Formal Methods in System Design, 26, 267–292, 2005

Checking Timed Büchi Automata Emptiness Efficiently

STAVROS TRIPAKIS Stavros.Tripakis@imag.fr
CNRS–VERIMAG, Centre Équation, 2 avenue de Vignate, 38610 Gières, France

SERGIO YOVINE Sergio.Yovine@imag.fr
CNRS–VERIMAG, Centre Équation, 2 avenue de Vignate, 38610 Gières, France

AHMED BOUAJJANI Ahmed.Bouajjani@liafa.jussieu.fr
LIAFA, Univ. of Paris 7, Case 7014, 2 Place Jussieu, 75251 Paris Cedex 5, France

Abstract. This paper presents an on-the-fly and symbolic technique for efficiently checking timed automata emptiness. It is symbolic because it uses the simulation graph (instead of the region graph). It is on-the-fly because the simulation graph is generated during the test for emptiness. We have implemented a verification tool called PROFOUNDER based on this technique. To our knowledge, PROFOUNDER is the only available tool for checking emptiness of timed Büchi automata. To illustrate the practical interest of our approach, we show the performances of the tool on a non-trivial case study.

Keywords: timed automata, symbolic model-checking, on-the fly verification, verification tools

1. Introduction

Formal methods provide a rigorous framework for modeling and analyzing the behavior of critical systems. Formal specifications of the system's behavior and the requirements, using suitable description languages, permit to formally prove that the requirements are met. The development of efficient software tools supporting the use of formal methods in system design is an active area of research. Many of these tools follow the so-called *model-checking* approach, which is based on the exploration of the system's state-space, yielding a yes/no answer to the verification question "does the system meet its requirements?", and providing a counter-example whenever the answer is no. The main problem with model checking is the so-called *state-explosion problem*, that is, the fact that the size of the state-space of most realistic systems grows prohibitively large with the number of system variables and system components.

Two techniques, among others, have been proven useful in practice for tackling the state-explosion problem: *symbolic* and *on-the-fly* model-checking. Symbolic model-checking algorithms reason in terms of *sets* of states, which are represented implicitly by means of predicates. This is in contrast to *enumerative* algorithms, which reason in terms of *single* states, represented explicitly by listing the values of all variables composing the state. The advantage of symbolic techniques is that the number of states in the state-space is not directly related to the size of the symbolic (predicate) representation, which therefore results in significant gains in space as well as time complexity.

On-the-fly model-checking algorithms check the property *during* the generation of the state-space, as opposed to non-on-the-fly techniques, where the entire state-space needs to be generated before-hand. The advantage of on-the-fly techniques is that sometimes the property is found true or false early enough during the search, with only part of the state-space having been generated.

Many formal frameworks that have been proposed to reason about real-time systems are based on *timed automata* [5]. These automata are equipped with *clocks*, continuous variables used to measure time, ranging over the positive reals. Consequently, the state-space is infinite and cannot be explicitly represented by enumerating all states. Fortunately, there exists a finite partition of the state-space (called the *region graph*) into equivalence classes (called *regions*). The region graph preserves the properties of the infinite state space that can be expressed formally: region-equivalent states satisfy the same properties. Thus, from the point of view of the developer of verification algorithms for timed automata, the region graph is the finest possible finite representation of the state-space. Consequently, we can classify algorithms based on the region graph as enumerative.

Unfortunately, the size of the region graph is exponential in the size of the timed automaton (this blow-up adds to the exponential blow-up of the discrete state space by composing many automata in parallel). In order to tackle this problem, symbolic or on-the-fly approaches have been proposed:

1. A symbolic model-checking algorithm which computes backwards a fixpoint over *unions* of regions rather than individual regions [14].
2. An on-the-fly algorithm which works on the region graph [13].
3. Algorithms working on the quotient graphs of *time-abstracting bisimulations* [4, 22, 28], which can be generated using a *partition refinement* technique [17, 20].

One drawback of the fixpoint technique is that the fixpoint is calculated over the entire set of potential states, which means also states which are unreachable from the set of initial states. Another drawback is that it uses non-convex polyhedra to represent general unions of regions. Non-convex polyhedra do not have an efficient canonical representation, which results in expensive operations. The second technique uses the region graph, thus, it is viable only in case a counter-example is found quickly, so that only a small part of the region graph has to be generated. The third technique also suffers from state explosion, since the quotient graph is usually too big (there is a trade-off between the abstraction power and the class of properties that are preserved). Moreover, refinement is costly, since the same node must generally be refined many times.

Another approach, which is both symbolic and on-the-fly, is based on the so-called *simulation graph*. The nodes of this graph are unions of regions which can be represented in an efficient manner. Moreover, the simulation graph can be generated forward and on-the-fly, using standard depth-first or breadth-first search. Although the simulation graph can be exponentially large in the worst-case, in practice, it is orders of magnitude smaller than the region graph. Until now, the simulation graph has been used only for the verification of simple *safety* properties, which can be reduced to *reachability* [11, 16].

In this paper, we show that the simulation graph can also be used for on-the-fly verification of *liveness* properties, and in particular, in order to check language emptiness of *timed Büchi automata* (TBA). The algorithms we propose can be classified as both symbolic and on-the-fly in the following sense. They are symbolic, because they use the simulation graph (instead of the region graph). They are on-the-fly, because the simulation graph is generated during the test for emptiness. Thus, if the language of the TBA is not empty, a *witness* can be generated as soon as one is found, without having to generate the entire simulation graph. Our contribution is both theoretical and practical.

At the theoretical level, we establish a correspondence between abstract runs (corresponding to cycles in the simulation graph) and concrete runs of the TBA. More precisely, we show that every infinite run of the automaton is *inscribed* in a cycle in the simulation graph and that every cycle contains an inscribed run. The second part of the theorem is non-trivial, since the simulation graph does not have the *pre-stability* property of the region graph, that is, an edge $S \to S'$ between two symbolic states S and S' does not imply that every state in S has a successor in S'.

Having established the above correspondence, we study the problem of acceptance. Timed Büchi automata introduce two types of acceptance conditions, namely, *discrete* (standard Büchi acceptance conditions), as well as *timed* (implicit in the requirement of *time divergence*). In region graph algorithms like the ones in [5, 13] time divergence is reduced to checking some kind of *fairness* or *generalized Büchi acceptance* condition which actually results in even more expensive algorithms.

To avoid the problem of timed acceptance whenever possible, we use the notion of *strongly non-zeno* timed automata, introduced in [26]. Strong non-zenoness ensures that discrete acceptance conditions imply time divergence. We show that the complexity of checking language emptiness for a strongly non-zeno TBA is linear in the size of its simulation graph. A sufficient condition for strong non-zenoness is *structural non-zenoness* [26]. A TBA A is structurally non-zeno if every accepting structural loop of A spends a strictly positive amount of time. Structural non-zenoness holds in practice more often than not. It is both *syntactic* and *compositional* (i.e., preserved by parallel composition), and therefore, it is easy to check it, even for large systems.

In the general case, where strong non-zenoness does not hold, we provide two alternatives for checking emptiness. First, we show that any TBA A can be transformed into a strongly non-zeno TBA $\mathsf{snz}(A)$ by adding one extra clock, such that the language of A is empty iff the language of $\mathsf{snz}(A)$ is empty. Second, we show that for a special class of TBA, namely, automata with *persistent acceptance conditions* (once an accepting state is entered, only accepting states can be visited), emptiness can be checked by examining only the *simple* cycles, that is those where each node appears only once. We also show that exploring simple cycles is not sufficient in the general case. That is, there exist TBA (having non-persistent acceptance conditions) whose language is non-empty, yet no simple cycle in their simulation graph is both accepting and lets time progress.

At the practical level, we justify the interest of our work by analyzing a non-trivial case study. The case study has been treated using a prototype implementation of our techniques in a tool called PROFOUNDER. The case study involves verifying an asynchronous circuit which is a component of the Post Office communication co-processor [23].

2. Timed Büchi automata

Let N denote the set of natural numbers and R the set of non-negative real numbers. Let $\mathcal{X}$ be a finite set of variables taking values in R. An *$\mathcal{X}$-valuation* is a function $\mathbf{v} : \mathcal{X} \to \mathsf{R}$ that assigns to each variable in $\mathcal{X}$ a value in R. $\mathbf{0}$ denotes the valuation assigning 0 to all variables in $\mathcal{X}$. Given a valuation $\mathbf{v}$ and $\delta \in \mathsf{R}$, $\mathbf{v} + \delta$ is defined to be the valuation $\mathbf{v}'$ such that $\mathbf{v}'(x) = \mathbf{v}(x) + \delta$ for all $x \in \mathcal{X}$. Given a valuation $\mathbf{v}$ and $X \subseteq \mathcal{X}$, $\mathbf{v}[X := 0]$ is defined to be the valuation $\mathbf{v}'$ such that $\mathbf{v}'(x) = 0$ if $x \in X$ and $\mathbf{v}'(x) = \mathbf{v}(x)$ otherwise.

An *atomic constraint* on $\mathcal{X}$ is a constraint of one of the forms $x \leq c$, $x < c$, $x - y \leq c$, $x - y < c$, where $x, y \in \mathcal{X}$ and c is an integer. A valuation $\mathbf{v}$ *satisfies* an atomic constraint α, denoted $\mathbf{v} \models \alpha$, if substituting the values of the clocks in the constraint yields a valid inequality. For example, $\mathbf{v} \models x \leq 5$ iff $\mathbf{v}(x) \leq 5$. A boolean expression on atomic constraints defines a set of $\mathcal{X}$-valuations, called an *$\mathcal{X}$-polyhedron*. For example, $x \leq 5 \wedge y > 3$ defines the set of all valuations $\mathbf{v}$ such that $\mathbf{v}(x) \leq 5 \wedge \mathbf{v}(y) > 3$. A conjunction of atomic constraints or negations of atomic constraints defines a *convex* $\mathcal{X}$-polyhedron.[1]

Definition 2.1 (Timed Büchi automata). A *timed Büchi automaton* (TBA) [5] is a tuple $A = (\mathcal{X}, Q, q_0, E, \mathsf{invar}, F)$, where:

- $\mathcal{X}$ is a finite set of clocks.
- Q is a finite set of *discrete states*, $q_0 \in Q$ being the *initial* discrete state.
- $F \subseteq Q$ is a finite set of *accepting states*.
- E is a finite set of *edges* of the form $e = (q, \zeta, X, q')$, where $q, q' \in Q$ are the *source* and *target* discrete states, ζ is a convex $\mathcal{X}$-polyhedron, called the *guard* of e, and $X \subseteq \mathcal{X}$ is a set of clocks to be *reset* upon crossing the edge.
- invar is a function associating with each discrete state q a convex $\mathcal{X}$-polyhedron, called the *invariant* of q.

Given an edge $e = (q, \zeta, X, q')$, we write $\mathsf{source}(e)$, $\mathsf{target}(e)$, $\mathsf{guard}(e)$ and $\mathsf{reset}(e)$ for q, q', ζ and X, respectively. Given a discrete state q, we write $\mathsf{in}(q)$ (resp. $\mathsf{out}(q)$) for the set of edges of the form $(\_, \_, \_, q)$ (resp. $(q, \_, \_, \_)$). We assume that for each $e \in \mathsf{out}(q)$, $\mathsf{guard}(e) \subseteq \mathsf{invar}(q)$.

A *state of* A is a pair $s = (q, \mathbf{v})$, where $q \in Q$ and $\mathbf{v} \in \mathsf{invar}(q)$. We write $\mathsf{discrete}(s)$ to denote q. The *initial state* of A is $s_0 = (q_0, \mathbf{0})$.

An edge $e = (q_1, \zeta, X, q_2)$ can be seen as a (partial) function on states. Given a state $s = (q_1, \mathbf{v})$ such that $\mathbf{v} \in \zeta$ and $\mathbf{v}[X := 0] \in \mathsf{invar}(q_2)$, $e(s)$ is defined to be the state $s' = (q_2, \mathbf{v}[X := 0])$. Whenever $e(s)$ is defined, we say that a *discrete transition* can be taken from s to s'.

A number $\delta \in \mathsf{R}$ can also be seen as a (partial) function on states. Given a state $s = (q, \mathbf{v})$, if $\mathbf{v} + \delta \in \mathsf{invar}(q)$ then $\delta(s)$ is defined to be the state $s' = (q, \mathbf{v} + \delta)$, otherwise $\delta(s)$ is undefined. Whenever $\delta(s)$ is defined, we say that a *time transition* can be taken from s to s'.

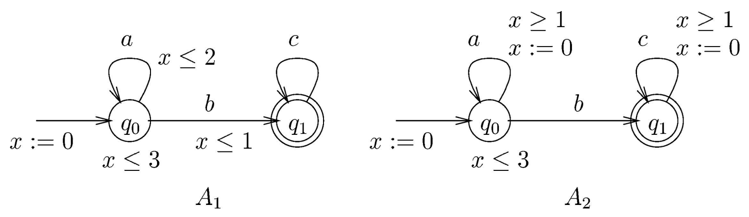


Figure 1. A TBA with zeno runs (left) and a strongly non-zeno TBA (right).

An infinite sequence of pairs $(\delta_0, e_0), (\delta_1, e_1), \dots$, where for all $i = 0, 1, \dots, \delta_i \in \mathsf{R}$ and $e_i \in E$, defines a *run of A starting at state s*, if s is a state of A and the sequence of states $s_0 = s$, $s_{i+1} = e_i(\delta_i(s_i))$ is defined for all $i = 0, 1, \dots$. The run is called *accepting* if there exists an infinite set of indices $J \subseteq \mathsf{N}$, such that for all $i \in J$, $\mathsf{discrete}(s_i) \in F$. The run is called *zeno* if the sequence $\delta_0, \delta_0 + \delta_1, \delta_0 + \delta_1 + \delta_2, \dots$ *converges*, that is, if $\exists \delta \in \mathsf{R}$, $\forall k = 0, 1, \dots, \Sigma_{i=0,\dots,k} \delta_i < \delta$. Otherwise, the run is called *non-zeno*.

Example 2.2. Consider the two TBA shown in figure 1. Circles represent discrete states, double circles represent accepting states, and arrows represent edges. Labels a, b, c refer to edges. A run of A_1 starting at state $(q_0, \mathbf{0})$ is $(0.5, a), (0.25, a), (0.125, a), \dots$: this run is zeno. In fact, any run of A_1 taking a-transitions forever is zeno. On the other hand, the run $(0, b), (1, c), (1, c), \cdots$ of A_1 is non-zeno. Finally, every accepting run of A_2 is non-zeno.

Definition 2.3 (Language and emptiness problem for TBA). The *language* of A, denoted *Lang*(A), is defined to be the set of all non-zeno accepting runs of A starting at the initial state s_0. The *emptiness problem* for A is to check whether *Lang*$(A) = \emptyset$.

The emptiness problem for TBA is known to be PSPACE-complete [5]. More precisely, the worst-case complexity of the problem is linear in the number of discrete states of the automaton, exponential in the number of clocks, and exponential in the encoding of the constants appearing in guards or invariants.This worst-case complexity is inherent to the problem: as shown in [10], both the number of clocks and the magnitude of the constants render PSPACE-hardness independently of each other.

Parallel composition of TBA. In most practical applications, the system to be verified is composed of many components executing in a concurrent fashion. Each of these components can be modeled as an automaton (timed or untimed, with or without acceptance conditions), and a *composition* operator can be used to define a *product* automaton, which captures the concurrent execution of the components. Many choices exist for the definition of parallel composition. We will briefly describe the most popular one for timed automata. We omit the formal details, referring the reader to [30].

The usual parallel composition operator for timed automata is based on the *interleaving* of a set of their discrete transitions and the *synchronization* of another set of discrete transitions.

Interleaving means that only one automaton takes the transition, while the rest do not change state. Synchronization of a set of transitions means that the guards of all corresponding edges must be satisfied, all transitions are taken simultaneously, and all clocks reset in the corresponding edges are reset in parallel. This composition operator, which will be denoted $\|$, corresponds semantically to the intersection of the timed languages that each individual timed automaton defines. The product automaton can be generated on-the-fly, given the component automata and the synchronization sets.

Example 2.4. Two examples of TBA are shown in figure 2. The examples also serve to illustrate how common properties can be verified by reducing the problem to a TBA emptiness problem. Automaton A_1 can be used to verify the so-called *bounded-response* property "every p is followed by a q within at most 3 time units". A_1 expresses the negation of this property, namely, existence of a run where p is followed by a time elapse of more than 3 time units during which q does not hold. A system A satisfies this bounded-response property iff $Lang(A\|A_1) = \emptyset$ (we assume that in the parallel product A and A_1 synchronize on every discrete transition). Automaton A_2 can be used to verify the (untimed) property "p holds forever after some point on". A_2 expresses the negation of this property, namely, existence of a run where $\neg p$ is true infinitely often. A system A satisfies the above untimed property iff $Lang(A\|A_2) = \emptyset$.

Definition 2.5 (*Strong non-zenoness*). A TBA A is called *strongly non-zeno* if all accepting runs starting at the initial state of A are non-zeno.

A *structural loop* of a TBA A is a sequence of distinct edges $e_1 \cdots e_m$ such that $\mathsf{target}(e_i) = \mathsf{source}(e_{i+1})$, for all $i = 1, \ldots, m$ (the addition $i+1$ is modulo m). We say that the structural loop is *accepting* if there exists some $i = 1, \ldots, m$ such that $\mathsf{target}(e_i)$ is an accepting state. We say that the structural loop *spends time* if there exist a clock x of A and indices $0 \le i, j \le m$ such that:

1. x is reset in step i, that is, $x \in \mathsf{reset}(e_i)$, and
2. x is bounded from below in step j, that is, $(x < 1) \cap \mathsf{guard}(e_j) = \emptyset$.

Definition 2.6 (*Structural non-zenoness*). We say that a TBA A is *structurally non-zeno* if every accepting structural loop of A spends time.

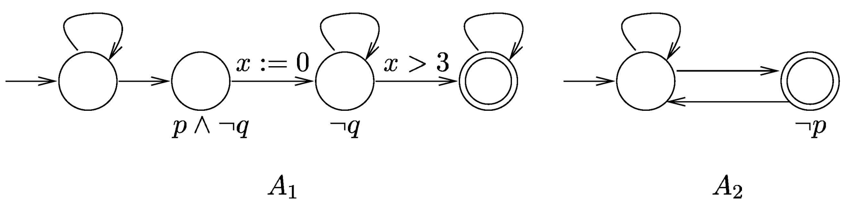


Figure 2. Two timed Büchi automata.

For example, in figure 1, automaton A_1 is not structurally non-zeno, while automaton A_2 is. A_2 would not be structurally non-zeno if any of the guards $x \geq 1$ was missing.

Lemma 2.7. *If A is structurally non-zeno then A is strongly non-zeno.*

Proof: Let $\rho = (\delta_0, e_0)(\delta_1, e_1)\cdots$ be an accepting run of A starting from $s_0 = (q_0, \mathbf{0})$. Since A has only a finite number of edges, there exist some $i_1, i_2, \ldots, i_m$ such that $e_{i_1} e_{i_2} \ldots e_{i_m}$ form a structural loop and ρ takes infinitely often every transition e_{i_j}. Since ρ is accepting, the structural loop must also be accepting. Thus, by hypothesis, the structural loop spends time. That is, there exist a clock x and indices $j_1, j_2 \in \{i_1, i_2, \ldots, i_m\}$ such that $x \in \mathsf{reset}(e_{j_1})$ and $(x < 1) \cap \mathsf{guard}(e_{j_2}) = \emptyset$. Now, each time ρ takes an e_{j_1}-transition, clock x is reset to 0. The next time ρ takes an e_{j_2}-transition, at least 1 time unit has passed, since x must be greater or equal to 1 for e_{j_2} to be taken. Since e_{j_1}- and e_{j_2}-transitions are taken infinitely often, an infinite number of one-unit delays are accumulated, thus ρ is non-zeno. Hence, A is strongly non-zeno. □

Remark 2.8. Structural non-zenoness is compositional, in the sense that, if automata $A_1, \ldots, A_n$ are structurally non-zeno, then so is their composition, $A_1 \| \cdots \| A_n$. This result is formalized and proven in [25]. The result holds also in case some of the components A_i are untimed (untimed components can be considered structurally non-zeno by convention).

Theorem 2.9. *Any TBA A can be transformed into a strongly non-zeno TBA* $\mathsf{snz}(A)$, *such that* $Lang(A) = \emptyset$ *iff* $Lang(\mathsf{snz}(A)) = \emptyset$.

Proof: The transformation is depicted in figure 3. Let $\mathcal{X}$ be the set of clocks of A and t be a new clock, not in $\mathcal{X}$. The set of clocks of $\mathsf{snz}(A)$ will be $\mathcal{X} \cup \{t\}$. Let q be an accepting discrete state of A. Let $\mathsf{in}(q) = \{e_1, \ldots, e_m\}$ and let $\mathsf{source}(e_i) = q_i$, $\mathsf{guard}(e_i) = \zeta_i$ and $\mathsf{reset}(e_i) = X_i$, for $i = 1, \ldots, m$. For each such situation, $\mathsf{snz}(A)$ will contain the following:

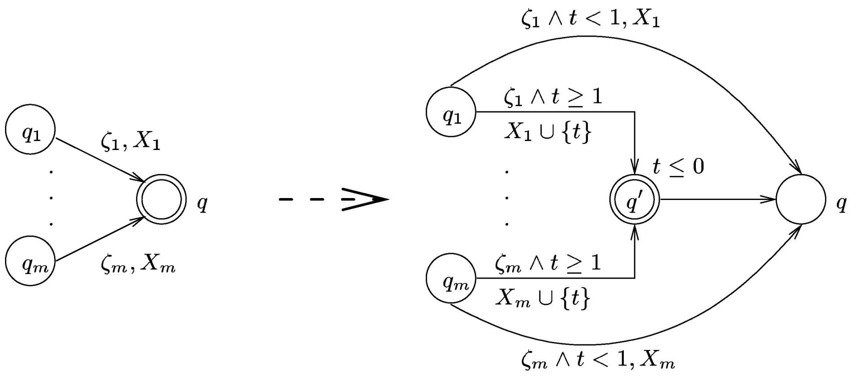

q_1
ζ_1, X_1
q
q_m
ζ_m, X_m
$\zeta_1 \wedge t < 1, X_1$
q_1
$\zeta_1 \wedge t \geq 1$
$X_1 \cup \{t\}$
$t \leq 0$
q'
q
q_m
$\zeta_m \wedge t \geq 1$
$X_m \cup \{t\}$
$\zeta_m \wedge t < 1, X_m$

Figure 3. Transforming a TBA into a strongly non-zeno TBA.

- a new accepting state q', with invariant $\mathsf{invar}(q') = t \leq 0$,
- m new edges $e'_i = (q_i, \zeta_i \wedge t \geq 1, X_i \cup \{t\}, q')$, for $i = 1, \ldots, m$,
- m new edges $e''_i = (q_i, \zeta_i \wedge t < 1, X_i, q)$, for $i = 1, \ldots, m$,
- a new edge $e = (q', \mathsf{true}, \emptyset, q)$.

Moreover:

- q will not be accepting in $\mathsf{snz}(A)$,
- the edges $e_1, \ldots, e_m$ will not exist in $\mathsf{snz}(A)$.

Now, suppose $\rho \in \mathit{Lang}(A)$, $\rho = (\delta_0, e_0), (\delta_1, e_1), \ldots$. By definition, ρ is accepting and non-zeno. We build a run ρ' of $\mathsf{snz}(A)$ as follows. Let $i \geq 0$ be the first index such that e_i is an edge leading to an accepting state of A. The run ρ' will be identical to ρ up to point i. The pair (δ_i, e_i) will be replaced by: either (δ_i, e''_i), if $\Sigma_{j \leq i} \delta_j < 1$; or $(\delta_i, e'_i), (0, e)$, if $\Sigma_{j \leq i} \delta_j \geq 1$. Indeed, notice that t has not been reset up to point i, therefore its value right before e_i is equal to $\Sigma_{j \leq i} \delta_j$. If this value is smaller than 1, then e''_i can be taken, otherwise, e'_i and e can be taken. Since time does not pass between e'_i and e, the values of all clocks of A after e are the same as after e_i. Therefore, the remaining transitions of ρ are still possible and the construction can be continued *ad infinitum*. The resulting run ρ' will be non-zeno, since it contains the same time-passing transitions δ_i as ρ. It will also be accepting, since edges e_i which lead to an accepting state have to appear infinitely often in ρ. These edges have to be replaced infinitely many times by e'_i and e in ρ', otherwise, t must remain bounded by 1 without being reset, which would contradict the hypothesis that ρ is non-zeno.

In the other direction, suppose there is an accepting run ρ' of $\mathsf{snz}(A)$. Since ρ' visits accepting states of $\mathsf{snz}(A)$ infinitely often, it must take edges of the form e'_i and e infinitely often, therefore it is non-zeno, which means that $\mathsf{snz}(A)$ is strongly non-zeno. Moreover, since no time is allowed to pass in states q'_i (because of the invariant $t \leq 0$), ρ' can be transformed into a run ρ of A by simply replacing e'_i, e and e''_i edges by the corresponding e_i edges. Run ρ will also be accepting and non-zeno, since it will contain the same time-passing transitions δ_i as ρ'. □

Notice that the transformed automaton, $\mathsf{snz}(A)$, has one more clock than A. Also note that $\mathsf{snz}(A)$ can be easily generated on-the-fly, even in the case where A itself is generated on-the-fly, as the parallel composition of many components.

We distinguish a sub-class of TBA based on the structure of the acceptance conditions.

Definition 2.10 (*Persistent acceptance conditions*). We say that a TBA A with set of accepting discrete states F has *persistent acceptance conditions* if $\forall q \in F$, $\forall e \in \mathsf{out}(q)$, $\mathsf{target}(e) \in F$.

The above condition says that once A enters F it never exits. TBA with persistent acceptance conditions are interesting, because checking emptiness of such automata is easier than in the general case. TBA with persistent acceptance conditions arise often in practice, as the following example shows.

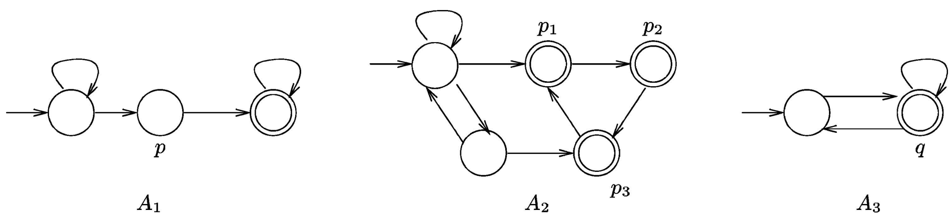


Figure 4. TBA with persistent (A_1 and A_2) and non-persistent (A_3) acceptance conditions.

Example 2.11. Automata A_1 and A_2 of figure 4 have persistent acceptance conditions, whereas automaton A_3 has non-persistent acceptance conditions. A_1 can be used to check the reachability property "eventually p". A_2 can be used to check the property "after some point on, p_1, p_2, p_3 alternate, starting from p_1 or p_3". Other properties, such as the bounded-response property expressed by automaton A_1 of figure 2, are also expressible by TBA with persistent acceptance conditions.

3. The simulation graph

In this section we define the simulation graph essentially as an abstraction of the region graph. First, we recall the definition of the region graph and its properties of interest.

3.1. *The region graph*

The region graph is the finite quotient of the infinite state-space of a timed automaton, with respect to the so-called *region equivalence* [2, 3, 5]. Although finite, the region graph preserves many interesting properties of the infinite state-space, such as *linear-time* and *branching-time* properties [15], expressed, respectively, as TBA emptiness problems or using logics such as TCTL [3].[2]

Consider a TBA A with set of clocks $\mathcal{X}$ and let $(q, \mathbf{v})$ and $(q, \mathbf{v}')$ be two states of A. Let c be a natural constant. The states $(q, \mathbf{v})$ and $(q', \mathbf{v}')$ are *region equivalent*, denoted $(q, \mathbf{v}) \simeq_c (q', \mathbf{v}')$, if they satisfy the following conditions:

1. $q = q'$,
2. for any constraint $\alpha \in \bigcup_{x\in\mathcal{X}}(\bigcup_{i=0,\ldots,c-1}\{x = i, i < x < i+1\} \cup \{x > c\})$, $\mathbf{v} \models \alpha$ iff $\mathbf{v}' \models \alpha$,
3. for any constraint $\beta \in \bigcup_{x,y\in\mathcal{X}}(\bigcup_{i=0,\ldots,c-1}\{x - y = i, y - x = i, i < x - y < i+1, i < y - x < i+1\} \cup \{x - y > c, y - x > c\})$, $\mathbf{v} \models \beta$ iff $\mathbf{v}' \models \beta$.

It can be checked that $\simeq_c$ is an equivalence relation, for any c. The equivalence classes induced by $\simeq_c$ are called *regions*. It can be seen that, for a given timed automaton, the number of regions is finite and in the order of $O(m \cdot c^n)$, where m is the number of discrete

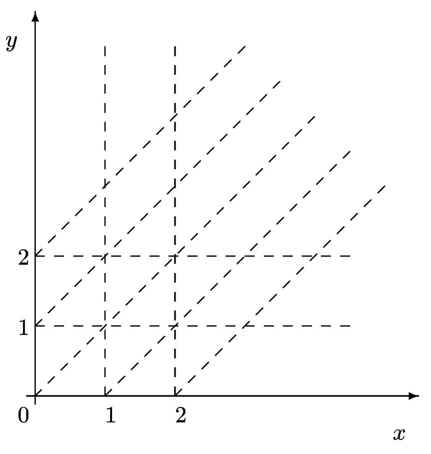


Figure 5. A partition of the clock space into 78 regions.

states and n the number of clocks of the automaton. Although many of these regions may not be reachable, in general, the number of reachable ones is still too large in realistic applications. As an example, the region space for two clocks x, y and $c = 2$ is shown in figure 5.

Assuming that c is the greatest constant appearing in a guard or invariant of the timed automaton, $\simeq_c$ induces a finite graph, called the *region graph*. The nodes of the region graph are regions. The edges are of two types:

- Time-passing edges: $r \xrightarrow{\epsilon} r'$, if there exist $s \in r$ and $\delta \in \mathsf{R}$ such that $\delta(s) \in r'$.
- Discrete-jump edges: $r \xrightarrow{e} r'$, if $e \in E$ and there exists $s \in r$ such that $e(s) \in r'$.

Notice that the $\xrightarrow{\epsilon}$ is *reflexive* ($r \xrightarrow{\epsilon} r$) and *transitive* (if $r \xrightarrow{\epsilon} r'$ and $r' \xrightarrow{\epsilon} r''$ then $r \xrightarrow{\epsilon} r''$).

The essential property of the region graph is *pre-stability*, namely, the fact that for each region r:

- if $r \xrightarrow{e} r'$ is an edge in the region graph, then for each state $s \in r$, $e(s) \in r'$,
- if $r \xrightarrow{\epsilon} r'$ is an edge in the region graph, then for each state $s \in r$, there exists $\delta \in \mathsf{R}$, such that $\delta(s) \in r'$.

Thanks to pre-stability, an infinite run can be easily extracted from every infinite path $r_0 \to r_1 \to \cdots$ in the region graph, by choosing some state $s_0 \in r_0$, then letting s_1 be a successor of s_0 in r_1, and so on, *ad infinitum.*

An infinite path $\pi = r_0 \to r_1 \to \cdots$ in the region graph of A is called *accepting* if there is an infinite set of indices $J \subseteq \mathsf{N}$, such that for all $i \in J$, for all $(q_i, \mathbf{v}_i) \in r_i$, q_i is accepting.

Definition 3.1 (*Progressive paths in the region graph*). An infinite path $\pi = r_0 \to r_1 \to \cdots$ in the region graph of A is called *progressive*[3] if for each clock x

- either x is reset and grows strictly positive infinitely often in π, that is, $\forall i,\ \exists j, k,\ (k > j > i) \wedge (r_j \models x = 0) \wedge (r_k \models x > 0)$,
- or x remains unbounded in π after some point on, that is, $\exists i,\ \forall j > i,\ r_j \models x > c$,

where $r \models \alpha$ is defined as $\forall (q, \mathbf{v}) \in r, \mathbf{v} \models \alpha$ (notice that, by definition of the region equivalence, this is equivalent to $\exists (q, \mathbf{v}) \in r, \mathbf{v} \models \alpha$).

Recall that c in the definition above is the greatest constant appearing in a guard or invariant of A.

It is easy to see that a non-progressive path contains only zeno runs. The converse is also shown to be true in [2], that is, a progressive path contains non-zeno runs (it might contain zeno runs as well). Therefore, we have the following result:

Theorem 3.2 [2]. *Lang*(A) *is non-empty iff there exists an accepting progressive infinite path in the region graph of* A.

In other words, checking emptiness of a TBA A can be reduced to a problem of *model checking* the following *linear temporal logic* (LTL) [21] formula on the region graph of A:

$$\Box\Diamond\, \mathit{accepting} \ \wedge\ \bigwedge_{x \in \mathcal{X}} (\Diamond\Box x > c) \ \vee\ (\Box\Diamond\, (x = 0 \wedge \Diamond x > 0))$$

where $x = 0$, $x > c$ and $\mathsf{accepting}$ can be seen as atomic propositions labeling the nodes of the region graph. The complexity of LTL model checking is known to be linear in the size of the model and exponential in the size of the formula [18].

3.2. The simulation graph

Consider a TBA $A = (\mathcal{X}, Q, q_0, E, \mathsf{invar}, F)$. A *symbolic state* S is a finite set of regions $r_i = (q, \zeta_i)$, $i = 1, \ldots, k$, all having the same discrete state q. We will sometimes denote S by (q, ζ), where $\zeta = \cup\{\zeta_1, \ldots, \zeta_k\}$.

Given a symbolic state S, let $e(S)$ be the set of all regions r' for which there exists $r \in S$ such that $r \xrightarrow{e} r'$. Similarly, let $\epsilon(S)$ be the set of all regions r' for which there exists $r \in S$ such that $r \xrightarrow{\epsilon} r'$.

Given an edge $e \in E$ and a symbolic state S, we define

$$\mathsf{post}(S, e) = \epsilon(e(S)).$$

Definition 3.3 (*Simulation graph*). The *simulation graph* of a TBA A, denoted $SG(A)$, is a graph whose nodes are non-empty symbolic states of A and edges represent post operations. More precisely, the set of nodes $\mathcal{S}$ of $SG(A)$ is defined to be the least set of symbolic states of A, such that:

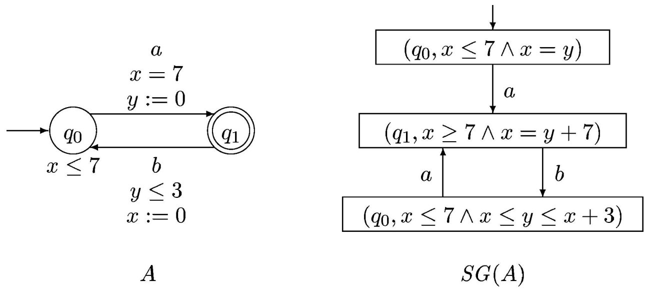

a
$x = 7$
$y := 0$
q_0
q_1
$x \leq 7$
b
$y \leq 3$
$x := 0$
A
$(q_0, x \leq 7 \wedge x = y)$
a
$(q_1, x \geq 7 \wedge x = y + 7)$
a
b
$(q_0, x \leq 7 \wedge x \leq y \leq x + 3)$
SG(A)

Figure 6. A TBA and its simulation graph.

1. $(q_0, \epsilon(\{\mathbf{0}\})) \in \mathcal{S}$ is the initial node of $SG(A)$),
2. if $e \in E$, $S \in \mathcal{S}$ and $S' = \mathsf{post}(S, e)$ is non-empty, then $S' \in \mathcal{S}$.

The set of edges of $SG(A)$ is defined as follows. $SG(A)$ has an edge $S \xrightarrow{e} S'$ iff $S, S' \in \mathcal{S}$ and $S' = \mathsf{post}(S, e)$.

Since the number of regions is finite and each node of the simulation graph is a union of regions, the simulation graph is also finite.

An example of a TBA and its simulation graph is shown in figure 6. This simulation graph was automatically generated using the tool KRONOS [11].

4. Properties of the simulation graph

Theorem 3.2 is based on the correspondence between runs of a TBA and paths of its region graph: each run is contained in a unique path, and each path is guaranteed to contain at least one run. The objective of this section is to "lift" this correspondence to the simulation graph. We then prove a number of results similar to Theorem 3.2, distinguishing classes of TBA where checking emptiness becomes simpler.

The essential property which induces the correspondence of runs and paths in the case of the region graph is pre-stability. However, pre-stability does not hold in the simulation graph. Indeed, if $S \xrightarrow{e} S'$ is an edge of the simulation graph, then there might exist a region $r \in S$ such that r has no successor in S'. For example, let $S = (q, x \leq 2)$ and e be an edge with $\mathsf{guard}(e) = x \leq 1$. Then, $\mathsf{post}(S, e) = (\mathsf{target}(e), x \leq 1)$, however, no state $(q, \mathbf{v}) \in S$ with $\mathbf{v}(x) > 1$ can take the transition e.

Even though the simulation graph is not pre-stable, we can still prove that each run of the TBA is contained in a simulation-graph path, and that each simulation-graph path is guaranteed to contain a run. We first introduce some concepts that will lead us to this result.

By definition, the simulation graph is *post-stable*. That is, if $S \xrightarrow{e} S'$ is an edge in the simulation graph, then: (a) for every region $r' \in S'$, there exists a region $r \in S$ and a region-graph path from r to r'; and (b) for every region $r \in S$, every successor of r by the relation $\xrightarrow{e}\xrightarrow{\epsilon}$ is in S'.

We say that a region-graph path $r_0 \xrightarrow{e_0}\xrightarrow{\epsilon} r_1 \xrightarrow{e_1}\xrightarrow{\epsilon} \cdots$ is *inscribed* in an infinite sequence of symbolic states $\pi = S_0 \xrightarrow{e_0} S_1 \xrightarrow{e_1} \cdots$, if for all $i = 0, 1, \ldots, r_i \in S_i$.

Lemma 4.1. *Every infinite path of the region graph is inscribed in an infinite path of the simulation graph.*

Proof: Let $r_0 \xrightarrow{e_0}\xrightarrow{\epsilon} r_1 \xrightarrow{e_1}\xrightarrow{\epsilon} \cdots$ be a path of the region graph. By definition, $r_0 \in S_0$. By post-stability, for all $i = 0, 1, \ldots$, since $r_i \xrightarrow{e_i}\xrightarrow{\epsilon} r_{i+1}$, if $r_i \in S_i$ then $r_{i+1} \in S_{i+1}$. □

Let $\pi = S_0 \xrightarrow{e_0} S_1 \xrightarrow{e_1} \cdots$ be an infinite path in $SG(A)$, where $S_i = (q_i, \zeta_i)$. We say that π is *accepting* if there are infinitely many $i \geq 0$ such that q_i is accepting. We say that π is *ultimately periodic* if there exist $i \geq 0$, $l \geq 1$, such that for all $j \geq 0$, $S_{i+j} = S_{i+j \bmod l}$. This means that π consists of a finite prefix $S_0 \xrightarrow{e_0} \cdots S_{i-1} \xrightarrow{e_{i-1}}$, followed by the "infinite unfolding" of a *cycle* $S_i \xrightarrow{e_i} \cdots S_{i+l-1} \xrightarrow{e_{i+l-1}} S_i$. The node S_i is called the *root* of the cycle and l is its *length*. The cycle is called *simple* if for all $0 \leq j \neq k < l$, $S_{i+j} \neq S_{i+k}$, that is, the cycle does not visit the same node twice.

Let $\pi = S_0 \xrightarrow{e_0} S_1 \xrightarrow{e_1} \cdots$. A *sub-path* of π is an infinite sequence $\pi' = S'_0 \xrightarrow{e_0} S'_1 \xrightarrow{e_1} \cdots$, such that, for all $i \geq 0$, $S'_i \neq \emptyset$ and $S'_i \subseteq S_i$. We say that π' is *pre-stable* if for all $i \geq 0$, for all $r \in S'_i$, there exists $r' \in S'_{i+1}$ such that $r \xrightarrow{e_i}\xrightarrow{\epsilon} r'$. A sub-path π' of π is called *maximal* if for any other sub-path π'' of π, with $\pi'' = S''_0 \xrightarrow{e_0} S''_1 \xrightarrow{e_1} \cdots$, we have: $\forall i \geq 0$, $S''_i \subseteq S'_i$.

Lemma 4.2. *For every pre-stable sub-path π' of a path in the simulation graph, there is a region-graph path inscribed in π'.*

Proof: Let $\pi' = S'_0 \xrightarrow{e_0} S'_1 \xrightarrow{e_1} \cdots$. Pick some region $r_0 \in S'_0$. Since π' is pre-stable, there exists $r_1 \in S'_1$ such that $r \xrightarrow{e_0}\xrightarrow{\epsilon} r_1$. Similarly, we can find $r_2 \in S'_2$ such that $r_1 \xrightarrow{e_1}\xrightarrow{\epsilon} r_2$. We can continue this process *ad infinitum*, building a region-graph path. □

Lemma 4.3. *Every ultimately periodic infinite path π in the simulation graph has a unique maximal pre-stable sub-path π'. The latter contains a region-graph cycle. All infinite region-graph paths inscribed in π are also inscribed in π'.*

We first illustrate the idea of the proof of the above lemma using figure 7. The figure shows a simple simulation-graph cycle which generates the ultimately periodic infinite path

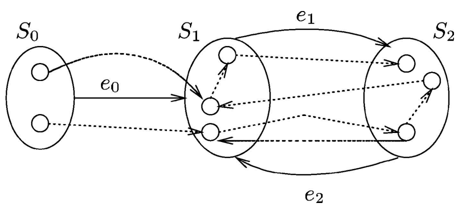


Figure 7. Every simulation graph cycle contains a pre-stable sub-cycle.

$\pi = S_0 \xrightarrow{e_0} S_1 \xrightarrow{e_1} S_2 \xrightarrow{e_2} S_1 \xrightarrow{e_1} \ldots$. The large ellipses represent symbolic states S_0, S_1, S_2, while the small circles represent regions. Solid arrows represent edges of the simulation graph, while dotted arrows represent edges of the region graph.

The proof is based on the following idea. Pick some arbitrary region $r_2^0 \subseteq S_2$. By post-stability of the simulation graph, there exists $r_1^0 \subseteq S_1$ and a path in the region graph from r_1^0 to r_2^0. Similarly, there exists $r_2^1 \subseteq S_2$ and a region-graph path from r_2^1 to r_1^0. We can continue going backwards in the same way, until we find some r_2^j which is the same as r_2^i, for $i < j$ (this is bound to happen, since the number of regions included in a symbolic state is finite). At this point, we have found a cycle in the region graph. Since the region graph is pre-stable and a cycle defines an infinite path, we have found a pre-stable infinite sub-path of π. We now conduct the proof in the general case.

Proof: Let $\pi = S_0 \xrightarrow{e_0} \cdots \xrightarrow{e_{i-1}} S_i \xrightarrow{e_i} \cdots S_{i+l} \xrightarrow{e_{i+l}} S_i \xrightarrow{e_i} \cdots$ be an ultimately periodic infinite path in the simulation graph. S_i is the root of the corresponding cycle and l is the cycle's length. We know that $S_{j+1} = \mathsf{post}(S_j, e_j)$, for all $j = i, \ldots, i + l$. First, we pick some arbitrary region $r_{i+l}^0 \subseteq S_{i+l}$. Then, using post-stability, we find regions $r_{i+l-1}^0 \subseteq S_{i+l-1}$, ..., $r_i^0 \subseteq S_i$, r_{i+l}^1, such that there is a region-graph path from r_{i+l}^1 to r_i^0, to r_{i+1}^0, and eventually to r_{i+l-1}^0. We continue the same way, finding r_{i+l}^2, r_{i+l}^3, and so on, until we find $r_{i+l}^m = r_{i+l}^k$, for some $m > k$ (this is bound to happen, since the number of regions contained in a symbolic state is finite).

The path from r_{i+l}^k to r_{i+l}^m defines a cycle in the region graph, and consequently, an infinite path in the region graph. In turn, an infinite path in the region graph defines an infinite path π' in the simulation graph as follows: for every finite sequence $r \xrightarrow{e} r_1 \xrightarrow{\epsilon} r_2 \xrightarrow{\epsilon} \cdots \xrightarrow{\epsilon} r_m \xrightarrow{e'}$ in the cycle, group together all regions $r_1, r_2, \ldots, r_m$ into a single symbolic state. By construction, π' is a non-empty sub-path of π. Since the region graph is pre-stable, π' is pre-stable. That is, we have found an infinite pre-stable non-empty sub-path of π. We can extend π' to a maximal sub-path by adding to every symbolic state as many regions as possible, while preserving pre-stability. The maximal sub-path is unique since pre- and post-stability are preserved by union, that is, if $S_1 \xrightarrow{e} S_1'$ and $S_2 \xrightarrow{e} S_2'$ are both pre-stable and post-stable, then $S_1 \cup S_2 \xrightarrow{e} S_1' \cup S_2'$ is also pre-stable and post-stable. □

Theorem 4.4. *Let A be a strongly non-zeno TBA. Lang$(A) \neq \emptyset$ iff there exists a simple accepting cycle in the simulation graph of A.*

Proof: Suppose there is a simple accepting cycle in $SG(A)$. This cycle defines an infinite accepting ultimately periodic path π. By Lemma 4.3, π has a pre-stable sub-path π'. By Lemma 4.2, there is a region-graph path λ inscribed in π'. Since π' is accepting, λ is also accepting. Since A is strongly non-zeno, λ is progressive. By Theorem 3.2, $Lang(A) \neq \emptyset$.

In the other direction, assume $Lang(A) \neq \emptyset$. By Theorem 3.2, there exists a region-graph infinite accepting progressive path λ. By Lemma 4.1, λ is inscribed in an infinite accepting path π in $SG(A)$. Since $SG(A)$ is finite, π must visit infinitely often a set of nodes in a strongly connected component C of $SG(A)$. Since π is accepting, C contains at least one accepting node S. From graph theory, we know that we can find a simple cycle in C which visits S, that is, a simple accepting cycle. □

Theorem 4.4 takes care of the case of strongly non-zeno TBA. In the next section, we use this result to show how emptiness of a strongly non-zeno TBA can be checked in time and space linear in the size of the simulation graph.

We now turn to the case of checking emptiness of a TBA A which is not strongly non-zeno. One possibility is to transform A into a strongly non-zeno TBA $\mathsf{snz}(A)$, using Theorem 2.9, and then check whether $\mathit{Lang}(\mathsf{snz}(A)) = \emptyset$. However, $\mathsf{snz}(A)$ contains one more clock than A, which means that $\mathit{SG}(\mathsf{snz}(A))$ will be in general larger than $\mathit{SG}(A)$. In the rest of this section, we explore other possibilities, working directly with $\mathit{SG}(A)$. As we shall see, this is significantly more difficult than the strongly non-zeno case. The reason is that infinite paths in the simulation graph must satisfy two types of acceptance: "discrete" acceptance (visiting accepting states infinitely often), as well as "time" acceptance (letting time diverge).[4] We will show that there exist cases where simple cycles in the simulation graph cannot capture both types of acceptance. In other words, there exist automata whose simulation graph contains accepting cycles which let time diverge, but no such cycle is simple.

First, we need to identify conditions on paths and cycles in the simulation graph which capture time divergence.

Let S be a symbolic state and $x \in \mathcal{X}$ a clock. We define

$$\mathsf{nonzero}(x, S) \overset{\text{def}}{=} \exists r \in S, r \models x > 0,$$
$$\mathsf{unbounded}(x, S) \overset{\text{def}}{=} \exists r \in S, r \models x > c.$$

Definition 4.5 (*Progressive paths in the simulation graph*). Let π be an infinite path in the simulation graph and let $\pi' = S_0 \overset{e_0}{\rightarrow} S_1 \cdots$ be the maximal pre-stable sub-path of π. The path π is called *progressive* if for each clock $x \in \mathcal{X}$

- either x is reset and grows strictly positive infinitely often in π', that is, $\forall i,\ \exists j, k,\ k > j > i \wedge x \in \mathsf{reset}(e_j) \wedge \mathsf{nonzero}(x, S_k)$,
- or x remains unbounded in π' from some point on, that is, $\exists i,\ \forall j > i,\ \mathsf{unbounded}(x, S_j)$.

Since a cycle defines an infinite path, the notions of accepting and progressive paths naturally extend to accepting and progressive cycles.

Remark 4.6. Definition 4.5 is different from Definition 3.1: the former refers to the maximal pre-stable sub-path of a path π, rather than π itself. This is because, in general, π does not contain enough information to check time divergence. To see why, consider the automata A_1 and A_2 of figure 8. As can be seen in the figure, both automata generate identical simulation graphs. However, only the pre-stable cycle of A_2 satisfies the condition for progressiveness.

Lemma 4.7. *Let π be an ultimately periodic infinite path in the simulation graph.*

(1) *If π is progressive, then there exists an infinite progressive region-graph path inscribed in π.*

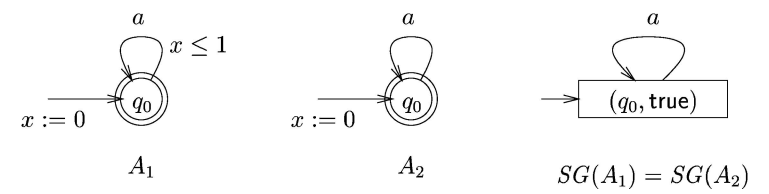


Figure 8. Two TBA.

(2) *If π is not progressive, then no region-graph path inscribed in π is progressive.*

Proof: (1) By Lemma 4.3, π has a maximal pre-stable sub-path $\pi' = S_0 \xrightarrow{e_0} S_1 \xrightarrow{e_1} \cdots$. We will construct a progressive region-graph path λ inscribed in π'. Let $X \subseteq \mathcal{X}$ be the set of clocks reset infinitely often in π' and let $Y = \mathcal{X} \setminus X$. By Definition 4.5: (a) $\forall x \in X, \forall i, \exists j, k, k > j > i \wedge x \in \mathsf{reset}(e_j) \wedge \exists r \in S_k, r \models x > 0$ and (b) $\forall y \in Y, \exists i, \forall j > i, \exists r \in S_j, r \models y > c$. If Y is empty, let r_0 be any region in S_0, otherwise, we choose r_0 as follows. Let $y_0 \in Y$ be one of the clocks in Y which is reset last in π'. If no clock in Y is ever reset in π', any clock in Y can be chosen, since all clocks start by being reset. By condition (b) above, there exists i_0 and $r_0 \in S_{i_0+1}$ such that $r_0 \models y_0 > c$. Since y_0 is reset last, we have $\forall y \in Y, r_0 \models y \geq y_0$. Thus, $\forall y \in Y, r_0 \models y > c$. Since no clock in y is ever reset after i_0, for any region-graph path starting at r_0, all clocks in Y remain unbounded in this path. Moreover, these clocks do no longer affect the evaluation of guards, thus can be ignored in the rest of the analysis.

To build λ, we first build a finite region-graph path λ_0 inscribed in $S_0 \xrightarrow{e_0} \cdots \xrightarrow{e_{i_0}} S_{i_0+1}$ and reaching r_0. By post-stability of the simulation graph, such a path exists. Then we extend λ_0 to an infinite path starting from r_0. If X is empty, then any infinite path starting from r_0 is guaranteed to be progressive.

Otherwise, we extend λ_0 as follows. We pick some $x \in X$. By condition (a), there exists $j > i_0+1$ such that $x \in \mathsf{reset}(e_j)$. We build a finite region-graph path λ_x starting from r_0 and inscribed in $S_{i_0+1} \xrightarrow{e_{i_0+1}} \cdots \xrightarrow{e_j} S_{j+1}$ (this can be done since π' is pre-stable). Let $r_1 \in S_{j+1}$ be the region reached by λ_x. Since $x \in \mathsf{reset}(e_j)$, $r_1 \models x = 0$. We then extend λ_x starting from r_1 and choosing successor regions inscribed in π', until a region r_2 is reached such that time can elapse from r_2, that is, $r_2 \xrightarrow{\epsilon} r_3$ and $r_2 \neq r_3$ (thus, $r_3 \models x > 0$). We claim that such a region r_2 can always be reached. Indeed, due to condition (a), eventually all clocks in X will be reset. If no ϵ transition can occur up to that point, region r is reached such that $\forall x \in X, r \models x = 0$. By condition (a) and post-stability of π', a region where $x > 0$ can be reached from r, thus, an ϵ transition is eventually enabled from r. Let λ'_x be the path from r_1 to r_3. The concatenation $\lambda_x \cdot \lambda'_x$ is a finite path where clock x is reset and grows strictly greater than zero. We can repeat this process *ad infinitum*, for each clock in X, say, in a "round-robin" fashion. We thus get an infinite region-graph path $\lambda = \lambda_0 \cdot \lambda_{x_1} \cdot \lambda'_{x_1} \cdot \lambda_{x_2} \cdot \lambda'_{x_2} \cdots$, where all clocks in X are reset and grow strictly greater than zero infinitely often, while all clocks in Y remain unbounded after λ_0. Thus, λ is progressive.

(2) By Lemma 4.3, π has a maximal pre-stable sub-path $\pi' = S_0 \xrightarrow{e_0} S_1 \cdots$ and all infinite region-graph paths inscribed in π are also inscribed in π'. Since π is not progressive, there

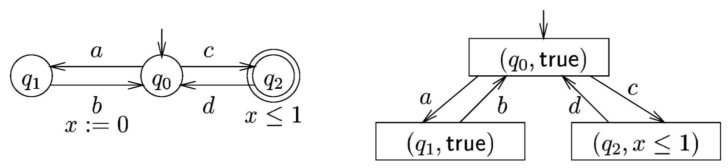


Figure 9. A TBA with no simple progressive accepting cycles.

must exist a clock x such that (a) x is either not reset or does not grow positive after some point on in π', and (b) x is bounded infinitely often in π'. This implies that x is bounded after some point on in π' (either x remains zero or x is bounded and never reset). Thus, every region-graph path inscribed in π' must be non-progressive. $\square$

Theorem 4.8. *Let A be any TBA. $Lang(A) \neq \emptyset$ iff there exists a progressive accepting cycle in the simulation graph of A.*

Proof: The proof is similar to the one of Theorem 4.4. If $Lang(A) \neq \emptyset$ then let λ be an accepting progressive region-graph path, inscribed in a simulation-path cycle. The cycle in which λ is inscribed must be progressive, otherwise, by part (2) of Lemma 4.7, λ cannot be progressive. In the other direction, part (1) of Lemma 4.7 can be used to extract a progressive region-graph path from the simulation-graph cycle. $\square$

We now give an example that demonstrates that the cycle mentioned in Theorem 4.8 need not be simple. The TBA shown on the left of figure 9 generates the simulation graph shown on the right. This graph contains a progressive accepting cycle, namely, $(q_0, \mathsf{true}) \xrightarrow{a} (q_1, \mathsf{true}) \xrightarrow{b} (q_0, \mathsf{true}) \xrightarrow{c} (q_2, x \le 1) \xrightarrow{d} (q_0, \mathsf{true})$. However, there is no such simple cycle. Indeed, there are only two simple cycles in this graph, namely, $(q_0, \mathsf{true}) \xrightarrow{a} (q_1, \mathsf{true}) \xrightarrow{b} (q_0, \mathsf{true})$ and $(q_0, \mathsf{true}) \xrightarrow{c} (q_2, x \le 1) \xrightarrow{d} (q_0, \mathsf{true})$. The first one is not accepting, while the second one is not progressive.

Although simple cycles do not capture both discrete and time acceptance, we can still prove that they capture discrete acceptance and time acceptance separately. The first result comes directly from graph theory, that is, in every finite graph, the existence of a cycle that visits an accepting state implies the existence of a simple cycle that visits an accepting state. The fact that simple cycles capture progressiveness is not at all trivial (especially since progressiveness is a conjunction of liveness properties, one for each clock) and is proven in what follows.

First, we need some definitions. Let $\lambda = S_1 \xrightarrow{e_1} \cdots \xrightarrow{e_l} S_1$ be a cycle. We say that a cycle λ' is *part of* λ if there exist $1 \le i \le k \le l$ such that $\lambda' = S_i \xrightarrow{e_i} \cdots \xrightarrow{e_k} S_k$ and $S_i = S_k$. For instance, in the example of figure 9, cycle $\lambda_1 = (q_0, \mathsf{true}) \xrightarrow{a} (q_1, \mathsf{true}) \xrightarrow{b} (q_0, \mathsf{true})$ is part of cycle $\lambda = (q_0, \mathsf{true}) \xrightarrow{a} (q_1, \mathsf{true}) \xrightarrow{b} (q_0, \mathsf{true}) \xrightarrow{c} (q_2, x \le 1) \xrightarrow{d} (q_0, \mathsf{true})$. We define an order $<$ on cycles, such that $\lambda < \lambda'$ if λ is part of λ' and $\lambda \neq \lambda'$. Since cycles are finite structures, $<$ is *well-founded*, that is, there cannot exist an infinite decreasing sequence $\lambda_1 > \lambda_2 > \cdots$. Moreover, it can be shown that for every non-simple cycle λ there exist two cycles λ_1 and λ_2 such that $\lambda_1 < \lambda$, $\lambda_2 < \lambda$ and λ_1 and λ_2 have the same root S, which is a node of λ.

Lemma 4.9. *Let λ be a progressive cycle in a simulation graph. Then, there exists a simple progressive cycle which is part of λ.*

Proof: The proof can be viewed as a proof diagram.

`Repeat:` Is λ simple ? If yes, we are done. Otherwise, there exist two cycles λ_1 and λ_2 such that $\lambda_1 < \lambda$ and $\lambda_2 < \lambda$ and λ_1 and λ_2 have the same root S, which is a node of λ. We distinguish two cases.

1. There exist two clocks x and y such that x is reset in λ_1 but not in λ_2 and y is reset in λ_2 but not in λ_1. Let $\xrightarrow{e_1} S_1$ be the last edge before S where x is reset in λ_1. Let $\xrightarrow{e_2} S_2$ be the last edge before S where y is reset in λ_2. Since x was reset last in λ_1 (in fact, y is not reset at all), all regions in S_1 must satisfy $x \leq y$. Since neither x nor y are reset in λ_1 from S_1 until S, the difference $x - y$ does not change from S_1 until S, therefore, all regions in S must also satisfy $x \leq y$. Reasoning symmetrically for λ_2, we obtain that all regions in S must satisfy $y \leq x$. Thus, all regions in S must satisfy $x = y$.
   Now, since λ is progressive, by part (1) of Lemma 4.7, there is a progressive region-graph path κ inscribed in λ. Immediately after κ takes transition e_1, the value of x is 0, since x is reset in e_1. By the fact that $x - y$ does not change from S_1 until S and the fact that $x = y$ in S, it must be the case that $y = 0$ right after e_1. Reasoning symmetrically on λ_2, we obtain that $x = y = 0$ right after each appearance of e_2 in κ. But then, no time elapses from e_1 until e_2, since y is not reset anywhere in-between. Similarly, no time elapses from e_2 until e_1, since x is not reset anywhere in-between. Thus, no time elapses at all along the path κ. We have a contradiction since κ was assumed to be progressive.
2. The negation of case 1, that is, at least one of λ_1, λ_2 resets all clocks that are reset in λ. Without loss of generality, we assume that it is λ_1. Every clock not reset in λ remains unbounded in λ, thus, also in λ_1. Therefore, λ_1 is progressive. Then, replace λ by λ_1 and repeat the reasoning starting from `Repeat`.

Since $\lambda_1 < \lambda$ and $<$ is well-founded, the replacement process of case 2 cannot be repeated *ad infinitum*. It must eventually terminate yielding a simple cycle. □

Theorem 4.10. *Let A be a TBA with persistent acceptance conditions. $Lang(A) \neq \emptyset$ iff there exists a simple progressive accepting cycle in the simulation graph of A.*

Proof: Assume that $Lang(A) \neq \emptyset$. By Theorem 4.8, the simulation graph of A has a progressive accepting cycle λ. By Lemma 4.9, there exists a simple progressive cycle λ' which is part of λ. Since λ is accepting and A has persistent acceptance conditions, all nodes of λ must be accepting. Since the nodes of λ' are also nodes of λ, they must also be accepting, therefore, λ' is accepting.

The other direction follows directly from Theorem 4.8. □

Table 1. Summary of results and algorithms.

	Persistent acceptance conditions	Non-persistent acceptance conditions
Strongly non-zeno	simple accepting cycle Simple DFS	simple accepting cycle Double DFS or Maximal SCCs
Not strongly non-zeno	simple accepting progressive cycle Full DFS or Transformation	accepting progressive cycle Incomplete search or Transformation

5. Algorithms

Based on the results of the previous sections, we now propose algorithms to check language emptiness of a timed Büchi automaton. All algorithms use the simulation graph as the basic structure to be explored. Table 1 summarizes the proposed algorithms. We distinguish four cases, depending on whether the automaton is strongly non-zeno or not, and whether it has persistent acceptance conditions or not. In each case, we propose an algorithm which exploits the special characteristics of the automaton.

5.1. Algorithms for strongly non-zeno timed Büchi automata

We first consider the case where the automaton to be checked, say A, is strongly non-zeno. The following is a direct consequence of Theorem 4.4.

Corollary 5.1. *Let A be a strongly non-zeno TBA. Lang*$(A) \neq \emptyset$ *iff the simulation graph of A has a maximal strongly-connected component containing an accepting node.*

According to Corollary 5.1, a standard algorithm for finding maximal strongly-connected components in a graph [24] can be used to check language emptiness of a strongly non-zeno TBA. An alternative is to use the *double-DFS* algorithm of [9], which is memory-efficient (a single extra bit is added to the state space). The double-DFS algorithm conducts two DFSs, one "embedded" into the other. Each DFS maintains a separate stack and a separate set of visited nodes (the extra bit serves to distinguish between the two sets of visited nodes). The outer-level DFS calls the inner-level DFS for every accepting node that it visits, and the inner-level DFS starts exploring from that node. As it is shown in [9], if an accepting cycle exists, then the inner-level DFS will find it the first time an accepting node of this cycle is visited. Each node of the graph is explored at most twice, once by the outer-level DFS and once by the inner-level DFS.

Checking language emptiness for a strongly non-zeno TBA with persistent acceptance conditions is even easier than in the general strongly non-zeno case. Indeed, it can be done using a *simple DFS* which keeps only a stack and a set of visited nodes V. The first time a node is visited, it is added to V. The DFS stops exploring further whenever it reaches a node that already belongs to V. Thus, the simple DFS explores each node of the graph at most once.

Corollary 5.2. *Let A be a strongly non-zeno TBA with persistent acceptance conditions. $Lang(A) \neq \emptyset$ iff the simple DFS algorithm visits an accepting node which is already in the stack.*

Proof: If the simple DFS algorithm visits an accepting node which is already in the stack, then a simple accepting cycle has been found, therefore, $Lang(A) \neq \emptyset$.

In the other direction, suppose $Lang(A) \neq \emptyset$. Then, by Theorem 4.4, the simulation graph of A has a simple accepting cycle. Let this cycle be $S_0 \xrightarrow{e_0} S_1 \xrightarrow{e_1} \cdots \xrightarrow{e_{l-1}} S_l \xrightarrow{e_l} S_0$. Since A has persistent acceptance conditions, all nodes in this cycle are accepting. Without loss of generality, we can assume that node S_0 is visited by the DFS before any other node S_i, $i = 1, \ldots, l$. By the properties of depth-first search, if a node B can be reached from node A and A is visited before B by the DFS, then the first time B is visited, A is still in the stack. Indeed, A is not popped from the stack until all successors of A visited after A are popped from the stack. Thus, when S_l is visited, the edge $S_l \xrightarrow{e_l} S_0$ will be explored and the accepting node S_0 will be found in the stack. □

5.2. Algorithms for general timed Büchi automata

In this section, we show how language emptiness can be checked for a TBA A which is not necessarily strongly non-zeno. As before, we distinguish two cases, depending on whether the acceptance conditions of A are persistent or not.

If A has persistent acceptance conditions, then, by Theorem 4.10, $Lang(A) \neq \emptyset$ iff the simulation graph of A contains a simple progressive accepting cycle. Such a cycle can be found using a *full DFS*, which, contrary to the simple DFS, does not maintain a set of visited nodes, but only maintains a stack. The full DFS stops exploring further only when it reaches a node already in the stack. Therefore, the full DFS finds *all* simple cycles in the simulation graph, and can check each one of them to determine whether it is progressive and accepting.

It is worth noting that, although Theorems 4.4 and 4.10 look similar, they give algorithms of very different worst-case complexities: the DFS algorithms of the previous section are linear in the size of the simulation graph, whereas the full DFS algorithm proposed above is exponential in the worst case, since the number of simple cycles in a graph can be exponential in the number of nodes.[5] To illustrate the fact that a simple DFS does not suffice in the general case, we give an example, shown in figure 10. The simulation graph of the automaton in the figure is isomorphic to the automaton structure. A simple DFS would first find the cycle $1 \to 2 \to 3 \to 1$ (which is not progressive) and then stop after exploring the path $1 \to 4 \to 2$, since node 2 has been already visited. In that way, the progressive cycle $1 \to 4 \to 2 \to 3 \to 1$ would be missed.

In the most general case, A is not strongly non-zeno and has non-persistent acceptance conditions. In this case, Theorem 4.8 applies, however, it requires us to find cycles which are non-simple in general, as the example of figure 9 shows. Since there is an infinite number of non-simple cycles in the graph, a "blind" enumeration of them would not terminate, unless the language of the automaton is non-empty. In any case, such an enumeration is not efficient. Therefore, the most pragmatic choices in this case seem to be the following.

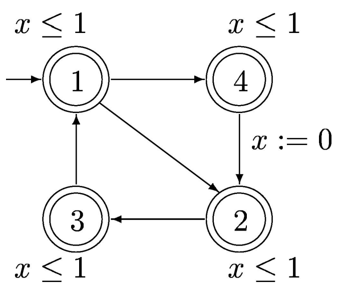


Figure 10. A timed Büchi automaton with persistent acceptance conditions.

- Either transform A into the strongly non-zeno automaton $\mathsf{snz}(A)$, as shown in Theorem 2.9, and apply the algorithms of Section 5.1 (notice that this option is feasible also when A has persistent acceptance conditions).
- Or use an *incomplete* search, which explores only a subset of all possible cycles. Then, if a progressive accepting cycle is found, we know that the language of A is non-empty, otherwise, no conclusion can be made.

6. Case study

The algorithms described in Section 5 have been implemented in a prototype tool called PROFOUNDER. PROFOUNDER takes as input a system of extended timed automata and a boolean expression which defines the accepting states. An extended timed automaton is a timed automaton with discrete variables of bounded domain (e.g., bounded integers, booleans, etc.). The system of extended timed automata is described as the parallel composition and synchronization of the various automata. PROFOUNDER is based on the SMI open toolbox [7].

PROFOUNDER generates C code, which is then compiled and linked with a library of symbolic states and operations. This produces an executable which can perform two types of analysis: (1) simple reachability, in search of a path that reaches an accepting state, or (2) TBA emptiness, in search of a progressive cycle that visits an accepting state. In TBA emptiness mode, the user can specify whether persistent ("almost-always") or non-persistent ("infinitely-often") acceptance conditions should be used. If the analysis succeeds, an example trace is generated: in case (1) the trace is a finite run; in case (2) the trace is *symbolic* and corresponds to a finite path followed by a cycle. Three algorithms have been implemented for TBA emptiness. Simple DFS, full DFS and the double DFS algorithm [9], as discussed in Section 5. For the moment, PROFOUNDER does not perform the transformation of Theorem 2.9. It does not check structural non-zenoness either.

In what follows, we present a case study where the verification of an asynchronous circuit is reduced to checking emptiness of a (strongly non-zeno) timed Büchi automaton. Apart from this example, we have used PROFOUNDER for numerous other examples, including the FDDI protocol [6], an automated vehicle control system [27, 29] and real-time scheduling applications [1].

Verification of asynchronous circuits

Circuits consisting of networks of interconnected boolean gates with bounded delays can be modeled using timed automata [19]. The behavior of the circuit is given as a set of equations of the form

$$x_i = b_i \longrightarrow [l_i, u_i]\ f_i(x_1, \ldots, x_n)$$

where f_i is the boolean function that defines the logical behavior of the gate x_i, b_i is the initial value of x_i, and $[l_i, u_i]$ is the range of possible delays.

Informally, the behavior is as follows. Whenever the evaluation of f_i changes, x_i changes accordingly, within some delay in the interval $[l_i, u_i]$. If, before the change in the value of x_i has taken place, f_i changes back to its previous value, x_i keeps its value. In other words, as long as x_i is *stable* (i.e., has the same value as $f_i(x_1, \cdots, x_n)$) no changes occur. As soon as x_i becomes unstable, it is "programmed" to stabilize within l_i to u_i time units, by changing value. This change is "canceled" if x_i becomes again stable meanwhile, because of a change in f_i.

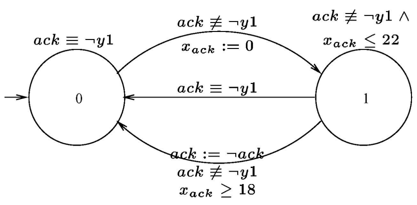
Formally, each equation is translated into a timed automaton whose set of behaviors coincides with the set of solutions of the corresponding equation. The behavior of the circuit is the set of runs of the composed automata.

As an example, we consider the equations in figure 11. These equations define the behavior of the `sbuf-ram-write` asynchronous circuit, which is a component of the Post Office communication co-processor [23]. The inputs are req, $prech$, $done$, $wenin$ and $wsldin$. The outputs are ack, $prbar$, $wsen$, wen, $wsld$, $y1$, and $y0$. The rest of the variables represent internal gates.

Figure 12 shows the timed automaton for the ack equation. The automaton is depicted graphically on the left of the figure and textually on the right, in the input format of PROFOUNDER. In the textual description, the header (first line) says that there are 2 discrete states, 3 edges and the initial discrete state is 0. The following three lines describe the three edges. The last two lines specify the invariants of the discrete states. PROFOUNDER allows invariants on the discrete state space as well as on clocks. For instance, the invariant $ack \equiv \neg y1$ on state 0 models the fact that the automaton moves to state 1 as soon as ack becomes unstable.

$$\begin{array}{rcl@{\qquad}rcl}
ack & = & 0 \longrightarrow [18,22]\ \neg y1 & B69 & = & 1 \longrightarrow [27,33]\ \neg wenin \wedge y1 \\
prbar & = & 0 \longrightarrow [27,33]\ \neg y0 \vee \neg y1 & B85 & = & 0 \longrightarrow [27,33]\ B158 \wedge \neg B69 \\
B104 & = & 0 \longrightarrow [27,33]\ \neg y1 \vee prech & y1 & = & 1 \longrightarrow [27,33]\ \neg B85 \vee \neg done \\
wen & = & 0 \longrightarrow [27,33]\ \neg y0 \wedge \neg B104 & B81 & = & 0 \longrightarrow [27,33]\ req \wedge \neg done \\
B95 & = & 0 \longrightarrow [27,33]\ \neg wenin \wedge \neg y1 & B154 & = & 1 \longrightarrow [27,33]\ \neg B81 \vee \neg prech \\
wsld & = & 0 \longrightarrow [27,33]\ \neg y0 \wedge B95 & B156 & = & 0 \longrightarrow [27,33]\ \neg y0 \vee \neg B154 \\
B91 & = & 1 \longrightarrow [27,33]\ \neg req \wedge \neg wsldin & y0 & = & 1 \longrightarrow [27,33]\ \neg B156 \vee wsldin \\
B158 & = & 0 \longrightarrow [27,33]\ \neg B91 \vee \neg y0 & wsen & = & 1 \longrightarrow [0,0]\ y1
\end{array}$$

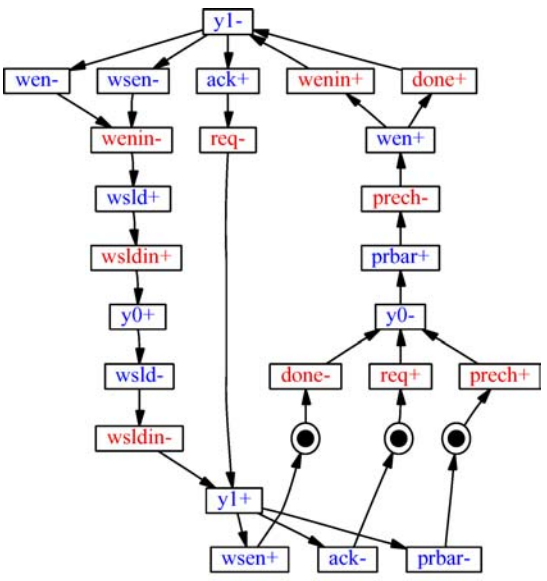
Figure 11. Equations for part of the Post Office communication co-processor [23].



```
des(0, 3, 2)
(0, [~(ack<=>(~y1))] Xack:=0, 1)
(1, [ (ack<=>(~y1))] , 0)
(1, [~(ack<=>(~y1)) /\ Xack>=18] ack:=~ack, 0)

[0, ack<=>(~y1)]
[1, (ack<=>y1) /\ (Xack<=22)]
```

Figure 12. Timed automaton for the *ack* gate: graphical (left) and textual (right) format.



INPUTS: req,prech,done,wenin,wsldin
OUTPUTS: ack,prbar,wsen,wen,wsld,y1,y0

Figure 13. Environment of the circuit.

The model of the environment of this circuit is depicted in figure 13 (the figure has been generated using the `petrify` toolbox which uses the input format *STG* [8]). The graph represents the precedence relation between inputs and outputs (inputs are in red and outputs in blue). For instance, the value of output y_0 is expected to change to false (y_0-) after inputs *done*, *req*, and *prech* change, respectively, to false ($done-$) and true ($req+$, $prech+$). The inputs switch from true to false and back, within a delay of 900 to 1111 time units. The equations in figure 14 define the timing behavior of the inputs.

The entire system consists of 16 timed automata modeling the behavior of the circuit, 5 timed automata modeling the timing behavior of the 5 inputs, and the automaton of the environment. The latter is untimed and has 2^7 states. Each of the 21 timed automata has two discrete states, one boolean variable and one clock. Let S be the product timed automaton

$$\begin{aligned}
req &= 0 \longrightarrow [900,1111]\ \neg req\\
prech &= 0 \longrightarrow [900,1111]\ \neg prech\\
done &= 1 \longrightarrow [900,1111]\ \neg done\\
wenin &= 0 \longrightarrow [900,1111]\ \neg wenin\\
wsldin &= 0 \longrightarrow [900,1111]\ \neg wsldin
\end{aligned}$$

Figure 14. Input equations.

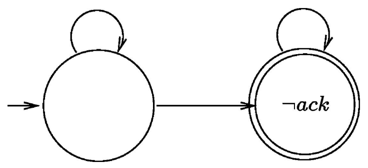


Figure 15. Observer automaton modeling the negation of the property "infinitely often *ack*".

of the entire system. S is strongly non-zeno: this is because each of the 21 component timed automata are strongly non-zeno.

One of the properties that the circuit must satisfy is that *ack* becomes true infinitely often. To verify this property, we compose the system with the *observer* automaton shown in figure 15. The observer is a Büchi automaton B modeling the negation of the above property: B accepts all runs where *ack* remains forever false after some point in the run. Let A be the TBA obtained by composing S with B. The circuit satisfies the property iff the language of A is empty.

PROFOUNDER checks that the language of A is empty in about 50 seconds, on a Pentium III running at 650 MHz with 128 MB RAM. During the search, PROFOUNDER generates the entire simulation graph (194517 symbolic states and 428071 transitions) and explores 7335 symbolic cycles.

7. Conclusions

The contribution of the work presented in this paper is both theoretical and practical.

From a theoretical point of view, we have shown that the so-called simulation graph of a timed Büchi automaton can be used to check language emptiness. This is not trivial, since the simulation graph does not satisfy the basic property of the region graph (and all time-abstracting bisimulation graphs), namely, pre-stability. Nevertheless, we show that any cycle in the simulation graph contains a cycle of the region graph. In fact, as shown in [6], similar techniques can be used for model-checking a larger class of properties expressed in the temporal logic ETCTL$^*_\exists$, which is more expressive than TCTL.

From a practical point of view, the use of the simulation graph instead of the region graph is of great interest, since in most cases in practice, the simulation graph is much smaller than the region graph. Still, checking emptiness for timed Büchi automata is harder than for untimed ones. This is because both discrete acceptance conditions and timed acceptance conditions (non-zenoness) need to be checked. In general, cycles which satisfy both conditions need not be simple. We have identified classes of timed Büchi automata where emptiness can be checked more efficiently. In the case of strongly non-zeno systems, discrete acceptance implies non-zenoness, and emptiness can be checked either by simple depth-first or strongly-connected component searches, with complexity linear on the size of the simulation graph. In the case of automata which are not strongly non-zeno, but have persistent ("almost-always" type) discrete acceptance conditions, checking simple cycles suffice. However, there is generally an exponential number of them in the size of the simulation graph.

Finally, we have presented a prototype implementation of the above techniques in the tool PROFOUNDER. To our knowledge, PROFOUNDER is the only available tool for checking emptiness of timed Büchi automata.

Notes

1. Convex $\mathcal{X}$-polyhedra are particularly interesting since they can be represented using space-efficient data-structures such as *difference-bound matrices* [12] ($O(n^2)$, where n is the number of clocks). Standard operations on these data-structures are also time-efficient (e.g., intersection in $O(n^2)$, test for emptiness in $O(n^3)$).
2. In fact, the region equivalence is a *time-abstracting bisimulation* (although not the coarsest one, in general) and all such bisimulations preserve both linear-time and branching-time properties [28].
3. This definition of progressiveness is slightly different from the one given in [2]. This is because in the model of [2], the delay between two discrete transitions must be strictly greater than zero, whereas in our model we allow this delay to be zero. Zero delays between discrete transitions are useful for capturing *atomic* sequences of actions.
4. In the strongly non-zeno case, time acceptance need not be directly checked, since it is implied by discrete acceptance.
5. Actually, the complexity of the full DFS algorithm can be improved by keeping a set of visited non-accepting nodes. Then, exploration can be stopped whenever a node already in that set is visited. This is because such a node cannot lead to any progressive accepting cycle, therefore, how the node is reached does not matter.

References

1. K. Altisen, G. Goessler, A. Pnueli, J. Sifakis, and S. Tripakis, "A framework for scheduler synthesis," in *IEEE Real-Time Systems Symposium, RTSS'99*, 1999.
2. Rajeev Alur, "Techniques for automatic verification of real-time systems", PhD thesis, Department of Computer Science, Stanford University, 1991.
3. R. Alur, C. Courcoubetis, and D.L. Dill, "Model checking in dense real time," *Information and Computation*, Vol. 104, No. 1, pp. 2–34, 1993.
4. R. Alur, C. Courcoubetis, N. Halbwachs, D.L. Dill, and H. Wong-Toi, "Minimization of timed transition systems," in *3rd Conference on Concurrency Theory CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, Springer-Verlag, 1992, pp. 340–354.
5. R. Alur and D.L. Dill, "A theory of timed automata," *Theoretical Computer Science*, Vol. 126, pp. 183–235, 1994.
6. A. Bouajjani, S. Tripakis, and S. Yovine, "On-the-fly symbolic model checking for real-time systems," in *Proc. of the 18th IEEE Real-Time Systems Symposium, San Francisco, CA*, IEEE, December 1997, pp. 25–34.
7. M. Bozga, "Smi: An open toolbox for symbolic protocol verification," Technical report, VERIMAG, 1997.
8. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: A tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Transactions on Information and Systems*, Vol. E80-D, No. 3, pp. 315–325, 1997.

9. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis, "Memory efficient algorithms for the verification of temporal properties," *Formal Methods in System Design*, Vol. 1, pp. 275–288, 1992.
10. C. Courcoubetis and M. Yannakakis, "Minimum and maximum delay problems in real-time systems," in *Computer-Aided Verification*, LNCS 575, Springer-Verlag, 1991.
11. C. Daws, A. Olivero, S. Tripakis, and S. Yovine, "The tool KRONOS," in *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, Springer-Verlag, 1996, pp. 208–219.
12. D.L. Dill, "Timing assumptions and verification of finite-state concurrent systems," in J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, Lecture Notes in Computer Science 407, Springer-Verlag, 1989, pp. 197–212.
13. T. Henzinger, O. Kupferman, and M. Vardi, "A space-efficient on-the-fly algorithm for real-time model-checking," in *CONCUR'96*. LNCS 1119, 1996.
14. 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.
15. L. Lamport, "Sometimes is sometimes "not never"—on the temporal logic of programs," in *7th ACM Symp. POPL*, 1980, pp. 174–185.
16. K. Larsen, P. Petterson, and W. Yi, "Uppaal in a nutshell," *Software Tools for Technology Transfer*, Vol. 1, Nos. (1/2), 1997.
17. D. Lee and M. Yannakakis, "Online minimization of transition systems," in *ACM Symp. on Theory of Computing*, 1992.
18. O. Lichtenstein and A. Pnueli, "Checking that finite state concurrent programs satisfy their linear specification," in *12th ACM Symp. POPL*, New Orleans, January 1985, pp. 97–107.
19. O. Maler and A. Pnueli, "Timing analysis of asynchronous circuits using timed automata," in P.E. Camurati, H. Eveking (Eds.), *Proc. CHARME'95. LNCS 987*, Springer Verlag, 1995.
20. R. Paige and R. Tarjan, "Three partition refinement algorithms," *SIAM Journal on Computing*, Vol. 16, No. 6, 1987.
21. A. Pnueli, "A temporal logic of concurrent programs," *Theoretical Computer Science*, Vol. 13, pp. 45–60, 1981.
22. O. Sokolsky and S. Smolka, "Local model checking for real-time systems," in *CAV'95*. LNCS 939, 1995.
23. K.S. Stevens, S.V. Robinson, and A.L. Davis, "The post office—communication support for distributed ensemble architectures," in *Sixth International Conference on Distributed Computing Systems*, 1986.
24. R. Tarjan, "Depth first search and linear graph algorithms," *SIAM Journal on Computing*, Vol. 1, No. 2, pp. 146–170, 1972.
25. S. Tripakis, "The formal analysis of timed systems in practice," PhD thesis, Université Joseph Fourrier de Grenoble, 1998.
26. S. Tripakis, "Verifying progress in timed systems," in *5th Intl. AMAST Workshop on Real-Time and Probabilistic Systems (ARTS)*, LNCS 1601, 1999.
27. S. Tripakis, "Description and schedulability analysis of the software architecture of an automated vehicle control system," in *EMSOFT'02*, 2002. To appear in LNCS.
28. S. Tripakis and S. Yovine, "Analysis of timed systems using time-abstracting bisimulations," *Formal Methods in System Design*, Vol. 18, No. 1, pp. 25–68, 2001.
29. S. Tripakis and S. Yovine, "Timing analysis and code generation of vehicle control software using Taxys," in *Workshop on Runtime Verification (RV'01)*. Volume 55, Issue 2 of ENTCS, Elsevier, 2001.
30. S. Yovine, "KRONOS: A verification tool for real-time systems," *Software Tools for Technology Transfer*, 1997.