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

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

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

Frédéric Herbreteau[1], B. Srivathsan[2], Thanh-Tung Tran[3], and Igor Walukiewicz[4]

1 Université de Bordeaux, Bordeaux INP, CNRS, LaBRI UMR5800, France
2 Chennai Mathematical Institute, India
3 Université de Bordeaux, Bordeaux INP, CNRS, LaBRI UMR5800, France
4 Université de Bordeaux, Bordeaux INP, CNRS, LaBRI UMR5800, France

Abstract

The liveness problem for timed automata asks if a given automaton has a run passing infinitely often through an accepting state. We show that unless P=NP, the liveness problem is more difficult than the reachability problem; more precisely, 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 NP-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 the existing solutions.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Timed automata, model-checking, liveness invariant, state subsumption

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2016.48

1 Introduction

Timed automata [1] are one of the standard models of 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. This typo removes practically all the interesting behaviors 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 indispensable.

Verification of reachability properties of timed automata is possible in practice thanks to zones and their abstractions [4, 3, 9]. Roughly, the standard approach used nowadays for safety properties performs a breadth first search (BFS) over the set of pairs (state, zone)

* This work has been supported by project AVeRTS – CEFIPRA – Indo-French Program in ICST – DST/CNRS ref. 218093. Author B. Srivathsan is partially funded by a grant from Infosys Foundation.

36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016).
Editors: Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen; Article No. 48; pp. 48:1–48:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

reachable in the automaton, storing only pairs with the maximal abstracted zones (with respect to inclusion). In jargon: the algorithm constructs a zone graph with subsumption.

In this paper we give a 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, there is a problem with the approach outlined above as it is no longer sound to keep only maximal zones with respect to inclusion (i.e. to use subsumption) [11, 13]. It is possible to use the zone graph without subsumption, but this one is almost always too big to handle. One could hope though 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 that for safety properties. We show that this is impossible. We present a family of examples where reachability is much easier to decide than verification of Büchi properties. This proves that unless P=NP, there is no hope to obtain 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 safety 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 zone graph with unsafe subsumptions. This algorithm computes the zone graph without subsumption in the worst case - this as we show is anyway necessary in some cases. The expected advantage is that in many cases our algorithm can stop sooner. We have implemented our algorithm and run it on a set of benchmarks from [11]. On these examples indeed the algorithm mostly stops after the first iteration, and constructs witnesses of size very close to those for safety. To complete the picture we also give a set of particularly hard examples for our algorithm.

Related work: Verification of liveness properties is decidable thanks to the region construction [1]. The use of zones and (certain) abstractions for this problem was developed in [13]. Later Li [12] has shown that existence of a Büchi run is preserved by every abstraction based on simulation. In particular, this is the case for the $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction [3] that is the coarsest abstraction depending only on lower and upper bounds in clock guards (LU-bounds) [7]. Thanks to these results the 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 [13]. Laarman et al. [11] 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 [11]. 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 implying the above stated algorithmic difference between verification of liveness and reachability properties. In Section 5 we propose an algorithm for finding such witnesses and prove its correctness. Section 6 reports some experimental results.

2 Preliminaries

In this section we present the basic definitions. In particular, we define abstract zone graphs, and the use of subsumption. We also present 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 liveness properties from [11].

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 $\mathbb{R}_{\geq 0}^X$. 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 $v \models \phi$, 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 1** (Timed Büchi Automata [1]). A *Timed Büchi Automaton (TBA in short)* 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 $(q, v) \in Q \times \mathbb{R}_{\geq 0}^X$, 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 $v \models g$ 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 is termed the *Büchi non-emptiness problem*.

▶ **Definition 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.

The Büchi non-emptiness problem is known to be PSPACE-complete [1]. Standard solutions to this problem construct an untimed Büchi automaton and check for its emptiness. There are various methods to handle the non-Zeno requirement [15, 8]. In this paper, we will assume that the automata are strongly non-Zeno [13], that is, every infinite accepting run is non-Zeno. The strongly non-Zeno construction could lead to an exponential blowup [8, 6] 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.

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* $ZG(\mathcal{A})$ has as nodes pairs (q, Z) consisting of a state of the TBA and a zone. The initial node is (q_0, Z_0) where $Z_0 = \{\mathbf{0} + \delta \mid \delta \in \mathbb{R}_{\geq 0}\}$. For every $t = (q, g, R, q') \in T$, and every set of valuations W, we define the transition $\Rightarrow^t$ as: $(q, W) \Rightarrow^t (q', W')$ where $W' = \{v' \mid \exists v \in W,\ \exists \delta \in \mathbb{R}_{\geq 0} : (q, v) \xrightarrow{t}\xrightarrow{\delta} (q', v')\}$. 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 $t \in T$.

Although the zone graph $ZG(\mathcal{A})$ groups together valuations, the number of zones is still infinite [4]. For effectiveness, zones are further abstracted. An *abstraction operator* is a function $\mathfrak{a} : \mathcal{P}(\mathbb{R}_{\geq 0}^{|X|}) \to \mathcal{P}(\mathbb{R}_{\geq 0}^{|X|})$ such that $W \subseteq \mathfrak{a}(W)$ 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: $(q, W) \Rightarrow_{\mathfrak{a}}^t (q', \mathfrak{a}(W'))$ when $\mathfrak{a}(W) = W$ and $(q, W) \Rightarrow^t (q', W')$. 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 as nodes 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 $q \in F$.

Abstractions for timed automata are parameterized by the maximum constants appearing in the guards of the automaton. The structure of the automaton determines two functions $L : X \mapsto \mathbb{N}$ and $U : X \mapsto \mathbb{N}$. For a clock x, the value $L(x)$ denotes the maximum constant occurring in guards of the form $x \geq c$ or $x > c$; and the value $U(x)$ denotes the maximum constant occurring in guards $x \leq c$ or $x < c$. 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. It was shown in [7] that the $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction induces the smallest zone graphs, for a given bound function LU. Moreover, we know from [12] that $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$ is sound and complete for Büchi non-emptiness: TBA $\mathcal{A}$ has a run satisfying the Büchi condition iff $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$ has one. 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 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 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 [7]. The algorithm for computing $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$ is shown in Figure 1. Since we consider only $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$ in the rest of the paper, we will denote the transition relation $\Rightarrow_{\mathfrak{a}_{\preccurlyeq_{LU}}}$ by $\rightarrow$, as shown in Figure 1, for convenience. 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.

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

```
procedure abstract_zone_graph(𝒜)
V := {(q0, Z0)}, Waiting := {(q0, Z0)}
→ := ∅ // edge relation
while (Waiting ≠ ∅)
  take and remove (q, Z) from Waiting
  for each t = (q, g, R, q') ∈ 𝒜
    compute (q, Z) ⇒^t (q', Z')
    if ∃(q', Z1) ∈ V s.t 𝔞_≼LU(Z') = 𝔞_≼LU(Z1)
      add (q, Z) → (q', Z1)
    else
      add (q', Z') to V and Waiting
      add (q, Z) → (q', Z')
return (V, →)

procedure subsumption_graph(𝒜)
V := {(q0, Z0)}, Waiting := {(q0, Z0)}
→ := ∅ // edge relation
⇝ := ∅ // subsumption relation
while (Waiting ≠ ∅)
  take and remove (q, Z) from Waiting
  for each t = (q, g, R, q') ∈ 𝒜
    compute (q, Z) ⇒^t (q', Z')
    if ∃(q', Z1) ∈ V s.t 𝔞_≼LU(Z') = 𝔞_≼LU(Z1)
      add (q, Z) → (q', Z1)
    else if ∃(q', Z1) ∈ V s.t. Z' ⊆ 𝔞_≼LU(Z1)
      add (q', Z') to V
      add (q, Z) → (q', Z') ⇝ (q', Z1)
    else
      add (q', Z') to V and Waiting
      add (q, Z) → (q', Z')
return (V, →, ⇝)
```

Figure 1 Algorithm on the left computes $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$. The algorithm on the right uses subsumption. Methods for testing $Z' \subseteq \mathfrak{a}_{\preccurlyeq_{LU}}(Z_1)$ and $\mathfrak{a}_{\preccurlyeq_{LU}}(Z') = \mathfrak{a}_{\preccurlyeq_{LU}}(Z_1)$ are given in [7].

```
1  procedure ndfs()
2    Cyan := Blue := Red := ∅
3    dfsBlue(s0)
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}
```

Figure 2 Nested DFS algorithm with subsumption [11] to compute a subgraph of $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$.

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 $t.Z \subseteq \mathfrak{a}_{\preccurlyeq_{LU}}(s.Z)$. 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 Figure 1.

Subsumption optimization is known to give substantial gains for the reachability problem [10]. However, subsumption is not a priori correct for liveness, and the question of how it can be used for liveness was raised in [13]. An algorithm proposed in [11] (illustrated in Figure 2) gives a restricted way of using subsumption in a nested dfs algorithm for detecting accepting cycles. 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 (cf. 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 as to when subsumption can be used correctly for liveness analysis. We start with an example. Consider the TBA $\mathcal{A}$ and $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$ illustrated in Figure 3. The zone graph has an accepting cycle on the node $(1, 101 \leq y-x)$. For each of the states 1, 2 and 3 of the TBA, there are at least 100 nodes in the zone graph. Note that $(1, 1 \leq y-x) \sqsubseteq (1, 0 \leq y-x)$ and $(2, 1 \leq y-x \leq 101) \sqsubseteq (2, 0 \leq y-x)$. If we allow the luxury to use subsumptions freely, we would get the graph consisting only of the green nodes in the figure. However, in this graph there is no accepting cycle made uniquely of $\rightarrow$ edges. There are cycles containing subsumption edges but, as we will see later, it is not sound to take such cycles as witnesses for the existence of an accepting computation. Hence the green graph is not complete for liveness: to detect an accepting computation we should not use subsumption on $(1, 1 \leq y - x)$. Observe that using subsumption on the node $(2, 1 \leq y - x \leq 101)$ 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 green and grey nodes, along with the subsumption edge on the right is a liveness complete graph according to the definition below. Algorithm in Figure 2 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 $\rightarrow$ edges iff $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$ contains one. Including the subsumption edges as part of a cycle is not sound in general - for instance in Figure 3, the subsumption edge on state 2 forms a cycle, whereas there is no cycle containing 2 in $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$. In this paper, we do not include the subsumption edges as part of cycles. Hence in the graphs that we construct, cycles are actual cycles in $ZG^{\mathfrak{a}_{\preccurlyeq_{LU}}}(\mathcal{A})$ - so every such cycle with an accepting state gives 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 zone graph with subsumptions, and then follow up with a condition that makes a zone graph with subsumption complete for liveness.

▶ **Definition 3** (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 $s \rightarrow s'$ 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.

▶ **Lemma 4.** *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.*

Lemma 4 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

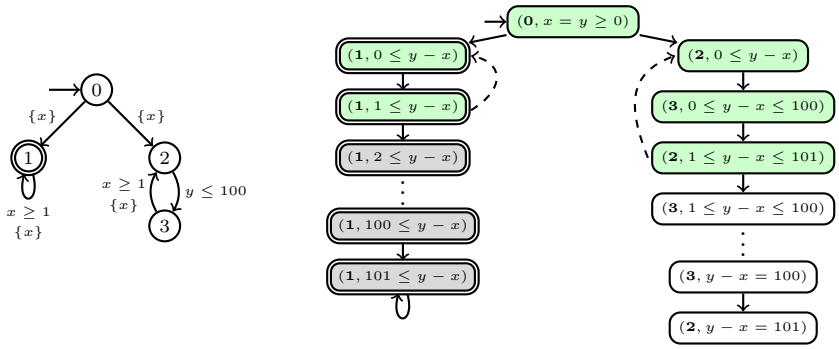

Figure 3 On the left is a TBA $\mathcal{A}$; on the right the graph without the dashed edges is $ZG^{\mathfrak{a}\preccurlyeq_{LU}}(\mathcal{A})$. Assume that $L = U = 100$ at every state - this can be achieved by adding more transitions on each state (which are not shown for clarity). Dashed edges show subsumption. The part of $ZG^{\mathfrak{a}\preccurlyeq_{LU}}(\mathcal{A})$ restricted to green nodes and dashed edges is the zone graph with subsumption. In this green graph there is no accepting cycle consisting of $\rightarrow$ edges. Removing the dashed edge on the node $(1, 1 \leq y - x)$ and adding the grey nodes identifies the accepting cycle.

– for a cycle of $\rightarrow$ edges in the zone graph, we may not get a corresponding cycle of $\rightarrow$ edges in the subsumption graph (cf. Figure 3). We now give an extra criterion.

▶ **Definition 5** (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 3, the zone graph restricted to green nodes and the dashed edges is not liveness compatible. There is a cycle containing an accepting node $(1, 1 \leq y - x)$ and a subsumption edge from this node. However, removing this subsumption edge and adding the grey nodes makes it liveness compatible. The only subsumption edge is from $(2, 1 \leq y - x \leq 101)$ 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 6.** *$ZG^{\mathfrak{a}\preccurlyeq_{LU}}(\mathcal{A})$ 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 4, 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 to s'. This gives an infinite path in G made of $\rightarrow$ edges that visits an accepting node s' infinitely often. ◀

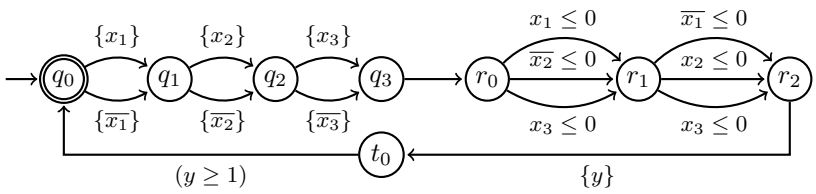

Figure 4 Automaton for $\phi = (p_1 \vee \neg p_2 \vee p_3) \wedge (\neg p_1 \vee p_2 \vee p_3)$.

4 Liveness is more difficult than reachability

Theorem 6 says that to solve the Büchi non-emptiness problem, one can compute a liveness compatible subsumption graph and check for cycles containing an accepting node and no subsumption edge. In general, liveness compatible subsumption graphs are smaller than the zone graph, but bigger than the usual subsumption graphs computed for reachability (cf. Figure 3). A natural question now is to ask if one can quantify this overhead created due to the liveness compatibility. In this section, we show that deciding liveness from a (reachability compatible) subsumption graph is NP-hard. Therefore, unless P=NP, one needs an object exponentially bigger than subsumption graphs to decide liveness. The proof of the following theorem uses the same kind of gadget as in [6].

▶ **Theorem 7.** *Given a strongly non-Zeno TBA and its subsumption graph, deciding Büchi non-emptiness is* NP*-hard.*

Proof. We give a reduction from 3SAT. Let $P = \{p_1, \ldots, p_k\}$ be a set of propositional variables and let $\phi = C_1 \wedge \cdots \wedge C_n$ be a 3CNF formula with n clauses. We will construct an automaton $\mathcal{B}_\phi$ such that ϕ has a satisfying assignment iff $\mathcal{B}_\phi$ has an infinite run satisfying the Büchi condition. Moreover, we give a subsumption graph with the same number of nodes as the number of states in $\mathcal{B}_\phi$.

The timed automaton $\mathcal{B}_\phi$ is defined as follows. For each propositional variable p_i, there are two clocks, x_i and $\overline{x}_i$, which encode the value of p_i when the value of one of the two clocks is 0 and the other is not. In addition, there is a clock y. In all, the set of clocks X is $\{x_1, \overline{x_1}, \ldots, x_k, \overline{x_k}, y\}$. For a literal λ, let $cl(\lambda)$ denote the clock x_i when $\lambda = p_i$ and the clock $\overline{x_i}$ when $\lambda = \neg p_i$. The set of states of $\mathcal{B}_\phi$ is $\{q_0, \ldots, q_k, r_0, \ldots, r_n, t_0\}$ with q_0 being the initial and the accepting state. The transitions are as follows: for each propositional variable p_i we have transitions $q_{i-1} \xrightarrow{\{x_i\}} q_i$ and $q_{i-1} \xrightarrow{\{\overline{x_i}\}} q_i$; for each clause $C_m = \lambda_1^m \vee \lambda_2^m \vee \lambda_3^m$, $m = 1, \ldots, n$, there are three transitions $r_{m-1} \xrightarrow{cl(\lambda) \leq 0} r_m$ for $\lambda \in \{\lambda_1^m, \lambda_2^m, \lambda_3^m\}$; a transition $q_k \to r_0$ with no guards and no resets; transitions $r_n \xrightarrow{\{y\}} t_0$ and $t_0 \xrightarrow{y \geq 1} q_0$.

Figure 4 shows the automaton for the formula $(p_1 \vee \neg p_2 \vee p_3) \wedge (\neg p_1 \vee p_2 \vee p_3)$. A path from q_0 to q_3 encodes an assignment with the following convention: a reset of x_i represents $p_i \mapsto true$ and a reset of $\overline{x_i}$ means $p_i \mapsto false$. Then, from r_0 to r_2 we check if the formula is satisfied by this guessed assignment (clearly there is no point to let the time pass in this process). The additional parts are the transitions from $r_2 \to t_0$ and $t_0 \to q_0$. Note that this makes the automaton strongly non-Zeno: between two consecutive visits to q_0, at least 1 time unit should elapse.

If ϕ has a satisfying assignment, then for every p_i the path that resets x_i if p_i is true and resets $\overline{x}_i$ if p_i is false can be extended to a run reaching t_0. For instance, the formula from Figure 4 is satisfied by every assignment that maps p_3 to $true$. The path corresponding to such an assignment passes through the transition $q_2 \xrightarrow{\{x_3\}} q_3$. Then, it has the possibility

to follow transitions $r_0 \xrightarrow{x_3 \leq 0} r_1$ and $r_1 \xrightarrow{x_3 \leq 0} r_2$. Note that this is possible independently on the amount of time elapsed at q_0. In particular, after the first iteration from $q_0 \to q_0$ the values of all clocks become bigger than 1. By following the path corresponding to the satisfying assignment each time, q_0 can be visited infinitely often. Conversely, suppose $\mathcal{B}_\phi$ has an infinite run that visits q_0 infinitely often. Since after one iteration $q_0 \to q_0$ the value of all clocks are above 1, the only way to take transitions $r_0 \to r_1 \cdots \to r_n$ is by resetting clocks appropriately so that at least one guard is satisfied at every r_i. One such sequence of resets would then give a satisfying assignment for ϕ.

We now give an idea of the subsumption graph G_ϕ for $\mathcal{B}_\phi$ as computed by the algorithm in the right hand side of Figure 1. It turns out that for every state $q_0, \ldots, q_n$ and $r_0, \ldots, r_n$ there is a single node with the zone $\mathfrak{a}_{\preccurlyeq_{LU}}(Z_0)$, where Z_0 is the initial zone, consisting of the time successors of v_0 (the valuation mapping every clock to 0). This is because $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction does not remember the order of resets. If it had, there would be exponentially many nodes at r_n. It can be checked that transitions $r_n \to t_0 \to q_0$ yield a node $(q_0, \mathfrak{a}_{\preccurlyeq_{LU}}(Z_0'))$ where $\mathfrak{a}_{\preccurlyeq_{LU}}(Z_0')$ is strictly included in $\mathfrak{a}_{\preccurlyeq_{LU}}(Z_0)$. ◀

The graph G_ϕ in the previous proof is not a liveness compatible subsumption graph. An algorithm for liveness would explore further from $(q_0, \mathfrak{a}_{\preccurlyeq_{LU}}(Z_0'))$ till a complete graph is obtained. Theorem 7, says that this process essentially leads to SAT solving. The above theorem holds even if the less coarse abstraction Extra^+_{LU} [3] is used instead of $\mathfrak{a}_{\preccurlyeq_{LU}}$.

5 An algorithm

We now consider the algorithmic problem of computing a liveness compatible subsumption graph for a given automaton. 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 1. A better solution is the nested DFS algorithm in Figure 2 - 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 5. Figure 5 illustrates the idea. The picture on the left shows the situation after the first reachability-SCC analysis. At this point, each violating subsumption edge is removed and a subsumption graph computation is started from the corresponding covered node (nodes a and e in the figure). During this exploration, the use of subsumption is restricted in order to avoid falling repeatedly into the same bad cycle: as a is covered by b then i must be covered by c, but this covering edge will form a bad cycle anyway, so there is no point of introducing it. To achieve this behavior, a *level* field is added to each node and subsumption is allowed only to nodes of a higher level (subsumption $j \rightsquigarrow k$ and the subsumption on node l, assuming nodes in the white part get level ∞). The iteration between reachability and cycle detection phases continues till the computed graph does not have violating cycles.

The problem with this approach is that the same edges will be considered in an unbounded number of SCC analyses. To avoid this, each SCC analysis phase is restricted to the nodes and edges of the current level. This however could miss out certain violations that span across levels (like the one caused by the edge $o \to f$). The following algorithm handles this issue by considering all such exit edges as potential causes of violation.

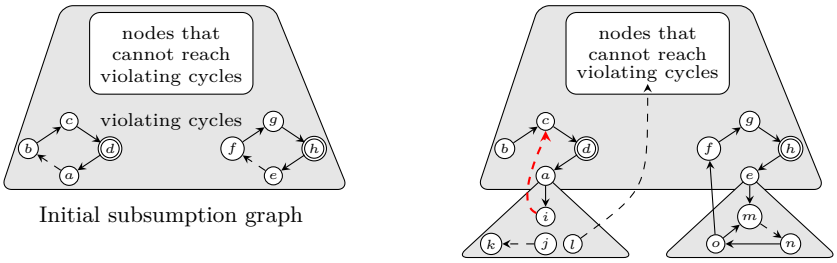

Figure 5 On the left is an illustration of a subsumption graph. An SCC decomposition of this graph gives the nodes that are part of violating cycles, and those that cannot reach violating cycles. On the right, is a re-exploration from bad subsumptions. In the re-exploration, subsumption is restricted to nodes of same level, or nodes that are known not to reach violating cycles.

Iterative SCC based algorithm with subsumption:

Phase 0 Let $K = 1$ and let S_{init} and S be the sets containing the initial (state,zone) pair.

Phase 1 Construct a subsumption graph from nodes in S_{init}. Set the level field of all nodes in S_{init} to K and let S_{init} be the pool of nodes to be explored. Every node added in this phase will have level field set to K. Repeatedly, take a node (q, Z) from the pool. For every edge $(q, Z) \Rightarrow (q', Z')$, add node (q', Z') to S and to the pool unless there is already $(q', Z_1) \in S$ s.t., either:

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

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

In the first case, add the edge $(q, Z) \to (q', Z_1)$ to S. In the second case, add the node (q', Z') to S and the edges $(q, Z) \to (q', Z')$, $(q', Z') \rightsquigarrow (q', Z_1)$ to S. By the end of this phase, graph S will be extended with some nodes of level K.

Phase 2 Consider the subgraph G_K of S induced by nodes of level K, and containing all the $\to$ and $\rightsquigarrow$ edges between these nodes. Decompose G_K into maximal SCCs by considering both $\to$ and $\rightsquigarrow$ as the same kind of edges. Mark a maximal SCC as *bad* if:

2.1 either it contains an accepting node and a subsumption edge (both nodes adjacent to the $\rightsquigarrow$ edge must be in the SCC), or

2.2 it has a node s with a successor edge $s \to s'$ to a node s' of level strictly less than K. Note that node s is in G_K, but s' is not.

For every node in G_K: set the flag of the node to ∞ if it cannot reach a bad SCC.

Phase 3 Let S_{init} be all covered nodes in G_K which still have level K. Remove the corresponding subsumption edges. Set $K := K + 1$.

Repeat If S_{init} is non-empty, restart from Phase 1.

Final If S_{init} is empty, perform an SCC decomposition of S. If there is an accepting cycle of $\to$ edges, return *non-empty*, else return *empty*.

In the above algorithm, Phase 1 is a reachability computation and Phases 2 and 3 are SCC-analysis. The final phase is a usual cycle detection algorithm.

▶ **Theorem 8.** *A strongly non-Zeno TBA $\mathcal{A}$ is non-empty iff the iterative SCC based algorithm with subsumption returns non-empty.*

Although the iterative algorithm performs an unbounded number of calls to an SCC decomposition, each edge is visited in only two SCC decompositions - one in the Phase 2 of the iteration corresponding to the level of the source node when it appeared, and the second

Table 1 Comparison of the size of liveness invariants generated by nested DFS algorithm, nested DFS algorithm with subsumption [11] and our Iterative algorithm running a Topological Search (see [10]) on the benchmark from [11]. We used a Linux station with an Intel i7-2600 3.40GHz processor and 8Gb of memory.

Prop	Sat	Nested DFS		NDFS subsumption		Iterative TS			UPPAAL TS
		# nodes	sec.	# nodes	sec.	# nodes	K	sec.	# nodes
Fi1	√	26 651	0.2	26 651	0.2	7 737	1	0.3	7 737
Fi2	√	132 808	1.4	132 808	1.3	38 238	1	1.0	38 238
Fi3		26 679	0.4	26 679	0.4	20 768	1	0.8	20 768
FD1	√	19 858	0.3	18 246	0.2	705	1	0.1	705
CC1	√	1 786 399	31.6	1 786 399	32.7	15 837	1	1.3	15 837
CC2	√	22 070	0.4	22 070	0.4	12 898	1	0.3	12 898

one during the final phase. This gives a constant upper bound on the number of times an edge is visited, irrespective of the number of iterations needed for stabilization. Therefore the time spent by the algorithm is linear in the size of the final graph computed, similar to the algorithm in Figure 2 which visits each edge twice. Moreover the result is always included in an abstract zone graph (without subsumption).

Let us now compare this approach with the nested DFS algorithm from Figure 2. The advantages of our algorithm are clearly visible on a (quite pathological) case of an automaton having no reachable accepting state. In this case, nested DFS does not use subsumption at all as there are no red nodes. On the other hand, the iterative algorithm computes just a reachability subsumption graph. After the first iteration, there would be no bad SCCs, and hence the entire graph would be marked ∞. Another important class of automata for which we get a significant gain is weak timed Büchi automata (every cycle in a weak TBA consists entirely of accepting nodes or non-accepting nodes). As there can be no SCCs containing both accepting and non-accepting nodes, the iterative algorithm stabilizes after 1 iteration. The automaton of Figure 3 is a weak Büchi automaton. The nested DFS algorithm computes the entire zone graph - no subsumption will be allowed. The iterative algorithm would allow subsumption on the non-accepting part. It computes the graph consisting of the green and grey nodes. There are examples where the nested DFS approach can outperform the iterative algorithm. Modify the automaton in Figure 3 as follows: make all states accepting; add another clock w and an edge $0 \xrightarrow{w \geq 5} 2$. Note that the iterative algorithm cannot use subsumption at all since all nodes are accepting. Moreover this additional edge leads to $n_1 : (2, 0 \leq y - x \wedge w \geq 5)$ which starts a long thread of zones due to the transitions between 2 and 3. Note that $n : (2, 0 \leq y - x \wedge w \geq 5)$ is subsumed by $(2, 0 \leq y - x)$. If the NDFS chooses a good order and finishes exploring the latter zone first, it allows subsumption $n_1 \rightsquigarrow n$, and avoids computing the subtree below n_1.

6 Experiments

We have implemented our algorithm in our tool TChecker. Our implementation includes various optimizations that are not presented in this paper (the algorithm stops after Phase 1 when no accepting state is reachable, covered nodes are not kept in the liveness invariants, etc.). We have compared our algorithm with our own implementation of the nested DFS algorithm with subsumption from [11]. We have conducted experiments on the classical benchmarks for Timed Automata, that we describe below. 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.

Table 2 Comparison of the nested DFS algorithm with subsumption[11] and our Iterative algorithm running a Topological Search (see[10]) on properties with timed constraints. We used a Linux station with an Intel i7-2600 3.40GHz processor and 8Gb of memory.

Prop	Sat	NDFS subsumption		Iterative TS			UPPAAL TS
		# visited	# nodes	# visited	# nodes	K	# nodes
Fig 4		58	50	116	50	3	8
(CC3) 8		16 165	16 164	219 561	205 656	1	205 656
(CC4) 3	√	44 873	35 787	449 724	150 078	282	367
(CC5) 3	√	240 111	237 548	606 421	201 740	60	772
(Fi4) 8	√	466 572	382 936	154 854	77 427	1	77 427
(Fi5) 4	√	48 299	24 979	88 430	29 677	17	704

As a first experiment, we considered the models and properties in [11]. Fischer's protocol is a mutual-exclusion protocol based on real-time constraints. Let c denote the number of processes in critical section. We checked properties (Fi1) $\mathsf{G}(c \leq 1)$ mutual exclusion; (Fi2) $\mathsf{GF}(c = 0) \wedge \mathsf{GF}(c = 1)$ non-blocking from [12]; 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 3 models with 7 processes and standard parameter values from the literature.

Table 1 shows a comparison between the standard nested DFS algorithm, nested DFS algorithm with subsumption [11] and our Iterative algorithm. The first column corresponds to the property checked as described above. A tick in the second column indicates that the property holds: the Büchi automaton for the complement has no accepting run. For every algorithm, we report the size of the liveness invariant (# nodes) and the running time (sec.). We also report the maximum level (K) for the Iterative algorithm. The last column gives the size of the reachability invariant as computed by our implementation of UPPAAL's algorithm [16]. This is a lower bound on the size of the liveness invariant. Both our Iterative algorithm and UPPAAL's algorithm explore the state-space of the automata using a topological search [10].

The results show that our Iterative algorithm computes liveness invariants that are significantly smaller that those computed by both nested-DFS algorithms. Indeed, for all these examples, the reachability invariants are liveness compatible as shown by the comparison to UPPAAL's algorithm. Our algorithm also visits significantly less nodes than nested-DFS algorithms. It 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, the Iterative algorithm stops after Phase 2 and skips Phase 3 and the final phase. Finally, (Fi3) has an accepting run that is found during Phase 2.

We have also compared the algorithms on examples that are particularly difficult for Iterative algorithm. The results are shown in Table 2. We report for each model the number of visited nodes (# visited) and the size of the liveness invariants (# nodes). We also compare to the size of reachability invariants generated by UPPAAL's algorithm. The first example corresponds to the automaton in Figure 4. Since it is built from a formula that is satisfiable, the automaton has an accepting run. However, the accepting cycle spans over two levels. As a result, our Iterative algorithm required 2 refinements ($K = 3$), and it had to run all phases to detect the accepting run. The nested-DFS algorithm explores significantly less nodes.

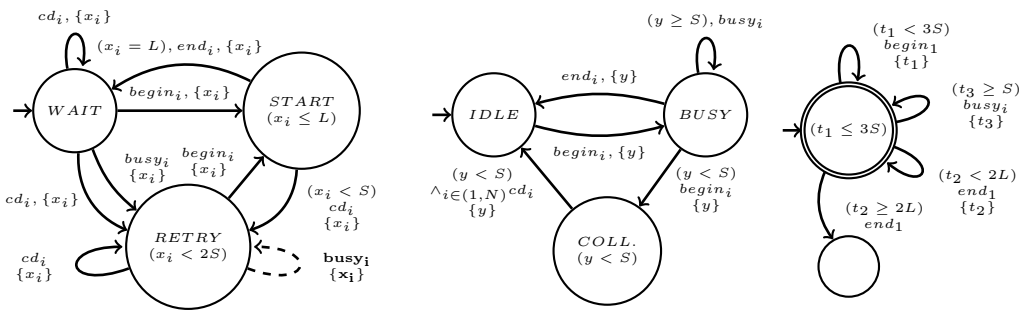

Figure 6 Model of CSMA/CD: station (left) and bus (middle); property (CC3) (right).

The other examples are built from the CSMA/CD model (CC) and the Fischer model (Fi) described above. We consider timed specifications with timed constraints that make covering harder. Property (CC3) is depicted in Figure 6 (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 $t_4 \leq K$ to the accepting state. Property (CC5) checks that if collisions are infrequent and station 1 tries infinitely often to send, then it effectively sends messages infinitely often. Property (Fi4) expresses that if process 1 can infinitely often access the critical section for K 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 (CC3) has accepting runs, thus the nested-DFS algorithm with subsumption performs significantly better than our Iterative algorithm. Indeed, the Iterative algorithm first computes a reachability invariant in Phase 1 before being able to detect the accepting run in Phase 2. Properties (CC4) and (Fi5) with bounded iterations of cycles are difficult for our Iterative algorithm. These two automata generate many bad SCCs, hence many refinements. At each refinement step, our algorithm needs to generate many new nodes as covering is restricted to nodes within the same level (or nodes with level ∞). This results in bigger liveness invariants than those computed by nested-DFS algorithm with subsumption. On the examples (CC5) and (Fi4), on the contrary, our algorithm generates smaller invariants than nested-DFS with subsumption. Notice that in most examples, the liveness invariants computed by both algorithms are huge w.r.t. reachability invariants computed by UPPAAL's algorithm.

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 [14, 17]. In consequence, once some process enters in 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 6. 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.

7 Conclusions

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

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, and I. Walukiewicz. Better abstractions for timed automata. In *LICS*, pages 375–384. IEEE Computer Society, 2012.
8. F. Herbreteau, B. Srivathsan, and I. Walukiewicz. Efficient emptiness check for timed Büchi automata. *Formal Methods in System Design*, 40(2):122–146, 2012.
9. F. Herbreteau, B. Srivathsan, and I. Walukiewicz. Lazy abstractions for timed automata. In *CAV*, volume 8044 of *LNCS*, pages 990–1005, 2013.
10. F. Herbreteau and T.-T. Tran. Improving search order for reachability testing in timed automata. In *FORMATS*, pages 124–139. Springer, 2015.
11. A. Laarman, M. C. Olesen, A. E. Dalsgaard, K. G. Larsen, 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.
12. G. Li. Checking timed Büchi automata emptiness using LU-abstractions. In *FORMATS*, pages 228–242, 2009.
13. S. Tripakis. Checking timed büchi automata emptiness on simulation graphs. *ACM Transactions on Computational Logic (TOCL)*, 10(3):15, 2009.
14. S. Tripakis and S. Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1):25–68, 2001.
15. S. Tripakis, S. Yovine, and A. Bouajjani. Checking timed Büchi automata emptiness efficiently. *Formal Methods in System Design*, 26(3):267–292, 2005.
16. UPPAAL. `http://www.uppaal.org`.
17. UPPAAL CSMA/CD model. `http://www.it.uu.se/research/group/darts/uppaal/benchmarks/genCSMA_CD.awk`. Accessed: 2014-10-08.