Layered and Collecting NDFS with Subsumption for Parametric Timed Automata

Hoang Gia Nguyen, Laure Petrucci
LIPN, Université Paris 13, CNRS UMR 7030
Villetaneuse, France

Jaco van de Pol
Formal Methods and Tools, University of Twente
Enschede, Netherlands

***Abstract*—This paper studies the analysis and parameter synthesis problems for Parametric Timed Automata (PTA) with properties in Linear-time Temporal Logic (LTL). It introduces a series of variations of Nested Depth-First Search (NDFS).**

We first study the LTL model checking problem for PTA. Based on a careful analysis of parametric zones, we introduce a new *layered NDFS approach* to LTL model checking. We integrate this with several techniques to prune the search space. In particular, we apply subsumption abstraction to PTA for the first time. We also propose heuristics on the search order to improve the performance.

Next, we study parameter synthesis. To this end, this new layered approach and subsumption are added to a Collecting NDFS scheme. We implemented all algorithms in the IMITATOR tool and analyse their efficiency in a number of experiments.

1. Introduction

Model-checking is a common approach to formally verify that a system, described by its model, satisfies some requirement, expressed by a property in temporal logic. Timed Automata (TA) [1] are widely and successfully used to verify real-time systems. However, at the design phase, when the timing constants are not yet known, the system is incompletely specified. Parametric Timed Automata (PTAs) [2] extend Timed Automata by allowing timing parameters in the model, where the timing constants are not precisely known.

In such a setting, model-checking problems become synthesis problems, *i.e.* instead of simply evaluating the truth value for the satisfaction of the formula, the set of parameters values for which the formula holds is computed.

The means for such a computation consists in exploring the Parametric Zone Graph (PZG), an extension of the Zone Graph for TAs that includes parameters. Even though the PZG may be infinite, and most problems for PTAs are undecidable [3], semi-algorithms such as EF-synthesis have proved successful for model-checking. Such algorithms either return a correct answer, or do not terminate.

In [5], several strategies are studied to exhibit efficient exploration orders for Breadth-First Search (BFS), that address parameter synthesis for reachability properties.

This work was supported by the PHC Van Gogh project PAMPAS.

However, reachability properties are often insufficient, and it is necessary to model-check liveness properties. These are generally described as a Linear Temporal Logic (LTL) formula. The LTL model-checking approach boils down to a Büchi emptiness problem. This is achieved by the search of accepting cycles in the state space, typically in a Depth-First Search fashion.

Contribution. This paper presents DFS exploration to find accepting cycles for Parametric Time Automata, using several reductions of the state space and smart exploration orders.

First the subsumption of [18] for Timed Automata, that reduces the explored state space while preserving cycle detection, is extended to the parametric case.

Furthermore, PTAs enjoy properties on the projection of zones on the parameters which can be used to stop the exploration of a branch at an early stage or check for cycles in layers of the graph. Note that such a layered approach could also be used for other types of models which exhibit some progress measure, as is done with the sweep-line exploration [13].

Finally, DFS for LTL model-checking usually exits as soon as an accepting cycle is found. However, in the case of parameter synthesis, it is desirable to obtain all parameters for which such a cycle exists. As they may lie in different branches, the exploration must be continued and the parameters zones collected all along the process. This results in a collecting algorithm.

Related work. LTL parameter synthesis for PTA was addressed in [9], where parameter valuations are restricted to bounded integers. We make no such restriction, giving up decidability of the problem. Nevertheless, our semi-algorithm either gives an exact result or does not terminate. Although extrapolation is used in the model checking algorithm of [9] as a form of abstraction, there was no early pruning, subsumption, or layered verification. Therefore branches had to be explored in depth, which was feasible only due to the fact that the system is finite after extrapolation. The pruning and layering introduced in our algorithm sometimes avoids infinite branches, and provides speedup for the finite case, as demonstrated in the experiments.

Our basic algorithm extends NDFS with subsumption from [18] from the setting of TA to Parametric TA. Due to the nature of the parametric zones we identified additional

pruning opportunities. Another extension of NDFS with subsumption for TA was studied in [15]. Those authors proved that LTL model checking for TA is inherently harder than reachability checking for TA. They also proposed to search accepting cycles in the subsumed state space first. If there are no such cycles, the original system is correct. Otherwise, the Strongly Connected Components that contain accepting cycles must be further refined, since subsumption may introduce accepting cycles. Our layered approach also tends to find abstract cycles first, but it is fully integrated in the NDFS procedure.

Outline. We first recall in Sec. 2 the basic definition of Parametric Timed Büchi Automata and their semantics, as well as the Parametric Zone Graph which allows for model-checking such models. Then, Sec. 3 shows that the subsumption introduced in [18] is still valid in the parametric case. Therefore, we can provide an extension of the *ndfs* algorithm with subsumption for PTBA in Sec. 4. The parameter synthesis requires obtaining all possible constraints on accepting cycles, as described by the collecting algorithm in Sec. 5 together with some pruning optimisations. The experimental evaluation of these algorithms is detailed in Sec. 6. Finally, Sec. 7 concludes and provides insight on future work.

2. Preliminaries: Parametric Timed Büchi Automata

In this section, we recall the formalism of Parametric Timed Büchi Automata (PTBA) and its semantics.

2.1. Clocks, parameters and constraints

We first recall the basic concepts of clocks, parameters and constraints. Let $X = \{x_1, \ldots, x_H\}$ be a set of *clocks*, *i.e.* real-valued variables that evolve at the same rate. A clock valuation w is a function $w : X \to \mathbb{R}_+$. We denote by $X = 0$ the conjunction of equalities that assigns 0 to all clocks in X. We also use a special zero-clock x_0, always equal to 0.

Let $P = \{p_1, \ldots, p_M\}$ be a set of *parameters*, *i.e.* unknown constants. A *parameter valuation* v is a function $v : P \to \mathbb{Q}_+$. We will often identify a valuation v with the *point* $(v(p_1), \ldots, v(p_M))$.

In the following, let *xplt* denote a linear term over $X \cup P$ of the form $\sum_{1 \leq i \leq H} \gamma_i x_i + \sum_{1 \leq j \leq M} \beta_j p_j + d$, with $x_i \in X$, $p_j \in P$, and $\gamma_i, \beta_j, d \in \mathbb{Z}$. Let *plt* denote a parametric linear term over P, that is a linear term without clocks ($\gamma_i = 0$ for all i).

The synthesis of parameters leads to expressing *constraints* on their values in order to guarantee that the model satisfies the expected properties.

Definition 1 (Constraints on clocks and timing parameters). A *constraint* over $X \cup P$ is a conjunction of inequalities of the form $xplt \bowtie xplt'$, where $xplt$ and $xplt'$ are two linear terms over $X \cup P$ and $\bowtie \in \{<, \leq, \geq, >\}$,

Given a constraint C and a parameter valuation v, $v(C)$ denotes the clock constraint over X obtained by replacing each parameter p in C with $v(p)$. Likewise, given a clock valuation w, $w(v(C))$ denotes the closed expression obtained by replacing each clock x in $v(C)$ with $w(x)$. We denote by $w|v$ the valuation over $X \cup P$ such that for all clocks x, $w|v(x) = w(x)$ and for all timing parameters p, $w|v(p) = v(p)$.

A valuation $w|v$ satisfies a constraint C, denoted $w|v \models C$, if the expression obtained by replacing in C each clock and timing parameter by its valuation as in $w|v$ evaluates to `true`.

Given two constraints C_1 and C_2, we write $C_1 \subseteq C_2$ whenever, for any v, w, $w|v \models C_1$ implies $w|v \models C_2$. Constraint *True* abbreviates $0 \leq 0$, so it holds for all clock and parameter valuations.

Definition 2 (Zones and guards). A *parametric zone* Z is a constraint whose linear conjuncts can be written in the form $x_i - x_j \bowtie plt$, where $x_i, x_j \in X \cup \{x_0\}$. A *parametric guard* g is a zone such that each of its linear conjuncts can be written in the form $x_i \bowtie plt$.

The *time elapsing* of a zone Z, denoted by $Z^{\nearrow}$, is the constraint over $X \cup P$ obtained from Z by delaying all clocks by any arbitrary amount of time. That is, $Z^{\nearrow} = \{(w', v) \mid \exists w.\, w|v \models Z \wedge \forall x \in X : w'(x) = w(x) + d, d \in \mathbb{R}_+\}$.

Given $R \subseteq X$, the *reset* of Z, denoted by $[Z]_R$, is the constraint obtained from Z by resetting the clocks in R to 0, and keeping the other clocks and the parameters unchanged.

The projection of Z onto P is denoted by $Z{\downarrow_P}$. It can be defined as $\exists x_1 \cdots \exists x_H . Z$, where $X = \{x_1, \ldots, x_H\}$.

2.2. Parametric Timed Büchi Automata

We now provide the basic definitions of Parametric Timed Büchi Automata and their semantics.

Definition 3 (Parametric Timed Büchi Automaton). A *Parametric Timed Büchi Automaton* (PTBA) is a tuple $\mathcal{B} = (\Sigma, L, l_0, F, X, P, I, E)$, where:

- Σ is a finite set of *actions*,
- L is a finite set of *locations*,
- $l_0 \in L$ is the *initial location*,
- $F \subseteq L$ is the set of *accepting locations*,
- X is a set of *clocks*,
- P is a set of *parameters*,
- I is the *invariant* function, assigning to every $l \in L$ a guard $I(l)$, and
- E is a set of *edges* (l, g, a, R, l') where $l, l' \in L$ are the source and target locations, g is the transition *guard*, $a \in \Sigma$ is the *action*, and $R \subseteq X$ is a set of clocks to be *reset*.

The actions, which can be used as synchronisation labels in networks of PTBAs, play no role in our algorithms. However, since actions provide convenient names for edges in examples, we kept them in the definition.

Example 1. Fig. 1 shows a PTBA, where only location l_1 is accepting. It has two clocks x and y, and three

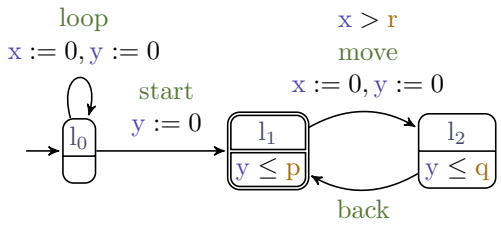

Figure 1. A Parametric Timed Büchi Automaton

parameters p, q and r used in the guard of action move and in the invariants of locations l_1 and l_2.

The *concrete semantics* defines a transition relation between concrete states (ℓ, x), where ℓ is a location and x a clock valuation. For the definition of the concrete semantics of a PT(B)A, we refer to *e.g.* [16].

This paper focuses on the symbolic semantics instead. Let us now recall the symbolic semantics of a PTBA, following *e.g.* [5], which we will use throughout this paper.

Definition 4 (Symbolic state). A *symbolic state* of a PTBA is a pair (l, Z) where $l \in L$ is a location, and Z its associated zone.

Definition 5 (Symbolic semantics). Given $\mathcal{B} = (\Sigma, L, l_0, F, X, P, I, E)$, its symbolic semantics is the *parametric zone graph* $\mathcal{PZG}(\mathcal{B}) = (\mathbf{S}, \mathbf{s}_0, \Rightarrow)$, with

- $\mathbf{S} = \{(l, Z) \mid Z \subseteq I(l)\}$,
- $\mathbf{s}_0 = \big(l_0, (\bigwedge_{1 \le i \le H} x_i = 0)^{\nearrow} \wedge I(l_0)\big)$, and
- $(l, Z) \overset{e}{\Rightarrow} (l', Z')$ if $e = (l, g, a, R, l') \in E$ and $Z' = \big([(Z \wedge g)]_R \wedge I(l')\big)^{\nearrow} \wedge I(l')$ with Z' non-empty.

We write $(l, Z) \Rightarrow (l', Z')$, if there exists an $e \in E$, s.t. $(l, Z) \overset{e}{\Rightarrow} (l', Z')$.

The constraint of the target symbolic state is obtained by first intersecting the constraint of the source symbolic state with the outgoing guard, then resetting the clocks of the transition, making sure the target invariant is satisfied, letting time elapse, and finally intersecting it with the invariant of the target location.

Example 2. Fig. 2 displays the PZG of the PTBA in Fig. 1.

Definition 6 (Symbolic run of a PTA). An *(infinite) symbolic run* of a PTA is an infinite sequence of symbolic states starting from the initial symbolic state, $\mathbf{s}_0 \Rightarrow \mathbf{s}_1 \Rightarrow \mathbf{s}_2 \Rightarrow \cdots$, such that for all $0 \le i$, $\mathbf{s}_i \Rightarrow \mathbf{s}_{i+1}$.

Definition 7 (Accepting states and runs). A *state* $\mathbf{s} \in \mathbf{S}$ is *accepting* (denoted by $\mathbf{s} \in F$), when $\mathbf{s} = (l, Z_\mathbf{s})$ and $l \in F$.

An *accepting run* in $\mathcal{PZG}(\mathcal{B})$ is a symbolic run $\pi = \mathbf{s}_0 \Rightarrow \mathbf{s}_1 \Rightarrow \mathbf{s}_2 \ldots$, for which there exist an infinite number of indices i s.t. $\mathbf{s}_i \in F$.

With $\Rightarrow^*$ we denote the reflexive, transitive closure of $\Rightarrow$. Let $\mathcal{R}(\mathcal{PZG}(\mathcal{B})) = \{\mathbf{s} \mid \mathbf{s}_0 \Rightarrow^* \mathbf{s}\}$ denote the set of *reachable states* in $\mathcal{PZG}(\mathcal{B})$.

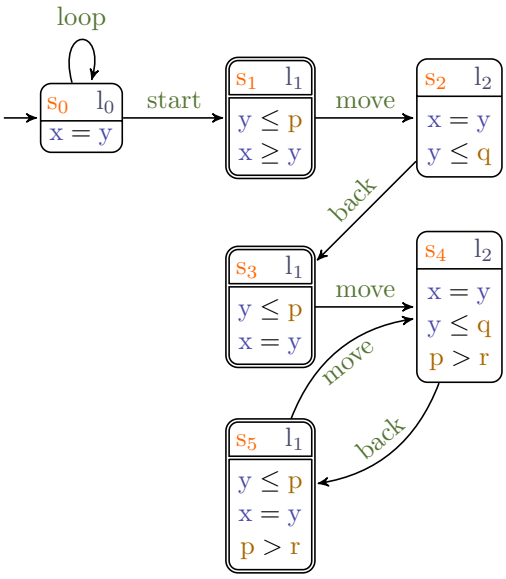

Figure 2. A Parametric Zone Graph.

If a symbolic run has a bounded time, then it is Zeno, *i.e.* an infinite number of actions can take place in a finite time. This is considered as a modelling artefact as the behaviour it models cannot occur in any real system. These can be detected using model transformations such as in [6]. Therefore, w.l.o.g., we assume PTBAs are non-Zeno.

Example 3. The PTBA of Fig. 1, with its PZG in Fig. 2, has several infinite runs in the language $\text{loop}^\omega \cup [\text{p} \ge \text{r} \implies \text{loop}^*.\text{start}.(\text{move}.\text{back})^\omega]$. Note that the infinite run using only action loop is not accepting but can be Zeno while the other ones are accepting since they visit location l_1 infinitely often but are non-Zeno as long as $\text{r} > 0$, since clock x must be strictly larger than r for action move to occur.

Definition 8 (A PTBA's language and the emptiness problem). The *language* accepted by $\mathcal{B}$, denoted $\mathcal{L}(\mathcal{B})$, is defined as the set of non-Zeno accepting runs. The language *emptiness problem* for $\mathcal{B}$ is to check whether $\mathcal{L}(\mathcal{B}) = \emptyset$.

3. Subsumption in the Parametric Zone Graph

Note that, similar to the case in timed automata, the parametric zone graph may be infinite. In (plain) timed automata, extrapolations are used to obtain finite zone graphs. For instance, k- or lu-extrapolation uses the extremal constant values appearing as lower- or upper-bounds in clock constraints of the TA, in order to identify symbolic zones that cannot be distinguished. It is not obvious how extrapolation carries over in case clock constraints have parameters. Given the undecidability of many problems of PTA [3], for example EF-emptyness, we are forced to accept that parametric zone graphs can be infinite. As a consequence, we will deal

with semi-algorithms, which either return correct answers, or diverge.

The strategy in this paper will be to identify techniques that avoid some infinite computations, by either reducing the state space, or by manipulating the search order, in a way that bugs are revealed earlier, before the algorithm would diverge.

3.1. Subsumption abstraction

We now discuss how to carry over subsumption abstraction from timed automata to the parametric case. For timed automata, subsumption was introduced in [12] and studied in [18] in the context of LTL model checking. The idea is that a symbolic state may be replaced by a "larger" one, without losing behaviour.

Definition 9 (Subsumption $\sqsubseteq$). A state $\mathbf{s} = (l, Z) \in \mathbf{S}$ is *subsumed* by another $\mathbf{s}' = (l', Z')$, denoted $\mathbf{s} \sqsubseteq \mathbf{s}'$, when $l = l'$ and $Z \subseteq Z'$.

Definition 10 (Subsumption Abstraction). An *abstraction* over the Parametric Zone Graph $\mathcal{PZG}(\mathcal{B}) = (\mathbf{S}, \mathbf{s}_0, \Rightarrow)$ is a total mapping $\alpha : \mathbf{S} \to \mathbf{S}$ s.t. for all reachable symbolic states $\mathbf{s}$, we have $\mathbf{s} \sqsubseteq \alpha(\mathbf{s})$.

Definition 11 (Induced PZG). An abstraction α over the Parametric Zone Graph $\mathcal{PZG}(\mathcal{B}) = (\mathbf{S}, \mathbf{s}_0, \Rightarrow)$ induces an abstracted Parametric Zone Graph $\mathcal{PZG}_\alpha(\mathcal{B}) = (\mathbf{S}_\alpha, \alpha(\mathbf{s}_0), \Rightarrow_\alpha)$, where:

- $\mathbf{S}_\alpha = \{\alpha(\mathbf{s}) \mid \mathbf{s} \in \mathbf{S}\}$ is the set of states, s.t. $\mathbf{S}_\alpha \subseteq \mathbf{S}$,
- $\alpha(\mathbf{s}_0)$ is the initial state, and
- the transition relation is: $\mathbf{s} \Rightarrow_\alpha \mathbf{s}'$ iff there exists $\mathbf{s}''$ s.t. $\mathbf{s} \Rightarrow \mathbf{s}''$ and $\mathbf{s}' = \alpha(\mathbf{s}'')$.

Note that if the image of the abstraction is finite, the abstract PZG is finite as well. However, this is not always the case. The following lemma shows that indeed abstraction doesn't lose behaviour.

Proposition 1 ($\sqsubseteq$ is a simulation relation). If $\mathbf{s}_1 \sqsubseteq \mathbf{s}_2$ and $\mathbf{s}_1 \Rightarrow \mathbf{s}_1'$ then there exists $\mathbf{s}_2'$ s.t. $\mathbf{s}_2 \Rightarrow \mathbf{s}_2'$ and $\mathbf{s}_1' \sqsubseteq \mathbf{s}_2'$.

Proof 1. By the definition of $\sqsubseteq$, and the fact that the symbolic transition relation $\Rightarrow$ is monotone w.r.t. $\subseteq$ of zones.

Note that in practice, the subsumption abstraction is defined only over the reachable state space, and it might introduce extra behaviour that the unabstracted system cannot simulate. Typically α is constructed on-the-fly, *i.e.* during analysis, by only abstracting to states that have already been found to be reachable. This makes its performance depend heavily on the search order. In particular, finding "large" states as early as possible can make the abstraction coarser [11].

To emphasize that this graph can be generated *on-the-fly*, we use the notation NEXT-STATE($\mathbf{s}$), which returns the set of successor states for $\mathbf{s}$: $\{\mathbf{s}' \in \mathcal{S} \mid \mathbf{s} \Rightarrow \mathbf{s}'\}$.

3.2. Preserving Accepting Runs with Subsumption

Recall that an accepting run from $\mathbf{s}$ is an infinite run $\mathbf{s} = \mathbf{s}_0 \Rightarrow \mathbf{s}_1 \Rightarrow \cdots$ that hits an accepting location infinitely often.

Proposition 2. If there exists an accepting run from $\mathbf{s}$ and $\mathbf{s} \sqsubseteq \mathbf{s}'$, then there exists an accepting run from $\mathbf{s}'$.

Proof 2. By Prop. 1 the infinite run from $\mathbf{s}$, $\mathbf{s}_1 \Rightarrow \mathbf{s}_2 \Rightarrow \cdots$, can be simulated by an abstract infinite run from $\mathbf{s}'$, $\mathbf{s}_1' \Rightarrow \mathbf{s}_2' \Rightarrow \cdots$, where each $\mathbf{s}_i \sqsubseteq \mathbf{s}_i'$. By Def. 9, the new run passes through the exact same locations, so it is still accepting.

Prop. 2 shows that a subsumption abstraction preserves Büchi emptiness in one direction. Unfortunately, $\mathcal{PZG}_\alpha(\mathcal{B})$ may introduce accepting runs which were not present in $\mathcal{PZG}(\mathcal{B})$. This is already the case for (plain) timed automata [18]. This can be illustrated by their example which is prensented in Fig. 3. The figure visualises $\mathcal{PZG}_\alpha(\mathcal{B})$ by drawing subsumed states inside subsuming states (*e.g.* $s_3 \sqsubseteq s_1$).

Still, subsumption implies a property on paths that we can use. We adapt the results from [18] to a setting where $\mathcal{PZG}$ might be infinite. In subsequent sections, we exploit these properties to improve algorithms that implement the PTBA emptiness check.

Proposition 3. If $\mathbf{s} \Rightarrow^* \mathbf{s}'$, $\mathbf{s} \sqsubseteq \mathbf{s}'$ and $\mathbf{s} \in F$, then $\mathcal{PZG}(\mathcal{B})$ has an accepting run.

Proof 3. Let $\mathbf{s} \sqsubseteq \mathbf{s}'$ and $\mathbf{s} \in F$, then by Def. 9, $\mathbf{s}' \in F$. Let $\mathbf{s} \Rightarrow^* \mathbf{s}'$, then by Prop. 1 we can simulate this same trace from $\mathbf{s}'$, leading to some $\mathbf{s}''$ s.t. $\mathbf{s}' \Rightarrow^* \mathbf{s}''$ and $\mathbf{s}' \sqsubseteq \mathbf{s}''$ and $\mathbf{s}'' \in F$. This process can be repeated to obtain an infinite accepting run.

We finish this section with an observation for the special case of a finite abstraction $\mathcal{PZG}_\alpha(\mathcal{B})$:

Proposition 4. If $\mathcal{PZG}_\alpha(\mathcal{B})$ is finite and does not contain an accepting cycle, then $\mathcal{PZG}(\mathcal{B})$ does not contain an accepting run, hence $\mathcal{L}(\mathcal{B}) = \emptyset$.

Proof 4. Let $\mathcal{PZG}(\mathcal{B})$ contain an accepting run from $\mathbf{s}_0$. Then $\mathcal{PZG}_\alpha(\mathcal{B})$ contains an accepting run as well from its initial state $\alpha(\mathbf{s}_0)$, by Prop. 2. Since $\mathcal{PZG}_\alpha(\mathcal{B})$ is finite and the accepting run visits infinitely many accepting states, some accepting state, say $\mathbf{s} \in F$ is visited infinitely often. Then we can construct an accepting cycle $\alpha(\mathbf{s}_0) \Rightarrow^* \mathbf{s} \Rightarrow^* \mathbf{s}$.

4. Parametric Timed Nested Depth-First Search with Subsumption

This section extends the NDFS algorithm with subsumption for TA [18] to PTA and introduces early checks and cuts to optimise the search. The algorithm detects accepting cycles, the absence of which implies Büchi emptiness. It is correct for the case of finite graphs.

Figure 3. Taken from [18]. Consider the TA obtained from the PBTA in Fig. 1, by setting $p = q = r = 2$. The symbolic state space $\mathcal{PZG}(\mathcal{B})$ of this TA, with $\ell_1 \in F$, contains 4 states (shown on the left): $\mathbf{s}_0$, $\mathbf{s}_1 = (\ell_1, Z_1)$, $\mathbf{s}_2 = (\ell_2, Z_2)$ and $\mathbf{s}_3 = (\ell_1, Z_3)$. Note that there is no accepting cycle. The graphical representation of the zones Z_1–Z_3 (right) reveals that $Z_3 \subseteq Z_1$ and hence $\mathbf{s}_3 \sqsubseteq \mathbf{s}_1$. As $\mathbf{s}_3 \sqsubseteq \mathbf{s}_1$ and both are reachable, the subsumption abstraction might map $\alpha(\mathbf{s}_3) = \mathbf{s}_1$, introducing a cycle $\mathbf{s}_1 \Rightarrow \mathbf{s}_2 \Rightarrow \mathbf{s}_1$ in $\mathcal{PZG}_\alpha(\mathcal{B})$.

4.1. NDFS with subsumption for PTA

In the following, with *soundness*, we mean that when NDFS reports a cycle, indeed an accepting cycle exists in the graph, while completeness indicates that NDFS always reports an accepting cycle if the graph contains one.

Alg. 1 Classical NDFS

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

The classical NDFS algorithm shown in Alg. 1 consists of an outer DFS (*dfsBlue*) that sorts accepting states $\mathbf{s}$ in DFS *post-order* ([10]) and an inner DFS (*dfsRed*) that searches for cycles over each $\mathbf{s}$, called the *seed*. States are maintained in 3 colour sets:

1) *Blue*, states fully *explored* by *dfsBlue*,
2) *Cyan*, states on the stack of *dfsBlue*, so they are on the path from the initial state to the current state, and
3) *Red*, visited by *dfsRed*.

The goal of the blue search is to visit all states, and launch a red search on all accepting states in post-order. The goal of the red search is to detect cycles on these accepting states, by finding edges to cyan states. These would close cycles early, at l.17 [19].

The NDFS-algorithm with subsumption from [18] for TA is presented in Alg. 2. Here the parts introduced by subsumption are highlighted in light blue. We now show why the same algorithm is valid for PTA as well. The part highlighted in yellow is new and specific for PTA, and will be explained in the next subsection.

As for TA, subsumption can be used to prune both searches, but we should be careful, since $\mathcal{PZG}_\sqsubseteq(\mathcal{B})$ may introduce additional cycles (Fig. 3). To express subsumption checks on sets we write $\mathbf{s} \sqsubseteq S$, meaning $\exists \mathbf{s}' \in S \colon \mathbf{s} \sqsubseteq \mathbf{s}'$. And $S \sqsubseteq \mathbf{s}$, meaning $\exists \mathbf{s}' \in S \colon \mathbf{s}' \sqsubseteq \mathbf{s}$.

According to Prop. 3, a state that subsumes a state on the cyan stack leads to a cycle. This explains the cycle detection in l.18. According to Prop. 2, if there is a cycle from t, then there is a cycle from all t' with $t \sqsubseteq t'$. So, if $t \sqsubseteq \mathit{Red}$, it cannot be on an accepting cycle, which explains subsumption in l.19. By definition (Def. 11), $\mathcal{PZG}_\sqsubseteq(\mathcal{B})$ contains a "larger" state for all reachable states in $\mathcal{PZG}(\mathcal{B})$, so this is sufficient to find all accepting cycles.

Since red states do not lead to accepting cycles, red states can even prune the blue search. We can strengthen the condition on l.8 to $t \notin \mathit{Blue} \cup \mathit{Cyan} \cup \mathit{Red}$. However, this is by itself of no use since, $\mathit{Red} \subseteq \mathit{Blue}$. Luckily, even

Alg. 2 NDFS with subsumption checks and red prune of *dfsBlue* (in light-blue) and early pruning (in yellow).

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

states subsumed by red do not lead to accepting cycles (contraposition of Prop. 2), so we can use subsumption again: $t \notin Blue \cup Cyan \wedge t \not\sqsubseteq Red$, as in l.8. The benefit of this can be illustrated using Fig. 3. Once *dfsBlue* backtracks over s_1, we have $s_1, s_2, s_3 \in Red$ by *dfsRed* at l.11. Any hypothetical other path from s_0 to a state subsumed by these red states can be ignored.

Finally, if the algorithm reaches l.4, the algorithm has completely traversed the subsumed state space $\mathcal{PZG}_{\sqsubseteq}(\mathcal{B})$, which was apparently finite. Since no accepting cycle was detected, by Prop. 4 the algorithm may conclude that $\mathcal{B}$ is empty.

4.2. Early pruning of the red search

Some simple observation allows for avoiding unnecessary explorations: the projection of a zone Z onto the parameter set P, $Z{\downarrow_P}$ decreases along a path.

Notation 1. Let $\mathbf{s} = (l, Z)$ and $\mathbf{s}' = (l', Z')$. By $\mathbf{s} =_p \mathbf{s}'$ we denote that $Z{\downarrow_P} = Z'{\downarrow_P}$, *i.e.* they have the same parametric zone. Similarly, by $\mathbf{s} \subseteq_p \mathbf{s}'$ we denote that $Z{\downarrow_P} \subseteq Z'{\downarrow_P}$.

Note that in both notations, the locations l and l' may be different, as opposed to the requirement for $\sqsubseteq$.

Proposition 5. Let $\mathbf{s}, \mathbf{s}'$ be two states, s.t. $\mathbf{s} \Rightarrow \mathbf{s}'$. Then $\mathbf{s}' \subseteq_p \mathbf{s}$.

Proof 5. Let $\mathbf{s} = (l, Z)$ and $\mathbf{s}' = (l', Z')$. By Def. 5: $(l, Z) \overset{e}{\Rightarrow} (l', Z')$ if $e = (l, g, a, R, l')$ and $Z' = \big([(Z \wedge g)]_R \wedge I(l')\big)^{\nearrow} \wedge I(l')$.
Since the projection onto P does not contain clocks, it is not affected by resets nor time elapsing. Hence, we have:

$$\begin{aligned} Z'{\downarrow_P} &= \big[\big([(Z \wedge g)]_R \wedge I(l')\big)^{\nearrow} \wedge I(l')\big]{\downarrow_P} \\ &= \big[\big((Z \wedge g){\downarrow_P} \wedge I(l')\big){\downarrow_P} \wedge I(l')\big]{\downarrow_P} \\ &= \big((Z \wedge g){\downarrow_P} \wedge I(l')\big){\downarrow_P} \\ &\subseteq \big(Z{\downarrow_P} \wedge I(l')\big){\downarrow_P} \\ &\subseteq Z{\downarrow_P} \end{aligned}$$

Prop. 5 allows for an early pruning of the red search, as highlighted in yellow in Alg. 2. Indeed, the red search aims at finding a cycle, which necessarily has the same parametric zone $Z_{\mathbf{s}}{\downarrow_P}$ for all its states. Thus, if a successor has a smaller parametric zone $Z_t{\downarrow_P} \subset Z_s{\downarrow_P}$, it cannot be part of the cycle, so should not be considered by the red search.

4.3. Starting the red search early: A layered NDFS

As seen in Sec. 4.2, all symbolic states in an accepting cycle have the same parametric zone $Z{\downarrow_P}$. From this property, we can organise our search by considering *layers* of states with the same parametric zone. Contrary to standard NDFS (for automata or timed automata), a red search in a $\mathcal{PZG}$ cannot interfere with a red search in another layer of the same path, since they concern different parametric zones.

Thus the new *layered* NDFS shown in Alg. 3 works layer by layer, looking at the larger parametric zones first. The changes from Alg. 2 are highlighted in yellow.

Notation 2. With $t \sqsubseteq_{=_p} X$ we denote that $\exists t' \in X.t \sqsubseteq t' \wedge t' =_p t$. That is, t is subsumed by some element of X in the same parametric layer.

In order to obtain a result as fast as possible, and with the largest parametric zone as possible, we run the blue search until the parametric zone changes (ll.13–15). The last state thus constructed is kept as *Pending* (l.14), *i.e.* its successors are not generated yet, and we continue the NDFS algorithm on other branches if any. When backtracking, the deepest accepting state in the layer will be encountered first, thus preserving the post-order for the red search in the layer.

If an accepting cycle is found it is reported by the algorithm, otherwise the exploration continues in the current layer. When the layer is finished, the algorithm is applied to the pending states (ll.4–6); the order of processing those is left as implementation freedom. In particular, processing large zones first tends to be more efficient.

In the red search, the comparison with cyan states at l.25 is still valid. Indeed, all cyan states are on the path leading to the state examined; if one of them is subsumed by the current state t, its parametric zone is smaller than or equal to the one of t, but it is also larger than or equal to it as it is on the path. Hence they have the same parametric zone, and only the subsumption on clocks applies as for TA.

The comparison with red states at l.26, however, needs to be limited to the current layer, so as not to interfere with a previous red search on another layer. This is sufficient since any red state encountered on the same layer should have led, during its red search, to a cycle. A similar argument applies to the comparison with red states of the same layer only in the blue search at l.12.

Remark 1. Such a layered NDFS can also be applied to automata or TA, provided the model exhibits some progress measure that allows for determining layers. This is similar to the sweepline approach [13].

5. Collecting NDFS for Parameter Synthesis

This section addresses the use of NDFS to synthesize parameter values that lead to an accepting cycle, *i.e.* to find all possible valuations of the parameters such that there exists an accepting run.

To achieve synthesis, finding one accepting cycle is not sufficient anymore: they should all be found. Therefore, the NDFS algorithm of Sec. 4 is extended to a *collecting* NDFS that continues the exploration.

The reporting of a cycle in Alg. 4 does no longer exit: it just collects in *Constraints* (l.28) the constraint $Z_t{\downarrow_P}$ that was just found. Moreover, in order to avoid exploring smaller parametric zones, we will only process states that are not contained in *Constraints* (l.10).

Alg. 3 Layered NDFS

```
1:  procedure layered_ndfs( )
2:      Cyan := Blue := Red := ∅
3:      Pending := {s_0}
4:      while Pending ≠ ∅ do
5:          Pick s from Pending
6:          if s ∉ Blue then dfsBlue(s)
7:      report no cycle
8:  procedure dfsBlue(s)
9:      Cyan := Cyan ∪ {s}
10:     for all t in NEXT-STATE(s) do
11:         if t ∉ Blue ∪ Cyan
12:             ∧ t ⋢_{=_p} Red then
13:             if t ⊂_p s then
14:                 Pending := Pending ∪ {t}
15:             else
16:                 dfsBlue(t)
17:     if s ∈ F then
18:         dfsRed(s)
19:     Blue := Blue ∪ {s}
20:     Cyan := Cyan \ {s}
21: procedure dfsRed(s)
22:     Red := Red ∪ {s}
23:     for all t in NEXT-STATE(s)
24:         s.t. t =_p s do
25:         if Cyan ⊑ t then report cycle
26:         if t ⋢_{=_p} Red then dfsRed(t)
```

Alg. 4 Layered collecting NDFS

```
1:  procedure layered_ndfs( )
2:      Cyan := Blue := Red := ∅
3:      Constraints := ∅
4:      Pending := {s_0}
5:      while Pending ≠ ∅ do
6:          Pick s from Pending
7:          if s ∉ Blue then dfsBlue(s)
8:      return Constraints
9:  procedure dfsBlue(s = (l_s, Z_s))
10:     if s↓_P ⋢ Constraints then
11:         Cyan := Cyan ∪ {s}
12:         for all t in NEXT-STATE(s) do
13:             if t ∉ Blue ∪ Cyan
14:                 ∧ t ⋢_{=_p} Red then
15:                 if t ⊂_p s then
16:                     Pending := Pending ∪ {t}
17:                 else
18:                     dfsBlue(t)
19:         if s ∈ F then
20:             dfsRed(s)
21:         Blue := Blue ∪ {s}
22:         Cyan := Cyan \ {s}
23: procedure dfsRed(s)
24:     Red := Red ∪ {s}
25:     for all t = (l_t, Z_t) in NEXT-STATE(s)
26:         s.t. t =_p s do
27:         if Cyan ⊑ t then
28:             Constraints := Constraints ∪ Z_t↓_P
29:         else if t ⋢_{=_p} Red then
30:             dfsRed(t)
```

6. Experiments

To evaluate the performances of our algorithms, we ran our experiments on a Dell Precision 3620 i7-7700 3.60 GHz with 64 GiB memory running Linux Mint 19 beta 64 bits.

6.1. Implementation

We implemented our algorithms in IMITATOR [4] (Working version 2.9.2), where polyhedra operations are performed using the PPL library [7].

For better performance, in algorithms 3 and 4, we reuse the idea from the PRIOR strategy in [5] for implementing the *Pending* list: each explored state is inserted in a decreasing zone fashion into *Pending*, and thus the state having largest zone or the state at the beginning of *Pending* is popped out first. Besides, we use an additional index, storing information on the sets of comparable zones, to speed up the state insertion.

6.2. Experimental results

We used 25 benchmarks from the IMITATOR benchmarks library that cover a representative selection of examples, including hardware circuits (`flipflop`, `spsmall`), network or software protocols (`BRP`, `FDDI-4`, `Fischer`, `F3`, `F4`, `Lynch-2`, `Lynch-5`, `critical-region`, `RCP`, `simop`, `WFAS`), real-time systems (`Thales-1`, `Thales-3`, `Sched2.i.j`), variants of a producer-consumer (`Pipeline` [17]), and few additional case studies `coffee`, `train-gate`, `JLR13`. Since these benchmarks were used for reachability, they did not include an accepting state. Hence, we have added some accepting states by hand.

In this experiment, we mainly focus on two parameter synthesis problems. The first problem, called EC-EX, is the counter-example synthesis: "find at least some parameter valuations for which an accepting cycle is found". Counter-example synthesis is of high practical importance, as it is often desirable to find at least some valuations for which an accepting cycle is found, not necessarily all existing ones. We implemented a procedure EC-EX that stops as soon as some parameter valuations allowing for reaching an accepting cycle are synthesized. In order to evaluate the effectiveness of our new approach, we compared three versions: plain NDFS (Alg. 1), NDFS with subsumption (Alg. 2), and NDFS with subsumption and layering (Alg. 3).

Instead of finding some parameter valuations for a single accepting cycle as in EC-EX, the second problem addressed, EC-CYCLES, synthesizes all possible valuations:

"find all parameter valuations for which an accepting cycle exists". Note that, due to the undecidability of the EF-emptiness problem [2], algorithms for ECCYCLES or ECCEX are not guaranteed to terminate.

In order to show the performance of our algorithm LAYERCOLLECTNDFSSUB (Alg. 4) is compared with the breadth first search STATESPACE synthesis with the inclusion reduction [5] which explores all possible states of the system. Evidently, it is not entirely fair that ECCYCLES only explores a part of the whole state space while STATESPACE generates it all w.r.t. the inclusion. Nevertheless, it makes sense since the reduction criterion is similar, and STATESPACE analyses reachability of accepting locations.

From left to right in table 1 are the models' names followed by some information on each model (number of clocks, parameters, and locations) and computation times in seconds for each of the five algorithms. Additionally, the number of different zones of accepting cycles found by LAYERCOLLECTNDFSSUB is indicated next to it. Note that the green and yellow cells are the fastest and the second-fastest approaches respectively, for each of the two categories of algorithms, and "TO" stands for a time-out after 30 minutes.

The table is divided into two parts by a grey vertical line so as to reflect the two distinct comparisons. The first part comprises NDFS (Alg. 1) and its variants NDFSSUB (Alg. 2) and LAYERNDFSSUB (Alg. 3). The other is a comparison between LAYERCOLLECTNDFSSUB (Alg. 4) and PRIOR with inclusion ([5]), which is a BFS (breadth first search) based algorithm with optimised search order for parametric zone inclusion. The PRIOR with inclusion is the current efficient BFS algorithm in IMITATOR for reducing state space explosion and improving termination.

In the first part of the table, NDFSSUB dominates other algorithms, since it is the fastest on more than half of benchmarks (17/25 cases). However, in 4 cases, LAYERNDFSSUB is even faster. It is interesting that this layered algorithm terminates very quickly in two cases where the non-layered versions do not terminate. Clearly, layering can prevent the algorithm to diverge. Note however, that in 4 other cases, the layering algorithm times out, while the non-layered algorithms provide an answer quickly. In these cases the zone graph is broad already in the top layers, so apparently the strict NDFS versions find an accepting cycle more efficiently.

NDFS and NDFSSUB suffer from the undecidability and cannot terminate and reach the time-out in some benchmarks. Algorithm LAYERNDFSSUB was proposed to cope with the termination problem. We observe that this works in two cases: `coffee` and `F4`. By exploring the state having the largest zone first (especially true zone states), the layered algorithm can avoid exploring infinite paths having smaller zones in the beginning and thus they can also prune these paths later. The timeout in 4 other cases is not caused by infinite behaviour, but by exploring many branches in the top-layers. Here a result would have been produced with larger timeout value.

Let us interpret the second part of the table, where we compare the termination of LAYERCOLLECTNDFSSUB with PRIOR. The results of LAYERCOLLECTNDFSSUB are similar to those of LAYERNDFSSUB: only 2 more cases hit the timeout limit. Note that, for efficiency purpose, all algorithms explore states on-the-fly so that in LAYERNDFSSUB, we receive a zone result when some cycle containing an accepting location is found, while LAYERCOLLECTNDFSSUB would continue the search. PRIOR exhibits some complimentary unterminated cases. In the terminating cases, LAYERCOLLECTNDFSSUB is the fastest in 9/26 cases, where PRIOR is the fastest in 8/26 cases. The reason is probably that PRIOR uses full parametric zone inclusion, which is sound for reachability but not for liveness. We conclude that both the efficiency and the termination behaviour of these algorithms are quite complementary. However, note that these algorithms address a different problem.

Finally, we note that the number of symbolic zones collected during the exploration is widely different across the benchmarks (ranging from 1-1026). Even in two of the timed-out runs, LAYERCOLLECTNDFSSUB collected 2, resp. 13, parametric zones that lead to an accepting cycle (`coffee` and `WFAS`).

7. Conclusion

In this paper, we have proposed a new *nested depth-first search* (NDFS) algorithm and its variants for model-checking LTL properties of Parametric Timed Automata. This algorithm features several reduction mechanisms: subsumption, as in [18], early pruning, and a layered approach. It addresses both the problem of existence of an accepting cycle, and the synthesis of parameters that allow for a such a cycle in a collecting version.

Our approach has the advantage of performing verification as early as possible, instead of fully exploring a branch of the parametric zone graph, which may be infinite. Experimental results show the efficiency of the layered approach. Subsumption and/or layering improves the efficiency of the algorithm, and also the chance of termination. We noted that the behaviour of subsumption with pure NDFS or with layered NDFS are quite complementary.

Moreover, we show that such an approach can be used not only for Parametric Timed Automata but also for models featuring a progress measure which can be used to determine layers. Another nice feature is that the absence of cycles in the subsumed graph guarantees that no cycle exists in the PZG either, thus providing a quick answer when the formula holds, as exploited in [15].

Future work could investigate intricate exploration orders that combine the strengths of NDFS with subsumption and layering, to terminate (faster) in even more cases. Other directions for future work include adapting this new nested depth-first search algorithm to a multi-core setting, similar to [14], [18]. Alternative parallel approaches will also be studied, as described in the survey [8].

EC Algorithms							STATESPACE Algorithms		
Benchmark	**#**	**#**	**#**	ECC-EX			ECCYCLES		STATESPACE
Models	***X***	***P***	***L***	**NDFS (s)**	**NDFSSUB (s)**	**LAYERNDFSSUB (s)**	**LAYERCOLLECTNDFSSUB (s)**	**#Zones**	**BFS PRIOR incl (s)**
BRP	7	2	22	0.231	0.237	0.035	0.043	4	176.535
coffee	2	3	4	**TO**	**TO**	0.008	**TO**	2	0.006
critical-region	2	2	20	**TO**	**TO**	**TO**	**TO**	0	**TO**
F3	3	0	18	0.026	0.026	0.006	0.003	1	0.255
F4	4	2	23	**TO**	**TO**	0.007	0.006	1	**TO**
FDDI4	13	2	34	0.305	0.235	1.260	7.004	136	1.319
FischerAHV93	2	4	13	0.010	0.009	0.013	0.013	1	0.040
flipflop	5	2	52	0.010	0.010	0.010	0.012	1	0.015
fmtv1A1-v2	3	3	15	0.060	0.057	46.723	68.223	29	14.040
fmtv1A3-v2	3	3	15	0.063	0.062	302.061	1129.284	67	215.020
JLR13	2	2	2	**TO**	**TO**	**TO**	**TO**	0	**TO**
lynch	2	1	18	0.007	0.007	0.010	0.010	5	0.016
lynch5	5	1	45	0.012	0.012	0.016	0.019	9	3.126
Pipeline-KP12-2-3	4	6	14	0.510	0.508	7.738	492.995	369	**TO**
Pipeline-KP12-2-5	4	6	18	0.791	0.787	**TO**	**TO**	0	**TO**
Pipeline-KP12-3-3	5	6	19	1.960	1.962	**TO**	**TO**	0	**TO**
RCP	6	5	48	0.024	0.013	0.023	0.034	7	10.095
Sched2.50.0	6	2	17	0.011	0.011	0.539	4.168	71	1.940
Sched2.50.2	6	2	17	0.011	0.011	**TO**	**TO**	0	**TO**
Sched2.100.0	6	2	17	0.008	0.008	0.417	3.769	31	2.425
Sched2.100.2	6	2	17	0.008	0.008	**TO**	**TO**	0	**TO**
simop	8	2	46	**TO**	**TO**	**TO**	**TO**	0	**TO**
spsmall	11	2	51	0.244	0.053	0.310	36.549	1026	5.650
train-gate	5	9	11	0.016	0.015	0.047	0.268	15	0.018
WFAS	4	2	10	0.023	0.021	0.056	**TO**	13	**TO**

Table 1. EXPERIMENTAL COMPARISON OF NESTED DFS ALGORITHMS

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *TCS*, 126(2):183–235, 1994.

[2] R. Alur, T.A. Henzinger, and M.Y. Vardi. Parametric real-time reasoning. In *STOC*, pages 592–601. ACM, 1993.

[3] É. André. What's decidable about parametric timed automata? *International Journal on Software Tools for Technology Transfer*, 2017.

[4] É. André, L. Fribourg, U. Kühne, and R. Soulat. Imitator 2.5: A tool for analyzing robustness in scheduling problems. In *FM*, volume 7436 of *LNCS*. Springer, 2012.

[5] É. André, H.G. Nguyen, and L. Petrucci. Efficient parameter synthesis using optimized state exploration strategies. In Z. Hu and G. Bai, editors, *Proceedings of the 22nd International Conference on Engineering of Complex Computer Systems (ICECCS 2017)*, pages 1–10. IEEE, 2017.

[6] É. André, H.G. Nguyen, L. Petrucci, and J. Sun. Parametric model checking timed automata under non-zenoness assumption. In *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, pages 35–51, 2017.

[7] R. Bagnara, P.M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[8] J. Barnat, V. Bloemen, A. Duret-Lutz, A.W. Laarman, L. Petrucci, J.C. van de Pol, and É. Renault. Parallel model checking algorithms for linear-time temporal logic. In Y. Hamadi and L. Sais, editors, *Handbook of Parallel Constraint Reasoning.*, pages 457–507. Springer, 2018.

[9] P. Bezdek, N. Benes, J. Barnat, and I. Cerná. LTL parameter synthesis of parametric timed automata. In *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016, Proceedings*, pages 172–187, 2016.

[10] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[11] A.E. Dalsgaard, A.W. Laarman, K.G. Larsen, M.C. Olesen, and J.C. van de Pol. Multi-core reachability for timed automata. In *FORMATS*, LNCS 7595, 2012.

[12] C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *TACAS*, LNCS 1384, pages 313–329. Springer, 1997.

[13] S. Evangelista and L.M. Kristensen. A sweep-line method for Büchi automata-based model checking. *Fundam. Inform.*, 131(1):27–53, 2014.

[14] S. Evangelista, A.W. Laarman, L. Petrucci, and J.C. van de Pol. Improved multi-core nested depth-first search. In *ATVA*, LNCS 7561, pages 269–283, 2012.

[15] F. Herbreteau, B. Srivathsan, T.-T. Tran, and I. Walukiewicz. Why liveness for timed automata is hard, and what we can do about it. In *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India*, pages 48:1–48:14, 2016.

[16] A. Jovanović, D. Lime, and O.H. Roux. Integer parameter synthesis for timed automata. *TSE*, 41(5):445–461, 2015.

[17] M. Knapik and W. Penczek. Bounded model checking for parametric timed automata. *ToPNoC*, 6900(5):141–159, 2012.

[18] A.W. Laarman, M.C. Olesen, A.E. Dalsgaard, K.G. Larsen, and J.C. van de Pol. Multi-core emptiness checking of timed Büchi automata using inclusion abstraction. In *Computer Aided Verification (CAV'13)*, pages 968–983, 2013.

[19] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In *TACAS*, LNCS 3440, pages 174–190. Springer, 2005.