Fast zone-based algorithms for reachability in pushdown timed automata

S Akshay, Paul Gastin, Karthik R Prakash

Fast zone-based algorithms for reachability in pushdown timed automata*

S. Akshay[1][0000−0002−2471−5997], Paul Gastin[2][0000−0002−1313−7722], and Karthik R. Prakash[1][0000−0003−4304−1382]

[1] Department of CSE, Indian Institute of Technology Bombay, Mumbai, India
{akshayss,karthikrprakash}@cse.iitb.ac.in
[2] Université Paris-Saclay, ENS Paris-Saclay, CNRS, LMF, 91190, France
paul.gastin@lsv.fr

Abstract. Given the versatility of timed automata a huge body of work has evolved that considers extensions of timed automata. One extension that has received a lot of interest is timed automata with a, possibly unbounded, stack, also called pushdown timed automata (PDTA). While different algorithms have been given for reachability in different variants of this model, most of these results are purely theoretical and do not give rise to efficient implementations. One main reason for this is that none of these algorithms (and the implementations that exist) use the so-called zone-based abstraction, but rely either on the region-abstraction or other approaches, which are significantly harder to implement.
In this paper, we show that a naive extension, using simulations, of the zone based reachability algorithm for the control state reachability problem of timed automata is not sound in the presence of a stack. To understand this better we give an inductive rule based view of the zone reachability algorithm for timed automata. This alternate view allows us to analyze and adapt the rules to also work for pushdown timed automata. We obtain the first zone-based algorithm for PDTA which is terminating, sound and complete. We implement our algorithm in the tool TChecker and perform experiments to show its efficacy, thus leading the way for more practical approaches to the verification of timed pushdown systems.

Keywords: Timed automata, Zone-based abstractions, Pushdown automata, Simulations, Reachability

1 Introduction

Timed automata [7] are a popular formalism for capturing real-time systems, and of use for instance, in model checking of cyber-physical systems. They extend finite automata with real variables called clocks whose values increase over time; transitions are guarded by constraints over these variables. The main problem

* This work was partly supported by ReLaX CNRS IRL 2000, DST/CEFIPRA/INRIA project EQuaVE and SERB Matrices grant MTR/2018/00074.

of interest is the reachability problem, which asks whether a given state can be reached while satisfying the constraints imposed by the guards. This problem is known to be PSPACE-complete (already shown in [7]). The PSPACE algorithm, uses the so-called region-automaton construction, which essentially abstracts the timed automaton into an exponentially larger finite automaton of regions (collections of clock valuations), which is sound and complete for reachability.

Despite this complexity-theoretic hardness, the model of timed automata has proved to be extremely influential and versatile, resulting in an enormous body of work on its theory, variants and extensions over the past 25 years. Almost since its inception, researchers also began to develop tools to extend from theoretical algorithms to solve practical problems. Such tools range from the classical and richly featured tool UPPAAL [9,23] to the more recent open-source tool TChecker [19], which have been used on industry strength benchmarks and perform rather well on many of them. These tools use a different algorithm for reachability, where reachable sets of valuations are represented as zones and explored in a graph. While a naive exploration of zones does not terminate, the algorithms used identify different strategies [8,21,18], e.g., subsumption or simulations, extrapolations, for pruning the zone-based exploration graphs, while preserving soundness and completeness of reachability. While this does not change the worst case complexity, in practice, the zone exploration results in much better *practical* performance as it allows on-the-fly computation of reachable zones. One could even argue that the wider adoption of timed automata paradigm in the verification community has been a result of scalable implementations and tools built on this zone-based approach.

In light of this, zone-based algorithms are often looked for to improve practical performance of extensions of timed automata as well. For instance, for timed automata with diagonal constraints, classical zone-based approaches were shown to be unsound [11,12], but recently, an approach has been developed which adapts the existing construction and obtains fast zone-based algorithms [17]. In the present paper, we are interesting in adding a different feature to timed automata, namely an unbounded lifo-stack. This results in a powerful model of *pushdown timed automata (PDTA for short)*, in which the source of "infinity" is both from real-time and the unbounded stack. Unsurprisingly, this model and its variants have been widely studied over the last 20 years with several old and recent results on decidability of reachability, related problems and their complexity, including [1,2,3,4,5,10,13,14,15,16]. A wide variety of techniques have been employed to solve these problems, from region-based abstractions, to using atoms and systems of constraints, to encoding into different logics etc. However, except for [5,4], to the best of our knowledge, none of the others carry an implementation. In [5], the implementation uses a tree-automaton implicitly based on regions and the focus in [4] is towards multi-pushdown systems. A common factor of all these works is that none of them consider zone-based abstractions.

In this paper, we ask whether zone-based abstractions can be used to decide efficiently reachability questions in PDTA. We focus on the problem of well-nested control-state reachability of PDTA, i.e., given a PDTA, an initial and

a target state, does there exist a run of the PDTA that starts at the initial state with empty stack and reaches the target state with an empty stack (in between, i.e., during the run, the stack can indeed be non-empty). As with timed automata, our goal here is towards its applicability to build powerful tools which could lead to wider adoption of the PDTA model and showcase its utility to model-checking timed recursive systems. As the first step, we examine the difficulties involved in mixing zones with stacks and point out that a naive adaptation of the zone-based algorithm would not be sound. Then we propose a new algorithm that modifies the zone-based algorithm to work for pushdown timed automata. This is done in three steps.

- First we view the zone-graph exploration at the heart of the zone-based reachability algorithm for timed automata as a least fixed point computation of two inductive rules. When applied till saturation, they compute a sound and complete finite abstraction of the set of all reachable zones.
- Next, this view allows us to generalize the approach in the presence of a stack by adding new inductive rules that correspond to push and pop transitions, and hence are specific to the stack operation. There are two main technical difficulties in this. First, we need to ensure termination of the fixed point computation, using a strong enough pruning condition of the (a priori infinite) zone graph to ensure finiteness, while being sound and not adding spurious runs. Second, we want to aggressively prune the graph as much as possible to obtain an efficient zone-exploration algorithm. We show how we can minimally change the condition of pruning in the zone exploration graph to achieve this delicate balance. Indeed, in doing so we use a judicious combination of the subsumption (or simulation) relation and an equivalence relation for obtaining a fixed point computation for PDTA that is terminating, while being sound and complete.
- Finally, we build new data structures that allow us to write an efficient algorithm that implements this fixed point computation. While getting a correct algorithm is relatively simple, to obtain an efficient one, we must again encounter and overcome several technical difficulties.

We implement our approach to build the first zone-based tool that efficiently solves well-nested control state reachability for PDTA. Our tool is built on top of existing infrastructure of TChecker [19], an open source tool and benefits from many existing optimizations. We perform experiments to show the practical performance of multiple variants of our algorithm and show how our most optimized version is vastly better in performance than other variants and of course the earlier region-based approach on a suite of example benchmarks.

We note that our PDTA model differs slightly from the model considered in [1,3], as there is no age on stack and time spent on stack cannot be compared with clocks. Hence our model is closer to [10,16]. However, in [13], it was shown that these two models are equivalent, more specifically, the stack can be untimed without loss of expressivity (albeit with an exponential blowup). Thus our approach can be applied to the other model as well by just untiming the stack. There are other more powerful extensions [15,14] studied especially in the

context of binary reachability, where only theoretical results are known. We also remark that the idea of combining the subsumption relation between zones with an equivalence relation also occurs while tackling liveness, or Buchi acceptance, in timed automata. This has been studied in depth [24,22,20], where the naive zone-based algorithm does not work, forcing the authors to strengthen the simulation relation in different ways. Though these problems are quite different, there are surprising similarities in the issues faced, as explained in Section 3.

The structure of the paper is as follows: we start with preliminaries and move on to the difficulty in using zones and simulation relations in solving reachability in PDTA. Then, we introduce in Section 4 our inductive rules for timed automata and PDTA and show their correctness. In Section 5, we present our algorithm and helpful data-structural advancements. We show the experimental performance in Section 6 and end with a brief conclusion. Proofs that are missing and more experimental results can be found in the long version of the paper available at [6].

2 Preliminaries

2.1 Timed automata

Timed automata extend finite-state automata with a set X of (non-negative) real-valued variables called *clocks*. We let $\Phi(X)$ denote the set of constraints φ that can be formed using the grammar: $\varphi ::= x \sim c \mid x - y \sim c \mid \varphi \wedge \varphi$, where $x, y \in X$, $c \in \mathbb{N}$, $\sim \in \{\leq, \geq, <, >\}$, where each $x \sim c$ is called an atomic constraint. A clock valuation is a map $v\colon X \to \mathbb{R}_{\geq 0}$ and is said to satisfy φ, denoted $v \models \varphi$, if φ evaluates to true when each clock $x \in X$ is replaced with $v(x)$. For $\delta \in \mathbb{R}^{\geq 0}$, we write $v + \delta$ to denote the valuation defined as $(v + \delta)(x) = v(x) + \delta$ for all clocks x. For $R \subseteq X$, we write $[R]v$ to denote the valuation obtained by resetting clocks in R, i.e., $([R]v)(x) = 0$ if $x \in R$, and $([R]v)(x) = v(x)$ otherwise. Finally, v_0 is the valuation that sets all clocks to 0.

A timed automaton $\mathcal{A}$ is a tuple (Q, X, q_0, Δ, F), where Q is a finite set of states, X is a finite set of clocks, $q_0 \in Q$ is an initial state, $F \subseteq Q$ is the set of final states and $\Delta \subseteq Q \times \Phi(X) \times 2^X \times Q$ is a set of transitions. A transition $t \in \Delta$ is of the form (q, g, R, q'), where q, q' are states, $g \in \Phi(X)$ is the guard of the transition and $R \subseteq X$ is the set of clocks that are reset at the transition. The semantics of a timed automaton $\mathcal{A}$ is given as a transition system $TS(\mathcal{A})$ over configurations. A configuration is a pair (q, v) where $q \in Q$ is a state and v is a valuation, with the initial configuration being (q_0, v_0). The transitions are of two types. First, for a configuration (q, v) and $\delta \in \mathbb{R}^{\geq 0}$, $(q, v) \xrightarrow{\delta} (q, v + \delta)$ is a delay transition. Second, for $t = (q, g, R, q') \in \Delta$, $(q, v) \xrightarrow{t} (q', v')$ is a discrete transition if $v \models g$ and $v' = [R](v)$. A run is an alternating sequence of delays and discrete transitions starting from the initial configuration, and is said to be accepting if the last state in the sequence is a final state.

2.2 Reachability, Zones and simulations

The problem of control-state reachability asks whether a given timed automaton has an accepting run. This problem is known to be PSPACE-complete [7], originally shown via the so-called region abstraction. Note that, since $TS(\mathcal{A})$ is infinite, some abstraction is needed to get an algorithm. In practice however, the abstraction used to solve reachability, e.g., in tools such as UPPAAL [23] or TChecker [19] is the *zone abstraction*. A zone Z is defined as a set of valuations defined by a conjunction of atomic clock constraints. Given a guard g and reset R, we define the following operations on zones: time elapse $\overrightarrow{Z} = \{v + \delta \mid v \in Z, \delta \in \mathbb{R}^{\geq 0}\}$, guard intersection $g \cap Z = \{v \in Z \mid v \models g\}$ and reset $[R]Z = \{[R]v \mid v \in Z\}$. The resulting sets are also zones. With this, we can define the zone graph $ZG(\mathcal{A})$ as a transition system obtained as follows: the nodes are (state, zone) pairs and $(q, Z) \xrightarrow{t} (q', Z')$, if $t = (q, g, R, q')$ is a transition of $\mathcal{A}$ and $Z' = \overrightarrow{[R](g \cap Z)}$. The initial node is $(q_0, Z_0 = \overrightarrow{\{v_0\}})$ and a path in the zone graph is said to be accepting if it ends at an accepting state. The zone graph is known to be sound and complete for reachability, but as the graph may still be infinite, this does not give an algorithm for solving reachability yet.

To obtain an algorithm, one resorts to different techniques such as extrapolation or simulation. Here we focus on *simulation relations* which will lead to finite abstractions. Given a timed automaton $\mathcal{A}$, a binary relation $\preceq$ on configurations is called a *simulation* if whenever $(q, v) \preceq (q', v')$, we have $q = q'$ and

- for each delay $\delta \in \mathbb{R}^{\geq 0}$, $(q, v + \delta) \preceq (q, v' + \delta)$ and
- for each $t = (q, g, R, q_1) \in \Delta$, if $v \models g$ then $v' \models g$ and $(q_1, [R]v) \preceq (q_1, [R]v')$.

We often simply write $v \preceq_q v'$ instead of $(q, v) \preceq (q, v')$. We can now lift this to sets Z, Z' of valuations as $Z \preceq_q Z'$ if for all $v \in Z$ there exists $v' \in Z'$ such that $v \preceq_q v'$. We say that node (q, Z) is subsumbed by node (q, Z') when $Z \preceq_q Z'$. As a consequence we obtain the following lemma.

Lemma 1. *If $(q, Z) \xrightarrow{t} (q_1, Z_1)$ in $ZG(\mathcal{A})$ and $Z \preceq_q Z'$, then $(q, Z') \xrightarrow{t} (q_1, Z_1')$ and $Z_1 \preceq_{q_1} Z_1'$.*

Proof. Indeed, let $v_1 \in Z_1 = \overrightarrow{[R](g \cap Z)}$. We find $v \in Z$ and $\delta \geq 0$ such that $v \models g$ and $v_1 = [R]v + \delta$. Since $Z \preceq_q Z'$, we find $v' \in Z'$ with $v \preceq_q v'$. We deduce that $v' \models g$ and $[R]v \preceq_{q_1} [R]v'$, which implies $v_1 \preceq_{q_1} v_1'$ with $v_1' = [R]v' + \delta \in Z_1' = \overrightarrow{[R](g \cap Z')}$. □

A simulation $\preceq$ is said to be *finite* if for every sequence of nodes (q_1, Z_1), $(q_2, Z_2), \ldots$ there exist two nodes (q_i, Z_i) and (q_j, Z_j) with $i < j$ such that $q_i = q_j$ and $Z_j \preceq_{q_i} Z_i$. The importance of the finiteness is that it allows us to stop exploration of zones along a branch of the zone graph: when a node (q_j, Z_j) is reached which is subsumed by an earlier node (q_i, Z_i), we may cut the exploration since all control states reachable from the latter are already reachable from the former. For a timed automaton $\mathcal{A}$, we call this pruned graph as $ZG_{\preceq}(A)$. Thus, if the simulation relation $\preceq$ is finite, then $ZG_{\preceq}(\mathcal{A})$ is finite,

sound and complete for control state reachability. We formalize this algorithm in Section 4, using inductive rules.

Various finite simulations have been shown to exist in the literature, including the famous LU-abstractions [8], and more recent $\mathcal{G}$-abstractions based on sets of guards [17]. Hence this theory indeed has resulted in better implementations and is used in standard tools in this domain.

We will see that using simulation in the context of pushdown timed automata is not always sound, in some cases we need a stronger condition to stop the exploration. Towards this, we consider the equivalence relation on nodes induced by the simulation relation: $Z \sim_q Z'$ if $Z \preceq_q Z'$ and $Z' \preceq_q Z$. We say that the simulation $\preceq$ is strongly finite if the induced equivalence relation $\sim$ has finite index. Notice that strongly finite implies finite but the converse does not necessarily hold. Fortunately, the usual simulations for timed automata, in particular the LU-simulation and the $\mathcal{G}$-simulation, are strongly finite.

2.3 Pushdown timed automata (PDTA)

A Pushdown Timed Automaton $\mathcal{A}$ is a tuple $(Q, X, q_0, \Gamma, \Delta, F)$, where Q is a finite set of states, X is a finite set of clocks, $q_0 \in Q$ is an initial state, Γ is the stack alphabet, $F \subseteq Q$ is the set of final states and Δ is a set of transitions. A transition $t \in \Delta$ is of the form $(q, g, \mathrm{op}, R, q')$, where q, q' are states, $g \in \Phi(X)$ is the guard of the transition and $R \subseteq X$ is the set of clocks that are reset at the transition, op is one of three stack operations: nop or push_a or pop_a for some $a \in \Gamma$.

The semantics of a PDTA $\mathcal{A}$ is given as a transition system $TS(\mathcal{A})$ over configurations. A configuration here is a tuple (q, v, χ) where $q \in Q$ is a state, v is a valuation, $\chi \in \Gamma^*$ is the stack content, with the initial configuration being (q_0, v_0, ε). The transitions are of two types. First, for a configuration (q, v, χ) and $\delta \in \mathbb{R}^{\geq 0}$, $(q, v, \chi) \xrightarrow{\delta} (q, v + \delta, \chi)$ is a delay transition. Second, for $t = (q, g, \mathrm{op}, R, q') \in \Delta$, $(q, v, \chi) \xrightarrow{t} (q', v', \chi')$ is a discrete transition if $v \models g$, $v' = [R](v)$ and

- if op = nop, then $\chi' = \chi$,
- if op = push_a then $\chi' = \chi \cdot a$,
- if op = pop_a, then $\chi = \chi' \cdot a$.

A run is an alternating sequence of delays and discrete actions starting from the initial configuration. It is accepting if the last state in the sequence is final.

Our main focus is the *well-nested control state reachability* problem for PDTA, which asks whether a configuration (q, v, ε) with $q \in F$ is reachable, where the stack is empty. Later, in Section 7, we remark how our solution can be extended to solve general control state reachability, i.e., asking whether a configuration (q, v, χ) with $q \in F$ is reachable, possibly with a nonempty stack χ.

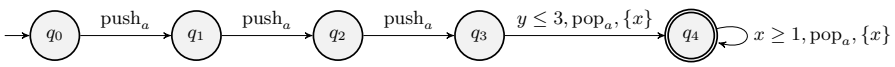


Fig. 1: A simple PDTA with 2 clocks $\{x, y\}$. Note that if we ignore the push/pop actions we get a TA, say A.

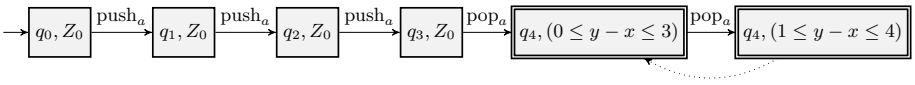


Fig. 2: Zone graph with simulation edges for finiteness. Again ignoring push/pop actions gives us a zone graph for the TA. Z_0 is the initial zone.

3 Zones in PDTA and the problem with simulations

As mentioned earlier, zones are collections of clock valuations defined by conjunctions of timing constraints, and exploring zones reached by a timed automaton gives a sound and complete abstraction for state reachability. To make sure that the exploration is finite we need to prune the graph and one way this is done by simulation, i.e., not exploring paths from some nodes if they are "subsumed" by earlier nodes visited in the graph. Consider Figure 1, in which we ignore the push_a and pop_a or we can think of them as internal actions. Then the usual zone-graph construction with simulation would give the graph depicted in Figure 2. In this section, just for illustration we instantiate the simulation relation to be the well-known LU-simulation (we do not give the definition here as it is not relevant to what comes later, instead we refer to earlier work [8]). Using this, we obtain that the rightmost node is subsumed by the previous one, and hence the dotted simulation edge. If we did not do this we immediately observe that we get an infinite graph with increasing sets of zones.

Now, our first question is whether this zone exploration with simulation can be lifted to PDTA. In this example, if we were to add back the push/pop edges, we get exactly the same Zone graph with annotations, and further, the final state is indeed reachable. Hence, for this particular example we do obtain a finite, sound and complete graph exploration. However, in general it turns out that the procedure is not sound.

Consider the example in Figure 3. In this example, again considering it as a TA (ignoring the push/pops), we would get the zone graph below, which would be finite, sound and complete for reachability in that TA. But if we consider it as a PDTA, now doing the same does not preserve soundness. In other words, in the PDTA, q_3 is no longer reachable. However, in the zone graph we would conclude that it is reachable due to the simulation edge. If, to fix this, we remove the dotted simulation edge, then we will lose finiteness.

Thus, it seems that we have a difficult situation where zones with the simulation relation, needed for termination, do not preserve soundness. This situation resembles the situation studied in [24,22,20], where the authors study liveness or Buchi-acceptance conditions in timed automata. Again in that situation, the

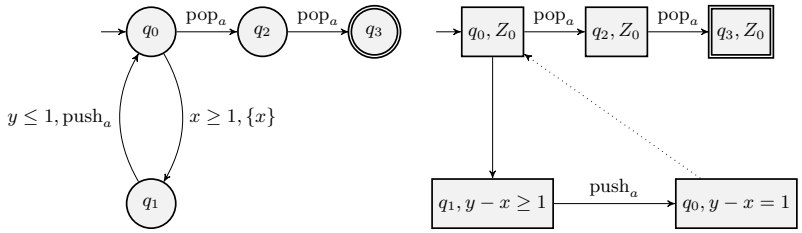


Fig. 3: A PDTA and its zone graph with simulation. With the simulation (dotted) edges, q_3 is reachable in the zone graph, but its not reachable in the PDTA.

naive algorithm with zone simulation does not work and the authors are forced to strengthen the simulation relation in different ways.

Surprisingly, it turns out, that even in our very different problem setting of reachability in PDTA, a similar solution works. That is, we replace simulation by equivalence (defined in the previous section) as the pruning criterion. However, there are two issues (i) it is not easy to prove its correctness and (ii) this is far from efficient as shown in the experimental section. Our goal to use zones in the first place was efficiency and hence we would like to prune the zone graph as much as possible, i.e., we would like to use simulation edges as much as possible. In the next two sections, we describe our fix. We first show a different view of the exploration algorithm as a fixed point rule based approach. This allows us to then describe our fix in the same language, which is much easier to understand conceptually. Also as a corollary we will be able to show that using equivalence everywhere also gives a correct algorithm. After proving the correctness of our rule-based algorithm, we then tackle the challenges in implementing it.

4 Viewing reachability algorithms using rewrite rules

In this section, our goal is to compute a set S of nodes of the zone graph of a PDTA, as a least fixed point of a small set of inductive rules, such that a control state q occurs in S, i.e., $(q, Z) \in S$ for some Z iff q is reachable in the PDTA from its initial state. To understand the rules and their correctness it is easier to first visualize this on plain timed automata without any push-pop edges.

4.1 Rewrite rules for Timed automata.

Given a TA $\mathcal{A} = (Q, X, q_0, \Delta, F)$, the set S containing all reachable nodes of the zone graph, can be obtained as the least fixed point of the following inductive rules, with a natural deduction style of presentation.

$$\frac{}{S := \{(q_0, Z_0)\}} \text{ start}$$

$$\frac{(q, Z) \in S \qquad q \xrightarrow{g,R} q' \qquad Z' = \overrightarrow{R(g \cap Z)} \neq \emptyset}{S := S \cup \{(q', Z')\}} \text{ Trans}$$

Let S^* denote the set at the fixed point by starting with the start rule and repeatedly applying the trans rule. It is easy to see that this computes the set of all reachable nodes of the zone graph: the start rule starts with the initial node and each application of trans rule takes a reachable node and applies a transition of the automaton and includes the resulting node reached. However, this set S^* is a priori infinite since number of zones is infinite.

To make it finite we add a condition under which we will apply the transition rule based on a finite simulation relation (let us denote it $\preceq$) for $\mathcal{A}$.

$$\frac{(q,Z) \in S \qquad q \xrightarrow{g,R} q' \qquad Z' = \overrightarrow{R(g \cap Z)} \neq \emptyset}{S := S \cup \{(q',Z')\}, \text{ unless } \exists (q',Z'') \in S,\ Z' \preceq_{q'} Z''} \text{ Trans-}\preceq$$

Thus to obtain an algorithm, we would explore all nodes in the Zone graph using a search algorithm (say DFS/BFS) and we would add a node only if it is not subsumed by an already visited node, according to the simulation relation. We explained in Section 2.2 that doing this preserves soundness and completeness and gives a finite exploration.

Lemma 2. *Let $S^*_{\preceq}$ denote any set obtained from the start rule and by repeatedly applying Trans-$\preceq$ till a fixed point is reached. Note that depending on the order of applications we may have different sets. Then we have:*

1. *(finiteness) $S^*_{\preceq}$ is finite.*
2. *(soundness and completeness) For all $q \in Q$, a configuration (q,v) is reachable from (q_0,v_0) in the TA $\mathcal{A}$ iff $(q,Z) \in S^*_{\preceq}$ for some zone Z.*

We do not give the proof here as (i) it is only a reformulation of known results and (ii) it will be subsumed by the much stronger theorem we prove next.

4.2 Rewrite rules for PDTA.

Let $\mathcal{A} = (Q, X, q_0, \Gamma, \Delta, F)$ be a PDTA, we will need not just a set but a tuple of sets. More precisely, we maintain a set of nodes $\mathfrak{S}$ called *root* nodes. For each root node $(q,Z) \in \mathfrak{S}$, we also maintain a set of nodes, denoted $S_{(q,Z)}$. The intuition is that root nodes are those that can be reached after pushing a symbol to the stack, whereas $S_{(q,Z)}$ will be the set of nodes that can be reached from (q,Z) with a well-nested run, i.e., starting with an empty stack and ending in an empty stack. This is to avoid storing the stack contents in our algorithm, which would be another source of infinity. Again, we use simulations to make the computation finite. So we fix a strongly finite simulation relation $\preceq$ for $\mathcal{A}$.

Our inductive rules for the control state reachability of pushdown timed automata are given in Table 1. Note that the internal rule is just the same as for timed automata above. The start rule not only starts the set of nodes computation but also the set of roots computation as described above. So the only interesting rules are the Push and Pop rules. The push rule says that when a push is encountered, then we must start exploring from a new root (i.e., context). So the only complicated rule is the Pop rule. Here the intuition is that if we see

$$\frac{}{\mathfrak{S} := \{(q_0, Z_0)\},\ S_{(q_0,Z_0)} := \{(q_0, Z_0)\}}\ \text{Start}$$

$$\frac{(q,Z) \in \mathfrak{S} \qquad (q',Z') \in S_{(q,Z)} \qquad q' \xrightarrow{g,\mathrm{nop},R} q'' \qquad Z'' = \overrightarrow{R(g \cap Z')} \neq \emptyset}{S_{(q,Z)} := S_{(q,Z)} \cup \{(q'',Z'')\},\ \text{unless } \exists (q''',Z''') \in S_{(q,Z)},\ Z'' \preceq_{q''} Z'''}\ \text{Internal}$$

$$\frac{(q,Z) \in \mathfrak{S} \qquad (q',Z') \in S_{(q,Z)} \qquad q' \xrightarrow{g,\mathrm{push}_a,R} q'' \qquad Z'' = \overrightarrow{R(g \cap Z')} \neq \emptyset}{\mathfrak{S} := \mathfrak{S} \cup \{(q'',Z'')\},\ S_{(q'',Z'')} = \{(q'',Z'')\},\ \text{unless } \exists (q''',Z''') \in \mathfrak{S},\ Z'' \sim_{q''} Z'''}\ \text{Push}$$

$$\frac{\begin{array}{llll}(q,Z) \in \mathfrak{S} & (q',Z') \in S_{(q,Z)} & q' \xrightarrow{g,\mathrm{push}_a,R} q'' & Z'' = \overrightarrow{R(g \cap Z')} \sim_{q''} Z_1 \\ (q'',Z_1) \in \mathfrak{S} & (q'_1,Z'_1) \in S_{(q'',Z_1)} & q'_1 \xrightarrow{g_1,\mathrm{pop}_a,R_1} q_2 & Z_2 = \overrightarrow{R_1(g_1 \cap Z'_1)} \neq \emptyset\end{array}}{S_{(q,Z)} := S_{(q,Z)} \cup \{(q_2,Z_2)\},\ \text{unless } \exists (q_2,Z'_2) \in S_{(q,Z)},\ Z_2 \preceq_{q_2} Z'_2}\ \text{Pop}$$

Table 1: Inductive rules for control state reachability of PDTA

a push at a node and from a root equivalent to the root created from it, (i.e., its context) we see a matching pop reaching a new node, then this push-pop context is complete, and we can add this new node to the set of reachable nodes. This is precisely the point where we *need* equivalence rather than simulation and this will be made clear in the proof of the theorem below.

Theorem 1. *Let $\mathfrak{S}^*$ and $(S_{(q,Z)})_{(q,Z)\in\mathfrak{S}^*}$ denote any tuple of sets obtained from the start rule and by repeatedly applying the rules in Table 1 till a fixed point is reached*[3]*. Note that we always have $(q_0, Z_0) \in \mathfrak{S}^*$. The following statements hold:*

1. *(finiteness) $\mathfrak{S}^*$ is finite and for each $(q, Z) \in \mathfrak{S}^*$, $S_{(q,Z)}$ is finite.*
2. *(completeness) For each $(q, Z) \in \mathfrak{S}^*$, if there exists a run $(q, v, \varepsilon) \xrightarrow{*} (q', v', \varepsilon)$ of $\mathcal{A}$ with $\{v\} \preceq_q Z$, then there exists $(q', Z') \in S_{(q,Z)}$ such that $\{v'\} \preceq_{q'} Z'$.*
3. *(soundness) For each $(q, Z) \in \mathfrak{S}^*$, $(q', Z') \in S_{(q,Z)}$ and $v' \in Z'$, there exists a run in PDTA from (q, v, ε) to (q', v'', ε) with $v \in Z$ and $v' \preceq_{q'} v''$.*

Proof. **1.** Note that only the Push rule creates new root nodes and the red condition states that a new root node is added only if there isn't already an equivalent node in $\mathfrak{S}^*$. Since the simulation relation is strongly finite, the set of roots $\mathfrak{S}^*$ must be finite. Also, before adding a node to some $S_{(q,Z)}$ with the internal rule or the pop rule, we check that the node is not subsumed by an existing one. Since the simulation relation is finite, this ensures that each set $S_{(q,Z)}$ is finite.

2. Let $(q, Z) \in \mathfrak{S}^*$ and assume that (q', v', ε) is reachable from some (q, v, ε) with $v \preceq_q Z$, i.e., there exists a run $(q, v, \varepsilon) = (q_1, v_1, \chi_1) \to \cdots \to (q_n, v_n, \chi_n) =$

[3] As before, there could be several such sets depending on the order in which the rules are applied.

$$\begin{array}{ccccccc}
(q,v,\varepsilon) & \xrightarrow{*} & (q_i,v_i,\varepsilon) & \xrightarrow{t} & (q_{i+1},v_{i+1},a) & \xrightarrow{*} & (q_{n-1},v_{n-1},a) \\
v \preceq_q Z & & v_i \preceq_{q_i} Z_i & & v_{i+1} \preceq_{q_{i+1}} Z_{i+1} & & v_{n-1} \preceq_{q_{n-1}} Z_{n-1} \\
(q,Z) \in \mathfrak{S} & & (q_i,Z_i) \in S_{(q,Z)} & & Z_{i+1} = \overrightarrow{R(g \cap Z_i)} & & (q_{n-1},Z_{n-1}) \in S_{(q_{i+1},Z'_{i+1})} \\
 & & & & Z_{i+1} \sim_{q_{i+1}} Z'_{i+1} & & \\
 & & & & (q_{i+1},Z'_{i+1}) \in \mathfrak{S} & &
\end{array}
\qquad
\begin{array}{ccc}
 & \xrightarrow{t_1} & (q_n,v_n,\epsilon) \\
 & & v_n \preceq_{q_n} Z_n \\
 & & Z_n = \overrightarrow{R_1(g_1 \cap Z_{n-1})} \\
 & & Z_n \preceq_{q_n} Z'_n \\
 & & (q_n,Z'_n) \in S_{(q,Z)}
\end{array}$$

Fig. 4: Construction for the completeness-push-pop last sub-case.

(q', v', ε). We will then show that $v_n \preceq_{q_n} Z'$ for some $(q_n, Z') \in S_{(q,Z)}$. The proof is by induction on n. Base case: For $n = 1$ we have $q' = q$ and $v' = v$. The result is obtained by taking $Z' = Z$. Notice that $(q, Z) \in S_{(q,Z)}$ follows immediately from the start rule if $q = q_0$, $Z = Z_0$ or from the push-create rule.

Let us then assume that the statement holds for runs of length at most $n-1$. Consider any run of the form $(q, v, \varepsilon) = (q_1, v_1, \chi_1) \to \cdots \to (q_n, v_n, \chi_n = \varepsilon)$ with $v \preceq_q Z$. Notice that its last transition $(q_{n-1}, v_{n-1}, \chi_{n-1}) \to (q_n, v_n, \chi_n)$ cannot be a push transition (in the PDTA) since $\chi_n = \varepsilon$. Hence, we have three subcases, depending on the last transition.

- Time elapse. $\chi_{n-1} = \chi_n = \varepsilon$, $q_{n-1} = q_n = q'$, $v_n = v_{n-1} + \delta$ for some $\delta \in \mathbb{R}^{\geq 0}$. Applying induction hypothesis, we have $v_{n-1} \preceq_{q'} Z'$ for some $(q', Z') \in S_{(q,Z)}$. Since zones are closed under time elapse, we get $Z' = \overrightarrow{Z'}$ and by definition of the simulation relation $v_n = v_{n-1} + \delta \preceq_{q'} \overrightarrow{Z'} = Z'$. This completes the case.
- Discrete internal transition. In this case $\chi_{n-1} = \chi_n = \varepsilon$, $t = q_{n-1} \xrightarrow{g,\mathrm{nop},R} q_n$, $v_{n-1} \models g$ and $v_n = [R]v_{n-1}$. Then applying induction hypothesis, there exists $(q_{n-1}, Z') \in S_{(q,Z)}$ such that $v_{n-1} \preceq_{q_{n-1}} Z'$. Now let $Z'' = \overrightarrow{R(g \cap Z')}$. From the definition of the simulation relation we get $v_n \preceq_{q_n} Z''$. Then, applying the Internal rule, there exists $(q_n, Z''') \in S_{(q,Z)}$ such that $Z'' \preceq_{q_n} Z'''$, with possibly $Z''' = Z''$. Hence, $v_n \preceq_{q_n} Z'' \preceq_{q_n} Z'''$, which completes the case.
- Pop transition. Then there exists $1 \leq i < n-1$ such that the run has the form: $(q_1, v_1, \varepsilon) \to \ldots \to (q_i, v_i, \chi_i = \varepsilon) \xrightarrow{\mathrm{push}_a} (q_{i+1}, v_{i+1}, \chi_{i+1} = a) \to \ldots \to (q_{n-1}, v_{n-1}, \chi_{n-1} = a) \xrightarrow{\mathrm{pop}_a} (q_n, v_n, \chi_n = \varepsilon)$, where the push and pop are matching transitions, i.e., $|\chi_j| \geq 1$ for all $i < j < n-1$ (see Figure 4). Then by induction hypothesis at i, we have

$$v_i \preceq_{q_i} Z_i \text{ for some } (q_i, Z_i) \in S_{(q,Z)}\,. \tag{1}$$

From the push transition we have

$$\exists t = q_i \xrightarrow{g,\mathrm{push}_a,R} q_{i+1} \in \Delta \text{ with } v_i \models g \text{ and } v_{i+1} = [R]v_i\,. \tag{2}$$

Let $Z_{i+1} = \overrightarrow{R(g \cap Z_i)}$. By definition of the simulation relation, we deduce from $v_i \preceq_{q_i} Z_i$ that $v_{i+1} \preceq_{q_{i+1}} Z_{i+1}$. We can apply the Push rule to obtain

$$(q_{i+1}, Z'_{i+1}) \in \mathfrak{S}^* \text{ for some } Z'_{i+1} \sim_{q_{i+1}} Z_{i+1} \tag{3}$$

possibly with $Z'_{i+1} = Z_{i+1}$ as a special case.
Further the segment of run $(q_{i+1}, v_{i+1}, a) \to \dots (q_{n-1}, v_{n-1}, a)$ in the PDTA never pops the symbol a (by choice, since otherwise the push and pop would not be matching). Hence we will also have the same sequence of transitions forming a run $(q_{i+1}, v_{i+1}, \varepsilon) \to \dots (q_{n-1}, v_{n-1}, \varepsilon)$. Using $v_{i+1} \preceq_{q_{i+1}} Z_{i+1} \sim_{q_{i+1}} Z'_{i+1}$, we deduce that $v_{i+1} \preceq_{q_{i+1}} Z'_{i+1}$. By induction hypothesis,

$$v_{n-1} \preceq_{q_{n-1}} Z_{n-1} \text{ for some } (q_{n-1}, Z_{n-1}) \in S_{(q_{i+1}, Z'_{i+1})}\,. \tag{4}$$

Finally, we have the pop transition

$$t_1 = q_{n-1} \xrightarrow{g_1, \mathrm{pop}_a, R_1} q_n \in \Delta \text{ with } v_{n-1} \models g_1 \text{ and } v_n = [R_1]v_{n-1}\,. \tag{5}$$

We let $Z_n = \overrightarrow{R_1(g_1 \cap Z_{n-1})}$. From $v_{n-1} \preceq_{q_{n-1}} Z_{n-1}$ and the definition of the simulation relation we obtain $v_n \preceq_{q_n} Z_n$. Then, combining all the above equations (1–5), and applying the Pop-rule we obtain some $(q_n, Z'_n) \in S_{(q,Z)}$ with $Z_n \preceq_{q_n} Z'_n$ (possibly $Z'_n = Z_n$). Finally we get $v_n \preceq_{q_n} Z_n \preceq_{q_n} Z'_n$. This completes the proof.

3. We will show that the following property is invariant by rule applications:

$$\begin{aligned} &\forall (q, Z) \in \mathfrak{S},\ \forall (q', Z') \in S_{(q,Z)}, \forall v' \in Z',\ \text{there is a run} \\ &\qquad (q, v, \varepsilon) \xrightarrow{*} (q', v'', \varepsilon) \text{ with } v \in Z \text{ and } v' \preceq_{q'} v'' \end{aligned} \tag{Inv}$$

The invariant holds initially, i.e., after application of the start rule. Indeed, in this case we have $\mathfrak{S} = \{(q_0, Z_0)\}$ and $S_{(q_0,Z_0)} = \{(q_0, Z_0)\}$. Hence $(q', Z') = (q, Z) = (q_0, Z_0)$ and for all $v \in Z_0$ we can choose the empty run $(q_0, v, \varepsilon) \xrightarrow{0} (q_0, v, \varepsilon)$.

We show below that (Inv) is preserved by application of an internal/push/pop rule. Therefore, the invariant still holds when reaching the fixed point, which proves the soundness. Let us write $\mathfrak{S}^-$ and $S^-_{(q,Z)}$ for the sets before the application of the rule and $\mathfrak{S}$ and $S_{(q,Z)}$ for the sets after the application of the rule.

Internal rule. Let $(q, Z) \in \mathfrak{S} = \mathfrak{S}^-$, $(q', Z') \in S_{(q,Z)}$ and $v' \in Z'$. If $(q', Z') \in S^-_{(q,Z)}$ then we get the result since (Inv) holds before applying the internal rule. Otherwise, there is some $(q_1, Z_1) \in S^-_{(q,Z)}$ and a transition $t = q_1 \xrightarrow{g, \mathrm{nop}, R} q'$ with $Z' = \overrightarrow{R(g \cap Z_1)}$.

By definition, there exists $v_1 \in Z_1$ such that $v_1 \models g$ and $v' = [R]v_1 + \delta$ for some $\delta \geq 0$. Hence we have a run $(q_1, v_1, \varepsilon) \xrightarrow{t} \xrightarrow{\delta} (q', v', \varepsilon)$. Since the invariant holds before the internal rule, there is a run $(q, v, \varepsilon) \xrightarrow{*} (q_1, v'_1, \varepsilon)$ with $v \in Z$ and $v_1 \preceq_{q_1} v'_1$. Now since $\preceq$ is a simulation we obtain that $(q_1, v'_1, \varepsilon) \xrightarrow{t} \xrightarrow{\delta} (q', v'', \varepsilon)$ with $v' \preceq_{q'} v''$ and we are done.

Push rule. Let $(q, Z) \in \mathfrak{S}$, $(q', Z') \in S_{(q,Z)}$ and $v' \in Z'$. If $(q, Z) \in \mathfrak{S}^-$ then we get the result since (Inv) holds before applying the Push rule. Otherwise, we must have $(q', Z') = (q, Z)$ and we can choose the empty run $(q, v', \varepsilon) \xrightarrow{0} (q, v', \varepsilon)$.

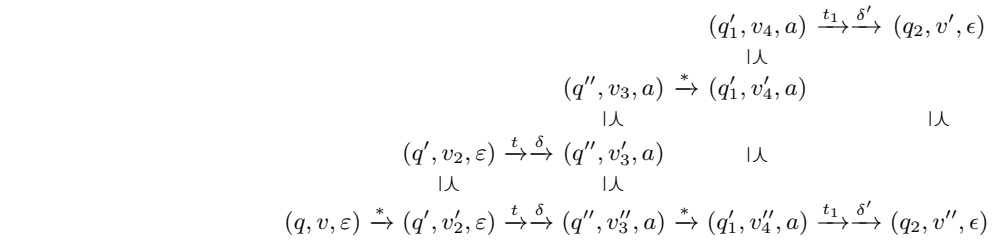


Fig. 5: Construction for the soundness.

Pop rule. Let $(q, Z) \in \mathfrak{S} = \mathfrak{S}^-$, $(q_2, Z_2) \in S_{(q,Z)}$ and $v' \in Z_2$. Again, if $(q_2, Z_2) \in S^-_{(q,Z)}$ then we get the result since (Inv) holds before applying the Pop rule. Otherwise, by definition of the pop rule we have:

1. some $(q', Z') \in S_{(q,Z)}$,
2. some push transition $t = q' \xrightarrow{g, \mathrm{push}_a, R} q''$,
3. some $(q'', Z_1) \in \mathfrak{S}$ with $Z_1 \sim_{q''} Z'' = \overrightarrow{R(g \cap Z')}$,
4. some $(q_1', Z_1') \in S_{(q'', Z_1)}$,
5. some pop transition $t_1 = q_1' \xrightarrow{g_1, \mathrm{pop}_a, R_1} q_2$,

with $Z_2 = \overrightarrow{R_1(g_1 \cap Z_1')}$. The construction below is illustrated in Figure 5.

Since $v' \in Z_2$, we get some $v_4 \in Z_1'$ such that $v_4 \models g_1$ and $v' = [R_1]v_4 + \delta'$ for some $\delta' \geq 0$. Hence we have a run $(q_1', v_4, a) \xrightarrow{t_1} \xrightarrow{\delta'} (q_2, v', \varepsilon)$.

Now, applying the invariant to $(q'', Z_1) \in \mathfrak{S}$, $(q_1', Z_1') \in S_{(q'', Z_1)}$ and $v_4 \in Z_1'$, we get a run $(q'', v_3, \varepsilon) \xrightarrow{*} (q_1', v_4', \varepsilon)$ with $v_3 \in Z_1$ and $v_4 \preceq_{q_1'} v_4'$. Hence, we also have a run $(q'', v_3, a) \xrightarrow{*} (q'', v_4', a)$.

Let $v_3' \in Z'' = \overrightarrow{R(g \cap Z')} \sim_{q''} Z_1$ with $v_3 \preceq_{q''} v_3'$. we get some $v_2 \in Z'$ such that $v_2 \models g$ and $v_3' = [R]v_2 + \delta$ for some $\delta \geq 0$. Hence we have a run $(q', v_2, \varepsilon) \xrightarrow{t} \xrightarrow{\delta} (q'', v_3', a)$.

Finally, we apply the invariant to $(q, Z) \in \mathfrak{S}$, $(q', Z') \in S_{(q,Z)}$ and $v_2 \in Z'$, we get a run $(q, v, \varepsilon) \xrightarrow{*} (q', v_2', \varepsilon)$ with $v \in Z$ and $v_2 \preceq_{q'} v_2'$.

By repeatedly applying the property of simulation $\preceq$, we may extend the run from (q', v_2', ε) with $(q', v_2', \varepsilon) \xrightarrow{t} \xrightarrow{\delta} (q'', v_3'', a) \xrightarrow{*} (q_1', v_4'', a) \xrightarrow{t_1} \xrightarrow{\delta'} (q_2, v'', \varepsilon)$ where $v_3 \preceq_{q''} v_3' \preceq_{q''} v_3''$ and $v_4 \preceq_{q_1'} v_4' \preceq_{q_1'} v_4''$. Finally $v' \preceq_{q_2} v''$. Therefore, the invariant holds after the pop rule. □

5 Algorithm for PDTA Reachability via Zones

In this section, we describe Algorithm 1 implementing the fixed point computation defined by the inductive rules in Table 1. We describe the structure of the algorithm and its main data-structures.

Notice first that the sets $\mathfrak{S}$ and $S_{(q,Z)}$ for $(q,Z) \in \mathfrak{S}$ can be alternatively represented as a single set of pairs of nodes:

$$\mathcal{S} = \{[(q,Z),(q',Z')] \mid (q,Z) \in \mathfrak{S} \text{ and } (q',Z') \in S_{(q,Z)}\}\,.$$

We can recover $\mathfrak{S}$ as the first projection of $\mathcal{S}$ and $S_{(q,Z)}$ as the second projection of $\mathcal{S}$ filtered by the first component being (q,Z). We use both notations below depending on which is more convenient. The start rule initializes $\mathcal{S}$ to $\{[(q_0,Z_0),(q_0,Z_0)]\}$.

Let us consider first the rule for internal transitions. For each already discovered pair of nodes $[(q,Z),(q',Z')] \in \mathcal{S}$ (or $(q',Z') \in S_{(q,Z)}$ with $(q,Z) \in \mathfrak{S}$), we have to consider each possible internal transition $q' \xrightarrow{g,\mathrm{nop},R} q''$ and check whether the node (q'',Z'') with $Z'' = \overrightarrow{R(g \cap Z')}$ should be added to $S_{(q,Z)}$ or is subsumed by an existing node. This is like a graph traversal. The set $\mathcal{S}$ stores the already discovered pairs of nodes, and we will use a ToDo (unordered) list to store the newly discovered nodes from which outgoing transitions should be considered. The ToDo list should also consist of pairs $[(q,Z),(q',Z')]$ so that when a new node (q'',Z'') is discovered by an internal transition from (q',Z') we know to which set $S_{(q,Z)}$ it should be added.

As we can see from Theorem 1-soundness, given $(q,Z) \in \mathfrak{S}$, the set $S_{(q,Z)}$ should consist of nodes reachable from (q,Z) via a well-nested run. Hence, when dealing with a pair $[(q,Z),(q',Z')] \in \mathcal{S}$ and we see a push transition $q' \xrightarrow{g,\mathrm{push}_a,R} q''$ with $Z'' = \overrightarrow{R(g \cap Z')}$, we should not try to add the pair (q'',Z'') to $S_{(q,Z)}$ since the corresponding run would not be well-nested. Instead, we should search for a matching pop transition which could be taken after a well-nested run starting from (q'',Z''). This is why the push rule adds the new root (q'',Z'') to $\mathfrak{S}$ (unless it is equivalent to an existing root). The pair of nodes $[(q'',Z''),(q'',Z'')]$ is newly discovered and added to the ToDo list for further exploration.

The push transition may be matched with several pop transitions (which could be already discovered or yet to be discovered by the algorithm). To avoid revisiting the push transition many times, it will be stored by the algorithm in an additional set $\mathcal{S}_{\mathrm{push}}$. More precisely, we will store in $\mathcal{S}_{\mathrm{push}}$ the tuple $[(q,Z),a,(q'',Z'')]$ meaning that the root node (q'',Z'') may be reached from the root node (q,Z) via a well-nested run reaching some (q',Z') followed by a transition pushing a onto the stack.

Finally, assume that, when dealing with a pair $[(q_1,Z_1),(q_1',Z_1')] \in \mathcal{S}$, we see a pop transition $q_1' \xrightarrow{g_1,\mathrm{pop}_a,R_1} q_2$ with $Z_2 = \overrightarrow{R_1(g_1 \cap Z_1')}$. We will check whether it can be matched with an already visited push transition, stored in the set $\mathcal{S}_{\mathrm{push}}$ as a pair $[(q,Z),a,(q'',Z'')]$ with $(q'',Z'') = (q_1,Z_1)$. If this is the case, the pop rule may be applied and the node (q_2,Z_2) added to $S_{(q,Z)}$ (unless it is subsumed by an existing node). The newly discovered pair of nodes $[(q,Z),(q_2,Z_2)]$ is also added to the ToDo list for further exploration. Once again, the pop transition may also be matched with push transitions that will be discovered later by the algorithm. To avoid revisiting the pop transition many times, we store the tuple $[(q_1,Z_1),a,(q_2,Z_2)]$ in a new set $\mathcal{S}_{\mathrm{pop}}$.

Data structures. We use a data structure TLM to store the triple of sets $(\mathcal{S}, \mathcal{S}_{\text{push}}, \mathcal{S}_{\text{pop}})$ and which is accessed with the following methods.

- TLM.create() creates the data structure with the three sets empty.
- TLM.add(q, Z, q', Z') adds $[(q, Z), (q', Z')]$ to $\mathcal{S}$.
- TLM.addPush(q, Z, a, q', Z') adds $[(q, Z), a, (q', Z')]$ to $\mathcal{S}_{\text{push}}$.
- TLM.addPop(q, Z, a, q', Z') adds $[(q, Z), a, (q', Z')]$ to $\mathcal{S}_{\text{pop}}$.
- TLM.isNewRoot(q, Z) returns $[\mathsf{false}, Z']$ if there exists some $(q, Z') \in \mathfrak{S}$ with $Z' \sim_q Z$, and returns $[\mathsf{true}, Z]$ otherwise.
- TLM.isNewNode(q, Z, q', Z') returns false if $\exists[(q, Z), (q', Z'')] \in \mathcal{S}$ with $Z' \preceq_{q'} Z''$, and returns true otherwise.
- TLM.isNewPop(q, Z, a, q', Z') returns false if $\exists[(q, Z), a, (q', Z'')] \in \mathcal{S}_{\text{pop}}$ with $Z' \preceq_{q'} Z''$, true otherwise.
- TLM.isNewPush(q, Z, a, q', Z') returns false if $[(q, Z), a, (q', Z')] \in \mathcal{S}_{\text{push}}$, and returns true otherwise.
- TLM.iterPop(q, Z, a) returns the list of (q', Z') with $[(q, Z), a, (q', Z')] \in \mathcal{S}_{\text{pop}}$.
- TLM.iterPush(a, q', Z') returns the list of (q, Z), s.t. $[(q, Z), a, (q', Z')] \in \mathcal{S}_{\text{push}}$.

Concretely, the data structure should store sets of nodes (q, Z) and be able to search or iterate through such sets. In order to make the algorithm slightly faster, we will segregate our sets of nodes, with the name of the state. We will use a hashmap in order to accomplish this task. See Figure 6 where the concrete data structure is depicted.

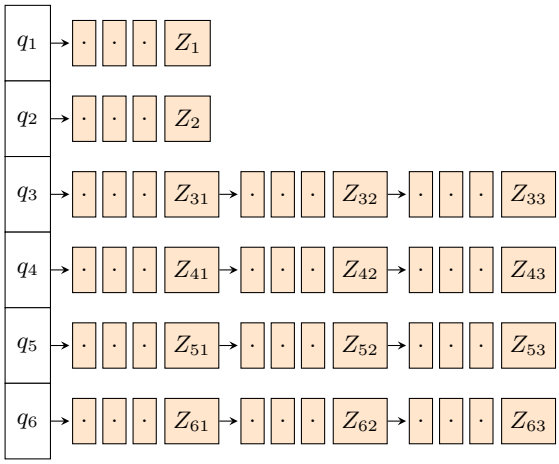
We will use a first level hashmap to store the set of roots $\mathfrak{S}$. To implement TLM.isNewNode(q, Z, q', Z'), we first search for (q, Z) in the first level map, then a pointer TLM$[q][Z][0]$ will lead to a second level hashmap for the set of nodes $S_{(q,Z)}$ and we search for (q', Z') in this second level map. See Figure 6(b).

To implement TLM.isNewPop(q, Z, a, q', Z') and TLM.iterPop(q, Z, a), we first search the root node (q, Z) in the first level map, then a pointer TLM$[q][Z][2]$ will lead to a second level hashmap storing the set of triples (a, q', Z') such that $[(q, Z), a, (q', Z')] \in \mathcal{S}_{\text{pop}}$. To speed up the access, this second level pop map is segregated first on the key a, then on the key q' to get the list of corresponding zones Z'. See Figure 6(c,d).

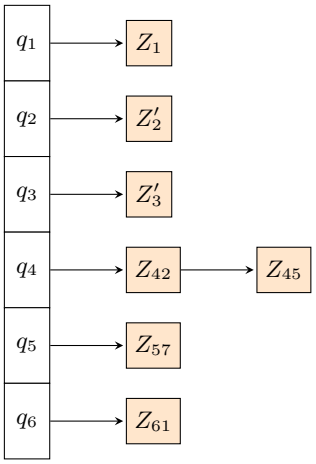
Finally, we also store the set $\mathcal{S}_{\text{push}}$ to implement TLM.isNewPush(q, Z, a, q', Z') and TLM.iterPush(a, q', Z'). Notice that $\mathcal{S}_{\text{push}}$ consists of triples $[(q, Z), a, (q', Z')]$ where both (q, Z) and (q', Z') are root nodes from $\mathfrak{S}$. Notice also that for the iteration we fix the second node (q', Z'). To get an efficient implementation, we first search the root node (q', Z') in the first level map, then a pointer TLM$[q'][Z'][1]$ will lead to a second level hashmap storing the set of triples (a, q, Z) such that $[(q, Z), a, (q', Z')] \in \mathcal{S}_{\text{push}}$. To speed up the access, this second level push map is segregated first on the key a, then on the key q to get the list of corresponding zones Z. See Figure 6(c,d).

Algorithm 1 PDTA Reachability Using Zones.

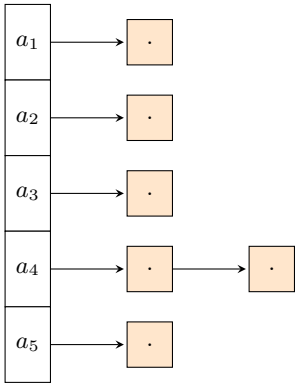
procedure PDTAReach
  TLM.create()
  TLM.add(q_0, Z_0, q_0, Z_0) ▷ Start Rule
  ToDo $= \{[(q_0, Z_0), (q_0, Z_0)]\}$
  while ToDo $\neq \emptyset$ **do**
    $[(q, Z), (q', Z')]$ = ToDo.get() ▷ $(q, Z) \in \mathfrak{S} \wedge (q', Z') \in S_{(q,Z)}$
    for $t = q' \xrightarrow{g,\mathrm{op},R} q''$ and $Z'' = \overrightarrow{R(g \cap Z')} \neq \emptyset$ **do**
      if op $=$ nop $\wedge$ TLM.isNewNode(q, Z, q'', Z'') **then**
        TLM.add(q, Z, q'', Z'') ▷ Internal Rule
        ToDo.add$([(q, Z), (q'', Z'')])$
      else if op $= \mathrm{push}_a$ **then**
        $[isNew, Z_1]$ = TLM.isNewRoot(q'', Z'')
        if $isNew ==$ true **then**
          TLM.add(q'', Z'', q'', Z'') ▷ Push Rule
          ToDo.add$([(q'', Z''), (q'', Z'')])$
        end if
        if TLM.isNewPush(q, Z, a, q'', Z_1) **then**
          TLM.addPush(q, Z, a, q'', Z_1)
          for (q_2, Z_2) in TLM.iterPop(q'', Z_1, a) **do**
            if TLM.isNewNode(q, Z, q_2, Z_2) **then**
              TLM.add(q, Z, q_2, Z_2) ▷ Pop Rule
              ToDo.add$([(q, Z), (q_2, Z_2)])$
            end if
          end for
        end if
      else if op $= \mathrm{pop}_a$ **then**
        if TLM.isNewPop(q, Z, a, q'', Z'') **then**
          TLM.addPop(q, Z, a, q'', Z'')
          for (q_3, Z_3) in TLM.iterPush(a, q, Z) **do**
            if TLM.isNewNode(q_3, Z_3, q'', Z'') **then**
              TLM.add(q_3, Z_3, q'', Z'') ▷ Pop Rule with $q = q_3, Z = Z_3$
              ToDo.add$([(q_3, Z_3), (q'', Z'')])$ ▷ $q_2 = q'', Z_2 = Z''$
            end if
          end for
        end if
      end if
    end for
  end while
end procedure



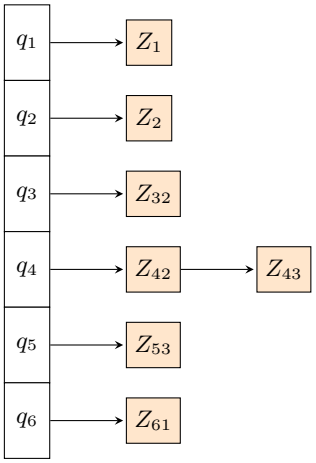
(a) First level map constructed using equivalence $\sim_q$ for controlling size. Keys will be state names, values will be lists of quadruplets, each of which has four pointers to second level maps, second level pushes maps, second level pops maps, and zones.



(b) Second level map corresponding to $S_{(q_1,Z_1)}$. Each first level map node will have its own second level map.



(c) Pushes/Pops map corresponding to root node (q_1, Z_1). Each pointer points to a different map where (q, Z) are stored.



(d) For pushes/pops map, this is a map corresponding to root node (q_1, Z_1), and symbol a_2 (say). The (q, Z) stored here is constructed using equivalence (pushes map), or using simulation (pops map).

Fig. 6: Two level map implementing the data structure TLM storing the sets $\mathcal{S}$, $\mathcal{S}_{\text{push}}$, $\mathcal{S}_{\text{pop}}$.

We now show correctness of Algorithm 1. Note that TLM encodes a triple of sets $(\mathcal{S}, \mathcal{S}_{\text{push}}, \mathcal{S}_{\text{pop}})$ defined by:

$$\begin{aligned}
\mathcal{S} &= \{[(q,Z),(q',Z')] \mid (q',Z') \in \mathsf{TLM}[q][Z][0]\} \\
\mathcal{S}_{\text{push}} &= \{[(q,Z),a,(q',Z')] \mid (a,q,Z) \in \mathsf{TLM}[q'][Z'][1]\} \\
\mathcal{S}_{\text{pop}} &= \{[(q,Z),a,(q',Z')] \mid (a,q',Z') \in \mathsf{TLM}[q][Z][2]\}
\end{aligned}$$

Recall also the correspondence explained at beginning of Section 5 between a set $\mathcal{S}$ of pairs of nodes, and the set of roots $\mathfrak{S}$ together with the sets of nodes $S_{(q,Z)}$ for $(q,Z) \in \mathfrak{S}$.

Theorem 2. *The set $\mathcal{S}$ encoded by* TLM *computed by Algorithm 1 is a fixed point obtained starting from the empty set by applying the inductive rules in Table 1. Therefore, Algorithm 1 terminates and is sound and complete for well-nested control state reachability of pushdown timed automata.*

Proof (sketch).

1. For termination, if we look at our algorithm, we can clearly see that before adding a pair of nodes to the ToDo list, we add the same pair to $\mathcal{S}$ with TLM.add, and before that, we always check whether the pair is already in our TLM or not (isNewNode or isNewRoot). Since the size of the TLM is always bounded because we check either the first level map or the second level map before adding, the outer while loop will be called only a finite number of times. From this we can conclude that the algorithm will terminate.
2. For soundness we have prove that any change to the TLM is equivalent to applying one of the rewrite rules to $(\mathcal{S}, \mathcal{S}_{\text{push}}, \mathcal{S}_{\text{pop}})$, which is already known to be sound from Theorem 1. The changes to the TLM occur in lines 3, 9, 14, 21, 31. Since line 3 simply adds $[(q_0,Z_0),(q_0,Z_0)]$ to $\mathcal{S}$, it simulates the start rule. For line 9, we can see that the pre conditions of internal rule Table 1 are met, with $(q,Z) \in \mathfrak{S}$, $(q',Z') \in S_{(q,Z)}$, the *if*-statement (just above the line) stating that there is an *nop* transition from q to q', and $Z'' \neq \phi$. Using all these we can see that indeed the operation can be performed. Similar arguments can be made for line 14, which simulates the push rule, and line numbers 21, 31 both for the pop-rule.
3. For completeness we have to prove that after termination of the algorithm, using $(\mathcal{S}, \mathcal{S}_{\text{push}}, \mathcal{S}_{\text{pop}})$ to encode TLM, we cannot use any of the rules mentioned in Table 1, to add anything extra to the TLM. Then from Theorem 1-completeness we can conclude. For the start rule we can simply say that it was definitely executed (Line 3), so it cannot be executed again. For the internal rule we argue that if it can be applied after termination, then it should have been applied during execution. Since all transitions are considered in the *for*-loop, and the conditions before line 9 checks all the preconditions of the internal rule, it is certainly the case that a node (q'',Z'') could not be added because either it was already added, or $(q'',Z''') \in S_{(q,Z)}, Z'' \preceq_{q''} Z'''$. The argument for the push rule is similar. For the pop rule to be applied

we argue that there must be a push transition and a pop transition satisfying the pre-conditions in the pop rule. Since both of these are already present for zones in the TLM, we say that they must have been added to $\mathcal{S}_{\text{push}}$ and $\mathcal{S}_{\text{pop}}$. We then concern ourselves with the order, arguing that if the push transition was discovered later the node either must already have been added (Line 21) or another node simulating the node must have been present in the TLM (Line 20). A similar argument is made in case the order of discovery is reversed.

For the full proof details, we refer the reader to the long version [6]. □

6 Experiments and Results

Implementation We build on the existing architecture of an open-source tool for analysis of timed automata, TChecker [19]. Our tool along with the benchmarks we used is available at `https://github.com/karthik-314/PDTAreachability` and more details can be found [6]. The input for our implementation are PDTA, rather than TA so we modify TChecker in order to run our experiments. While most of the TChecker file format will remain the same, the only place where we make a change to the syntax of the input, will be the edges. TChecker uses the following format, for its transitions,

```
edge:<Process>:<src>:<tgt>:<label>{
    do:<Reset1(x=0)> ; <Reset2(y=0)> :
    provided: <guard1(x==0)> && <guard2(y>=1)>}
```

The new format in order to incorporate the pushes and pops will be,

```
edge:<Process>:<src>:<tgt>:<label>{
    do:<Reset1(x=0)> ; <Reset2(y=0)> :
    provided: <guard1(x==0)> && <guard2(y>=1)>}
    [<push/pop>:<symbol>]
```

In case the operation is nop, the square brackets are left empty.

We have implemented two variants of Algorithm 1 for PDTA and we will compare these between each other and also with a region-based approach. More precisely, we consider the following 3 algorithms:

- **Simulation Based Approach** ($\preceq_{LU}$): Direct implementation of Algorithm 1.
- **Equivalence Based Approach** ($\sim_{LU}$): This is a variation of Algorithm 1, with two methods changed,
  - `TLM.isNewNode`(q, Z, q', Z'): Returns `false` if $\exists[(q, Z), (q', Z'')] \in \mathcal{S}$ with $Z' \sim_{q'} Z''$, and `true` otherwise.
  - `TLM.isNewPop`(q, Z, a, q', Z'): Returns `false` if $\exists[(q, Z), a, (q', Z'')] \in \mathcal{S}_{pop}$ with $Z' \sim_{q'} Z''$, and `true` otherwise.

  As mentioned in Section 4, if instead of simulation, we just use equivalence everywhere, we do obtain a correct algorithm for reachability in PDTA. Hence it is interesting to compare it with the above approach.

– **Region Based Implementation** (RB): A previous implementation [5], uses a region based approach in order to solve the non-emptiness problem in PDTA. We note two features of the algorithm. First, it uses a tree-automaton based approach for efficiency and correctness, but underlying it is the region (rather than zone) construction. Second, it works only with closed guards, while our approach works with closed and open guards.

We note the following important points regarding our implementation:

1. The $\preceq$ used in our implementation will be $\preceq_{LU}$ [8], without extrapolation and with global clock bounds.
2. The `ToDo` list used currently uses LIFO (stack) ordering for popping of elements. This corresponds to a DFS exploration of the zone-graph. But we can use other data structures for this purpose as well, e.g., changing it to FIFO would give us a BFS exploration etc.
3. Both the simulation based and equivalence based approach are tested on PDTA with empty and non-empty languages, but we have ensured that both of them return an answer only after the entire exploration has been completed. In other words, we do not stop the exploration when we reach a final state. This is to make fair comparisons, where we do not terminate because of being "lucky" in encountering the final state early. Of course in practice we would not do this. In contrast, we note that the RB approach is an on the fly approach which returns non-empty as soon as the final state turns out to be reachable.

All experiments are run on Intel-i5 10th Generation processor, with an 8GB RAM, with a timeout of 120 seconds.

Benchmarks. We used a total of 10 benchmarks in our experiments, but parameterized several of them in order to test the scalability and to give us more insight into performance comparisons. The benchmark and their parameterizations are explained in [6]. We highlight only some salient points here. The benchmark B_1 is the PDTA from Figure 1. $B_2(k)$ is directly adapted from Figure 3 with the constant $y \leq 1$ parametrized to $y \leq k$, and $k+1$ pops between q_0 and q_2. Note that q_3 is unreachable regardless of the value of k. Benchmarks B_3, B_4 are adapted from [5] with B_3 involving untiming of a stack age into normal clocks. B_5, B_6 involve significant interplay of push/pop edges and clocks and B_6, B_7 also have open guards. More details can be found in [6]. We also note that automata $B_1,\ B_3(3,4), B_5(k_1,k_2), B_8, B_9(k_1,k_2)$ accept a nonempty language, while the rest are empty. As described earlier this does not change the performance of the simulation and equivalence based approaches, but may significantly change the performance of the Region Based Approach.

Results Table 2 contains a selection of our experimental results; more can be found in [6]. From the table, we conclude first that the zone based approach is indeed faster than the Region Based Approach for all examples. Second, the simulation based approach runs faster than the equivalence based approach for all examples if the `ToDo` priority for removal remains the same. In fact, the performance of the simulation based approach depends mostly on the size of the

Benchmark	$\preceq_{LU}$ Time	$\preceq_{LU}$ # nodes	$\sim_{LU}$ Time	$\sim_{LU}$ # nodes	RB Time	RB # nodes
B_1	0.2	17	0.2	17	235.6	4100
$B_2(10)$	0.8	77	0.8	77	6835.8	30200
$B_2(100)$	20.0	5252	20.7	5252	T.O.	$\geq$154700
$B_3(4,3)$	0.2	6	0.2	6	1043.8	14300
$B_3(3,4)$	0.2	9	0.2	9	98.8	3400
B_4	0.2	8	0.1	8	0.3	17
$B_5(100,10)$	0.8	202	5.4	2212	OoM	OoM
$B_5(100,1000)$	0.7	202	3564.3	201202	OoM	OoM
$B_5(5000,100)$	23.2	10002	3429.3	1010102	OoM	OoM
$B_6(5,4,1000)$	0.3	30	611.8	30047	NA	NA
$B_6(5,4,10000)$	0.3	30	60271.9	300047	NA	NA
$B_6(501,500,100)$	38.2	3006	501.0	34799	NA	NA
B_7	112.4	4475	113.1	4475	NA	NA

Table 2: Results on the Benchmarks. Time recorded in ms, and timeout (T.O.) used is 120 seconds. OoM stands for Out of Memory kill. Results rounded up to 1 decimal. # nodes refers to the number of nodes in the zone/region graph explored. In case of timeout $\geq n$, refers to recorded number of nodes n before timeout occurred. NA in RB columns represents that the region based approach does not handle open guards in transitions (B_6, B_7 have open guards.)

PDTA, but the equivalence based approach is dependant on the constants used in guards as well, which is even more the case for the region based approach. Finally, our approach can easily handle closed and open guards.

Most of the timeouts that occurred during the experiments are due to Out of Memory (OoM) kills, especially for larger sized PDTAs. For smaller sized PDTA such as $B_2(100)$, the recorded number of nodes before timeout was 154700.

Regarding the performance, we would like to emphasize that B1, B2, B3, B4, B7 were designed to compare the Zone approach to the region (RB) approach. As a consequence these models are very simple and the number of explored nodes remains almost the same regardless of whether we use $\sim$ or $\preceq$ to prune, which reflects in the times/sizes not being too different. However, the other examples B5, B6 are more complex and have nodes that get pruned during exploration (both using $\sim$ and $\preceq$). Here we can see the clear improvement of $\preceq$ over $\sim$ both in terms of time taken and also of number of explored nodes.

7 Discussion and Future work

In this paper, we examined how an unbounded stack can be integrated seamlessly with zone-abstractions in timed automata. We would like to point out that two easy extensions of our work are possible. First, as remarked earlier, our algorithm checks for well-nested reachability, i.e., it requires to reach a final state with empty stack for acceptance. But we can generalize this to general control-state reachability by showing that a control state q is reachable in the PDTA (with

possibly a non-empty stack) iff some node (q, Z) is discovered by our algorithm and added to some $S_{(q',Z')}$ (and not just to $S_{(q_0,Z_0)}$ as in the well-nested case). While this idea is simple and requires only minor edits to the existing algorithm, the proof of correctness requires more work and we leave this for future work.

Secondly, we can handle the model with ages in stack as in [3,1] with an exponential blowup (thanks to [13]). However, an open question is whether this blowup can be avoided in practice. As noted earlier, there exist extensions [15,14] studied especially in the context of binary reachability, which are expressively strictly more powerful, for which decidability results are known. It would be interesting to see how we can extend the zone-based approach to those models.

Finally, it seems interesting to examine further the link to the liveness problem, possibly allowing us to transfer ideas and obtain faster implementations. Another possibility would be to use the extrapolation operator (rather than, or in addition to, simulation), which we have not considered in this work.

References

1. Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Jari Stenman. Dense-timed pushdown automata. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June*, pages 35–44, 2012.
2. S. Akshay, Paul Gastin, Vincent Jugé, and Shankara Narayanan Krishna. Timed systems through the lens of logic. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June*, pages 1–13, 2019.
3. S. Akshay, Paul Gastin, and Shankara Narayanan Krishna. Analyzing Timed Systems Using Tree Automata. *Logical Methods in Computer Science*, Volume 14, Issue 2, May 2018.
4. S. Akshay, Paul Gastin, Shankara Narayanan Krishna, and Sparsa Roychowdhury. Revisiting underapproximate reachability for multipushdown systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Ireland, Proceedings, Part I*, volume 12078 of *Lecture Notes in Computer Science*, pages 387–404. Springer, 2020.
5. S. Akshay, Paul Gastin, Shankara Narayanan Krishna, and Ilias Sarkar. Towards an efficient tree automata based technique for timed systems. In *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, pages 39:1–39:15, 2017.
6. S. Akshay, Paul Gastin, and Karthik R. Prakash. Fast zone-based algorithms for reachability in pushdown timed automata. *CoRR arXiv preprint:2105.13683*, `https://arxiv.org/abs/2105.13683`, 2021.
7. Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.
8. Gerd Behrmann, Patricia Bouyer, Kim Guldstrand Larsen, and Radek Pelánek. Lower and upper bounds in zone based abstractions of timed automata. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 312–326. Springer, 2004.

9. Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal—a tool suite for automatic verification of real-time systems. In *International Hybrid Systems Workshop*, pages 232–243. Springer, 1995.
10. Ahmed Bouajjani, Rachid Echahed, and Riadh Robbana. On the automatic verification of systems with continuous variables and unbounded discrete data structures. In *International Hybrid Systems Workshop*, pages 64–85. Springer, 1994.
11. Patricia Bouyer. Forward analysis of updatable timed automata. *Formal Methods Syst. Des.*, 24(3):281–320, 2004.
12. Patricia Bouyer, François Laroussinie, and Pierre-Alain Reynier. Diagonal constraints in timed automata: Forward analysis of timed systems. In *Formal Modeling and Analysis of Timed Systems, Third International Conference, FORMATS 2005, Uppsala, Sweden, September 26-28, 2005, Proceedings*, volume 3829 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 2005.
13. Lorenzo Clemente and Slawomir Lasota. Timed pushdown automata revisited. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, page 738–749, 2015.
14. Lorenzo Clemente and Slawomir Lasota. Reachability relations of timed pushdown automata. *J. Comput. Syst. Sci.*, 117:202–241, 2021.
15. Lorenzo Clemente, Slawomir Lasota, Ranko Lazic, and Filip Mazowiecki. Timed pushdown automata and branching vector addition systems. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017.
16. Zhe Dang. Pushdown timed automata: a binary reachability characterization and safety verification. *Theor. Comput. Sci.*, (1-3):93–121, 2003.
17. Paul Gastin, Sayan Mukherjee, and B. Srivathsan. Fast algorithms for handling diagonal constraints in timed automata. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, USA, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 41–59. Springer, 2019.
18. Frédéric Herbreteau, Dileep Kini, B. Srivathsan, and Igor Walukiewicz. Using non-convex approximations for efficient analysis of timed automata. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2011, December 12-14, 2011, Mumbai, India*, volume 13 of *LIPIcs*, pages 78–89. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011.
19. Frédéric Herbreteau and Gerald Point. Tchecker. Available at `https://github.com/fredher/tchecker`.
20. Frédéric Herbreteau, B. Srivathsan, Thanh-Tung Tran, and Igor Walukiewicz. Why liveness for timed automata is hard, and what we can do about it. *ACM Trans. Comput. Log.*, 21(3):17:1–17:28, 2020.
21. Frédéric Herbreteau, B. Srivathsan, and Igor Walukiewicz. Better abstractions for timed automata. In *Proceedings of the 27th Annual IEEE Symposium on Logic in Computer Science, LICS 2012, Dubrovnik, Croatia, June 25-28, 2012*, pages 375–384. IEEE Computer Society, 2012.
22. Alfons Laarman, Mads Chr. Olesen, Andreas Engelbredt Dalsgaard, Kim Guldstrand Larsen, and Jaco van de Pol. Multi-core emptiness checking of timed büchi automata using inclusion abstraction. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 968–983. Springer, 2013.
23. Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1(1-2):134–152, 1997.
24. Stavros Tripakis. Checking timed büchi automata emptiness on simulation graphs. *ACM Trans. Comput. Log.*, 10(3):15:1–15:19, 2009.