clock constraints $\delta$ and $\gamma$, we write $\delta \leq \gamma$ to denote that for all clock $c$, $ub(\delta, c) \leq ub(\gamma, c)$, i.e., all clock upper bounds are non-decreasing from $\delta$ to $\gamma$. We write $\delta \not\leq \gamma$ otherwise, i.e., when some clock upper bound is decreasing. Recall that given a clock constraint $\delta$, $\delta \setminus X$ denotes the constraint obtained by removing constraints on clocks in X. In an abuse of notations, assuming $ub(\delta_1, c_1) = (n_1, \sim_1)$ and $ub(\delta_2, c_2) = (n_2, \sim_2)$, we write $ub(\delta_1, c_1) \wedge ub(\delta_2, c_2)$ to denote $c_1 \sim_1 n_1 \wedge c_2 \sim_2 n_2$. Furthermore, we write $ub(\delta, C)$ to denote $\bigwedge_{c \in C} ub(\delta, c)$ where $C$ is a set of clocks, i.e., the conjunction of upper bounds on all clocks in $C$.

The algorithm has four parts. Part one (line 1 to 12) finds out 'problematic' transitions for every location $s$ in $S$ and collects all clock constraints which have some decreasing upper bounds in the set $constraints(s)$. Given a transition $(s, e, \delta, X, s')$ in $T$ and any clock $c \in C$, if $c \notin C$ (or $c \in C$), $ub(L(s), c)$ is compared to $ub(\delta, c)$ and $ub(L(s'), c)$ (or $ub(\delta, c)$). Thus, at line 6 and 8, the location invariant $L(s)$ is compared to $\delta \wedge (L(s') \setminus X)$ (or $\delta \wedge (\gamma \setminus X)$), which is a conjunction of the constraint on the transition and the location invariant of $s'$ (or a constraint from $constraints(s')$) on the clocks which are not reset along the transition. If some clock's upper bound is decreasing in $\delta \wedge (L(s') \setminus X)$ (or $\delta \wedge (\gamma \setminus X)$), $L(s) \wedge \delta \wedge (L(s') \setminus X)$ (or $L(s) \wedge \delta \wedge (\gamma \setminus X)$) is added to $constraints(s)$. Notice that every constraint $\gamma \in constraints(s)$ conjuncts $L(s)$ and thus $\gamma \leq L(s)$. The set $constraints(s)$ is a monotonically increasing set, which reaches a fixed point when the loop from line 2 to 12 finishes. We remark that if the given TSA is a CUB-TA, $constraints(s)$ will be empty. Part two (line 13 to 18)

**Algorithm 3:** CUB-TA transformation

**Input**: a Timed Automaton $\mathcal{A} = (S, Init, \Sigma, C, L, T)$
**Output**: a CUB-TA $\mathcal{C}$ which is equivalent to $\mathcal{A}$

1 **set** $constraints(s) := \varnothing$ for all $s \in S$ and $adding := true$;
2 **while** *adding is true* **do**
3  $adding := false$;
4  **for** *each location s in S* **do**
5  **for** *each transition* $(s, e, \delta, X, s')$ *in T* **do**
6  if $L(s) \not\leq \delta \wedge (L(s') \setminus X)$ and $\theta = L(s) \wedge \delta \wedge (L(s') \setminus X) \notin constraints(s)$, add $\theta$ into $constraints(s)$ and set *adding* to be true;
7  **for** *each constraint* $\gamma$ *in* $constraints(s')$ **do**
8  if $L(s) \not\leq \delta \wedge (\gamma \setminus X)$ and $\theta' = L(s) \wedge \delta \wedge (\gamma \setminus X) \notin constraints(s)$, add $\theta'$ into $constraints(s)$ and set *adding* to be true;
9  **end**
10  **end**
11  **end**
12 **end**
13 **for** *each location s in S* **do**
14  set $clones(s)$ to be an empty set;
15  **for** *each constraint* $\delta$ *in* $constraints(s)$ **do**
16  add a new location $st$ to $clones(s)$ and $S$; set $L(st)$ to be $ub(\delta, C)$;
17  **end**
18 **end**
19 **for** *each location s in S* **do**
20  **for** *each location st in* $clones(s)$ **do**
21  **for** *each incoming transition* $(s', e, \delta', X, s)$ *with* $s'$ *in S* **do**
22  add transition $(s', e, \delta', X, st)$;
23  add one transition $(st', e, \delta', X, st)$ for each $st'$ in $clones(s')$;
24  **end**
25  **for** *each outgoing transition* $(s, e, \delta', X, s')$ *with* $s'$ *in S* **do**
26  add transition $(st, e, \delta', X, s')$;
27  add one transition $(st, e, \delta', X, st')$ for each $st'$ in $clones(s')$;
28  **end**
29  **end**
30 **end**
31 remove all transitions $(s, e, \delta, X, s')$ such that $L(s) \not\leq \delta \wedge (L(s') \setminus X)$;
32 set the set of initial locations to be $\{clones(s) \mid s \in Init\} \cup Init$;

'clones' every location $s$ in $S$ multiple times, once for each constraint in $constraints(s)$. Notice that $clones(s)$ denotes the set of all clones of $s$ (exclusive). Part three (line 19 to 30) then copies all incoming/outgoing transitions to/from any location $s$ such that for each transition $(s, e, \delta, X, s')$ in the original TSA, there is a transition $(sc, e, \delta, X, sc')$ for every $sc \in clones(s) \cup \{s\}$ and $sc' \in clones(s') \cup \{s'\}$. Lastly, part four (line 31) removes all 'problematic' transitions so that the result is guaranteed to be a CUB-TA.

For instance, Fig. 9 shows the resultant CUB-TA from the one shown in Fig. 1(b). Fig. 1(b) is not CUB as $ub(L(Appr), c)$ (which is 20) is larger than the upper bound on $c$ in the transition $(Appr, stop[i]?, c \leq 10, \{c\}, StopS)$ (which is 10). To turn it into a CUB-TA using Algorithm 3, we split *Appr* into two locations, i.e., location *Appr* labeled with the original invariant and a new location *New* with invariant $c \leq 10$. Next, incoming/outgoing transitions to the original *Appr* are copied to *New*.

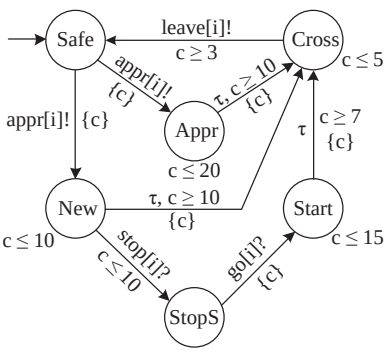



Fig. 9. Equivalent CUB-TA of train process

Afterwards, we remove the 'problematic' transition $(Appr, stop[i]?, c \leq 10, \{c\}, StopS)$. This is safe because the transition must occur when $c \leq 10$ and the same

transition can now take place from *New*. Afterwards, the Train Process becomes a CUB-TA.

**Theorem 5:** Given an arbitrary TSA, applying Algorithm 3 results in an equivalent CUB-TA.

***Proof***: Let $\mathcal{A} = (S_0, Init_0, \Sigma, C, L_0, T_0)$ be the input TSA and $\mathcal{C} = (S_1, Init_1, \Sigma, C, L_1, T_1)$ be the resultant one. It is easy to see that $\mathcal{C}$ is a CUB-TA because of line 31. Next, we show first that the language of $\mathcal{C}$ is a subset of that of $\mathcal{A}$. If line 31 is removed from Algorithm 3, it is easy to see that $\mathcal{C}$ is equivalent to $\mathcal{A}$ as all that has been done is duplicating some locations and transitions. Therefore, we conclude that with line 31, the language of $\mathcal{C}$ is no more than that of $\mathcal{A}$. Next, we argue that removing the 'problematic' transitions at line 31 does not remove any timed word so that the language of $\mathcal{C}$ is no less than that of $\mathcal{A}$.

Given a transition $(s, e, \delta, X, s')$, we say that it is CUB if $L(s) \leq \delta \wedge (L(s') \setminus X)$. In the following, we write $trans_i$ to denote $(s_i, e_i, \delta_i, X_i, s_{i+1})$ and $trans'_i$ to denote $(s'_i, e'_i, \delta'_i, X'_i, s'_{i+1})$. Let $Q = \langle trans_0, trans_1, \cdots, trans_n \rangle$ be a sequence of connected transitions in $\mathcal{A}$. Let $\pi$ be a timed word of $\mathcal{A}$ generated by firing the transitions in $Q$ in sequence. We show, by an induction on the length of $Q$, that the following is true (referred to as $\star$ hereafter): there must be a sequence of connected transitions $Q' = \langle trans'_0, trans'_1, \cdots, trans'_n \rangle$ in $\mathcal{C}$ such that $s_{n+1} = s'_{n+1}$ and $Q'$ generates an equivalent timed word in $\mathcal{C}$.

The base case is when $Q$ contains one transition $Q = \langle trans_0 \rangle$. If $trans_0$ is CUB, it is not removed by line 31 in $\mathcal{C}$ and hence we let $Q' = \langle trans_0 \rangle$ and thus $\star$ holds. If $trans_0$ is not CUB, there must be a location $s'_0$ in $clones(s_0)$ such that $L(s'_0) = ub(L(s_0) \wedge \delta_0 \wedge (L(s_1) \setminus X), C)$ because $L(s_0) \not\leq \delta_0 \wedge (L(s_1) \setminus X_0)$ at line 6 in Algorithm 3. Thus, $(s'_0, e_0, \delta_0, X_0, s_1)$ must be a transition in $\mathcal{C}$ and we set $Q' = \langle (s'_0, e_0, \delta_0, X_0, s_1) \rangle$. A timed word $\pi$ which can be generated by $Q$ can also be generated by $Q'$ (since $s'_0$ is an initial location of $\mathcal{C}$), i.e., if the clock upper bounds in $\delta_0 \wedge (L(s_1) \setminus X)$ are satisfied, they must be satisfied at location $s_0$ and thus also $s'_0$. That is, going through $s'_0$ instead of $s_0$ does not rule out any timed word.

In the following, we assume that $\star$ holds for any $Q$ of length $i \leq k$ (i.e., induction hypothesis) and we prove that $\star$ holds for any $Q$ of length $k+1$. Let $Q = \langle trans_0, \cdots, trans_{k-1}, trans_k \rangle$ be a sequence of $k+1$ connected transition in $\mathcal{A}$. If $trans_k$ is CUB, by induction hypothesis, there exists $\langle trans'_0, \cdots, trans'_{k-1} \rangle$ such that $\star$ is satisfied. We set $Q' = \langle trans'_0, \cdots, trans'_{k-1}, trans_k \rangle$ and it is easy to see that $Q'$ satisfies $\star$ and all transitions in $Q'$ are CUB.

If $trans_k$ is not CUB, there must be a location $s'_k$ in $clones(s_k)$ such that $L(s'_k) = ub(L(s_k) \wedge \delta_k \wedge (L(s_{k+1}) \setminus X_k), C)$ because $L(s_k) \not\leq \delta_k \wedge (L(s_{k+1}) \setminus X_k)$ at line 6 in Algorithm 3. We modify $Q$ such that the last transitions $trans_{k-1}$ and $trans_k$ are replaced with $(s_{k-1}, e_{k-1}, \delta_{k-1}, X_{k-1}, s'_k)$ and $(s'_k, e_k, \delta_k, X_k, s_{k+1})$ respectively. By a simple argument, we can show that doing so does not rule out any timed word, i.e., if the clock upper bounds in $\delta_k \wedge (L(s_{k+1}) \setminus X_k)$ are satisfied, they must be satisfied at location $s_k$ and thus also $s'_k$. Afterwards, if the transition $(s_{k-1}, e_{k-1}, \delta_{k-1}, X_{k-1}, s'_k)$ is CUB, by induction hypothesis, we can generate a sequence of transitions in $\mathcal{C}$, say $\langle trans'_0, \cdots, trans'_{k-2} \rangle$ such that $s'_{k-1} = s_{k-1}$ and we append $(s_{k-1}, e_{k-1}, \delta_{k-1}, X_{k-1}, s'_k)$ and $(s'_k, e_k, \delta_k, X_k, s_{k+1})$ to the sequence so as to obtain a sequence of CUB transitions in $\mathcal{C}$. The resulting sequence evidences that $\star$ holds. Otherwise, we repeat the above to replace $s_{k-1}$ with a state in $clones(s_{k-1})$, and similarly $s_{k-2}, s_{k-3}, \cdots$ afterwards if necessary. We set $Q'$ to be the resulting sequence. Because no timed word is ruled out during each replacement, we can generate the same timed word from $Q'$. Therefore, every timed word of $\mathcal{A}$ is also a timed word of $\mathcal{C}$.

By induction, $\star$ holds for any sequence of transitions in $\mathcal{A}$ and thus $\mathcal{A}$ and $\mathcal{C}$ are equivalent. □

The complexity of Algorithm 3 is linear in $|S| \times |T| \times |C|$ where $|S|$ is the number of locations and $|T|$ is the number of transition and $|C|$ is the number of clocks. In the worst case, every transition introduces a new upper bound for every clock and the upper bounds propagate through every location. That is, the number of iterations of the loop from line 2 to 12 is bounded by the number of locations. The number of extra locations introduced by the above algorithm depends on the number of 'problematic' transitions. The best case is that the given Timed Automaton is a CUB-TA and hence no locations are added. It is possible that a clock upper bound may propagate through multiple paths and results in introducing multiple locations. In the worst case, the number of locations in the resultant TSA is $|S| \times |T|$, i.e., every location is copied for every transition. In practice, however, the worst case is rare as we show in the next section. Compared to the GZG approach, it may be that many of the extra locations are not necessary (e.g., distinguishing a location with a lower upper bound from the original location may not be essential if non-Zenoness can be concluded through other means) and therefore it is possible that transforming a TSA into an CUB-TA is not beneficial. Furthermore, given a network of Timed Automata, adding even one extra location into a component automaton may significantly increase the size of the zone graph. It is thus necessary to evaluate and compare the performance of different approaches with real-world systems.

Given a network of TSA, if clocks are local to each TSA, we can apply Algorithm 3 to each TSA separately and guarantee that parallel composition of the resultant TSA is a CUB-TA. If there are shared clocks, either we can compute the product of the TSA with shared clocks and then apply Algorithm 3 to the product or we can transform the model so as to avoid shared clocks if possible [32].

## 5 EVALUATION

We developed a tool, named TA@PAT, implementing the above mentioned algorithms for model checking networks of TSA with non-Zenoness assumption against LTL properties. TA@PAT is developed based on the PAT framework [18], with 36K lines of C# code excluding libraries

from the PAT framework. A model in TA@PAT is a network of TSA. In addition, we support features like shared variables and pair-wise channel synchronization (similar to those supported in UPPAAL). A property is in the form of an LTL formula constituted by propositions on shared variables. We adopt the automata-based approach for model checking LTL properties [33], i.e., a Büchi automaton is constructed from negation of the formula and the product of the system and the Büchi automaton is constructed on-the-fly in order to determine whether the property is satisfied or not, while checking whether non-Zenoness is satisfied or not.

In order to evaluate the efficiency of the non-Zenoness checking algorithms in a practical and fair environment, we conducted a systematic comparison by model checking 14 benchmark systems with/without the assumption of non-Zenoness. We collect the models in the UPPAAL distribution as well as models which we can find online or published previously, i.e., Fischer's mutual exclusion algorithm (Fischer), box sorter unit (Box), a simple two doors example (2doors), Lynch-Shavit's algorithm (Lynch), the bridge crossing puzzle (Bridge), a time triggered architecture (TTA), Bang-Olufsen's collision detection protocol (BOCDP), a gear controller (Gear), a collision avoidance protocol for an Ethernet like medium (CAPEM), Wendi-Anantha's communication protocol for wireless microsensor networks (WACP), the railway control system (Railway), the CSMA/CD protocol (CSMA), the TDMA protocol with or without error-tolerance (TDMA_1/TDMA_2) and the fibre distributed data interface protocol (FDDI). The LTL formulae are specified according to different models, e.g., whenever a process makes a request, it will enter the critical section (Fischer model), or whenever a frame is sent, the frame will be destroyed (BOCDP model). All the models and TA@PAT are available online[2]. To be fair, we re-implemented the previous approaches so that all algorithms are programmed in the same programming language (i.e., C#), running on the same platform and built on the same underlying data structure (e.g., the DBM library).

Fig. 10 shows the experimental results, obtained on a server running 64-bit Windows with Intel(R) Core(TM) i7-2600 CPU at 3.40GHz and 8GB RAM. The horizontal axis shows the models. The vertical axis shows the verification time, where 'infinity' means out of memory or more than 2 hours; the columns with different shading represent the verification time for different methods as shown in Fig. 10. Besides, we list the number of states visited (as well as the verification time) during the verification for different approaches, as shown in Table 2, where '–' means the same as 'infinity'. Every model is checked using the approach without the assumption of non-Zenoness (W/o non-Zenoness), the CUB-TA based approach (denoted as CUB), the approach based on adding states in the zone graph (denoted as GZG) and the one based on adding one extra clock (denoted as SNZ). The verification results include 'valid', 'invalid' and 'inv-val' as shown in Table 2, where 'inv-val' means that the property is invalid without non-Zenoness (due to Zeno counterexamples) and valid with non-Zenoness. In such cases, the counterexamples found by the approach without non-Zenoness are Zeno ones.

A number of observations can be made on the verification results. First of all, Zeno counterexamples do occur (e.g., in the models of Bridge, TTA and Gear with 'inv-val', and also in some models with 'invalid' like the FDDI model), which shows that model checking with non-Zenoness is necessary. Furthermore, depending on the model and the non-Zenoness checking algorithm, model checking with non-Zenoness incurs minor to significant computational overhead compared to model checking without non-Zenoness. This suggests that an efficient algorithm for model checking with non-Zenoness can potentially reduce verification time significantly.

The performance of the three non-Zenoness checking approaches varies on different models. In all the 20 cases, the SNZ approach is significantly slower. This is because not only there is one additional clock, but also additional accepting locations must be added into the model so as to make sure one time unit is elapsed during any accepting loop in the zone graph. Notice that the number of additional locations equals the number of accepting locations in the original model, which is the number of locations in the setting of TSA. In all the 20 cases, we can get three observations for the CUB and GZG approaches: (1) the CUB approach won 13 cases (with 9 models of Box, Lynch, BOCDP, Gear, CAPEM, WACP, Railway, CSMA and TDMA), i.e., it is faster and visits less states than the GZG approach; (2) the GZG approach only won 2 cases of the FDDI model; (3) for the other 5 cases (with 4 models of Fischer, 2doors, bridge and TTA), the GZG approach has similar performance as the CUB approach, and the two approaches visit the same number of states. A closer investigation of the models reveals that among the fourteen systems, ten are CUB-TA, whereas the Railway model, the CSMA model, the TDMA model and the FDDI model are not. This suggests that models with non-decreasing upper bounds do exist in practice and, as a result, the CUB-TA based approach could be useful.

For the 13 cases with result 'invalid' or 'inv-val' in observation (1), the CUB approach often performs significantly better. The reason is that whenever it is necessary to re-explore a blocking SCC or deal with zero-checks (i.e., expand the zone graph to build a part of the guessing zone graph rather than only the zone graph) in order to check non-Zenoness, the GZG approach in [15] often suffers, especially when the SCC is large. Notice that zero-checks exist in most of the models, e.g., an urgent location or a committed location freezes the time. This is expected as for CUB-TA, the CUB approach solves the non-Zenoness problem without introducing any clock or state. The performance improvement is not so significant for the Gear model because the SCC is small and the re-exploration and zero-check are done quickly, but the GZG approach does visit more states than the CUB approach.

2. http://www.comp.nus.edu.sg/~pat/zeno

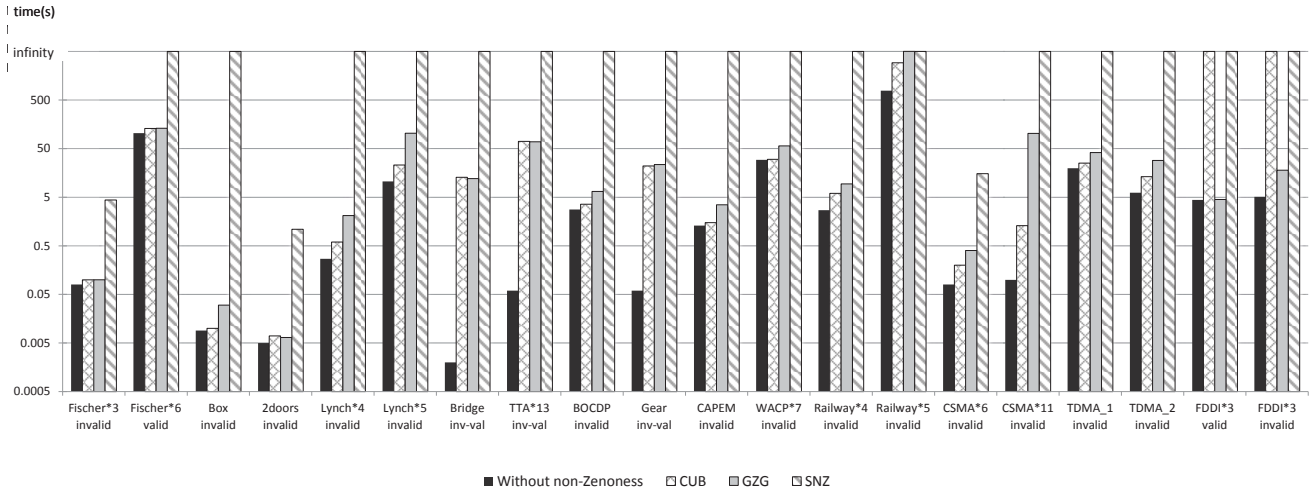



Fig. 10. The time used for three approaches by model checking LTL with non-Zenoness

TABLE 2
The visited states of three approaches by model checking LTL with non-Zenoness

| Model | W/o non-Zenoness | | CUB | | GZG | | SNZ | |
|---|---|---|---|---|---|---|---|---|
| | #States | Time(s) | #States | Time(s) | #States | Time(s) | #States | Time(s) |
| Fischer*3 invalid | 454 | 0.08 | 454 | 0.1 | 454 | 0.1 | 44647 | 4.4 |
| Fischer*6 valid | 1527961 | 104.5 | 1527961 | 129.7 | 1527961 | 130.9 | – | – |
| Box invalid | 494 | 0.01 | 494 | 0.01 | 954 | 0.03 | – | – |
| 2doors invalid | 143 | 0.005 | 143 | 0.007 | 143 | 0.006 | 11744 | 1.1 |
| Lynch*4 invalid | 10877 | 0.3 | 17328 | 0.6 | 47501 | 2.1 | – | – |
| Lynch*5 invalid | 241729 | 10.6 | 407019 | 23.0 | 1327249 | 103.7 | – | – |
| Bridge inv-val | 48 | 0.002 | 497634 | 12.9 | 497634 | 12.1 | – | – |
| TTA*13 inv-val | 464 | 0.06 | 125896 | 70.3 | 125896 | 69.2 | – | – |
| BOCDP invalid | 38955 | 2.8 | 38955 | 3.6 | 100407 | 6.6 | – | – |
| Gear inv-val | 84 | 0.06 | 486476 | 22.1 | 501566 | 23.5 | – | – |
| CAPEM invalid | 17416 | 1.3 | 17416 | 1.5 | 31212 | 3.5 | – | – |
| WACP*7 invalid | 14879 | 29.5 | 14879 | 30.2 | 141979 | 56.9 | – | – |
| Railway*4 invalid | 19527 | 2.7 | 32569 | 6.0 | 116573 | 9.4 | – | – |
| Railway*5 invalid | 389146 | 779.2 | 585573 | 2918.8 | – | – | – | – |
| CSMA*6 invalid | 97 | 0.1 | 689 | 0.2 | 2662 | 0.4 | 65925 | 15.3 |
| CSMA*11 invalid | 232 | 0.1 | 3494 | 1.3 | 349280 | 102.6 | – | – |
| TDMA_1 invalid | 236547 | 19.7 | 236553 | 25.1 | 663922 | 41.5 | – | – |
| TDMA_2 invalid | 235089 | 6.2 | 280786 | 13.3 | 659773 | 28.5 | – | – |
| FDDI*3 valid | 88698 | 4.4 | – | – | 88698 | 4.5 | – | – |
| FDDI*3 invalid | 88571 | 5.1 | – | – | 177113 | 18.0 | – | – |

Among the 9 models of the 13 cases, 3 models are not CUB-TA (Railway, CSMA and TDMA). However, if the number of added locations is small, the CUB approach may still perform better than the GZG approach. For the Railway model, the CSMA model and the TDMA model, the ratios of the number of added locations to the number of original locations are around 17%, 30% and 50% respectively.

As shown in the observation (2), for the 4 models which are not CUB-TA, the GZG approach outperforms CUB approach only on one model, i.e., the FDDI model. It is because 11 extra locations are added for each automaton during the process of transforming the model into an equivalent CUB-TA, which makes the zone graph very large for the CUB approach. The ratio of the number of added locations to the number of original locations is around 137%.

For the observation (3), the GZG approach has similar performance as the CUB approach. This is due to the heuristics proposed in [15], which allows the GZG algorithm to avoid adding states into the zone graph in certain cases. With the GZG approach applied on the 5 cases, for the 'valid' case (Fischer), there is no SCC in the zone graph; for the two 'invalid' cases (Fischer and 2doors), the SCC found for the first time is non-blocking and there is not any zero-check; for the two 'inv-val' cases (Bridge and TTA), the SCC is blocked but the re-exploration finishes quickly without finding any non-blocking SCC, and as a result, zero-check is unnecessary (notice that the re-exploration

does not introduce new states, whereas expanding the zone graph to a guessing zone graph for zero-checks introduces more states). Moreover, all the models in this observation are CUB-TA, therefore no extra location is introduced for the CUB approach. As a result, the GZG and CUB approaches visit the same number of states.

Furthermore, we can see from Table 2 that the CUB approach visited the same number of states with the approach without non-Zenoness, for all CUB-TA models (except for the Lynch, Bridge, TTA and Gear models, since they contain Zeno runs), while the GZG approach and the SNZ approach increase the number of visited states significantly, especially when a property is invalid. This also suggests why the CUB approach performs better. For those models which are not CUB-TA, since extra locations are introduced to transform the model into CUB-TA, the number of visited states increases using the CUB approach, especially so for the FDDI model. However, the number of visited states is still less than that of the GZG approach and the SNZ approach for the Railway, CSMA and TDMA examples, which do not introduce too many extra locations.

Based on the above observations, we develop a heuristic algorithm for choosing the right approach in TA@PAT. Fig. 11 shows the workflow of TA@PAT. Given an input TA, TA@PAT works as follows.

1) Check if the TA is an SNZ (refer to [34] for the algorithm). If the answer is yes, we perform model checking without non-Zenoness. From Fig. 10 and Table 2 we can see that the approach without non-Zenoness has the best performance. If the TA is an SNZ itself, all runs of the TA are guaranteed to be non-Zeno and the approach without non-Zenoness is sufficient.
2) If the TA is not SNZ, check if the TA is an CUB-TA. If the answer is yes, we perform model checking with non-Zenoness using the CUB approach. We can see from the observation (1) and observation (3) that for all the CUB-TA models, CUB approach performs better than or similar to GZG approach. The SNZ approach is not considered since it is always much slower.
3) If the TA is not a CUB-TA, we transform the TA into a CUB-TA. If the number of the added locations is no more than 50% of the original locations in the model, we model check with non-Zenoness using the CUB approach. We conservatively propose the threshold 50% which is the largest ratio from the results of the observation (1). We also make it user configurable so that the users can control which approach to use when the CUB approach has to introduce more locations.
4) Otherwise we model check using the GZG approach.

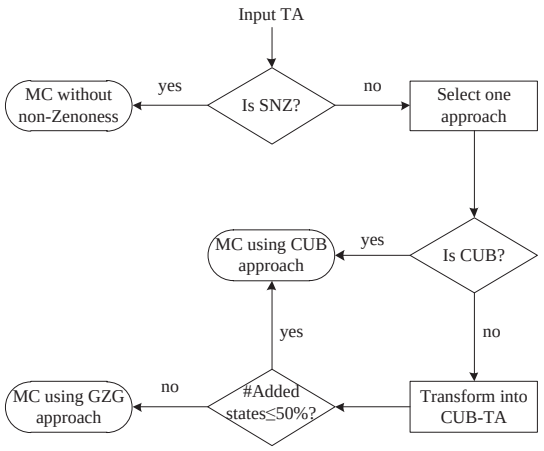



Fig. 11. Workflow of TA@PAT

## 6 Discussion

Our contribution in this work is threefold. Firstly, we show that for CUB-TA, non-Zenoness checking can be solved based on the zone graph only without introducing any extra clock or states. Furthermore, we develop an algorithm to transform an arbitrary Timed Automaton into an equivalent CUB-TA. Secondly, we make a systematic and experimental evaluation on the problem of model checking with the non-Zenoness assumption and compare different approaches in the context of model checking LTL through various benchmark systems. Lastly, we develop a software toolkit integrated with these approaches, which heuristically selects different approaches for different models.

In addition to the above-mentioned approaches on checking non-Zenoness, this work is related to research on non-Zenoness in general. In [13], it has been shown that a Timed Automaton is strongly 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 of the SNZ method is identified in [34] (e.g., instead of checking all structural loops, only some loops are checked). The authors argue that a network is non-Zeno if the product automaton is SNZ, i.e., if every loop in the product automaton is SNZ. The proposed conditions to guarantee absence of Zeno runs in a network include: (a) it suffices to consider loops that correspond to elementary cycles (i.e., cycles with exactly one repeating location), and (b) Zeno runs cannot occur if all non-SNZ loops have at least one observable action, which cannot be matched against any other non-SNZ loop (blocking synchronisation with any SNZ loop guarantees non-Zenoness). Effectively, the work in [34] weakens the requirements imposed in [13], and proves that the composition between an SNZ loop and another non-SNZ loop can yield an SNZ loop in the product automaton.

The analysis in [34] is able to assert absence of Zeno runs for a larger class of specifications, but it assumes a simple Timed Automaton model. In [35], 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 for

the situations when urgent and committed locations, urgent channels, parameters and selections exist. Thus a more comprehensive analysis to deal with additional features introduced by UPPAAL networks is proposed.

However, preventing Zeno runs altogether by construction would be too restrictive for users [35]. 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 [13], [14], 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 [12]. The proposed remedy is the GZG approach. In addition, this work is related to the work on non-Zeno real-time game strategy [31], which however is not based on zone abstraction, whereas our work is on solving a problem on combining zone abstraction and non-zenoness.

In terms of tool support for model checking with non-Zenoness, UPPAAL [3] and KRONOS [4] and RT Spin [36] allow some form of non-Zenoness detection. UPPAAL relies on test automata [37] and leads-to properties. The problem with this approach is that it is sufficient-only. KRONOS supports an expressive language for specifying properties, which allows encoding of a sufficient and necessary condition for non-Zenoness. Checking for non-Zenoness in KRONOS is expensive. The non-Zenoness checking algorithm implemented in RT Spin is unsound [36]. While it is possible to check LTL properties indirectly using existing tools like KRONOS, as far as we know, TA@PAT is the only model checker which supports model checking LTL with the non-Zenoness assumption directly.

## REFERENCES


[1] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[2] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic Model Checking for Real-Time Systems," *Information and Computation*, vol. 111, no. 2, pp. 193–244, 1994.

[3] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a Nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[4] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A Model-Checking Tool for Real-Time Systems," in *CAV*, ser. LNCS, vol. 1427. Springer, 1998, pp. 546–550.

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

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

[7] S. Cattani and M. Z. Kwiatkowska, "A Refinement-based Process Algebra for Timed Automata," *Formal Aspects of Computing*, vol. 17, no. 2, pp. 138–159, 2005.

[8] R. Alur, L. Fix, and T. A. Henzinger, "Event-Clock Automata: A Determinizable Class of Timed Automata," *Theor. Comput. Sci.*, vol. 211, no. 1-2, pp. 253–273, 1999.

[9] J. Ouaknine and J. Worrell, "On the Language Inclusion Problem for Timed Automata: Closing a Decidability Gap," in *LICS*. IEEE Computer Society, 2004, pp. 54–63.

[10] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski, "ECDAR: An Environment for Compositional Design and Analysis of Real Time Systems," in *ATVA*, ser. LNCS, vol. 6252. Springer, 2010, pp. 365–370.

[11] J. S. Dong, P. Hao, S. Qin, J. Sun, and W. Yi, "Timed Automata Patterns," *IEEE Transactions on Software Engineering*, vol. 34, no. 6, pp. 844–859, 2008.

[12] F. Herbreteau, B. Srivathsan, and I. Walukiewicz, "Efficient Emptiness Check for Timed Büchi Automata," *Formal Methods in System Design*, vol. 40, no. 2, pp. 122–146, 2012.

[13] S. Tripakis, "Verifying Progress in Timed Systems," in *ARTS*, ser. LNCS, vol. 1601. Springer, 1999, pp. 299–314.

[14] S. Tripakis, S. Yovine, and A. Bouajjani, "Checking Timed Büchi Automata Emptiness Efficiently," *FMSD*, vol. 26, no. 3, pp. 267–292, 2005.

[15] F. Herbreteau and B. Srivathsan, "Efficient On-the-Fly Emptiness Check for Timed Büchi Automata," in *ATVA*, 2010, pp. 218–232.

[16] J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, and E. André, "Modeling and Verifying Hierachical Real-time Systems using Stateful Timed CSP," *TOSEM*, 2012, to appear.

[17] J. Ouaknine and J. Worrell, "Timed CSP = Closed Timed Safety Automata," *Electr. Notes Theor. Comput. Sci.*, vol. 68, no. 2, 2002.

[18] J. Sun, Y. Liu, J. S. Dong, and J. Pang, "PAT: Towards Flexible Verification under Fairness," in *CAV*, ser. LNCS, vol. 5643, 2009.

[19] W. Yi, P. Pettersson, and M. Daniels, "Automatic Verification of Real-Time Communicating Systems by Constraint-Solving," in *ICFDT*, 1994, pp. 223–238.

[20] T. A. Henzinger, P. Kopke, and H. Wong-Toi, "The Expressive Power of Clocks," in *ICALP*, 1995, pp. 417–428.

[21] D. L. Dill, "Timing Assumptions and Verification of Finite-State Concurrent Systems," in *Automatic Verification Methods for Finite State Systems*, ser. LNCS, vol. 407. Springer, 1989, pp. 197–212.

[22] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi, "Efficient Timed Reachability Analysis Using Clock Difference Diagrams," in *CAV*, ser. LNCS, vol. 1633, 1999, pp. 341–353.

[23] T. G. Rokicki, "Representing and Modeling Digital Circuits," Ph.D. dissertation, Stanford Uni., 1993.

[24] P. Bouyer, "Forward Analysis of Updatable Timed Automata," *Formal Methods in System Design*, vol. 24, no. 3, pp. 281–320, 2004.

[25] S. Tripakis, "Checking Timed Büchi Automata Emptiness on Simulation Graphs," *ACM Trans. Comput. Log.*, vol. 10, no. 3, 2009.

[26] S. Schwoon and J. Esparza, "A Note on On-the-fly Verification Algorithms," in *TACAS*. Springer, 2005, pp. 174–190.

[27] F. Wang, "Efficient Verification of Timed Automata with BDD-Like Data-Structures," in *VMCAI*, 2003, pp. 189–205.

[28] F. Wang, G.-D. Huang, and F. Yu, "TCTL Inevitability Analysis of Dense-Time Systems: From Theory to Engineering," *IEEE Trans. Softw. Eng.*, vol. 32, no. 7, pp. 510–526, 2006.

[29] F. Wang, "Efficient Model-Checking of Dense-Time Systems with Time-Convexity Analysis," in *RTSS*, 2008, pp. 195–205.

[30] F. Wang, L.-W. Yao, and Y.-L. Yang, "Efficient Verification of Distributed Real-time Systems with Broadcasting Behaviors," *Real-Time Syst.*, vol. 47, no. 4, pp. 285–318, 2011.

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

[32] S. Balaguer and T. Chatain, "Avoiding Shared Clocks in Networks of Timed Automata," in *CONCUR*, ser. LNCS, vol. 7454. Springer, 2012, pp. 100–114.

[33] M. Y. Vardi and P. Wolper, "Reasoning About Infinite Computations," *Information & Computation*, vol. 115, no. 1, 1994.

[34] H. Bowman and R. Gómez, "How to Stop Time Stopping," *Formal Aspects of Computing*, vol. 18, no. 4, pp. 459–493, 2006.

[35] R. Gómez and H. Bowman, "Efficient Detection of Zeno Runs in Timed Automata," in *FORMATS*, ser. LNCS, vol. 4763. Springer, 2007, pp. 195–210.

[36] S. Tripakis and C. Courcoubetis, "Extending Promela and Spin for Real Time," in *TACAS*, ser. LNCS, vol. 1055. Springer, 1996, pp. 329–348.

[37] L. Aceto, P. Bouyer, A. Burgueño, and K. G. Larsen, "The Power of Reachability Testing for Timed Automata," *Theoretical Computer Science*, vol. 300, no. 1-3, pp. 411–475, 2003.

**Ting Wang** received her bachelor's degree in software engineering from Zhejiang University of China in 2008. She is currently a PhD student in College of Computer Science and Technology of Zhejiang University. She has studied in Singapore University of Technology and Design (SUTD) as a visiting student in 2012, and also studied in National University of Singapore (NUS) since 2013. Her research interests include formal methods and software engineering, in particular, system verification and model checking.

**Jun Sun** received the bachelor's and PhD degrees in computing science from the National University of Singapore (NUS) in 2002 and 2006, respectively. In 2007, he received the prestigious LEE KUAN YEW postdoctoral fellowship in the School of Computing of NUS. In 2010, he joined the Singapore University of Technology and Design (SUTD) as an assistant professor. He was a visiting scholar at MIT from 2011 to 2012. His research focuses on software engineering and formal methods, in particular, system verification and model checking. He is the cofounder of the PAT model checker.

**Xinyu Wang** received his bachelor's and PhD degrees in computer science from Zhejiang University of China in 2002 and 2007. He was a research assistant in Zhejiang University, during 2002 2007. He is currently an associate professor in the College of Computer Science, Zhejiang University. His research interests include software engineering, formal methods and very large information systems.

**Yang Liu** received the bachelor's of computing degree in 2005 from the National University of Singapore (NUS), the PhD degree in 2010, and continued with postdoctoral work at NUS. Since 2012, he has been with the Nanyang Technological University (NTU) as an assistant professor. His research focuses on software engineering, formal methods, and security. Particularly, he specializes in software verification using model checking techniques. This work led to the development of a state-of-the-art model checker, process analysis toolkit.

**Yuanjie Si** received his bachelor's and PhD degrees in computer science from Zhejiang University of China in 2007 and 2013. He has studied in Singapore University of Technology and Design (SUTD) as a visiting student in 2011. His research interests include software engineering and formal verification, in particular, system verification, model checking and software reliability evaluation techniques.

**Jin Song Dong** received the bachelor's and PhD degrees in computing from the University of Queensland, Australia, in 1992 and 1996, respectively. From 1995 to 1998, he was a research scientist at CSIRO in Australia. Since 1998, he has been at the School of Computing of the National University of Singapore (NUS), where he is currently an associate professor and one of the PhD supervisors at the NUS Graduate School. He is on the editorial board of Formal Aspects of Computing and Innovations in Systems and Software Engineering. His research interests include formal methods, software engineering, pervasive computing, and semantic technologies.

**Xiaohu Yang** received his PhD degree in computer science from Zhejiang University of China in 1993. Since 1994, he has been a faculty member in the College of Computer Science, Zhejiang University. He is currently a professor in Zhejiang University. His research interests include software engineering, formal methods and very large information systems.

**Xiaohong Li** is a professor in School of Computer Science at Tianjin University of China. She received her Computer Science PhD degree from Tianjin University in 2005. Her research interests are software engineering, formal methods and information security. In recent years, she has published more than 40 academic articles. She obtained Tianjin city award of progress of science and technology. She is also a senior member of China Computer Federation (CCF).