Learning-based compositional parameter synthesis for event-recording automata*

Étienne André[1] and Shang-Wei Lin[2]

[1] Université Paris 13, LIPN, CNRS, UMR 7030, France
[2] SCSE, Nanyang Technological University, Singapore

Abstract. We address the verification of timed concurrent systems with unknown or uncertain constants considered as parameters. First, we introduce parametric event-recording automata (PERAs), as a new subclass of parametric timed automata (PTAs). Although in the non-parametric setting event-recording automata yield better decidability results than timed automata, we show that the most common decision problem remains undecidable for PERAs. Then, given one set of components with parameters and one without, we propose a method to compute an abstraction of the non-parametric set of components, so as to improve the verification of reachability properties in the full (parametric) system. We also show that our method can be extended to general PTAs. We implemented our method, which shows promising results.

Keywords: parametric timed automata, learning, abstractions, parametric reachability checking, parameter synthesis

1 Introduction

Verifying distributed systems involving timing constraints is notoriously difficult, especially when timing constants may be uncertain. This problems becomes even more difficult (often intractable) in the presence of timing parameters, i. e., unknown timing constants. Parametric reachability synthesis aims at synthesizing timing parameter valuations for which a set of (usually bad) states is reachable. Parametric timed automata (PTAs) [AHV93] is a parametric extension of timed automata (TAs) to model and verify models involving (possibly parametric) timing constraints and concurrency. Its high expressiveness comes with the drawback that most interesting problems are undecidable [And15].

* This work is partially supported by the ANR national research program "PACS" (ANR-14-CE28-0002). This is the author version of the manuscript of the same name published in the proceedings of the 37th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2017). The final publication is available at link.springer.com. When compared to the official publication, this author version also includes all proofs, as well as additional experiments.

Related work Despite undecidability of the theoretical problems, several monolithic (non-compositional) techniques for parametric reachability synthesis in PTAs have been proposed in the past, either in the form of semi-algorithms (a procedure that is correct but may not terminate), or using approximations. In [AHV93], a basic semi-algorithm (called EFsynth in [JLR15]) has been proposed: it explores the symbolic state space until bad states are found, and gathers the associated parameter constraints. In [FJK08], approximated parametric reachability synthesis is performed using counter-example guided abstraction refinement (CEGAR) techniques for parametric linear hybrid automata, a class of models more expressive than PTAs. In [CPR08], the computation of schedulability regions for real-time systems reduces to a parametric reachability synthesis using PTAs; the algorithm does not terminate in general, but does on a subclass of real-time tasks. In [JLR15,ALR15], an abstraction technique based on the integer hull is used in an algorithm with guaranteed termination, the result of which contains at least all integer parameter valuations in a bounded parameter domain, and all rational-valued valuations between consecutive integer points (as well as possibly some other rational-valuations). In [ALNS15], we proposed a point-based technique: instead of attacking the reachability synthesis in a brute-force manner, we iterate on (some) integer parameter valuations, and derive for each of them a constraint around this valuation that preserves the (non-)reachability of the bad locations. Although numerous iterations may be needed, each of them explores a much smaller part of the state space than the brute-force exploration of EFsynth, often resulting in a faster execution than EFsynth. In addition, this technique can be distributed on a cluster [ACN15].

Distributed systems are often made of a set of components interacting with each other; taking advantage of the compositionality is a goal often desired to speed up verification. In [CGP03], a learning-based approach is proposed to automate compositional verification of untimed systems modeled by labeled transition systems (LTS). For timed systems, we proposed a learning-based compositional verification framework [LAL+14] for event-recording automata (ERAs), a subclass of TAs for which language inclusion is decidable [AFH99]. This approach showed to be much faster than monolithic verification.

The recent work [ABB+16] is close to our goal, as it proposes an approach for compositional parameter synthesis, based on the derivation of interaction and component invariants. The method relies on the concept of history clocks. The method is implemented in a prototype in Scala, making use of IMITATOR [AFKS12]. Whereas both [ABB+16] and our approach address reachability or safety properties, the class of PTAs of [ABB+16] is larger; conversely, we add no further restrictions on the models, whereas in [ABB+16] all clocks and (more problematically) parameters must be local to a single component and cannot be shared.

Contribution In this work, we propose an approach relying on a point-based technique for parametric reachability synthesis, combined with learning-based abstraction techniques, for a subclass of PTAs, namely parametric event-recording automata. We propose this subclass due to the decidability of the

language inclusion in the non-parametric setting. We consider a set of parametric components A (where parameters are dense in a bounded parameter domain D_0) and a set of non-parametric components B, with their parallel composition denoted by $\mathsf{A} \parallel \mathsf{B}$. For each integer parameter valuation v not yet covered by a good or bad constraint, we try to compute, by learning, an abstraction $\widetilde{\mathsf{B}}$ of B s.t. $v(\mathsf{A}) \parallel \mathsf{B}$ does not reach the bad locations. We then "enlarge" the valuation v using the abstract model $\mathsf{A} \parallel \widetilde{\mathsf{B}}$, which yields a dense good constraint; we prove the correctness of this approach. If the learning fails to compute an abstraction, we derive a counter-example, and we then replay it in the fully parametric model $\mathsf{A} \parallel \mathsf{B}$, which allows us to derive very quickly a bad dense constraint. We iterate until (at least) all integer points in D_0 are covered. In practice, we cover not only all rational-valued in D_0, but in fact the entire parameter space (except for one benchmark for which we fail to compute a suitable abstraction).

We propose the following technical contributions:

1. we introduce a parametrization of event-recording automata (PERAs);
2. we show that the reachability emptiness problem is undecidable for PERAs;
3. we then introduce our approach that combines iteration-based synthesis with learning-based abstraction;
4. we implement our approach into a toolkit using IMITATOR and CV, and we demonstrate its efficiency on several case studies.

Outline Section 2 introduces the necessary preliminaries. Section 3 recalls the parametric reachability preservation [ALNS15]. Section 4 introduces parametric event-recording automata, and proves the undecidability of the reachability emptiness problem. Section 5 introduces our main contribution, and Section 6 evaluates it on benchmarks. Section 7 concludes the paper.

2 Preliminaries

2.1 Clocks, parameters and constraints

Let $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}_+$ and $\mathbb{R}_+$ denote the sets of non-negative integers, integers, non-negative rational and non-negative real numbers respectively.

Throughout this paper, we assume a set $X = \{x_1, \ldots, x_H\}$ of *clocks*, i. e., real-valued variables that evolve at the same rate. A clock valuation is a function $\mu : X \to \mathbb{R}_+$. We write $\mathbf{0}$ for the clock valuation that assigns 0 to all clocks. Given $d \in \mathbb{R}_+$, $\mu + d$ denotes the valuation such that $(\mu + d)(x) = \mu(x) + d$, for all $x \in X$. Given $R \subseteq X$, we define the *reset* of a valuation μ, denoted by $[\mu]_R$, as follows: $[\mu]_R(x) = 0$ if $x \in R$, and $[\mu]_R(x) = \mu(x)$ otherwise.

We assume a set $P = \{p_1, \ldots, p_M\}$ of *parameters*, i. e., unknown rational-valued constants. A parameter *valuation* (or *point*) v is a function $v : P \to \mathbb{Q}_+$.

In the following, we assume $\lhd \in \{<, \leq\}$ and $\bowtie \in \{<, \leq, \geq, >\}$. Throughout this paper, *lt* denotes a linear term over $X \cup P$ of the form $\sum_{1 \leq i \leq H} \alpha_i x_i + \sum_{1 \leq j \leq M} \beta_j p_j + d$, with $\alpha_i, \beta_j, d \in \mathbb{Z}$. Similarly, *plt* denotes a parametric linear

term over P, that is a linear term without clocks ($\alpha_i = 0$ for all i). A *constraint* C (i. e., a convex polyhedron) over $X \cup P$ is a conjunction of inequalities of the form $lt \bowtie 0$. Given a parameter valuation v, $v(C)$ denotes the constraint over X obtained by replacing each parameter p in C with $v(p)$. Likewise, given a clock valuation μ, $\mu(v(C))$ denotes the Boolean value obtained by replacing each clock x in $v(C)$ with $\mu(x)$.

A *guard* g is a constraint over $X \cup P$ defined by a conjunction of inequalities of the form $x \bowtie plt$.

A parameter constraint K is a constraint over P. We write $v \models K$ if $v(K)$ evaluates to true. $\bot$ (resp. $\top$) denotes the special parameter constraint containing no (resp. all) parameter valuations. We will sometime manipulate *non-convex* constraints over P, i. e., finite unions of parameter constraints. Such non-convex constraints can be implemented using finite lists of constraints, and therefore all definitions extend in a natural manner to non-convex constraints.

A *parameter domain* is a box parameter constraint, i. e., a conjunction of inequalities of the form $p \bowtie d$, with $d \in \mathbb{N}$. A parameter domain D is *bounded* if, for each parameter, there exists in D an inequality $p \lhd d$ (recall that, additionally, all parameters are bounded below from 0 as they are non-negative). Therefore D can be seen as a hypercube in M dimensions.

2.2 Parametric Timed Automata

Parametric timed automata (PTAs) extend timed automata with parameters within guards and invariants in place of integer constants [AHV93].

Definition 1 (PTA). *A* parametric timed automaton *(hereafter PTA)* A *is a tuple* $(\Sigma, L, l_0, X, P, I, E)$*, where: i)* Σ *is a finite set of actions, ii)* L *is a finite set of locations, iii)* $l_0 \in L$ *is the initial location, iv)* X *is a finite set of clocks, v)* P *is a finite set of parameters, vi)* I *is the invariant, assigning to every* $l \in L$ *a guard* $I(l)$*, vii)* E *is a finite set of edges* $e = (l, g, a, R, l')$ *where* $l, l' \in L$ *are the source and target locations,* $a \in \Sigma$*,* $R \subseteq X$ *is a set of clocks to be reset, and* g *is a guard.*

Given a PTA A and a parameter valuation v, we denote by $v(\mathsf{A})$ the non-parametric timed automaton where all occurrences of a parameter p_i have been replaced by $v(p_i)$.

As usual, PTAs can be composed by performing their parallel composition, i. e., their synchronized product on action names.

Concrete Semantics

Definition 2 (Concrete semantics of a TA). *Given a PTA* $\mathsf{A} = (\Sigma, L, l_0, X, P, I, E)$*, and a parameter valuation* v*, the concrete semantics of* $v(\mathsf{A})$ *is given by the timed transition system* $(S, s_0, \rightarrow)$*, with* $S = \{(l, \mu) \in L \times \mathbb{R}_+^H \mid \mu(v(I(l)))$ *is true*$\}$*,* $s_0 = (l_0, \mathbf{0})$*, and* $\rightarrow$ *consists of the discrete and (continuous) delay transition relations:*

- *discrete transitions:* $(l, \mu) \xrightarrow{e} (l', \mu')$*, if* $(l, \mu), (l', \mu') \in S$*, there exists* $e = (l, g, a, R, l') \in E$*,* $\mu' = [\mu]_R$*, and* $\mu(v(g))$ *is true.*

– *delay transitions:* $(l,\mu) \xrightarrow{d} (l,\mu{+}d)$, *with* $d \in \mathbb{R}_+$, *if* $\forall d' \in [0,d], (l,\mu{+}d') \in S$.

A *(concrete) run* is a sequence $\rho = s_0\gamma_0 s_1\gamma_1 \cdots s_n\gamma_n \cdots$ such that $\forall i, (s_i,\gamma_i,s_{i+1}) \in \rightarrow$. We consider as usual that concrete runs strictly alternate delays d_i and discrete transitions e_i and we thus write concrete runs in the form $\rho = s_0 \xrightarrow{(d_0,e_0)} s_1 \xrightarrow{(d_1,e_1)} \cdots$. The corresponding *timed word* is $(a_0,t_0),(a_1,t_1),\cdots$ where a_i is the action of e_i and $t_i = \sum_{j=0}^{i} d_i$. Note that when a run is finite, it must end with a state. A maximal run is a run that is either infinite, or that cannot be extended. Given a state $s = (l,\mu)$, we say that s is reachable (or that $v(\mathsf{A})$ reaches s) if s belongs to a run of $v(\mathsf{A})$. By extension, we say that l is reachable in $v(\mathsf{A})$, if there exists a state (l,μ) that is reachable. Given $L^{☹} \subseteq L$, we say that $L^{☹}$ is reachable in $v(\mathsf{A})$ if $\exists l \in L^{☹}$ s.t. l is reachable.

Let $\rho = (l_0,\mu_0) \xrightarrow{(d_0,e_0)} (l_1,\mu_1) \xrightarrow{(d_1,e_1)} \cdots (l_n,\mu_n) \xrightarrow{(d_n,e_n)} \cdots$ be a run of $v(\mathsf{A})$. The *trace* of this run (denoted by $trace(\rho)$) is the sequence $e_0e_1\cdots e_n\cdots$, and the *untimed word* of this run is $a_0a_1\cdots a_n\cdots$, where a_i is the action of e_i for all i. The *trace set* of $v(\mathsf{A})$ is the set of traces associated with all maximal runs of A.

Symbolic semantics Let us recall the symbolic semantics of PTAs (as in e. g., [ACEF09,JLR15]). We define the *time elapsing* of a constraint C, denoted by $C^{\nearrow}$, as the constraint over X and P obtained from C by delaying all clocks by an arbitrary amount of time. That is, $C^{\nearrow} = \{(\mu,v) \mid \mu \models v(C) \wedge \forall x \in X : \mu'(x) = \mu(x) + d, d \in \mathbb{R}_+\}$. Given $R \subseteq X$, we define the *reset* of C, denoted by $[C]_R$, as the constraint obtained from C by resetting the clocks in R, and keeping the other clocks unchanged. We denote by $C{\downarrow_P}$ the projection of C onto P, i. e., obtained by eliminating the clock variables (e. g., using Fourier-Motzkin).

A *parametric zone* is a convex polyhedron over $X \cup P$ in which constraints are of the form $x \bowtie plt$ (parametric rectangular constraints), or $x_i - x_j \bowtie plt$ (parametric diagonal constraints), where $x_i, x_j \in X$ and plt is a parametric linear term over P.

A symbolic state is a pair $\mathbf{s} = (l,C)$ where $l \in L$ is a location, and C its associated parametric zone. The initial symbolic state of A is $\mathbf{s}_0 = \big(l_0, (\{\mathbf{0}\} \wedge I(l_0))^{\nearrow} \wedge I(l_0)\big)$.

The symbolic semantics relies on the Succ operation. Given a symbolic state $\mathbf{s} = (l,C)$ and an edge $e = (l,g,a,R,l')$, $\mathsf{Succ}(\mathbf{s},e) = (l',C')$, with $C' = \big([(C \wedge g)]_R \wedge I(l')\big)^{\nearrow} \wedge I(l')$. The Succ operation is effectively computable, using polyhedra operations; also note that the successor of a parametric zone C is a parametric zone (see e. g., [JLR15]).

A symbolic run of a PTA is an alternating sequence of symbolic states and edges starting from the initial symbolic state, of the form $\mathbf{s}_0 \overset{e_0}{\Rightarrow} \mathbf{s}_1 \overset{e_1}{\Rightarrow} \cdots \overset{e_{m-1}}{\Rightarrow} \mathbf{s}_m$, such that for all $i = 0,\ldots,m-1$, we have $e_i \in E$, and $\mathbf{s}_{i+1} = \mathsf{Succ}(\mathbf{s}_i, e_i)$.

Given a symbolic run $\mathbf{s}_0 \overset{e_0}{\Rightarrow} \mathbf{s}_1 \overset{e_1}{\Rightarrow} \cdots \overset{e_n}{\Rightarrow} \mathbf{s}_n \cdots$, its *trace* is the sequence $e_0e_1\cdots e_n\cdots$. Two runs (symbolic or concrete) are *equivalent* if they have the same trace.

3 Parametric reachability preservation

Let us briefly recall the parametric reachability preservation algorithm PRP defined in [ALNS15]. Given a set of locations $L^{☹}$, $\mathsf{PRP}(\mathsf{A}, v, L^{☹})$ synthesizes a dense (convex) constraint K containing at least v and such that, for all $v' \in K$, $v'(\mathsf{A})$ preserves the reachability of $L^{☹}$ in $v(\mathsf{A})$. By preserving the reachability of $L^{☹}$ in $v(\mathsf{A})$, we mean that some locations of $L^{☹}$ are reachable in $v'(\mathsf{A})$ iff they are in $v(\mathsf{A})$. That is, if $v(\mathsf{A})$ is safe (i. e., it does not reach $L^{☹}$), then $v'(\mathsf{A})$ is safe too. Conversely, if $v(\mathsf{A})$ is unsafe (i. e., $L^{☹}$ is reachable for some runs), then $v'(\mathsf{A})$ is unsafe too.

Lemma 1 (Soundness of PRP [ALNS15]). *Let* A *be a PTA,* v *a parameter valuation, and* $L^{☹}$ *a subset of locations. Let* $K = \mathsf{PRP}(\mathsf{A}, v, L^{☹})$.

For all $v' \models K$, $v'(\mathsf{A})$ *reaches* $L^{☹}$ *iff* $v(\mathsf{A})$ *reaches* $L^{☹}$.

A specificity of PRP is that it does *not* aim at completeness; instead, it focuses on behaviors "similar" to that of $v(\mathsf{A})$ so as not to explore a too large part of the state space, and outputs valuations neighboring v. A sort of completeness can be achieved by iterating PRP on various parameter valuations: when $v(\mathsf{A})$ has computed K, the algorithm can be called again on a valuation v_2 "neighbor" of the result K, and so on until either the entire parameter space has been covered, or when a certain coverage of a bounded parameter domain has been achieved (e. g., 99 %). This iterated version is called PRPC (for PRP cartography), takes as input a PTA A and a bounded parameter domain D_0, and iteratively calls PRP on parameter valuations of D_0 with a given precision (e. g., at least all integer-valued). This gives a cartography of D_0 with a union K_{good} of safe constraints (valuations for which $L^{☹}$ is unreachable) and a union K_{bad} of unsafe constraints (for which $L^{☹}$ is reachable). Although only the coverage of the discrete points (e. g., integer-valued) can be theoretically guaranteed, PRPC often covers most (if not all) of the dense state space within D_0, and often outside too.

In practice, PRPC turned to be efficient when compared to the brute force synthesis EFsynth, not only in terms of valuations coverage (the constraint output by PRPC is often weaker than that of EFsynth) but also in terms of efficiency (time and memory). A possible explanation is that, although PRPC performs repeated calls to PRP (sometime performing redundant computation), it only explores a limited part of the state space at once.

In addition, PRPC showed good results when being distributed on a cluster [ALNS15], as different calls to PRP can be made in parallel (with a risk of redundancy though).

4 Parametric event-recording automata

Event-recording automata (ERAs) [AFH99] are a subclass of timed automata, where each action label is associated with a clock such that, for every edge with a label, the associated clock is reset. We propose here a parametric extension of ERAs, following the parameterization of TAs into PTAs.

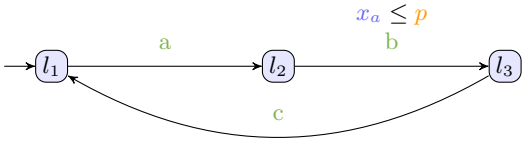


Fig. 1: An example of a PERA

Formally, let Σ be a set of actions: we denote by X_Σ the set of clocks associated with Σ, i. e., $\{x_a \mid a \in \Sigma\}$. A Σ-guard is a guard on $X_\Sigma \cup P$.

Definition 3 (PERAs). *A* parametric event-recording automaton (PERA) *is a tuple $(\Sigma, L, l_0, P, I, E)$, where: i) Σ is a finite set of actions, ii) L is a finite set of locations, iii) $l_0 \in L$ is the initial location, iv) P is a finite set of parameters, v) I is the invariant, assigning to every $l \in L$ a Σ-guard $I(l)$, vi) E is a finite set of edges $e = (l, g, a, x_a, l')$ where $l, l' \in L$ are the source and target locations, $a \in \Sigma$, x_a is is the clock to be reset, and g is a Σ-guard.*

Just as for ERAs, PERAs can be seen as a syntactic subclass of PTAs: a PERA is a PTA for which there is a one-to-one matching between clocks and actions and such that, for each edge, the clock corresponding to the action is the only clock to be reset.

Following the conventions used for ERAs, we do not explicitly represent graphically the clock x_a reset along an edge labeled with a: this is implicit.

Example 1. Figure 1 depicts an example of PERA with 3 actions (and therefore 3 clocks x_a, x_b and x_c), and one parameter p. Only clock x_a is used in a guard.

It is well-known that the EF-emptiness problem ("is the set of parameter valuations for which it is possible to reach a given location empty?") is undecidable for PTAs [AHV93,ALR16a]. Reusing the proof of [ALR16a], we show below that this remains undecidable for PERAs.

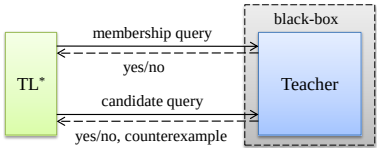
Theorem 1. *The EF-emptiness problem is undecidable for PERAs, even with bounded parameters.*

Proof. The proof works by adapting to PERAs the proof of [ALR16a, Theorem 1], and is therefore given in Appendix A.

This negative result rules out the possibility to perform exact synthesis for PERAs. Still, in the next section, we propose an approach that is sound, though maybe not complete: the synthesized valuations are correct, but some may be missing. More pragmatically, we aim at improving the synthesis efficiency.

$$\frac{\mathsf{A} \parallel \widetilde{\mathsf{B}} \models \varphi \qquad \mathsf{B} \models \widetilde{\mathsf{B}}}{\mathsf{A} \parallel \mathsf{B} \models \varphi}$$

(a) AGR proof rule



(b) TL^* and Teacher

Fig. 2: AGR proof rule (left) and TL^* (right)

5 Compositional parameter synthesis for PERAs

Figure 2a recalls the common proof rule used in Assume-Guarantee Reasoning (AGR), which is one of the compositional verification techniques. Given two components A, B and a safety property φ, the proof rule tells us that if A can satisfy the property φ under an assumption $\widetilde{\mathsf{B}}$ and B can guarantee this assumption $\widetilde{\mathsf{B}}$, then we can conclude that $\mathsf{A} \parallel \mathsf{B}$ satisfies φ. As one can observe that the proof rule decomposes one model checking problem ($\mathsf{A} \parallel \mathsf{B} \models \varphi$) into two sub problems ($\mathsf{A} \parallel \widetilde{\mathsf{B}} \models \varphi$ and $\mathsf{B} \models \widetilde{\mathsf{B}}$). The following subsections introduce how this proof rule is utilized in our approach.

5.1 Partitioning the system

The proof rule is presented in the context of two components. If a system consists of more than two components, an intuitive way is to partition the components into two groups to fit the proof rule. For example, if we have four components M_1, M_2, M_3, and M_4, we could partition them as $\mathsf{A} = M_1 \parallel M_2$ and $\mathsf{B} = M_3 \parallel M_4$. However, the number of possible partitions is exponential to the number of components. In addition, an investigation [CAC08] showed that a good partition is very critical to AGR because it affects the verification performance significantly. In this work, we adopt the following heuristics:

1. If a component has timing parameters, it is collected in group A;
2. If a component shares common action labels with the property, the component is collected in group A.

Other components are collected in group B.

Heuristics 1 is required for our approach to be sound. Concerning heuristics 2, in AGR, the ideal case is when A satisfies the property with the weakest assumption $\widetilde{\mathsf{B}}$ that allows everything, i. e., A itself is sufficient to prove the property no matter how B behaves. Based on this observation, the rationale behind heuristics 2 is that if a component shares common action labels with the property, it is very likely to be necessary to prove the property. Thus, we collect it in group A. We will show that heuristics 2 indeed yields good performance in practice.

5.2 Computing an abstraction via learning

Although the proof rule provides a way to decompose the verification problem, we still have one critical issue: where does the assumption $\widetilde{\mathsf{B}}$ come from? Let us explain how to automatically generate the assumption $\widetilde{\mathsf{B}}$ by learning for non-parametric timed systems. We adopt the TL^* algorithm [LAL+14], which is a learning algorithm to infer ERAs. We briefly introduce the TL^* algorithm here. Interested readers are referred to [LAL+14] for more details. The TL^* algorithm has to interact with a teacher. The interaction between them is shown in Figure 2b. Notice that only the teacher knows about the ERA (say U) to be learned. During the learning process, the TL^* algorithm actively makes two types of queries: membership queries and candidate queries.

A *membership query* asks whether a word σ is accepted by U. The teacher answers "yes" ("no") if σ is accepted (rejected) by U. After several membership queries, the TL^* algorithm constructs a candidate ERA C, and makes a candidate query for it. A *candidate query* asks whether an ERA accepts the same timed language as U. If the teacher answers "yes", then the learning process is finished, and C is the ERA learned by TL^*. If the candidate C accepts more (or less) timed words than U, the teacher answers "no" with a counterexample run ρ. The TL^* algorithm will refine the candidate ERA based on the counterexamples provided by the teacher until the answer to the candidate query is "yes". We omit the detailed learning process of TL^*. See [LAL+14] for details.

To guide the TL^* algorithm for learning the assumption $\widetilde{\mathsf{B}}$, we need a teacher to answer membership and candidate queries. Thanks to the proof rule since it also gives us the hint to answer the two queries. According to the proof rule, as long as the candidate C satisfies the two conditions: *1)* $\mathsf{A} \parallel \mathsf{C} \models \varphi$ and *2)* $\mathsf{B} \models \mathsf{C}$, the answer to the candidate query would be "yes" because the candidate C is sufficient to be the assumption $\widetilde{\mathsf{B}}$ to conclude that $\mathsf{A} \parallel \mathsf{B} \models \varphi$. If C fails to satisfy each condition, the answer to the candidate query is "no". The two condition checkings in Figure 3 ($\mathsf{A} \parallel \mathsf{C} \models \varphi$ and $\mathsf{B} \models \mathsf{C}$) can be done by model checking, and counterexamples given by model checking can also serve as counterexamples to the TL^* algorithm. Figure 3 shows our overall procedure $\mathsf{LearnAbstr}(\mathsf{B}, \mathsf{A}, \varphi)$ that returns either an assumption (denoted by $\mathsf{Abstraction}(\widetilde{\mathsf{B}})$) when it is proved that $\mathsf{A} \parallel \mathsf{B} \models \varphi$ holds, or a counterexample (denoted by $\mathsf{Counterex}(\tau)$) otherwise. $\mathsf{Counterex}$ and $\mathsf{Abstraction}$ are "tags" containing a value, in the spirit of data exchanged in distributed programming or types in functional programming; these tags will be used later on to differentiate between the two kinds of results output by $\mathsf{LearnAbstr}$. Also note that, in our setting, we need a counterexample in the form of a trace τ, which is why $\mathsf{LearnAbstr}$ returns $\mathsf{Counterex}(\mathit{trace}(\rho))$. We omit the technical details of $\mathsf{LearnAbstr}(\mathsf{B}, \mathsf{A}, \varphi)$ here. Interested readers are referred to [LAL+14].

Lemma 2. *Let* A, B *be two ERAs. Assume* $\mathsf{LearnAbstr}(\mathsf{B}, \mathsf{A}, \varphi)$ *terminates with result* $\mathsf{Abstraction}(\widetilde{\mathsf{B}})$. *Then* $\mathsf{A} \parallel \widetilde{\mathsf{B}} \models \varphi$ *and* $\mathsf{A} \parallel \mathsf{B} \models \varphi$.

Algorithm 1: $\mathsf{ReplayTrace}(\mathsf{A}, \tau)$

input : PTA A, finite trace $\tau = e_0, e_1, \cdots e_{n-1}$
output : Constraint over the parameters

```
1 s = s_0
2 for i = 0 to n − 1 do s ← Succ(s, e_i) ;
3 return s↓_P
```

Proof. $\mathsf{Abstraction}(\widetilde{\mathsf{B}})$ is returned only if $\mathsf{A} \parallel \widetilde{\mathsf{B}} \models \varphi$ and $\mathsf{B} \models \widetilde{\mathsf{B}}$. Thus, $\mathsf{A} \parallel \widetilde{\mathsf{B}} \models \varphi$ holds. In addition, according to the proof rule Figure 2a, we can conclude that $\mathsf{A} \parallel \mathsf{B} \models \varphi$.

5.3 Replaying a trace

In this section, we explain how to synthesize the exact set of parameter valuations for which a finite trace belongs to the trace set.

Replaying a trace is close to two undecidable problems for PTAs: *i*) the reachability of a location is undecidable for PTAs [AHV93], and therefore this result trivially extends to the reachability of a single edge; *ii*) the emptiness of the set of valuations for which the set of untimed words is the same as a given valuation is undecidable for PTAs [AM15] (where a proof is provided even for a *unique* untimed word). Nevertheless, computing the set of parameter valuations for which a given *finite* trace belongs to the trace set can be done easily by exploring a small part of the symbolic state space as follows.

We give our procedure $\mathsf{ReplayTrace}(\mathsf{A}, \tau)$ in Algorithm 1. Basically, ReplayTrace computes the symbolic run equivalent to τ, and returns the projection onto P of the last symbolic state of that run. The correctness of ReplayTrace comes from the following results (proved in, e. g., [HRSV02]):

Lemma 3. *Let* A *be a PTA, and let* ρ *be a run of* A *reaching* (l, C)*. Let* v *be a parameter valuation. There exists an equivalent run in* $v(\mathsf{A})$ *iff* $v \models C{\downarrow_P}$*.*

Proof. From [HRSV02, Propositions 3.17 and 3.18].

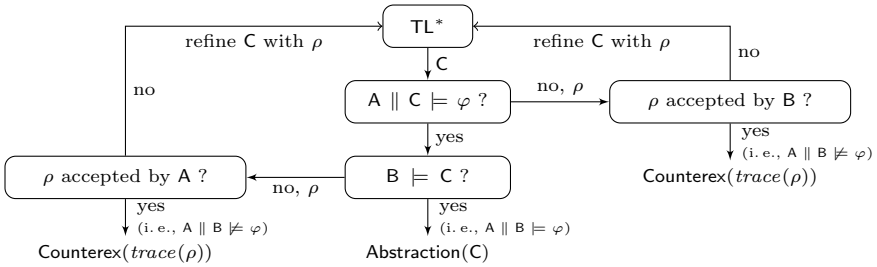


Fig. 3: $\mathsf{LearnAbstr}(\mathsf{B}, \mathsf{A}, \varphi)$

Algorithm 2: $\mathsf{CompSynth}(\mathsf{A}, \mathsf{B}, D_0, L^{☹})$

input : PERA A, ERA B, parameter domain D_0, subset $L^{☹}$ of locations
output : Good and bad constraint over the parameters

1 $K_{bad} \leftarrow \bot$; $K_{good} \leftarrow \bot$
2 **while** $D_0 \cap \mathbb{N} \cap (K_{bad} \cup K_{good}) \neq \emptyset$ **do**
3 Pick v in $D_0 \cap \mathbb{N} \cap (K_{bad} \cup K_{good})$
4 **switch** $\mathsf{LearnAbstr}(\mathsf{B}, v(\mathsf{A}), AG\neg L^{☹})$ **do**
5 **case** $\mathsf{Abstraction}(\widetilde{\mathsf{B}})$
6 $K_{good} \leftarrow K_{good} \cup \mathsf{PRP}(\mathsf{A} \parallel \widetilde{\mathsf{B}}, v, L^{☹})$
7 **case** $\mathsf{Counterex}(\tau)$
8 $K_{bad} \leftarrow K_{bad} \cup \mathsf{ReplayTrace}(\mathsf{A} \parallel \mathsf{B}, \tau)$
9 **return** (K_{good}, K_{bad})

Lemma 4. *Let* A *be a PTA, let* v *be a parameter valuation. Let* ρ *be a run of* $v(\mathsf{A})$ *reaching* (l, μ).

Then there exists an equivalent symbolic run in A *reaching* (l, C), *with* $v \models C{\downarrow_P}$.

Proof. From [HRSV02, Proposition 3.18].

Proposition 1. *Let* A *be a PTA, let* τ *a trace of* $v_0(\mathsf{A})$ *for some* v_0. *Let* $K = \mathsf{ReplayTrace}(\mathsf{A}, \tau)$.

Then, for all v, τ *is a trace of* $v(\mathsf{A})$ *iff* $v \models K$.

Proof. τ is a trace of $v_0(\mathsf{A})$ for some v_0, and therefore it corresponds to some run ρ of $v_0(\mathsf{A})$. Then from Lemma 4 there exists an equivalent symbolic run in A reaching (l, C), with $v_0 \models C{\downarrow_P}$. Now, from Lemma 3, for all v, there exists an equivalent run in $v(\mathsf{A})$ iff $v \models C{\downarrow_P}$. As $\mathsf{ReplayTrace}(\mathsf{A}, \tau)$ returns exactly $K = C{\downarrow_P}$ therefore τ is a trace of $v(\mathsf{A})$ iff $v \models K$.

5.4 Exploiting the abstraction and performing parameter synthesis

We give our procedure in Algorithm 2: it takes as arguments a set of PERA components A, a set of ERA components B, a bounded parameter domain D_0 and a set of locations to be avoided. We maintain a safe non-convex parameter constraint K_{good} and an unsafe non-convex parameter constraint K_{bad}, both initially containing no valuations (line 1). Then $\mathsf{CompSynth}$ iterates on integer points: while not all integer points in D_0 are covered, i. e., do not belong to $K_{bad} \cup K_{good}$ (line 2), such an uncovered point v is picked (line 3). Then, we try to learn an abstraction of B w.r.t. $v(\mathsf{A})$ (line 5) so that $L^{☹}$ is unreachable ("$AG\neg L^{☹}$" stands for "no run should ever reach $L^{☹}$"). If an abstraction is successfully learned, then PRP is called on v and the abstract model $\mathsf{A} \parallel \widetilde{\mathsf{B}}$ (line 6); the constraint K_{good} is then refined. Note that K_{good} is refined because, if an abstraction is computed, then necessarily the property is satisfied and therefore

the (abstract) system is safe. Alternatively, if LearnAbstr fails to compute a valid abstraction, then a counterexample trace τ is returned (line 7); then this trace is replayed using ReplayTrace (line 8), and the constraint K_{bad} is updated.

5.5 Soundness

Proposition 2 (soundness). *Let* $\mathsf{A} \parallel \mathsf{B}$ *be a PERA and* D_0 *be a bounded parameter domain. Assume* $\mathsf{CompSynth}(\mathsf{A}, \mathsf{B}, D_0, L^{\circledcirc})$ *terminates with result* (K_{good}, K_{bad}).

Then, for all v *i) if* $v \models K_{good}$ *then* $v(\mathsf{A} \parallel \mathsf{B})$ *does not reach* $L^{\circledcirc}$*; ii) if* $v \models K_{bad}$ *then* $v(\mathsf{A} \parallel \mathsf{B})$ *reaches* $L^{\circledcirc}$.

Proof. Let v be a parameter valuation.

i) Assume $v \models K_{good}$. From Algorithm 2, K_{good} is a finite union of convex constraints, each of them being the result of a call to PRP. Necessarily, $v \models K$, where K is one of these convex constraints, resulting from a call to $(\mathsf{A} \parallel \widetilde{\mathsf{B}}, v')$, for some v'. From Lemma 2, $v'(\mathsf{A}) \parallel \widetilde{\mathsf{B}} \models (AG\neg L^{\circledcirc})$. Since B and $\widetilde{\mathsf{B}}$ are non-parametric, we can write $v'(\mathsf{A} \parallel \widetilde{\mathsf{B}}) \models (AG\neg L^{\circledcirc})$, i. e., $v'(\mathsf{A} \parallel \widetilde{\mathsf{B}})$ does not reach $L^{\circledcirc}$. From Lemma 1, for all $v'' \models K$, $v''(\mathsf{A} \parallel \widetilde{\mathsf{B}})$ does not reach $L^{\circledcirc}$. Now, since $\widetilde{\mathsf{B}}$ is a valid abstraction of B (i. e., $\mathsf{B} \models \widetilde{\mathsf{B}}$), therefore $\widetilde{\mathsf{B}}$ contains more behaviors than B. Therefore for all $v'' \models K$, $v''(\mathsf{A} \parallel \mathsf{B})$ does not reach $L^{\circledcirc}$ either. Since $v \models K$, therefore $v(\mathsf{A} \parallel \mathsf{B})$ does not reach $L^{\circledcirc}$.

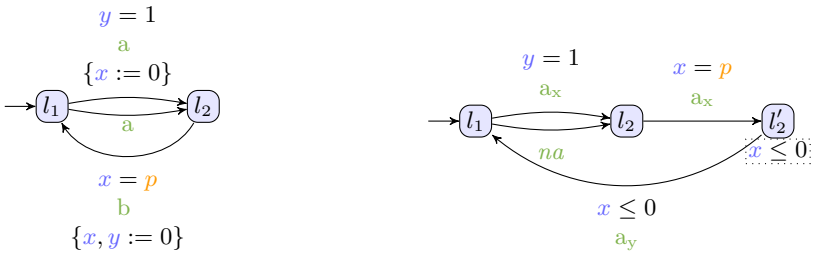
ii) Assume $v \models K_{bad}$. From Algorithm 2, K_{bad} is a finite union of convex constraints, each of them being the result of a call to ReplayTrace. Necessarily, $v \models K$, where K is one of these convex constraints, resulting from a call to $\mathsf{ReplayTrace}(\mathsf{A} \parallel \mathsf{B}, \tau)$ for some trace τ reaching $L^{\circledcirc}$. This trace was generated by LearnAbstr for some v' and is a valid counter-example, i. e., this trace τ reaches $L^{\circledcirc}$ in $v'(\mathsf{A}) \parallel \mathsf{B}$. From Lemma 4, this trace is also a trace reaching $L^{\circledcirc}$ in $\mathsf{A} \parallel \mathsf{B}$. Then, from Proposition 1, for all $v'' \models K$, τ is a valid trace of $v''(\mathsf{A} \parallel \mathsf{B})$ which reaches $L^{\circledcirc}$ and therefore $v''(\mathsf{A} \parallel \mathsf{B})$ reaches $L^{\circledcirc}$. Since $v \models K$, then $v(\mathsf{A} \parallel \mathsf{B})$ reaches $L^{\circledcirc}$.

Proposition 3 (integer-completeness). *Let* A *be a PERA and* D_0 *be a bounded parameter domain. Assume* $\mathsf{CompSynth}(\mathsf{A}, \mathsf{B}, D_0, L^{\circledcirc})$ *terminates with result* (K_{good}, K_{bad}).

Then, for all $v \in D_0 \cap \mathbb{N}$, $v \in K_{good} \cup K_{bad}$.

Proof. From the fact that Algorithm 2 only terminates after all integer points are covered (line 2).

Remark 1. Note that the integerness can be scaled down to, e. g., multiples of 0.1, or in fact arbitrarily small numbers. The time needed to perform the verification might grow, but the coverage of all these discrete points is still guaranteed.



(a) A PTA

(b) Translation to a PERA

Fig. 4: General PTA and its translation to a PERA

6 Experiments

6.1 Handling general PTAs

So far, we showed that our framework is sound for PERAs. We now show that, since we address only reachability, any PTA can be transformed into an equivalent PERA, and therefore our framework is much more general. The idea is that, since we are interested in reachability properties, we can rename some of the actions so that the PTA becomes a PERA.

Basically, we remove any action labels along the edges, and we add them back as follows: *1)* if clock x is reset along an edge, the action label will be a_x; *2)* if no clock is reset along an edge, the action label will be na, where na is a (unique) label, the clock associated to which (say x_{na}) is never used (in guards and invariants) in the PERA; note that, by definition, x_{na} is reset along each edge labeled with na (although this has no impact in the PERA); *3)* if more than one clock is reset along the edge, we split the edge into 2 consecutive edges in 0-time, where each clock is reset after the other, following the mechanism described above. Note that the 0-time can be ensured using an invariant $x \leq 0$, where x is the first clock to be reset.

Basically, our transformation leaves the structure of the PTA unchanged (with the exception of a few consecutive transitions in 0-time to simulate multiple simultaneous clock resets). For each parameter valuation, the resulting PERA has the same timed language as the original PTA – up to action renaming and with the introduction of some 0-time transitions (that could be considered as silent transitions if the language really mattered). Therefore, reachability is preserved.

Note that this construction provides an alternative proof for Theorem 1.

Example 2. Figure 4a shows a PTA, and Figure 4b its translation into an equivalent PERA. (Recall that clock resets are implicit in PERAs.)

Remark 2. In our benchmarks, although we only address reachability, action labels are not entirely useless: they are often used for action synchronization between components. Therefore, renaming all actions is not a valid transformation,

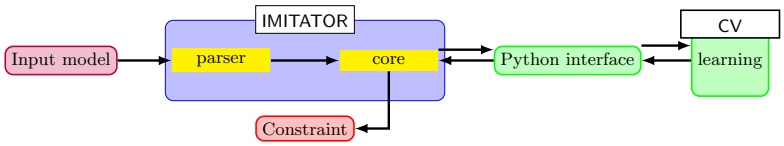


Fig. 5: Architecture of our toolkit

as components may not synchronize anymore the way it was expected. In fact, we ensured that our models either only work using interleaving (no action synchronization) or, when various components of a PTA synchronize on an action label, at most one clock is reset along that transition for all PTAs synchronizing on this action label.

6.2 Experiments

Experimental environment and software We implemented our method in a toolkit made of the following components:

- IMITATOR [AFKS12] is a state of the art tool for modeling and verifying real-time systems modeled by IMITATOR parametric timed automata (IPTAs), an extension of PTAs with stopwatches, broadcast synchronization and integer-valued shared variables. IMITATOR is implemented in OCaml, and the polyhedra operations rely on the Parma Polyhedra Library (PPL) [BHZ08].
- CV (Compositional Verifier) is a prototype implementation (in C++) of the proposed learning-based compositional verification framework for ERAs.

The architecture of our toolkit is shown in Figure 5. The leading tool is IMITATOR, that takes the input model (in the IMITATOR input format), and eventually outputs the result. IMITATOR implements both algorithms CompSynth (Algorithm 2) and ReplayTrace (Algorithm 1), while CV implements LearnAbstr (Section 5.2). The interface between both tools is handled by a Python script, that is responsible for retrieving the abstraction of B computed by CV and re-parameterizing the components A. The whole toolkit can be installed easily: both IMITATOR and CV have standalone binaries for Linux, and therefore only the installation of Python (for the interface script) is required.

We used IMITATOR 2.9-alpha1, build 2212.[3] Experiments were run on a MacBook Pro with an i7 CPU 2.67GHz and 3,7 GiB memory running Kubuntu 14.04 64 bits.

Benchmarks We evaluated our approach using several benchmarks, with various (reachability) properties. We give in Table 1 the case studies, with the numbers of PERAs in parallel, of clocks (equal to the number of actions, by definition) and of parameters, followed by the specification number; then, we

[3] Sources, binaries, models and all results are available at https://www.imitator.fr/static/FORTE17.

Case study	#A	$\#X$	$\#P$	Spec	EFsynth	PRPC		CompSynth			
						#iter	total	#abs	#c.-ex.	learning	total
FMS-1	6	18	2	1	0.299	2	0.654	1	1	0.074	**0.136**
				2	**0.010**	1	0.372	0	1	0.038	0.046
				3	0.282	1	0.309	1	0	0.090	**0.242**
FMS-2	11	37	2	1	T.O.	-	T.O.	1	1	84.2	**88.9**
				2	T.O.	-	T.O.	1	0	81.4	**85.2**
				3	**0.051**	-	T.O.	0	2	1.10	2.44
				4	**0.062**	-	T.O.	0	1	1.42	1.53
				5	T.O.	-	T.O.	1	0	31.4	**40.8**
				6	T.O.	-	T.O.	1	0	37.2	**42.4**
AIP	11	46	2	1	0.551	-	T.O.	0	1	0.086	**0.114**
				2	2.11	-	T.O.	0	1	1.22	**1.25**
				3	**3.91**	-	T.O.	0	1	8.50	8.54
				4	**0.235**	-	T.O.	1	1	8.39	8.42
				5	T.O.	-	T.O.	1	0	0.394	**0.871**
				6	T.O.	-	T.O.	1	0	5.32	**9.58**
				7	T.O.	-	T.O.	1	0	1.76	**3.19**
				8	T.O.	-	T.O.	1	0	1.13	**4.35**
				9	T.O.	-	T.O.	1	1	0.762	**1.84**
				10	**0.022**	-	T.O.	0	1	0.072	0.094
Fischer-3	5	12	2		**2.76**	4	14.0	0	1	-	T.O.
Fischer-4	6	16	2		T.O.	-	T.O.	0	1	-	T.O.

Table 1: Experiments: comparison between algorithms

compare the computation time (in seconds) for EFsynth, PRPC, and CompSynth (for which we also give the number of abstractions and counter-examples generated by LearnAbstr, and the learning time required by LearnAbstr). "T.O." denotes a timeout ($> 600\,s$). FMS-1 and -2 are two versions of a flexible manufacturing system [LAL+14] (Figure 1 depicts the conveyor component of FMS-1). AIP is a manufacturing system producing two products from two different materials [LAL+14]. Fischer-3 (resp. 4) is a PERA version of the mutual exclusion protocol with 3 (resp. 4) processes; it was obtained using the transformation in Section 6.1.

Comparison Although reachability synthesis is intractable for PERAs (Theorem 1), CompSynth always terminates for our case studies (except for Fischer, for which the abstraction computation is too slow). In contrast, EFsynth does often not terminate. In addition, CompSynth always gives a complete (dense) result not only within D_0 but in fact in the entire parameter domain ($\mathbb{Q}_+^M$).

First, CompSynth outperforms PRPC for all but one benchmark: this suggest to use CompSynth instead of PRPC in the future.

Second, CompSynth is faster than EFsynth in 13/20 cases. In addition, whereas EFsynth often does not terminate, CompSynth always outputs a result (except for Fischer). In some cases (FMS-2:3, FMS-2:4, AIP:4), EFsynth is much faster because it immediately derives a false constraint $\bot$, whereas CompSynth has to compute the abstraction first. Even in these unfavorable cases, CompSynth is never much behind EFsynth: the worst case is AIP:4, with 8 seconds slower. This suggests that CompSynth may be preferred to EFsynth for PERAs benchmarks.

Interestingly, in almost all benchmarks, at most one abstraction (for good valuations) and one counter-example (for bad valuations) is necessary for CompSynth. In addition, most of the computation time of CompSynth (71 %

Case study	#A	#X	#P	Spec	D_0	CompSynth				
						#abs	#c.-ex.	find next point	learning	total
FMS-2	11	37	2	1	2,500	1	1	0.0	81.0	85.7
					10,000	1	1	0.1	82.5	87.3
					250,000	1	1	2.2	82.0	89.0
					1,000,000	1	1	8.9	83.1	96.7
					25,000,000	1	1	221.2	83.1	309.0
					100,000,000	1	1	888.1	83.5	976.4

Table 2: Experiments: scalability w.r.t. the reference domain

in average) comes from LearnAbstr; this suggests to concentrate our future optimization efforts on this part. Perhaps an on-the-fly composition mixed with synthesis could help speeding-up this part; this would also solve the issue of false constraints $\bot$ synthesized only after the abstraction phase is completed (FMS-2:3, FMS-2:4, AIP:4).

For Fischer, our algorithm is very inefficient: this comes from the fact that the model is strongly synchronized, and the abstraction computation does not terminate within 600 s. In fact, in both cases, LearnAbstr successfully derives very quickly a counter-example that is used by CompSynth to immediately synthesize all "bad" valuations; but then, as LearnAbstr fails in computing an abstraction, the good valuations are not synthesized. Improving the learning phase for strongly synchronized models is on our agenda.

We were not able to perform a comparison with [ABB+16]; the prototype of [ABB+16] always failed to compute a result (models are available online). In addition, our Fischer benchmark does not fit in [ABB+16] as Fischer makes use of shared parameters.

Size of the parameter domain Algorithm 2 is based on an enumeration of integer points: although we could use an SMT solver to find the next uncovered point, in our implementation we just enumerate *all* points, and therefore the size of D_0 may have an impact on the efficiency of CompSynth. Table 2 shows the impact of the size of D_0 w.r.t. CompSynth. "find next point" is the time to find the next uncovered point (and therefore includes the enumeration of all points). The overhead is reasonable up to 1,000,000 points, but then becomes very significant. Two directions can be taken to overcome this problem for very large parameter domains: *1*) using an SMT solver to find the next uncovered point; or *2*) using an on-the-fly refinement of the precision (e. g., start with multiples of 100, then 10 for uncovered subparts of D_0, then 1... until $D_0 \subseteq K_{bad} \cup K_{good}$).

Partitioning Finally, although the use of heuristic 2 is natural, we still wished to evaluate it. Results show that our partitioning heuristic yields always the best execution time, or almost the best execution time (see Appendix B for details).

7 Conclusion and perspectives

We proposed a learning-based approach to improve the verification of parametric distributed timed systems, that turns to be globally efficient on a set of bench-

marks; most importantly, it outputs an exact result for most cases where the monolithic procedure EFsynth fails.

Among the limitations of our work is that the input model must be a PERA (although we provide an extension to PTAs), and that all parametric ERAs must be in the same component A. How to lift these assumptions is on our agenda.

Another perspective is the theoretical study of PERAs, i. e., their expressiveness and decidability (beyond EF-emptiness, that we proved to be undecidable). As they represent a strict subclass of PTAs, some problems undecidable for PTAs (see [And15]) may become decidable for PERAs. In addition, studying the expressiveness of PERAs (starting with the definition of parametric timed formalisms recently proposed in [ALR16b]) is also of interest.

We would like to combine our approach with the approach based on the integer hull approximation of [ALR15], with a guarantee on the termination and the minimum size of the under-approximated result.

Finally, addressing other properties than reachability is also on our agenda.

Acknowledgment We warmly thank Lăcrămioara Aştefănoaei for her appreciated help with installing and using the prototype tool of [ABB+16].

References

ABB+16. Lăcrămioara Aştefănoaei, Saddek Bensalem, Marius Bozga, Chih-Hong Cheng, and Harald Ruess. Compositional parameter synthesis. In *FM*, volume 9995 of *Lecture Notes in Computer Science*, pages 60–68, 2016.

ACEF09. Étienne André, Thomas Chatain, Emmanuelle Encrenaz, and Laurent Fribourg. An inverse method for parametric timed automata. *International Journal of Foundations of Computer Science*, 20(5):819–836, 2009.

ACN15. Étienne André, Camille Coti, and Hoang Gia Nguyen. Enhanced distributed behavioral cartography of parametric timed automata. In *ICFEM*, Lecture Notes in Computer Science. Springer, 2015.

AFH99. Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata: A determinizable class of timed automata. *Theoretical Computer Science*, 211(1-2):253–273, 1999.

AFKS12. Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In *FM*, volume 7436 of *Lecture Notes in Computer Science*. Springer, 2012.

AHV93. Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. Parametric real-time reasoning. In *STOC*, pages 592–601. ACM, 1993.

ALNS15. Étienne André, Giuseppe Lipari, Hoang Gia Nguyen, and Youcheng Sun. Reachability preservation based parameter synthesis for timed automata. In *NFM*, volume 9058 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2015.

ALR15. Étienne André, Didier Lime, and Olivier H. Roux. Integer-complete synthesis for bounded parametric timed automata. In *RP*, volume 9328 of *Lecture Notes in Computer Science*, pages 7–19. Springer, 2015.

ALR16a. Étienne André, Didier Lime, and Olivier H. Roux. Decision problems for parametric timed automata. In *ICFEM*, volume 10009 of *Lecture Notes in Computer Science*, pages 400–416. Springer, 2016.

ALR16b. Étienne André, Didier Lime, and Olivier H. Roux. On the expressiveness of parametric timed automata. In *FORMATS*, volume 9984 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2016.

AM15. Étienne André and Nicolas Markey. Language preservation problems in parametric timed automata. In *FORMATS*, volume 9268 of *Lecture Notes in Computer Science*, pages 27–43. Springer, 2015.

And15. Étienne André. What's decidable about parametric timed automata? In *FTSCS*, volume 596 of *Communications in Computer and Information Science*, pages 1–17. Springer, 2015.

BHZ08. Roberto Bagnara, Patricia M. Hill, and Enea 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.

CAC08. Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Transactions on Software Engineering and Methodology*, 17(2):7:1–7:52, 2008.

CGP03. Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346, 2003.

CPR08. Alessandro Cimatti, Luigi Palopoli, and Yusi Ramadian. Symbolic computation of schedulability regions using parametric timed automata. In *RTSS*, pages 80–89. IEEE Computer Society, 2008.

FJK08. Goran Frehse, Sumit Kumar Jha, and Bruce H. Krogh. A counterexample-guided approach to parameter synthesis for linear hybrid automata. In *HSCC*, volume 4981 of *Lecture Notes in Computer Science*, 2008.

HRSV02. Thomas Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager. Linear parametric model checking of timed automata. *Journal of Logic and Algebraic Programming*, 52-53:183–220, 2002.

JLR15. Aleksandra Jovanović, Didier Lime, and Olivier H. Roux. Integer parameter synthesis for timed automata. *Transactions on Software Engineering*, 41(5):445–461, 2015.

LAL+14. Shang-Wei Lin, Étienne André, Yang Liu, Jun Sun, and Jin Song Dong. Learning assumptions for compositional verification of timed systems. *Transactions on Software Engineering*, 40(2):137–153, 2014.

Min67. Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., 1967.

A Proof of Theorem 1

Theorem 1 (recalled). *The EF-emptiness problem is undecidable for PERAs, even with bounded parameters.*

Proof. We adapt the proof of [ALR16a, Theorem 1], that we adapt to the case of PERAs. The adaptation mostly consists in renaming clocks and actions from the original proof so that the PTA becomes a PERA. We build a PERA that encodes a 2-counter machine (2CM) [Min67], such that the machine halts iff there exists some valuation of the parameters of the PERA such that it reaches a specific location.

Recall that a 2-counter machine has two non-negative counters C_1 and C_2, a finite number of states and a finite number of transitions, which can be of the form:

- when in state s_i, increment C_k and go to s_j;
- when in state s_i, decrement C_k and go to s_j;
- when in state s_i, if $C_k = 0$ then go to s_j, otherwise block.

The machine starts in state s_0 and halts when it reaches a particular state $l_{\texttt{halt}}$. The halting problem for 2-counter machines is undecidable [Min67].

Given such a machine $\mathcal{M}$, we now provide an encoding as a PERA $\mathsf{A}(\mathcal{M})$: each state s_i of the machine is encoded as a location of the automaton, which we also call s_i.

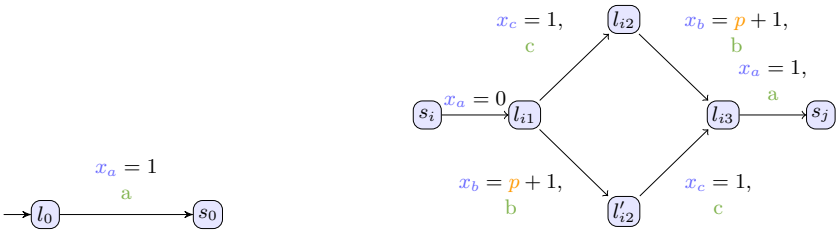
The counters are encoded using clocks x_a, x_b and x_c and one parameter p, with the following relations with the values c_1 and c_2 of counters C_1 and C_2: in any location s_i, when $x_a = 0$, we have $x_b = 1 - pc_1$ and $x_c = 1 - pc_2$. We will see that p is a rational-valued bounded parameter, typically in $[0, 1]$. The main idea of our encoding is that, to correctly simulate the machine, the parameter must be sufficiently small to encode the maximum value of the counters, i. e., for $1 - pc_1$ and $1 - pc_2$ to stay non-negative along the execution of the machine.

We initialize the clocks with the gadget in Figure 6a: note that it implicitly resets clock x_a. Clearly, when in s_0 with $x_a = 0$, we have $x_b = x_c = 1$, which indeed corresponds to counter values 0.

We now present the gadget encoding the increment instruction of C_1 in Figure 6b. The transition from s_i to l_{i1} only serves to clearly indicate the entry in the increment gadget and is done in 0 time unit. Note that it features no action label; one could use an extra label d, the clock of which is never used in the PERA.

Since we use only equalities, there are really only two paths that go through the gadget: one going through l_{i2} and one through l'_{i2}. Let us begin with the former.

We start from some encoding configuration: $x_a = 0$, $x_b = 1 - pc_1$ and $x_c = 1 - pc_2$ in s_i (and therefore the same in l_{i1}). We can enter l_{i2} (after elapsing enough time) if $1 - pc_2 \leq 1$, i. e., $pc_2 \geq 0$, which implies that $p \geq 0$, and when entering l_{i2} we have $x_a = pc_2$, $x_b = 1 - ac_1 + ac_2$ and $x_c = 0$. Then we can enter



(a) EF-emptiness: initial gadget

(b) EF-emptiness: increment gadget

Fig. 6: EF-emptiness: gadgets

l_{i3} if $1 - pc_1 + pc_2 \leq 1 + p$, i. e., $p(c_1 + 1) \geq pc_2$. When entering l_{i3}, we then have $x_a = p(c_1 + 1)$, $x_b = 0$ and $x_c = p(c_1 + 1) - pc_2$. Finally, we can go to s_j if $p(c_1 + 1) \leq 1$ and when entering s_j we have $x_a = 0$, $x_b = 1 - p(c_1 + 1)$ and $x_c = 1 - pc_2$, as expected.

We now examine the second path. We can enter l'_{i2} if $1 - pc_1 \leq p + 1$, i. e., $p(c_1 + 1) \geq 0$, and when entering l'_{i2} we have $x_a = p(c_1 + 1)$, $x_b = 0$ and $x_c = 1 - pc_2 + p(c_1 + 1)$. Then we can go to l_{i3} if $1 - pc_2 + p(c_1 + 1) \leq 1 + p$, i. e., $p(c_1 + 1) \leq pc_2$. When entering l_{i3}, we then have $x_a = pc_2$, $x_b = pc_2 - p(c_1 + 1)$ and $x_c = 0$. Finally, we can go to s_j if $pc_2 \leq 1$ and when entering s_j we have $x_a = 0$, $x_b = 1 - p(c_1 + 1)$ and $x_c = 1 - pc_2$, as expected.

Remark that exactly one path can be taken depending on the respective order of $c_1 + 1$ and c_2, except when both are equal or $p = 0$, in which cases both paths lead to the same configuration anyway.

Decrement is done similarly by replacing guards $x_b = p + 1$ with $x_b = 1$, and guards $x_a = 1$ and $x_c = 1$ with $x_a = p + 1$ and $x_c = p + 1$, respectively.

From s_i, to encode zero-testing C_1 and going to s_j, we only need to add a transition from s_i to s_j with guard $x_b = 1 \wedge x_a = 0$.

All those gadgets also work for C_2 by swapping x_b and x_c.

Finally, we add another location $l'_{\texttt{halt}}$ and a transition from $l_{\texttt{halt}}$ to $l'_{\texttt{halt}}$ with guard $0 < x_a < 1$ and $x_a = p$. This implies the constraint $0 < p < 1$ when reaching $l'_{\texttt{halt}}$. This is important, in order to remove the $p = 0$ value, which does not encode the counters properly. (Note that we could also do this as early as the initialization gadget.)

Let us now prove that the machine halts iff there exists a parameter valuation v such that $v(\mathsf{A})$ can reach $l'_{\texttt{halt}}$. Consider two cases:

1. Either the machine halts, then the automaton can go into the $l'_{\texttt{halt}}$ location, with constraints $0 < p < 1$ and, if c is the maximum value of both C_1 and C_2 over the (necessarily finite) halting run of the machine, and if $c > 0$, then $p \leq \frac{1}{c}$. The set of such valuations for p is certainly non-empty: $p = \frac{1}{2}$ belongs to it if $c = 0$ and $p = \frac{1}{c}$ does otherwise;
2. Or the machine does not halt. There are two subcases:
   (a) either the counters stay bounded. Let c be their maximal value. As before, if $c = 0$ and $0 < p \leq 1$ or $c > 0$ and $cp < 1$, then the machine is

Case study	#A	#X	#P	Spec	Partitioning		CompSynth			
					A	B	#abs	#c.-ex.	learning	total
FMS-1	6	18	2	1	CM	R_1R_2A	1	1	1.071	1.137
					CMR_1	R_2A	1	1	0.077	**0.148**
					CMR_2	R_1A	1	1	5.152	5.406
					CMA	R_1R_2	1	1	5.663	5.980
					CMR_1R_2	A	1	1	0.123	0.290
					CMR_1A	R_2	1	1	0.119	0.360
					CMR_2A	R_1	1	1	6.150	6.690
				2	CM	R_1R_2A	0	1	0.133	0.149
					CMR_1	R_2A	0	2	0.077	0.123
					CMR_2	R_1A	0	1	0.040	0.056
					CMA	R_1R_2	0	1	0.824	0.842
					CMR_1R_2	A	0	1	0.034	**0.044**
					CMR_1A	R_2	0	2	0.096	0.144
					CMR_2A	R_1	0	1	0.042	0.051
				3	CM	R_1R_2A	1	0	0.211	0.270
					CMR_1	R_2A	1	0	0.082	**0.186**
					CMR_2	R_1A	1	0	1.094	1.208
					CMA	R_1R_2	1	0	0.729	0.881
					CMR_1R_2	A	1	0	0.119	0.279
					CMR_1A	R_2	1	0	0.314	0.634
					CMR_2A	R_1	1	0	0.104	0.257

Table 3: Experiments: partitioning into components

correctly encoded and the PTA cannot reach $l'_{\texttt{halt}}$. Otherwise, at some point during an incrementation of, say, C_1 we will have $p(c_1 + 1) > 1$ when taking the transition from l_{i2} to l_{i3} and the PTA will be blocked;

(b) or one of the counters is not bounded, say C_1. Then whatever the value of $p > 0$, we have the same situation as in the previous item: the automaton blocks during some incrementation.

In both subcases, the automaton cannot reach the $l'_{\texttt{halt}}$ location and the set of parameters such that it does is obviously empty.

B Additional experiments: partitioning heuristics

We evaluated the validity of heuristic 2 by performing additional experiments. For all situations, our partitioning heuristics yields either the fastest computation, or almost the fastest.

Table 3 presents the FMS-1 case study, and considers all three specifications, with all possible partitions. The 5 components are denoted by C (conveyor 1), M (mill), R_1 (robot 1), R_2 (robot 2) and A (assembly station). The 6th component is the specification, and is handled separately (it plays the role of φ in our framework). Components C and M contain parameters and must be therefore placed in A (according to heuristic 1). Recall that this is strong an assumption in our approach, that cannot be (easily) lifted. Let us now evaluate all possible repartitions of the other components into A and B. This gives 7 combinations; the combinations are relatively limited, as B must not be empty, and C and M cannot be moved.

In Table 3, we highlight with a yellow background the combination corresponding to heuristic 2. We highlight in bold with a green background the best

execution time. In 2 cases, heuristic 2 yields the best result; in the 3rd case, heuristic 2 yields the second best result. Similar experiments were performed for the other case studies (though not exhaustive, due to combinatorial explosion of the possible partitions), and led to the same results.

Finally note that computation time of EFsynth and PRPC is not impacted by partitioning as these algorithms do not rely on compositional verification.