Efficient On-The-Fly Emptiness Check for Timed Büchi Automata

F. Herbreteau and B. Srivathsan

LaBRI (Université de Bordeaux - CNRS)

Abstract. The Büchi non-emptiness problem for timed automata concerns deciding if a given automaton has an infinite non-Zeno run satisfying the Büchi accepting condition. The solution to this problem amounts to searching for a cycle in the so-called zone graph of the automaton. Since non-Zenoness cannot be verified directly from the zone graph, additional constructions are required. In this paper, it is shown that in many cases, non-Zenoness can be ascertained without extra constructions. An on-the-fly algorithm for the non-emptiness problem, using an efficient non-Zenoness construction only when required, is proposed. Experiments carried out with a prototype implementation of the algorithm are reported and the results are seen to be promising.

1 Introduction

Timed automata [1] are finite automata extended with clocks. The clock values can be compared with constants and reset to zero on the transitions, while they evolve continuously in the states. The emptiness problem for timed automata is decidable. Consequently, timed automata are used in tools like Uppaal [3] and Kronos [8] for model-checking reachability properties of real-time systems. They have also been successfully used for industrial case studies, e.g. [4].

For the verification of liveness properties, timed automata with Büchi conditions have been considered. The emptiness problem demands if there exists a non-Zeno run that visits an accepting state infinitely often. A run is non-Zeno if the total time elapsed during the run is unbounded. This further requirement makes the problem harder than classical Büchi emptiness.

The solution proposed in [1] relies on a symbolic semantics called the region graph. The emptiness problem for timed Büchi automata is reduced to classical Büchi emptiness on the region graph with an additional time progress requirement. However, the region graph construction is very inefficient for model checking real life systems. Zones are used instead of regions in the aforementioned tools. The zone graph is another symbolic semantics that is coarser than the region graph. Although it is precise enough to preserve reachability properties, it is too abstract to directly infer time progress and hence non-Zenoness.

Two approaches have been proposed in order to ascertain time progress. In [16], the automaton is transformed into a so-called strongly non-Zeno automaton, which has the property that all the runs visiting an accepting state infinitely often are non-Zeno. Hence, non-Zenoness is reduced to Büchi emptiness. However it has been proved in [12] that this construction sometimes results in an exponential blowup.

A careful look at the conditions for Zenoness yields another solution to verify time progress. Two situations can prevent time from diverging on a run. Either the value of a clock is bounded and not reset anymore, or some clock is reset and later checked to zero preventing time elapse in-between. In [12] a guessing zone graph has been introduced to identify clear nodes where time can elapse despite zero-checks. Hence timed Büchi automata emptiness reduces to Büchi emptiness on the guessing zone graph with two infinitary conditions namely clear nodes and accepting nodes. The guessing zone graph is only $|X|+1$ times bigger than the zone graph, where $|X|$ stands for the number of clocks. The straightforward algorithm handling both the above mentioned situations explores the guessing zone graph $\mathcal{O}(|X|)$ number of times.

On a parallel note, the emptiness problem for (untimed) Büchi automata has been extensively studied (see [13] for a survey). The best known algorithms in the case of multiple Büchi conditions are inspired by the Tarjan's SCC algorithm. An on-the-fly algorithm which computes the graph during the search, and terminates as soon as a satisfactory SCC is found, is crucial in the timed settings as the zone graph is often huge. It is known that the Couvreur's algorithm [9] outperforms all on-the-fly SCC-based algorithms.

In this paper, we present an on-the-fly algorithm that solves the emptiness problem for timed Büchi automata. We noticed from several experiments that Büchi emptiness can sometimes be decided directly from the zone graph, i.e. without using any extra construction. This is the way we take here. The Couvreur's algorithm is employed to search for an accepting SCC directly on the zone graph. Then, we face two challenges.

The first challenge is to detect if this SCC has *blocking* clocks, that is clocks that are bounded from above but are not reset in the transitions of the SCC. The SCCs with blocking clocks have to be re-explored by discarding transitions bounding these clocks. Notice that discarding edges may split the SCC into smaller SCCs. The difficult task is to discover the smaller SCCs and explore them *on-the-fly*.

The second challenge is to handle SCCs with *zero-checks*, that is clocks that are tested for value being zero. In this case, the zone graph does not contain enough information to detect time progress. This is where the guessing zone graph construction is required. The involving part of the problem is to introduce this construction on-the-fly. In particular the part of the guessing zone graph that is explored should be restricted only to the transitions from the SCC containing zero-checks.

We propose an algorithm that fulfills both challenges. In the worst case, it runs in time $\mathcal{O}(|ZG|.(|X|+1)^2)$. When the automaton does not have zero checks it runs in time $\mathcal{O}(|ZG|.(|X|+1))$. When the automaton further has no blocking clocks, it runs in time $\mathcal{O}(|ZG|)$. Our algorithm further incorporates an improvement that proves powerful in practice. Indeed, non-Zenoness could be established from situations where the value of some clock is known to increase by 1 infinitely often. Even in the presence of zero-checks, this property avoids exploring the guessing zone graph. Again, this improvement is applied on-the-fly.

Next, we show that all these operations on SCCs can be handled efficiently as the memory needed by our algorithm is comparable to the memory usage of the strongly non-Zeno approach in [16].

Finally, we also report some successful experiments of our algorithm on examples from the literature. All the experiments that we have conducted show that non-Zenoness constructions need not be applied mechanically, hence validating our approach.

Related work. Timed automata with Büchi conditions have been introduced in the seminal paper [1]. The zone approach to model-checking has been introduced in [5] and later used in the context of timed automata [11]. The Büchi emptiness problem using zones has been studied in [6, 16, 15]. The idea of the strongly non-Zeno approach first appears in [14]. The strongly non-Zeno construction has been implemented in the tool Profounder [16]. Finally, the recent paper [12] introduces the guessing zone graph construction.

Outline. The paper is organized as follows. In Section 2 we define the emptiness problem. Then we present the two approaches that ensure time progress in Section 3. We detail our algorithm and its optimizations in Section 4. Finally, we discuss experiments and future work in Section 5.

2 The Emptiness Problem for Timed Büchi Automata

2.1 Timed Büchi Automata

Let X be a set of clocks, i.e., variables that range over $\mathbb{R}_{\geq 0}$, the set of non-negative real numbers. *Clock constraints* are conjunctions of comparisons of variables with integer constants, e.g. $(x \leq 3 \wedge y > 0)$. Let $\Phi(X)$ denote the set of clock constraints over clock variables X.

A *clock valuation* over X is a function $\nu : X \to \mathbb{R}_{\geq 0}$. We denote $\mathbb{R}_{\geq 0}^X$ the set of clock valuations over X, and $\mathbf{0} : X \to \{0\}$ the valuation that associates 0 to every clock in X. We write $\nu \models \phi$ when ν satisfies ϕ, i.e. when every constraint in ϕ holds after replacing every x by $\nu(x)$.

For a valuation ν and $\delta \in \mathbb{R}_{\geq 0}$, let $(\nu + \delta)$ be the valuation such that $(\nu + \delta)(x) = \nu(x) + \delta$ for all $x \in X$. For a set $R \subseteq X$, let $[R]\nu$ be the valuation such that $([R]\nu)(x) = 0$ if $x \in R$ and $([R]\nu)(x) = \nu(x)$ otherwise.

A *Timed Büchi Automaton (TBA)* is a tuple $\mathcal{A} = (Q, q_0, X, T, Acc)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, X is a finite set of clocks, $Acc \subseteq Q$ is a set of accepting states, and $T \subseteq Q \times \Phi(X) \times 2^X \times Q$ is a finite set of transitions (q, g, R, q') where g is a *guard*, and R is a *reset* of the transition. Examples of TBA are depicted in Figure 1.

A *configuration* of $\mathcal{A}$ is a pair $(q, \nu) \in Q \times \mathbb{R}_{\geq 0}^X$; with $(q_0, \mathbf{0})$ being the *initial configuration.* A *discrete transition* between configurations $(q, \nu) \xrightarrow{t} (q', \nu')$ for

$t = (q, g, R, q')$ is defined when $\nu \vDash g$ and $\nu' = [R]\nu$. We also have *delay transitions* between configurations: $(q, \nu) \xrightarrow{\delta} (q, \nu + \delta)$ for every q, ν and $\delta \in \mathbb{R}_{\geq 0}$. We write $(q, \nu) \xrightarrow{\delta,t} (q', \nu')$ if $(q, \nu) \xrightarrow{\delta} (q, \nu + \delta) \xrightarrow{t} (q', \nu')$.

A *run* of $\mathcal{A}$ is a finite or infinite sequence of configurations connected by $\xrightarrow{\delta,t}$ transitions, starting from the initial state q_0 and the initial valuation $\nu_0 = \mathbf{0}$:

$$(q_0, \nu_0) \xrightarrow{\delta_0,t_0} (q_1, \nu_1) \xrightarrow{\delta_1,t_1} \cdots$$

A run σ *satisfies the Büchi condition* if it visits *accepting configurations* infinitely often, that is configurations with a state from Acc. A *duration* of the run is the accumulated delay: $\sum_{i\geq 0} \delta_i$. A run σ is *Zeno* if its duration is bounded.

Definition 1. *The* Büchi non-emptiness problem *is to decide if $\mathcal{A}$ has a non-Zeno run satisfying the Büchi condition.*

The class of TBA we consider is usually known as diagonal-free TBA since clock comparisons like $x - y \leq 1$ are disallowed. Since we are interested in the Büchi non-emptiness problem, we can consider automata without an input alphabet and without invariants since they can be simulated by guards.

The Büchi non-emptiness problem is known to be PSPACE-complete [1].

2.2 The Zone Graph

A zone is a set of valuations defined by a conjunction of two kinds of constraints: comparison of the difference between two clocks with a constant, or comparison of the value of a single clock with a constant. For instance $(x - y \geq 1) \wedge (y < 2)$ is a zone.

The transition relation on valuations can be transposed to zones. Let $\overrightarrow{Z}$ be a zone denoting the result of time elapse from Z, that is, the set of all valuations ν' such that $\nu' = \nu + \delta$ for some $\nu \in Z$ and $\delta \in \mathbb{R}_{\geq 0}$. We have $(q, Z) \xrightarrow{t} (q', Z')$ if Z' is the set of valuations ν' such that $(q, \nu) \xrightarrow{t} (q', \nu')$ for some $\nu \in \overrightarrow{Z}$. It can be checked that Z' is a zone. Difference Bound Matrices (DBMs) can be used for the representation of zones [11]. Transitions are computed efficiently for zones represented by DBMs.

However, the number of reachable symbolic configurations (q, Z) may not be finite [10]. Hence tools like Kronos or Uppaal use an approximation operator *Approx* that reduces the set of zones to a finite set. The *zone graph* of $\mathcal{A}$, denoted $ZG(\mathcal{A})$, has nodes of the form $(q, \mathit{Approx}(Z))$. Observe that $ZG(\mathcal{A})$ only has finitely many nodes. The initial node is (q_0, Z_0) where q_0 is the initial state of $\mathcal{A}$ and Z_0 is the zone where all the clocks are equal to zero. The transitions of the zone graph are $(q, Z) \xrightarrow{t} (q', \mathit{Approx}(Z'))$ instead of $(q, Z) \xrightarrow{t} (q', Z')$.

It remains to define *Approx* in such a way that verification results for $ZG(\mathcal{A})$ entail similar results for $\mathcal{A}$. Let M be the maximal constant in $\mathcal{A}$. M defines the precision of the approximation. We define the *region equivalence* over valuations as $\nu \sim_M \nu'$ iff for every $x, y \in X$:

1. $\nu(x) > M$ iff $\nu'(x) > M$;
2. if $\nu(x) \leq M$, then $\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor$;
3. if $\nu(x) \leq M$, then $\{\nu(x)\} = 0$ iff $\{\nu'(x)\} = 0$;
4. for every integer $c \in (-M, M)$: $\nu(x) - \nu(y) \leq c$ iff $\nu'(x) - \nu'(y) \leq c$.

The first three conditions ensure that ν and ν' satisfy the same guards. The last one enforces that for every $\delta \in \mathbb{R}_{\geq 0}$ there is $\delta' \in \mathbb{R}_{\geq 0}$, such that valuations $\nu + \delta$ and $\nu' + \delta'$ satisfy the same guards. This definition of region equivalence is introduced in [7]. Notice that it is finer than the equivalence defined in [1] thanks to the last condition which is needed for correctness of Theorem 1 below.

An equivalence class of $\sim_M$ is called a *region*. For a zone Z, we define $\mathit{Approx}(Z)$ as the smallest union of regions that is convex and contains Z. Notice that there are finitely many regions, hence finitely many approximated zones.

We call a *path* in the zone graph a finite or infinite sequence of transitions:

$$(q_0, Z_0) \xrightarrow{t_0} (q_1, Z_1) \xrightarrow{t_1} \cdots$$

A path is *initialized* when it starts from the initial node (q_0, Z_0). A run of $\mathcal{A}$ $(q_0, \nu_0) \xrightarrow{\delta_0, t_0} (q_1, \nu_1) \xrightarrow{\delta_1, t_1} \cdots$ is an *instance* of a path if $\nu_i \in Z_i$ for every $i \geq 0$. The path is called an *abstraction* of the run.

Approx preserves many verification properties, and among them Büchi accepting paths as stated in the following theorem.

Theorem 1 ([15]). *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 $ZG(\mathcal{A})$.*

As a consequence, every run of $\mathcal{A}$ that satisfies the accepting conditions yields a path in $ZG(\mathcal{A})$ that also satisfies the accepting conditions, and conversely. However, it is not possible to determine from a path in $ZG(\mathcal{A})$ if it can be instantiated to a non-Zeno run of $\mathcal{A}$. Hence, we need to preserve more information either in the automaton or in the zone graph.

3 Solutions to the Zenoness Problem

Theorem 1 is a first step towards an algorithm for the Büchi emptiness problem working from the zone graph. It remains to ensure that a given path in $ZG(\mathcal{A})$ has a non-Zeno instance in $\mathcal{A}$. Observe that the zone graph is not pre-stable: a transition $(q, Z) \xrightarrow{t} (q', Z')$ may not have an instance for a valuation $\nu \in Z$. Hence zones are not precise enough to detect time progress.

For example, the state 3 of $\mathcal{A}_1$ in Figure 1 is reachable with zone $Z = (0 \leq x \wedge x \leq y)$ on path from 0 via 2. At first sight, it seems that time can elapse in configuration $(3, Z)$. This is not the case as the transition from 3 to 0 forces x to be equal to 0. Hence, Z has to be separated in two zones: $x = 0 \wedge x \leq y$ that can take the transition from 3 to 0, and $x > 0 \wedge x \leq y$ that is a deadlock node.

We present two approaches from the literature to overcome this problem. Both are used in our algorithm in section 4. They rely on the properties of the

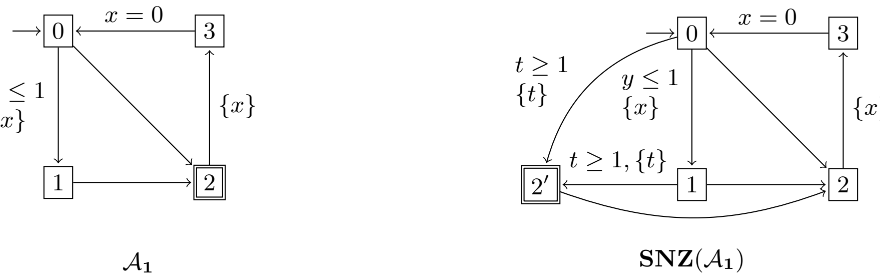


Fig. 1. A Timed Automaton and a Strongly non-Zeno Timed Automaton.

transitions $(q, Z) \xrightarrow{g;R} (q', Z')$ in $ZG(\mathcal{A})$. We say that a transition *bounds x from above* if there exists $c \geq 1$ such that $\overrightarrow{Z} \wedge g \Rightarrow (x \leq c)$. Observe that this is equivalent to $\nu''(x) \leq c$ for every valuation ν'' such that both $(q, \nu) \xrightarrow{\delta} (q, \nu'')$ and $\nu'' \models g$. Similarly a transition *bounds x from below* when $\overrightarrow{Z} \wedge g \Rightarrow (x \geq c)$; it *zero checks* x when $\overrightarrow{Z} \wedge g \Rightarrow (x = 0)$. Finally, a transition *resets* x when x belongs to R. Notice that all these properties are easily checked using DBMs for zones and guards.

3.1 Adding One Extra Clock

A common solution to deal with Zeno runs is to transform a TBA into a *strongly non-Zeno automaton*, i.e. such that all runs satisfying the Büchi condition are guaranteed to be non-Zeno.

The construction in [16, 2] adds one clock t and duplicates the accepting states in order to ensure that at least one unit of time has passed between two visits to an accepting state. Figure 1 presents $\mathcal{A}_1$ along with its strongly non-Zeno counterpart $SNZ(\mathcal{A}_1)$. The correctness of the approach comes from Lemma 1 below, the proof of which is omitted.

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

The converse also holds: if $\mathcal{A}$ has a non-Zeno accepting run, then 1 time unit elapses infinitely often. Hence, the guard $t \geq 1$ is satisfied infinitely often and there exists an accepting run of $SNZ(\mathcal{A})$. By Theorem 1 one can find an accepting path in $ZG(SNZ(\mathcal{A}))$.

Despite being simple, this construction may lead to an exponential blowup. It is shown in [12] that there exists some TBA $\mathcal{A}$ such that the zone graph of $SNZ(\mathcal{A})$ has size $|ZG(\mathcal{A})|.2^{\mathcal{O}(|X|)}$. Our algorithm described in section 4 will use a similar idea as an optimization. However, no new clocks are added and no blowup is inflicted.

3.2 The Guessing Zone Graph

Another solution was introduced in [12] in order to avoid the exponential blowup. Consider a path in $ZG(\mathcal{A})$ that only has Zeno instances. Two situations may occur. Either the path has infinitely many transitions that bound some clock x from above, but only finitely many transitions that reset x. Then, the total time elapsed is bounded. Or, time cannot elapse at all because of zero-checks, as this is the case in state 3 of $\mathcal{A}_1$ in Figure 1.

Zero-checks are handled thanks to a modification of the zone graph. The nodes will now be triples (q, Z, Y) where $Y \subseteq X$ is the set of clocks that can potentially be checked to 0. It means in particular that other clock variables, i.e. those from $X - Y$ are assumed to be bigger than 0. We write $(X - Y) > 0$ for the constraint saying that all the variables in $X - Y$ are not 0.

Definition 2. *Let $\mathcal{A}$ be a TBA over a set of clocks X. The* guessing zone graph *$GZG(\mathcal{A})$ has nodes of the form (q, Z, Y) where (q, Z) is a node in $ZG(\mathcal{A})$ and $Y \subseteq X$. The initial node is (q_0, Z_0, X), with (q_0, Z_0) the initial node of $ZG(\mathcal{A})$. There is a transition $(q, Z, Y) \xrightarrow{t} (q', Z', Y \cup R)$ in $GZG(\mathcal{A})$ if there is a transition $(q, Z) \xrightarrow{t} (q', Z')$ in $ZG(\mathcal{A})$ with $t = (q, g, R, q')$, and there are valuations $\nu \in Z$, $\nu' \in Z'$, and δ such that $\nu + \delta \vDash (X - Y) > 0$ and $(q, \nu) \xrightarrow{\delta,t} (q, \nu')$. We also introduce a new auxiliary letter τ, and put transitions $(q, Z, Y) \xrightarrow{\tau} (q, Z, Y')$ for $Y' = \emptyset$ or $Y' = Y$.*

Observe that the definition of transitions reflects the intuition about Y we have described above. Indeed, the additional requirement on the transition $(q, Z, Y) \xrightarrow{t} (q', Z', Y \cup R)$ is that it should be realizable when the clocks outside Y are strictly positive; so there should be a valuation satisfying $(X - Y) > 0$ that realizes this transition. This construction entails that from a node $(q, Z, \emptyset)$ every reachable zero-check is preceded by the reset of the variable that is checked, and hence nothing prevents a time elapse in this node. We call such a node *clear*. We call a node (q, Z, Y) *accepting* if it contains an accepting state q. Figure 4 depicts the part of $GZG(\mathcal{A}_1)$ reachable from node $(3, x == 0, \{x, y\})$ with state 1 and all its transitions removed from $\mathcal{A}_1$.

Notice that directly from Definition 2 it follows that a path in $GZG(\mathcal{A})$ determines a path in $ZG(\mathcal{A})$ obtained by removing τ transitions and the third component from nodes.

We say that a path is *blocked* if there is a variable that is bounded from above by infinitely many transitions but reset by only finitely many transitions on the path. Otherwise the path is called *unblocked*. We have the following result.

Theorem 2 ([12]). *A TBA $\mathcal{A}$ has a non-Zeno run satisfying the Büchi condition iff there exists an unblocked path in $GZG(\mathcal{A})$ visiting both an accepting node and a clear node infinitely often.*

It is shown in the same paper that the proposed solution does not produce an exponential blowup. This is due to the fact that the zones reachable in $ZG(\mathcal{A})$

order the clocks. More precisely, if Z is the zone of a reachable node in $ZG(\mathcal{A})$ then for every two clocks x, y, Z implies that at least one of $x \leq y$, or $y \leq x$ holds [12]. Now, it can be checked that for every node (q, Z, Y) reachable in $GZG(\mathcal{A})$, the set Y respects the order given by Z, that is whenever $y \in Y$ and Z implies $x \leq y$ then $x \in Y$. In the end, a nice polynomial bound is obtained on the size of the guessing zone graph.

Lemma 2 ([12]). *Let $|ZG(\mathcal{A})|$ be the size of the zone graph, and $|X|$ be the number of clocks in $\mathcal{A}$. The number of reachable nodes of $GZG(\mathcal{A})$ is bounded by $|ZG(\mathcal{A})|.(|X| + 1)$.*

4 Solving the emptiness problem on-the-fly

4.1 On-the-fly emptiness check on the guessing zone graph

We propose an on-the-fly algorithm for the Büchi emptiness problem based on Theorem 2. It requires to find an unblocked path in $GZG(\mathcal{A})$ that visits both an accepting node and a clear node infinitely often. To make the presentation of the algorithm simpler, we assume that there is one distinguished Büchi condition that labels exactly all the clear nodes. Hence we search for an unblocked path in $GZG(\mathcal{A})$ that satisfies the Büchi conditions.

Observe that we do not need to exhibit the path; proving its existence is simpler and sufficient[1]. Our problem can thus be stated over the SCCs of $GZG(\mathcal{A})$. A clock is *blocking* in an SCC if it is not reset by any transition in the SCC, but it is bounded from above by one of them. An SCC is *unblocked* when it has no blocking clock. Hence, finding an SCC that is unblocked and that contains an accepting node (for every Büchi condition) yields the required path. Conversely, from such a path, we have a satisfactory SCC.

The algorithm essentially involves three tasks: (1) detecting an accepting SCC, (2) inferring any blocking clocks in the SCC, and (3) re-exploring the SCC with blocking transitions removed. Since the blocking transitions might involve resets of other clocks, the SCCs obtained after removing blocking transitions might in turn contain new blocking clocks. These tasks have to be accomplished on-the-fly. The Couvreur's algorithm [9] finds accepting SCCs in (untimed) Büchi automata on-the-fly. We first adapt it to infer blocking clocks. Next we propose a modification to enable performing the third task. This last contribution is particularly involving. Our algorithm is depicted in Figure 2.

Couvreur's algorithm The Couvreur's algorithm, can be viewed as three functions *check_dfs*, *merge_scc* and *close_scc*. The algorithm annotates every node with an integer *dfsnum* and a boolean *opened*. The variable *dfsnum* is assigned based on the order of appearance of the nodes during the depth-first

[1] The path is a counter-example to the emptiness of the automaton, hence it can be an expected outcome of the algorithm. ¿s in the untimed case, a short counter-example can be generated in a separate task from the outcome of our algorithm.

search. The *opened* bit is set to *true* when the node is just opened for exploration by *check_dfs* and is set to false by *close_scc* when the entire SCC of the node has been completely explored. The algorithm uses two stacks *Roots* and *Active*. The *Roots* stack stores the root of each SCC in the current search path. The root is the node of the SCC that was first visited by the DFS. If the roots stack is $s_0 s_1 \ldots s_n$, then for $0 \leq i \leq n-1$, s_i is the root of the SCC containing all the nodes with *dfsnum* between $s_i.dfsnum$ and $s_{i+1}.dfsnum$ that have *opened* set to *true* and s_n is the root of the SCC containing all nodes with *dfsnum* greater than $s_n.dfsnum$ which have *opened* set to *true*.

The main function *check_dfs* proceeds by exploring the graph in depth-first search (DFS) order. When a successor t of the current node s is found, $t.dfsnum$ being zero implies t has not been visited yet and $t.opened$ being true implies that t belongs to the same SCC as s. When t belongs to the SCC of s, all the nodes visited in the path from t to s also belong to the SCC of s. These nodes are collected by the function *merge_scc*, which finds the root s_i of t and repeatedly pops the *Roots* stack so that s_i comes to the top, signifying that it is the root of the SCC containing all the nodes visited from t to s. A *maximal* SCC is detected when all the transitions of the current node s have been explored, with s being on the top of the *Roots* stack. The *close_scc* function is now called that sets the *opened* bit of all nodes in the SCC rooted at s to $false$. To identify these nodes, the *Active* stack is used, which stores all the nodes of the partially explored SCCs, in the order of the *dfsnum*.

Detecting blocking clocks Observe that each node s in the *Roots* stack represents an SCC rooted at s. The idea is to augment the *Roots* stack so that along with each node s, the set of clocks (ub) that are bounded above and the set of clocks r that are reset in a transition of the SCC rooted at s, are also stored. The set $ub - r$ gives the set of blocking clocks of the SCC rooted at s. To achieve this, SCCs are stored in the *Roots* stack as tuples $(s, a, ub, r, ub_{in}, r_{in})$ where the extra sets ub_{in} and r_{in} store respectively the clocks that are bounded above and reset in the transition leading to s. The second modification occurs in *merge_scc* which now accumulates these sets while popping the nodes from the *Roots* stack. Figure 3 gives a schematic diagram of the merging procedure.

Re-exploring blocked SCCs When a maximal SCC Γ is detected, the set $ub-r$ of the root gives the set of blocking clocks. If $ub-r$ is not empty, Γ needs to be re-explored with the transitions bounding clocks of $blk \cup (ub-r)$ removed. Here blk denotes the blocked clocks when Γ was being explored. Doing this would split Γ into some p smaller maximal SCCs $\Gamma_1, \ldots, \Gamma_p$ such that $\Gamma = \bigcup_{i=1}^{i=p} \Gamma_i$. Each Γ_i is reachable from the root of Γ. Thus, $GZG(\mathcal{A})$ would have an accepting run if some Γ_i contains an unblocked accepting cycle. The objective is to identify and explore the sub SCCs $\Gamma_1, \ldots, \Gamma_p$ on-the-fly. The function *close_scc* is modified to enable on-the-fly re-exploration with blocked clocks.

Firstly, an additional bit per node called *explore* is needed to identify the nodes that have to be explored again. When Γ is detected and *close_scc* called,

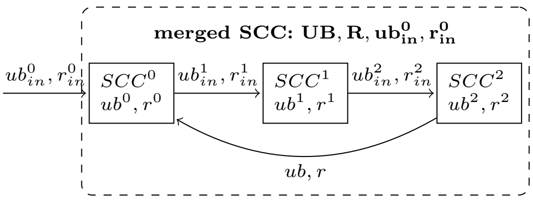
```
function emptiness_check()
count := 0; Roots, Active, Todo := $\emptyset$
check_scc($s_0$,$\emptyset$,$\emptyset$,$\emptyset$)
report $L(\mathcal{A}) = \emptyset$

function check_scc($s$,$ub_{in}$,$r_{in}$,$blk$)
if ($s$.dfsnum = 0)
  count++; $s$.dfsnum := count
$s$.opened := $\top$; $s$.explore := $\bot$
push(Roots, ($s$,$s$.labels,$\emptyset$,$\emptyset$,$ub_{in}$,$r_{in}$))
push(Active, $s$)
for all $s \xrightarrow{g;r} t$ do
  if ($UB(s,g) \cap blk \neq \emptyset$) and ($t \notin$Todo)
    push(Todo, $t$)
  else if ($t$.explore)
    check_scc($t$,$UB(s,g)$,$r$,$blk$)
  else if ($t$.opened)
    merge_scc($UB(s,g)$,$r$,$t$)
if top(Roots) = ($s$,$\cdots$)
  close_scc($blk$)

function merge_scc($ub'_{in}$,$r'_{in}$,$t$)
A:=$\emptyset$; U:=$ub'_{in}$; R:=$r'_{in}$
($s$,$a$,$ub$,$r$,$ub_{in}$,$r_{in}$) := pop(Roots)
while $s$.dfsnum > $t$.dfsnum do
  A:=A$\cup a$; U:=U$\cup ub \cup ub_{in}$
  R:=R$\cup r \cup r_{in}$
  ($s$,$a$,$ub$,$r$,$ub_{in}$,$r_{in}$) := pop(Roots)
A:=A$\cup a$; U:=U$\cup ub$; R:=R$\cup r$
push(Roots, ($s$,A,U,R,$ub_{in}$,$r_{in}$))
if (Acc$\subseteq$A) and (U$\subseteq$R)
  report $L(\mathcal{A}) \neq \emptyset$

function close_scc($blk$)
($s$,$a$,$ub$,$r$,$ub_{in}$,$r_{in}$):=pop(Roots)
repeat
  $u$ := pop(Active)
  $u$.opened := $\bot$
  if (Acc$\subseteq$ $a$) and ($ub \not\subseteq r$)
    $u$.explore := $\top$
until $u = s$
if (Acc$\subseteq$ $a$) and ($ub \not\subseteq r$)
  push(Todo, \$); push(Todo, $s$)
  while top(Todo) $\neq$ \$ do
    $s'$ := pop(Todo)
    if ($s'$.explore)
      check_scc($s'$,$\emptyset$,$\emptyset$,$blk \cup (ub - r)$)
  pop(Todo)
```

Fig. 2. Emptiness Check Algorithm on $GZG(\mathcal{A})$.

the *explore* bit is set to *true* for all the nodes of Γ, if Γ contains blocking clocks. The exploration is started from the root of Γ with the augmented set *blk* of the blocking clocks. Assume that Γ_i is currently being explored with the set *blk*. Each time a transition $s \xrightarrow{g;r} t$ is detected, *check_dfs* checks if it bounds a blocking clock, that is, a clock in *blk*. This way the discarded transitions are detected on-the-fly. Since t may belong to some $\Gamma_j \neq \Gamma_i$, it is added to the *Todo* stack for later re-exploration. The *Todo* stack is the crucial element of the algorithm. The recursion ensures that Γ_i is completely closed before considering a Γ_j on the *Todo* stack. After Γ_i completes, the node t' on the top of the *Todo* stack is considered for exploration. However, it is possible that although t' was added due to a discarded transition, it could now belong to another SCC that has been completely closed, through a different enabled transition. Hence t' is explored only if its *explore* bit is true.

Note that it is possible that more blocking clocks are detected on some Γ_i. The re-exploration on Γ_i then updates the *Todo* stack on-the-fly. However, while popping a node from *Todo*, in order to identify the right set of blocking clocks it has to be explored with, a marker \$ is pushed to the *Todo* stack whenever a new re-exploration with a new set of blocking clocks is started.

Theorem 3. *The above algorithm is correct and runs in time* $\mathcal{O}(|ZG(\mathcal{A})| \cdot |X|^2)$.



$$UB = \bigcup\{ub^0, ub^1_{in}, ub^1, ub^2_{in}, ub^2, ub\}$$
$$R = \bigcup\{r^0, r^1_{in}, r^1, r^2_{in}, r^2, r\}$$

Fig. 3. How ub sets and r sets are merged.

4.2 On-the-fly emptiness check on the zone graph

As stated in section 3.2, the guessing zone graph construction detects nodes where time elapse is not prohibited by future zero checks. However, in the absence of zero checks, this construction is not necessary. Recall that $GZG(\mathcal{A})$ has $|X|+1$ times more nodes than $ZG(\mathcal{A})$. Therefore, it is sufficient to consider the guessing zone graph construction only on maximal SCCs with zero checks. We propose an on-the-fly algorithm that checks for an accepting and unblocked SCC on the zone graph $ZG(\mathcal{A})$, detects zero checks and adopts the emptiness check algorithm on the guessing zone graph only when required. The algorithm shown in Figure 2 is run on $ZG(\mathcal{A})$ instead of $GZG(\mathcal{A})$. When a maximal SCC Γ is detected in line 19, it is required to know if Γ contains any zero checks.

Detecting zero checks This is similar to the detection of blocking clocks. Two extra bits zc and zc_{in} are stored along with every tuple $(s, \dots)$ in the *Roots* stack. zc_{in} tracks zero checks in the transition leading to s and zc remembers if there is a zero check in the SCC rooted at s. When an SCC is detected, the information is merged in *merge_scc* in a way similar to the schematic diagram shown in Figure 3. Thus when Γ is detected, the zc bit of the root reveals the presence of zero checks in Γ. The following lemma says that the algorithm can be terminated if Γ is accepting, unblocked and free from zero-checks.

Lemma 3. *If a reachable SCC in $ZG(\mathcal{A})$ is accepting, unblocked and free from zero-checks, then $\mathcal{A}$ has a non-Zeno accepting run.*

The interesting case occurs when Γ does have zero checks. In this case, we apply the guessing zone graph construction only to the nodes of Γ. If (q^Γ, Z^Γ) is the root of Γ, the guessing zone graph is constructed on-the-fly starting from (q^Γ, Z^Γ, X). Intuitively, we assume that the set of clocks X could potentially be zero at this node and start the exploration. We say that a run ρ of $\mathcal{A}$ *ends* in a maximal SCC Γ of $ZG(\mathcal{A})$ if a suffix of ρ is an instance of a path in Γ. Let $GZG(\mathcal{A})_{|\Gamma}$ be the part of $GZG(\mathcal{A})$ restricted to nodes and transitions that occur in Γ.

Lemma 4. *Let Γ be a reachable maximal SCC in $ZG(\mathcal{A})$, with root (q^Γ, Z^Γ). $\mathcal{A}$ has an accepting non-Zeno run ending in Γ iff $GZG(\mathcal{A})_{|\Gamma}$ has an unblocked accepting SCC reachable from (q^Γ, Z^Γ, X).*

Handling the zero checks The function *close_scc* is modified to identify any zero-checks in the current maximal SCC Γ that was detected. If Γ contains zero checks, then *check_dfs* is called with the node $(q^{\Gamma}, Z^{\Gamma}, X)$ and the current set *blk* of blocking clocks. Note that the *explore* bit is *true* for all the nodes in Γ. Each time a new node $t = (q, Z, Y)$ of $GZG(\mathcal{A})$ is discovered in the for loop $s \xrightarrow{g;r} t$, it is explored only when $(q, Z).explore$ is true. When the exploration of $GZG(\mathcal{A})$ terminates, the *explore* bit of all the nodes of Γ are set to $false$. The *explore* bit is thus responsible for the restriction of the search on $GZG(\mathcal{A})$ only to the nodes that occur in Γ. In particular, this happens on-the-fly.

4.3 An optimization

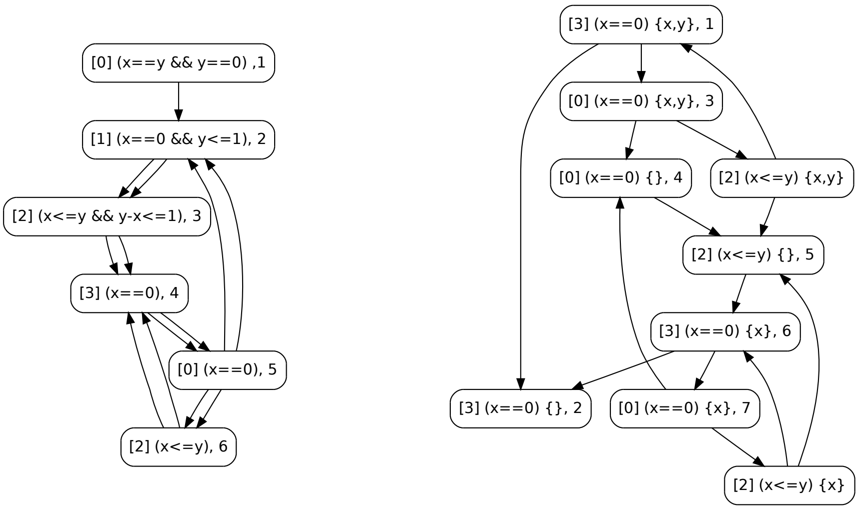
We propose an optimization to the algorithm to enable quicker termination in some cases. We take advantage of clocks in an SCC that are bounded from below, that is of the form $x > c$ or $x \geq c$ with $c \geq 1$. We make use of Lemma 1 which says that if an SCC Γ contains a transition that bounds a clock from below and a transition that resets the same clock, then any path of Γ including both these transitions should have non-Zeno instantiations. This is irrespective of whether Γ contains zero checks. Notice that no new clock is added.

To implement the above optimization, we add clock sets lb and lb_{in} to the *Roots* stack entries. The set lb keeps track of the clocks that are bounded below in the SCC. lb and lb_{in} are handled as explained in Figure 3 for the sets ub and ub_{in}. We also modify the condition for $L(\mathcal{A})$ to be empty. We conclude as soon as an SCC, not necessarily maximal, is accepting and it bounds below and resets the same clock. Observe that it is not useful to keep track of the clocks that are bounded from below while solving zero-checks on $GZG(\mathcal{A})$ as any transition that sets a lower bound in $GZG(\mathcal{A})$ has already been visited in $ZG(\mathcal{A})$.

4.4 The Global Algorithm and Implementation Issues

A run of our algorithm is illustrated in Figure 4. Exploring $ZG(\mathcal{A}_1)$, it computes the maximal SCC rooted at node 2 that is blocked by y. Hence, it re-explores that SCC with set of blocking clocks $blk = \{y\}$. This is seen from the double edges. Notice that the edge from node 5 to node 2 is not crossed as it is discarded by blk. The algorithm now computes the SCC Γ that contains the nodes 4, 5 and 6, that is maximal w.r.t. blk. Γ is unblocked and accepting, but it has zero-checks. Hence the algorithm explores $GZG(\mathcal{A}_1)$ starting from node $(3, x == 0, \{x, y\})$. It does not explore any transition outside of Γ. It eventually finds a non-maximal SCC in $GZG(\mathcal{A}_1)$ that proves $L(\mathcal{A}_1) \neq \emptyset$.

Finally, we discuss memory issues for the implementation of our algorithm. Information about the SCCs in $ZG(\mathcal{A})$ is stored in the *Roots* stack as tuples $(s, a, lb, ub, r, zc, lb_{in}, ub_{in}, r_{in}, zc_{in})$. An amount of $6.|X| + 2$ bits of memory is required to store the membership of each clock to the 6 sets, and the zero-check information. In the worst case, all the nodes in $ZG(\mathcal{A})$ have an entry in the *Roots* stack. Similarly, SCCs in $GZG(\mathcal{A})$ are stored as tuples $(s, a, ub, r, ub_{in}, r_{in})$ in



$\mathbf{ZG}(\mathcal{A}_1)$ Part of $\mathbf{GZG}(\mathcal{A}_1)$

Fig. 4. Zone Graphs of $\mathcal{A}_1$.

the *Roots* stack. $4.|X|$ bits allow to represent the content of those 4 sets. We estimate than the overhead due to our algorithm is $4.|X|.(|X|+1)$ due to the size of $GZG(\mathcal{A})$ w.r.t. the size of $ZG(\mathcal{A})$.

We eventually compare to the strongly non-Zeno construction. Running the Couvreur's algorithm requires to store smaller tuples (s, a). However, the zone in s is represented by a DBM which is an $|X|+1$-square matrix containing 32 bits integers. Hence, adding one clock increases the memory footprint by $(2.|X|+3).32$ bits. Recall furthermore than $ZG(SNZ(\mathcal{A}))$ can be $2^{\mathcal{O}(|X|)}$ bigger than $ZG(\mathcal{A})$. In the worst case all these nodes are on the *Roots* stack.

5 Experiments and Conclusion

We have implemented our algorithms in a prototype verification tool. The table below presents the results that we obtained on several classical examples. The models are products of Timed Büchi Automata that encode both the processes in the system and the property to verify. The "Zone Graph" column gives the number of nodes in the zone graph. Next, for the "Strongly non-Zeno" construction, we give the size of the resulting zone graph followed by the number of nodes that are visited during verification; similarly for the "Guessing Zone Graph" where the last column corresponds to our fully optimized algorithm.

We have considered three types of properties: reachability properties (mutual exclusion, collision detection for CSMA/CD), liveness properties (access to the resource infinitely often), and bounded response properties (which are reachability properties with real-time requirements). When checking reachability properties we would like counter examples to be feasible by real systems.

This is not the case with Zeno counter examples. Hence Timed Büchi Automata offer a nice framework as they discard Zeno runs.

The strongly non-Zeno construction outperforms the guessing zone graph construction for reachability properties. This is particularly the case for mutual exclusion on the Fischer's protocol and collision detection for the CSMA/CD protocol. For liveness properties, the results are mitigated. On the one hand, the strongly non-Zeno property is once again more efficient for the CSMA/CD protocol. On the other hand the differences are tight in the case of Fischer protocol. The guessing zone graph construction distinguishes itself for bounded response properties. Indeed, the Train-Gate model is an example of exponential blowup for the strongly non-Zeno construction.

We notice that on-the-fly algorithms perform well. Even when the graphs are big, particularly in case when automata are not empty, the algorithms are able to conclude after having explored only a small part of the graph. Our optimized algorithm outperforms the two others on most examples. Particularly, for the CSMA/CD protocol with 5 stations our algorithm needs to visit only 4841 nodes while the two other methods visited 8437 and 21038 nodes. This confirms our initial hypothesis: most of the time, the zone graph contains enough information to ensure time progress.

Our last optimization also proves useful for the FDDI protocol example. One of its processes has zero checks, but since some other clock is bounded from below and reset, it was not necessary to explore the guessing zone graph to conclude non-emptiness.

Models	Zone Graph	Strongly non-Zeno		Guessing Zone Graph		
	size	size	visited	size	visited	visited opt.
Train-Gate2 (mutex)	134	194	194	400	400	134
Train-Gate2 (bound. resp.)	988	227482	352	3840	1137	292
Train-Gate2 (liveness)	100	217	35	298	53	33
Fischer3 (mutex)	1837	3859	3859	7292	7292	1837
Fischer4 (mutex)	46129	96913	96913	229058	229058	46129
Fischer3 (liveness)	1315	4962	52	5222	64	40
Fischer4 (liveness)	33577	147167	223	166778	331	207
FDDI3 (liveness)	508	1305	44	3654	79	42
FDDI5 (liveness)	6006	15030	90	67819	169	88
FDDI3 (bound. resp.)	6252	41746	59	52242	114	60
CSMA/CD4 (collision)	4253	7588	7588	20146	20146	4253
CSMA/CD5 (collision)	45527	80776	80776	260026	260026	45527
CSMA/CD4 (liveness)	3038	9576	1480	14388	3075	832
CSMA/CD5 (liveness)	32751	120166	8437	186744	21038	4841

Conclusion and Future Work. In this paper, we have presented an algorithm for the emptiness problem of Timed Büchi Automata. We claim that our algorithm is on-the-fly as (1) it computes the zone graph during the emptiness check; (2) it does not store the graph, in particular when it splits blocked SCCs; and (3) it stops as soon as an SCC that contains a non-Zeno accepting run is found. Our algorithm is inspired from the Couvreur's algorithm. However, the handling of bounded clocks and the application of the guessing zone graph construction on-the-fly, are two substantial increments. The previous examples are academic case studies; still they show that Büchi properties can be checked as efficiently as reachability properties for Timed Automata, despite the non-Zenoness requirement.

As a future work we plan to extend our algorithm to commonly used syntactic extensions of Timed Automata. For instance Uppaal and Kronos allow to reset clocks to arbitrary values, which is convenient for modeling real life systems. This would require to extend the guessing zone graph construction, and consequently our algorithm. Another interesting question is to precisely characterize the situations where an exponential blowup occurs in the strongly non-Zeno construction. To conclude, it can be seen that the ideas of the prototype could be used to construct a full-fledged tool like Profounder, which implements the strongly non-Zeno construction.

References

1. R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. R. Alur and P. Madhusudan. Decision problems for timed automata: A survey. In *SFM-RT'04*, volume 3185 of *LNCS*, pages 1–24, 2004.
3. G. Behrmann, A. David, K. G Larsen, J. Haakansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *QEST'06*, pages 125–126, 2006.
4. B. Bérard, B. Bouyer, and A. Petit. Analysing the pgm protocol with UPPAAL. *Int. Journal of Production Research*, 42(14):2773–2791, 2004.
5. B. Berthomieu and M. Menasche. An enumerative approach for analyzing time petri nets. In *IFIP Congress*, pages 41–46, 1983.
6. A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly symbolic model checking for real-time systems. In *RTSS'97*, pages 25–, 1997.
7. P. Bouyer. Forward analysis of updatable timed automata. *Formal Methods in System Design*, 24(3):281–320, 2004.
8. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. KRONOS: a mode-checking tool for real-time systems. In *CAV'98*, volume 1427 of *LNCS*, pages 546–550. Springer, 1998.
9. J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized büchi automata. In *SPIN'05*, volume 3639 of *Lecture Notes in Computer Science*, pages 169–184, 2005.
10. C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *TACAS'98*, volume 1384 of *LNCS*, pages 313–329, 1998.
11. D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proc. Int. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer, 1990.
12. F. Herbreteau, B. Srivathsan, and I. Walukiewicz. Efficient emptiness check for timed büchi automata. In *CAV'10*, 2010. to appear.
13. S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In *TACAS'05*, volume 3440 of *LNCS*, pages 174–190, 2005.
14. S. Tripakis. Verifying progress in timed systems. In *Formal Methods for Real-Time and Probabilistic Systems*, volume 1601 of *LNCS*, pages 299–314, 1999.
15. S. Tripakis. Checking timed büchi emptiness on simulation graphs. *ACM Transactions on Computational Logic*, 10(3), 2009.
16. S. Tripakis, S. Yovine, and A. Bouajjani. Checking timed büchi automata emptiness efficiently. *Formal Methods in System Design*, 26(3):267–292, 2005.