Repairing Timed Automata Clock Guards through Abstraction and Testing[⋆]

Étienne André$^{1,2,3[0000-0001-8473-9555]}$, Paolo Arcaini$^{3[0000-0002-6253-4062]}$, Angelo Gargantini$^{4[0000-0002-4035-0131]}$, and Marco Radavelli$^{4[0000-0002-1165-9981]}$

[1] Université Paris 13, LIPN, CNRS, UMR 7030, F-93430, Villetaneuse, France
[2] JFLI, CNRS, Tokyo, Japan
[3] National Institute of Informatics, Tokyo, Japan
[4] University of Bergamo, Bergamo, Italy

Abstract. Timed automata (TAs) are a widely used formalism to specify systems having temporal requirements. However, exactly specifying the system may be difficult, as the user may not know the exact clock constraints triggering state transitions. In this work, we assume the user already specified a TA, and (s)he wants to validate it against an oracle that can be queried for acceptance. Under the assumption that the user only wrote wrong guard transitions (i. e., the structure of the TA is correct), the search space for the correct TA can be represented by a Parametric Timed Automaton (PTA), i. e., a TA in which some constants are parametrized. The paper presents a process that (i) abstracts the initial (faulty) TA ta_{init} in a PTA pta; (ii) generates some test data (i. e., timed traces) from pta; (iii) assesses the correct evaluation of the traces with the oracle; (iv) uses the IMITATOR tool for synthesizing some constraints φ on the parameters of pta; (v) instantiate from φ a TA ta_{rep} as final repaired model. Experiments show that the approach is successfully able to partially repair the initial design of the user.

1 Introduction

Timed automata (TA) [4] represent a widely used formalism for modeling and verifying concurrent timed systems. A common usage is to develop a TA describing the running system and then apply analysis techniques to it (e. g., [15]). However, exactly specifying the system under analysis may be difficult, as the user may not know the exact clock constraints that trigger state transitions, or may perform errors at design time. Therefore, validating the produced TA against the real system is extremely important to be sure that we are analyzing a faithful representation of the system. Different testing techniques have been proposed for timed automata, based on different coverage criteria as, e. g., transition coverage [25] and fault-based coverage [2,3], and they can be used for TA

⋆ This work is partially supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST and by the ANR national research program PACS (ANR-14-CE28-0002).

validation. However, once some failing tests have been identified, it remains the problem of detecting and removing (*repair*) the fault from the TA under validation. How to do this in an automatic way is challenging. One possible solution could be to use mutation-based approaches [2,3] in which mutants are considered as possible repaired versions of the original TA; however, due to the continuous nature of timed automata, the number of possible mutants (i. e., repair actions) is too big also for small TAs and, therefore, such approaches do not appear to be feasible. We here propose to use a *symbolic representation* of the possible repaired TAs and we reduce the problem of repairing to finding an assignment of this symbolic representation.

Contribution In this work, we address the problem of testing/validating TAs under the assumption that only clock guards may be wrong, that is, we assume that the structure (states and transitions) is correct. Moreover, we assume to have an oracle that we can query for acceptance of timed traces, but whose internal structure is unknown: this oracle can be a Web-service, a medical device, a protocol, etc. In order to symbolically represent the search space of possible repaired TAs, we use the formalism of parametric timed automata (PTAs) [6] as an abstraction to represent all possible behaviors under all possible clock guards.

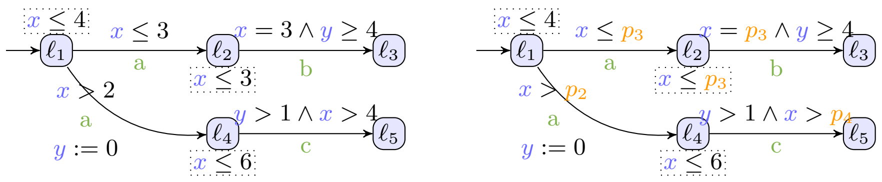
We propose a framework for automatic repair of TAs that takes as input a TA ta_{init} to repair and an *oracle*. The process works as follows: *i*) starting from ta_{init}, we build a PTA *pta* where to look for the repaired TA; *ii*) we build a symbolic representation of the language accepted by *pta* in terms of an *extended parametric zone graph* $\mathcal{EPZG}$; *iii*) we then generate some test data *TD* from $\mathcal{EPZG}$; *iv*) we assess the correct evaluation of *TD* by querying the *oracle*, so building the test suite *TS*; *v*) we feed the tests *TS* to the IMITATOR[10] tool that finds some constraints φ that restrict *pta* only to those TAs that correctly evaluate all the tests in *TS*; *vi*) as the number of TAs that are correct repairs may be infinite, we try to obtain, using a constraint solver based on local search, the TA ta_{rep} closest to the initial TA ta_{init}. Note that trying to modify as less as possible the initial TA is reasonable if we assume the competent programmer hypothesis [22].

To evaluate the feasibility of the approach, we performed some preliminary experiments showing that the approach is able to (partially) repair a faulty TA.

Outline Section 2 explains the definitions we need in our approach. Then Section 3 presents the process we propose that combines model abstraction, test generation, constraint generation, and constraint solving. Section 4 describes experiments we performed to evaluate our process. Finally, Section 5 reviews some related work, and Section 6 concludes the paper.

2 Definitions

A *timed word* [4] over an alphabet of actions Σ is a possibly infinite sequence of the form $(a_0, d_0)(a_1, d_1)\cdots$ such that, for all integer $i \geq 0$, $a_i \in \Sigma$ and $d_i \leq d_{i+1}$. A timed language is a (possibly infinite) set of timed words.



(a) A TA to be repaired (b) An abstract PTA

Fig. 1: Running example

We assume a set $\mathbb{X} = \{x_1, \dots, x_H\}$ of *clocks*, i. e., real-valued variables that evolve at the same rate. A clock valuation is $\mu : \mathbb{X} \to \mathbb{R}_{\geq 0}$. We write $\vec{0}$ for the clock valuation assigning 0 to all clocks. Given $d \in \mathbb{R}_{\geq 0}$, $\mu + d$ is s.t. $(\mu + d)(x) = \mu(x) + d$, for all $x \in \mathbb{X}$. Given $R \subseteq \mathbb{X}$, 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 $\mathbb{P} = \{p_1, \dots, p_M\}$ of *parameters*. A parameter *valuation* v is $v : \mathbb{P} \to \mathbb{Q}_+$. We assume $\bowtie \in \{<, \leq, =, \geq, >\}$. A *clock guard* g is a constraint over $\mathbb{X} \cup \mathbb{P}$ defined by a conjunction of inequalities of the form $x \bowtie \sum_{1 \leq i \leq M} \alpha_i p_i + d$, with $p_i \in \mathbb{P}$, and $\alpha_i, d \in \mathbb{Z}$. Given g, we write $\mu \models v(g)$ if the expression obtained by replacing each x with $\mu(x)$ and each p with $v(p)$ in g evaluates to true.

2.1 Parametric timed automata

Definition 1 (PTA). *A PTA $\mathcal{A}$ is a tuple $\mathcal{A} = (\Sigma, L, \ell_0, F, \mathbb{X}, \mathbb{P}, I, E)$, where: i) Σ is a finite set of actions, ii) L is a finite set of locations, iii) $\ell_0 \in L$ is the initial location, iv) $F \subseteq L$ is the set of accepting locations, v) $\mathbb{X}$ is a finite set of clocks, vi) $\mathbb{P}$ is a finite set of parameters, vii) I is the invariant, assigning to every $\ell \in L$ a clock guard $I(\ell)$, viii) E is a finite set of edges $e = (\ell, g, a, R, \ell')$ where $\ell, \ell' \in L$ are the source and target locations, $a \in \Sigma$, $R \subseteq \mathbb{X}$ is a set of clocks to be reset, and g is a clock guard.*

Given $e = (\ell, g, a, R, \ell')$, we define $\mathsf{Act}(e) = a$.

Example 1. Consider the PTA in Fig. 1b, containing two clocks x and y and three parameters p_2, p_3 and p_4. The initial location is ℓ_1.

Given v, we denote by $v(\mathcal{A})$ the non-parametric structure where all occurrences of a parameter p_i have been replaced by $v(p_i)$. We denote as a *timed automaton* any such structure $v(\mathcal{A})$.

The *synchronous product* (using strong broadcast, i. e., synchronization on a given set of actions), or *parallel composition*, of several PTAs gives a PTA (see [8] for a common formal definition).

Definition 2 (Concrete semantics of a TA). *Given a PTA $\mathcal{A} = (\Sigma, L, \ell_0, F, \mathbb{X}, \mathbb{P}, I, E)$, and a parameter valuation v, the semantics of $v(\mathcal{A})$ is given by the timed transition system (TTS) $(S, s_0, \to)$, with*

– $S = \{(\ell, \mu) \in L \times \mathbb{R}_{\geq 0}^H \mid \mu \models v(I(\ell))\}$, $s_0 = (\ell_0, \vec{0})$,
– $\rightarrow$ *consists of the discrete and (continuous) delay transition relations: i) discrete transitions:* $(\ell, \mu) \overset{e}{\mapsto} (\ell', \mu')$, *if* $(\ell, \mu), (\ell', \mu') \in S$, *and there exists* $e = (\ell, g, a, R, \ell') \in E$, *such that* $\mu' = [\mu]_R$, *and* $\mu \models v(g)$. *ii) delay transitions:* $(\ell, \mu) \overset{d}{\mapsto} (\ell, \mu + d)$, *with* $d \in \mathbb{R}_{\geq 0}$, *if* $\forall d' \in [0, d], (\ell, \mu + d') \in S$.

Moreover we write $(\ell, \mu) \xrightarrow{(e,d)} (\ell', \mu')$ for a combination of a delay and discrete transition if $\exists \mu'' : (\ell, \mu) \overset{d}{\mapsto} (\ell, \mu'') \overset{e}{\mapsto} (\ell', \mu')$.

Given a TA $v(\mathcal{A})$ with concrete semantics $(S, s_0, \rightarrow)$, we refer to the states of S as the *concrete states* of $v(\mathcal{A})$. A *run* of $v(\mathcal{A})$ is an alternating sequence of concrete states of $v(\mathcal{A})$ and pairs of edges and delays starting from the initial state s_0 of the form $s_0, (e_0, d_0), s_1, \cdots$ with $i = 0, 1, \ldots$, $e_i \in E$, $d_i \in \mathbb{R}_{\geq 0}$ and $s_i \xrightarrow{(e_i,d_i)} s_{i+1}$. The *associated timed word* is $(\mathsf{Act}(e_0), d_0)(\mathsf{Act}(e_1), \sum_{0 \leq i \leq 1} d_i) \cdots$. A run is *maximal* if it is infinite or cannot be extended by any discrete action. The (timed) language of a TA, denoted by $\mathcal{L}(v(\mathcal{A}))$, is the set of timed words associated with maximal runs of $v(\mathcal{A})$. Given $s = (\ell, \mu)$, we say that s is reachable in $v(\mathcal{A})$ if s appears in a run of $v(\mathcal{A})$. By extension, we say that ℓ is reachable in $v(\mathcal{A})$; and by extension again, given a set T of locations, we say that T is reachable if there exists $\ell \in T$ such that ℓ is reachable in $v(\mathcal{A})$.

Example 2. Consider the TA $\mathcal{A}$ in Fig. 1a. Consider the following run ρ of $\mathcal{A}$:

$$\left(\ell_1, \begin{pmatrix} x = 0 \\ y = 0 \end{pmatrix}\right) \xrightarrow{(a,2.5)} \left(\ell_4, \begin{pmatrix} x = 2.5 \\ y = 0 \end{pmatrix}\right) \xrightarrow{(c,2)} \left(\ell_5, \begin{pmatrix} x = 4.5 \\ y = 2 \end{pmatrix}\right)$$

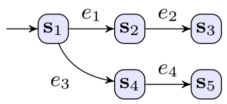
We write "$x = 2.5$" instead of "μ such that $\mu(x) = 2.5$". The associated timed word is $(a, 2.5)(c, 4.5)$.

2.2 Symbolic semantics

Let us now recall the symbolic semantics of PTAs (see e. g., [18,9,19]).

Constraints We first need to define operations on constraints. A linear term over $\mathbb{X} \cup \mathbb{P}$ is 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 $x_i \in \mathbb{X}$, $p_j \in \mathbb{P}$, and $\alpha_i, \beta_j, d \in \mathbb{Z}$. A *constraint* C (i. e., a convex polyhedron) over $\mathbb{X} \cup \mathbb{P}$ is a conjunction of inequalities of the form $lt \bowtie 0$, where lt is a linear term.

Given a parameter valuation v, $v(C)$ denotes the constraint over $\mathbb{X}$ obtained by replacing each parameter p in C with $v(p)$. Likewise, given a clock valuation μ, $\mu(v(C))$ denotes the expression obtained by replacing each clock x in $v(C)$ with $\mu(x)$. We say that v *satisfies* C, denoted by $v \models C$, if the set of clock valuations satisfying $v(C)$ is nonempty. Given a parameter valuation v and a clock valuation μ, we denote by $\mu|v$ the valuation over $\mathbb{X} \cup \mathbb{P}$ such that for all clocks x, $\mu|v(x) = \mu(x)$ and for all parameters p, $\mu|v(p) = v(p)$. We use the notation $\mu|v \models C$ to indicate that $\mu(v(C))$ evaluates to true. We say that C is *satisfiable* if $\exists \mu, v$ s.t. $\mu|v \models C$.



$$
\begin{aligned}
\mathbf{s}_1 &= (\ \ell_1\ ,\ 0 \leq x = y \leq 4 \wedge p_2 \geq 0 \wedge p_3 \geq 0 \wedge p_4 \geq 0\ )\\
\mathbf{s}_2 &= (\ \ell_2\ ,\ 0 \leq x = y \leq p_3 \wedge p_2 \geq 0 \wedge p_3 \geq 0 \wedge p_4 \geq 0\ )\\
\mathbf{s}_3 &= (\ \ell_3\ ,\ x = y \geq p_3 \wedge p_2 \geq 0 \wedge p_3 \geq 4 \wedge p_4 \geq 0\ )\\
\mathbf{s}_4 &= (\ \ell_4\ ,\ p_2 < x \leq 6 \wedge y \geq 0 \wedge p_2 < x - y \leq 4 \wedge 4 > p_2 \geq 0 \wedge p_3 \geq 0 \wedge p_4 \geq 0\ )\\
\mathbf{s}_5 &= (\ \ell_5\ ,\ p_2 < x \wedge p_4 < x \wedge y > 1 \wedge p_2 < x - y \leq 4 \wedge 4 > p_2 \geq 0 \wedge p_3 \geq 0 \wedge 6 > p_4 \geq 0\ )
\end{aligned}
$$

Fig. 2: Parametric zone graph of Fig. 1b

We define the *time elapsing* of C, denoted by $C^{\nearrow}$, as the constraint over $\mathbb{X}$ and $\mathbb{P}$ obtained from C by delaying all clocks by an arbitrary amount of time. That is, $\mu'|v \models C^{\nearrow}$ iff $\exists \mu : \mathbb{X} \to \mathbb{R}_+, \exists d \in \mathbb{R}_+$ s.t. $\mu|v \models C \wedge \mu' = \mu + d$. Given $R \subseteq \mathbb{X}$, 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 $C{\downarrow_{\mathbb{P}}}$ the projection of C onto $\mathbb{P}$, i. e., obtained by eliminating the variables not in $\mathbb{P}$ (e. g., using Fourier-Motzkin [24]). $\perp$ denotes the constraint over $\mathbb{P}$ representing the empty set of parameter valuations.

Definition 3 (Symbolic semantics). *Given a PTA $\mathcal{A} = (\Sigma, L, \ell_0, F, \mathbb{X}, \mathbb{P}, I, E)$, the symbolic semantics of $\mathcal{A}$ is the labeled transition system called* parametric zone graph $\mathcal{PZG} = (E, \mathbf{S}, \mathbf{s}_0, \Rightarrow)$, *with*

– $\mathbf{S} = \{(\ell, C) \mid C \subseteq I(\ell)\}$, $\mathbf{s}_0 = \big(\ell_0, (\bigwedge_{1 \leq i \leq H} x_i = 0)^{\nearrow} \wedge I(\ell_0)\big)$, *and*
– $\big((\ell, C), e, (\ell', C')\big) \in \Rightarrow$ *if* $e = (\ell, g, a, R, \ell') \in E$ *and* $C' = \big([(C \wedge g)]_R \wedge I(\ell')\big)^{\nearrow} \wedge I(\ell')$ *with* C' *satisfiable.*

That is, in the parametric zone graph, nodes are symbolic states, and arcs are labeled by *edges* of the original PTA. A symbolic state is a pair (ℓ, C) where $\ell \in L$ is a location, and C its associated constraint. In the successor state computation in Definition 3, the constraint is intersected with the guard, clocks are reset, the resulting constraint is intersected with the target invariant, then time elapsing is applied, and finally intersected again with the target invariant. This graph is (in general) *infinite* and, in contrast to the zone graph of timed automata, no finite abstraction can be built for properties of interest; this can be put in perspective with the fact that most problems are undecidable for PTAs [7].

Example 3. Consider again the PTA $\mathcal{A}$ in Fig. 1b. The parametric zone graph of $\mathcal{A}$ is given in Fig. 2, where e_1 is the edge from ℓ_1 to ℓ_2 in Fig. 1b, e_2 is the edge from ℓ_2 to ℓ_3, e_3 is the edge from ℓ_1 to ℓ_4, and e_4 is the edge from ℓ_4 to ℓ_5. The inequalities of the form $0 \leq x = y \leq 4$ come from the fact that clocks are initially both equal to 0, evolve at the same rate, and are constrained by the invariant.

2.3 Reachability synthesis

We will use reachability synthesis to solve the problem in Section 3. This procedure, called `EFsynth`, takes as input a PTA $\mathcal{A}$ and a set of target locations T, and attempts to synthesize all parameter valuations v for which T is reachable in $v(\mathcal{A})$. $\mathsf{EFsynth}(\mathcal{A}, T)$ was formalized in e. g., [19] and is a procedure that traverses the parametric zone graph of $\mathcal{A}$; $\mathsf{EFsynth}$ may not terminate (because of the infinite nature of the graph), but computes an exact result (sound and complete) if it terminates.

Example 4. Consider again the PTA $\mathcal{A}$ in Fig. 1b. $\mathsf{EFsynth}(\mathcal{A}, \{\ell_5\})$ returns $0 \leq p_2 < 4 \wedge 0 \leq p_4 \leq 6 \wedge p_3 \geq 0$. Intuitively, this corresponds to all parameter constraints in the parametric zone graph in Fig. 2 associated to symbolic states with location ℓ_5 (there is a single such state).

3 A repairing process using abstraction and testing

In this paper, we address the *guard-repair* problem of timed automata. Given a reference TA ta_{init} and an oracle $\mathcal{O}$ knowing an unknown timed language $\mathcal{TL}$, our goal is to modify ("repair") the timing constants in the clock guards of $\mathcal{A}$ such that the repaired automaton matches the timed language $\mathcal{TL}$. The setting assumes that the oracle $\mathcal{O}$ can be queried for acceptance of timed words by $\mathcal{TL}$; that is, $\mathcal{O}$ can decide whether a timed word belongs to $\mathcal{TL}$, but the internal structure of the object leading to $\mathcal{TL}$ (e. g., an unknown timed automaton) is unknown. This setting makes practical sense when testing black-box systems.

guard-repair problem:
INPUT: an initial TA ta_{init}, an unknown timed language $\mathcal{TL}$
PROBLEM: Repair the constants in the clock guards of ta_{init} so as to obtain a TA ta_{rep} such that $\mathcal{L}(ta_{rep}) = \mathcal{TL}$

While the ultimate goal is to solve this problem, in practice the best we can hope for is to be *as close as possible* to the unknown oracle TA, notably due to the undecidability of language equivalence of timed automata [4] (e. g., if $\mathcal{TL}$ was generated by another TA).

3.1 Overview of the method

From now on, we describe the process we propose to automatically repair an initial timed automaton ta_{init}. Fig. 3 describes the approach:

Step ① a PTA pta is generated starting from the initial TA ta_{init}.
Step ② the extended parametric zone graph $\mathcal{EPZG}$ (an extension of $\mathcal{PZG}$) is built.
Step ③ a test generation algorithm generates relevant test data TD from $\mathcal{EPZG}$.
Step ④ TD is evaluated using the oracle, therefore building the test suite TS.
Step ⑤ some constraints φ are generated, restricting pta to the TAs that evaluate correctly the generated tests TS.
Step ⑥ one possible TA satisfying the constraints φ is obtained.

Algorithm 1 formalizes steps ①–⑤ for which we can provide some theoretical guarantees (i. e., the non-emptiness of the returned valuation set, and its

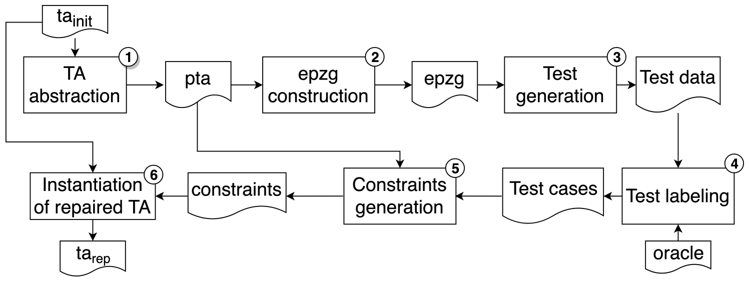


Fig. 3: Automatic repair process

Algorithm 1: Automatic repair process Repair($ta_{init}, \mathcal{O}$)

```
input  : $ta_{init}$: initial timed automaton to repair
input  : $\mathcal{O}$: an oracle assessing the correct evaluation of timed words
output : $\varphi$: set of valuations repairing $ta_{init}$
1 $pta \leftarrow$ AbstractInPta($ta_{init}$)
2 $\mathcal{EPZG} \leftarrow$ BuildEpzg($pta$)
3 $TD \leftarrow$ GenerateTestData($\mathcal{EPZG}$);   /* Generate test data from $\mathcal{EPZG}$  */
4 $TS \leftarrow$ LabelTests($TD, \mathcal{O}$);    /* A test is a pair (trace, assessment) */
5 return $\varphi \leftarrow$ GenConstraints($pta$, $TS$)
```

Function GenConstraints(pta, TS)

```
1 $MBA \leftarrow \{w \mid (w, \text{true}) \in TS\}$;         /* Tests that must be accepted */
2 $MBR \leftarrow \{w \mid (w, \text{false}) \in TS\}$;        /* Tests that must be rejected */
3 return $\bigwedge_{w \in MBA}$ ReplayTW$(pta, w) \wedge \bigwedge_{w \in MBR} \neg$ReplayTW$(pta, w)$
```

inclusion of $\mathcal{TL}$). For step ⑥, instead, different approaches could be adopted: in the paper, we discuss a possible one. We emphasize that, with the exception of Step ①, our process is entirely automated. We describe each phase in details in the following sections.

3.2 Step ①: Abstraction

Starting from the initial ta_{init}, through *abstraction*, the user obtains a PTA pta that generalizes ta_{init} in all the parts that can be possibly changed in order to repair ta_{init} (line 1 in Algorithm 1). For instance, a clock guard with a constant value can be parametrized. Therefore, pta represents the set of all the TAs that can be obtained when repairing ta_{init}. pta is built on the base of the domain knowledge of the developer who has a guess of the guards that may be faulty.

Example 5. Consider again the TA in Fig. 1a. A possible abstraction of this TA is the PTA in Fig. 1b, where we chose to abstract some of the timing constants with parameters. Note that not all timing constants must necessarily be substituted with parameters; also note that a same parameter can be used in different places (this is the case of p_3).

Assumption We define below an important assumption for our method; we will discuss in Section 3.9 how to lift it.

Assumption 1 *We here assume that pta is a correct abstraction, i. e., it contains a TA that precisely models the oracle. That is, there exists $v_{\mathcal{O}}$ such that $\mathcal{L}(v_{\mathcal{O}}(pta)) = \mathcal{TL}$.*

Note that this assumption is trivially valid if faults lay in the clock guards (which is the setting of this work), and if all constants used in clock guards are turned to parameters.

3.3 Step ②: construction of the extended parametric zone graph

Starting from *pta*, we build a useful representation of its computations in terms of an *extended parametric zone graph* $\mathcal{EPZG}$ (line 2 in Algorithm 1). This original data structure will be used for test generation. In the following, we describe how we build $\mathcal{EPZG}$ from $\mathcal{PZG}$. We extend the parametric zone graph $\mathcal{PZG}$ with the two following pieces of information:

the parameter constraint characterizing each symbolic state: from a state (ℓ, C), the parameter constraint is $C{\downarrow_{\mathbb{P}}}$ and gives the exact set of parameter valuations for which there exists an equivalent concrete run in the automaton. That is, a state (ℓ, C) is reachable in $v(\mathcal{A})$ iff $v \models C$ (see [19] for details).

the minimum and maximum arrival times: that is, we compute the minimum (m_i) and maximum (M_i) over all possible parameter valuations of the possible absolute times reaching this symbolic state.

While the construction of the first information is standard, the second one is original to our work and requires more explanation. We build for each state a (possibly unbounded) interval that encodes the absolute minimum and maximum arrival time. This can be easily obtained from the parametric zone graph by adding an extra clock never reset (that encodes the absolute time), and projecting the obtained constrained on this extra clock, thus giving minimum and maximum times over all possible parameter valuations.

Example 6. Consider again the PTA $\mathcal{A}$ in Fig. 1b and its parametric zone graph in Fig. 2. The parameter constraints associated to each of the symbolic states, and the possible absolute reachable times, are given in Table 1.

Remark 1. If all locations of the original PTA contain an invariant with at least one inequality of the form $x \triangleleft p$ or $x \triangleleft d$, with $\triangleleft \in \{<, \leq\}$, and if the parameters are bounded, then the maximum arrival time in each symbolic state will always be finite. Note that this condition is not fulfilled in Example 6 because ℓ_2 features an invariant $x \leq p_3$, with p_3 unbounded, thus allowing to remain arbitrarily long in ℓ_2 for an arbitrarily large value of p_3. Therefore, the arrival time in ℓ_3 is $x_{abs} \in [4, \infty)$.

Table 1: Description of the states of the extended parametric zone graph

Symbolic states	Parameter constraint	Reachable times
$\mathbf{s}_1$	$p_2 \geq 0 \wedge p_3 \geq 0 \wedge p_4 \geq 0$	$x_{abs} = 0$
$\mathbf{s}_2$	$p_2 \geq 0 \wedge p_3 \geq 0 \wedge p_4 \geq 0$	$x_{abs} \in [0, 4]$
$\mathbf{s}_3$	$p_2 \geq 0 \wedge p_3 \geq 4 \wedge p_4 \geq 0$	$x_{abs} \in [4, \infty)$
$\mathbf{s}_4$	$4 > p_2 \geq 0 \wedge p_3 \geq 0 \wedge p_4 \geq 0$	$x_{abs} \in (0, 4]$
$\mathbf{s}_5$	$4 > p_2 \geq 0 \wedge p_3 \geq 0 \wedge 6 > p_4 \geq 0$	$x_{abs} \in (1, 6]$

3.4 Step ③: Test data generation

Starting from $\mathcal{EPZG}$, we generate some test data (line 3 in Algorithm 1) in terms of timed words.

Constructing timed words We use the minimal and maximum arrival times in the abstract PTA to generate test data. That is, we will notably use the *boundary* information, i. e., runs close to the fastest and slowest runs, to try to discover the actual timing guards of the oracle.

The procedure to generate a timed word from the EPZG is as follows:

1. Pick a run $\mathbf{s}_0 e_0 \mathbf{s}_1 \cdots \ldots \mathbf{s}_n$ from $\mathcal{EPZG}$.
2. Construct the timed word $(a_0, d_0)(a_1, d_1) \cdots (a_{n-1}, d_{n-1})$, where $a_i = \mathsf{Act}(e_i)$ and d_i belongs to the interval of reachable times associated with symbolic state $\mathbf{s}_{i+1}$, for $0 \leq i \leq n-1$. Note that, depending on the policy (see below), we sometimes choose on purpose valuations *outside* of the reachable times.

Given an EPZG, we generate, for each finite path of the EPZG up to a given depth K, one timed word. In order to chose a timed word from a (symbolic) path of the EPZG, we identified different policies.

Policies For each $k < K$, we instantiate (a_0, d_0) $(a_1, d_1) \cdots$ (a_k, d_k) by selecting particular values for each d_i using different policies:

- $\mathsf{P}_{\pm 1}$: $d_j \in I_{\pm 1}$, where $I_{\pm 1} = \{m_i - 1, m_i, m_i + 1, M_i - 1, M_i, M_i + 1\}$ and m_i and M_i are the minimum and maximum arrival times of the symbolic state.
- $\mathsf{P}_{\mathsf{minMax2}}$: $d_j \in I_{minMax2}$ with $I_{minMax2} = I_{\pm 1} \cup \{(m_i + M_i)/2\}$.
- $\mathsf{P}_{\mathsf{minMax4}}$: $d_j \in I_{minMax4}$ with $I_{minMax4} = I_{minMax2} \cup \{m_i + (M_i - m_i)/4, m_i + ((M_i - m_i)/4) * 3\}$.
- $\mathsf{P}_{\mathsf{rnd}}$: d_j being a random value such that $m_i \leq d_j \leq M_i$.

Example 7. Consider again the PTA $\mathcal{A}$ in Fig. 1b and its parametric zone graph in Fig. 2 together with reachable times in Table 1. Pick the run $\mathbf{s}_1 e_3 \mathbf{s}_4 e_4 \mathbf{s}_5$. First note that $\mathsf{Act}(e_3) = \mathrm{a}$ and $\mathsf{Act}(e_4) = \mathrm{c}$. According to Table 1, the reachable times associated with $\mathbf{s}_4$ are $(0, 4]$ while those associated with $\mathbf{s}_5$ are $(1, 6]$. Therefore, a possible timed word generated with $\mathsf{P}_{\pm 1}$ is $(\mathrm{a}, 1)(\mathrm{c}, 5)$. Note that this timed word does not belong to the TA to be repaired (Fig. 1a) because of the guard $x > 2$; however, it does belong to an instance TA of Fig. 1b for a sufficiently small value of p_2 (namely $v(p_2) < 1$). We will see later that such tests are tagged as failing.

3.5 Step ④: Test labeling

Then, every test sequence in TD is checked against the oracle in order to label it as accepted or not (line 4 in Algorithm 1), therefore the test suite TS; a test case in TS is a pair $(w, \mathcal{O}(w))$, being w a timed word, and $\mathcal{O}(w)$ the evaluation of the oracle, i. e., $\mathcal{O}(w)$ is defined as a Boolean the value of which is $w \in \mathcal{TL}$.[5] A test case *fails* if $ta_{init}(w) \neq \mathcal{O}(w)$, i. e., the initial TA and the oracle timed language disagree. Note that, if is no test case fails, ta_{init} is considered correct[6] and the process terminates.

In different settings, different oracles can be used. In this work, we assume that the oracle is the real system of which we want to build a faithful representation; the system is black-box, and it can only be queried for acceptance of timed words. In another setting, the oracle could be the user who can easily assess which words should be accepted, and wants to validate their initial design. Of course, the type of oracle also determines how many test data we can provide for assessment: while a real implementation can be queried a lot (modulo the time budget and the execution time of a single query), a human oracle usually can evaluate only few tests.

3.6 Step ⑤: Generating constraints from timed words

Given the test suite TS, our approach generates constraints φ that restrict pta to only those TAs that correctly evaluate the tests (line 5 in Algorithm 1).

In this section, we explain how to "replay a timed word", i. e., given a PTA $\mathcal{A}$, how to synthesize the exact set of parameter valuations v for which a finite timed word belongs to the timed language of $v(\mathcal{A})$. Computing the set of parameter valuations for which a given *finite* timed word belongs to the timed language can be done easily by exploring a small part of the symbolic state space. Replaying a timed word is also very close to the ReplayTrace procedure in [12] where we synthesized valuations corresponding to a trace, i. e., a timed word without the time information—which is decidable.

From timed words to timed automata First, we convert the timed word into a (non-parametric) timed automaton. This straightforward procedure was introduced in [11], and simply consists in converting a timed word of the form $(a_1, d_1), \cdots, (a_n, d_n)$ into a sequence of transitions labeled with a_i and guarded with $x_{abs} = d_i$ (where x_{abs} measures the absolute time, i. e., is an extra clock never reset). Let TW2PTA denote this procedure.[7]

[5] To limit the number of tests, we only keep the maximal accepted traces (i. e., we remove accepted traces included in longer accepted traces), and the minimal rejected traces (i. e., we remove rejected traces having as prefix another rejected trace).

[6] This does not necessarily mean that both TAs have the same language, but that the tests did not exhibit any discrepancy.

[7] This procedure transforms the word to a non-parametric TA; we nevertheless use the name TW2PTA for consistency with [11].

Example 8. Consider again the timed word w mentioned in Example 7: $(a, 0.5)(c, 5)$. The result of $\mathsf{TW2PTA}(w)$ is given in Fig. 4.

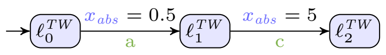


Fig. 4: Translation of timed word $(a, 0.5)(c, 5)$

Synchronized product and synthesis The second part of step ⑤ consists in performing the synchronized product of $\mathsf{TW2PTA}(w)$ and $\mathcal{A}$, and calling $\mathsf{EFsynth}$ on the resulting PTA with the last location of the timed word as the target of $\mathsf{EFsynth}$. Let $\mathsf{ReplayTW}(pta, w)$ denote the entire procedure of synthesizing the valuations associated that make a timed word possible.

Example 9. Consider again the PTA $\mathcal{A}$ in Fig. 1b and the timed word $(a, 0.5)(c, 5)$ translated to a (P)TA in Fig. 4. The result of $\mathsf{EFsynth}$ applied to the synchronized product of these two PTAs with $\{\ell_2^{TW}\}$ as target location set is $0 \leq p_4 < 5 \land 0 \leq p_2 < \frac{1}{2} \land p_3 \geq 0$. This set indeed represents all possible valuations for which $(a, 0.5)(c, 5)$ is a run of the automaton. Note that the result can be non-convex. If we now consider the simpler timed word $(a, 3)$, then the result of $\mathsf{ReplayTW}(\mathcal{A}, w)$ becomes $p_3 \geq 3 \land p_2 \geq 0 \land p_4 \geq 0 \;\lor\; p_2 < 3 \land p_3 \geq 0 \land p_4 \geq 0$ This comes from the fact that the action a can correspond to either e_1 (from ℓ_1 to ℓ_2) or e_3 (from ℓ_1 to ℓ_4) in Fig. 1b.

Remark 2. Despite the non-guarantee of termination of the general $\mathsf{EFsynth}$ procedure, $\mathsf{ReplayTW}$ not only always terminates, but is also very efficient in practice: indeed, it only explores the part of the PTA corresponding to the sequence of (timed) transitions imposed by the timed word. This comes from the fact that we take the synchronized product of $\mathcal{A}$ with $\mathsf{TW2PTA}(w)$, the latter PTA being linear and finite.

Lemma 1. *Let pta be a PTA, and w be a timed word. Then* $\mathsf{ReplayTW}(pta, w)$ *terminates.*

3.7 Correctness

Recall that Assumption 1 assumes that there exists a valuation $v_{\mathcal{O}}$ such that $\mathcal{L}(v_{\mathcal{O}}(pta)) = \mathcal{TL}$. We show that, under Assumption 1, our resulting constraint is always non-empty and contains the valuation $v_{\mathcal{O}}$.

Theorem 1. *Let* $\varphi = \mathsf{Repair}(ta_{init}, \mathcal{O})$. *Then* $\varphi \neq \bot$ *and* $v_{\mathcal{O}} \models \varphi$.

Proof. Proofs of Lemma 1 and Theorem 1 can be found in [8].

3.8 Step ⑥: Instantiation of a repaired TA

Any assignment satisfying φ characterizes a correct TA w.r.t. the generated tests in TS; however, not all of them exactly capture the oracle behaviour. If the user wants to select one TA, (s)he can select one assignment v_{rep} of φ, and use it to instantiate the final repaired TA ta_{rep}.

In order to select one possible assignment v_{rep}, different strategies may be employed, on the base of the assumptions of the process. In this work, we assume the *competent programmer hypothesis* [22] that the developer produced an initial TA ta_{init} close to be correct; therefore, we want to generate a final TA ta_{rep} that is *not too different* from ta_{init}. In particular, we assume that the developer did small mistakes on setting the values of the clock guards.

In order to find the closest values of the clock guards that respect the constraints, we exploit the local search capability of the constraint solver Choco [23]:

1. we start from the observation that ta_{init} is an instantiation of pta. We therefore select the parameter evaluation v_{init} that generates ta_{init} from pta, i. e., $ta_{init} = v_{init}(pta)$;
2. we initialize Choco with v_{init}; Choco then performs a local search trying to find the assignment closest (according to a notion of distance defined later in Section 4) to v_{init}, and that satisfies φ.

3.9 Discussing Assumption 1

Assumption 1 assumes that the user provides a PTA pta that contains the oracle. If this is not the case, the test generation phase (Section 3.6) may generate a negative test (i. e., not accepted by any instance of pta) that is instead accepted by the oracle or a positive test that is not accepted by the oracle; in this case, the constraints generation phase would produce an unsatisfiable constraint φ. In this case, the user should refine the abstraction by parameterizing some other clock guards, or by relaxing the constraints on some existing parameters.

Moreover, it could be that the correct oracle has a different structure (additional states and transitions): as future work, we plan to apply other abstractions as CoPtA models [21] that allow to parametrize states and transitions.

Note that, even if the provided abstraction is wrong, our approach could still be able to refine it. In order to do this, we must avoid to use for constraint generation (step ⑤) tests that produce unsatisfiable constraints. We use a greedy incremental version of GenConstraints in which ReplayTW is called incrementally: if the constraint generated for a test w is not compatible with the other constraints generated previously, then it is discarded; otherwise it is conjuncted.

4 Experimental evaluation

In order to evaluate our approach, we selected some benchmarks from the literature to be used as initial TA ta_{init}: the model of a coffee machine (`CF`) [3], of a car alarm system (`CAS`) [3], and the running case study (`RE`). For each benchmark model, Table 2 reports its number of locations and transitions.

The proposed approach requires that the developer, starting from ta_{init}, provides an abstraction in terms of a PTA pta. For the experiments, as we do not have any domain knowledge, we took the most general case and we built pta by adding a parameter for each guard constant; the only optimization that we did

Table 2: Benchmarks: data

Benchmark	size of ta_{init}		# params	SD	SC (%)
	#locs.	#trans.			
`RunningEx` (`RE`)	5	4	5	2	98.33
`Coffee` (`CF`)	5	7	9	11	99.18
`CarAlarmSystem` (`CAS`)	16	25	10	12	84.24
`RunningEx` – different oracle ($\mathtt{RE}_{do}$)	5	4	5	-	98.72

is to use the same parameter when the same constant is used on entering and/or exiting transitions of the same location (as in Fig. 1b).

In the approach, the oracle should be the real system that we can query for acceptance; in the experiments, the oracle is another TA ta_o that we obtained by slightly changing some constants on the guards. The oracle has been built in a way that it is an instance of pta, following Assumption 1.

In order to measure *how much* a TA (either the initial one ta_{init} or the final one ta_{rep}) is different from the oracle, we introduce a syntactic and a semantic measure, that provide different kinds of comparison with the oracle ta_o.

Given a model ta, the oracle ta_o, and a PTA pta having parameters $p_1, \ldots, p_n$, let v and v_o be the corresponding evaluations, i. e., $ta = v(pta)$ and $ta_o = v_o(pta)$. We define the *syntactic distance* of ta to the oracle as follows:

$$SD(ta) = \sum_{i=1}^{n} |v(p_i) - v_o(p_i)|$$

The syntactic distance roughly measures how much ta must be changed (under the constraints imposed by pta) in order to obtain ta_o.

The *semantic conformance*, instead, tries to assess the distance between the languages accepted by ta and the oracle ta_o. As the set of possible words is infinite, we need to select a representative set of test data TD_{SC}; to this aim, we generate, from ta_{init} and ta_o, sampled test data in the two TAs; moreover, we also add negative tests by extending the positive tests with one forbidden transition at the end. The semantic conformance is defined as follows:

$$SC(ta) = \frac{|\{t \in TD_{SC} | (t \in \mathcal{L}(ta) \wedge t \in \mathcal{L}(ta_o)) \vee (t \notin \mathcal{L}(ta) \wedge t \notin \mathcal{L}(ta_o))\}|}{|TD_{SC}|}$$

Table 2 also reports SD and SC of each benchmark ta_{init}.

Experiments have been executed on a Mac OS X 10.14, Intel Core i3, with 4 GiB of RAM. Code is implemented in Java, IMITATOR 2.11 "Butter Kouign-amann" [10] is used for constraint generation, and Choco 4.10 for constraint solving. The code and the benchmarks are available at `https://github.com/ERATOMMSD/repairTAsThroughAbstraction`.

4.1 Results

Table 3 reports the experimental results. For each benchmark and each test generation policy (see Section 3.4), it reports the execution time (divided between

Table 3: Experimental results

Bench.	Policy	time (s)					# failed tests/	ta_{rep}	
		total	Steps ②-③	Step ④	Step ⑤	Step ⑥	# tests	SD	SC (%)
RE	$\mathsf{P}_{\pm1}$	1.070	0.010	0.008	1.030	0.019	1/ 38	0	100.00
RE	$\mathsf{P}_{\mathsf{minMax2}}$	1.148	0.007	0.006	1.130	0.005	1/ 41	0	100.00
RE	$\mathsf{P}_{\mathsf{minMax4}}$	1.191	0.004	0.004	1.177	0.004	1/ 41	0	100.00
RE	$\mathsf{P}_{\mathsf{rnd}}$	0.006	0.006	0.001	0.000	0.000	0/ 3	2	98.33
CF	$\mathsf{P}_{\pm1}$	25.921	0.050	0.267	25.546	0.045	45/293	8	99.86
CF	$\mathsf{P}_{\mathsf{minMax2}}$	32.717	0.129	0.578	31.845	0.147	62/422	7	100.00
CF	$\mathsf{P}_{\mathsf{minMax4}}$	76.137	0.857	1.907	73.058	0.769	102/737	7	100.00
CF	$\mathsf{P}_{\mathsf{rnd}}$	0.134	0.098	0.035	0.000	0.000	1/ 11	8	99.96
CAS	$\mathsf{P}_{\pm1}$	59.511	0.043	0.160	59.261	0.037	174/392	2	100.00
CAS	$\mathsf{P}_{\mathsf{minMax2}}$	61.791	0.040	0.159	61.544	0.036	199/416	2	100.00
CAS	$\mathsf{P}_{\mathsf{minMax4}}$	68.341	0.716	0.467	67.037	0.584	245/464	2	100.00
CAS	$\mathsf{P}_{\mathsf{rnd}}$	0.024	0.017	0.007	0.000	0.000	0/ 20	12	84.24

the different phases), the total number of generated tests, the number of tests that fail on ta_{init}, and SD and SC of the final TA ta_{rep}.

We now evaluate the approach answering the following research questions.

RQ1: *Is the approach able to repair faulty TAs?*

We evaluate whether the approach is actually able to (partially) repair ta_{init}. From the results, we observe that, in three cases out of four, the process can completely repair RE since SD becomes 0, meaning that we obtain exactly the oracle (therefore, also SC becomes 100%). For CF and CAS, it almost always reduces the syntactical distance SD, but it never finds the exact oracle. On the other hand, the semantic conformance SC is 100% in five cases. Note that SC can be 100% with SD different from 0 for two reasons: either the test data TD_{SC} we are using for SC are not able to show the unconformity, or ta_{rep} is indeed equivalent to the oracle, but with a different structure of the clock guards.

RQ2: *Which is the best test generation strategy?*

In Section 3.4, we proposed different test generation policies over $\mathcal{EPZG}$. We here assess the influence of the generation policy on the final results. $\mathsf{P}_{\pm1}$, $\mathsf{P}_{\mathsf{minMax2}}$, and $\mathsf{P}_{\mathsf{minMax4}}$ obtain the same best results for two benchmarks (RE and CAS), meaning that the most useful tests are those on the boundaries of the clock guards: those are indeed able to expose the failure if the fault is not too large. On the other hand, for CF, $\mathsf{P}_{\pm1}$ performs slightly worse than the other two, meaning that also generating tests inside the intervals (as done by $\mathsf{P}_{\mathsf{minMax2}}$ and $\mathsf{P}_{\mathsf{minMax4}}$) can be beneficial for repair. $\mathsf{P}_{\mathsf{rnd}}$ is able to improve (but not totally repair) only CF; for the other two benchmarks, it is not able to improve neither SD nor SC.

RQ3: *How long does the approach take?*

The time taken by the process depends on the size of ta_{init} and on the test generation policy. The most expensive phase is the generation of the constraints, as it requires to call IMITATOR for each test that must be accepted. As future

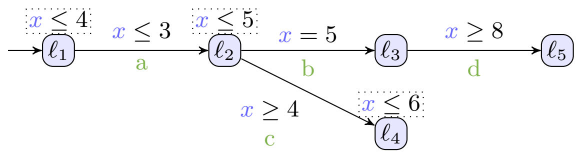


Fig. 5: Repairing TAs with different structures – Another oracle TA

Table 4: Experimental results – Different oracle

Bench.	Policy	time (s)					# failed tests/	ta_{rep}
		total	Steps ②-③	Step ④	Step ⑤	Step ⑥	# tests	SC (%)
$\texttt{RE}_{do}$	$\mathsf{P}_{\pm 1}$	2.083	0.007	0.005	2.055	0.013	10/ 44	98.72
$\texttt{RE}_{do}$	$\mathsf{P}_{\mathsf{minMax2}}$	2.718	0.005	0.004	2.693	0.013	11/ 47	98.72
$\texttt{RE}_{do}$	$\mathsf{P}_{\mathsf{minMax4}}$	2.686	0.004	0.003	2.658	0.012	11/ 47	98.72
$\texttt{RE}_{do}$	$\mathsf{P}_{\mathsf{rnd}}$	0.763	0.007	0.001	0.676	0.075	1/ 3	99.5

work, we plan to optimize this phase by modifying IMITATOR to synthesize valuations guaranteeing the acceptance of multiple timed words in a single analysis. In the experiments, we use as oracle another TA that we can visit for acceptance; this visit is quite fast and so step ④ does not take too much time. However, in the real setting, the oracle is the real system whose invocation time may be not negligible; in that case, the invocation of the oracle could become a bottleneck and we would need to limit the number of generated tests.

RQ4: *Which is the process performance if pta does not include the oracle?*

Assumption 1 assumes that the user provides a PTA that contains the oracle. In Section 3.9, we discussed about the possible consequences when this assumption does not hold. We here evaluate whether the approach is still able to partially repair ta_{init} using an oracle having a different structure. We took the TA shown in Fig. 5 as oracle of the running example, that is structurally different from ta_{init} and pta shown in Fig. 1 (we name this experiment as $\texttt{RE}_{do}$); the semantic conformance SC of ta_{init} w.r.t. the new oracle is shown at the last row of Table 3[8]. We performed the experiments with the new oracle using the greedy approach described in Assumption 1, and results are reported in Table 4. We observe that policies $\mathsf{P}_{\pm 1}$, $\mathsf{P}_{\mathsf{minMax2}}$, and $\mathsf{P}_{\mathsf{minMax4}}$, although they find some failing tests, they are not able to improve SC. This is partially due to the fact that SC is computed on some timed words TD_{SC} that may be not enough to judge the improvement. On the other hand, as the three policies try to achieve a kind of coverage of pta (so implicitly assuming Assumption 1), it could be that they are not able to find *interesting* failing tests (i. e., they cannot be repaired); this seems to be confirmed by the fact that the random policy $\mathsf{P}_{\mathsf{rnd}}$ is instead able to partially repair the initial TA using only three tests, out of which one

[8] Note that it does not make sense to measure the syntactical distance, as the structure of the oracle is different.

fails. We conclude that, if the assumption does not hold, trying to randomly select tests could be more efficient.

5 Related Work

Testing timed automata Works related to ours are approaches for test case generation for timed automata. In [3,1], a fault-based approach is proposed. The authors defined 8 mutation operators for TAs and a test generation technique based on bounded-model checking; tests are then used for model-based testing to check that System Under Test (SUT) is conformant with the specification. Our approach is different, as we aim at building a faithful representation of the SUT (i. e., the oracle). Their mutation operators could be used to repair our initial TA, as done in [14]; however, due to continuous nature of TAs, the possible mutants could be too many. For this reason, our approach symbolically represents all the possible variations of the clock guards (similar to "change guard" mutants in [3]). Other classical test generation approaches for timed automata are presented in [25,17]; while they aim at coverage of a single TA, we aim at coverage of a family of TAs described by *pta*.

Learning timed systems The (timed) language inclusion is undecidable for timed automata [4], making learning impossible for this class. In [16,20], timed extensions of the L^* algorithm [13] were proposed for event-recording automata [5], a subclass of timed automata for which language equivalence can be decided. Learning is essentially different from our setting, as the system to be learned is usually a white-box system, in which the equivalence query can be decided. In our setting, the oracle does not necessarily know the structure of the unknown system, and simply answers membership queries. In addition, we address in our work the full class of timed automata, for which learning is not possible.

6 Conclusion and perspectives

This paper proposes an approach for automatically repairing timed automata, notably in the case where clock guards shall be repaired. Our approach generates an abstraction of the initial TA in terms of a PTA, generates some tests, and then refines the abstraction by identifying only those TAs contained in the PTA that correctly evaluate all the tests.

As future work, we plan to adopt also other formalisms to build the abstraction where to look for the repaired timed automata; The CoPtA model [21], for example, extends timed automata with feature models and allows to specify additional/alternative states and transitions. In addition, when the oracle acts as a white-box, i. e., when the oracle is able to test language equivalence, we could also make use of learning techniques for timed automata, using the often terminating procedure for language inclusion in [26].

References

1. Aichernig, B.K., Hörmaier, K., Lorber, F.: Debugging with timed automata mutations. In: Bondavalli, A., Giandomenico, F.D. (eds.) SAFECOMP. Lecture Notes in Computer Science, vol. 8666, pp. 49–64. Springer (2014). https://doi.org/10.1007/978-3-319-10506-2_4
2. Aichernig, B.K., Jöbstl, E., Tiran, S.: Model-based mutation testing via symbolic refinement checking. Science of Computer Programming **97**(P4), 383–404 (Jan 2015). https://doi.org/10.1016/j.scico.2014.05.004
3. Aichernig, B.K., Lorber, F., Nickovic, D.: Time for mutants – Model-based mutation testing with timed automata. In: Veanes, M., Viganò, L. (eds.) TaP. Lecture Notes in Computer Science, vol. 7942, pp. 20–38. Springer (2013). https://doi.org/10.1007/978-3-642-38916-0_2
4. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science **126**(2), 183–235 (Apr 1994). https://doi.org/10.1016/0304-3975(94)90010-8
5. Alur, R., Fix, L., Henzinger, T.A.: Event-clock automata: A determinizable class of timed automata. Theoretical Computer Science **211**(1-2), 253–273 (1999). https://doi.org/10.1016/S0304-3975(97)00173-4
6. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Kosaraju, S.R., Johnson, D.S., Aggarwal, A. (eds.) STOC. pp. 592–601. ACM, New York, NY, USA (1993). https://doi.org/10.1145/167088.167242
7. André, É.: What's decidable about parametric timed automata? International Journal on Software Tools for Technology Transfer **21**(2), 203–2019 (2019). https://doi.org/10.1007/s10009-017-0467-0, to appear
8. André, É., Arcaini, P., Gargantini, A., Radavelli, M.: Repairing timed automata clock guards through abstraction and testing. arXiv (2019), http://arxiv.org/abs/1907.02133
9. André, É., Chatain, Th., Encrenaz, E., Fribourg, L.: An inverse method for parametric timed automata. International Journal of Foundations of Computer Science **20**(5), 819–836 (Oct 2009). https://doi.org/10.1142/S0129054109006905
10. André, É., Fribourg, L., Kühne, U., Soulat, R.: IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In: Giannakopoulou, D., Méry, D. (eds.) FM. Lecture Notes in Computer Science, vol. 7436, pp. 33–36. Springer (Aug 2012). https://doi.org/10.1007/978-3-642-32759-9_6
11. André, É., Hasuo, I., Waga, M.: Offline timed pattern matching under uncertainty. In: Lin, A.W., Sun, J. (eds.) ICECCS. pp. 10–20. IEEE CPS (2018). https://doi.org/10.1109/ICECCS2018.2018.00010
12. André, É., Lin, S.W.: Learning-based compositional parameter synthesis for event-recording automata. In: Bouajjani, A., Alexandra, S. (eds.) FORTE. Lecture Notes in Computer Science, vol. 10321, pp. 17–32. Springer (2017). https://doi.org/10.1007/978-3-319-60225-7_2
13. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation **75**(2), 87–106 (1987). https://doi.org/10.1016/0890-5401(87)90052-6
14. Arcaini, P., Gargantini, A., Radavelli, M.: Achieving change requirements of feature models by an evolutionary approach. Journal of Systems and Software **150**, 64–76 (2019). https://doi.org/10.1016/j.jss.2019.01.045
15. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets, Advances in Petri Nets. Lecture Notes in Computer Science, vol. 3098, pp. 87–124. Springer (2003). https://doi.org/10.1007/978-3-540-27755-2_3

16. Grinchtein, O., Jonsson, B., Leucker, M.: Learning of event-recording automata. Theoretical Computer Science **411**(47), 4029–4054 (2010). https://doi.org/10.1016/j.tcs.2010.07.008
17. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using UPPAAL. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers, Lecture Notes in Computer Science, vol. 4949, pp. 77–117. Springer (2008). https://doi.org/10.1007/978-3-540-78917-8_3
18. Hune, T., Romijn, J., Stoelinga, M., Vaandrager, F.W.: Linear parametric model checking of timed automata. Journal of Logic and Algebraic Programming **52-53**, 183–220 (2002). https://doi.org/10.1016/S1567-8326(02)00037-1
19. Jovanović, A., Lime, D., Roux, O.H.: Integer parameter synthesis for real-time systems. IEEE Transactions on Software Engineering **41**(5), 445–461 (2015). https://doi.org/10.1109/TSE.2014.2357445
20. Lin, S.W., André, É., Liu, Y., Sun, J., Dong, J.S.: Learning assumptions for compositional verification of timed systems. Transactions on Software Engineering **40**(2), 137–153 (mar 2014). https://doi.org/10.1109/TSE.2013.57
21. Luthmann, L., Gerecht, T., Stephan, A., Bürdek, J., Lochau, M.: Minimum/maximum delay testing of product lines with unbounded parametric real-time constraints. Journal of Systems and Software **149**, 535–553 (2019). https://doi.org/10.1016/j.jss.2018.12.028
22. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M.: Mutation testing advances: An analysis and survey. In: Advances in Computers. Advances in Computers, Elsevier (2018). https://doi.org/10.1016/bs.adcom.2018.03.015
23. Prud'homme, C., Fages, J.G., Lorca, X.: Choco Documentation. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. (2017), http://www.choco-solver.org
24. Schrijver, A.: Theory of linear and integer programming. Wiley-Interscience series in discrete mathematics and optimization, Wiley (1999)
25. Springintveld, J., Vaandrager, F., D'Argenio, P.R.: Testing timed automata. Theoretical Computer Science **254**(1-2), 225–257 (Mar 2001). https://doi.org/10.1016/S0304-3975(99)00134-6
26. Wang, T., Sun, J., Liu, Y., Wang, X., Li, S.: Are timed automata bad for a specification language? Language inclusion checking for timed automata. In: Ábrahám, E., Havelund, K. (eds.) TACAS. Lecture Notes in Computer Science, vol. 8413, pp. 310–325. Springer (2014). https://doi.org/10.1007/978-3-642-54862-8_21