Active learning of timed automata with unobservable resets*

Léo Henry, Thierry Jéron, and Nicolas Markey

Univ Rennes, Inria, CNRS – Rennes, France
firstname.lastname@inria.fr

Abstract. Active learning of timed languages is concerned with the inference of timed automata by observing some of the timed words in their languages. The learner can query for the membership of words in the language, or propose a candidate model and ask if it is equivalent to the target. The major difficulty of this framework is the inference of *clock resets*, which are central to the dynamics of timed automata but not directly observable.
Interesting first steps have already been made by restricting to the subclass of *event-recording automata*, where clock resets are tied to observations. In order to advance towards learning of general timed automata, we generalize this method to a new class, called *reset-free* event-recording automata, where some transitions may reset no clocks.
Central to our contribution is the notion of *invalidity*, and the algorithm and data structures to deal with it, allowing on-the-fly detection and pruning of reset hypotheses that contradict observations. This notion is a key to any efficient active-learning procedure for generic timed automata.

1 Introduction

Active learning [Ang87a] is a type of learning in which a teacher assesses the learner's progress and direct the learning effort toward meaningful decisions. The learner can request information from the teacher via *membership queries*, asking about a specific observation, and *equivalence queries*, proposing to compare the current hypothesis to the correct model; in the latter case, the teacher either accepts the hypothesis or returns a counter-example exemplifying mispredictions of the learner's hypothesis.

This framework is well-studied in the setting of finite-state automata [Ang87a, Ang87b, Ang90], and allows to make sound proofs for both correctness and complexity of learning algorithms. As most real-life systems dispose of *continuous* components, attempts have been made to leverage this framework to take them into account. One of the most classic additions is *time*. An observation is then a timed word, made of actions and delays between them. One of the most recognized models for such timed languages is the timed automaton (TA), but its dynamics are complex: TAs measure time using a set of *clocks* that hold a positive real

* This work was partially funded by ANR project Ticktac (ANR-18-CE40-0015).

value progressing with time, can be compared with integer constants to allow or disallow transitions, and reset to zero along those transitions. For a learning algorithm, one of the main challenges is to deal with those resets, that are typically not observable, but play a central role in the system dynamics.

Some work has already been done in the active learning of subclasses of TAs, mostly deterministic TAs with only one clock [ACZ+20] and deterministic event-recording automata (DERA) [GJP06, Gri08], which have as many clocks as actions in the alphabet, and where each clock encodes exactly the time elapsed since the last corresponding action was taken. These classes of automata present the advantages of having a low-dimensional continuous behaviour (for 1-clock TAs) and to allow to derive the resets of the clocks directly from the observations (for DERA). Other approaches have been investigated for the learning of timed systems. Learning of TAs from tests has been studied using genetic algorithms [TALL19], which is a very different approach to ours to exploit a similar setting. Inference of simple TAs from positive data [VWW08, VWW12] has also been well studied. These works are more loosely related to ours, as our setting greatly differs from positive inference.

We propose in this work to generalize to a class of timed automata enjoying both several clocks and different possible resets that can not be inferred directly from observations. This allows us to design and prove algorithms that handle all the main difficulties that arise in deterministic TAs, making this contribution an important first step towards active learning for generic deterministic TAS.

To our knowledge, the closest works are Grinchtein's thesis on active learning of DERA [Gri08] and the paper proposing to learn one clock TAs [ACZ+20]. The work of Grinchtein *et al.* [GJP06] is the most related to ours, as we use some of the data structures they developed and keep the general approach based on timed decision trees. The main difference between our work and this one is that we handle the inference of resets in a class of models in which they can not be *directly* deduced from observations. The approach reported in [ACZ+20] proposes to deal with reset guessing, but makes it in a somewhat "brute force" manner, by directly applying a branch-and-bound algorithm and jumping from model to model. In order to be able to deal with larger dimensions, *e.g.* to handle TAs with a large set of clocks, we need to be more efficient by exploiting the theory built around TAs and detecting invalid models as early as possible.

For reasons of space, the proofs of our claims and the pseudo-code of our algorithms are left in appendix.

2 Preliminaries

2.1 Timed automata

For the rest of this paper, we fix a finite alphabet Σ.

Let X be a finite set of variables called *clocks*. A valuation for X is a function $v\colon X \to \mathbb{R}_{\geq 0}$. We write $\mathbf{0}$ for the clock valuation associating 0 with all clocks. For any $\delta \in \mathbb{R}_{\geq 0}$ and any valuation v we write $v + d$ for the valuation such that

$(v+\delta)(x) = v(x) + \delta$ for each clock x; this corresponds to elapsing δ time units from valuation v. The future of a valuation v is the set $v^{\nearrow} = \{v + t \mid t \in \mathbb{R}_{\geq 0}\}$ of its time successors. Finally, for any $X' \subseteq X$ and any valuation v, we write $v_{[X' \leftarrow 0]}$ for the valuation such that $v_{[X' \leftarrow 0]}(x) = v(x)$ for all $x \notin X'$ and $v_{[X' \leftarrow 0]}(x) = 0$ for all $x \in X'$.

Simple clock constraints are expressions of the forms $x - x' \sim n$ and $x \sim n$, for $x, x' \in X$, $\sim \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{N}$. We call *zone* over X any finite conjunction of such constraints, and write $\mathcal{Z}_X$ for the set of zones over X. Given a valuation v and a zone z, we write $v \models z$ when v satisfies all the constraints in z. We may identify a zone z with the set of valuations z such that $v \models z$. We call *guard* any zone not involving constraints of the form $x - x' \sim n$, and write $\mathcal{G}_X$ for the set of guards. We extend all three operations on valuations to zones elementwise.

Definition 1. *A* timed automaton *(TA) over Σ is a tuple $\mathcal{T} = (\mathcal{L}, l_0, X, E, \mathsf{Accept})$ such that: $\mathcal{L}$ is a finite set of* locations*, and $l_0 \in \mathcal{L}$ is the* initial location*; X is a finite set of clocks; $\mathsf{Accept} \subseteq \mathcal{L}$ is a set of accepting locations; $E \subseteq \mathcal{L} \times \Sigma \times \mathcal{G}_X \times 2^X \times \mathcal{L}$ is a set of transitions. For a transition (l, a, g, r, l'), we call g its guard, a its action and r its reset.*

We write $K_{\mathcal{T}}$ (or K when the context is clear) for the *maximal constant* appearing in $\mathcal{T}$. We say that a TA is *deterministic* when, for any two transitions (l, a, g, r, l') and (l, a, g', r', l'') where $g \wedge g'$ is satisfiable, it holds $l' = l''$ and $r = r'$. We only consider deterministic TAs in the sequel, as active-learning methods can only target this (strict) subclass of TAs.

Definition 2. *With a TA $\mathcal{T} = (\mathcal{L}, l_0, X, E, \mathsf{Accept})$, we associate the transition system $\mathcal{S}^{\mathcal{T}} = (S = \mathcal{L} \times \mathbb{R}_{\geq 0}^{|X|}, (l_0, \mathbf{0}), \Delta, \mathsf{Accept}_{\mathcal{S}^{\mathcal{T}}})$ where $\mathcal{L} \times \mathbb{R}_{\geq 0}^{|X|}$ is the set of* configurations*, $(l_0, \mathbf{0})$ is the initial configuration, $\mathsf{Accept}_{\mathcal{S}^{\mathcal{T}}} = \{(l, v) \mid l \in \mathsf{Accept}\}$ is the set of accepting configurations, and $\Delta \subset S \times (\mathbb{R}_{\geq 0} \cup E) \times S$ a set of transitions, such that for any $(l, v) \in S$: (a) for any $\delta \in \mathbb{R}_{\geq 0}$, we have $((l, v), \delta, (l, v + \delta))$ in Δ; (b) for any $e = (l, a, g, r, l') \in E$ s.t. $v \models g$, we have $((l, v), e, (l', v_{[r \leftarrow 0]}))$ in Δ.*

A path in a timed automaton $\mathcal{T}$ is a sequence of transitions in the associated transition system $\mathcal{S}^{\mathcal{T}}$. A *timed word with resets* of $\mathcal{T}$ is a path $w_{tr} = ((l_i, v_i) \xrightarrow{e_i} (l_{i+1}, v_{i+1}))_{i \in [0,n]} \in (S \times (\Delta \cup \mathbb{R}_{\geq 0}))^* \times S$ of its semantics $\mathcal{S}^{\mathcal{T}}$. A timed word with resets is *accepting* when its final configuration is in $\mathsf{Accept}_{\mathcal{S}^{\mathcal{T}}}$.

In order to obtain a finite representation of the infinite set of timed words with resets, we use an abstraction based on the following notion of *K-equivalence*.

Definition 3. *Two nonnegative reals x and y are K*-equivalent*, noted $x \approx_K y$, when either $x > K$ and $y > K$, or $x = y$ are integers, or x and y are non-integers and they have the same integral part. Two valuations v and v' are K-equivalent if $v(x) \approx_K v'(x)$ for all $x \in X$. We say that two configurations are K-equivalent when their valuations are, and that two timed words with reset are K-equivalent when they have the same size and the configurations of same indices in both words are K-equivalent.*

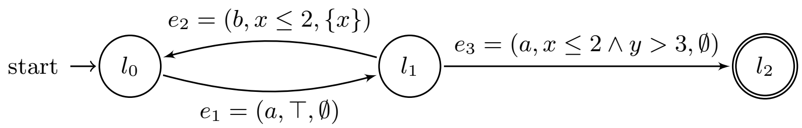


Fig. 1: A simple TA

Notice that K-equivalence is coarser than the usual notion of *region equivalence* of [AD94], as it aims to encode direct indistinguishability by a guard along words, instead of indistinguishability in the future.

We call *zone-word with resets* a timed word with resets in which all valuations are replaced with zones. A timed word with resets $r = ((l_i, v_i) \xrightarrow{e_i} (l_{i+1}, v_{i+1}))_{i \in [0,n]}$ is *compatible* with a zone word with resets $zr = ((l_i, z_i) \xrightarrow{e_i} (l_{i+1}, z_{i+1}))_{i \in [0,n]}$, written $r \models zr$, when $v_i \models z_i$ for all i. We call *K-closed word* a zone word in which all zones are K-equivalence classes.

Lemma 4. *For any timed word with reset r of a (deterministic) timed automaton $\mathcal{T}$, there is a unique K-closed word zr such that $r \models zr$. For any timed word with resets r' compatible with zr, r' is accepting if, and only if, r is.*

Event recording automata (ERA) [AFH99] are a subclass of TAs in which there is one clock x_a per letter a of the alphabet, such that x_a is reset exactly along a-transitions. We slightly extend them as *reset-free ERAs* (RERAs), in which transitions may or may not reset their clock: we let $X_\Sigma = \{x_a \mid a \in \Sigma\}$, and $\mathcal{Z}_\Sigma$ and $\mathcal{G}_\Sigma$ be shortcuts for $\mathcal{Z}_{X_\Sigma}$ and $\mathcal{G}_{X_\Sigma}$ respectively.

Definition 5. *A* reset-free event recording automaton *(RERA) over Σ is a TA $\mathcal{T} = (\mathcal{L}, l_0, X_\Sigma, E, \mathsf{Accept})$ such that for all transitions $(l, a, g, r, l') \in E$, it holds $r \in \{\{x_a\}, \emptyset\}$.*

Example 1. Consider the timed automaton depicted in Fig. 1. This TA is actually a RERA, by associating clock x to letter b and clock y to letter a. An accepting timed word with resets of this automaton is $(l_0, \mathbf{0}) \xrightarrow{1.5} (l_0, (\begin{smallmatrix}1.5\\1.5\end{smallmatrix})) \xrightarrow{a,\emptyset} (l_1, (\begin{smallmatrix}1.5\\1.5\end{smallmatrix})) \xrightarrow{b,\{x\}} (l_0, (\begin{smallmatrix}0\\1.5\end{smallmatrix})) \xrightarrow{a,\emptyset} (l_1, (\begin{smallmatrix}0\\1.5\end{smallmatrix})) \xrightarrow{2} (l_1, (\begin{smallmatrix}2\\3.5\end{smallmatrix})) \xrightarrow{a,\emptyset} (l_2, (\begin{smallmatrix}2\\3.5\end{smallmatrix}))$. The corresponding path is $l_0 \xrightarrow{e_1} l_1 \xrightarrow{e_2} l_0 \xrightarrow{e_1} l_1 \xrightarrow{e_3} l_2$.

Although closely related, ERA and RERA differ in a central way w.r.t. our learning problem: while the resets of an ERA can be directly inferred from observations, in a RERA this is not directly possible. Thus, generalizing a learning method from ERA to RERA requires dealing with the inference of resets—one of the central challenges of the learning of general deterministic TA.

2.2 Timed languages

Automata-learning techniques are based on the identification of a candidate automaton that generalizes the *observations* obtained during the learning process. Angluin's tabular approach [Ang87a] directly identifies a set of observations (*i.e.* words) having good properties, and builds a deterministic automaton from it. Our contribution, as well as all the active-learning algorithms that we are aware of, follow a similar approach. An important issue for extending this approach to timed words is the infinite number of observations fitting even the simplest model, due to time density. We thus have to use good abstractions to represent classes of these words, and use these classes to direct the learning process. A first such extension was initiated in [GJP06].

A timed word with resets of a RERA can be seen as an element of $(\mathbb{R}_{\geq 0} \times \Sigma \times \{\top, \bot\})^*$. A timed word is the projection of a timed word with resets on $(\mathbb{R}_{\geq 0} \times \Sigma)^*$; timed words correspond to observations of timed words with resets.

In order to represent infinitely many timed words with resets in a succinct way, we define *guarded words with resets* $w_{gr} \in (\mathcal{G}_\Sigma \times \Sigma \times \{\top, \bot\})^*$, which correspond to paths in a RERA. For a timed word w_t and a guarded word with resets w_{gr} we say that w_t *satisfies* w_{gr}, noted $w_t \models w_{gr}$, if w_t is a possible observation of w_{gr}. We extend this correspondence to timed words with resets by ensuring that the resets match. The satisfiability relation between timed words and guarded words with resets will be central in the rest of the paper, as it relates an observation to the unfolding of a RERA (or of our hypothesis).

Example 2. The timed word $w_t = (1.3, a).(0.4, b)$ satisfies the guarded word with reset $w_{gr} = (x_b > 1, a, \{x_a\}).(x_a < 1, b, \emptyset)$: indeed, w_t and w_{gr} have the same untimed projection, and the timed word with resets $w_{tr} = \mathbf{0} \xrightarrow{1.3} \left(\begin{smallmatrix} 1.3 \\ 1.3 \end{smallmatrix}\right) \xrightarrow{a, \{x_a\}} \left(\begin{smallmatrix} 0 \\ 1.3 \end{smallmatrix}\right) \xrightarrow{0.4} \left(\begin{smallmatrix} 0.4 \\ 1.7 \end{smallmatrix}\right)$ satisfies the guards of w_{gr}. Notice that $w_t \not\models w'_{gr} = (x_b > 1, a, \emptyset).(x_a < 1, b, \emptyset)$, as modifying resets changes the valuations that appear in the corresponding timed word with resets.

Zone words with resets can be seen as elements w_z of $(\mathcal{Z}_\Sigma \times \Sigma \times \{\top, \bot\})^*.\mathcal{Z}_\Sigma$. From a guarded word with resets $w_{gr} = (g_i, a_i, r_i)_{i \in [0,n]}$ we can define the corresponding zone word with resets $w_z = (z_i, a_i, r_i)_{i \in [0,n]} z_{n+1}$ with $z_0 = \{\mathbf{0}\}^{\nearrow}$ and $z_{i+1} = (z_i \wedge g_i)^{\nearrow}$ if $r_i = \bot$ and $z_{i+1} = (z_i \wedge g_i)_{[x_{a_i} \leftarrow 0]}{}^{\nearrow}$ otherwise.

In our learning process, we will manipulate linear combinations of timed words. For two timed words $w_t^1 = ((t_i^1, a_i))_{i \in [0,n]}$ and $w_t^2 = ((t_i^2, a_i))_{i \in [0,n]}$ with the same untimed projection, we define their λ-*weighted sum* $w_t^3 = \lambda.w_t^1 + (1 - \lambda)w_t^2$, as the timed word $w_t^3 = ((\lambda.t_i^1 + (1 - \lambda).t_i^2, a_i)_{i \in [0,n]})$. Such linear combinations have the following property:

Proposition 6. *For any two timed words $w_t^j = (t_i^j, a_i)_{i \in [0,n]}$ for $j \in \{1, 2\}$ with the same untimed projection, for any $\lambda \in [0, 1]$ and for any reset word $(r_i)_{i \in [0,n]}$, all the valuations $v_{i,r}^3$ reached along $w_{tr}^3 = ((\lambda.t_i^1 + (1 - \lambda).t_i^2, a_i, r_i)_{i \in [0,n]})$ are such that for all clocks $x_a \in X_\Sigma$, $v_{i,r}^3(x_a) = \lambda.v_{i,r}^1(x_a) + (1 - \lambda).v_{i,r}^2(x_a)$ for $v_{i,r}^j$ the valuations reached along $w_{tr}^j = (t_i^j, a_i, r_i)$.*

3 Observation structure

The general principle of (untimed) active-learning is to learn a model from observations acquired by membership queries and equivalence queries 2. In membership queries, a timed word is provided to a teacher, who in return informs us about the membership of this world in the target language. In an equivalence query, we propose an hypothesis (model) to the teacher; she either accepts it if it is equivalent to the model we wish to learn, or otherwise provides us with a counterexample, *i.e.*, a timed word that separate the language of the model and that of our hypothesis. The set of observations is formalized as a partial

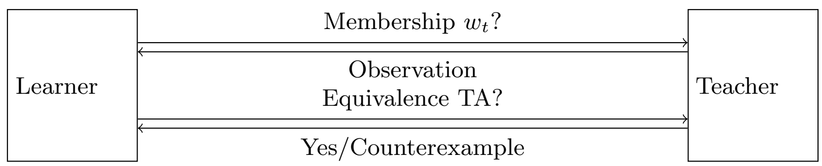


Fig. 2: The basic active learning framework

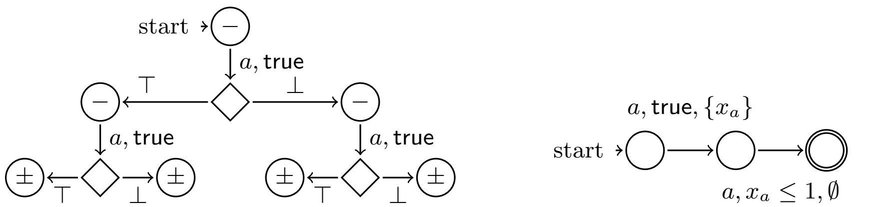
function Obs mapping words to acceptance status (+ or −). To build a model, we then want to identify a prefix-closed subset U of $\mathsf{Dom}(\mathsf{Obs})$ such that for all letters a in the alphabet and words $u \in U$, $u.a \in \mathsf{Dom}(\mathsf{Obs})$ and either $u.a \in U$, or there is another word $u' \in U$ having the same observed behaviour as $u.a$. When transferring this approach to timed words, one has to deal with two difficulties: first, the uncountable number of possible delays before each discrete action; second, the fact that observations do not include clock valuations (nor clock resets), which we also have to learn.

In this section, we describe the structures used to represent and process these timed observations acquired during the learning *and* the decisions on the built structures made based on those observations. We generalize timed decision trees defined in [GJP06], so as to encode timed words *with possible resets*. We basically use a *timed decision graph*, a model close to acyclic timed automata, to encode the current knowledge inferred about the model from observations, and a *timed observation graph* (TOG) to *implement* Obs with a step of abstraction and help decisions.

Our data structure is centered around the notion of *observation structure* composed of a *timed decision graph*, which stores the current hypothesis (and will later be folded into a TA), and an *observation function*, which stores current observations.

Definition 7. *An* observation structure *is a pair* $(\mathcal{N}, \mathsf{Obs})$ *made of a* timed decision graph *(TDG) and a partial mapping* Obs *from timed words to* $\{+,-\}$*. The TDG is a labelled bipartite graph* $\mathcal{N} = (S, E)$ *with* $S = S_l \uplus S_d$ *where:*

– $S_l \subseteq \{s_0 = (\epsilon, \{\mathbf{0}\}^{\nearrow})\} \cup (\mathcal{G}_\Sigma \times \Sigma \times \{\top, \bot\})^+ \times \mathcal{Z}_\Sigma$ *is a set of* language states*, made of a prefix-closed finite set of guarded words with resets paired with zones;* s_0 *is the root state.*



(a) A first observation structure (b) The RERA providing observations

Fig. 3: An active-learning setting

- $S_d \subseteq S_l \times \Sigma \times \mathcal{G}_\Sigma$ *is a set of* decision states *such that for any* $s_l \in S_l$ *and* $a \in \Sigma$*, if* $I = \{g \in \mathcal{G}_\Sigma \mid (s_l, a, g) \in S_d\}$ *is non-empty, then* $\bigvee_{g \in I} g \equiv \top$ *and for all* g *and* g' *in* I*, if* $g \neq g'$ *then* $g \wedge g' \equiv \bot$*;*
- $E \subseteq S \times (\Sigma \times \mathcal{G}_\Sigma \cup \{\top, \bot\}) \times S$ *is defined such that transitions to a decision state* $s_d = (s_l, a, g)$ *are of the form* (s_l, a, g, s_d) *and if* $s_l = w_{gr}.z$ *transitions from* s_d *are* $(s_d, \top, (w_{gr}.(g, a, \top), (z \wedge g)_{[x_a \leftarrow 0]}{}^{\nearrow}))$ *and* $(s_d, \bot, (w_{gr}.(g, a, \bot), (z \wedge g)^{\nearrow}))$*.*

The labelling *of an observation structure maps language states to the set of observations compatible with them:*

$$\mathit{label}(s_l = (w_{gr}.z)) = \{\mathit{Obs}(w_t) \mid w_t \in \mathit{Dom}(\mathit{Obs}) \wedge w_t \models w_{gr}\}.$$

It can be seen from this definition that TDGs are trees (see the proof in Appendix B). For a guarded word w_{gr}, we note $s_0 \xrightarrow{w_{gr}}_{\mathcal{N}} s_l$ when there is a path in $\mathcal{N}$ from s_0 to s_l labelled with w_{gr}, and note $w_{gr} \in \mathcal{N}$ when such a path exists.

Observation structures store both the words that have been observed (in Obs) and the inferred guards and enforced resets (or absence thereof) (in $\mathcal{N}$). We can extend Obs to guarded words with resets by considering them as language states and using their labels. The labels are used to carry the observation information to the TDG.

Example 3. Fig. 3a represents an observation structure storing some words observed from the RERA in Fig. 3b. Language states are depicted as circles and decision states as diamonds. Notice that in this example the leaves have labels of size 2: they model both accepting and non-accepting observations *e.g.* $((0.7, a)(0.9, a), +)$ and $((0.7, a)(1.2, a), -)$.

We define some desired properties of information structures.

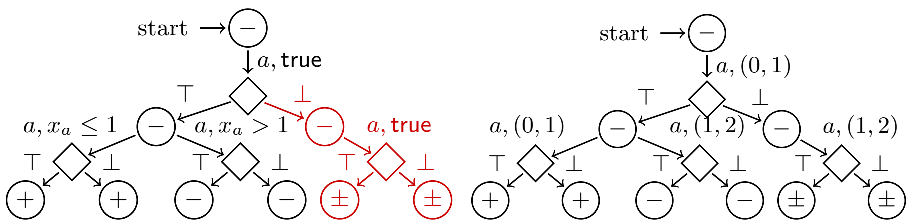
Definition 8. *For an observation structure* $(\mathcal{N}, \mathit{Obs})$*, a subtree* $\mathcal{N}'$ *of* $\mathcal{N}$ *rooted in* $s_l^{\mathcal{N}'} = w'_{gr}.z'$ *is said:*

- complete *when all observations in* Obs *are taken into account, i.e. for any* $w_t \in \mathit{Dom}(\mathit{Obs})$ *such that* $w_t = w'_t.w''_t$ *with* $w'_t \models w'_{gr}$ *there is* $s_l^{\mathcal{N}'} \xrightarrow{w''_{gr}} s_l$ *such that* $w''_t \models w''_{gr}$ *and for all such* w''_{gr} *and* s_l*,* $\mathit{Obs}(w_t) \in \mathit{label}(s_l)$*;*

– consistent *when it separates accepting and non accepting behaviours, i.e. for any* s_l *in the subtree,* $|\mathsf{label}(s_l)| = 1$.

We say that an observation structure is complete or consistent when $\mathcal{N}$ *is.*

Detecting and handling inconsistencies is central to our algorithms, as it characterizes the need to introduce new guards to split language nodes in the timed decision graph.



(a) After handling the inconsistency (b) A timed observation graph

Example 4. The leaves of the TDG in Fig. 3a are inconsistent. The inconsistency can be resolved for the left branch by splitting the transition, as made in Fig. 4a. This leaves a label of size two in the right branch, but there exists no guard that can separate the observations.

We now define *timed observation graphs*, a structure used to encode the observation function Obs efficiently and abstractly. More precisely, it represents the undistinguishable tube around each observation (*i.e.* the K-closed-words with resets), and allows to detect on-the-fly when two observations sharing the same K-closed word do not agree on acceptance and when reset combinations cannot happen.

Definition 9. *A* timed observation graph *(TOG) is a TDG where all guards and zones correspond to K-equivalence classes, language states are called* observation states $s_O \in S_O$ *and transitions from decision to observation states do not use the future operator,* i.e. *for* $(s_d = (w.z), \top, s_O) \in E$, $s_O = w.(g, a, \top).g_{[x_a \leftarrow 0]}$ *and same for* $\bot$. *We add a labelling* $l \colon S_O \to \mathcal{P}(\{+, -\})$ *for observation states and* $\mathsf{words} \colon S_O \to \mathcal{P}((\mathbb{R}_{\geq 0}.\Sigma)^*)$ *a function associating to each observation state a set of observations that it represents. For two observation states* s_O *and* s'_O*, we note* $s_O \xrightarrow{w_{zr}} s'_O$ *if there is a path from* s_O *to* s'_O *and there exists a zone word* $w.z$ *such that* $s_O = w.z$ *and* $s'_O = w.w_{zr}$.

As for TDGs, TOGs are trees (see the proof in appendix B). Timed observation graphs will allow to detect impossible combinations of resets denoted by labels of observation states of cardinality larger than one. This is ensured by an encoding of Obs into the TOG, in a way defined as follows:

Definition 10. *A timed observation graph* Obs_e *is said to* implement *an observation function* Obs *when the following two conditions are fulfilled:*

Correspondence: *all observations are encoded in the TOG, i.e. for all* $w_t \in \mathsf{Dom}(\mathsf{Obs})$*, for any* w_{tr} *compatible with* w_t*, there is a path* $s_\epsilon \xrightarrow{w_{zr}} s_O = w_{zr}$ *in* Obs_e *such that* $w_{tr} \models w_{zr}$*,* $w_t \in \mathsf{words}(s_O)$ *and* $\mathsf{Obs}(w_t) \in l(s_O)$*;*

Coverage: *all observation states are covered by* $\mathsf{Dom}(\mathsf{Obs})$*, i.e. for any* $s_O = (w_{zr}) \in S_O$*,* $\mathsf{words}(s_O) \neq \emptyset$ *and for any* $w_t \in \mathsf{words}(s_O)$*,* $w_t \in \mathsf{Dom}(\mathsf{Obs})$*,* $w_t \models w_{zr}$ *and* $\mathsf{Obs}(w_t) \in l(s_O)$*.*

Example 5. The TOG in Fig. 4b corresponds to the observation structure displayed in our previous examples. Notice that it has a label of size two on the leafs of the right branch.

The pruning of the timed decision graph relies on *invalidity* of words and states, our key contribution to the active learning framework for timed automata. It allows to characterize reset combinations that are impossible for a given K-closed word. This complements inconsistency and allows to prune resets and schedule guards to be added when resets are not tied to observations.

Definition 11. *A* K-closed *word with reset* $w_{zr} = (z_i, a_i, r_i)_{i\in[0,n]}z$ *is* invalid *with respect to an observation graph* Obs_e *if one of the following conditions holds:* $|l(w_{zr})| = 2$*, or a prefix of* w_{zr} *is invalid w.r.t.* Obs_e*, or there exists* z_{n+1}, a_{n+1} *such that both* $(z_i, a_i, r_i)_{i\in[0,n]}\cdot(z_{n+1}, a_{n+1}, \top){z_{n+1}}_{[a\leftarrow 0]}$ *and* $(z_i, a_i, r_i)_{i\in[0,n]}\cdot(z_{n+1}, a_{n+1}, \bot)z_{n+1}$ *are invalid w.r.t.* Obs_e*.*

A zone word with reset (or a guarded word with reset) is invalid if it models an invalid K*-closed word with reset.*

Invalid guarded words with resets encode behaviours that can not correspond to any model, and thus should be pruned in the TDG:

Proposition 12. *If a timed observation graph* Obs_e *has an invalid observation state* $s_O = w_{zr}$*, there is no TA model having execution* w_{zr}*.*

Situations may arise where a guarded word with reset is not invalid but all its successors by a given action are; an example is presented below. In such situations, two different K-closed words with resets make the successors invalid, and a guard has to be added.

Example 6. Consider the partial set of observations $\{((1.7, a)(1, a), +),\ ((1.7, a)(1.1, a), -),\ ((2.9, a)(1.1, a), -),\ ((2.7, a)(1.1, a), +)\}$ over the alphabet $\Sigma = \{a\}$. The corresponding partial timed observation graph Obs_e is displayed in Fig. 5[1]. We do not represent the actual K-equivalent classes on the graph so as to keep the figure as simple as possible. It can be seen that both resetting and not resetting the clock after the first action may sometimes lead to an invalidity. Hence, taking

[1] In order to avoid overloading the explanation, we call the observation and graph partial because we do not mention some of the observations that would be necessary to have the implementation property.

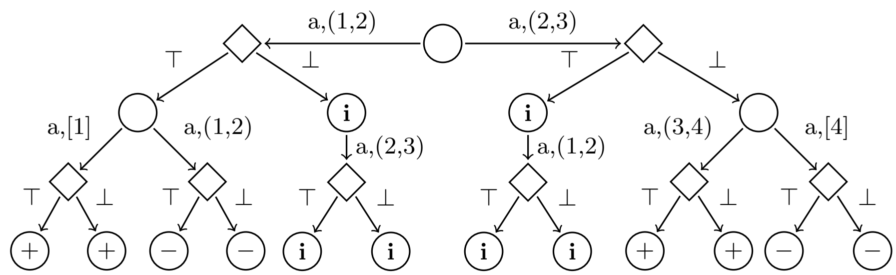


Fig. 5: A (partial) timed observation graph with some invalid nodes.

these observations into account in a timed decision graph with a $\top$ guard on this transition leads to pruning both successors of a decision tree.

This is problematic, as a decision state should always have successors. Hence it is necessary to introduce a guard to distinguish the different invalidities.

4 Updating a timed observation structure

We define the algorithms used to update the previously defined data structures. The general idea is to add observations while preserving the good properties of the data structures, which requires detecting inconsistencies and invalidities on-the-fly, and resolving them by adding new guards.

The algorithms in Sec. 4.1 handle new observations while keeping most of the good properties of the structures, except for consistency. When inconsistencies arise, calls are scheduled to the algorithms proposed in 4.2. Sec. 4.3 deals with a similar but different problem arising from different invalidities meeting each others. Finally an algorithm to rebuild (parts of) the structure using the informations gathered using the previous section algorithms is described in Sec. 4.4.

4.1 Adding a new observation

In essence, our algorithms propagate new words in the TDG $\mathcal{N}$, using satisfiability between guarded words with resets and timed words to guide the descent in the tree. When new states have to be created, membership queries are launched to get a label for them. All of this is complemented by a similar work on the TOG Obs_e, in order to take into account all the new observations. The main difference between the two algorithms is that in the TDG, labels of size 2 are detected and left for a future handling as the procedure to identify guards is potentially heavy, while in the TOG, invalidity leads to immediate pruning in order to limit the size of the structures.

We use the functions $\mathsf{FindPath}_{\mathcal{N}}$ (Algorithm 1)and $\mathsf{FindPath}_{\mathsf{Obs}_e}$ (Algorithm 2)to propagate new observations in the existing structures. Subsequent creation of new nodes is made with the functions $\mathsf{AddWord}_{\mathcal{N}}$ (Algorithm 3) and $\mathsf{AddWord}_{\mathsf{Obs}_e}$ (Algorithm 4). Membership queries and the resulting function calls are handled by the $\mathsf{Request}$ function (Algorithm 5), and the effective pruning is made in $\mathsf{SearchPrune}$ (Algorithm 6).

The $\mathsf{FindPath}_{\mathcal{N}/\mathsf{Obs}_e}$ algorithms execute the descent through the existing structures, while the $\mathsf{AddWord}_{\mathcal{N}/\mathsf{Obs}_e}$ ones extend the structures, and make calls to $\mathsf{Request}$. The latter algorithm first checks if a fitting observation already exists before making a membership query if necessary. The $\mathsf{SearchPrune}$ procedure follows the lines of the definition of invalidity and finds the root of the invalid subtree before pruning it.

The following three statements express soundness of our algorithms. They ensure that the good properties of the structures are invariant by the call to the $\mathsf{FindPath}$ algorithms. Property 13 states that $\mathsf{FindPath}_{\mathcal{N}}$ keeps the good properties of $\mathcal{N}$, except consistency, that is handled by in later. Property 14 does the same for $\mathsf{FindPath}_{\mathsf{Obs}_e}$ and Obs_e, while property 15 ensures that the calls made to $\mathsf{SearchPrune}$ during the execution of the $\mathsf{FindPath}$ algorithm prunes exactly the invalid words.

Proposition 13. *Starting from a complete observation structure $(\mathcal{N}, \mathsf{Obs})$ such that $|\mathsf{Obs}(w_{gr})| \geq 1$ for all $w_{gr} \in \mathcal{N}$, and a new word w_t associated with an observation o, a call to $\mathsf{FindPath}_{\mathcal{N}}(w_t, o, \epsilon, \mathbf{0}, s_0)$ terminates and modifies the observation structure in such a way that it is complete, $w_t \in \mathsf{Dom}(\mathsf{Obs})$, $\mathsf{Obs}(w_t) = o$, and $|\mathsf{Obs}(w_{gr})| \geq 1$ for all $w_{gr} \in \mathcal{N}$.*

Proposition 14. *Starting from a timed observation graph Obs_e implementing an observation function Obs, and a new timed word w_t associated with the observation o, a call to $\mathsf{FindPath}_{\mathsf{Obs}_e}(w_t, o, \epsilon, \epsilon, \mathbf{0}, s_\epsilon)$ terminates and modifies the timed observation graph in such a way that it implements the valid part of Obs extended to w_t.*

Proposition 15. *Starting from an observation structure $(\mathcal{N}, \mathsf{Obs})$ where Obs is implemented by Obs_e and no invalid states can be reached in $\mathcal{N}$, calling $\mathsf{FindPath}_{\mathcal{N}}$ or $\mathsf{FindPath}_{\mathsf{Obs}_e}$ modifies Obs_e and $\mathcal{N}$ in such a way that no invalid states can be reached in $\mathcal{N}$. Furthermore, no valid words are made unreachable.*

4.2 Dealing with inconsistency

An inconsistency arises when a language state of the TDG contains both accepting and non-accepting observations. It means that a guard must be added somewhere in the structure in order to distinguish between these observations.

For this we search for a pair of *adjacent* words, which intuitively identify the boundary between accepting and non-accepting behaviours. We then build a finite set of *differences* between adjacent words, each of which corresponds to a possible guard. This procedure is described in the $\mathsf{AdjPair}$ algorithm.

We use K-equivalence to define the notion of adjacency. Intuitively adjacent words have the same projection on actions and resets, and their valuations either are K-equivalent, or they materialize a boundary between the accepted and non-accepted words.

Definition 16. *For two timed words with resets* $w_{tr} = (v_i \xrightarrow{t_i,a_i,r_i} v_{i+1})_{i\in[0,n]}$ *and* $w'_{tr} = (v'_i \xrightarrow{t'_i,a_i,r_i} v'_{i+1})_{i\in[0,n]}$*, we say that* w_{tr} is adjacent to w'_{tr} *when for all* $i \in [0,n]$ *and* $x_a \in X_\Sigma$*:*

- *if* $v_i(x_a) + t_i \in \mathbb{N}$ *then* $|(v_i(x_a) + t_i) - (v'_i(x_a) + t'_i)| < 1$*,*
- *otherwise,* $v_i(x_a) + t_i \approx_K v'_i(x_a) + t'_i$*.*

Notice that adjacency is not a symmetric relation. We will sometimes abuse the notations and say that a pair w, w' is adjacent to mean that w is adjacent to w'. We use adjacency to identify *differences* between the words as possible new guards that resolve the inconsistency.

Definition 17. *The* difference *between two words* $w_{tr} = (v_i \xrightarrow{t_i,a_i,r_i} v_{i+1})_{i\in[0,n]}$ *adjacent to* $w'_{tr} = (v'_i \xrightarrow{t'_i,a_i,r_i} v'_{i+1})_{i\in[0,n]}$*, noted* $\mathsf{diff}(w_{tr}, w'_{tr})$ *is the set of quadruples defined as: if for a clock* x*,* $v_i(x) + t_i = k \in \mathbb{N}$*, then if* $v'_i(x_a) + t'_i < k$*,* $(i, x, k, \geq) \in \mathsf{diff}(w_{tr}, w'_{tr})$ *and if* $v'_i(x_a) + t'_i > k$*,* $(i, x, k, \leq) \in \mathsf{diff}(w_{tr}, w'_{tr})$*.*

Using these definitions, we can derive from two adjacent words a set of candidates to make a new guard. AdjPair makes membership queries on linear combinations of the two initial observations to perform a binary search until the clock values of the pair have less than 1 time unit of distance. Then it forces every non-K-equivalent pair of clock values to have one of its elements be an integer with more linear combinations. Finally, in order to ensure that only one of the two words have such integer distinctions, it compares them with their mean. This gives an adjacent pair.

Proposition 18. *The AdjPair algorithm constructs an adjacent pair using at most* $\mathcal{O}(m|\Sigma|log(K))$ *membership queries.*

Proof. We refer the reader to the proof of Theorem 5.8 in [GJP06].

4.3 Dealing with invalidity

A label of size two in the TOG indicates an invalidity. It points to a combination of resets being impossible combined with those precise observations. Invalidity is simply dealt with by pruning the invalid parts of the TDG and TOG. But a challenge can arise, as explained in Example 6: sometimes *all* successors of a decision state of the TDG following a *valid* language state are pruned, due to invalidities. In this case, a guard must be introduced to separate the different invalidities and allow to rebuild the graph accordingly. As for inconsistencies, it is important to introduce guards that model as closely as possible the changes in behaviours of the observation.

For this purpose, we again use a binary search, but this time manipulating a pair of *sets* of words. Furthermore, as the invalidities are often detected by the precise combination of *fractional values*, the delays in the words are only modified by *integer values*. For two timed words $w^i_t = (t^i_j, a_j)_{j\in[1,n_i]}$ with $n_1 \leq n_2$, we define the operator $w_t^1 \odot w_t^2 = (\lfloor t_j^1 \rfloor + \langle t_j^2 \rangle, a_j)_{j\in[1,n_1]} \cdot (t_j^2, a_j)_{j\in[n_1+1,n_2]}$ to describe the operation used in the algorithm (where $\lfloor t \rfloor$ and $\langle t \rangle$ respectively represent the integral and fractional parts of t).

Of course, it is impossible to obtain a good precision while keeping all fractional values: clock values can not be modified to become integers. For this reason our algorithm only identifies a set of integer constants separating two behaviours, but does not find which behaviour the constants belong to. This means that we have to wait for a counterexample from an equivalence query to correct the possible wrong guesses we made.

Procedure InvalidityGuard is described in Algorithm 8. It outputs a *validity guard* $(s_l, a, g, x, \sim, k)$ where $a \in \Sigma$, g is a guard, x a clock, $k \in \mathbb{N}$ and $\sim \in \{<, \leq, \top\}$. Such validity guard states that in the language state s_l, after playing a with guard g, adding $x \sim k$ to the guard separates the two causes of invalidity. We use $\top$ to denote that both strict and large inequalities could fit the current observations. The InvalidityGuard algorithm conducts a binary search between two sets of timed words, while keeping the fractional part of the clock values unchanged thanks to the $\odot$ operator, while the K-closed sets corresponding to the sets of words do not touch each other.

Proposition 19. *Algorithm InvalidityGuard terminates after $\mathcal{O}(m(|W_1| + |W_2|) \cdot |\Sigma| \cdot \log(K))$ membership queries, where m is the size of a largest word in $W_1 \cup W_2$.*

Proof. The proof uses the same arguments as the one of AdjPair.

4.4 Rebuilding the graph

To rebuild a subtree is to introduce new guards using adjacent pairs and validity guards only when necessary, and re-propagate the informations in the new guarded words with resets they satisfy. We use Algorithm Rebuild for this. From an adjacent pair, we extract *consistency guards*, which will be used to reconstruct a decision graph that is consistent with respect to the adjacent pair.

Definition 20. *For an adjacent pair w_{tr}, w'_{tr}, clock constraint $x_a \leq k$ is a* consistency guard *at depth i if $(i, x_a, k, \prec) \in \mathit{diff}(w_{tr}, w'_{tr})$ and there is no $(j, x_a, l, \prec') \in \mathit{diff}(w_{tr}, w'_{tr})$ such that $j < i$ or $j = i$ and $l < k$.*

The consistency guards are taken on the first difference, so as to ensure that they can not be overwritten later (there are no guards that can separate the pair before the guard), and to avoid large constants as much as possible.

Notice that we can not always infer a unique guard from an adjacent pair, as multiple clocks can be different at the same time. Intuitively, Rebuild only introduces guards "when needed", which is formalized by the following *well-guardedness* property.

Definition 21. *A timed decision graph is said* well guarded *if, for all transitions $(s_l, a, g, s_d) \in E_\Sigma$ and all constraints $x_b \prec k$ in g, either there is w_{tr} adjacent to w'_{tr} such that both pass by s_l and $x_b \prec k$ is a consistency guard for the pair at this depth* or *$(s_l, a, g', x_b, \sim, k)$ is a validity guard with $g \subset g'$ and $\prec$ is either $\sim$ or $\neg \sim$.*

Rebuild constructs a complete, consistent and well-guarded subtree if it is called high enough in the tree.

Proposition 22. *Running Rebuild on a valid and consistent state s_l of which no successors have inconsistencies that lead to consistency guards at a depth lesser than $|s_l|$, constructs a subtree rooted in its argument that is* complete, consistent *and* well-guarded. *It furthermore does not have* invalid *states.*

This proposition tells us we can keep the timed decision graph up-to-date with respect to observations (*i.e.*, complete and consistent) while keeping the good properties that were ensured by the previous algorithms. It remains to show how a candidate timed automaton can be constructed from this structure.

5 Building a candidate timed automaton

Following the active learning approach, our purpose is to identify a subset of nodes in the decision graph that will correspond to locations of the automaton, and then fold transitions according to an order on the remaining nodes. [GJP06] discusses such orders when resets are fixed. To handle RERA we first have to fix a *reset strategy* before applying the original method. This gives as many hypotheses as we have strategies.

Reset selection. We present the general framework but do not discuss good strategies in the following. Such strategies would rely on heuristics.

Definition 23. *A* reset strategy *over a timed decision graph $\mathcal{N}$ is a mapping $\pi\colon S_d \to \{\top, \bot\}$, assigning a decision to each decision states.*

A reset strategy π is said *admissible* if for any state s_d, there is a language state s_l such that $(s_d, \pi(s_d), s_l) \in E$.

Proposition 24. *In a timed decision graph constructed using the FindPath and Rebuild algorithms and where every scheduled call to Rebuild has been done, there always exists at least one admissible reset strategy.*

An admissible reset strategy is used to prune the decision graph in such a way that only one reset combination is considered for each transition. The effect of an admissible reset strategy π on its timed decision graph $\mathcal{N}$ is the TDG $\pi(\mathcal{N})$ defined from $\mathcal{N}$ by keeping only outgoing transitions from decision states that agree with π. We call this TDG the *resulting graph* of π. It can be seen quite directly that a resulting graph always has exactly one successor to each decision state. Using this, we can notice that those resulting graphs are very close to timed decision trees of [GJP06], in which no decision states exist and the transitions from language states to language states directly hold the (only possible) reset.

Orders and folding. Once an admissible reset strategy is fixed, it is possible to fold the resulting graph into a RERA. This is made through the use of a preorder on states: we want to find a maximal subset for this order.

We define the *height* of a language state s_l, noted $height(s_l)$, as the height of the subtree it is the root of. A preorder $\sqsubseteq$ on language states is said *height-monotone* when $s_l \sqsubseteq s_l'$ implies $height(s_l) \leq height(s_l')$.

Definition 25. *Let $\mathcal{N}$ be a timed decision graph and $\sqsubseteq$ a preorder on its language states. A prefix-closed subset U of $\mathcal{N}$ is called* $\sqsubseteq$-closed *if $s_l \sqsubseteq U$ for all successors of U and* $\sqsubseteq$-unique *if for all $s_l, s_l' \in U$, $s_l \neq s_l' \Rightarrow \neg(s_l \sqsubseteq s_l')$.*

$\sqsubseteq$-closedness is used to construct a RERA by folding the successors of U into comparable states of U. $\sqsubseteq$-uniqueness is useful to bound the number of states in U and thus the size of the resulting automaton.

The following lemma (Lemma 6.2 in [GJP06]) ensures that there always exists a satisfying set of states U. For its constructive proof, we refer the reader to the original paper.

Lemma 26. *Let $\sqsubseteq$ be a height-monotone preorder on states in a resulting graph $\pi(\mathcal{N})$. Then there exists a $\sqsubseteq$-closed and $\sqsubseteq$-unique prefix-closed subset of the language states of $\pi(\mathcal{N})$.*

Using such a subset, we can fold the resulting graph into a RERA as follows:

Definition 27. *Let $(\mathsf{Obs}, \mathcal{N})$ be a consistent observation structure, π an admissible reset strategy and $\sqsubseteq$ a preorder on language states of $\pi(\mathcal{N})$. Consider a $\sqsubseteq$-unique, $\sqsubseteq$-closed and prefix-closed subset U of $\pi(\mathcal{N})$. Then a $U_{\sqsubseteq}$-*merging *of $(\mathsf{Obs}, \mathcal{N})$ according to π is a RERA $(U, \epsilon, X_\Sigma, E, \mathsf{Accept})$ such that $\mathsf{Accept} = \{u \in U \mid \mathsf{label}(u) = \{+\}\}$ and for any language node $u.(a, g, r)$ of $\pi(\mathcal{N})$ with $u \in U$, there is exactly one edge of the form $(u, a, g, r, u') \in E$ with $u.(a, g, r) \sqsubseteq u'$. Notice that, by the second condition, a $U_{\sqsubseteq}$-merging RERA is deterministic.*

Furthermore, if the observation structure is complete, a $U_{\sqsubseteq}$-merging generalizes the observations obtained so far.

Constructing a candidate RERA. Using the results of the previous subsections, we can now construct a candidate RERA from our observation structure. All admissible reset strategies can be constructed by branch and bound. Then a merging is constructed for each resulting graph, and equivalence queries are launched.

For each of the RERA constructed by merging, either a counter-example will be returned by the equivalence query, or the candidate is deemed correct. In the latter case, we return this RERA; in the former case, we include the counter-example in our observation structure and repeat the process.

6 Conclusion

In this paper, we propose an active learning method for deterministic reset-free event recording automata. We add a key feature to the state of the art:

invalidity, that allows to detect incorrect guesses of resets when they are not tied to observations. This required to rework all the data structures and algorithms involved to handle invalidity on the fly. Most importantly, this brings the lacking notion to scale up to the class of deterministic timed automata (DTAs).

A clear future work is to generalize this method to actually handle DTAs. This mostly requires to handles resets of sets of clocks instead of single ones. As the complexity would be greatly increased, this calls for some optimization. An promising addition would be to use an implicit structure. Instead of storing all possible reset configurations, only storing a small set of them at the same time would decrease the memory cost. As the models are built directly from observations, and not from previous states, the computational overhead may be limited. An other interesting trail for future development is to find a way to build a timed automaton from the observation structure that exploits the different admissible reset strategies without building all of them. Works on approximate determinization of timed automata through games [BSJK11] deal with similar problems and offer interesting leads. Finally, in [GJP06], the authors propose to refine the adjacent pairs into *critical pairs*, that have a minimal set of differences. This allows to better identify the guards to be added, and thus can have a positive effect on both the size of the constructed models and the computational cost. Sadly, no precise procedure is given to construct the pairs, so creating one would be beneficial to the approach. More generally, studying the efficiency of this algorithm and of the variants proposed as future work could help better understand the applicability and bottlenecks of the approach.

References

[ACZ+20] Jie An, Mingshuai Chen, Bohua Zhan, Naijun Zhan, and Miaomiao Zhang. Learning one-clock timed automata. In Armin Biere and David Parker, editors, *Proceedings of the 26th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'20) – Part I*, volume 12078 of *Lecture Notes in Computer Science*, pages 444–462. Springer-Verlag, April 2020.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[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, January 1999.

[Ang87a] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[Ang87b] Dana Angluin. Queries and concept learning. *Mach. Learn.*, 2(4):319–342, 1987.

[Ang90] Dana Angluin. Negative results for equivalence queries. *Mach. Learn.*, 5:121–150, 1990.

[BSJK11] Nathalie Bertrand, Amélie Stainer, Thierry Jéron, and Moez Krichen. A game approach to determinize timed automata. In Martin Hofmann, editor, *Foundations of Software Science and Computational Structures*, 2011.

[GJP06] Olga Grinchtein, Bengt Jonsson, and Paul Pettersson. Inference of event-recording automata using timed decision trees. In *CONCUR*, 2006.

[Gri08] Olga Grinchtein. *Learning of Timed Systems*. PhD thesis, Uppsala University, Sweden, 2008.

[TALL19] Martin Tappler, Bernhard K. Aichernig, Kim Guldstrand Larsen, and Florian Lorber. Time to learn - learning timed automata from tests. In Étienne André and Mariëlle Stoelinga, editors, *Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Amsterdam, The Netherlands, August 27-29, 2019, Proceedings*, volume 11750 of *Lecture Notes in Computer Science*, pages 216–235. Springer, 2019.

[VWW08] Sicco Verwer, Mathijs Weerdt, and Cees Witteveen. Efficiently learning simple timed automata. *Proceedings of the Second International Workshop on the Induction of Process Models at ECML PKDD*, pages 61–68, 2008.

[VWW12] Sicco Verwer, Mathijs Weerdt, and Cees Witteveen. Efficiently identifying deterministic real-time automata from labeled data. *Machine Learning*, 86:295–333, 03 2012.

A Algorithms

FindPath$_{\mathcal{N}}$;

Input: a timed word w_t and its observation o, its past p_t , a valuation v and a language state s_l

if $w_t = \epsilon$ **then**
  add o to the set of labels of s_l
else
  $(t, a).w'_t = w_t$;
  if $\exists (s_l, g, a, s_d) \in E$ **then**
    for $(s_l, g, a, s_d) \in E$ **do**
      if $v + t \models g$ **then**
        for $(s_d, r, s'_l) \in E$ **do**
          if $r = \top$ **then**
            FindPath$_{\mathcal{N}}(w'_t, o, p_t.(t, a), (v + t)_{[x_a \leftarrow 0]}, s'_l)$
          else
            FindPath$_{\mathcal{N}}(w'_t, o, p_t.(t, a), v + t, s'_l)$
        break
  else
    AddWord$_{\mathcal{N}}(w_t, o, p_t, s_l)$

Algorithm 1: Adding a new observed timed word in $\mathcal{N}$

We define common borders for the use of the InvalidityGuard algorithm.

Definition 28. *For two K-equivalence classes z, z', we define their* common borders $cb(z, z') \subset X_\Sigma \times \{<, \leq, \top\} \times \mathbb{N}$ *as:*

$$\frac{z \subset x = k,\ z' \subset k - 1 < x < k}{(x, <, k) \in cb(z, z')} \quad \frac{z \subset x = k,\ z' \subset k < x < k + 1}{(x, \leq, k) \in cb(z, z')}$$

$$\frac{z \subset k - 1 < x < k + 1,\ z' \subset k < x < k + 1}{(x, \top, k) \in cb(z, z')}$$

and the same rules, inverting z and z'.

The common borders are used both to measure the proximity of K-equivalence classes and to create validity guards.

Remark 1. In the rebuild function, we use Request on guarded words with resets instead of timed word. The extension is quite simple thanks to the resets, as searching in Obs$_e$ if an observation modelling the argument exists is only a dive in the tree, and if none is found, making an membership query from the last guess is the same.

Remark 2. As written, Rebuild completely erases the subtree and then reconstructs it. An obvious optimization is to only suppress transitions and nodes when necessary to avoid invalidity or inconsistency. We do not develop this here to keep the algorithm short and simple.

$\mathsf{FindPath}_{\mathsf{Obs}_e}$

Input: a timed word w_t and its observation o, its past p_t and reset history r, a valuation v and an observation state s_O

if $w_t = \epsilon$ **then**
  add o to $l(s_O)$;
  if $|l(s_O)| > 1$ **then**
    $s_O.\mathsf{invalid} = True$;
    $s_d = \mathsf{parent}(s_O)$;
    while *all successors of* s_d *are invalid* **do**
      $s_O = \mathsf{parent}(s_d)$;
      $s_O.\mathsf{invalid} = True$;
      remove the last letters from p_t and r;
      $s_d = \mathsf{parent}(s_O)$;
    $\mathsf{SearchPrune}(p_t, r, s_0, \mathbf{0}, \epsilon)$
else
  $(t, a).w'_t = w_t$;
  if $\exists (s_O, g, a, s_d) \in E$ **then**
    for $(s_O, g, a, s_d) \in E$ **do**
      if $v + t \models g$ **then**
        for $(s_d, r, s'_O) \in E$ **do**
          if $|l(s'_O)| = 1$ **then**
            if $r = \top$ **then**
              $\mathsf{FindPath}_{\mathsf{Obs}_e}(w'_t, o, p_t.(t, a), r.\top, (v + t)_{[x_a \leftarrow 0]}, s'_O)$
            else
              $\mathsf{FindPath}_{\mathsf{Obs}_e}(w'_t, o, p_t.(t, a), r.\bot, v + t, s'_O)$
        break
  else
    $\mathsf{AddWord}_{\mathsf{Obs}_e}(w_t, o, p_t, s_O)$

Algorithm 2: Adding a new observed timed word in Obs_e

$\mathsf{AddWord}_{\mathcal{N}}$

Input: a non-empty timed word w_t, its observation o, its past p_t and a language state s_l

$(t, a).w'_t = w_t$;
create $s_d = (s_l, a, \mathsf{true})$ in S_d;
$w_s, z_s = s_l$;
create $s'_l = (w_s.(a, \mathsf{true}), z_s^{\nearrow})$;
create $s''_l = (w_s.(a, \mathsf{true}), {z_s}_{[x_a \leftarrow 0]}^{\nearrow})$;
create $(s_l, a, \mathsf{true}, s_d)$, $(s_d, \bot, s'_l)$ and $(s_d, \top, s''_l)$ in E;
if $w'_t = \epsilon$ **then**
  label s'_l and s''_l by $\{o\}$
else
  $o' = Request(p_t)$;
  label s'_l and s''_l by $\{o'\}$;
  $\mathsf{AddWord}_{\mathcal{N}}(w'_t, o, s'_l)$;
  $\mathsf{AddWord}_{\mathcal{N}}(w'_t, o, s''_l)$

Algorithm 3: Extending $\mathcal{N}$ to satisfy a new timed word

$\mathsf{AddWord}_{\mathsf{Obs}_e}$

Input: a non-empty timed word w_t, its observation o, its past p_t and an observation state s_O

$(t, a).w'_t = w_t$;
create $s_d = (s_l, a, \mathit{reg}(v + t))$ in S_d;
$w_s, z_s = s_O$;
create $s'_O = (w_s.(a, \mathit{reg}(v + t)), \mathit{reg}(v + t))$;
create $s''_O = (w_s.(a, \mathit{reg}(v + t)), \mathit{reg}(v + t)_{[x_a \leftarrow 0]})$;
create $(s_O, a, \mathit{reg}(v + t), s_d)$, $(s_d, \perp, s'_O)$ and $(s_d, \top, s''_O)$ in E;
if $w'_t = \epsilon$ **then**
  label s'_O and s''_O by $\{o\}$
else
  $o' = \mathit{Request}(p_t)$;
  label s'_O and s''_O by $\{o'\}$;
  AddWord(w'_t, o, s'_O);
  AddWord(w'_t, o, s''_O)

Algorithm 4: Extending Obs_e to satisfy a new timed word

$\mathsf{Request}$

Input: A timed word w_t.

Output: A unit label in $\{\{+\}, \{-\}\}$

if $w_t \in \mathit{Dom}(\mathit{Obs})$ **then**
  return $\mathsf{Obs}(w_t)$
else
  make an membership query on w_t and add its result o to Obs;
  $\mathsf{FindPath}_{\mathsf{Obs}_e}(w_t, o, \epsilon, \mathbf{0}, s_0)$;
  return o

Algorithm 5: Requesting an observation.

SearchPrune

Input: A non-empty timed word w_t and a set of resets r of same length, a language state s_l, a valuation v and an history $h_d \in (S_d \times \{\top, \bot\})^*$.

$(t, a).w'_t = w_t$;
$r_a.r' = r$;
for $(s_l, g, a, s_d) \in E$ **do**
 if $v + t \models g$ **then**
  for $(s_d, r'_a, s'_l) \in E$ **do**
   if $r'_a = r_a$ **then**
    if $w'_t = \epsilon$ **then**
     remove (s_d, r'_a, s'_l) from E and recursively delet the subtree;
     if $|\{(s_d, \_, \_) \in E\}| = 0$ **then**
      Schedule the subtree rooted in s_l to be rebuilt.
    else
     if $r_a = \top$ **then**
      SearchPrune$(s'_l, (v + t)_{[x_a \leftarrow 0]}, h_d.(s_d, r_a))$
     else
      SearchPrune$(s'_l, (v + t), h_d.(s_d, r_a))$
    break
  break

Algorithm 6: Pruning $\mathcal{N}$ after detecting an invalid timed word with resets.

B Proofs

We conduct here the proofs of different claims.

B.1 Proofs of section 2

Proposition 6. *For any two timed words $w_t^j = (t_i^j, a_i)_{i \in [0,n]}$ for $j \in \{1, 2\}$ with the same untimed projection, for any $\lambda \in [0, 1]$ and for any reset word $(r_i)_{i \in [0,n]}$, all the valuations $v_{i,r}^3$ reached along $w_{tr}^3 = ((\lambda.t_i^1 + (1 - \lambda).t_i^2, a_i, r_i)_{i \in [0,n]})$ are such that for all clocks $x_a \in X_\Sigma$, $v_{i,r}^3(x_a) = \lambda.v_{i,r}^1(x_a) + (1 - \lambda).v_{i,r}^2(x_a)$ for $v_{i,r}^j$ the valuations reached along $w_{tr}^j = (t_i^j, a_i, r_i)$.*

Proof. The proof is made by induction on w_t^3. For ϵ, the only valuation encountered is $v_{0,r}^3 = \mathbf{0} = \lambda.v_{0,r}^1 + (1 - \lambda).v_{0,r}^2$. For $w_t^j = w_t'^j.(t^j, a)$, $j \in [1, 3]$, assume that we have the property for all valuations reached along the prefixes, and especially that for all clocks $v_r'^3(x_a) = \lambda.v_r'^1(x_a) + (1 - \lambda).v_r'^2(x_a)$ for the last valuations encountered along the $w_t'^j$. Then we have that $v_r'^3 + t_3 = v_r'^3 + \lambda.t^1 + (1 - \lambda).t^2 = \lambda.(v_r'^1 + t^1) + (1 - \lambda).(v_r'^2 + t^2)$ and by applying the resets commended by r we obtain the result desired for the last valuation.

B.2 Proofs of section 3

We prove formally that TDGs and TOGs are trees.

AdjPair

Input: A timed word with resets w such that $\mathsf{Obs}(w_{tr}) = +$ and a second one w' such that $\mathsf{Obs}(w') = -$.

Output: An adjacent pair

while *not*$(\forall i \forall a,\ |v_i(x_a)+t_i-(v'_i(x_a)+t'_i)| < 1 \vee ((v_i(x_a)+t_i > K) \wedge (v'_i(x_a)+t'_i > K)))$ **do**

  $w'' = 0.5w + 0.5w'$

  if *Request* $(w")$ **then**

    $w = w''$

  else

    $w' = w''$

for $i \in [0, n]$, $a \in \Sigma$ **do**

  if $\lfloor v_i(x_a) + t_i \rfloor \neq \lfloor v'_i(x_a) + t'_i \rfloor \wedge v_i(x_a) + t_i \notin \mathbb{N} \wedge v'_i(x_a) + t'_i \notin \mathbb{N}$ **then**

    if $v_i(x_a) + t_i < v'_i(x_a) + t'_i$ **then**

      $\lambda = (\lfloor v'_i(x_a) + t'_i \rfloor - (v_i(x_a) + t_i))/(v'_i(x_a) + t'_i - (v_i(x_a) + t_i))$

      $w'' = \lambda.w' + (1 - \lambda).w$

    else

      $\lambda = (\lfloor v_i(x_a) + t_i \rfloor - (v'_i(x_a) + t'_i))/(v_i(x_a) + t_i - (v'_i(x_a) + t'_i))$

      $w'' = \lambda.w + (1 - \lambda).w'$

    if *Request*$(w'') = +$ **then**

      $w = w''$

    else

      $w' = w''$

$w'' = 0.5w + 0.5w'$

if *Request*$(w'') = +$ **then**

  return (w', w'')

else

  return (w, w'')

Algorithm 7: Finding an adjacent pair corresponding to an inconsistency.

InvalidityGuard

Input: two sets W_1 and W_2 of timed words with resets corresponding to invalidities. Each set of words passes through a w_{gr} and then shares a same K-equivalence class to play a common action a. W_1 is invalid for reset $\top$ while W_2 is invalid for reset $\bot$.

Output: A validity guard

$n = |w_{gr}|$

We note the shared K-equivalence classes to play action a z_n^1 for W_1 and z_n^2 for W_2

$W_1', W_2' = \emptyset$

if $cb(z_n^1, z_n^2) = \emptyset$ **then**

  for $w_t^1 \in W_1$*)* **do**

    for $w_t^2 \in W_2$ **do**

      $w = 0.5w_t^1[1, n+1] + 0.5w_t^2[1, n+1]$

      ${w'}_t^1 = w \odot w_t^1$ and ${w'}_t^2 = w \odot w_t^2$

      Request $({w'}_t^1)$; Request $({w'}_t^2)$

      $W_1' += \{{w'}_t^1\}$, $W_2' += \{{w'}_t^2\}$

  if *the new observations have made w_{gr} invalid* **then**

    stop

  if *w completed with the resets of w_{gr} plus $\top$ is invalid* **then**

    InvalidityGuard (W_1', W_2, w_{gr}, a)

  if *w completed with the resets of w_{gr} plus $\bot$ is invalid* **then**

    InvalidityGuard (W_1, W_2', w_{gr}, a)

  if *w completed with the resets of w_{gr} plus $\top$ and $\bot$ are valid* **then**

    InvalidityGuard (W_1', W_2, w_{gr}, a) InvalidityGuard (W_1, W_2', w_{gr}, a)

else

  we call s_l the language state in which w_{gr} ends.

  return $(s_l, a, x, \sim, k)$ where $(x, \sim, k) \in cb(z_n^1, z_n^2)$

Algorithm 8: Finding a guard to separate two invalidities.

Rebuild

Input: A (valid) language state s_l

Suppress recursively all successors of s_l;

for *$a \in \Sigma$ such that there is at least an observation passing $s_l.(a, \top)$* **do**

  for *each guard $g \in$ FindGuard$(s_l, a, \top)$* **do**

    create $s_d = (s_l, a, g)$ and $(s_l, a, g, s_d) \in E_\Sigma$;

    if *$s_l.(a, g, \top)$ is not invalid* **then**

      create $s_l' = s_l.(a, g, \top)$;

      label(s_l'') = Request(s_l'');

      Rebuild (s_l')

    if *$s_l.(a, g, \bot)$ is not invalid* **then**

      create $s_l'' = s_l.(a, g, \bot)$;

      label(s_l'') = Request(s_l'');

      Rebuild (s_l'')

Algorithm 9: Rebuilds a subtree of $\mathcal{N}$ to handle consistency.

FindGuard

Input: A (valid) language state s_l, an action a and a guard g

Output: a partition of g

if *there is an adjacent pair passing* $s_l.(a, g)$ *from which a consistency guard at depth* $|s_l|$ *can be deduced* **then**

  let g' be this guard;

  return $\mathsf{FindGuard}(s_l, a, g \wedge g') \cup \mathsf{FindGuard}(s_l, a, g \wedge \neg g')$

else

  if $s_l, g, \top$ *or* $s_l, g, \bot$ *is not invalid* **then**

    return $\{g\}$

  else

    let $s_l, a, g', x, \sim, k$ for $g \subseteq g'$ be a validity guard that differentiate the invalidities.

    Let $g'' = x < k$ if $\sim = <$ and $g'' = x \leq k$ otherwise

    return $\mathsf{FindGuard}(s_l, a, g \wedge g'') \cup \mathsf{FindGuard}(s_l, a, g \wedge \neg g'')$

Algorithm 10: Find a partition in guards to be applied to an action in a language node

Proposition 29. *The part of a timed decision graph reachable from s_0 is a bipartite tree.*

Proof. Consider a timed decision graph $\mathcal{N}$. By definition of E, $\mathcal{N}$ is bipartite. It is acyclic because any path from a language state to an other one leads to a state a guarded word with resets of strictly greater length. And any given state has exactly a unique predecessor, but s_0 that has none. Indeed, for a decision state $s_d = (s_l, a, g)$ the only possible predecessor is the language state s_l, by definition of E; for a language state different from s_0, $s_l = (w.(g, a, r), z)$ the only possible king of incoming transition in E is $s_d = ((w, z'), a, g), r, s_l$ with an a priori unknown z'. But as we dispose in w of the sequence of guards, and precise resets that occurred, we can inductively compute z' by iterating taking the future of the current zone, intersecting it with the guard and applying the desired reset, starting from the zone in s_0. Thus there is only one possible z', and a unique predecessor to s_l.

Proposition 12. *If a timed observation graph Obs_e has an invalid observation state $s_O = w_{zr}$, there is no TA model having execution w_{zr}.*

Proof. By coverage, there are two timed words $w_t, w_t' \in \mathsf{Dom}(\mathsf{Obs})$ such that $w_t, w_t' \models w_{zr}$ and $\mathsf{Obs}(w_t) \neq \mathsf{Obs}(w_t')$. For any given timed automaton, if w_{zr} is a subset of executions, then as every zone in this word is a K-equivalence class, it corresponds to the same sequence of pairs of locations and K-equivalence classes. It then comes that w_t and w_t' are both executions leading to equivalent configurations. Hence the automaton fails to model the differences of the corresponding observation, as such configurations have the same location, which can't be both accepting and not accepting.

B.3 Proofs of section 4

Proposition 13. *Starting from a complete observation structure* $(\mathcal{N}, \mathsf{Obs})$ *such that* $|\mathsf{Obs}(w_{gr})| \geq 1$ *for all* $w_{gr} \in \mathcal{N}$*, and a new word* w_t *associated with an observation* o*, a call to* $\mathsf{FindPath}_{\mathcal{N}}(w_t, o, \epsilon, \mathbf{0}, s_0)$ *terminates and modifies the observation structure in such a way that it is complete,* $w_t \in \mathsf{Dom}(\mathsf{Obs})$*,* $\mathsf{Obs}(w_t) = o$*, and* $|\mathsf{Obs}(w_{gr})| \geq 1$ *for all* $w_{gr} \in \mathcal{N}$*.*

Proof. First of all, a call to $\mathsf{FindPath}_{\mathcal{N}}$ terminates, as recursive calls are in finite number and on words of strictly decreasing length, there is only 1 call to $\mathsf{AddWord}_{\mathcal{N}}$ from $\mathsf{FindPath}_{\mathcal{N}}$ along each explored path and recursive calls to $\mathsf{AddWord}_{\mathcal{N}}$ are in finite number and on words of strictly lower length.

We now prove the rest of the property by induction on the calls to both $\mathsf{FindPath}_{\mathcal{N}}$ and $\mathsf{AddWord}_{\mathcal{N}}$. We use the following induction hypothesis: A call to $\mathsf{AddWord}_{\mathcal{N}}/\mathsf{FindPath}_{\mathcal{N}}$ creates a subgraph that is complete (and takes into account the new observation) and such that for all reachable language states s_l in that subgraph, $\mid label(s_l) \mid \geq 1$.

- Basic case for $\mathsf{FindPath}_{\mathcal{N}}$. Here we have $w_t = \epsilon$. In this case the complete word to add was read before along the path, and the only performed action is to add the observation to the label of s_l. Hence the subgraph is complete with respect to the new observation (by adding delays and actions no other reachable states can correspond to that same word). No other states are created, hence the hypothesis on the initial observation structure suffices to conclude that the subgraph is complete and verifies that all labels have at least one element.
- Basic case for $\mathsf{AddWord}_{\mathcal{N}}$. As $\mathsf{AddWord}_{\mathcal{N}}$ is never called on empty words, we have $w_t = (t, a)$. The call to AddWord adds a new decision state $s_d = (s_l, a, \top)$ and two new language nodes stl' and s_l'' corresponding to the effect of resetting or not x_a after the action. As $\mathsf{AddWord}_{\mathcal{N}}$ was called in $\mathsf{FindPath}_{\mathcal{N}}$, we know that no successors for action a existed in s_l (as the graph is well defined and thus if one existed, one would have covered w_t). As we are in the base case, the labels of s_l' and s_l'' are augmented with the observation o, making them have a non-empty label. The edges constructed by this call are in accord with the definition, and by the hypothesis on the initial observation structure, the other successors of s_l are complete (except with respect to the new word, but their sequence of letters do not match) and have non-empty labels, hence in all cases we obtain a subgraph that is complete (as s_l' and s_l'' have no successors) and have only non-empty labels.
- Inductive case of $\mathsf{FindPath}_{\mathcal{N}}$. We consider that $w_t = (t, a)w_t'$. Thus we enter the else in line 3. If the else case is called in line 12, we only make a call to $\mathsf{AddWord}_{\mathcal{N}}$ on w_t hence by induction hypothesis, we have the desired properties. Else, as there is only one guard such that w_t can go through that guard, all successors satisfying a prefix of w_t are reached by the recursive calls and by induction hypothesis they lead to complete subgraphs with non-empty labels. Furthermore, other successors of s_l constitute, by the hypothesis on the initial structure, a complete subgrap (except that the guard g of the

considered transition is not covered) with only non-empty labels, except for the new word that may not be covered. But by uniqueness, they can not correspond to paths satisfying the new word and thus we have our properties.

– Inductive case for $\mathsf{AddWord}_{\mathcal{N}}$. This case works exactly as the base case, except that calls to "request" ensure that the new states have non-empty labels, and the completeness with respect to w_t is ensured by the induction hypothesis.

Proposition 14. *Starting from a timed observation graph Obs_e implementing an observation function Obs, and a new timed word w_t associated with the observation o, a call to $\mathsf{FindPath}_{\mathsf{Obs}_e}(w_t, o, \epsilon, \epsilon, \mathbf{0}, s_\epsilon)$ terminates and modifies the timed observation graph in such a way that it implements the valid part of Obs extended to w_t.*

Proof. As for the proof of Prop.13 termination is clearly ensured by the structure of recursive calls. Notice that we do not count in this the calls to $\mathsf{FindPath}_{\mathsf{Obs}_e}$ made in $\mathsf{Request}$, as they deal with different words. The same kind of induction on calls of $\mathsf{FindPath}_{\mathsf{Obs}_e}$ and $\mathsf{AddWord}_{\mathsf{Obs}_e}$ suffices to prove correspondence and coverage if no ancestor of the state has a label of cardinality two. If one has, then by definition the state is invalid.

Proposition 15. *Starting from an observation structure $(\mathcal{N}, \mathsf{Obs})$ where Obs is implemented by Obs_e and no invalid states can be reached in $\mathcal{N}$, calling $\mathsf{FindPath}_{\mathcal{N}}$ or $\mathsf{FindPath}_{\mathsf{Obs}_e}$ modifies Obs_e and $\mathcal{N}$ in such a way that no invalid states can be reached in $\mathcal{N}$. Furthermore, no valid words are made unreachable.*

Proof. Invalidity is detected along the calls to $\mathsf{FindPath}_{\mathsf{Obs}_e}$, and the propagation of the invalid tag follows the definition for all ascendant states. No descendant are tagged, but this does not matter as they can not be reached without passing by invalid states as Obs_e is a tree. A call to $\mathsf{SearchPrune}$ is then made, that targets exactly the root of the invalid subtree that has been detected, and prune it. As this is made for all detected invalidities and the subtrees are detected, when the procedure terminates, no language state invalid because of an invalidity detected in $\mathsf{FindPath}_{\mathsf{Obs}_e}$ can be reached. Furthermore only the invalid subtree is suppressed, hence no valid state is made unreachable (as by definition all descendant of invalid states are invalid).

To conclude, it suffices to notice that every new membership query gives rise to a corresponding call to $\mathsf{FindPath}_{\mathsf{Obs}_e}$, leaving no invalidity undetected.

Proposition 22. *Running $\mathsf{Rebuild}$ on a valid and consistent state s_l of which no successors have inconsistencies that lead to consistency guards at a depth lesser than $|s_l|$, constructs a subtree rooted in its argument that is* complete, consistent *and* well-guarded. *It furthermore does not have* invalid *states.*

Proof. We prove these four properties independently.

Well-guardedness The well-guardedness comes directly from $\mathsf{FindGuard}$, as only consistency guards corresponding to passing adjacent pairs and validity guards are added to the guards.

Validity Validity comes from the validity of s_l (by hypothesis) and the validity test made for all descending language trees. Notice that there is also always a successor to any decision state, as the parent language state is valid, and FindPath ensures to construct a guard leaving a reset configuration open.

Consistency The label of each created state receive an element, so it can not be empty. Furthermore, as each inconsistency in the original subtree rooted in s_l lead to a consistency guard of depth greater than $|s_l|$ and FindGuard adds all those consistency guards to the required path, no state can have a label of cardinality two. Thus, combined with the hypothesis on s_l, the subtree is consistent.

Completeness As the Rebuild function continuously calls itself as long as an observation passes the current word, all observations model a word. The condition of label is ensured by the consistency proof: as each state has a non-empty label and each inconsistency has been split by FindGuard, the label of each observation is in the label of the state it models.

B.4 Proofs of section 5

Proposition 24. *In a timed decision graph constructed using the FindPath and Rebuild algorithms and where every scheduled call to Rebuild has been done, there always exists at least one admissible reset strategy.*

Proof. To ensure that an admissible reset strategy exists, one only needs to check that every decision state has at least one successor. We only prune the graph in the SearchPrune algorithm, and this algorithm schedules a call to Rebuild when no successors exist for a decision state. As Rebuild constructs a subtree where all decision states have at least a successor (thanks to the FindGuard function that explicitly checks for this), we have our property.