Why Liveness for Timed Automata Is Hard, and What We Can Do About It

Frédéric Herbreteau, B. Srivathsan, Tran Thanh Tung, Igor Walukiewicz

Why liveness for timed automata is hard, and what we can do about it

FRÉDÉRIC HERBRETEAU, Université de Bordeaux, Bordeaux INP, CNRS, LaBRI, UMR 5800, Talence, France and UMI 2000 ReLaX
B. SRIVATHSAN, Chennai Mathematical Institute, Chennai, India and UMI 2000 ReLaX
THANH-TUNG TRAN, School of Computer Science and Engineering, International University, Vietnam National University, Ho Chi Minh city (VNU-HCM), Viet Nam
IGOR WALUKIEWICZ, Université de Bordeaux, Bordeaux INP, CNRS, LaBRI, UMR 5800, Talence, France and UMI 2000 ReLaX

The *reachability problem* for timed automata asks if a given automaton has a run leading to an accepting state, and the *liveness problem* asks if the automaton has an infinite run which visits accepting states infinitely often. Both these problems are known to be PSPACE-complete.

We show that if P≠PSPACE, the liveness problem is more difficult than the reachability problem; in other words we exhibit a family of automata for which solving the reachability problem with the standard algorithm is in P but solving the liveness problem is PSPACE-hard. This leads us to revisit the algorithmics for the liveness problem. We propose a notion of a witness for the fact that a timed automaton violates a liveness property. We give an algorithm for computing such a witness and compare it with existing solutions.

CCS Concepts: • **Theory of computation → Verification by model checking**.

Additional Key Words and Phrases: Timed automata, liveness verification, complexity, algorithms

ACM Reference Format:
Frédéric Herbreteau, B. Srivathsan, Thanh-Tung Tran, and Igor Walukiewicz. 2019. Why liveness for timed automata is hard, and what we can do about it. *ACM Trans. Comput. Logic* 1, 1 (November 2019), 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Timed automata [1] are one of the standard models for timed systems. There has been an extensive body of work on the verification of reachability/safety properties of timed automata. In contrast, advances on verification of liveness properties are much less spectacular. For verification of liveness properties expressed in a logic like Linear Temporal Logic, it is best to consider a slightly more general problem of verification of Büchi properties. This means verifying if in a given timed automaton there is an infinite path passing through an accepting state infinitely often.

Testing Büchi properties of timed systems can be surprisingly useful. We give an example in Section 6 where we describe how with a simple liveness test one can discover a typo in the benchmark CSMA/CD model [16, 18]. This typo removes practically all the interesting behaviours from the model. Yet the CSMA/CD benchmark has been extensively used for evaluating verification tools, and nothing unusual has been observed. Therefore, even if one is interested solely in verification of safety properties, it is important to "test" the model under consideration, and for this Büchi properties are very useful.

1529-3785/2019/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Verification of reachability properties of timed automata is known to be PSPACE-complete [1]. In practice, reachability analysis is possible thanks to the so-called zones and their abstractions [3, 4, 10, 12]. Roughly, the standard approach used nowadays for reachability/safety properties performs a breadth first search (BFS) over the set of pairs (state, zone) reachable in the automaton, storing only pairs with the maximal abstracted zones (with respect to inclusion, called subsumption in this context). In jargon: the algorithm constructs a *zone graph with subsumption*.

Verifying Büchi properties for timed automata is also known to be PSPACE-complete [1]. In this paper, we give strong evidence that verification of Büchi properties is inherently more difficult than verification of reachability properties. For a long time it has been understood that for liveness checking, there is a problem with the approach outlined above - keeping only the maximal zones with respect to inclusion (i.e. zone graph with subsumption) is no longer sound [13, 15]. It is possible to use the zone graph without subsumption, but this one is almost always too big to handle. One could however hope that some modification of the notion of zone graph with subsumption can give an algorithm for Büchi properties that is provably not much more costly than the algorithm for reachability properties. We show that this is unlikely. We present a family of automata for which the size of the zone graph with subsumption is linear in the size of the automata and hence reachability can be decided in P; however deciding existence of a Büchi accepting run for these automata amounts to solving the halting problem for Linear Bounded Automata. This proves that unless P=PSPACE, there is no hope of obtaining an algorithm for Büchi properties that has provably similar complexity to the standard reachability algorithm (which constructs zone graph with subsumption).

Our goal in this paper is to rethink the foundations of verification of Büchi properties for timed automata, and propose some algorithmic solutions. The first question we address is this: what can be a witness to the fact that an automaton has no Büchi accepting run? As we have mentioned above, for reachability properties such a witness is a zone graph with subsumption. We propose a similar notion of a witness for Büchi properties that allows only "safe" subsumptions. As the next contribution, we give an algorithm for computing such a witness. Due to the hardness result mentioned above, we cannot hope to have as efficient an algorithm as for reachability. We propose an algorithm that will iteratively apply the reachability algorithm. It will first construct the zone graph with subsumption, stopping if it finds a Büchi run. If all subsumptions in this graph are safe according to our definition then this graph forms a witness for non-existence of a Büchi run. Otherwise the algorithm recursively refines strongly connected components of the graph with unsafe subsumptions. This algorithm computes the zone graph without subsumption in the worst case - this as we show is anyway the best that can be done in some cases. The expected advantage is that in many cases our algorithm can stop sooner. We have implemented our algorithm and tested it on a set of benchmarks from [13]. The results show that indeed the algorithm mostly stops after the first iteration, and constructs witnesses of size very close to those for safety.

A preliminary version of this work appears in [7], where we first prove that if P≠ NP, liveness is more difficult than reachability for timed automata and then give an iterative algorithm to compute a witness for a Büchi accepting run. Here we strengthen the complexity result: if P≠ PSPACE, liveness is more difficult than reachability. We also give a modified iterative algorithm for witness detection and compare its performance with the earlier version.

Related work: Verification of Büchi properties is decidable thanks to the region construction [1]. The use of zones and certain abstractions for this problem was developed in [15]. Later Li [14] has shown that existence of a Büchi run is preserved by every abstraction based on simulation relations. In particular, this is the case for the $\mathfrak{a}_{\preccurlyeq LU}$ abstraction [3], which is the coarsest abstraction depending only on lower and upper bounds in clock guards (LU-bounds) [8]. Thanks to these results liveness checking can be done on an abstract zone graph using $\mathfrak{a}_{\preccurlyeq LU}$ abstraction (but without subsumption).

The question of whether subsumption can be used to improve the liveness verification was raised in [15]. Laarman et al. [13] recently proposed a nested DFS based algorithm for checking Büchi properties of timed automata. They study in depth when it is sound to use subsumption in the nested DFS algorithm. Our conditions on the use of subsumption are expressed in terms of zone graphs and are independent of a particular algorithm. This allows us to focus on the task of finding a witness graph efficiently; in particular we can use BFS based algorithms for the task. We give a more detailed comparison of the two algorithms in Section 6.

Organization of the paper: In the next section we present the basic definitions, as well as the algorithms for constructing the abstract zone graph, and the abstract zone graph with subsumption. We also describe the nested DFS algorithm from [13]. In Section 3 we give our notion of a witness for non-existence of a Büchi run in a given automaton. Section 4 presents a theorem which exhibits the above stated algorithmic difference between verification of liveness and reachability properties. In Section 5 we give an algorithm for finding witnesses for non-existence of a Büchi run. In Section 6 we report on some experimental results.

2 PRELIMINARIES

In this section we present the definitions of timed Büchi automata, the Büchi non-emptiness problem and the abstract zone graphs used for deciding non-emptiness. We also describe the subsumption optimization and the standard algorithm for constructing an abstract zone graph with subsumption. This can be used to answer reachability properties. We finish this section with the nested DFS algorithm for Büchi properties [13].

2.1 Timed Büchi automata

Let $\mathbb{R}_{\geq 0}$ denote the set of non-negative reals. A *clock* is a variable that ranges over $\mathbb{R}_{\geq 0}$. Let $X = \{x_1, \ldots, x_n\}$ be a set of clocks. A *valuation* is a function $v : X \to \mathbb{R}_{\geq 0}$. The set of all clock valuations is denoted by $\mathbb{R}^X_{\geq 0}$. We denote by $\mathbf{0}$ the valuation that associates 0 to every clock in X. A *clock constraint* ϕ is a conjunction of constraints of the form $x \sim c$ where $x \in X$, $\sim \in \{<, \leq, =, \geq, >\}$ and $c \in \mathbb{N}$. Let $\Phi(X)$ denote the set of clock constraints over the set of clocks X. A valuation v is said to satisfy a constraint ϕ, written as $v \vDash \phi$, when every constraint in ϕ holds after replacing every x by $v(x)$. For $\delta \in \mathbb{R}_{\geq 0}$, let $v + \delta$ be the valuation that associates $v(x) + \delta$ to every clock x. For $R \subseteq X$, let $[R]v$ be the valuation that sets x to 0 if $x \in R$, and that sets x to $v(x)$ otherwise.

Definition 2.1 (Timed Büchi Automata (TBA) [1]). A *Timed Büchi Automaton* is a tuple $\mathcal{A} = (Q, q_0, X, T, F)$ in which Q is a finite set of states, q_0 is the initial state, X is a finite set of clocks, $F \subseteq Q$ is a set of accepting states, and $T \subseteq Q \times \Phi(X) \times 2^X \times Q$ is a finite set of transitions of the form (q, g, R, q') where g is a clock constraint called the *guard*, and R is a set of clocks that are *reset* on the transition from q to q'.

The semantics of a TBA $\mathcal{A} = (Q, q_0, X, T, F)$ is given by a transition system of its configurations. A *configuration* of $\mathcal{A}$ is a pair $(q, v) \in Q \times \mathbb{R}^X_{\geq 0}$, with $(q_0, \mathbf{0})$ being the initial configuration. There are two kinds of transitions:

- **delay:** $(q, v) \to^{\delta} (q, v + \delta)$ for $\delta \in \mathbb{R}_{\geq 0}$;
- **action:** $(q, v) \to^{t} (q', v')$ for $t = (q, g, R, q') \in T$ such that $v \vDash g$ and $v' = [R]v$.

A *run* of $\mathcal{A}$ is a (finite or infinite) sequence of transitions starting from the initial configuration: $(q_0, \mathbf{0}) \xrightarrow{\delta_0, t_0} (q_1, v_1) \xrightarrow{\delta_1, t_1} \cdots$, where $(q, v) \xrightarrow{\delta, t} (q', v')$ denotes a delay δ followed by action t starting from $(q, v + \delta)$. A configuration (q, v) is said to be *accepting* if $q \in F$. An infinite run *satisfies the Büchi condition* if it visits accepting configurations infinitely often. The run is *Zeno* if its

accumulated duration is finite, i.e., $\sum_{i\geq 0} \delta_i \leq c$ for some $c \in \mathbb{R}_{\geq 0}$. Else it is *non-Zeno*. The problem we are interested in is termed the *Büchi non-emptiness problem*.

Definition 2.2. The *Büchi non-emptiness problem* for TBA is to decide if a given TBA $\mathcal{A}$ has a non-Zeno run satisfying the Büchi condition.

Example 2.3. Figure 1 gives an example of a TBA $\mathcal{A}_1$ which has a Büchi accepting run: for instance the run which leaves state 0 and reaches state 1 at time unit 1, after which the loop transition at state 1 is taken at intervals of 1 time unit. Figure 4 gives a TBA $\mathcal{A}_2$ for which there is no Büchi accepting run. The reset of x and the guard $x \geq 1$ imply that between two consecutive visits to state 1, at least 1 time unit should have elapsed. Since y is never reset, the value of y keeps increasing by at least 1 unit at each arrival to state 1. Once y becomes > 100, the guard $y \leq 100$ disables the transition from 1 to 0.

The Büchi non-emptiness problem is known to be PSPACE-complete [1]. Standard solutions to this problem use regions or zones: they construct an untimed Büchi automaton and check for its emptiness. There are various methods to handle the non-Zeno requirement [9, 17].

REMARK 1. *In this paper, we will assume that the automata are strongly non-Zeno [15], that is, every infinite accepting run is non-Zeno. It is possible to convert every TBA into a strongly non-Zeno TBA. This strongly non-Zeno construction could lead to an exponential blowup [6, 9] to the abstract zone graph (which is defined below), but we prefer to employ this commonly used assumption in order not to divert from the main subject.*

We will now describe a translation which reduces the Büchi non-emptiness problem to checking non-emptiness of an untimed Büchi automaton.

2.2 Abstract zone graphs

As the semantics of a TBA is an infinite transition system, algorithms for TBA consider special sets of valuations called *zones*. A zone is a set of valuations described by a conjunction of two kinds of constraints: either $x_i \sim c$ or $x_i - x_j \sim c$ where $x_i, x_j \in X$, $c \in \mathbb{Z}$ and $\sim \in \{<, \leq, =, >, \geq\}$. For example $(x_1 > 3 \wedge x_2 - x_1 \leq -4)$ is a zone. Zones can be efficiently represented by Difference Bound Matrices (DBMs) [5].

The *zone graph* $ZG(\mathcal{A})$ of a TBA $\mathcal{A} = (Q, q_0, X, T, F)$ is a directed graph whose nodes are pairs of the form (q, Z) consisting of a state q of $\mathcal{A}$ and a zone Z. The initial node is (q_0, Z_0) with $Z_0 = \{\mathbf{0} + \delta \mid \delta \in \mathbb{R}_{\geq 0}\}$. For every transition $t = (q, g, R, q') \in T$, and every set of valuations W, we define the transition $\Rightarrow^t$ as: $(q, W) \Rightarrow^t (q', W')$ where $W' = \{v' \mid \exists v \in W,\ \exists \delta \in \mathbb{R}_{\geq 0} : (q, v) \rightarrow^t \rightarrow^\delta (q', v')\}$. In other words, W' is obtained by first computing the $\rightarrow^t$ successors of W, followed by all time successors. It can be shown that if W is a zone, then so is W'. In the zone graph, from every node (q, Z) there is a transition $(q, Z) \Rightarrow^t (q', Z')$ corresponding to the transitions t from q. The transition relation $\Rightarrow$ is the union of $\Rightarrow^t$ over all $t \in T$.

Observe that there is a slight difference in the definition of transitions in the automaton $(q, v) \xrightarrow{\delta, t} (q', v')$ where we first have a delay δ, followed by a transition t, and the definition of transitions in the zone graph: $(q, W) \Rightarrow^t (q', W')$ where W' is the set of valuations obtained from W by first taking transition t, and *then* doing a delay. The two definitions match since the initial zone encompasses an initial delay from the initial valuation $\mathbf{0}$.

Although the zone graph $ZG(\mathcal{A})$ groups together valuations, the number of zones is still infinite [4]. Figure 1 shows an example of a TBA whose zone graph consists of infinitely many nodes reachable from the initial node. For effectiveness, zones are further abstracted. Let us write $\mathcal{P}(S)$ for the set of subsets of S. An *abstraction operator* is a function $\mathfrak{a} : \mathcal{P}(\mathbb{R}_{\geq 0}^{|X|}) \rightarrow \mathcal{P}(\mathbb{R}_{\geq 0}^{|X|})$ such that

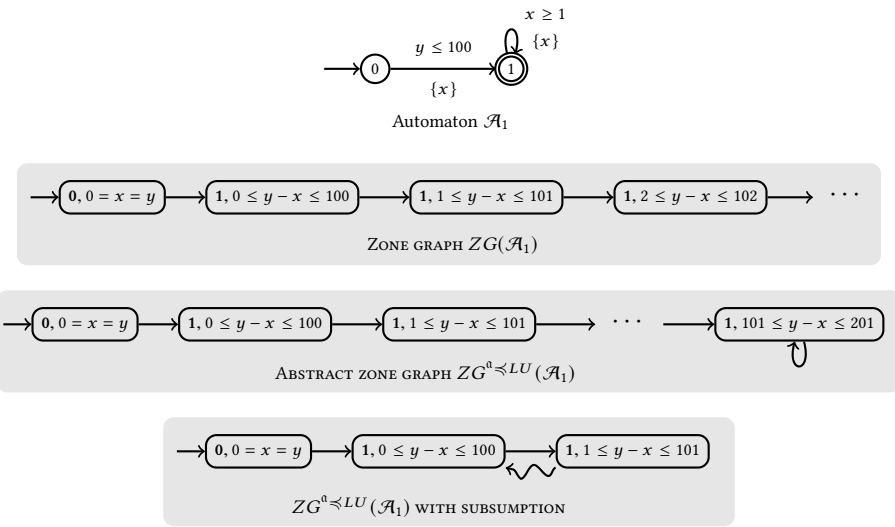

Fig. 1. Example of an automaton whose zone graph is infinite. The abstract zone graph is computed with $\mathfrak{a}_{\preccurlyeq LU}$ abstraction, taking $L(x) = 1, U(x) = -\infty, L(y) = -\infty, U(y) = 100$. It contains 103 nodes. The zone graph with subsumption contains only 3 nodes. The "squiggly" edge denotes the subsumption: $\mathfrak{a}_{\preccurlyeq LU}(1 \leq y - x \leq 101) \subseteq \mathfrak{a}_{\preccurlyeq LU}(0 \leq y - x \leq 100)$.

$W \subseteq \mathfrak{a}(W)$ and $\mathfrak{a}(\mathfrak{a}(W)) = \mathfrak{a}(W)$ for every set of valuations $W \in \mathcal{P}(\mathbb{R}_{\geq 0}^{|X|})$. The abstraction is *finite* if $\mathfrak{a}$ has a finite range. An abstraction operator defines an abstract symbolic semantics defined by: $(q, W) \Rightarrow_{\mathfrak{a}}^{t} (q', \mathfrak{a}(W'))$ when $\mathfrak{a}(W) = W$ and $(q, W) \Rightarrow^{t} (q', W')$. Essentially, each time a successor W' is computed, it is abstracted to $\mathfrak{a}(W')$. We define a transition relation $\Rightarrow_{\mathfrak{a}}$ to be the union of $\Rightarrow_{\mathfrak{a}}^{t}$ over all transitions t. For a finite abstraction operator $\mathfrak{a}$, the *abstract zone graph* $ZG^{\mathfrak{a}}(\mathcal{A})$ consists of node pairs (q, W) of the form $W = \mathfrak{a}(W)$. The initial node is $(q_0, \mathfrak{a}(Z_0))$ where (q_0, Z_0) is the initial node of $ZG(\mathcal{A})$. Transitions are given by the $\Rightarrow_{\mathfrak{a}}$ relation. Such a graph $ZG^{\mathfrak{a}}(\mathcal{A})$ can be seen as a Büchi automaton with the accepting states (q, W) for $q \in F$.

2.2.1 The $\mathfrak{a}_{\preccurlyeq LU}$ abstraction. Abstractions for timed automata are typically parameterized by the maximum of constants appearing in the guards of the automaton. The structure of the automaton determines two functions $L : X \mapsto \mathbb{N} \cup \{-\infty\}$ and $U : X \mapsto \mathbb{N} \cup \{-\infty\}$. For a clock x, the value $L(x)$ denotes the maximum constant occurring in guards of the form $x \geq c$ or $x > c$; if no such guard exists then $L(x) = -\infty$. The value $U(x)$ denotes the maximum constant occurring in guards $x \leq c$ or $x < c$; if no such guard exists then $U(x) = -\infty$. This can be further refined by considering LU bounds for each state of the automaton [2]. In this paper we will use the abstraction operator $\mathfrak{a}_{\preccurlyeq LU}$ [3] and the abstract zone graph $ZG^{\mathfrak{a}_{\preccurlyeq LU}}(\mathcal{A})$ induced by it. Another abstraction operator Extra_{LU}^{+} is commonly used in timed automata tools [3]. However, it is known that $\mathfrak{a}_{\preccurlyeq LU}$ is a coarser abstraction than Extra_{LU}^{+} and hence it could potentially lead to smaller abstract zone graphs [3]. We will therefore stick to $\mathfrak{a}_{\preccurlyeq LU}$ abstraction in this document. In order to define $\mathfrak{a}_{\preccurlyeq LU}$, we need to first define a simulation pre-order on valuations.

Definition 2.4 (LU-preorder [3]). Let $L, U : X \mapsto \mathbb{N} \cup \{-\infty\}$ be two bound functions. For a pair of valuations we set $v \preccurlyeq_{LU} v'$ if for every clock x:

- if $v'(x) < v(x)$ then $v'(x) > L(x)$, and
- if $v'(x) > v(x)$ then $v(x) > U(x)$.

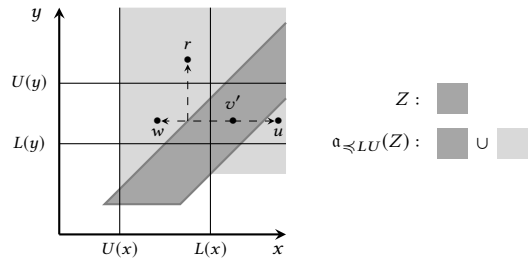

Fig. 2. The dark gray portion shows a zone Z. The light gray shows the valuations added in the $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction of Z, with respect to the LU bounds marked in the picture. Each valuation in the light gray zone is simulated by a valuation in the zone Z w.r.t. $\preccurlyeq_{LU}$. The abstraction $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$ is given by the union of the shaded portions. The picture illustrates a valuation v' such that $r \preccurlyeq_{LU} v'$, $u \preccurlyeq_{LU} v'$ and $w \preccurlyeq_{LU} v'$.

The $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction is based on this relation.

Definition 2.5 ($\mathfrak{a}_{\preccurlyeq_{LU}}$-abstraction [3]). Given L and U bound functions, for a set of valuations W we define:

$$\mathfrak{a}_{\preccurlyeq_{LU}}(W) = \{v \mid \exists v' \in W.\ v \preccurlyeq_{LU} v'\}.$$

Figure 2 gives an example of a zone Z and its abstraction $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$. Consider the valuation v' in the figure. Note that $U(x) < L(x) < v'(x)$, and $L(y) < v'(y) < U(y)$. Hence, for a valuation v to be simulated by v', i.e. for $v \preccurlyeq_{LU} v'$ in the definition above, we need $U(x) < v(x)$, and $v'(y) \leq v(y)$. We cannot have $v'(y) > v(y)$ as this requires $v(y) > U(y)$, that is impossible due to $v'(y) < U(y)$. Valuations r, u and w in the figure satisfy these criteria. In fact, $\mathfrak{a}_{\preccurlyeq_{LU}}(v')$ is the set of valuations given by $x > U(x)$ and $y \geq v'(y)$.

Figures 1 and 4 give examples of abstract zone graphs obtained using $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction. It was shown in [8] that the $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction induces the smallest zone graphs, for a given bound function LU. Moreover, we know from [14] that $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$ is sound and complete for Büchi non-emptiness:

Theorem 2.6. *[14] A TBA $\mathcal{A}$ has a run satisfying the Büchi condition iff the abstract zone graph $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$ has a run satisfying the Büchi condition.*

This gives an algorithm for the Büchi non-emptiness problem: given a TBA $\mathcal{A}$, compute the (finite) Büchi automaton $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$ and check for its emptiness. There is however a challenge due to the use of the $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction. There are zones Z for which $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$ is non-convex (c.f. Figure 2) and hence it is better to avoid storing $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$. Therefore, the solution to compute $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$ works with a graph consisting of (state, zone) pairs and uses the $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction indirectly [8]. The algorithm for computing $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$ is shown in Figure 3. We will denote the transition relation $\Rightarrow$ by $\rightarrow$ for convenience, as shown by the edge relation in Figure 3.

We now state a lemma which makes the optimization described in the next section possible.

Lemma 2.7 (Simulation property of $\mathfrak{a}_{\preccurlyeq_{LU}}$-abstraction). *Let $\mathcal{A}$ be a TBA, and LU be bound functions computed from guards of $\mathcal{A}$. For every pair of nodes (q, Z) and (q, Z') of $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$: if $\mathfrak{a}_{\preccurlyeq_{LU}}(Z) \subseteq \mathfrak{a}_{\preccurlyeq_{LU}}(Z')$, then for every transition $(q, Z) \rightarrow (q_1, Z_1)$, there exists $(q, Z') \rightarrow (q_1, Z_1')$ such that $\mathfrak{a}_{\preccurlyeq_{LU}}(Z_1) \subseteq \mathfrak{a}_{\preccurlyeq_{LU}}(Z_1')$.*

We refer the reader to [3] and [12] for a proof of the above lemma, and a more detailed account of the $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction (both of which are not required for the rest of this paper). We however present a brief intuition about why this property holds. Roughly, we want a relation between valuations v and v' which ensures that whenever v satisfies a guard g, valuation v' also satisfies g (and hence

```
1  procedure abstract_zone_graph(𝒜)
2    V := {(q₀, Z₀)}, Waiting := {(q₀, Z₀)}
3    → := ∅ // edge relation
4    while (Waiting ≠ ∅)
5      take and remove (q, Z) from Waiting
6      for each t = (q, g, R, q') ∈ 𝒜
7        compute (q, Z) ⇒ᵗ (q', Z')
8        if ∃(q', Z₁) ∈ V s.t 𝔞_{≼LU}(Z') = 𝔞_{≼LU}(Z₁)
9          add (q, Z) → (q', Z₁)
10       else
11         add (q', Z') to V and Waiting
12         add (q, Z) → (q', Z')
13   return (V, →)
14
15
16
17
18
```

```
19 procedure subsumption_graph(𝒜)
20   V := {(q₀, Z₀)}, Waiting := {(q₀, Z₀)}
21   → := ∅ // edge relation
22   ⇝ := ∅ // subsumption relation
23   while (Waiting ≠ ∅)
24     take and remove (q, Z) from Waiting
25     for each t = (q, g, R, q') ∈ 𝒜
26       compute (q, Z) ⇒ᵗ (q', Z')
27       if ∃(q', Z₁) ∈ V s.t 𝔞_{≼LU}(Z') = 𝔞_{≼LU}(Z₁)
28         add (q, Z) → (q', Z₁)
29       else if ∃(q', Z₁) ∈ V s.t. 𝔞_{≼LU}(Z') ⊆ 𝔞_{≼LU}(Z₁)
30         add (q', Z') to V
31         add (q, Z) → (q', Z') and (q', Z') ⇝ (q', Z₁)
32       else
33         add (q', Z') to V and Waiting
34         add (q, Z) → (q', Z')
35   return (V, →, ⇝)
```

Fig. 3. Algorithm on the left computes $ZG^{\mathfrak{a}_{\preccurlyeq LU}}(\mathcal{A})$. The test $\mathfrak{a}_{\preccurlyeq LU}(Z') = \mathfrak{a}_{\preccurlyeq LU}(Z_1)$ can be done using the method in [8]. On the right is an algorithm which uses subsumption.

runs from v can be *simulated* by runs from v'). The LU bound functions say that all lower bound guards $x \geq c, x > c$ have $c \leq L(x)$ and all upper bound guards $x \leq c, x < c$ have $c \leq U(x)$. Hence, when $v'(x) < v(x)$, a problem arises when a lower bound guard g is of the form $x \geq c$ or $x > c$ with $v'(x) \leq c \leq v(x)$ as this will lead to v satisfying g, but v' violating g; in this case v' cannot simulate v. This motivates the first condition in the definition: if $v'(x) < v(x)$ then $v'(x) > L(x)$. Similar reasoning holds for the symmetric case.

For a node $n \in ZG^{\mathfrak{a}_{\preccurlyeq LU}}(\mathcal{A})$ we write $n.q$ and $n.Z$ for the state and zone present in node n respectively.

2.3 Using subsumption to compute smaller graphs

Although $ZG^{\mathfrak{a}_{\preccurlyeq LU}}(\mathcal{A})$ is the smallest abstract zone graph for a given LU, its size could be (and usually is) exponential in the size of $\mathcal{A}$. An essential optimization that makes analysis of timed automata feasible is the use of subsumption. For two nodes t and s of $ZG^{\mathfrak{a}_{\preccurlyeq LU}}(\mathcal{A})$ we say t *is subsumed by* s, written as $t \sqsubseteq s$, if $t.q = s.q$ and $\mathfrak{a}_{\preccurlyeq LU}(t.Z) \subseteq \mathfrak{a}_{\preccurlyeq LU}(s.Z)$. When $t \sqsubseteq s$, the node s simulates t. Hence, at least for testing reachability, it is enough to keep in the graph only the maximal nodes with respect to subsumption. The algorithm incorporating subsumption is shown in the right hand side of Figure 3.

Subsumption optimization is known to give substantial gains for the reachability problem [11]. However, subsumption is not a priori correct for liveness: in Figure 1, the zone graph with subsumption contains no cycle consisting entirely of $\rightarrow$ edges but the zone graph has one. Admitting cycles with subsumption edges is not sound either: Figure 4 illustrates an example of an automaton whose zone graph has no cycles, but zone graph with subsumption has a cycle consisting of $\rightarrow$ and $\rightsquigarrow$ edges. Therefore, it is not immediately clear how to decide Büchi non-emptiness from subsumption graphs (in other words, zone graphs with subsumption edges). The question of how subsumption graphs can be used for Büchi non-emptiness was raised in [15]. An algorithm proposed in [13]

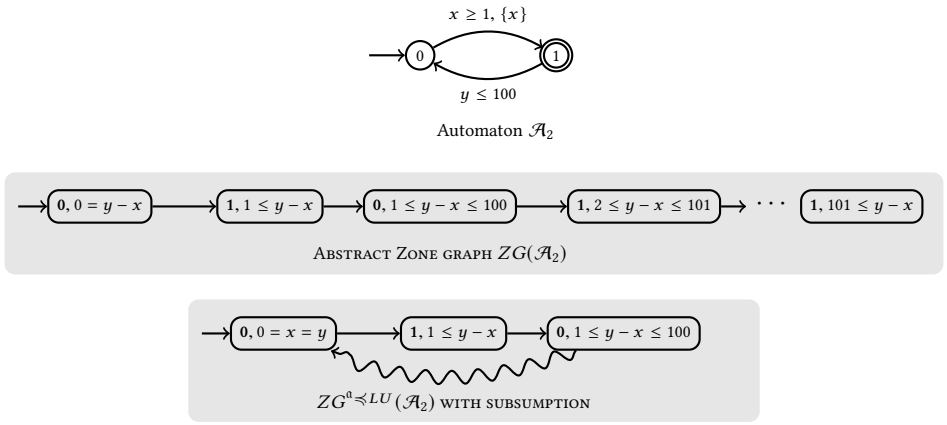

Fig. 4. Example of an automaton with no Büchi accepting run. Abstract zone graph is computed with $L(x) = 1, U(x) = -\infty, L(y) = -\infty, U(y) = 100$. There are no cycles in abstract zone graph. Allowing subsumption gives a smaller graph. However the graph with subsumption has an infinite path that does not correspond to any run of $\mathcal{A}_2$. In general, counting subsumption edge as part of cycles is incorrect for Büchi emptiness.

```
1  procedure ndfs()
2    Cyan := Blue := Red := ∅
3    dfsBlue(s_0)
4    report no cycle
5
6  procedure dfsRed(s)
7    Red := Red ∪ {s}
8    for all s → t do
9      if (Cyan ⊑ t) then report cycle
10     if (t ⋢ Red) then dfsRed(t)
11
12 procedure dfsBlue(s)
13   Cyan := Cyan ∪ {s}
14   for all s → t do
15     if (t ∉ Blue ∪ Cyan and t ⋢ Red)
16     then dfsBlue(t)
17   if (s ∈ F) then
18     dfsRed(s)
19   Blue := Blue ∪ {s}
20   Cyan := Cyan \{s}
```

Fig. 5. Nested DFS algorithm with subsumption [13] to compute a subgraph of $ZG^{\mathfrak{a}\preccurlyeq^{LU}}(\mathcal{A})$

(illustrated in Figure 5) gives a restricted way of using subsumption in a nested DFS algorithm for detecting accepting cycles. It exploits the following property: if we know that from a node s there is no reachable accepting cycle, then no node $t \sqsubseteq s$ needs to be explored. The red nodes in the nested DFS algorithm play the role of node s (c.f. Lines 10 and 15 in algorithm). Another optimization occurs in Line 9 - if there is a path from a node t to node s subsuming it, then a cycle can be concluded.

The goal of this paper is to find subsumption graphs of $ZG^{\mathfrak{a}\preccurlyeq^{LU}}(\mathcal{A})$ that are sound and complete for liveness, and to design efficient algorithms to compute them.

3 LIVENESS COMPATIBLE SUBSUMPTIONS

In this section, we are interested in understanding generic conditions for subsumption to be correct for liveness analysis. We start with an example. Consider the TBA $\mathcal{A}_3$ and $ZG^{\mathfrak{a}\preccurlyeq^{LU}}(\mathcal{A}_3)$ illustrated in Figure 6 - which is given by the graph without the two squiggly edges. The zone graph has

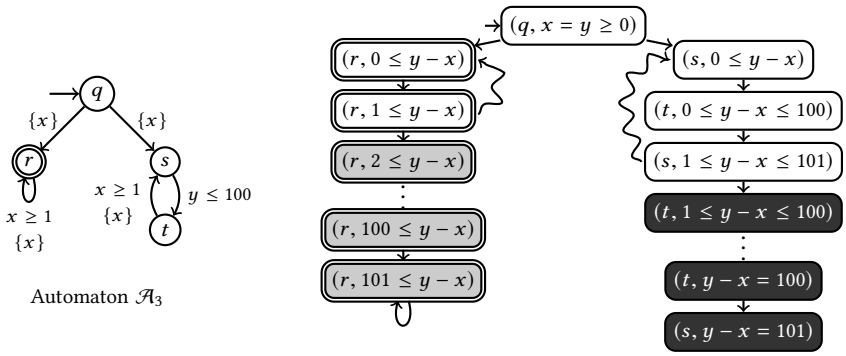

Fig. 6. On the left is a TBA $\mathcal{A}_3$; on the right the graph obtained by removing the two squiggly edges is $ZG^{\mathfrak{a}\preccurlyeq LU}(\mathcal{A}_3)$. Assume that $L = U = 100$ at every state. This can be achieved by adding more transitions on each state (which are not shown for clarity). Squiggly edges show subsumptions. The part of $ZG^{\mathfrak{a}\preccurlyeq LU}(\mathcal{A}_3)$ restricted to white nodes and squiggly edges is the zone graph with subsumption. In this graph there is no accepting cycle consisting only of $\longrightarrow$ edges. Removing the squiggly edge on the node $(r, 1 \leq y-x)$ and adding the grey nodes having state r identifies the accepting cycle.

an accepting cycle on the node $(r, 101 \leq y - x)$. For each of the states r, s and t of the TBA, there are at least 100 nodes in the zone graph. Note that $(r, 1 \leq y - x) \sqsubseteq (r, 0 \leq y - x)$ and $(s, 1 \leq y - x \leq 101) \sqsubseteq (s, 0 \leq y - x)$. If we allowed the luxury to use subsumptions freely, we would get the graph consisting only of the white nodes in the figure and the two squiggly edges denoting subsumption. However, in this graph there is no accepting cycle made uniquely of $\longrightarrow$ edges. There are cycles containing subsumption edges but, as we have seen in Figure 4, it is not sound to take such cycles as witnesses for the existence of an accepting computation in general. Hence, the subsumption on $(r, 1 \leq y - x)$ should not be used to detect accepting cycles. On the other hand, the graph of white nodes with no subsumption edge is not complete for liveness as it has no accepting run. Observe that using subsumption on the node $(s, 1 \leq y - x \leq 101)$ would do no harm, as further exploration would not lead to accepting cycles anyway. This subsumption gives already a significant gain. In fact, the zone graph restricted to the white and grey nodes, along with the subsumption edge on the right is a liveness complete graph according to our definition below. Algorithm in Figure 5 does not detect this possibility and explores the whole graph.

Our goal is to make use of subsumption as much as possible, subject to the restriction that the resulting graph contains an accepting cycle of $\longrightarrow$ edges iff $ZG^{\mathfrak{a}\preccurlyeq LU}(\mathcal{A})$ contains one. Since including subsumption edges as part of a cycle is not sound in general, we will avoid using subsumption edges in cycles that contain accepting states. Therefore, in the graphs that we construct, cycles with accepting states will be actual cycles in $ZG^{\mathfrak{a}\preccurlyeq LU}(\mathcal{A})$ - so every such cycle will give an accepting computation. The challenge is to decide what are the subsumptions that are safe and can be left in the graph. We first make precise the notion of a graph with subsumptions, and then follow up with a condition that makes a zone graph with subsumption complete for liveness.

Definition 3.1 (Subsumption graph). Let G be a graph consisting of a subset of nodes and edges of $ZG^{\mathfrak{a}\preccurlyeq LU}(\mathcal{A})$ together with new edges called subsumption edges. Each node is labeled either *covered* or *uncovered*. Such a graph is called a *subsumption graph* if it satisfies the following conditions:

- **C1** the initial node of $ZG^{\mathfrak{a}\preccurlyeq LU}(\mathcal{A})$ belongs to G and is labeled uncovered,
- **C2** for every uncovered node s, all its successor transitions $s \longrightarrow s'$ occurring in $ZG^{\mathfrak{a}\preccurlyeq LU}(\mathcal{A})$ should be present in G,

C3 for every covered node $t \in G$ there is an uncovered node $s \in G$ such that $t \sqsubseteq s$; moreover there is an explicit *subsumption edge* $t \rightsquigarrow s$ in G,
C4 there is a path of $\rightarrow$ edges from the initial node to every other node.

A path in a subsumption graph is made of both $\rightarrow$ and $\rightsquigarrow$ edges. We write $s_1 \dashrightarrow^* s_2$ to denote that there is a path from s_1 to s_2 in the subsumption graph. We now describe the relation between paths in a zone graph and in a subsumption graph. For the rest of this section, we fix an automaton $\mathcal{A}$ and a subsumption graph G for $\mathcal{A}$.

Lemma 3.2. *For every (finite or infinite) path $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$ in $ZG^{\mathfrak{a} \preccurlyeq LU}(\mathcal{A})$ there is a path $s'_0 \dashrightarrow^* s'_1 \dashrightarrow^* s'_2 \dashrightarrow^* \cdots$ in G such that for each i, $s_i \sqsubseteq s'_i$ and s'_i is uncovered.*

Proof. Proof proceeds by induction. From **C1**, the initial node s_0 is present in G and is uncovered. Suppose we have constructed $s'_0 \dashrightarrow^* s'_1 \dashrightarrow^* \cdots s'_n$. Since $s_n \sqsubseteq s'_n$, there is a transition $s'_n \rightarrow s'_{n+1}$ such that $s_{n+1} \sqsubseteq s'_{n+1}$ in $ZG^{\mathfrak{a} \preccurlyeq LU}(\mathcal{A})$ (c.f. Lemma 2.7). As s'_n is uncovered, this transition is present in G (**C2**). If s'_{n+1} is uncovered in G, we are done. Otherwise, there is an edge $s'_{n+1} \rightsquigarrow s''_{n+1}$ in G with $s'_{n+1} \sqsubseteq s''_{n+1}$ (**C3**). This gives $s'_n \dashrightarrow^* s''_{n+1}$ as required. □

Lemma 3.2 along with condition **C4** says that if there is a path $s_0 \rightarrow^* s$ in $ZG^{\mathfrak{a} \preccurlyeq LU}(\mathcal{A})$, there is a path $s_0 \rightarrow^* s'$ with $s \sqsubseteq s'$ in the subsumption graph G. This shows that subsumption graphs are complete for reachability. However, these conditions are not sufficient for liveness - for a cycle of $\rightarrow$ edges in the zone graph, we may not get a corresponding cycle of $\rightarrow$ edges in the subsumption graph (c.f. Figures 4 and 6). We now give an extra criterion.

Definition 3.3 (Liveness compatible subsumption graph). A subsumption graph G is said to be *liveness compatible* if it additionally satisfies the following condition:

C5 there is no cycle containing both an accepting node and a subsumption edge.

In Figure 6, the zone graph restricted to white nodes and the squiggly edges is not liveness compatible. There is a cycle containing an accepting node $(r, 1 \leq y - x)$ and a subsumption edge from this node. However, removing this subsumption edge and adding the grey nodes makes it liveness compatible. The only remaining subsumption edge is from $(s, 1 \leq y - x \leq 101)$ and it is not part of a cycle containing an accepting node. Intuitively, when we add a subsumption edge $t' \rightsquigarrow s'$, we know that paths in $ZG^{\mathfrak{a} \preccurlyeq LU}(\mathcal{A})$ starting from t' can be simulated from s' in the subsumption graph. But if there is a cycle containing $t' \rightsquigarrow s'$ in the subsumption graph, this would mean that the simulation from s' can bring us back to t'. Hence some accepting runs from t' in $ZG^{\mathfrak{a} \preccurlyeq LU}(\mathcal{A})$ could be lost in the subsumption graph. We show that condition **C5** above makes such a situation impossible.

Theorem 3.4. *$ZG^{\mathfrak{a} \preccurlyeq LU}(\mathcal{A})$ has an infinite accepting path iff a liveness compatible subsumption graph has an infinite accepting path consisting of $\rightarrow$ edges.*

Proof. Let G be a liveness compatible subsumption graph. Since all the $\rightarrow$ edges in G come from the zone graph, a cycle of $\rightarrow$ edges in G implies such a cycle in the zone graph. This shows the direction from right to left. Suppose $ZG^{\mathfrak{a} \preccurlyeq LU}(\mathcal{A})$ has an accepting run ρ: $s_0 \rightarrow s_1 \rightarrow \cdots$. From Lemma 3.2, we have a path ρ' in G of the form: $s'_0 \dashrightarrow^* s'_1 \dashrightarrow^* s'_2 \dashrightarrow^* \cdots$ such that each $s_i \sqsubseteq s'_i$. Since ρ is an accepting run, some accepting node s repeats infinitely often in ρ. Corresponding positions in ρ' contain nodes which subsume s. Since there are finitely many nodes in G, there should be some accepting node s' which occurs infinitely often in ρ'. Therefore there is a cycle containing s' in G. By liveness compatibility criterion **C5** this cycle should be made of only $\rightarrow$ edges. From condition **C4**, there should be a path consisting of $\rightarrow$ edges from the initial node of G

till s'. This gives an infinite path in G made of $\longrightarrow$ edges that visits an accepting node s' infinitely often. □

The above theorem gives a characterization for the use of subsumption to decide liveness. It now remains to design an algorithm that computes such graphs. Before we give an algorithm, we make an observation about the complexity of computing liveness compatible subsumption graphs. Note that these liveness compatible subsumption graphs have more nodes than subsumption graphs used for reachability. In the next section, we show that liveness compatible graphs can be much larger than subsumption graphs, and there is an intrinsic difficulty in inferring liveness from plain subsumption graphs (which are sound and complete for reachability).

4 DECIDING LIVENESS FROM SUBSUMPTION GRAPHS IS PSPACE-COMPLETE

In order to understand the overhead created due to the liveness compatibility condition, we consider a variant of the non-emptiness problem for TBA (Definition 2.2) where a subsumption graph of the automaton is also given as input. A subsumption graph G of a TBA $\mathcal{A}$ is *minimal* if for every two nodes s and t, if $s \sqsubseteq t$ then $s = t$ or s is the initial node of $ZG^{\preccurlyeq_{LU}}(\mathcal{A})$. In other words, G only contains incomparable nodes w.r.t. $\sqsubseteq$, except for the initial node that may not be maximal in G w.r.t. $\sqsubseteq$. We show that the emptiness problem for TBA remains PSPACE-complete even with G as the input. This contrasts with the reachability problem that can be solved in polynomial time from G. We first formalize the problem under consideration. The Büchi non-emptiness problem with subsumption graph (EMPTY-SUB) is defined by:

INPUT: A TBA $\mathcal{A}$ and a minimal subsumption graph G for $\mathcal{A}$
OUTPUT: "Yes" if $\mathcal{A}$ has a Büchi accepting run, "No" otherwise.

THEOREM 4.1. *The problem EMPTY-SUB is* PSPACE*-complete.*

The upper bound follows since determining Büchi non-emptiness from the automaton can be done in PSPACE. For the PSPACE-hardness, we give a reduction from the membership problem for Linear Bounded Automata (LBA). Recall that an LBA is a Turing machine whose tape is restricted to the part on which the input word is written. The membership problem for LBAs asks whether the accepting state is reachable from the initial state and initial tape content w. This problem is known to be PSPACE-complete, even for deterministic LBAs over a binary alphabet. Without loss of generality, we consider a deterministic LBA $\mathcal{B}$, over the binary alphabet $\{0, 1\}$ with initial tape content $w \in \{0, 1\}^N$, and hence tape of size N. We describe a construction of a TBA $\mathcal{A}_{\mathcal{B},w}$ such that:

(1) it has a Büchi accepting run iff $\mathcal{B}$ has an accepting run starting with w on the tape;
(2) $\mathcal{A}_{\mathcal{B},w}$ has a unique minimal subsumption graph G which has size polynomial w.r.t. the size of $\mathcal{B}$ and w;
(3) there is a polynomial time algorithm that given $\mathcal{B}$ and w, constructs $\mathcal{A}_{\mathcal{B},w}$ and G.

This gives a polynomial-time reduction from the membership problem of LBAs to EMPTY-SUB, thus proving Theorem 4.1.

We proceed in two steps: first we describe a TBA $\mathcal{A}'_{\mathcal{B},w}$ which satisfies requirement 1 given above; following this, to satisfy requirement 2, we make some modifications to $\mathcal{A}'_{\mathcal{B},w}$ to get the final automaton $\mathcal{A}_{\mathcal{B},w}$.

4.1 Building TBA $\mathcal{A}'_{\mathcal{B},w}$ that simulates the LBA

The main point is to encode the tape of $\mathcal{B}$. Each cell $i \in [1; N]$ of the tape is represented in $\mathcal{A}'_{\mathcal{B},w}$ by a clock x_i such that $x_i = 2i$ when cell i contains a 0, and $x_i = 2i + 1$ when it contains a 1. To

simplify the construction, we introduce an extra clock y. We let $X = \{x_1, \ldots, x_N, y\}$ denote the set of clocks of $\mathcal{A}'_{\mathcal{B},w}$. A clock valuation v encodes a tape content $w \in \{0,1\}^N$ when $v(y) = 0$ and $v(x_i) = 2i + w_i$ for every $i \in [1; N]$ (where w_i denotes the content of the i^{th} cell). We write $\mathrm{enc}(w)$ for the valuation that encodes w.

For every state q of $\mathcal{B}$, we have a state $q_{i,\cdot,N}$ in $\mathcal{A}'_{\mathcal{B},w}$ where $i \in [1; N]$ encodes the position of the tape head. The transitions of $\mathcal{B}$ are encoded by sequences of transitions in $\mathcal{A}'_{\mathcal{B},w}$. We introduce intermediate states in $\mathcal{A}'_{\mathcal{B},w}$ of the form $q_{i,t,k}$ where q is a state in $\mathcal{B}$, $i \in [1; N]$ is the position of the tape head, t is the transition in $\mathcal{B}$ that is simulated in $\mathcal{A}'_{\mathcal{B},w}$, and $k \in [0; N]$ is an index pointing to a cell that is being processed in our simulation. Let t be a transition $q \xrightarrow{\alpha,\beta,\Delta} q'$ in $\mathcal{B}$ with $\alpha, \beta \in \{0,1\}$ and $\Delta \in \{-1, 0, 1\}$: when $\mathcal{B}$ is at q and the tape head points to a cell with letter α, the LBA overwrites it with β, moves the tape head according to Δ and changes its state to q'. This transition t is simulated by the following sequence in $\mathcal{A}'_{\mathcal{B},w}$:

$$\begin{aligned} q_{i,\cdot,N} \xrightarrow{(y=0)\wedge(x_i=2i+\alpha)} q_{i,t,N} \xrightarrow[\{x_N\}]{x_N=2(N+1)} q_{i,t,N-1} \xrightarrow[\{x_{N-1}\}]{x_{N-1}=2(N+1)} \cdots \\ \cdots q_{i,t,i} \xrightarrow[\{x_i\}]{x_i=2(N+1)+\alpha-\beta} \cdots \\ \cdots q_{i,t,1} \xrightarrow[\{x_1\}]{x_1=2(N+1)} q_{i,t,0} \xrightarrow[\{y\}]{y=2(N+1)} q'_{i+\Delta,\cdot,N} \end{aligned} \tag{1}$$

Except for the names of the states, the only elements in (1) parameterized by t and i are the guard of the first transition $x_i = 2i + \alpha$ and the guard from the middle transition $x_i = 2(N+1) + \alpha - \beta$. In particular, all sequences have the same last transition that checks $y = 2(N+1)$ and that resets y. Clock y ensures that the duration of sequence (1) is exactly $2(N+1)$, and that time does not elapse in states of the form $q_{i,\cdot,N}$ thanks to the first guard $y = 0$.

The first transition checks that cell i contains α by testing if $x_i = 2i + \alpha$. Then, subsequent transitions simulate a sequential access to the tape, from its last cell, encoded by x_n, to its first cell, encoded by x_1. The value of any clock $x_j \neq x_i$ is left unchanged by (1). Indeed, if v is the value of x_j in $q_{i,\cdot,N}$, then $2(N+1) - v$ time units must elapse before the transition $\xrightarrow[\{x_j\}]{x_j=2(N+1)}$ is taken, and v time units elapse after the transition since the total delay on the sequence is $2(N+1)$. Hence $x_j = v$ in $q'_{i+\Delta,\cdot,N}$. The transition $\xrightarrow[\{x_i\}]{x_i=2(N+1)+\alpha-\beta}$ updates the value of the clock x_i. Due to the first transition in the sequence, we know that $x_i = 2i + \alpha$ in $q_{i,\cdot,N}$. Then, $2(N+1) - (2i + \beta)$ time units elapse before the transition is taken, and we delay $2i + \beta$ after this transition to reach a total delay of $2(N+1)$ along the sequence. Hence $x_i = 2i + \beta$ in $q'_{i+\Delta,\cdot,N}$, thereby encoding the new value β of cell i. Observe that every transition on (1), except the first one, is enabled since for every $k \in [1; N]$, $x_k \leq 2N + 1$ when $q_{i,t,k}$ is reached, and $y \leq 2N + 1$ when $q_{i,t,0}$ is reached. Finally, the automaton reaches state $q'_{i+\Delta,\cdot,N}$ hence simulating a move to state q' with tape head on cell $i + \Delta$ in $\mathcal{B}$.

$\mathcal{A}'_{\mathcal{B},w}$ has an initialisation sequence that encodes the initial word w in the clocks:

$$\overline{q}_{1,\cdot,N} \xrightarrow[\{x_{N-1}\}]{x_N=2+(w_N-w_{N-1})} \cdots \overline{q}_{1,\cdot,i} \xrightarrow[\{x_{i-1}\}]{x_i=2+(w_i-w_{i-1})} \overline{q}_{1,\cdot,i-1} \cdots \overline{q}_{1,\cdot,1} \xrightarrow[\{y\}]{x_1=2+w_1} q^0_{1,\cdot,N} \tag{2}$$

where q^0 is the initial state of $\mathcal{B}$ and $\overline{q}_{1,\cdot,k}$ are new intermediate states with $k \in [1; N]$ and $\overline{q}$ is distinct from all states in $\mathcal{B}$. Starting with all clocks equal to 0, the automaton reaches $q^0_{1,\cdot,N}$ with $y = 0$ and $x_i = 2i + w_i$ for all $i \in [1; N]$, hence valuation $\mathrm{enc}(w)$.

A configuration of $\mathcal{B}$ is a triple (q, i, u) where q is a state of $\mathcal{B}$, $i \in [1; N]$ is the position of the tape head, and $u \in \{0,1\}^N$ is the content of the tape. From the construction above, we get that LBAs can be simulated by TBAs:

Lemma 4.2. *A configuration (q, i, u) is reachable in an LBA $\mathcal{B}$ started on an input word w if and only if the configuration $(q_{i,\cdot,N}, \mathrm{enc}(u))$ is reachable in $\mathcal{A}'_{\mathcal{B},w}$ from the initial state $\overline{q}_{1,\cdot,N}$.*

The above lemma talks only about reachability while requirement 1 above talks about a Büchi run. The last step in our construction of $\mathcal{A}'_{\mathcal{B},w}$ is to cater to this. Without loss of generality, we can assume that $\mathcal{B}$ has a unique accepting state q^f with no outgoing transition. We make every state of the form $q^f_{i,\cdot,N}$ with $i \in [1; N]$ as accepting in $\mathcal{A}'_{\mathcal{B},w}$. For every tape head position $i \in [1; n]$, we add an extra edge:

$$q^f_{i,\cdot,N} \xrightarrow[\{x_1,\cdots,x_N,y\}]{} \overline{q}_{1,\cdot,N} \tag{3}$$

that restarts $\mathcal{A}'_{\mathcal{B},w}$ in its initial configuration once an accepting state has been reached. From Lemma 4.2 we get that:

Corollary 4.3. *$\mathcal{B}$ has a finite accepting run on input word w iff $\mathcal{A}'_{\mathcal{B},w}$ has a Büchi accepting run with initial state $\overline{q}_{1,\cdot,N}$.*

4.2 Small Subsumption Graph for $\mathcal{A}'_{\mathcal{B},w}$

A crucial point for our proof of Theorem 4.1 is to get a small minimal subsumption graph G for $\mathcal{A}'_{\mathcal{B},w}$. More precisely the size of G should be polynomial w.r.t. the size of $\mathcal{B}$ and w. To get this, we will add some states and edges to $\mathcal{A}'_{\mathcal{B},w}$.

A clock order is a total order on the set X of clocks of $\mathcal{A}'_{\mathcal{B},w}$. Let i be an integer between 0 and N. We denote $\trianglelefteq_i$ the clock order $x_{i+1} \trianglelefteq_i \cdots \trianglelefteq_i x_N \trianglelefteq_i y \trianglelefteq_i x_1 \trianglelefteq_i \cdots \trianglelefteq_i x_i$. A valuation v satisfies a clock order $\trianglelefteq_i$ if for all clocks $z, z' \in X$ we have $v(z) \leq v(z') \Leftrightarrow z \trianglelefteq_i z'$. A set S of clock valuations satisfies $\trianglelefteq_i$ if all clock valuations $v \in S$ satisfy $\trianglelefteq_i$. Sequences (1) and (2), reset the clocks in the same order: first x_N, then x_{N-1}, …, then x_1, and finally y. The initial clock valuation, where all clocks have value 0, obviously satisfies $\trianglelefteq_N$. Then, by construction of $\mathcal{A}'_{\mathcal{B},w}$, we get that:

Lemma 4.4. *For every configuration $(q_{i,t,k}, v)$ reachable in $\mathcal{A}'_{\mathcal{B},w}$ from the initial state $\overline{q}_{1,\cdot,N}$, the valuation v satisfies $\trianglelefteq_k$.*

All the reachable zones in the (unabstracted) zone graph of $\mathcal{A}'_{\mathcal{B},w}$ are sets of reachable valuations. Hence, from Lemma 4.4, we have that in every reachable node $(q_{i,t,k}, Z)$ in the zone graph of $\mathcal{A}'_{\mathcal{B},w}$, Z satisfies $\trianglelefteq_k$. We now use this observation to modify $\mathcal{A}'_{\mathcal{B},w}$ to get a small minimal subsumption graph. Let $Z_{\trianglelefteq_k} = \{v \in \mathbb{R}^{N+1}_{\geq 0} \mid v \text{ satisfies } \trianglelefteq_k\}$ be the set of valuations that satisfy $\trianglelefteq_k$. Observe that $Z_{\trianglelefteq_k}$ is a zone. The next step of the construction consists in modifying $\mathcal{A}'_{\mathcal{B},w}$ in such a way that its zone graph contains nodes $(q_{i,t,k}, Z_{\trianglelefteq_k})$ for every state $q_{i,t,k}$ in the automaton. From Lemma 4.4, these zones are maximal w.r.t. zone inclusion, and subsume every other reachable zone. This way we will ensure that (1) the minimal subsumption graph is unique, and (2) it does not depend on $\mathcal{A}'_{\mathcal{B},w}$, except for the names of states, and from this we will get an easy bound on its size.

We add a new initial state ι, and sequences from ι producing the zones $Z_{\trianglelefteq_k}$. For all $k, j \in [0; N]$, we introduce intermediate states $\iota_{k,j}$ and transitions:

$$\iota \xrightarrow[\{x_k\}]{} \iota_{k,N} \cdots \xrightarrow[\{x_1\}]{} \iota_{k,N-k+1} \xrightarrow[\{y\}]{} \iota_{k,N-k} \xrightarrow[\{x_N\}]{} \cdots \iota_{k,1} \xrightarrow[\{x_{k+1}\}]{} \iota_{k,0} \tag{4}$$

Starting from ι, every valuation obtained in $\iota_{k,0}$ satisfies $\trianglelefteq_k$. Moreover, the zone that is reachable in $\iota_{k,0}$ is precisely $Z_{\trianglelefteq_k}$. Notice that there is a linear number of sequences (4). We then connect

these sequences to all relevant states of $\mathcal{A}'_{\mathcal{B},w}$ with the effect that every state is reachable with a maximal zone:

$$\iota_{N,0} \longrightarrow q_{i,\cdot,N} \qquad \text{for every } i \in [1;N] \tag{5}$$

$$\iota_{k,0} \longrightarrow q_{i,t,k} \qquad \text{for every } i \in [1;N], t \in T,\ k \in [0;N] \tag{6}$$

$$\iota_{k,0} \longrightarrow \overline{q}_{1,\cdot,k} \qquad \text{for every } k \in [1;N] \tag{7}$$

We are now ready to formally define $\mathcal{A}_{\mathcal{B},w}$.

Definition 4.5. Let $\mathcal{B}$ be a deterministic LBA $(Q, q_0, q_f, \{0,1\}, T, N)$ such that the accepting state q_f has no outgoing transitions. Let $w \in \{0,1\}^N$ be an initial tape content of $\mathcal{B}$ of length $N \geq 0$. We define the TBA $\mathcal{A}_{\mathcal{B},w} = (S, \iota, F, X, \longrightarrow)$ by:

- $S = \{q_{i,\cdot,N} \mid q \in Q,\ i \in [1;N]\} \cup \{q_{i,t,k} \mid q \in Q,\ i \in [1;N],\ t \in T,\ k \in [0;N]\} \cup \{\overline{q}_{1,\cdot,k} \mid k \in [1;N]\} \cup \{\iota\} \cup \{\iota_{i,k} \mid i,k \in [0;N]\}$ is the set of states,
- $\iota \in Q$ is the initial state,
- $F = \{q^f_{i,\cdot,N} \mid i \in [1;N]\}$ is the set of final states,
- $X = \{x_1, \ldots, x_N, y\}$ is the set of clocks,
- $\longrightarrow$ is the transition relation defined by:
  - for every transition $t \in T$ and every tape head position $i \in [1;N]$, there are simulation transitions as in (1);
  - there are initialisation transitions as in sequence (2);
  - for every tape head position $i \in [1;N]$, there is a restart transition (3);
  - for every $k \in [0;N]$, there are reset sequences as (4);
  - and for every $i \in [1;N]$, every $t \in T$, and every $k \in [0;N]$ there are maximal zone transitions as in (5), (6), and (7).

We first state the reduction of the membership problem for Linear Bounded Automata to the emptiness for Timed Büchi Automata.

Lemma 4.6. *A deterministic LBA $\mathcal{B}$ accepts a word w iff $\mathcal{A}_{\mathcal{B},w}$ has a Büchi accepting run.*

This result follows from Corollary 4.3. Observe that on an accepting Büchi run, $\mathcal{A}'_{\mathcal{B},w}$ visits an accepting state infinitely many times. Each time it enters an accepting state it is re-initialised. In $\mathcal{A}_{\mathcal{B},w}$ the first part of the run may be perturbed because of maximal zone generation sequences (4), (5), (6) and (7). Nevertheless, after reaching an accepting state for the first time, the state of, $\mathcal{A}_{\mathcal{B},w}$ is reset and it works the same way as $\mathcal{A}'_{\mathcal{B},w}$.

As a result of adding sequences (4), (5), (6) and (7) that first generate maximal zones, the minimal subsumption graph for $\mathcal{A}_{\mathcal{B},w}$ is small.

Lemma 4.7. *Let $\mathcal{B}$ be an LBA, w be a word over $\{0,1\}$ and let $\mathcal{A}_{\mathcal{B},w}$ be the corresponding LBA. The automaton $\mathcal{A}_{\mathcal{B},w}$ and its minimal subsumption graph have a size polynomial in the size of $\mathcal{B}$ and w.*

Proof. Observe from Definition 4.5 that $\mathcal{A}_{\mathcal{B},w}$ is itself of size polynomial in the size of $\mathcal{B}$ and w. Let G be the minimal subsumption graph for $\mathcal{A}_{\mathcal{B},w}$. We show that it is linear in the size of $\mathcal{A}_{\mathcal{B},w}$. The bound on the size of G follows from the observation that there is at most one node in G for every state in $\mathcal{A}_{\mathcal{B},w}$. This can be seen from Figure 7. The states ι and $\iota_{k,j}$ with $k,j \in [0;N]$ can only be reached once following sequences (4). Now, transitions (5), (6) and (7) generate zones $Z_{\trianglelefteq_N}$ for states $q_{i,\cdot,N}$, and $Z_{\trianglelefteq_k}$ for states $q_{i,t,k}$ and $\overline{q}_{1,\cdot,k}$. Taking one more transition from these nodes by following sequences (1), (2) or (3) yield new nodes which are covered by the maximal zones $Z_{\trianglelefteq_i}$ at the previous level thanks to Lemma 4.4. □

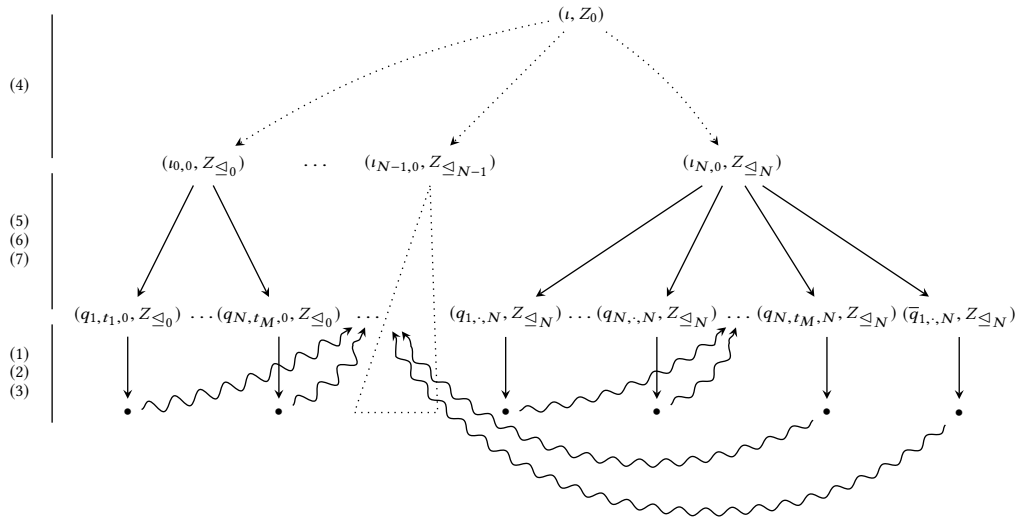

Fig. 7. The minimal subsumption graph for $\mathcal{A}_{\mathcal{B},w}$. The first level depicted as dotted lines represents the sequences of transitions and nodes corresponding to (4). The second level are the nodes generated by transitions (5), (6) and (7). The third level are nodes generated by one transition of (1), (2) and (3). The nodes reached after these transitions, denoted • in the picture, are subsummed by the nodes at the previous level, as shown by the squiggly edges.

4.3 Polynomial-time reduction for EMPTY-SUB

To finish the proof of Theorem 4.1, it remains to show that requirement 3 is satisfied by our construction: namely, the minimal subsumption graph of $\mathcal{A}_{\mathcal{B},w}$ can be computed in polynomial time. Let us recall what we have done till now. Given an LBA $\mathcal{B}$ and an input word w, we have shown how to construct in polynomial time automaton $\mathcal{A}_{\mathcal{B},w}$. Lemma 4.7 shows that the minimal subsumption graph G for $\mathcal{A}_{\mathcal{B},w}$ has polynomial size. To construct G in polynomial time we can simply run a breadth-first search on the unabstracted zone graph $ZG(\mathcal{A}_{\mathcal{B},w})$ as illustrated in Figure 7. Starting from (ι, Z_0) where Z_0 is the zone consisting of valuations in which all clocks are equal, one follows sequences (4). From the structure of $\mathcal{A}_{\mathcal{B},w}$, this computes all nodes $(\iota_{\kappa,0}, Z_{\trianglelefteq k})$. In the next step following transitions (5), (6) and (7), all states of $\mathcal{A}'_{\mathcal{B},w}$ will be visited along with their maximal zones $Z_{\trianglelefteq_i}$. Therefore, at depth $N+2$, the BFS generates exactly one node for each state in $\mathcal{A}_{\mathcal{B},w}$ containing the associated maximal zone. This takes polynomial time since up to depth $N+1$ the automaton consists just of a linear number of sequences as in (5), (6), and (7). Then, due to subsumption, BFS stops at depth $N+3$ after one transition from (1), (2) or (3). The number of nodes at level $N+3$ is bounded by the maximal out degree of the states in $\mathcal{A}_{\mathcal{B},w}$ times the number of nodes in G. This is again of polynomial size w.r.t. $\mathcal{B}$ and w. Hence we get:

Lemma 4.8. *There is a polynomial time algorithm constructing the minimal subsumption graph of $\mathcal{A}_{\mathcal{B},w}$ when given $\mathcal{B}$ and w.*

This completes the polynomial reduction and thus proves Theorem 4.1.

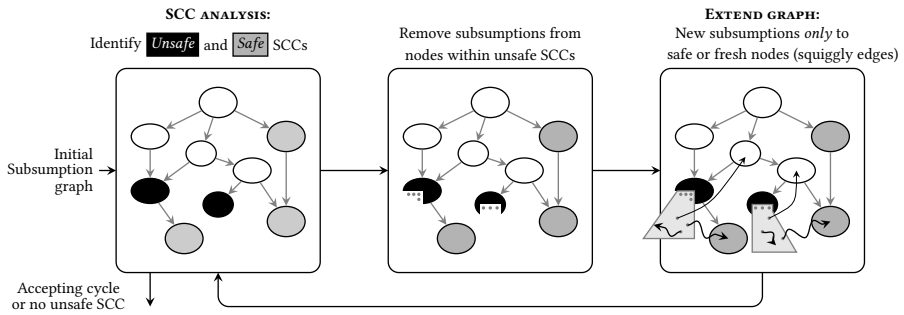

Fig. 8. Illustration of the iterative algorithm to construct liveness compatible subsumption graphs.

5 A NEW ALGORITHM FOR LIVENESS

We now consider the algorithmic problem of computing a liveness compatible subsumption graph for a given automaton. Section 3 says that such a graph allows us to determine if the automaton has a Büchi run. Section 4 indicated that computing such a graph may be much more complex than just computing the subsumption graph using the algorithm in the right of Figure 3. The objective is of course to compute a small graph, as otherwise we could just compute the entire abstract zone graph without subsumption using the algorithm from the left of Figure 3. A better solution is the nested DFS algorithm in Figure 5 - indeed the final graph computed by it is a liveness compatible subsumption graph. In this section, we present a different algorithm, and compare its performance with the nested DFS approach. Our algorithm iterates between a reachability computation and an SCC analysis of the computed graph to find cycles violating condition **C5** in Definition 3.3. Recall that a cycle violates **C5** if it contains both an accepting state and a subsumption edge; we will call such cycles *unsafe* in the rest of this section.

Figure 8 illustrates the idea of the algorithm. It starts with an initial subsumption graph obtained by a reachability analysis. This subsumption graph may potentially contain unsafe cycles. The algorithm proceeds by performing an SCC decomposition of the graph by considering $\rightarrow \cup \rightsquigarrow$ as the edge relation (shown by the left figure, with each bubble denoting an SCC). An SCC is marked Unsafe (black bubble) if it contains an accepting state and a subsumption edge (with both ends in the same SCC). An SCC is marked Safe (grey bubble) if it cannot reach an unsafe SCC. If an accepting cycle (consisting only of $\rightarrow$ edges) is found during this analysis, the algorithm terminates saying that the automaton has a Büchi accepting run. Otherwise, the algorithm proceeds to the next step. The SCC analysis guarantees that the grey part is liveness compatible and exploration-complete, whereas the remaining part is not yet liveness compatible. Subsumption edges from nodes within unsafe SCCs are removed. This opens nodes for exploration (shown in the middle picture as dots in the black bubbles). A subsumption graph construction is started from the grey dots with subsumption restricted to the previously found safe nodes or the fresh nodes which appear in the new exploration (shown by the squiggly edges in the picture on the right). This restriction is imposed in order to avoid falling repeatedly into the same bad cycle: the grey nodes were subsumed by nodes in their inhabiting black SCCs, and hence the successors of grey nodes will also be subsumed by the corresponding successors in the black SCCs. So such subsumptions would stop the new exploration in one step. Even with this restricted subsumption, it is still possible that new unsafe cycles are formed. Therefore, once the new exploration terminates, the obtained subsumption graph is passed on for the SCC analysis. This process is iterated till either the SCC analysis identifies an accepting cycle, or there are no more unsafe SCCs. The latter case gives a

liveness compatible subsumption graph with no accepting cycles and hence the algorithm can terminate saying that the automaton has no Büchi run. The algorithm needs to handle an extra detail: the removal of subsumption edges after the SCC analysis could lead to dangling nodes that are not $\rightarrow$ reachable from the initial node. All such nodes are removed before the next subsumption graph computation (in the picture on the right).

The algorithm can be viewed as an iterative refinement of the initial subsumption graph till a liveness compatible graph is obtained. To achieve the behavior of restricted subsumption, a *level* field is added to each node and subsumption is allowed only between nodes of the same level, or to nodes in the previous levels that do not reach unsafe cycles (nodes in grey SCCs).

We will now give a more detailed description of the algorithm. We will say that a node s is *covered* if it has an outgoing subsumption edge $s \rightsquigarrow s'$, for some s', otherwise s is *uncovered.*

Iterative SCC based algorithm with subsumption:

Phase 0 (Initialize). Let S_{init} and S be the singleton sets containing the initial node which is the initial (state, zone) pair. Set the level of this node to 1. Let $K = 1$.

Phase 1 (Construct level K subsumption graph). Construct a subsumption graph from nodes in S_{init} in the following manner. Set the pool of nodes to be explored to be equal to S_{init}. Every node added in this phase will have level field set to K. Repeatedly, pick a node (q, Z) from the pool (and remove it from the pool). For every edge $(q, Z) \Rightarrow (q', Z')$, check if there is already a node $(q', Z_1) \in S$ for which one of the conditions below holds:

1.1 $\mathfrak{a}_{\preccurlyeq_{LU}}(Z') = \mathfrak{a}_{\preccurlyeq_{LU}}(Z_1)$, or

1.2 $\mathfrak{a}_{\preccurlyeq_{LU}}(Z') \subset \mathfrak{a}_{\preccurlyeq_{LU}}(Z_1)$, node (q', Z_1) is uncovered and has the level K or ∞.

If there is no such node, then add the node (q', Z') to the pool as well as to S, and add the edge $(q, Z) \rightarrow (q', Z')$ to S. Moreover, for every uncovered (and not initial) node (q', Z_2) of level K with $\mathfrak{a}_{\preccurlyeq_{LU}}(Z_2) \subset \mathfrak{a}_{\preccurlyeq_{LU}}(Z')$: add $(q', Z_2) \rightsquigarrow (q', Z')$ and remove all other edges from (q', Z_2).

If there is (q', Z_1) satisfying the condition 1.1, add the edge $(q, Z) \rightarrow (q', Z_1)$ to S. If condition 1.2 is satisfied, choose one such node (q', Z_1), with preference given to level ∞ nodes; then add the node (q', Z') and the edges $(q, Z) \rightarrow (q', Z')$, $(q', Z') \rightsquigarrow (q', Z_1)$ to S.

By the end of this phase, graph S is extended with some nodes of level K.

Phase 2 (Check for good and bad cycles). Consider the subgraph G_K of S induced by nodes of level $\leq K$, and containing all the $\rightarrow$ and $\rightsquigarrow$ edges between these nodes. Decompose G_K into maximal SCCs by considering both $\rightarrow$ and $\rightsquigarrow$ as the same kind of edges. We single out two types of maximal SCCs:

- *accepting:* when it contains an accepting state and no subsumption edges with both ends in the SCC;
- *unsafe:* when it contains an accepting state and a subsumption edge with both ends of the edge in the SCC.

If there is an accepting SCC in G_K then stop, and return *non-empty*. Otherwise, identify nodes in G_K which cannot reach an unsafe SCC. Change the level of all such nodes to ∞.

Phase 3 (Remove potentially unsafe subsumptions). Let S'_K be the set of nodes which are subsumed (that is the only edge out of them is a subsumption edge) that still have level K. Remove subsumption edges with source in S'_K. Remove, from S and from S'_K, all nodes that are not $\rightarrow$ reachable from the initial node of the graph (the node created in Phase 0). Set S_{init} to S'_K, and set level $K + 1$ to all nodes in S_{init}. Set $K := K + 1$.

Repeat or stop If S_{init} is non-empty, restart from Phase 1; otherwise return *empty*, and stop.

Before we give the invariants and prove correctness of this algorithm, we illustrate this algorithm on an example.

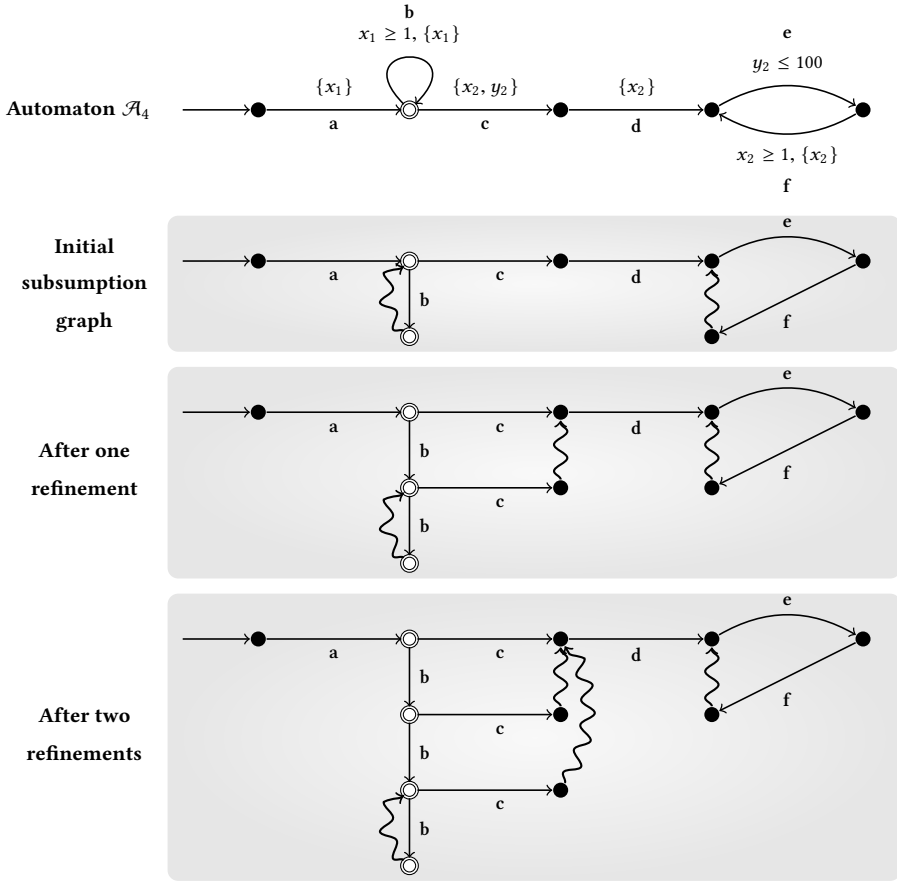

Fig. 9. The top figure shows an automaton with clocks $\{x_1, y_1, x_2, y_2\}$. Clock y_1 does not appear in the transitions for clarity. We also assume that $L = U = 100$ for each clock. The white state is an accepting state. The figures below describe subsumption graphs obtained after two refinements of the initial subsumption graph. We do not show the exact zones. The squiggly edges illustrate subsumptions.

Example 5.1. Consider the automaton $\mathcal{A}_4$ shown in Figure 9. We assume that it has 4 clocks $\{x_1, y_1, x_2, y_2\}$. We can also assume that the bounds L and U are 100 for each clock. The additional guards that can achieve these bounds have not been illustrated. The white state is an accepting state. The transitions a and b are similar to the left hand side of the automaton $\mathcal{A}_3$ (states q, r) in Figure 6 for clocks x_1 and y_1. Repeated application of the loop b gives zones $(y_1 - x_1 \geq 0)$, $(y_1 - x_1 \geq 1)$ and so on till $(y_1 - x_1 \geq 101)$. In all these zones, we will have $x_2 = y_2 = y_1$. The transitions c, d, e and f behave like the right hand side of $\mathcal{A}_3$ (states q, s, t) for clocks x_2 and y_2. Repeated application of the loop ef generates zones $1 \leq y_2 - x_2 \leq 101$, $2 \leq y_2 - x_2 \leq 101$ and so on. When this computation happens, the value of $y_1 - x_1$ remains unchanged. Moreover, due to the resets in c, we will have that both y_1 and x_1 are bigger than y_2.

In the discussion below we denote n_σ the node reached after the sequence of transitions σ. If subsumptions are not used, then zones in nodes $n_{ab^icd(ef)^j}$ reached after sequence $ab^icd(ef)^j$ will all be different for each $i \in \{1, \ldots, 100\}, j \in \{1, \ldots, 100\}$: the constraint between x_1 and y_1 distinguishes zones reached by different b^i and the constraint between x_2 and y_2 distinguishes

the zones reached from different $(ef)^j$. Termination occurs after loops b^i and $(ef)^j$ take at least 100 steps. Therefore the number of zones is bigger than 100×100. If we were allowed to use subsumptions freely, we would not need to explore each loop more than once. This is shown by the illustration of the initial subsumption graph in Figure 9. This initial subsumption graph is sound and complete for reachability. However, this graph is not liveness compatible since there is a cycle containing a subsumption edge and accepting nodes n_a and n_{ab}. Of course, if we had a mechanism to check whether iterating edge b indeed corresponds to an infinite run of $\mathcal{A}_4$, we could stop. This kind of "loop acceleration" is not our focus here.[1] The purpose of this example is to illustrate the gains of using the iterative algorithm where a restricted amount of subsumption is allowed.

Let us get back to the initial subsumption graph. The SCC analysis on this graph will reveal that the nodes $\{n_{ac}, n_{acd}, n_{acde}, n_{acdef}\}$ are all safe, and hence will have level ∞. This is because none of these nodes can reach the unsafe cycle formed by n_a and n_{ab}. For the next iteration, the subsumption edge $n_{ab} \rightsquigarrow n_b$ in this unsafe cycle will be removed and an exploration started from the opened state n_{ab}. In this new iteration, the already inferred safe nodes can be used for subsumption. Therefore, the node n_{abc} will be subsumed by n_{ac}. The node n_{abb} will be subsumed by n_{ab} as it is in the same level. This will now form the unsafe cycle $n_{ab} \rightarrow n_{abb} \rightsquigarrow n_{ab}$, and hence n_{abb} will be opened again for exploration. This process continues till b is iterated 101 times when an equality edge $n_{ab^{101}} \rightarrow n_{ab^{101}}$ would be obtained (as in Figure 6), which means we will have $K = 101$ in the end. In each iteration j, we get 2 new nodes n_{ab^jb} and n_{ab^jc}. The ef loop is never reached in the new iterations.

This example points out an advantage of using the iterative method: subsumptions can be used to cut out "non-accepting" parts of the graph. These are the parts in the graph from which a witness for Büchi non-emptiness cannot be obtained. The nested DFS algorithm adds a subsumption edge $x \rightsquigarrow y$ only if the currently explored part of the graph proves that there is no witness from y. In our iterative method, we are less restrictive about this. We allow to use subsumptions $x \rightsquigarrow y$ even if y has not been completely explored yet, and subsequently remove unsafe subsumptions in an iterative manner. This mechanism could potentially give rise to more nodes y that end up being safe for subsumption. This is exemplified by the behaviour of the algorithm on the automaton $\mathcal{A}_4$. Indeed for this example, the nested DFS algorithm computes the entire zone graph without subsumption: the node n_{acd} becomes ready for subsumption (that is, its colour is red) only after node n_a is completely explored (which is when red DFS is started from n_a). Hence the subsumptions on the right which cut out the ef loops cannot be made by the nested DFS.

A more detailed comparison of the iterative algorithm and the nested DFS approach is given in the Section 6.

We will now prove correctness of the iterative algorithm.

Theorem 5.2. *A strongly non-Zeno TBA $\mathcal{A}$ has a run satisfying the Büchi condition iff the iterative SCC based algorithm with subsumption returns non-empty.*

Proof. We will show that the invariants listed below hold after Phase 3 of the K^{th} iteration. Invariants I1, I3, I5, I7 imply that the graph S computed by the iterative algorithm is liveness compatible if S_{init} is empty. The conclusion then follows from Theorem 3.4 saying that in this case there is no Büchi cycle in the zone graph, and Theorem 2.6 saying that then the automaton does not have a run satisfying the Büchi condition.

I1 the set of nodes in S is a subset of nodes in $ZG^{\mathfrak{a} \preccurlyeq LU}(\mathcal{A})$;

I2 each node in S has a level between 1 and K, or level ∞; all nodes in S_{init} have level K,

[1] Note that the hardness result of Section 4 is independent of the underlying algorithm. Therefore no acceleration method can be both complete and efficient.

I3 for every uncovered node in $S \setminus S_{init}$ all its successors are present in S;
I4 every $\rightarrow$ or $\rightsquigarrow$ edge out of a level ∞ node leads to a level ∞ node.
I5 There is no cycle containing both an accepting node and a subsumption edge.
I6 Every covered node has level ∞.
I7 There is a path of $\rightarrow$ edges from the initial node to every node.

We will prove the above invariants by induction on K. Let us denote by S^i and S^i_{init} the sets S and S_{init} obtained after the execution of Phase 3 of the algorithm in the i^{th} iteration. Let S^0 and S^0_{init} be the respective sets before the first execution of Phase 1.

From Phase 0 of the algorithm, the sets S^0_{init} and S^0 contain only the initial node of $ZG^{\mathfrak{a}\preccurlyeq LU}(\mathcal{A})$. This node is uncovered and its level is 1. All the invariants mentioned above are clearly true.

Assume that the sets S^{K-1} and S^{K-1}_{init} satisfy the above invariants. If S^{K-1}_{init} is empty, the above invariants ensure that conditions **C1-C5** for liveness compatibility are satisfied. Hence the graph S^{K-1} is sound and complete for liveness. We will now consider the case when S^{K-1}_{init} is non-empty. In this case, the K^{th} execution of Phase 1 is started.

Invariants after Phase 1: From the algorithm, we can infer that all new nodes of the K^{th} iteration are added only during Phase 1. The remaining two phases just change levels of nodes and remove some nodes and subsumption edges. Since the added nodes are obtained during a subsumption graph computation starting from S^{K-1}_{init}, which is contained in $ZG^{\mathfrak{a}\preccurlyeq LU}(\mathcal{A})$, the new nodes added in S^K will belong to the zone graph $ZG^{\mathfrak{a}\preccurlyeq LU}(\mathcal{A})$. This gives us invariant I1. Invariant I2 follows from the fact that we add only nodes of level K. Invariant I3 is due to the fact that the algorithm does exhaustive exploration.

During this phase we have also an additional invariant

J1 There are no edges from level ∞ nodes in S^{K-1} to nodes in $S^K \setminus S^{K-1}$.

At the beginning of Phase 1 we have edges from S^{K-1} nodes to S^K, namely those to S^K_{init}. These edges go from nodes of level $K-1$. Note that new $\rightarrow$ edges start in level K and end in level $\leq K$ (the level can be smaller because of case 1.1); all new $\rightsquigarrow$ edges have level K nodes as a source.

Invariants after Phase 2: At the end of Phase 2 of K^{th} iteration, some nodes in S^K are marked ∞. This does not influence invariants I1-I3. Invariant I4 follows from invariant J1, and the code of Phase 2 since if a node from S^K has its level changed to ∞ then all its successors also have its level changed to ∞, or already have level ∞. Invariant I5 holds because the algorithm did not stop, so there is no accepting SCC in the part of the graph consisting of nodes of level $\leq K$. By invariant I4, such a cycle should have been contained completely in nodes of level ∞. But this is impossible by J1, and I5 from the previous phase.

Invariants after Phase 3: At Phase 3 we consider the set S'_K of nodes of level K with outgoing covering edges: we remove those edges. This way we satisfy I6. Invariant I3 still holds because all new uncovered nodes are put in S_{init}. We also remove from S^K and S'_K all nodes that are not $\rightarrow$ reachable from the initial node of the graph. This does not falsify invariants I1-I6, and makes I7 true. Finally, K is increased and the levels of nodes in S'_K are increased too in order to reestablish invariant I2.

Termination: We have now proved the invariants. Since from invariant I1, the obtained graph is always a subset of $ZG^{\mathfrak{a}\preccurlyeq LU}(\mathcal{A})$, the algorithm terminates after some finite number of iterations as each level is non-empty. The number of iterations is bounded by the number of nodes in $ZG^{\mathfrak{a}\preccurlyeq LU}(\mathcal{A})$.

This proves that after some number n of iterations, we would have S^n_{init} to be empty, and the corresponding graph S^n would be liveness compatible. If the algorithm has not stopped in Phase 2, then the invariant I4 says that there is no accepting cycle made of $\rightarrow$ edges in this graph; so it is correct to report *empty*. □

The algorithm from this section starts from the zone graph with subsumption and iteratively expands it till it becomes liveness compatible. From Section 4 we know that there exist cases when this process is long because of the conditional complexity bounds. But one can hope that on many examples the algorithm terminates after a couple of iterations. The next section presents an evaluation of the performance of this algorithm.

6 EXPERIMENTS

We first present a comparison of three algorithms. The nested depth-first search algorithm with subsumption presented in [13] (denoted "Nested DFS" hereafter), the iterative algorithm in [7] (refered to as "Iterative level-SCC" in the sequel) and our algorithm described in page 17 (named "Iterative full-SCC" from now on). The Iterative level-SCC algorithm [7] is a variant of the Iterative full-SCC algorithm (page 17) where the SCC decomposition of the graph in Phase 2 is done only on current level K instead of the whole subsumption graph. This guarantees that each node is visited at most once during Phase 2. In order to ensure correctness of the algorithm, every SCC within level K that can reach a node with level less than K is considered violating. This prevents the construction of bad cycles across levels that would not be detected in Phase 2. Finally, accepting runs that span accross levels would not be detected in Phase 2 since SCC decomposition is only performed on current level K. A last SCC decomposition is thus required upon termination of the main loop (i.e. when S_{init} becomes empty) in order to detect accepting runs. We refer the reader to [7] for further details.

All three algorithms have been implemented in our tool TChecker. Our implementation uses a slightly different representation of the subsumption graph in order to avoid storing covered nodes and as a result we obtain smaller subsumption graphs. More precisely, when we have an edge $(q, Z) \rightarrow (q', Z')$ and there exists a node (q', Z_1) such that $\mathfrak{a}_{\preccurlyeq_{LU}}(Z') \subset \mathfrak{a}_{\preccurlyeq_{LU}}(Z_1)$, then our algorithm from p. 17 adds an edge $(q', Z') \rightsquigarrow (q', Z_1)$. Instead, our implementation removes the node (q', Z') and adds an edge $(q, Z) \rightsquigarrow (q', Z_1)$. When the covering edge $(q, Z) \rightsquigarrow (q', Z_1)$ is deemed inconsistent, the node (q', Z') as well as the edge $(q, Z) \rightarrow (q', Z')$ are added back to the subsumption graph.

We have conducted experiments on the classical benchmarks for Timed Automata, that we describe below. The first six examples have been proposed in [13]. The others are meant to exhibit strengths and weaknesses of the three algorithms. Our testing scheme is as follows: we verify properties given by Büchi automata on these standard models. To do this, we take the product of the model with a property automaton, and check for Büchi non-emptiness on this product.

The results are shown in Table 1. The first column lists the models and properties that were considered (see description below). The black dots • indicate models with an accepting run. Comparison in these cases is difficult since performance depends very much on the order of exploration. Other models do not have accepting runs. As a result, the algorithms have to compute an invariant to prove non-existence of an accepting run in these models. We report the number of visited nodes ("Visited"), the number of nodes in the final invariants ("Size"), the running time ("sec.") as well as the maximum level ("K") for iterative algorithms. The last two columns report the size of subsumption graphs (Definition 3.1) produced by our implementation of UPPAAL's reachability algorithm, and the time needed to compute them. The size of the subsumption graph gives a lower bound on the size of liveness compatible subsumption graphs computed by Iterative algorithms. Both Iterative algorithms and the reachability algorithm explore the state-space of the automata using an optimized search order, called *topological search* proposed in [11]

The first six models and properties are taken from [13]. Fischer's protocol is a mutual-exclusion protocol based on real-time constraints. Let c denote the number of processes in the critical section. We checked properties (fi1) $\mathrm{G}(c \leq 1)$ mutual exclusion; (fi2) $\mathrm{GF}(c = 0) \wedge \mathrm{GF}(c = 1)$ non-blocking

from [14]; and (fi3) $\mathsf{G}(req_1 \implies \mathsf{F}cs_1)$ every request of process 1 is eventually satisfied. FDDI is a token ring protocol where the communication can be synchronous or asynchronous. We checked property (FD1) $\mathsf{FG}(async_1)$ process 1 eventually communicates asynchronously. Finally, CSMA/CD is a protocol to detect and solve message collisions that is used for communications over Ethernet networks. We checked two properties (CC1) $\mathsf{G}(collision \implies \mathsf{F}active)$ after a collision, the network becomes active again; and (CC2) $\mathsf{FG}(collision \implies \mathsf{G}\neg sent)$ after some collisions, no message can ever be sent. We consider instances of the three models with 7 processes and standard parameter values from the literature.

The results show that our Iterative full-SCC algorithm computes liveness invariants that are significantly smaller that those computed by the Nested DFS algorithm and the Iterative level-SCC algorithm. Indeed, for all 6 examples, the reachability invariants are liveness compatible as shown by the comparison to the reachability algorithm. It also visits less nodes on the first 5 examples. Our algorithm stops immediately after Phase 1 for (fi1) and (cc2) that have no reachable accepting state. Models (fi2), (fd1) and (cc1) have reachable accepting states, but no bad SCC. As a result, our algorithm stops after Phase 2 and skips Phase 3. Finally, (fi3) has an accepting run that is found during Phase 2. Examples (cc2) and (cc3) have accepting runs that can be detected early by the nested DFS algorithm. Both Iterative algorithms first entirely compute a subsumption graph (Phase 1) before detecting the accepting runs (in Phase 2). As a result, they visit and store more nodes. This is expected since these algorithms are specialized to compute small subsumption graphs in the absence of accepting runs.

The last five models and properties have been tailored to exhibit strengths and weaknesses of each algorithm. To enable this, we consider timed properties since adding new clocks yields complex zones and makes covering harder. These examples are built from the CSMA/CD model (cc) and the Fischer model (fi) described above. Property (cc3) is depicted in Figure 10 (right). The automaton checks that station 1 tries to transmit fast enough, and that it often achieves successful transmissions. Property (cc4) is a variation of (cc3) where cycles can be iterated only a bounded number of times. This is achieved by adding a new clock t_4 that is never reset, and an invariant $t_4 \leq M$ to the accepting state (for some constant M). Property (cc5) checks that if collisions are infrequent and station 1 tries to send infinitely often, then it effectively sends messages infinitely often. Property (fi4) expresses that if process 1 can infinitely often access the critical section for M time units, then it enters the critical section infinitely often. Finally, property (fi5) checks that process 1 requests access to the critical section frequently, but is only granted access in a certain time window. As for (cc4), the cycles in (fi5) can be iterated only a bounded number of times.

Property (cc4) with bounded number of iterations of cycles is difficult for Iterative algorithms. This example generates many bad SCCs and hence many refinements, as seen in column K. At level 1, Iterative algorithms unfold the cycle once. This leads to a bad SCC that is refined at the next iteration. The next iteration unfolds the cycle for the second time, building a bad SCC again. This goes on until the cycle has been unfolded enough times to detect that it cannot be iterated anymore. Nested DFS algorithm outperforms both Iterative algorithms on this example. We observe however that full-SCC decomposition has an edge over level-SCC decomposition in the number of stored nodes, number of refinements and the running time, at the expense of visiting significantly more nodes. Property (cc5) exhibits the same situation but with a different outcome. Indeed, the full-SCC decomposition turns out to be very effective. It generates many level-∞ nodes that can be used for covering. The liveness compatible subsumption graph produced by the Iterative full-SCC algorithm is several orders of magnitude smaller that those generated by both the nested DFS algorithm and the Iterative level-SCC algorithm. Notice also that the Iterative full-SCC algorithm performed only 1 refinement (i.e. $K = 2$) instead of 59 for the Iterative level-SCC algorithm. Both Iterative algorithms outperfom Nested DFS algorithm on (fi4). In this case, the subsumption graph obtained

Model		Nested DFS [13]			Iterative level-SCC [7]				Iterative full-SCC (p. 17)				Reachability	
		Visited	Size	sec.	Visited	Size	K	sec.	Visited	Size	K	sec.	Size	sec.
fi1		26651	26651	0.51	7737	7737	1	0.23	7737	7737	1	0.23	7737	0.25
fi2		205051	132808	2.87	114714	38238	1	1.34	114714	38238	1	1.33	38238	1.27
fi3	•	45749	26679	0.51	29190	20768	1	0.67	29190	20768	1	0.66	20768	0.67
fd1		21160	18246	0.33	2285	705	1	0.03	2285	705	1	0.02	705	0.08
cc1		41386	26878	0.52	50049	16683	1	0.41	50049	16683	1	0.41	16683	0.34
cc2	•	57	56	0.00	21640	14397	1	0.71	21640	14397	1	0.72	14397	0.55
cc3	•	16185	16184	0.09	1598686	970387	2	76.00	1768310	970387	2	75.94	205656	10.94
cc4		44873	35787	1.33	745259	150078	282	170.76	12228757	85894	157	85.23	365	0.01
cc5		240011	237458	3.21	821845	201424	60	202.56	7109	1562	2	0.04	772	0.01
fi4		466572	382936	9.06	232281	77427	1	3.39	232281	77427	1	3.40	77427	3.55
fi5		48299	24979	0.72	126932	29686	17	1.26	310918	11769	12	0.88	704	0.01

Table 1. Comparison of the size of liveness invariants, number of visited nodes, number of levels (K) and running time for three algorithms: nested DFS algorithm with subsumption [13], Iterative algorithm with level-by-level SCC decomposition [7] and Iterative algorithm with SCC decomposition of the full graph (p. 17). The last two columns show the size of the reachability subsumption graph and the time required to compute it. Iterative algorithms and reachability algorithms run a topological search [11]. The first 6 examples are from [13]. Black dots • mark models that have an accepting run. The tests have been run on a MacBook Pro with an Intel Core i5 2.4 GHz processeor and 16Gb of memory.

at level 1 is liveness compatible. In contrast, Nested DFS algorithm can only use for covering the nodes that have been visited during the red search (see Figure 5). We thus suspect that the nodes that are marked red are mostly "small" nodes, hence limiting the use of covering. Finally, on the last example (fi5), the Iterative full-SCC algorithm yields a smaller liveness compatible subsumption graph than the other two algorithms. Applying the SCC decomposition on the full graph allows to close more SCCs. As a result, the algorithm produces more level-∞ nodes that can later be used for covering.

Extending the benefits of full-SCC decomposition. Let us examine more closely the adavantages of full-SCC decomposition. This discussion will lead to one more optimization that further improves the algorithm.

As we have seen from Table 1, full-SCC decomposition yields smaller subsumption graphs than level-SCC decomposition (when the graph has more than one level), at the cost of visiting more nodes. For instance, the Iterative level-SCC algorithm visits 745259 nodes on (cc4) whereas the Iterative full-SCC algorithm visits 12228757 nodes. Yet, the latter algorithm is twice as fast as the former. This can be explained in two ways. First, the Iterative full-SCC algorithm stores less nodes, so searching for a covering node is faster. Second, and more importantly, the exploration of the zone graph in Phase 1 is much more expensive than all other explorations in subsequent phases. Indeed, computing an edge $(q, Z) \rightarrow (q', Z')$ in Phase 1 costs $O(n^3)$ where n is the number of clocks. This is the time required to compute the zone Z'. Once the edge is generated and stored in memory, the cost of traversing the edge is $O(1)$. As a result, the extra visits in Phase 2 and 3 are relatively cheap. Thus, bigger number of visits to nodes in Phase 2 by the Iterative full-SCC algorithm has a very limited impact on the running time. The difference in running times among the two Iterative algorithms on (cc4) comes from the lower number of calls in Phase 1: 157 for the full-SCC algorithm vs. 282 for the level-SCC algorithm. The advantage of full-SCC decomposition over level-SCC decomposition is that the former produces more level-∞ nodes, as it is able to close SCCs more often.

We now discuss how we can improve covering and produce even smaller liveness compatible subsumption graphs. First observe that nodes are marked ∞ during Phase 2 (see p. 17). These nodes can be later used for covering during Phase 1 to discard new nodes that are constructed. This means that when a node n gets level ∞ in Phase 2, it can only be used to cover nodes that will be visited during subsequent runs of Phase 1. Our algorithm will never try to cover by n, the nodes that have been created before n got level ∞.

We have thus implemented a variant of the Iterative full-SCC algorithm that tries to cover existing nodes when a node gets level ∞. Notice that to fullfil Definition 3.1, when a node n is covered by a level-∞ node n', we need to remove all outgoing edges from n, then add an edge $n \rightsquigarrow n'$. As a result, the successor nodes of n in the subsumption graph may become unreachable. Hence, this extra optimization is applied in the middle of Phase 3. The resulting modified phase is:

(Phase3') Let S'_K be the set of nodes which are subsumed (that is the only edge out of them is a subsumption edge) that still have level K. Remove subsumption edges with source in S'_K.
(Extended covering) for every node (q, Z) that has been marked level ∞ during Phase 2, and for every node (q, Z') such that $\mathfrak{a}_{\preccurlyeq_{LU}}(Z') \subset \mathfrak{a}_{\preccurlyeq_{LU}}(Z)$, remove all outgoing edges from (q, Z'), and add an edge $(q, Z') \rightsquigarrow (q, Z)$.
Remove, from S and S'_K, all nodes that are not $\rightarrow$ reachable from the initial node of the graph (the node created in Phase 0). Set S_{init} to S'_K, and set level $K + 1$ to all nodes in S_{init}. Set $K := K + 1$.

This optimization is correct since the graph built by the algorithm is a subsumption graph (see Definition 3.1) and thanks to invariant (I4). Indeed we remove outgoing edges of newly covered

nodes. Moreover the covering edges added by extended covering cannot create bad cycles in the graph since they go to level-∞ nodes. Notice that this optimization requires at most 2 extra traversals of the graph: one to detect covered nodes, and an other one to detect reachable nodes[2].

Table 2 shows the effect of extended covering for the Iterative full-SCC algorithm on the same set of examples as before. We compare three algorithms: Iterative level-SCC algorithm [7], Iterative full-SCC algorithm (p. 17) and "Iterative full-SCC + Ext. covering" that adds extended covering to the Iterative full-SCC algorithm. The last two columns in the table give the size of subsumption graphs generated by our implementation of UPPAAL's algorithm and the time required to compute it. Extended covering can also be implemented on top of the Iterative level-SCC algorithm. We do not report results here because it has very limited impact. The only example where it provides a significant gain is (cc3) where it generates the same subsumption graph as the full-SCC algorithm. We observe a minor gain (less that 1%) on examples (cc4), (cc5) and (fi5), and no gain at all on other examples. This is because applying SCC decomposition level by level does not generate many level-∞ nodes.

Extended covering significantly improves Iterative full-SCC algorithm. Since this algorithm generates many level-∞ nodes, we observe a significant gain on most examples. The size of the liveness compatible subsumption graphs is close to the size of the subsumption graphs generated by UPPAAL's reachability algorithm. Moreover, the costs of detecting covered nodes and unreachable nodes have no significant impact on running time. As explained above the state-space explorations are inexpensive compared to Phase 1. Moreover, the small number of stored nodes eases the detection of covered nodes. As a result adding extended covering has a positive impact on both stored nodes and running time.

On the importance of Büchi verification for Timed Automata. Finally, the case of CSMA/CD gives an interesting motivation for testing Büchi properties. Indeed property (cc2) holds for the CSMA/CD model. As a result, the model is not correct since communications should be enabled after a collision. It turns out that a transition is missing in the widely used model [16, 18]. In consequence, in this model there is no execution with infinitely many collisions and completed transmissions. Even more, once some process enters into a collision, no process can send a message afterwards. The model can be fixed by allowing the *busy* action in state *RETRY* as shown in Figure 10.

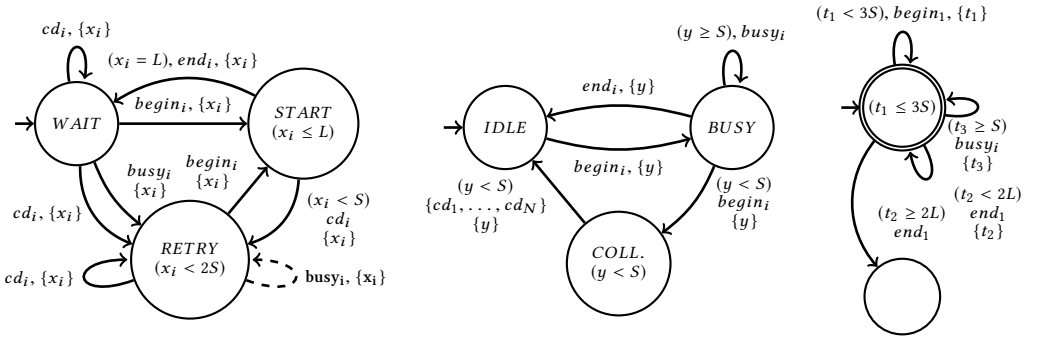

Fig. 10. Model of CSMA/CD: station (left) and bus (middle); property (CC3) (right). The dashed edge on state "RETRY" of the station should be added to the model to allow proper communication after a collision.

[2]Detection of reachable nodes can be achieved during Phase 2 when this optimization is not used. When the optimization is used, it has to be done after applying extended covering. Hence an extra traversal of the graph is needed.

Model		Iterative level-SCC [7]				Iterative full-SCC (p. 17)				Iterative full-SCC + Ext. covering				Reachability	
		Visited	Size	K	sec.	Visited	Size	K	sec.	Visited	Size	K	sec.	Size	sec.
fi1		7737	7737	1	0.23	7737	7737	1	0.23	7737	7737	1	0.23	7737	0.25
fi2		114714	38238	1	1.34	114714	38238	1	1.33	237890	38238	1	1.36	38238	1.27
fi3	•	29190	20768	1	0.67	29190	20768	1	0.66	29190	20768	1	0.66	20768	0.67
fd1		2285	705	1	0.03	2285	705	1	0.02	4482	705	1	0.02	705	0.08
cc1		50049	16683	1	0.41	50049	16683	1	0.41	100593	16683	1	0.42	16683	0.34
cc2	•	21640	14397	1	0.71	21640	14397	1	0.72	21640	14397	1	0.71	14397	0.55
cc3	•	1598686	970387	2	76.00	1768310	970387	2	75.94	2407176	875444	2	52.30	205656	10.94
cc4		745259	150078	282	170.76	12228757	85894	157	85.23	23208607	608	157	64.55	365	0.01
cc5		821845	201424	60	202.56	7109	1562	2	0.04	13181	1204	2	0.05	772	0.01
fi4		232281	77427	1	3.39	232281	77427	1	3.40	475411	77427	1	3.45	77427	3.55
fi5		126932	29686	17	1.26	310918	11769	12	0.88	457625	2639	11	0.88	704	0.01

Table 2. Comparison of the size of liveness invariants, number of visited nodes, number of levels (K) and running time for three variants of the iterative algorithm: Iterative algorithm with level-by-level SCC decomposition [7], Iterative algorithm with SCC decomposition of the full graph (p. 17) and Iterative algorithm with full-SCC decomposition and extended covering. The last two columns show the size of the reachability subsumption graph and the time required to compute it. All four algorithms run a topological search [11]. The first 6 examples are from [13]. Black dots • mark models that have an accepting run. The tests have been run on a MacBook Pro with an Intel Core i5 2.4 GHz processeor and 16Gb of memory.

This example confirms once more that timed models are compact descriptions of complicated behaviors due to both parallelism and interaction between clocks. Büchi properties can be extremely useful in making sure that a model works as intended: the missing behaviors can be detected by checking if there is a run where every collision is followed by a transmission. Adding the missing transition enables interesting behaviors where the stations have collisions and then they restart sending messages.

7 CONCLUSION

As we show in this paper, the liveness problem for timed automata is substantially more difficult algorithmically than the reachability problem.

The abstract zone graph with subsumption is a standard invariant for reachability properties. We have given examples where even knowing this graph upfront keeps the liveness problem PSPACE-hard. This partly explains why very little progress has been made in verification of liveness properties, since the hope was that the liveness problem can be solved almost as efficiently as reachability. In the light of the above hardness result, this is impossible modulo complexity theoretic assumptions.

We have defined a notion of an invariant for liveness properties: a graph proving that the property does not hold. We have also proposed a high-level algorithm for constructing such an invariant. Finally, we have reported on some experiments with a preliminary implementation of this algorithm. Further work will be required to understand the relation between sizes of liveness and safety invariants, as well as to develop better algorithms for constructing liveness invariants.

While some results of these experiments are very interesting, the others show that our implementation is far from optimal. Finding better algorithms for constructing liveness invariants is certainly the most important direction for further work.

ACKNOWLEDGMENTS

Research has been conducted within the context of the Joint targeted Program in Information and Communication Science and Technology- ICST, supported by CNRS, Inria, and DST. Author B Srivathsan is partially funded by grants from Infosys Foundation, India, Tata Consultancy Services, India and the MATRICS project of the Science and Education Research Board, India.

REFERENCES

[1] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
[2] G. Behrmann, P. Bouyer, E. Fleury, and K.G Larsen. Static guard analysis in timed automata verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–270. Springer, 2003.
[3] G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. Lower and upper bounds in zone-based abstractions of timed automata. *STTT*, 8(3):204–215, 2006.
[4] C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *TACAS*, volume 1384 of *LNCS*, pages 313–329, 1998.
[5] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, pages 197–212, 1989.
[6] F. Herbreteau and B Srivathsan. Coarse abstractions make zeno behaviours difficult to detect. *Logical Methods in Computer Science*, 9, 2013.
[7] F. Herbreteau, B. Srivathsan, T.-T. Tran, and I. Walukiewicz. Why liveness for timed automata is hard, and what we can do about it. In *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 48:1–48:14, 2016.
[8] F. Herbreteau, B Srivathsan, and I. Walukiewicz. Better abstractions for timed automata. In *LICS*, pages 375–384. IEEE Computer Society, 2012.
[9] F. Herbreteau, B. Srivathsan, and I. Walukiewicz. Efficient emptiness check for timed Büchi automata. *Formal Methods in System Design*, 40(2):122–146, 2012.

[10] F. Herbreteau, B. Srivathsan, and I. Walukiewicz. Lazy abstractions for timed automata. In *CAV*, volume 8044 of *LNCS*, pages 990–1005, 2013.
[11] F. Herbreteau and T.-T. Tran. Improving search order for reachability testing in timed automata. In *FORMATS*, pages 124–139. Springer, 2015.
[12] Frédéric Herbreteau, B. Srivathsan, and Igor Walukiewicz. Better abstractions for timed automata. *Information and Computation*, 251:67–90, 2016.
[13] A. Laarman, Olesen M. C., Dalsgaard A. E., Larsen K. G., and J. van de Pol. Multi-core emptiness checking of timed Büchi automata using inclusion abstraction. In *CAV*, volume 8044 of *LNCS*, pages 968–983, 2013.
[14] G. Li. Checking timed Büchi automata emptiness using LU-abstractions. In *FORMATS*, pages 228–242, 2009.
[15] S. Tripakis. Checking timed büchi automata emptiness on simulation graphs. *ACM Transactions on Computational Logic (TOCL)*, 10(3):15, 2009.
[16] S. Tripakis and S. Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1):25–68, 2001.
[17] S. Tripakis, S. Yovine, and A. Bouajjani. Checking timed Büchi automata emptiness efficiently. *Formal Methods in System Design*, 26(3):267–292, 2005.
[18] UPPAAL CSMA/CD model. http://www.it.uu.se/research/group/darts/uppaal/benchmarks/genCSMA_CD.awk. Accessed: 2014-10-08.