Noname manuscript No.
(will be inserted by the editor)

Control strategies for off-line testing of timed systems

Léo Henry · Thierry Jéron · Nicolas Markey

Received: date / Accepted: date

Abstract Partial observability and controllability are two well-known issues in test-case synthesis for reactive systems. We address the problem of partial control in the synthesis of test cases from timed-automata specifications We extend a previous approach to this problem from the untimed to the timed setting. This extension requires a deep reworking of the models, game interpretation and test-synthesis algorithms. We exhibit strategies of a game that try to minimize both cooperations of the system and distance to the satisfaction of a test purpose or to the next cooperation , and prove they are winning under some fairness assumptions. This entails that when turning those strategies into test cases, we get properties such as soundness and exhaustiveness of the test synthesis method. We finally propose a symbolic algorithm to compute those strategies.

Keywords Timed system, Game theory, Controllability, Conformance testing

1 Introduction

1.1 Testing real-time systems

Real-time reactive systems are open systems interacting with their environment and subject to timing constraints. Such systems are encountered in many contexts, in particular in critical applications such as transportation, control of manufacturing systems, etc. Their correctness is then of prime importance, but is also very challenging due to multiple factors: combination of discrete and continuous behaviours, concurrency aspects in distributed systems, partial observability and limited controllability over open systems.

To assess the correctness of such systems, testing remains the most-used validation technique, with variations depending on the design phase. Conformance testing is one of those variations, consisting in checking whether a real system, also called *implementations under test*, correctly implements its specification, which serves as a reference. One of the most challenging activities of conformance testing

Irisa, INRIA & CNRS & Univ. Rennes (France)
University College London (Léo Henry)

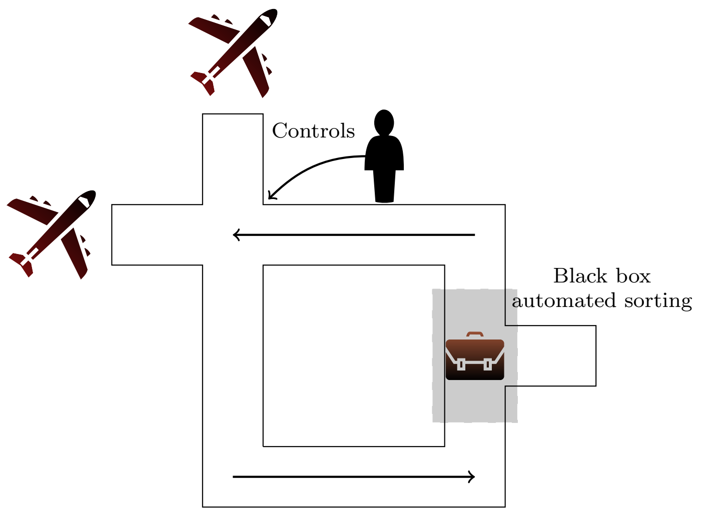


Fig. 1 An airport conveyor belt[1].

is the design of test cases that, when executed on the real system, would produce meaningful verdicts about the conformance of the system at hand with respect to its specification. Those implementations are often considered as black boxes, thereby offering only partial observability to the tester, for various reasons (*e.g.* because sensors cannot observe all actions, or because the system is composed of communicating components whose communications cannot be all observed, or because of intellectual property of peer software). Controllability of the system during the test execution is another well-known issue when the system makes its own choices upon which the tester has a limited control. The tester may then need to rely on the system to reach some targeted behaviors. This is illustrated by the following example.

Example 1. Consider the simple airport conveyor-belt described in Fig. 1. This toy example will serve to illustrate the controllability problem we consider and our approach. Its timed automaton model is given later in Fig. 5. Pieces of luggage arrive on the conveyor belt, and after some time they reach an automated sorting area, where the system may dispose of a luggage *without any control from the operator*. If it is not removed, the piece of luggage reaches an operator, who can choose to route it towards one of the two planes. If the operator fails to decide, after some time the piece of luggage loops on the belt and restarts the whole process.

When testing such a system from the operator's point of view, the tester would have no control over the sorting of the luggage by the automated-sorting system.

A possible test case could aim at verifying the requirement that the conveyor speed is as expected (*i.e.*, that time constraints are met during the system execution). During the execution of a test case for such a property, the automated-sorting device may choose to systematically dispose of pieces of luggage, which would not allow to measure the time they take to reach the operator. In such a situation,

[1] Briefcase icon, commercial airplane icon and person icon by Delapouite under CC BY 3.0 from `https://game-icons.net/`.

cooperation from the system is therefore eventually required in order to reach a conclusive verdict.

1.2 Testing with formal methods

Formal models and methods are good candidates to help test-case synthesis, both in terms of productivity gain and increased confidence in their verdict [Tre96]. Observability and controllability problems are central issues to overcome in this task. Controllability is the main focus of the present work, even though observability is considered.

The models we will use are variations of Timed Automata (TA) [AD94] which form a class of models tailored to the formal description of timed reactive systems. Syntactically, timed automata are finite-state automata equipped with real-valued clocks, which can be used to constrain their behaviours by imposing conditions on the delays between different transitions. TAs are popular in particular in formal verification because of the good balance they offer between their expressiveness and their algorithmic properties.

In the setting of formal testing, it is adequate to refine the model of timed automata by explicitly distinguishing inputs, which are controllable by the tester, and outputs, which are not. This gives rise to the model of TAIOs (Timed Automata with Inputs and Outputs) [KT04,KT09]. In the present paper, TAIOs will be used for most testing artifacts, namely specifications, implementations, and test cases. Notice that our approach does not strongly depend on this TAIO model and other variants of the TA model equipped with inputs and outputs already used in the context of testing could have been used, such as TIOA [SVD01,KLSV03], or TIOTS [LMN04], as soon as they do not preclude non-controllability, *i.e.*, allow the choice between several outputs in response to an input. Since completeness of testing is hopeless in practice[2], it is helpful and classical to rely on *test purposes*, that focus on some specific behaviours that need to be tested (because they are suspected to exhibit errors, or simply because they describe basic functionalities or requirements whose correction is essential). These may also be derived from coverage criteria . We specify test purposes using Open TAIOs (or OTAIOs) [BJSK12], an extension of TAIOs whose behaviours depend on *external* clocks (namely those of the specification). OTAIOs are powerful models and simpler ones could have been chosen to select behaviors. Formal testing also requires to formally define conformance as a relation between models of specifications and their correct implementations. In the timed setting, the classical tioco relation [KT09] states that, after an observable timed trace of the specification (a sequence of inputs, outputs and delays between them), the outputs and delays of the implementation should be possible in the specification.

This testing framework is sketched in Fig. 2. The general problem we address is the synthesis of test cases, from a specification and a test purpose, that direct the implementation towards the behaviours targeted by test purposes, with the intention that verdicts issued during the executions of the test cases on the implementation are consistent with the actual conformance between the implementation and the

[2] "*Program testing can be used to show the presence of bugs, but never to show their absence!*" (Edsger W. Dijkstra)

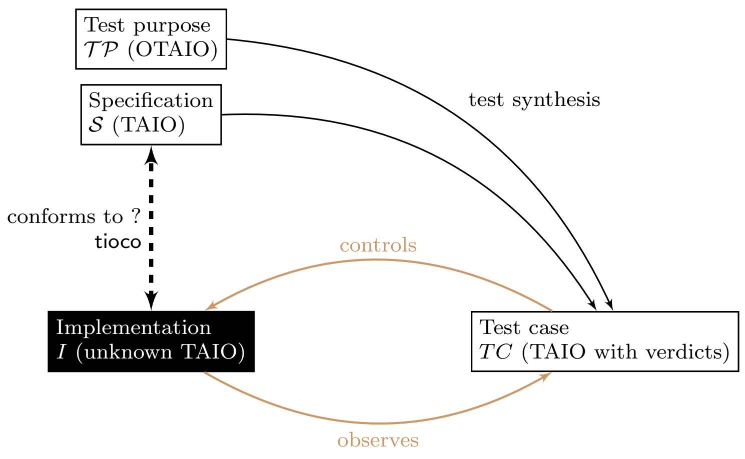


Fig. 2 The testing framework.

specification. In our case, the behaviours are identified as reachability objectives on the test purposes, which will translate into reachability games between the tester and the system during the execution of test cases. This formal framework will be described more precisely later on.

1.3 Test-case synthesis from timed automata

Test-case synthesis from TAs has been extensively studied over the last 20 years (see [COG98, CKL98, SVD01, ENDK02, NS03, BB04, LMN04, KT09], to cite a few). In *off-line testing*, the test cases are first computed and stored, and later executed on the implementation. They should thus anticipate all specified outputs after an observed trace. One of the difficulties then comes from partial observation. In the untimed framework, this is circumvented by determinizing the specification, which allows to obtain a unique execution to correspond to a given trace. Unfortunately, this is not always feasible for TAIO specifications, since determinization is not possible in general for TAs [AD94, Tri04, Fin06]. Possible solutions can be to turn to *on-line testing*, where a subset construction is made on-the-fly on the current execution trace [KT04, LMN04], or to restrict to determinizable sub-classes of TAs (see *e.g.* [NS03]). Some advances in off-line test synthesis were obtained in [BJSK12] by the use of an approximate determinization procedure using a game-based approach [BSJK15] that preserves tioco conformance, and is known to be exact for known sub-classes of TAs, namely Event-Recording TAs [AFH94], strongly non-Zeno TAs [AMPS98a], TAs with integer resets [SPKM08]. We will build on this work regarding observability issues and will recall or adapt the main aspects of this framework in Section 3.

1.4 Controllability of a system under test as a game

The problem of testing is often informally presented as a game between the tester and the system under test (see *e.g.* [Yan04]).

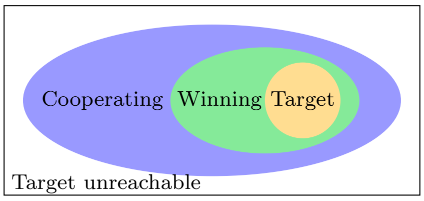


Fig. 3 Hierarchy of the winning and cooperating sets in [DLLN08a].

In the context of timed testing, a game-based approach has been studied in [DLLN08b, DLLN09], where test cases are synthesized as winning strategies of a reachability game. A winning strategy is, in this case, a strategy managing to reach a set of target states against *any possible* strategy of the system.

The work in [DLLN08b] is restricted to deterministic models, while [DLLN09] uses observation predicates to cope with partial observation, but those predicates assume the observability of clocks which is not reasonable from our perspective. In both cases test case synthesis is abandoned when no winning strategy can be found for the tester. Yet, in practice many test cases correspond to games with no winning strategies.

Example 2. Consider again the conveyor-belt example of Example 1. When testing the speed of the conveyor belt, one needs a piece of luggage to go to the operator and continue past this position, which corresponds in the automaton in Fig. 5 to the uncontrollable action !past. For this, the piece of luggage has to go to the Sort location and not be removed by the system (which corresponds to the !waste uncontrollable action) in order to reach the Boarding location, which corresponds to the operator. As one can see, a system always playing !waste as soon as possible could prevent any piece of luggage from ever reaching the operator. Hence it is not possible to construct a winning strategy to test the !past action as it can never be played.

This issue is mitigated in [DLLN08a] with the use of cooperative strategies, which rely on the cooperation of the system under test to win the game. These strategies can distinguish between *winning* states, where they can win despite the system's best effort, and *cooperating* states, where they need to rely on the system to win. The winning and cooperating states form two nested sets, as pictured in Fig. 3. This approach is somewhat limited by the strength of the tester in the cooperating states: the system under test is supposed to always act exactly as desired by the tester at all times while in cooperating states. This extremely strong requirement is hard to justify in practice during testing for a large number of applications.

A more in-depth approach to the controllability problem is the one of [Ram98] in the untimed setting, unfortunately a scarcely-known work. Test selection is modelled as a game where the tester tries to satisfy a test purpose while detecting non-conformance, but faces *control losses*, *i.e.*, states where the system proposes uncontrollable but correct outputs that move it away from its objective. This allows for more precision than the cooperative approach, as each reliance on the implementation is now isolated and counted separately, and interleaved with the classical adversarial game setup. The computed strategies could quantify on the number of *control losses*. Interestingly, this allows to minimize both the reliance on the system decisions and the distance to the next control loss.

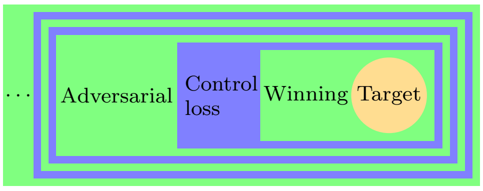


Fig. 4 Interleaving control and control losses to maximize control.

1.5 Contributions

As explained above and illustrated by Example 2, in most cases, focusing on winning strategies in order to build test cases for partially-controllable systems is hopeless. In this paper we address this problem by proposing a game-based approach to formal conformance testing of timed systems that goes beyond the limitations of winning strategies. Our approach accounts for the necessity for the tester to rely on the help of the implementation, and aims at minimizing such cooperation. This idea is illustrated in Fig. 4, with each *control loss* being interleaved with the usual adversarial game approach. On top of minimizing the need for cooperation in the general setting, we propose a notion of fairness under which the strategies we construct are winning.

The present paper extends the results presented in [HJM18]. It adapts the game approach proposed in [Ram98] to the timed context using the framework developed in [BJSK12]. Precisely, we develop *rank lowering strategies* inspired by [Ram98] on top of the testing framework of [BJSK12]. The latter work culminated in the construction of a game between the tester and the system under test. This game represents all observable behaviours of the specification targeted by the test purpose. Yet, it did not construct the strategies of the tester to solve the game, *i.e.*, the choice of inputs and delays that could control as much as possible the system through those behaviours. This paper constructs them as rank-lowering strategies.

Our strategies do not handle inconclusive states (*i.e.*, states that are not co-reachable from the test purpose target). We leverage this issue at the model level by introducing *restart transitions* that formalise "shutting the system off and on again" and ensure that there is always an execution leading to a test purpose target from any configuration of the models.

1.6 Related works

A number of the first papers on test-generation for timed automata(*e.g.* [LMN04, KT09, BJSK12]) did not tackle the issue of ensuring a conclusive verdict during the test case executions, focusing more on the generation of the said test cases. Later papers (*e.g.* [DLLN08a, DLLN09]) added this dimension, using game formalisms to reason about the test case executions.

Compared to [Ram98], the model of TAIO is much more complex than finite transition systems, the test purposes are also much more powerful than simple sub-sequences of the specification considered in that work, thus even if the approach is similar, the game has to be completely revised. Furthermore, we present some fairness assumptions that identify a reasonable restriction of the implementation behaviours, under which our strategies are winning. Recently another approach to strategies beyond winning states close to our own has recently been proposed also for untimed models [vdB20]. Even though differently expressed with a 3-player game (the additional one is a sort of scheduler), it is very similar to [Ram98], taking control losses into account (as "jokers" in so-called joker games) while minimizing them, and computing ranks using distance and number of control losses. Apart from the model, another difference with our own work and with [Ram98] is that they do not consider uncontrollable cycles, relying on the scheduler to win.

Other than the papers on which we build the testing framework ([BJSK12, BSJK15]) the works most related to ours is the aforementioned approach to test of timed automata using games[DLLN08a]. Partial observation is also addressed in [DLLN09] with a variant of the TA model where observations are described by observation predicates, composed of a set of locations together with clock constraints. Test cases are then synthesized as winning strategies, if they exist, in a game between the specification and its environment that tries to guide the system to satisfy the test purpose. Our model is a bit different to the one of [DLLN09], since we do not rely on observation predicates which require the observability of clock values, but partial observation comes from internal actions and non-determinism. While our approach handles non-deterministic specifications and test purposes (thanks to the determinization game presented in [BSJK15]), at some point in this work we do require exact determinization to make sure to avoid inconclusive verdicts with the combined help of restart actions. We will however explain what happens when relaxing these assumptions. In comparison, [DLLN09] avoids determinizing TAs, relying on the determinization of a finite state model, thanks to a projection on a finite set of observable predicates. Cooperative strategies of [DLLN08a] have similarities with our fairness assumptions, but their models are assumed deterministic, and their strategies can not count the number of control losses that will be faced, as they simply acquire complete control in their cooperative phase. However their strategies can handle inconclusive verdicts. In a continuation of the present work described in Léo Henry's PhD thesis [Hen21], rank-lowering strategies are also extended to handle inconclusive states, under the name *safety-first rank-lowering strategies.*

Controllability is a major concern in testing, with multiple levels, meanings. In the context of testing from Finite State Machines or Mealy machines, one often assumes that automata are *controllable* in the sense that they are both deterministic and the inputs uniquely determine the outputs. This is also the case for the timed I/O automata of [SVD01]. We do not make these assumptions for timed automata since we want to consider the case where the response of the system to inputs is unpredictable. In the context of distributed testing (see *e.g.* [HMN16]) some *controllability problems* arise when a tester cannot anticipate when to send some input that should follow a given output, because this output occurred at a distant interface. This is different from the problem we consider here.

Our work has links with the control of discrete event systems à la Ramadge and Wonham [RW89] and controller synthesis, *e.g.* for timed automata [AMPS98c]. In the control of discrete event systems, given a reachability property, one wants to generate the most permissive controller, *i.e.*, the sub-behaviour where reachability is ensured. Is it obtained by recursively eliminating states from which some uncontrollable actions lead to states where reachability is violated, or lead to eliminated states. This controller does not always exist. The controller synthesis is even stronger, because one wants to generate a winning strategy, *i.e.*, the reachability should be effective with a strategy that allows the player to reach the goal whatever the system does. Interpreting outputs as uncontrollable actions, test generation is weaker, since one has to accept to lose and go to inconclusive verdicts where reachability is violated. In our work we first assume that inconclusive verdicts do not exist by the existence of restart actions. Seeing test cases as strategies for reachability properties, we will accept to lose (or rely on the system cooperation to win), but we try minimize those situations along strategies.

1.7 Paper organization

Compared to the previous version [HJM18] of this work, we include the full proofs of our statements, and a forward symbolic algorithm to perform the test-case synthesis, inspired from the work on UPPAAL [CDF$^+$05].

The paper is organized as follows. Section 2 introduces basic models: TAs, TAIOs and their open counterparts OTAs, OTAIOs. Section 3 is dedicated to the testing framework with hypothesis on models of testing artifacts, the conformance relation and the construction of the *objective-centered tester* that denotes both non-conformant traces and the goal to reach according to a test purpose. Section 4 constitutes the core of the paper and the main contribution. After introducing timed game automata (TGA), the test-synthesis problem is interpreted as a timed game on the objective-centered tester. Rank-lowering strategies are proposed as candidate test cases, and a fairness assumption is introduced to make such strategies win. Then properties of test cases with respect to conformance are proved. Finally Section 5 presents the algorithms used to compute a machine-compatible symbolic representation of a strategy and some interesting properties of these algorithms.

2 Timed automata and extensions

In this section, we introduce our models for timed systems, along with some useful notions and operations.

2.1 Timed automata with inputs and outputs

Timed automata (TAs) [AD94] are one of the most widely-used classes of models for reasoning about computer systems subject to real-time constraints. Timed automata are finite-state automata augmented with real-valued variables (called *clocks*) to constrain the occurrence of transitions along executions. In order to adapt these models to the testing framework, we consider TAs with inputs and outputs (TAIOs), in which the alphabet of actions is split between input, output and internal actions (the latter being used to model partial observability). We also use a variation of TAs (and TAIOs) called *open* TAs (and open TAIOs) [BJSK12], in which a distinguished subset of clocks, named *observed* clocks, whose values can only observed by guards but cannot be controlled with resets. They will be usefull to specify test purposes, a sort of *observers* describing those behaviours of a TA (or TAIO) that require testing. TAs (and TAIOS) will be viewed as particular cases with no observed clocks.

Given a finite set of *clocks* X, a *clock valuation* over X is a function $v\colon X \to \mathbb{R}_{\geq 0}$. We denote by $\mathbf{0}_X$ (and often omit to mention X when clear from the context) the valuation assigning 0 to all clocks in X. Let v be a clock valuation; for any $t \in \mathbb{R}_{\geq 0}$, we denote by $v + t$ the valuation mapping each clock $x \in X$ to $v(x) + t$, and for a subset $X' \subseteq X$, we write $v_{[X' \leftarrow 0]}$ for the valuation mapping all clocks in X' to 0, and all clocks in $X \setminus X'$ to their values in v.

A *clock constraint* is a finite conjunction of atomic constraints of the form $x \sim n$ where $x \in X$, $n \in \mathbb{N}$, and $\sim \in \{<, \leq, =, \geq, >\}$. That a valuation v satisfies a clock constraint g, written $v \models g$, is defined in the obvious way. We write $\mathcal{G}(X)$ for the set of clock constraints over X.

Definition 1. An *open timed automaton* (OTA) is a tuple[3] $\mathcal{A} = (L^{\mathcal{A}}, l_0^{\mathcal{A}}, \Sigma^{\mathcal{A}}, X_p^{\mathcal{A}} \uplus X_o^{\mathcal{A}}, I^{\mathcal{A}}, E^{\mathcal{A}})$ where:

- $L^{\mathcal{A}}$ is a finite set of *locations*, with $l_0^{\mathcal{A}} \in L^{\mathcal{A}}$ the *initial location*;
- $\Sigma^{\mathcal{A}}$ is a finite alphabet;
- $X^{\mathcal{A}} = X_p^{\mathcal{A}} \uplus X_o^{\mathcal{A}}$ is a finite set of clocks, partitioned into *proper clocks* $X_p^{\mathcal{A}}$ and *observed clocks* $X_o^{\mathcal{A}}$; only proper clocks may be reset along transitions; we denote $M^{\mathcal{A}}(x)$ the maximal constant to which x is compared;
- $I^{\mathcal{A}}\colon L^{\mathcal{A}} \to \mathcal{G}(X^{\mathcal{A}})$ assigns invariant constraints to locations;
- $E^{\mathcal{A}} \subseteq L^{\mathcal{A}} \times \mathcal{G}(X^{\mathcal{A}}) \times \Sigma^{\mathcal{A}} \times 2^{X_p^{\mathcal{A}}} \times L^{\mathcal{A}}$ is a finite set of *transitions*; for $e = (l, g, a, X'_p, l') \in E^{\mathcal{A}}$, we write $\mathsf{act}(e) = a$.

Intuitively, proper clocks X_p are the usual ones, controlled by $\mathcal{A}$ through resets, while observed clocks X_o can only be observed by $\mathcal{A}$ through guards and invariants. The later belong to (are proper clocks of) another OTA $\mathcal{B}$ that controls them, and with which $\mathcal{A}$ is synchronized by product (see below).

An Open Timed Automaton with Inputs and Outputs (OTAIO) is an OTA in which $\Sigma^{\mathcal{A}} = \Sigma_?^{\mathcal{A}} \uplus \Sigma_!^{\mathcal{A}} \uplus \Sigma_\tau^{\mathcal{A}}$ is the disjoint union of *input actions* in $\Sigma_?^{\mathcal{A}}$ (denoted by $?a$, $?b$, ...), *output actions* in $\Sigma_!^{\mathcal{A}}$ (denoted by $!a$, $!b$, ...), and *internal actions* in $\Sigma_\tau^{\mathcal{A}}$ (denoted by τ_1, τ_2, ...). We write $\Sigma_{\mathsf{obs}} = \Sigma_? \uplus \Sigma_!$ for the alphabet of *observable* actions. Finally, a Timed Automaton (TA), as defined in [AD94], (resp. a Timed Automaton with Inputs and Outputs (TAIO)) is an OTA (resp. an OTAIO) with no observed clocks.

TAIOs will be sufficient to model most objects of the testing framework, but the ability of OTAIOs to observe other clocks will be essential to specify test purposes (see Section 3.1), which need to synchronize with the specification to focus on behaviours that need to be tested. The semantics of a OTA is defined as follows:

Definition 2. The *semantics* of an OTA $\mathcal{A} = (L^{\mathcal{A}}, l_0^{\mathcal{A}}, \Sigma^{\mathcal{A}}, X_p^{\mathcal{A}} \uplus X_o^{\mathcal{A}}, I^{\mathcal{A}}, E^{\mathcal{A}})$ is the infinite-state transition system $\mathcal{T}^{\mathcal{A}} = (S^{\mathcal{A}}, s_0^{\mathcal{A}}, \Gamma^{\mathcal{A}}, \to^{\mathcal{A}})$ where:

- $S^{\mathcal{A}} = \{(l, v) \in L^{\mathcal{A}} \times \mathbb{R}_{\geq 0}^X \mid v \models I(l)\}$ is the (infinite) set of *configurations* (or states) of $\mathcal{A}$; with initial configuration $s_0^{\mathcal{A}} = (l_0, \mathbf{0}_X)$.
- $\Gamma^{\mathcal{A}} = \mathbb{R}_{\geq 0} \uplus (E^{\mathcal{A}} \times 2^{X_o^{\mathcal{A}}})$ is the set of *transitions labels*;

[3] For this and the following definitions, we may omit to mention superscripts when the corresponding automaton is clear from the context.

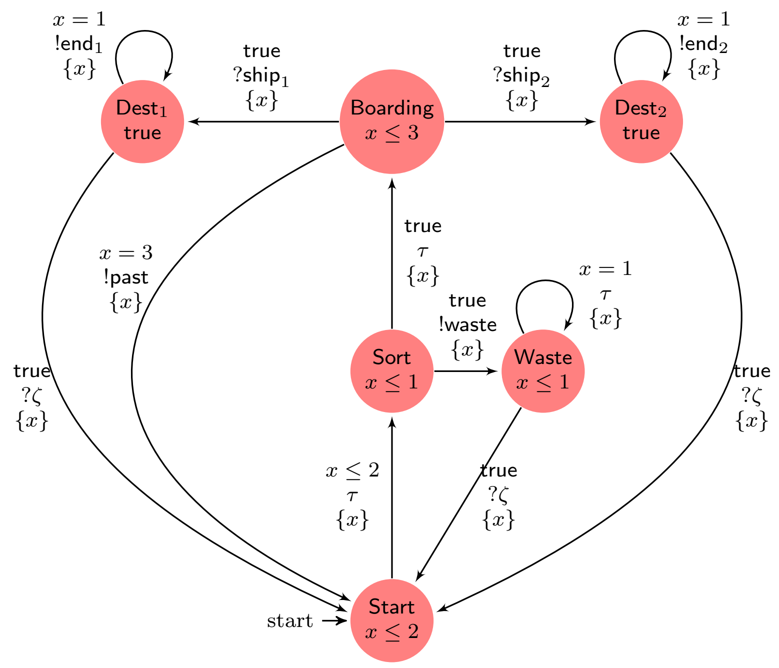


Fig. 5 A TAIO specifying a conveyor belt.

- $\rightarrow^{\mathcal{A}} \subseteq S^{\mathcal{A}} \times \Gamma^{\mathcal{A}} \times S^{\mathcal{A}}$ is the *transition relation*. It is defined as the union of
  - *time elapses*, which are triples $((l, v), t, (l, v + t)) \in S^{\mathcal{A}} \times \mathbb{R}_{\geq 0} \times S^{\mathcal{A}}$. By definition of $S^{\mathcal{A}}$ and by convexity of clock constraints all intermediary valuations $v + t'$ with $0 \leq t' \leq t$, satisfy the invariant $I(l)$;
  - *discrete moves*, which are all triples $((l, v), (e^{\mathcal{A}}, X_o'^{\mathcal{A}}), (l', v')) \in S^{\mathcal{A}} \times (E^{\mathcal{A}} \times 2^{X_o^{\mathcal{A}}}) \times S^{\mathcal{A}}$ such that, writing $e = (l_e, g, a, X'_p, l'_e)$, it holds $l_e = l$, $l'_e = l'$, $v \models g$, and $v' = v_{[X'_p \cup X'_o \leftarrow 0]}$. Again, by definition, $v \models I(l)$ and $v' \models I(l')$.

Notice that an OTA has no control over its observed clocks, the intention being to synchronize them later in a product (see Def. 3). Hence, in the semantics of a single OTA, when a discrete transition is taken, observed clocks are unconstrained and any subset X'_o of X_o may be reset. When dealing with (closed) TAs, where X_o is empty, we may write $(l, v) \xrightarrow{e} (l', v')$ in place of $(l, v) \xrightarrow{(e,\emptyset)} (l', v')$.

Example 3. Fig. 5 is an example of TAIO specifying a conveyor belt such as the one of Example 1. It uses one proper clock x (no observed one) that is used in invariants to bound the sojourn time in locations. It has as inputs ?ship$_1$ and ?ship$_2$, and a special restart action ?ζ (see later), outputs !end$_1$, !end$_2$, !past, !waste, and internal action τ. After a maximum of 2 time units in location *Start* (depending for example on their weight), packages reach a sorting point in location *Sort*, where they are automatically sorted between packages to reject and packages to ship. Packages to reject go to location *Waste*, while packages to ship are sent to a boarding platform (location *Boarding*), where an operator can send them to two different destinations Dest$_1$ or Dest$_2$. If the operator takes more than 3 time units to select a destination, the package goes past the boarding platform and restarts the process. The restart action $?\zeta$ also allows to go back to *Start* from several locations.

From the semantics of OTAs we first define runs. A *partial run* of $\mathcal{A}$ is a (finite or infinite) sequence $\rho = ((s_i, \gamma_i, s_{i+1}))_{1 \leq i < n}$ of transitions in $\mathcal{T}^{\mathcal{A}}$, with $n \in \mathbb{N} \cup \{+\infty\}$. We write $\mathsf{first}(\rho)$ for s_1 and, when $n \in \mathbb{N}$, $\mathsf{last}(\rho)$ for s_n. A *run* is a partial run starting in the initial configuration $s_0^{\mathcal{A}}$. The duration of ρ is $\mathsf{dur}(\rho) = \sum_{\gamma_i \in \mathbb{R}_{\geq 0}} \gamma_i$.

A finite run is *accepted* in a set of locations $F \subseteq L^{\mathcal{A}}$ if its last configuration belongs to $F \times \mathbb{R}_{\geq 0}$. We denote by $\mathsf{Run}(\mathcal{A})$ the set of runs of $\mathcal{A}$, $\mathsf{Run}_F(\mathcal{A})$ the subset of runs accepted in F (note that $\mathsf{Run}(\mathcal{A}) = \mathsf{Run}_{L^{\mathcal{A}}}(\mathcal{A})$), and $\mathsf{pRun}(\mathcal{A})$ the set of partial runs.

In the sequel, as usual we only consider infinite runs that are non-zeno, *i.e.*, have an infinite duration. This may restrict the set of theoretically possible runs, but only suppresses runs that have no interest for test-generation: either a zeno run expresses a finite delay as an infinite sum of delays converging to that value - which is clearly a mathematical artifact - or realises an infinite number of discrete actions in finite time, which is impossible for most systems under test in practice.

Configuration s is *reachable* from configuration s' when there exists a partial run from s' to s in $\mathcal{T}^{\mathcal{A}}$. We write $\mathsf{Reach}(\mathcal{A}, S')$ for the set of configurations that are reachable from some configuration in the set S', and $\mathsf{Reach}(\mathcal{A})$ for $\mathsf{Reach}(\mathcal{A}, \{s_0^{\mathcal{A}}\})$.

From runs, two other notions are obtained by consecutive projections. The *(partial) signature* associated with a (partial) run $\rho = ((s_i, \gamma_i, s'_i))_i$ is its projection $\mathsf{sig}(\rho) = (\mathsf{proj}(\gamma_i))_i \in (\mathbb{R}_{\geq 0} \cup (\Sigma \times 2^{X_p \cup X_o}))^*$, where only durations or pairs of actions and resets are kept, *i.e.*, $\mathsf{proj}(\gamma) = \gamma$ if $\gamma \in \mathbb{R}_{\geq 0}$, and $\mathsf{proj}(\gamma) = (a, X'_p \cup X'_o)$ if $\gamma = ((l, g, a, X'_p, l'), X'_o)$. We write $\mathsf{pSig}(\mathcal{A}) = \mathsf{proj}(\mathsf{pRun}(\mathcal{A}))$ and $\mathsf{Sig}(\mathcal{A}) = \mathsf{proj}(\mathsf{Run}(\mathcal{A}))$ for the sets of (partial) signatures of $\mathcal{A}$, and $\mathsf{Sig}_F(\mathcal{A}) = \mathsf{proj}(\mathsf{Run}_F(\mathcal{A}))$ for the subset of signatures accepted in F. We write $s \xrightarrow{\mu} s'$ when there exists a (partial) finite run ρ such that $\mu = \mathsf{proj}(\rho)$, $\mathsf{first}(\rho) = s$ and $\mathsf{last}(\rho) = s'$, and write $\mathsf{dur}(\mu) = \sum_{\gamma_i \in \mathbb{R}_{\geq 0}} \gamma_i$ its duration. Observe that as expected, when $\mathsf{sig}(\rho) = \mu$, it holds $\mathsf{dur}(\rho) = \mathsf{dur}(\mu)$. We write $s \xrightarrow{\mu}$ when $s \xrightarrow{\mu} s'$ for some s'.

If $\mathcal{A}$ is a TAIO, the *trace* of a (partial) signature corresponds to what can be observed by the environment, namely the sequence of delays and observable actions (internal actions are ignored) . It is defined by the limit of the following inductive definition, for $t, t' \in \mathbb{R}_{\geq 0}$, $a, b \in \Sigma$, $X' \subseteq X$, and partial signatures μ_1, μ_2:

$$\begin{aligned}
\mathsf{trace}(\varepsilon) &= 0 \\
\mathsf{trace}(t) &= t \\
\mathsf{trace}((a, X')) &= a \text{ if } a \in \Sigma_{\mathsf{obs}},\ 0 \text{ otherwise} \\
\mathsf{trace}(\mu_1 \cdot \mu_2) &= \mathsf{trace}(\mu_1) \cdot \mathsf{trace}(\mu_2)
\end{aligned}$$

with the concatenation of traces being defined with two special cases as follow:

$$\begin{aligned}
t \cdot t' &= t + t' \\
a \cdot b &= a \cdot 0 \cdot b\ .
\end{aligned}$$

Remark 1. This definition ensures that traces are always alternating, starting from a delay. The interest is to enforce a structure corresponding to an observation.

For example, for $\mu_1 = 0 \cdot a \cdot 0.7 \cdot \tau \cdot 0.3$ and $\mu_2 = 0.1 \cdot b$, we get $\mu_1 \cdot \mu_2 = 0 \cdot a \cdot 1.1 \cdot b$. We write $\mathsf{Traces}(\mathcal{A}) = \bigcup_{\mu \in \mathsf{Sig}(\mathcal{A})} \mathsf{trace}(\mu)$ for the set of traces corresponding to runs of $\mathcal{A}$, $\mathsf{pTraces}(\mathcal{A})$ for the set of traces corresponding to partial runs, and $\mathsf{Traces}_F(\mathcal{A})$ for the subset of traces of runs accepted in F. Two OTAIOs are said to be *trace-equivalent* if they have the same sets of traces, *i.e.*, if they exhibit the same observable behaviours. We furthermore define, for an OTAIO $\mathcal{A}$, a trace σ and a configuration s:

- $\mathcal{A}$ after $\sigma = \{s \in S \mid \exists \mu \in \mathsf{Sig}(\mathcal{A}), s_0 \xrightarrow{\mu} s \wedge \mathsf{trace}(\mu) = \sigma\}$ is the set of all configurations that can be reached when the trace σ has been observed from $s_0^{\mathcal{A}}$;
- $\mathsf{enab}(s) = \{e \in E^{\mathcal{A}} \mid s \xrightarrow{e}\}$ is the set of transitions enabled in s;
- $\mathsf{elapse}(s) = \{t \in \mathbb{R}_{\geq 0} \mid \exists \mu \in (\mathbb{R}_{\geq 0} \cup (\Sigma_\tau \times 2^X))^*, s \xrightarrow{\mu} \wedge \mathsf{dur}(\mu) = t\}$ is the set of delays that can be observed from location s without observing any action;
- $\mathsf{out}(s) = \{a \in \Sigma_! \mid \exists e \in \mathsf{enab}(s), \mathsf{act}(e) = a\} \cup \mathsf{elapse}(s)$ is the set of possible outputs and delays that can be observed from s. For $S' \subseteq S$, we write $\mathsf{out}(S') = \bigcup_{s \in S'} \mathsf{out}(s)$;
- $\mathsf{in}(s) = \{a \in \Sigma_? \mid \exists e \in \mathsf{enab}(s), \mathsf{act}(e) = a\}$ is the set of possible inputs that can be proposed when arriving in s. For $S' \subseteq S$, we write $\mathsf{in}(S') = \bigcup_{s \in S'} \mathsf{in}(s)$.

We now define some useful sub-classes of OTAIOs. An OTAIO $\mathcal{A}$ is said

- *deterministic* if for any pair of transitions $e_1 = (l, g_1, a, X'_{p1}, l'_1)$ and $e_2 = (l, g_2, a, X'_{p2}, l'_2)$ starting in the same location and sharing the same action, either $e_1 = e_2$ or $g_1 \cap g_2 = \emptyset$; this entails that for any $\sigma \in \mathsf{Traces}(\mathcal{A})$, $\mathcal{A}$ after σ is a singleton;
- *determinizable* if there exists a trace-equivalent deterministic OTAIO;
- *complete* if any action can be played from any configuration, formally $S = L \times \mathbb{R}_{\geq 0}^X$ (*i.e.*, all invariants are always true) and for any location $l \in L$ and action $a \in \Sigma$, $\bigvee_{(l,g,a,X',l') \in E^{\mathcal{A}}} g = \mathsf{true}$;
- *input-complete* if any input can be taken from any configuration, formally for any location $l \in L$ and action $a \in \Sigma_?$, $\bigvee_{(l,g,a,X',l') \in E^{\mathcal{A}}} g = \mathsf{true}$;

The following definition of the product of two OTAIOs extends the classical product of TAs. The intention is that the product of two OTAIOs corresponds to the intersection of the signatures of the individual OTAIOs, *i.e.*, $\mathsf{Sig}(\mathcal{A} \times \mathcal{B}) = \mathsf{Sig}(\mathcal{A}) \cap \mathsf{Sig}(\mathcal{B})$ [BSJK15]. Moreover if TAs are equipped with accepting locations $F^{\mathcal{A}} \subseteq L^{\mathcal{A}}$ and $F^{\mathcal{B}} \subseteq L^{\mathcal{B}}$, we also get $\mathsf{Sig}_{F^{\mathcal{A}} \times F^{\mathcal{B}}}(\mathcal{A} \times \mathcal{B}) = \mathsf{Sig}_{F^{\mathcal{A}}}(\mathcal{A}) \cap \mathsf{Sig}_{F^{\mathcal{B}}}(\mathcal{B})$.

Notice that, in this definition, proper clocks of the product are proper clocks of one of the operands, while observed clocks are observed clocks of an operand which are not proper of the other. Two transitions synchronize when they carry a common action; the resulting guard is the intersection of the guards of the operands, and clock resets are the union of proper resets of the operands, thus proper themselves. Notice also that if a proper clock of $\mathcal{A}$ is an observed clock of $\mathcal{B}$, its evolution is observed through guards of $\mathcal{B}$, and vice versa. Formally:

Definition 3. Given two OTAIOs $\mathcal{A} = (L^{\mathcal{A}}, l_0^{\mathcal{A}}, \Sigma, X_p^{\mathcal{A}} \uplus X_o^{\mathcal{A}}, I^{\mathcal{A}}, E^{\mathcal{A}})$ and $\mathcal{B} = (L^{\mathcal{B}}, l_0^{\mathcal{B}}, \Sigma, X_p^{\mathcal{B}} \uplus X_o^{\mathcal{B}}, I^{\mathcal{B}}, E^{\mathcal{B}})$ over the same alphabet Σ, their *product* is the OTAIO $\mathcal{A} \times \mathcal{B} = (L^{\mathcal{A}} \times L^{\mathcal{B}}, (l_0^{\mathcal{A}}, l_0^{\mathcal{B}}), \Sigma, (X_p^{\mathcal{A}} \cup X_p^{\mathcal{B}}) \uplus ((X_o^{\mathcal{A}} \cup X_o^{\mathcal{B}} \setminus (X_p^{\mathcal{A}} \cup X_p^{\mathcal{B}})), I, E)$

where $E\colon (l_1, l_2) \mapsto I^{\mathcal{A}}(l_1) \wedge I^{\mathcal{B}}(l_2)$ and E is the (smallest) set such that for each $(l^1, g^1, a, X_p'^1, l'^1) \in E^{\mathcal{A}}$ and $(l^2, g^2, a, X_p'^2, l'^2) \in E^{\mathcal{B}}$, E contains $((l^1, l^2), g^1 \wedge g^2, a, X_p'^1 \cup X_p'^2, (l'^1, l'^2))$.

2.2 Regions and zones

The semantics $\mathcal{T}^{\mathcal{A}}$ of a timed automaton $\mathcal{A}$ is an infinite graph. While it does not raise any issues in most of the reasoning made on them, a *finite* representation is sometimes necessary to handle them in practice. Such representations are usually based on *zones* [DT98], *i.e.*, finite conjunctions of clock constraints and comparisons. To simplify the notations of zones, we introduce a clock x_0 such that for all valuations v, $v(x_0) = 0$. It will be used to unify the comparison to clocks and constants.

Definition 4. For a set of clocks $X = \{x_1, ..., x_n\}$, a set of valuations Z is called a *zone* if it can be described as $\bigwedge_{i,j \in (0,n),\ i \neq j} (x_i - x_j \prec d_{ij})$ with $d_{ij} \in \mathbb{Z}$ and $\prec \in \{\leq, <\}$.

Zones present the advantage to dispose of an efficient representation: the difference bounded matrices [BY04, Min17]. They generalize the original finite abstraction based on *regions* [AD94], which is known to be exact and intuitively represents minimal zones. Regions can be defined as the equivalence classes of the following equivalence relation:

Definition 5. Let $\mathcal{A}$ be a TA with the set of clocks X and maximal constants $M^{\mathcal{A}}(x)$ for $x \in X$. The *regions* of $\mathcal{A}$ denoted by $\mathcal{R}_{\mathcal{A}}$ are the equivalence classes of the following relation $\approx$ between valuations: $v \approx v'$ if and only if the following conditions are met:

1. $\forall x \in X,\ v(x) > M^{\mathcal{A}}(x) \Leftrightarrow v'(x) > M^{\mathcal{A}}(x)$;
2. $\forall x \in X,\ v(x) \leq M^{\mathcal{A}}(x) \Rightarrow \big(\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor \wedge \big(\{v(x)\} = 0 \Leftrightarrow \{v'(x)\} = 0\big)\big)$;
3. $\forall x, x' \in X,\ \big(v(x) \leq M^{\mathcal{A}}(x) \wedge v(x') \leq M^{\mathcal{A}}(x')\big) \Rightarrow$
   $(\{v(x)\} \leq \{v(x')\} \Leftrightarrow \{v'(x)\} \leq \{v'(x')\})$.

where $\lfloor \cdot \rfloor$ is the integer part and $\{\cdot\}$ the fractional part.

The notion of enabled transitions and delay transitions are extended to regions as follows: for a region *reg*, we let $\mathsf{enab}((l, \mathit{reg})) = \mathsf{enab}((l, v))$ for any v in *reg*[4]; for any transition e, we let $\mathsf{next}(e, \mathit{reg})$ be the unique region reg' such that for all $s \in \mathit{reg}$, $s \xrightarrow{e} s'$ implies $s' \in \mathit{reg}'$, and we write $\mathsf{SucTemp}(\mathit{reg})$ for all strict time-successor regions reg' of *reg*. We also extend the notion of execution to regions[5].

Fig. 6 presents the regions and a zone for clocks $X = \{x, y\}$, $M(x) = 3$ and $M(y) = 2$. Each dot, each line segment and each open *minimal* polygon (not containing any other zone) is a region, while the highlighted set is the zone $1 \leq x \leq 2 \wedge -1 \leq x - y \leq 1$.

The standard symbolic representation of a set of configurations a TA is made of pairs (l, Z) of locations and zones. We will often abuse notations and write Z for (l, Z) when the context is clear.

[4] This definition is valid because Condition 2 in Def. 5 ensures that guards cannot distinguish between valuations in a given region. Hence the enabled transitions are the same.

[5] Observe that writing $\mathit{reg} \xrightarrow{t} \mathit{reg}'$ for a delay t is execution-specific, as that delay may lead to a region $\mathit{reg}'' \neq \mathit{reg}'$ from some configurations in *reg*.

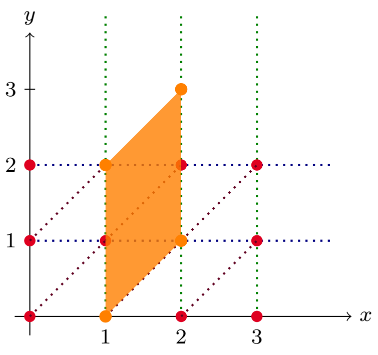


Fig. 6 Regions and a zone for $M(x) = 3$ and $M(y) = 2$.

3 The testing framework for timed automata

We now present the testing framework, defining *(i)* the main testing artifacts, *i.e.*, specifications, implementations, test purposes, and test cases, along with the assumptions we put on them; *(ii)* a conformance relation relating implementations and specifications. The combination of the test purposes and the specification and the construction of an approximate deterministic tester is explained afterwards.

This framework is almost taken from the one of [BJSK12], but we repeat it here for completeness. We thus get almost the same properties on the objects that are built. The main difference is the assumption on restart transitions in specifications and test purposes that ensures strong connectivity of the objective-centered tester built from them.

3.1 Timed testing context

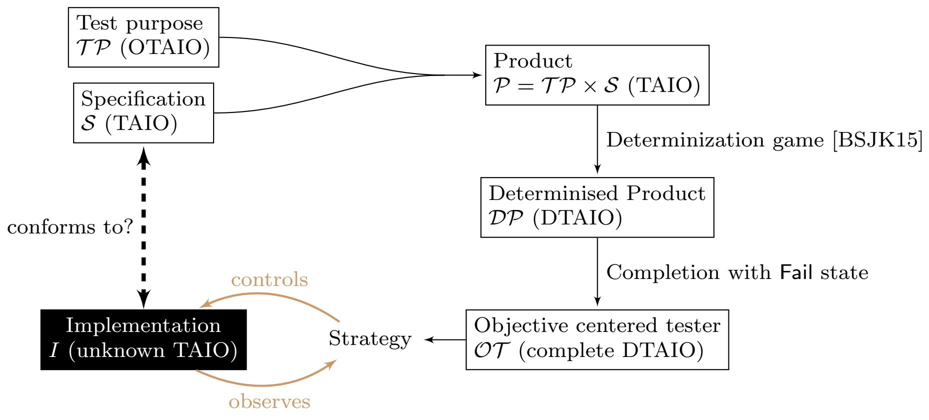


Fig. 7 The testing framework.

Fig. 7 refines the framework described in the introduction, Fig. 2 for the context of timed testing. We use TAIOs as models for the specification $\mathcal{S}$ and possible black-box implementations $\mathcal{I}$ of them. Conformance is described by the now standard tioco relation [KT09] that considers discrepancies between their respective traces. Checking whether $\mathcal{I}$ conforms to $\mathcal{S}$ is realized by test cases, here seen as strategies of a game between the tester and the implementation. In our context, this game and test cases are derived from the specification $\mathcal{S}$ and test purposes $\mathcal{TP}$, some sorts of observers specified as OTAIOS, used to focus on particular behaviours one wants to test. Expected verdicts should thus indicate non-conformance (***Fail***) or realization of the test purpose (***Pass***). While monitoring non-conformance, the strategy tries to control $\mathcal{I}$ toward the target of the test purpose, by proposing controllable inputs after some delays, and observes the responses of $\mathcal{I}$ in the form of delays and uncontrollable outputs, and this iteratively until delivering a verdict. The construction of test cases is as follows. From $\mathcal{S}$ and $\mathcal{TP}$, their product $\mathcal{P}$ is built, and a deterministic TAIO (DTAIO) $\mathcal{DP}$ by determinization of $\mathcal{P}$. $\mathcal{DP}$ characterizes traces of $\mathcal{S}$ and among them, those targeted by $\mathcal{TP}$. Determinization is needed simply because resulting test case executions need to be deterministic since tioco relies on traces. It relies on the approach of [BSJK15] that will not be detailed here, we only stick on the properties of the resulting $\mathcal{DP}$. Adding ***Fail*** by completion from $\mathcal{DP}$ produces the *objective centered tester* which is interpreted as a game from which the test cases are derived as strategies. We now detail the framework and some properties we get.

First we formalise specifications with TAIOs, equiped with *restart transitions*, corresponding to a kind of system shutdown and restart, and assume that from any (reachable) configuration, a restart transition is always reachable. In essence this property ensures that we always keep a chance to restart from the initial state. We also assume that specifications are:

- *non-blocking*, *i.e.*, do not block time waiting for an input, formally, for any $s \in S$ and any non-negative real t, there is a partial run ρ from s involving no input actions (*i.e.*, $\mathsf{proj}(\rho)$ is a sequence over $\mathbb{R}_{\geq 0} \cup (\Sigma_! \cup \Sigma_\tau) \times 2^X$) and such that $\mathsf{dur}(\rho) = t$. Intuitively an implementation that could stop time to wait for an input that it does not control is not desirable. The non-blocking hypothesis applies to both implementations and specifications, and thus rules out specifications having no conformant physically-possible implementation.
- *repeatedly observable*, *i.e.*, from any state some action can be observed, formally from any $s \in S^{\mathcal{A}}$, there exists a partial run ρ from $s$such that $\mathsf{trace}(\rho) \notin \mathbb{R}_{\geq 0}$. Repeated-observability will be useful for technical reasons, when analyzing exhaustiveness of test cases. It intuitively ensures that some output will always eventually be visible, allowing to detect the current configuration of the system.

Definition 6. A *specification with restarts* (or simply *specification*) on $(\Sigma_?, \Sigma_!, \Sigma_\tau)$ is a non-blocking, repeatedly-observable TAIO $\mathcal{S} = (L^{\mathcal{S}}, l_0^{\mathcal{S}}, (\Sigma_? \uplus \Sigma_! \uplus \Sigma_\tau, X_p^{\mathcal{S}}, I^{\mathcal{S}}, E^{\mathcal{S}})$, where $\zeta \in \Sigma_?$ is the restart action. We assume that transitions carrying ζ all belong to $(L^{\mathcal{S}} \times \mathcal{G}_{M^{\mathcal{S}}}(X^{\mathcal{S}}) \times \{\zeta\} \times \{X_p^{\mathcal{S}}\} \times \{l_0^{\mathcal{S}}\})$ (*i.e.*, they reset all clocks and go back to $l_0^{\mathcal{S}}$) and from any reachable configuration, there exists a finite partial execution containing ζ, *i.e.*, for any $s \in \mathsf{Reach}(\mathcal{S})$, there exists μ s.t. $s \xrightarrow{\mu \cdot \zeta} s_0^{\mathcal{S}}$.

The assumption on ζ-transitions directly entails that $Reach(\mathcal{T}_{\mathcal{S}})$ is strongly-connected. Being strongly connected ensures that there is always a possibility to reach any (reachable) configuration, and will help us ensure that a strategy always leads to a conclusive verdict, as it cannot be stuck in a situation where no path exists to a reachability goal. We discuss the consequences of releasing the strong connectivity hypothesis in Remark 2.

In practice, conformance testing links a mathematical model, the specification, and a black-box implementation, that is a real-life *physical* object observed by its interactions with the environment. As usual, in order to formally reason about conformance, one needs to bridge the gap between the mathematical world and the physical world. We then assume that the real implementation has the same behavior as an unknown TAIO that we call *implementation*, and has the same interface as the specification $\mathcal{S}$.

Definition 7. Let $\mathcal{S} = (L^{\mathcal{S}}, l_0^{\mathcal{S}}, \Sigma_? \uplus \Sigma_! \uplus \Sigma_\tau, X_p^{\mathcal{S}}, I^{\mathcal{S}}, E^{\mathcal{S}})$ be a specification with $\zeta \in \Sigma_?$. An *implementation* of $\mathcal{S}$ is an input-complete and non-blocking TAIO $\mathcal{I} = (L^{\mathcal{I}}, l_0^{\mathcal{I}}, \Sigma_? \uplus \Sigma_! \uplus \Sigma_\tau^{\mathcal{I}}, X_p^{\mathcal{I}}, I^{\mathcal{I}}, E^{\mathcal{I}})$. We denote by $\mathcal{I}(\mathcal{S})$ the set of possible implementations of $\mathcal{S}$.

The input-completeness and non-blocking hypotheses made on implementations are not restrictions, but model real-world contingencies: the environment might always provide any input and the system cannot alter the course of time. We do not assume that resets are correctly implemented, *i.e.*, that they reset all clocks and go back to $l_0^{\mathcal{I}}$. Indeed, as we will see below, since conformance only considers traces, the effect of incorrect resets could be later detected by a trace discrepancy.

Having defined the necessary objects, it is now possible to introduce the *timed input-output conformance* (tioco) relation [KT09]. Intuitively, it can be understood as "after any specified timed trace, outputs and delays of the implementation should be specified".

Definition 8. Let $\mathcal{S}$ be a specification and $\mathcal{I} \in \mathcal{I}(\mathcal{S})$. We say that $\mathcal{I}$ *conforms to* $\mathcal{S}$ for tioco, and write $\mathcal{I}$ tioco $\mathcal{S}$ when:

$$\forall \sigma \in \mathsf{Traces}(\mathcal{S}), \mathsf{out}(\mathcal{I} \text{ after } \sigma) \subseteq \mathsf{out}(\mathcal{S} \text{ after } \sigma)$$

3.2 Selecting behaviours with test purposes

In practice, test purposes are used to describe the intention behind test cases, typically some behaviours one wants to test because they describe basic functionalities that must be correct and/or because a non-conformance is suspected. For automatic test case synthesis, we formalize them with OTAIOs. These are non-intrusive observers of the specification (its actions and clocks) with accepting locations used to define those behaviours to be tested.

Definition 9. Given a specification $\mathcal{S} = (L^{\mathcal{S}}, l_0^{\mathcal{S}}, \Sigma_? \uplus \Sigma_! \uplus \Sigma_\tau, X_p^{\mathcal{S}}, I^{\mathcal{S}}, E^{\mathcal{S}})$, with $\zeta \in \Sigma_?$, a *test purpose for* $\mathcal{S}$ is a pair $(\mathcal{TP}, \mathsf{Accept}^{\mathcal{TP}})$ where $\mathcal{TP} = (L^{\mathcal{TP}}, l_0^{\mathcal{TP}}, \Sigma_? \uplus \Sigma_! \uplus \Sigma_\tau, X_p^{\mathcal{TP}} \uplus X_p^{\mathcal{S}}, I^{\mathcal{TP}}, E^{\mathcal{TP}})$ is a complete OTAIO and $\mathsf{Accept}^{\mathcal{TP}} \subseteq L^{\mathcal{TP}}$ is a subset of accepting locations; it is required that transitions carrying restart actions ζ in $\mathcal{TP}$ are of the form $(l, g, \zeta, X_p^{\mathcal{TP}}, l_0^{TP})$, *i.e.*, they reset all proper clocks and return to the initial state.

In the following, we may simply write $\mathcal{TP}$ in place of $(\mathcal{TP}, \mathsf{Accept}^{\mathcal{TP}})$. Notice that we force test purposes to be complete because they are non-intrusive observers: they should never constrain the runs of the specification they observe, but should only label those accepted behaviours to be tested. Observed clocks of $\mathcal{TP}$ correspond to the proper clocks of $\mathcal{S}$ that it observes through guards, but it cannot reset them. $\mathcal{TP}$ may have its own proper clocks that it can reset; those clocks may serve *e.g.*, to count some delays unspecified or indirectly specified in $\mathcal{S}$. The condition on restart transitions in $\mathcal{TP}$ encodes the fact that it forgets behaviours of $\mathcal{S}$ preceding ζ (and thus does not test it), and forces the test purpose to be strongly connected.

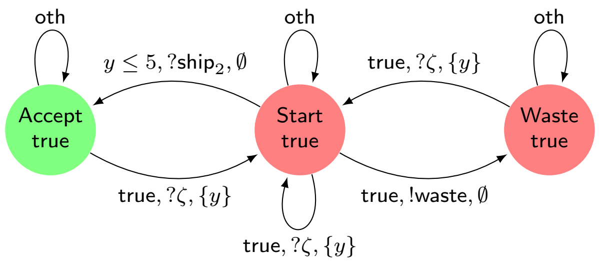


Fig. 8 A test purpose for the conveyor belt.

Example 4. Fig. 8 is a test purpose for our conveyor-belt example. It aims to test that it is possible to ship a package to destination 2 in less than 5 time units, while avoiding to visit Waste. The Accept set is limited to one location, named Accept. The test purpose has a proper clock y, and no observed clocks. We denote by oth (for *otherwise*) the set of transitions that reset no clocks, and are enabled for an action other than ζ when no other transition is possible for this action in this location. This set serves to complete the test purpose.

We now explain how the behaviours targeted by the test purpose $\mathcal{TP}$ are characterized on the specification $\mathcal{S}$ by the construction of the product OTAIO $\mathcal{P} = \mathcal{S} \times \mathcal{TP}$. Since $\mathcal{S}$ is a TAIO and the observed clocks of $\mathcal{TP}$ are exactly the clocks of $\mathcal{S}$, the product $\mathcal{P}$ is actually a TAIO. Furthermore, since $\mathcal{TP}$ is complete, $\mathsf{Sig}(\mathcal{P}) = \mathsf{Sig}(\mathcal{S})$. By defining accepting locations in the product by $\mathsf{Accept}^{\mathcal{P}} = L^{\mathcal{S}} \times \mathsf{Accept}^{\mathcal{TP}}$, we get that signatures accepted in $\mathcal{P}$ are exactly signatures of $\mathcal{S}$ accepted by $\mathcal{TP}$. Formally:

Proposition 10. *Let $\mathcal{S}$ be a specification and $\mathcal{TP}$ a test purpose on this specification, $\mathcal{P} = \mathcal{S} \times \mathcal{TP}$ their product. Then*

$$\mathsf{Sig}(\mathcal{P}) = \mathsf{Sig}(\mathcal{S}) \text{ and } \mathsf{Sig}_{\mathsf{Accept}^{\mathcal{P}}}(\mathcal{P}) = \mathsf{Sig}(\mathcal{S}) \cap \mathsf{Sig}_{\mathsf{Accept}^{\mathcal{TP}}}(\mathcal{TP})$$

Proof. Remember that the set of signatures of the product of two OTAIOs is the intersection of the signatures of the two original OTAIOs [BSJK15], so that $\mathsf{Sig}(\mathcal{S} \times \mathcal{TP}) = \mathsf{Sig}(\mathcal{S}) \cap \mathsf{Sig}(\mathcal{TP})$; since $\mathcal{TP}$ is complete (it cannot prevent any signature of $\mathcal{S}$), $\mathsf{Sig}(\mathcal{TP}) = (\mathbb{R}_{\geq 0} \cup (\Sigma \times 2^{X_p \cup X_o}))^*$, we conclude that $\mathsf{Sig}(\mathcal{S} \times \mathcal{TP}) = \mathsf{Sig}(\mathcal{S})$.

We also have $\mathsf{Sig}_{L^{\mathcal{S}} \times \mathsf{Accept}(\mathcal{TP})}(\mathcal{S} \times \mathcal{TP}) = \mathsf{Sig}_{L^{\mathcal{S}}}(\mathcal{S}) \cap \mathsf{Sig}_{Accept^{\mathcal{TP}}}(\mathcal{TP})$ thus $\mathsf{Sig}_{\mathsf{Accept}(\mathcal{P})}(\mathcal{P}) = \mathsf{Sig}(\mathcal{S}) \cap \mathsf{Sig}_{Accept^{\mathcal{TP}}}(\mathcal{TP})$. □

By projection on traces, we immediately get:

Corollary 11. *Let $\mathcal{S}$ be a specification, $\mathcal{TP}$ a test purpose, and $\mathcal{P}$ their product. $\mathcal{S}$ and $\mathcal{P}$ are trace-equivalent.*

This entails that $\mathcal{I}$ tioco $\mathcal{S}$ if, and only if, $\mathcal{I}$ tioco $\mathcal{P}$. Observe that ζ synchronize on $\mathcal{S}$ and on $\mathcal{TP}$ where it is available everywhere. This induces an extension to the product, of the fact that $Reach(\mathcal{T}_{\mathcal{S}})$ is strongly-connected:

Corollary 12. *Let $\mathcal{S}$ be a specification, $\mathcal{TP}$ a test purpose, $\mathcal{P}$ their product and $\mathcal{T}_{\mathcal{P}}$ its associated timed transition system. The reachable part of $\mathcal{T}_{\mathcal{P}}$ is strongly-connected.*

Proof. Let $((l^1, l^2), v)$ be a reachable configuration of $\mathcal{T}_{\mathcal{P}}$. There exists a finite partial execution of $\mathcal{S}$ starting in (l^1, v) whose trace contains ζ, and thus by Corollary 11 there exists a finite partial execution starting in $((l^1, l^2), v)$ whose trace contains ζ. Hence this transition leads to the configuration $s_0 = ((l_0^{\mathcal{S}}, l_0^{\mathcal{TP}}), \mathbf{0})$. It comes that there exists a finite partial execution from $((l^1, l^2), v)$ to s_0. Hence any reachable configuration of $\mathcal{T}_{\mathcal{P}}$ is reachable from $((l^1, l^2), v)$. It can then be concluded that the reachable part of $\mathcal{T}_{\mathcal{P}}$ is strongly-connected. □

Example 5. Fig. 9 represents the product of the conveyor-belt specification of Fig. 5 and the test purpose of Fig. 8. All nodes are named by the first letters of the corresponding states of the specification (first) and of the test purpose (second). The only accepting location is (D_2, A).

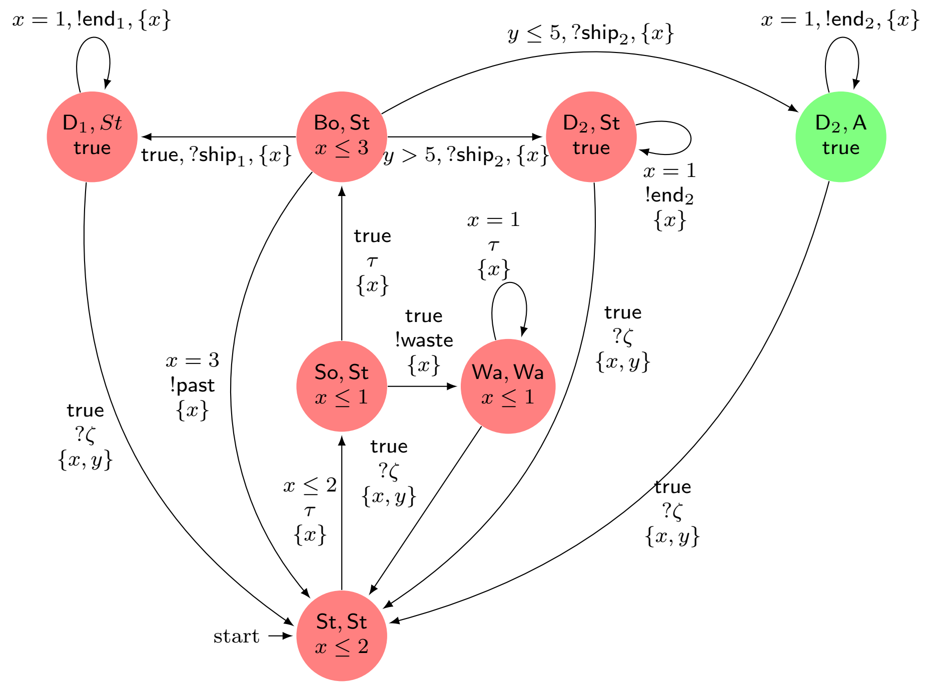


Fig. 9 Product of the conveyor belt specification and the presented test purpose.

We make one final hypothesis: we consider only pairs of specifications $\mathcal{S}$ and test purposes $\mathcal{TP}$ whose product $\mathcal{P}$ can be exactly determinized. This restriction is necessary for technical reasons: if the determinization is only approximated (*i.e.*, the deterministic product $\mathcal{DP}_a$ is not trace-equivalent to the product $\mathcal{P}$), we cannot ensure in general that restarts are still reachable and thus we lose strong connectivity. As noted in Remark 2 our approach can still be of interest in this general case, but can benefit of some refinements. One possible method to achieve determinization is to use the determinization game presented in [BSJK15]. This determinization is known to be exact for several classes of timed automata, such as strongly-non-Zeno automata, integer-reset automata, or event-recording automata. Furthermore this game always constructs a deterministic timed automaton, and when the set of traces is approximated, it preserves tioco-conformance in the following sense: if an implementation $\mathcal{I}$ conforms to its specification $\mathcal{S}$ (equivalently it conforms to $\mathcal{P}$), then it also conforms to the approximate determinization $\mathcal{DP}_a$; in other words, non-conformances with respect to the approximation $\mathcal{DP}_a$ are still non-conformances with respect to the specification $\mathcal{S}$.

Given the product $\mathcal{P} = \mathcal{S} \times \mathcal{TP}$, let $\mathcal{DP}$ be its exact determinization. Then $\mathsf{Traces}(\mathcal{DP}) = \mathsf{Traces}(\mathcal{P})$, and the reachability of restart transitions ζ is preserved. We also realize the closure by Σ_τ to obtain an observable model[6]. Moreover the traces leading to $\mathsf{Accept}^{\mathcal{DP}}$ and $\mathsf{Accept}^{\mathcal{P}}$ are the same.

Example 6. The automaton in Fig. 10 is a deterministic approximation of the product presented in Fig. 9. The internal transitions have collapsed, leading to an augmented Start location.

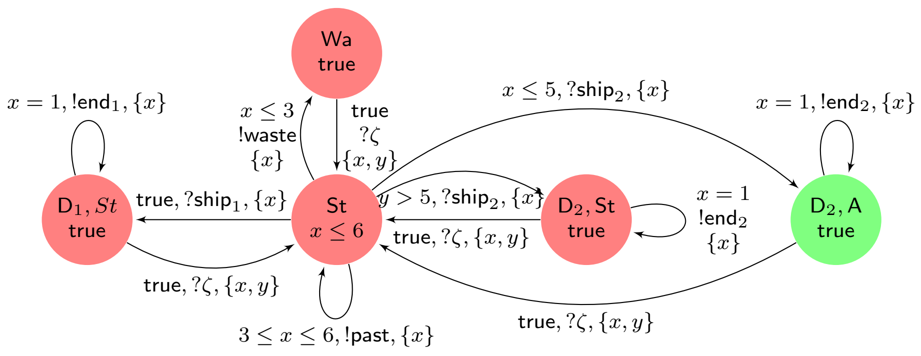

Wa
true
$x = 1, !\mathsf{end}_1, \{x\}$
$x \leq 3$
!waste
$\{x\}$
true
$?\zeta$
$\{x, y\}$
$x \leq 5, ?\mathsf{ship}_2, \{x\}$
$x = 1, !\mathsf{end}_2, \{x\}$
D_1, St
true
$\mathsf{true}, ?\mathsf{ship}_1, \{x\}$
St
$x \leq 6$
$y > 5, ?\mathsf{ship}_2, \{x\}$
$\mathsf{D}_2, \mathsf{St}$
true
$x = 1$
$!\mathsf{end}_2$
$\{x\}$
D_2, A
true
$\mathsf{true}, ?\zeta, \{x, y\}$
$\mathsf{true}, ?\zeta, \{x, y\}$
$3 \leq x \leq 6, !\mathsf{past}, \{x\}$
$\mathsf{true}, ?\zeta, \{x, y\}$

Fig. 10 A deterministic approximation of the product.

3.3 Taking failure into account

At this stage of the process, we dispose of a deterministic and fully-observable TAIO $\mathcal{DP}$ having exactly the same traces as the original specification $\mathcal{S}$, but equipped with a subset of locations labelled as $\mathsf{Accept}^{\mathcal{DP}}$ which identifies those

[6] The determinization procedure in [BSJK15] also realizes the closure.

traces of runs of $\mathcal{S}$ accepted by the test purpose. From this TAIO, we aim to build a tester, that can monitor the implementation, feeding it with inputs and selecting verdicts from the returned outputs.

We also need to explicitly model tioco-faulty behaviours (unspecified outputs and delays after a specified trace), we simply complete $\mathcal{DP}$ with respect to its output alphabet, by adding an explicit Fail location. This completed TAIO is called the *objective-centered tester.*

Definition 13. From the deterministic TAIO $\mathcal{DP} = (L^{\mathcal{DP}}, l_0^{\mathcal{DP}}, \Sigma_? \uplus \Sigma_!, X_p^{\mathcal{DP}}, I^{\mathcal{DP}}, E^{\mathcal{DP}})$, we construct its *objective-centered tester* $\mathcal{OT} = (L^{\mathcal{DP}} \cup \{\mathsf{Fail}\}, l_0^{\mathcal{DP}}, \Sigma_? \uplus \Sigma_!, X_p^{\mathcal{DP}}, I^{\mathcal{OT}}, E^{\mathcal{OT}})$ where $I^{\mathcal{OT}}(l) = \mathsf{true}$. The set of transitions $E^{\mathcal{OT}}$ is defined from $E^{\mathcal{DP}}$ by:

$$E^{\mathcal{OT}} = E^{\mathcal{DP}} \cup \Big(\bigcup_{\substack{l \in L^{\mathcal{DP}} \\ a \in \Sigma_!^{\mathcal{DP}}}} \{(l, g, a, \emptyset, \mathsf{Fail}) \mid g \in \overline{G}_{a,l}\}\Big) \cup \big\{(\mathsf{Fail}, \mathsf{true}, a, \emptyset, \mathsf{Fail}) \mid a \in \Sigma^{\mathcal{DP}}\big\}$$

where for each a and l, $\overline{G}_{a,l}$ is a set of guards complementing the set of all valuations v for which an a-transition is available from (l, v) (notice that $\overline{G}_{a,l}$ generally is non-convex, so that it cannot be represented by a single guard). Verdicts are defined on the configurations of $\mathcal{OT}$ as follows:

– $\boldsymbol{Pass} = \bigcup_{l \in \mathsf{Accept}^{\mathcal{DP}}} (\{l\} \times I^{\mathcal{DP}}(l))$;
– $\boldsymbol{Fail} = \{\mathsf{Fail}\} \times \mathbb{R}_{\geq 0} \cup \bigcup_{l \in L^{\mathcal{DP}}} \Big(\{l\} \times \big(\mathbb{R}_{\geq 0}^{X_p} \setminus I^{\mathcal{DP}}(l)\big)\Big)$.

Pass is the set of configurations reached by behaviours accepted by the test purpose, and where a *Pass* verdict will be given. It is defined as the set of configurations where the location l belongs to $\mathsf{Accept}^{\mathcal{DP}}$ and the clock valuation satisfies its invariant. ***Fail*** is the set of configurations supposedly corresponding to non-conformant behaviours, and where a ***Fail*** verdict will be given. It is defined as the set of configurations where either the reached location is the new location Fail, or the invariant is violated. This should be understood with the definition of transitions leading to this location Fail: the set of transitions $E^{\mathcal{DP}}$ is completed so that for each location $l \in L^{\mathcal{DP}}$, for each output $a \in \Sigma_!^{\mathcal{DP}}$, transitions lead to Fail in $E^{\mathcal{OT}}$ if no guard of a transition in $E^{\mathcal{DP}}$ carrying a is satisfied. Moreover loops are added on any action in Fail, making them trap locations. Usually ***Inconclusive*** is the set of configurations in which we cannot conclude to non-conformance, and we cannot satisfy the test purpose anymore because $\mathsf{Accept}^{\mathcal{DP}}$ is unreachable. With our hypothesis on ζ-transitions, strong connectivity is preserved and such configurations do not exist. We will thus enforce the apparition of ***Pass*** or ***Fail***. In the following, we present some of the useful properties of objective oriented testers, notably what properties are kept from $\mathcal{DP}$.

We first prove that strong connectivity is preserved, except for ***Fail*** states:

Proposition 14. *Let $\mathcal{OT}$ be an objective-centered tester and $\mathcal{T}^{\mathcal{OT}}$ its associated timed transition system. Then $\mathsf{Reach}(\mathcal{T}^{\mathcal{OT}}) \setminus$ **Fail** is strongly-connected.*

Proof. First, the fact that ζ-transitions are always reachable is preserved in $\mathcal{OT}$. Indeed, the same result holds from any state in $\mathcal{S}$, and $\mathsf{Traces}(\mathcal{S}) = \mathsf{Traces}(\mathcal{S} \times \mathcal{TP})$. The result then follows by exact determinizability of the product $\mathcal{S} \times \mathcal{TP}$ and then preserved in $\mathcal{OT}$ except in ***Fail*** states. □

Proposition 14 is the reason our method assumes exact determinizability: we cannot ensure in general strong connectivity: if determinization is approximated, some traces leading to a restart might be lost, and the proposition would not hold anymore.

A run ρ of an objective-centered tester $\mathcal{OT}$ is said *conformant* if it does not reach $\boldsymbol{Fail}$. We write $\mathsf{Run}_{\mathsf{conf}}(\mathcal{OT})$ for the set of conformant runs of $\mathcal{OT}$, and $\mathsf{Sig}_{\mathsf{conf}}(\mathcal{OT})$ (resp. $\mathsf{Traces}_{\mathsf{conf}}(\mathcal{OT})$) the corresponding signatures (resp. traces). We write $\mathsf{Run}_{\mathsf{fail}}(\mathcal{OT}) = \mathsf{Run}(\mathcal{OT}) \setminus \mathsf{Run}_{\mathsf{conf}}(\mathcal{OT})$ and similarly for the signatures and traces.

It is easy to prove the following:

Proposition 15. *Let $\mathcal{DP}$ be the exact determinization of the product $\mathcal{P}$ between a specification and a test purpose, and $\mathcal{OT}$ be its associated objective-centered tester. Then*

$$\mathsf{Traces}(\mathcal{DP}) = \mathsf{Traces}_{\mathsf{conf}}(\mathcal{OT}).$$

Proof. An execution is in $\mathsf{Run}_{\mathsf{conf}}(\mathcal{OT})$ if it avoids $\boldsymbol{Fail}$. This amounts to avoiding location Fail and respecting the invariants of $\mathcal{DP}$. By construction of $\mathcal{OT}$, this corresponds exactly to the runs of $\mathcal{DP}$. □

By construction we can easily see that $\mathsf{Traces}_{\mathsf{conf}}(\mathcal{OT})$ and $\mathsf{Traces}(\mathsf{Run}_{\mathsf{fail}}(\mathcal{OT}))$ are disjoint.

It remains to say that $\mathcal{OT}$ is repeatedly-observable, except for state $\boldsymbol{Fail}$.

Lemma 16. *For a repeatedly-observable specification $\mathcal{S}$, $\mathsf{Reach}(\mathcal{OT}) \setminus \boldsymbol{Fail}$ is repeatedly-observable.*

Proof. We know that $\mathsf{Traces}(\mathcal{S}) = \mathsf{Traces}(\mathcal{P})$, hence $\mathsf{out}(\mathcal{P} \text{ after } \sigma) = \mathsf{out}(\mathcal{S} \text{ after } \sigma)$ for all $\sigma \in \mathsf{Traces}(\mathcal{P})$. As $\mathsf{Traces}(\mathcal{DP}) = \mathsf{Traces}(\mathcal{P})$ by assumption, we also know that for all $\sigma \in \mathsf{Traces}(\mathcal{DP})$, $\mathsf{out}(\mathcal{P} \text{ after } \sigma) \subseteq \mathsf{out}(\mathcal{DP} \text{ after } \sigma)$. It comes

$$\forall \sigma \in \mathsf{Traces}(\mathcal{OT}),\ \mathsf{out}(\mathcal{S} \text{ after } \sigma) \subseteq \mathsf{out}(\mathcal{OT} \text{ after } \sigma)$$

as $\mathcal{OT}$ only adds traces to $\mathcal{DP}$. Hence for all $s \in \mathsf{Reach}(S^{\mathcal{OT}}) \setminus \boldsymbol{Fail}$, there exists $\mu \in \mathsf{Sig}(\mathcal{OT})$ s.t. $s \xrightarrow{\mu}$ and $\mathsf{trace}(\mu) \notin \mathbb{R}_{\geq 0}$. Indeed, there exists $\sigma \in \mathsf{Traces}_{\mathsf{conf}}(\mathcal{OT})$ such that $s = \mathcal{OT} \text{ after } \sigma$ (as $\mathcal{OT}$ is deterministic outside of $\boldsymbol{Fail}$) and for $s' \in \mathcal{S} \text{ after } \sigma$, there exists μ' such that $s' \xrightarrow{\mu'}$ and $\mathsf{trace}(\mu') \notin \mathbb{R}_{\geq 0}$. It suffices to take $\mu \in \mathsf{pSig}(\mathcal{OT})$ such that $\mathsf{trace}(\mu) = \mathsf{trace}(\mu')$, and by the previous trace-inclusion property, such a trace exists. □

In this section we explained the construction of the objective-centered tester $\mathcal{OT}$ from the specification and a test purpose and some of its properties. It represents the most general behaviours of testers that detect both non-conformance to the specification $\mathcal{S}$ and acceptance by the test purpose $\mathcal{TP}$. In the next section, we go further and explain how to tackle controllability problems by interpreting $\mathcal{OT}$ as a game whose winning strategies will be test cases that try to avoid control losses.

4 Interpreting objectives into games

In this section, we first briefly introduce timed games in Subsection 4.1. Next, we interpret objective-centered testers as reachability timed games between the tester and the implementation. In Subsection 4.2 we propose to define test cases as strategies of this game. In particular, we will compute rank-lowering strategies as strategies that try to minimize control losses, *i.e.*, situations where the tester needs help from the implementation to reach ***Pass***. We introduce a fairness criterion in which these optimal strategies are winning in Subsection 4.3, and discuss the properties of the resulting test cases (*i.e.*, game structure and built strategy) in Subsection 4.4.

4.1 Timed games

We introduce timed game automata [AMPS98b], which we later use to formalize the interactions between the tester and the system under test.

Definition 17. A *timed game automaton* (TGA) is a timed automaton $\mathcal{A}_g = (L, l_0, \Sigma_c \uplus \Sigma_u, X, I, E)$ where $\Sigma = \Sigma_c \uplus \Sigma_u$ is partitioned into actions that are controllable by the player (Σ_c), and those that are not (Σ_u).

Intuitively, the tester is the player for which we will try to find strategies against its opponent, the system under test, with the interpretation of Σ_c as inputs $\Sigma_?$ controlled by the tester, and Σ_u as outputs $\Sigma_!$ controlled by the system but only observed by the tester. We define the set of transitions with uncontrollable actions $E_u = \{e \in E \mid \mathsf{act}(e) \in \Sigma_u\}$ and similarly for controllable actions $E_c = \{e \in E \mid \mathsf{act}(e) \in \Sigma_c\}$. Winning conditions will be discussed later. All the notions of runs, signatures and traces defined previously for TAs naturally extend to TGAs.

We now define strategies on a game as functions associating to each run a delay and an action to take after this delay.

Definition 18. Let $\mathcal{A}_g = (L, l_0, \Sigma_c \uplus \Sigma_u, X, I, E)$ be a TGA. A *strategy* for the player is a partial function $f\colon \mathsf{Run}(\mathcal{A}_g) \to \mathbb{R}_{\geq 0} \times (\Sigma_c \cup \{\bot\}) \setminus \{(0, \bot)\}$ such that for any finite run ρ, letting $f(\rho) = (t, a)$, $t \in \mathsf{elapse}(\mathsf{last}(\rho))$ is a possible delay from $\mathsf{last}(\rho)$, and there is an a-transition available from the resulting configuration (unless $a = \bot$).

The special action $\bot$ is used to model situations where the player only wants to spend some delay: when he or she waits for the opponent to take a move, or when he or she has already won. We do not allow strategies to output $(0, \bot)$, as it would amount to instantly recompute a strategy, and would only loop until time delays. Strategies give rise to *outcomes i.e.*, a subset of runs of $\mathcal{A}_g$ that can be observed while following the strategy f:

Definition 19. Let $\mathcal{A}_g = (L, l_0, \Sigma_c \uplus \Sigma_u, X, I, E)$ be a TGA and f be a strategy over $\mathcal{A}_g$. The *set of outcomes* of f from a configuration s, denoted by $\mathsf{Outcome}(f, s)$ (we might omit to mention s when it is clear from the context), is the smallest subset of partial runs starting from s containing the empty partial run (whose last configuration is s), and s.t. for any $\rho \in \mathsf{Outcome}(f)$, letting $f(\rho) = (t, a)$ and $\mathsf{last}(\rho) = (l, v)$, we have

- $\rho \cdot ((l,v), t', (l, v+t')) \cdot ((l, v+t'), e, (l', v')) \in \mathsf{Outcome}(f)$ for any $0 \leq t' \leq t$ and $e \in E_u$ such that $((l, v+t'), e, (l', v')) \in \mathsf{pRun}(\mathcal{A}_g)$;
- and
  - either $a = \bot$, and $\rho \cdot ((l,v), t, (l, v+t)) \in \mathsf{Outcome}(f)$;
  - or $a \in \Sigma_c$, and $\rho \cdot ((l,v), t, (l,v+t)) \cdot ((l, v+t), e, (l', v')) \in \mathsf{Outcome}(f)$ with $\mathsf{act}(e) = a$.

An infinite partial run is in $\mathsf{Outcome}(f)$ if infinitely many of its finite prefixes are.

Intuitively, the first point corresponds to the opponent (the system) taking an uncontrollable transition while the player (the tester) waits. It also includes a race for actions between the player and the opponent, when $t' = t$. The second point adds the runs corresponding to the actions selected by the player, when the opponent does not interfere. Said differently, when the player decides to propose (t, a), the opponent may either preempt it by playing an uncontrollable action before t (first point), or let the player play (second point).

In this paper, we will be interested in reachability winning conditions, *i.e.*, a strategy is winning if it leads to ***Pass*** or ***Fail***. Formally, a strategy f is winning in a configuration s if and only if for any infinite $\rho \in \mathsf{Outcome}(f, s)$ there is a prefix of ρ ending in ***Pass*** or ***Fail***. We denote $\mathsf{Win}(\mathcal{A}_g, s)$ the set of such executions, and we say that the configuration s is winning for f.

The set of winning configurations can be computed in exponential time [AMPS98b, CDF$^+$05], by a backward iterative computation of the controllable predecessors, starting from the target configurations and using the region abstraction.

In the following, we augment the backward computation by adding a special step in the computation that encodes control losses, akin to the proposition in [DLLN08a]. This allows us to compute strategies outside of winning states. Still, we differ from [DLLN08a]'s approach in that we will only rely on that step when the classical backward computation reaches a fixpoint. This allows to minimize the dependency on the system, while resorting to it when necessary.

4.2 Rank-lowering strategies

We want to enforce conclusive verdicts when running test cases, *i.e.*, either the implementation does not conform to its specification (***Fail*** verdict) or the awaited behaviour appears (***Pass*** verdict). We thus say that an execution ρ is winning for the tester if it reaches a ***Fail*** or ***Pass*** configuration and denote by $\mathsf{Win}(\mathcal{A}_g)$ the set of such executions. Observe however that ***Fail*** is entirely controlled by the implementation, which may or may not reveal non-conformances. If it does, we will capture it. We will thus only target ***Pass***. In the following, we consider the previously defined DTAIO $\mathcal{OT} = (L^{\mathcal{OT}}, l_0^{\mathcal{OT}}, \Sigma_? \uplus \Sigma_!, X_p, I^{\mathcal{OT}}, E^{\mathcal{OT}})$ where $I^{\mathcal{OT}}(l) = \mathsf{true}$ and $\mathsf{Fail} \in L^{\mathcal{OT}}$ as a Timed Game Automaton, and rename it $\mathcal{A}_g^{\mathcal{OT}}$. Since we consider that the player is placed on the tester side, its controllable actions are the inputs $\Sigma_c = \Sigma_?$ and its uncontrollable actions are the outputs $\Sigma_u = \Sigma_!$, and $E^{\mathcal{OT}}$ is partitioned accordingly into $E_c^{\mathcal{OT}}$ and $E_u^{\mathcal{OT}}$. We then formalize a *test case* as a pair $(\mathcal{A}_g^{\mathcal{OT}}, f)$ where f is a strategy on $\mathcal{A}_g^{\mathcal{OT}}$.

We restrict our discussion to TGAs where ***Pass*** configurations are reachable (when seen as plain TAs). Indeed, when this is not the case (we will discuss the fact that the proposed method can detect this), trying to construct a strategy seeking

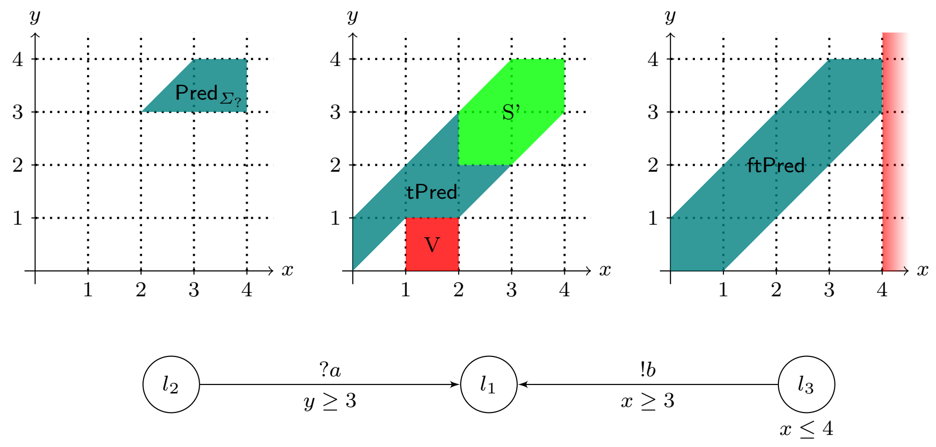


Fig. 11 The different types of predecessors considered.

a ***Pass*** verdict is hopeless. This is a natural restriction, as it only rules out test purposes that are unsatisfiable by the specification.

As discussed above, a strategy for the tester should target the ***Pass*** set in a partially controllable way, while monitoring ***Fail***. This partial controllability is discussed using *control losses*. They intuitively correspond to uncontrollable loops that could block the game progression. We define a hierarchy of configurations, depending on their "distance" to ***Pass***, based on the number of control losses one has to suffer to reach ***Pass***, and the number of transitions toward the next control loss. This uses a backward algorithm, for which we define the predecessors of a set of configurations.

Given a set of configurations $S' \subseteq S$ of $\mathcal{A}_g^{\mathcal{OT}}$, letting $\overline{V}$ denote the complement of V, we define three kinds of predecessors of S':

– discrete predecessors by a sub-alphabet $\Sigma' \subseteq \Sigma$

$$\mathsf{Pred}_{\Sigma'}(S') = \{(l,v) \mid \exists a \in \Sigma',\ \exists (l,a,g,X',l') \in E^{\mathcal{OT}}, v \models g \wedge (l', v_{[X' \leftarrow 0]}) \in S'\}$$

– timed predecessors, while avoiding a set V of configurations:

$$\mathsf{tPred}(S',V) = \{(l,v) \mid \exists t \in \mathbb{R}_{\geq 0},\ (l,v+t) \in S' \wedge \forall\, 0 \leq t' \leq t.\ (l,v+t') \notin V\}$$

We furthermore write $\mathsf{tPred}(S') = \mathsf{tPred}(S', \emptyset)$;

– final timed predecessors are defined for convenience (see below):

$$\mathsf{ftPred}(S') = \mathsf{tPred}(\textbf{\textit{Fail}}, \mathsf{Pred}_{\Sigma_u}(\overline{S'})) \cup \overline{\mathsf{tPred}(\mathsf{Pred}_{\Sigma}(\overline{S'}))}$$

These predecessors are illustrated in Fig. 11 and explained below. The first set $\mathsf{Pred}_{\Sigma'}(S')$ is the set of states from which a transition carrying an action in Σ' leads to S'; $\mathsf{tPred}(S',V)$ is the set of states from which delaying leads to S' without entering V; the final timed predecessors correspond to situations where the system is *cornered* into reaching S' unless it chooses non-conformance or to remain indefinitely idle: on the left side of the union, $\mathsf{tPred}(\textbf{\textit{Fail}}, \mathsf{Pred}_{\Sigma_u}(\overline{S'}))$ is the set of states from which the system under test only has the choice between taking an uncontrollable transition to S' (as no uncontrollable transition to $\overline{S'}$ will be available) or reach ***Fail***. On the right side are the states from which there are no transitions to anywhere but S' in the future, hence the system will end up playing a transition to S'. Intuitively, this is because the system will not remain infinitely idle when it can act (this will be formally enforced by a fairness hypothesis). Such situations are not considered as control losses, as the system can only take a beneficial transition for the tester (either by going to S' or to ***Fail***). Observe that tPred and ftPred need not return convex sets, but are efficiently computable using Pred and simple set constructions [CDF+05].

Now, using these notions of predecessors, a hierarchy of configurations based on the distance to ***Pass*** is defined.

Definition 20. The sequence $(W_i^j)_{j,i}$ of sets of configurations is defined as:

- $W_0^0 = \boldsymbol{Pass}$;
- $W_{i+1}^j = \pi_d(W_i^j)$ with $\pi_d(S') = \mathsf{tPred}\left(S' \cup \mathsf{Pred}_{\Sigma_c}(S'), \mathsf{Pred}_{\Sigma_u}(\overline{S'})\right) \cup \mathsf{ftPred}(S')$;
- $W_0^{j+1} = \pi_{cl}(W_\infty^j)$ with $\pi_{cl}(S') = \mathsf{tPred}(S' \cup \mathsf{Pred}_{\Sigma}(S'))$ and W_∞^j the limit[7] of the sequence $(W_i^j)_i$.

In this hierarchy, j corresponds to the minimum number of *control losses* the tester has to go through (in the worst case) in order to reach ***Pass***, and i corresponds to the minimal number of steps before the next control loss (or to ***Pass***).

The definition of W_0^{j+1} corresponds to the intuition of a control loss: for this step in the construction of the outcomes, we consider that the tester can get help from the implementation so as to enter W_∞^j. Said differently, the cooperation of the implementation is needed to construct a strategy beyond the fixpoint W_∞^j. This concerns several causes: first, no controllable actions lead to W_∞^j, an uncontrollable exists but another uncontrollable action reaches a configuration with a larger rank and can be taken either before, at the same time or after the one reaching W_∞^j[8]. Second, a controllable action leads to W_∞^j within some delay t but some uncontrollable actions with a shorter delay t' may preempt it and lead to a configuration with larger rank.

On the other hand, in the construction of W_{i+1}^j, the tester keeps full control to enter W_i^j, as there are no possible armful uncontrollable actions.

The computation of $(W_i^j)_i$ corresponds to finding the controllable zones, the least fix points W_∞^j. Indeed W_∞^0 are exactly the winning configurations for the reachability to ***Pass***, and W_∞^j for the reachability to W_0^j. From W_∞^j a control-loss step is taken to W_0^{j+1}. Finally, the hierarchy iterates this process and terminates on its least fix point, effectively finding all configurations coreachable from ***Pass*** and the minimal number of control losses for each of them.

Notice that the sequence (W_i^j) is a non-decreasing sequence of regions, and hence can be computed in time exponential in the size of $X^{\mathcal{OT}}$ and linear in the size of $L^{\mathcal{OT}}$.

We then have the following property saying that the computation of (W_i^j) covers all reachable states except Fail:

[7] The sequence $(W_i^j)_i$ is non-decreasing, and can be computed in terms of clock regions; hence the limit exists and is reached in a finite number of iterations [CDF+05].

[8] Notice that if no harmful uncontrollable action exists even after the one leading to W_∞^j, then $\mathsf{ftPred}(S')$ captures the behaviour and there is no control loss

Proposition 21. *There exists $i, j \in \mathbb{N}$ such that $(\mathsf{Reach}(\mathcal{A}_g^{\mathcal{OT}}) \setminus \textbf{\textit{Fail}}) \subseteq W_i^j$.*

Proof. Let $s \in \mathsf{Reach}(\mathcal{A}_g^{\mathcal{OT}}) \setminus \textbf{\textit{Fail}}$ be a reachable configuration. Since i) $\textbf{\textit{Pass}}$ is reachable from $s_0^{\mathcal{T}}$ (by hypothesis); ii) there is a path from s back to the initial configuration (Proposition 14), then $\textbf{\textit{Pass}}$ is reachable from s. Moreover, there is such a path with length bounded by the number of regions times the number of locations.

For each $s \in \mathsf{Reach}(\mathcal{A}_g^{\mathcal{OT}}) \setminus \textbf{\textit{Fail}}$, we fix a finite path to $\textbf{\textit{Pass}}$, and reason by induction on the length n of this path in order to prove that $s \in W_0^n$:

– case $n = 0$: in this case $s \in \textbf{\textit{Pass}} = W_0^0$;
– inductive case: we assume that the result holds for n, and take s with a path to $\textbf{\textit{Pass}}$ of length $n + 1$. Then $s \xrightarrow{\gamma} s'$ for some γ with $\mathsf{act}(\gamma) \in \mathit{\Gamma}$, and there is a path from s' to $\textbf{\textit{Pass}}$ of length at most n, so that $s' \in W_0^n$. Hence in the worst case $s \in W_0^{n+1}$.

This proves our result. ◻

As explained above, this property is based on the assumption that the $\textbf{\textit{Pass}}$ verdict is reachable. Nevertheless, if this were not the case, it would be detected during the hierarchy construction, which would then converge to a fixpoint not including $s_0^{\mathcal{OT}}$. As all the configurations in which we want to define a strategy are covered by the hierarchy, we can use it to define a preference relation (*i.e.*, a total preorder) that will guide the tester to better places in terms of control and distance to $\textbf{\textit{Pass}}$.

Definition 22. Let $s, s' \in \mathsf{Reach}(\mathcal{A}_g^{\mathcal{OT}}) \setminus \textbf{\textit{Fail}}$. The *rank* $r(s)$ of s is the pair

$$(j_s = \underset{j \in \mathbb{N}}{\arg\min}(s \in W_\infty^j),\ i_s = \underset{i \in \mathbb{N}}{\arg\min}(s \in W_i^{j_s}))$$

Moreover we define the *preference relation* $\preceq$ in $\mathsf{Reach}(\mathcal{A}_g^{\mathcal{OT}}) \setminus \textbf{\textit{Fail}}$ as follows:

$$s \preceq s' \text{ when } r(s) \leq_{\mathbb{N}^2} r(s') \text{ where } \leq_{\mathbb{N}^2} \text{ is the lexical order on } \mathbb{N}^2.$$

First by Prop. 21, r is a well defined function on $\mathsf{Reach}(\mathcal{A}_g^{\mathcal{OT}}) \setminus \textbf{\textit{Fail}}$. When $r(s) = (j_s, i_s)$, it holds $s \in W_{i_s}^{j_s}$ and j_s is the minimal number of control losses before reaching an accepting state, and i_s is the minimal number of steps in the strategy before the next control loss. Intuitively, if $s \preceq s'$ it is easier, *i.e.*, more controllable and shorter in terms of the number of transitions, to drive trajectories to $\textbf{\textit{Pass}}$ from s than from s'.

Proposition 23. *$\preceq$ is a total preorder on $\mathsf{Reach}(\mathcal{A}_g^{\mathcal{OT}}) \setminus \textbf{\textit{Fail}}$.*

Proof. $\preceq$ inherits transitivity and reflexivity from $\leq_{\mathbb{N}^2}$ by r. It is not antisymmetric since r is not injective: several configurations may have the same rank, for example, all configurations of $\textbf{\textit{Pass}}$ have rank $(0, 0)$. It is total since r is defined and $\leq_{\mathbb{N}^2}$ is total. ◻

Example 7. In Fig. 12, the (W_i^j) are represented on the game constructed from the $\mathcal{OT}$ corresponding to the deterministic TA presented in Fig 10. For clarity, the Fail location and the transitions leading to it are not represented. In this example, $W_0^0 = (\mathsf{D}_2, \mathsf{A}) \times \mathbb{R}_{\geq 0}$, $W_0^1 = \mathsf{St} \times (y \leq 5)$ and $W_1^1 = \mathit{St} \times (5 < x \leq 6) \cup \{\mathsf{Wa}, (\mathsf{D}_1, \mathsf{St}), (\mathsf{D}_2, \mathsf{St})\} \times \mathbb{R}_{\geq 0}$. The $\textbf{\textit{Fail}}$ verdict corresponds to $\mathsf{Fail} \cup (\mathsf{St} \times (x \geq 6))$.

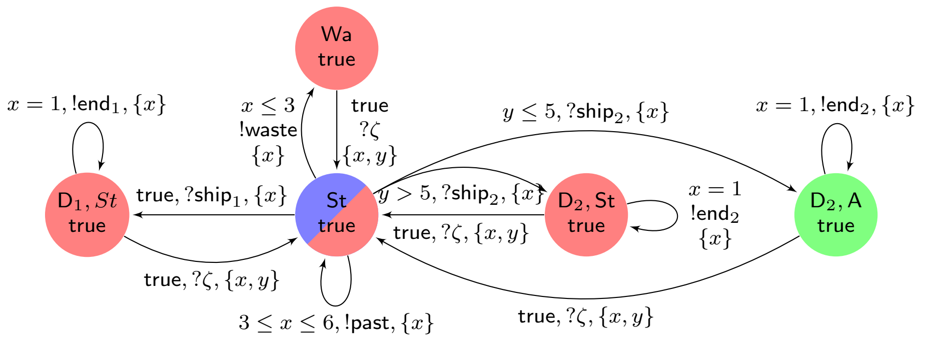


Fig. 12 The ranks of $\mathcal{A}_g$.

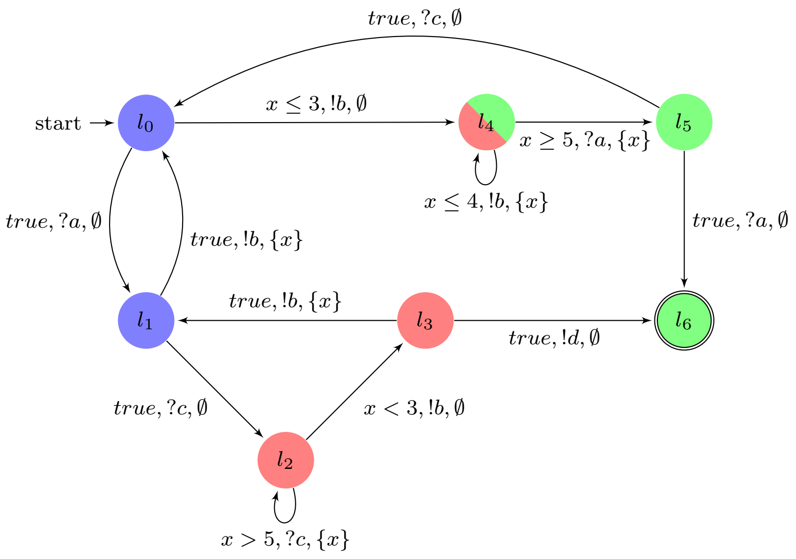


Fig. 13 The ranks in a complex game.

Example 8. This notion of rank can be applied to many difficult games, *i.e.*, games where you can not always win, even out of the testing framework. For example, the game in Fig. 13 has the following ranks, with all configurations in l_6 considered as accepting: $(0, 0)$ in $l_6 \times \mathbb{R}_{\geq 0}$, $(0, 1)$ in $l_5 \times \mathbb{R}_{\geq 0}$, $(0, 2)$ in $s_4 \times (x > 4)$, $(1, 0)$ in $l_3 \times \mathbb{R}_{\geq 0} \cup l_4 \times (x \leq 4)$, $(1, 1)$ in $l_2 \times (x < 3)$, $(1, 2)$ in $l_2 \times (x \geq 3)$, $(2, 0)$ in $l_0 \times (x \leq 3) \cup s_1 \times \mathbb{R}_{\geq 0}$ and $(2, 1)$ in $l_0 \times (x > 3)$. Notice how control losses correspond to the existence of uncontrollable loops stopping the tester from forcing the progress as *e.g.*, the self loop in l_4 and the loop l_1, l_2, l_3.

We use the preference relation $\preceq$ to define a strategy trying to decrease the rank during the execution, *i.e.*, decrease in each state both the distance to and the number of control losses. For any $s \in S$, we write $r^-(s)$ for the largest rank such that $r^-(s) <_{\mathbb{N}^2} r(s)$, and $W^-(s)$ for the associated set in $(W_i^j)_{j,i}$. We (partially) order pairs $(t, a) \in \mathbb{R}_{\geq 0} \times \Sigma$ according to t. We call our strategies *rank-lowering* because from any configuration s they target $W^-(s)$.

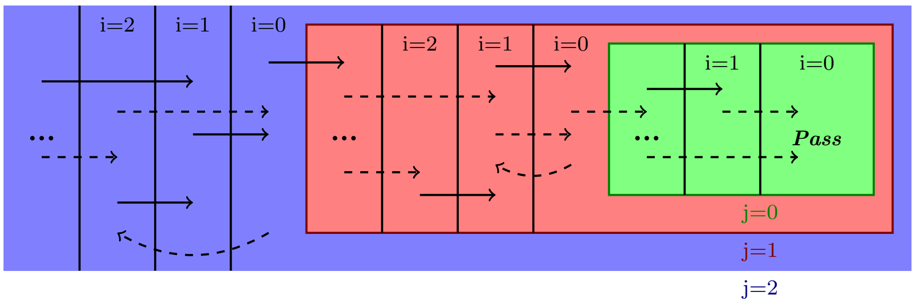


Fig. 14 A representation of the outcomes of a rank-lowering strategy.

Definition 24. A strategy f for the tester is *rank-lowering* if, for any finite run ρ with $\mathsf{last}(\rho) = s = (l, v)$, it selects the smallest delay t satisfying one of the following constraints:

- if $s \in \mathsf{tPred}(\mathsf{Pred}_{\Sigma_c}(W^-(s)))$, then $f(\rho) = (t, \mathsf{act}(e))$ for some $e \in E_c^{\mathcal{OT}}$ and $s \xrightarrow{t} s' \xrightarrow{e} s''$ with $s'' \in W^-(s)$, $s' \notin W^-$ and t is minimal in the following sense: if $s \xrightarrow{t'} \xrightarrow{e'} s'''$ with $s''' \in W^-(s)$, $e' \in E_c^{\mathcal{OT}}$ and $t' \leq t$, then $v + t$ and $v + t'$ belong to the same region;
- if $s \in \mathsf{tPred}(\mathsf{Pred}_{\Sigma_u}(W^-(s)))$, then $f(\rho) = (t, \bot)$ with t such that $s \xrightarrow{t} s' \notin \mathsf{Pred}_{\Sigma_u}(W^-(s))$, $\exists t' < t$ $s \xrightarrow{t'} s'' \in \mathsf{Pred}_{\Sigma_u}(W^-(s))$ and t is minimal if such a t exists. Else t is maximal in the same sense as above (maximal delay successor region);
- if $s \in \mathsf{tPred}(W^-(s))$, then $f(\rho) = (t, \bot)$ such that $s \xrightarrow{t} s'$ with $s' \in W^-(s)$, and t is minimal in the same sense as above;
- otherwise $f(\rho) = (t, \bot)$ where t is maximal in the same sense as above (maximal delay-successor region).

The first three cases follow the construction of the (W_i^j) and propose the shortest behaviour leading to W^-. The fourth case corresponds either to a configuration of ***Pass***, where W^- is undefined, or to a ftPred leading to ***Fail***, cases in which the tester has won. Interestingly, at least rank-lowering strategies exist as W_i^j covers all reachable configurations (Proposition 21). The constraints of rank-lowering strategies allow the creation of *memoryless* strategies, *i.e.*, strategies that only rely on properties of the current configuration without using the history of the trace used to reach it. Indeed, they only use local information, aggregated during the incremental construction of the (W_i^j) sets. Hence, a straightforward implementation only using these sets is memoryless.

Example 9. In Fig. 14 the structure of the (W_i^j) is represented together with the transitions that can be fired while playing a rank-lowering strategy. The dashed arrows depict uncontrollable transitions while the plain arrows depict controllable ones, both represented with their source and target W_i^j.

Example 10. An example of a rank-lowering strategy on the automaton of Fig. 12 is: in $(\mathsf{D}_2, \mathsf{A})$, play $\bot$ (as W_0^0 has been reached); in St, play $(0, ?\mathsf{ship}_2)$ from W_0^1, otherwise play $(1, \bot)$. In any other state, play $(0, ?\zeta)$. The omission of Fail in

Fig. 12 does not impact the strategy representation: ***Fail*** is a winning set of configurations that is hit if conformance is violated, but can not be targeted by the strategies.

Remark 2. It is worth noting that even in a more general setup where the models are not equipped with ζ-transitions, as in [BJSK12], rank-lowering strategies may still be useful: as they are defined on the co-reachable set of ***Pass***, they can still constitute test cases, and the configurations where they are not defined are exactly the configurations corresponding to a ***Fail*** verdict or to an ***Inconclusive*** verdict, *i.e.*, no conclusions can be made since an accepting configuration cannot be reached.

Yet, rank-lowering strategies would gain to be refined in this case in order to avoid ***Inconclusive*** verdicts as much as possible. Indeed, as defined in this paper, a rank-lowering strategy would choose a path that is shorter but more prone to inconclusiveness over a longer but safer path. This can be handled by adding a new layer to the ranks encoding the distance to an inconclusive verdict (*i.e.*, the minimum number of uncontrollable actions required to reach one) over the path to ***Pass***. Strategies are defined in a similar fashion, but in this general case we can not ensure that they win, as they could sometimes reach ***Inconclusive***. This approach has been developed in [Hen21].

The rank-lowering strategies allow to play beyond the scope of winning states by relying on the implementation help, which is formalized as control losses, while minimizing this reliance.

4.3 Making rank-lowering strategies win

A rank-lowering strategy is generally not a winning strategy: it relies on the implementation fairly exploring its different possibilities and not repeatedly avoiding an enabled transition. In this section, we introduce a notion of fairness and prove that the rank-lowering strategies are winning under this fairness assumption.

The following lemma ensures that we cannot end in a *livelock* situation where no transitions can be taken, forcing the system to delay indefinitely. It will be used with the support of fairness, and will be the key to ensuring victory on fair executions.

Lemma 25. *If $\mathcal{OT}$ is repeatedly-observable, then for any run $\rho = ((s_i, \gamma_i, s_{i+1}))_{i\in\mathbb{N}} \in \mathsf{Run}(\mathcal{A}_g^{\mathcal{OT}})$ ending with an infinite sequence of delays, if ρ does no reach **Fail**, there is an infinite number of states in ρ where some transition e is enabled, formally,*[9]

$$\rho \notin \mathsf{Run}_{\mathsf{fail}}(\mathcal{A}_g^{\mathcal{OT}}) \Rightarrow \exists e \in E^{\mathcal{OT}},\ \overset{\infty}{\exists}\ i \in \mathbb{N},\ e \in \mathsf{enab}(s_i).$$

Proof. We show this lemma by contradiction. Assume that for some $\rho \in \mathsf{Run}(\mathcal{A}_g^{\mathcal{OT}})$, we have

$$\rho \notin \mathsf{Run}_{\mathsf{fail}}(\mathcal{A}_g^{\mathcal{OT}}) \wedge \forall e \in E^{\mathcal{OT}},\ \overset{\infty}{\forall}\ i \in \mathbb{N},\ e \notin \mathsf{enab}(s_i).$$

Let $\rho_{\max}$ be the shortest prefix such that no transition is enabled after this prefix along ρ (it exists because $E^{\mathcal{OT}}$ is finite and there is only a finite number of these

[9] In this expression, $\overset{\infty}{\exists}\ i \in \mathbb{N},\ \phi(i)$ means that $\phi(i)$ is true for infinitely many integers. In the same way, $\overset{\infty}{\forall}\ i \in \mathbb{N}\ \phi(i)$ means that $\phi(i)$ is true for all but finitely many integers.

prefixes per element of $E^{\mathcal{OT}}$). Consider any prefix ρ' of ρ strictly containing $\rho_{\max}$; there is no partial signature μ such that $\mathsf{last}(\rho') \xrightarrow{\mu}$ and $\mathsf{trace}(\mu) \notin \mathbb{R}_{\geq 0}$, as there is no time successor of $\mathsf{last}(\rho')$ with an enabled transition. This contradicts the repeated-observability of $\mathcal{OT}$ out of Fail (as $\mathcal{A}_g$ and $\mathcal{OT}$ are the same automaton). □

In order to introduce our notion of fairness, we define three notions of the infinite support of a run. The first one characterizes the set of regions encountered infinitely often, while the other two distinguish the regions from which a discrete action was taken infinitely often from the regions left by elapsing time infinitely often. This distinction will help us precisely define the behaviour expected from the implementation in each case.

Definition 26. Let ρ be an infinite run, its *infinite regions support* $\mathsf{Inf}(\rho)$ is the set of regions appearing infinitely often in ρ:

$$\begin{aligned}\mathsf{Inf}((s_i,\gamma_i,s_{i+1})_{i\in\mathbb{N}}) = \{ reg \in \mathcal{R} \mid \overset{\infty}{\exists}\, i \in \mathbb{N},\ s_i \in reg \,\vee \\ (\gamma_i \in \mathbb{R}_{\geq 0} \wedge \exists s_i' \in reg,\ \exists t_i < \gamma_i,\ s_i \xrightarrow{t_i} s_i')\}\end{aligned}$$

Its *infinite transitions support* $\mathsf{Inf}_{E^{\mathcal{OT}}}(\rho)$ is the set of pairs (reg, e) of $\mathcal{R} \times E^{\mathcal{OT}}$ such that the transition e is taken infinitely often from the region *reg*:

$$\mathsf{Inf}_{E^{\mathcal{OT}}}((s_i,\gamma_i,s_{i+1})_{i\in\mathbb{N}}) = \{(reg, e) \in \mathsf{Inf}(\rho) \times E^{\mathcal{OT}} \mid \overset{\infty}{\exists}\, i \in \mathbb{N}, s_i \in reg \wedge \gamma_i = e\}$$

and its *infinite waiting support* is the set of regions left infinitely often by delaying along ρ.

$$\begin{aligned}\mathsf{Inf}_t((s_i,\gamma_i,s_{i+1})_{i\in\mathbb{N}}) = \{ reg \in \mathsf{Inf}(\rho) \mid \overset{\infty}{\exists}\, i \in \mathbb{N}, \gamma_i \in \mathbb{R}_{\geq 0} \,\wedge \\ \exists\, t_i < \gamma_i,\ s_i \xrightarrow{t_i} s_i' \in reg \wedge s_{i+1} \in reg' \in \mathsf{SucTemp}(reg)\}\end{aligned}$$

Using these notions of infinite support, we will define the fairness of a run. The intuition behind it is as follows: if a behaviour is *implemented*, then it will be ultimately displayed if repeatedly fireable. Formally, we only dispose of a model of the specification, not the black box implementation. Hence we have to state our fairness on the game model, which makes it stronger than this intuition.

We define the set of controllable enabled transitions for a region *reg*: $\mathsf{enab}_c(reg) = \mathsf{enab}(reg) \cap E_c^{\mathcal{OT}}$, and similarly the set of conformant uncontrollable enabled transitions (*i.e.*, that do not lead to ***Fail***):

$$\mathsf{enab}_{u,\mathsf{conf}}(reg) = \mathsf{enab}(reg) \cap E_u^{\mathcal{OT}} \setminus \{e \in E^{\mathcal{OT}} \mid \mathsf{next}(e, reg) \subseteq \boldsymbol{Fail}\}.$$

Definition 27. An infinite run ρ in the TGA $\mathcal{A}_g^{\mathcal{OT}}$ is said to be *fair* when:[10]

[10] See definitions of enab, next, SucTemp in subsection 2.2.

$$\forall reg \in \mathsf{Inf}(\rho), \begin{cases} \{reg\} \times \mathsf{enab}_{u,\mathsf{conf}}(reg) \subseteq \mathsf{Inf}_{E^{\mathcal{OT}}}(\rho) \wedge & (1) \\ \begin{bmatrix} \vee \ \{e \in E_c^{\mathcal{OT}} \mid (reg, e) \in \mathsf{Inf}_{E^{\mathcal{OT}}}(\rho)\} \neq \emptyset \\ \vee \ reg \in \mathsf{Inf}_t(\rho) \\ \mathsf{enab}_c(reg) = \emptyset \wedge \mathsf{SucTemp}(reg) \subseteq \boldsymbol{Fail} \end{bmatrix} & (2) \end{cases}$$

We denote by $\mathsf{Fair}(\mathcal{A}_g^{\mathcal{OT}})$ the set of fair runs of $\mathcal{A}_g^{\mathcal{OT}}$.

Fair runs model restrictions on the system runs corresponding to strategies of the system. The first part (1) of the definition ensures that for each region visited infinitely often, each tioco-conformant enabled action of the implementation will be played infinitely often from that region. Intuitively, it means that the implementation explores all of its options infinitely often. The second part (2) ensures that the implementation will infinitely often let the tester play in this region, by saying that, either a discrete transition controllable by the tester has been played infinitely often from this region (first disjunctive case), or that it has been possible to leave this region by waiting infinitely often (second disjunctive case). This is limited by the third disjunctive case, when no controllable transitions are enabled and the strict timed successors of the region (maximal delay successor region), if any, are included in ***Fail***. [11] This definition of fairness is related to the "strong fairness" notion used in model checking.

As we will in the following property, restricting to fair runs is sufficient to ensure a winning execution when the tester uses a rank-lowering strategy. Intuitively, combined with Lemma 25 and the repeated-observability assumption, it assures that the system will keep providing outputs until a verdict is reached. The proof reasons on regions, as fairness is defined on them, exploiting the fact that the sets W_i^j are coarser than regions: indeed a region is included in any W_i^j it intersects. In terms of testing, it means that if the implementation is fair, a tester following a rank-lowering strategy will reach a conclusive verdict (***Pass*** or ***Fail***).

Theorem 28. *Rank-lowering strategies are winning on* $\mathsf{Fair}(\mathcal{A}_g^{\mathcal{OT}})$ *(i.e., all fair outcomes are winning).*

We prove this claim by contradiction. The general idea of the proof is to suppose that there exists an infinite run in $\mathsf{Fair}(\mathcal{A}_g^{\mathcal{OT}})$ that does not reach W_0^0, and consider a zone with minimal rank that is visited infinitely often during this run. We then discuss according to the structure of the rank-lowering strategies, and, using the fairness assumption, contradict minimality.

Proof. Let $\mathcal{T} = (S^{\mathcal{OT}}, s_0^{\mathcal{OT}}, \Gamma^{\mathcal{OT}}, \rightarrow_{\mathcal{T}})$ be the timed transition system associated with $\mathcal{A}_g^{\mathcal{OT}}$, and let f be a rank-lowering strategy. We want to prove that $\mathsf{Outcome}(f) \cap \mathsf{Fair}(\mathcal{A}_g^{\mathcal{OT}}) \subseteq \mathsf{Win}(\mathcal{A}_g^{\mathcal{OT}})$. We proceed by contradiction and suppose that there exists an infinite run $\rho \in \mathsf{Outcome}(f) \cap \mathsf{Fair}(\mathcal{A}_g^{\mathcal{OT}})$ such that $\rho \notin \mathsf{Win}(\mathcal{A}_g^{\mathcal{OT}})$. We consider the set of prefixes of ρ ending in a "decision point" *i.e.*, where the tester plays a pair delay-action, and the regions where those prefixes end. Since there are infinitely many such prefixes but finitely many regions,

[11] To be perfectly precise, the fairness should depend on the strategy played by the tester to only restrict when the strategy tries to play a controllable action and not when it exists. We hide this dependency for the sake of simplicity but do not use the extra strength of the fairness in our results.

some region must appear infinitely many times, thus $\mathsf{Inf}(\rho) \neq \emptyset$. We denote by $r_{\min}$ the minimal rank obtained in these regions, and pick a region *reg* for which $r(\mathit{reg}) = r_{\min}$. By definition of *reg*, there are infinitely many prefixes ν of ρ ending in *reg* such that the strategy proposes the same pair $f(\mathsf{last}(\nu)) = (t^f_\nu, a^f_\nu)$.

Since Definition 24 distinguishes four possible ways to propose an action for a rank-lowering strategy, at least one of them appears infinitely often. We examine these four cases separately in our reasoning and each time construct a zone of smaller rank that has to be in $\mathsf{Inf}(\rho)$, thus contradicting the minimality hypothesis of $r_{\min}$:

- case $a^f_\nu \in \Sigma_c$, *i.e.*, $\mathsf{last}(\nu) \in \mathsf{tPred}(\mathsf{Pred}_{\Sigma_c}(W^-(\mathsf{last}(\nu))))$: in this case, there exists $e \in E^{\mathcal{OT}}_c$ such that $\mathsf{act}(e) = a^f_\nu$ and $\mathsf{last}(\nu) \xrightarrow{t^f_\nu} s'_\nu \xrightarrow{e} s'' \in W^-(\mathsf{last}(\nu))$. Because of the minimality constraint on t^f_ν there exists a unique region *reg'* such that for all ν, it holds $s'_\nu \in \mathit{reg}'$.
  - If $\mathsf{last}(\nu) \in \mathit{reg}'$, then $\mathsf{last}(\nu) \in \mathsf{Pred}_{\{a^f_\nu\}}(W^-(last(\nu)))$, and by the minimality constraint on t^f_ν, no rank-lowering strategies can delay out of *reg'*; hence $\mathit{reg}' \notin \mathsf{Inf}_t(\rho)$. Furthermore $e \in \mathsf{enab}_c(\mathit{reg})$. It follows from fairness that there exists a controllable transition e' such that $(\mathit{reg}, e') \in \mathsf{Inf}_{E^{\mathcal{OT}}}(\rho)$. As this can only be a transition labeled by a^f_ν by definition of $\mathsf{Outcome}(f)$ and $\mathcal{A}^{\mathcal{OT}}_g$ is deterministic, $e' = e$ and $\mathsf{next}(e, \mathit{reg}) \in \mathsf{Inf}(\rho) \cap W^-(\mathsf{last}(\nu)) = \mathsf{Inf}(\rho) \cap W^-(\mathit{reg})$. f will take a decision once arriving in this region after each of the infinitely many transitions, contradicting the minimality of $r_{\min}$;
  - otherwise, when $\mathsf{last}(\nu) \notin \mathit{reg}'$ we have $\mathit{reg}' \in \mathsf{SucTemp}(reg) \setminus \boldsymbol{\mathit{Fail}}$. Hence by construction of a rank-lowering strategy and definition of $\mathsf{Outcome}(f)$, there is no controllable transition taken in *reg* along ρ. Hence by fairness $\mathit{reg} \in \mathsf{Inf}_t(\rho)$ and the first delay-successor region of *reg* is in $\mathsf{Inf}(\rho)$. By induction on the number of regions between *reg* and *reg'*, using the case $\mathit{reg} = \mathit{reg}'$ as the base case and the previous remark as induction step, we know that $\mathit{reg}' \in \mathsf{Inf}(\rho)$. We can then conclude using the previous discussion;
- case $a^f_\nu = \bot$ and $\mathsf{last}(\nu) \in \mathsf{tPred}(\mathsf{Pred}_{\Sigma_u}(W^-(\mathsf{last}(\nu))))$: in this case there exists $e \in E^{\mathcal{OT}}_u$ such that $\exists t' \leq t^f_\nu$, $\mathsf{last}(\nu) \xrightarrow{t'}\xrightarrow{e} s'' \in W^-(\mathsf{last}(\nu))$. Moreover, because of the minimality constraint on t^f_ν there exists a unique region *reg'* such that for all ν, $\mathsf{last}(\nu) \xrightarrow{t^f_\nu} \in \mathit{reg}'$. As in the previous case,
  - if $\mathit{reg} = \mathit{reg}'$ then by fairness $(\mathit{reg}, e) \in \mathsf{Inf}_{E^{\mathcal{OT}}}(\rho)$ as e is uncontrollable, and thus $\mathsf{next}(e, \mathit{reg}) \in \mathsf{Inf}(\rho) \cap W^-(\mathit{reg})$, and f will take a new decision once arriving in this region after each of the infinitely many transitions, contradicting the minimality of $r_{\min}$.
  - otherwise $\mathit{reg}' \in \mathsf{SucTemp}(\mathit{reg}) \setminus \boldsymbol{\mathit{Fail}}$, and since a rank-lowering strategy will never play a discrete transition in *reg*, by minimality constraint on t^f_ν we have the same induction as in the previous case, and we can conclude using the case $\mathit{reg} = \mathit{reg}'$;
- case $a^f_\nu = \bot$ and $\mathsf{last}(\nu) \in \mathsf{tPred}(W^-(\mathsf{last}(\nu)))$: in this situation the simple induction on time successor regions is enough to conclude that we reach infinitely often a region in $W^-(\mathsf{last}(\nu))$ in which (by minimality of the delays proposed by a rank-lowering strategy) the strategy will take a decision infinitely often. This contradicts once again the minimality of $r_{\min}$;
- otherwise, either $W^-(\mathsf{last}(\nu))$ is undefined and hence $r_{\min} = (0, 0)$, contradicting the fact that ρ is not winning, or there is no partial execution from $\mathsf{last}(\nu)$ to

$W^-(\mathsf{last}(\nu))$ meaning that the system can only delay. By construction of the (W_i^j), this corresponds to a timed predecessor of ***Fail***. Hence the system is cornered and can only delay to ***Fail***, thus ρ should be winning. □

At this stage, we thus have identified a test-case generation method, starting from the specification with restarts and the test purpose, and constructing a test case as a strategy on the game created from the objective-centered tester. This strategy is winning under the hypothesis of a fair implementation, meaning that its execution is ensured to deliver in finite time a conclusive verdict ***Pass*** or ***Fail***. The complexity of this method is exponential in the size of $\mathcal{DP}$. More precisely:

Proposition 29. *Given a deterministic product $\mathcal{DP}$, the objective-centered tester $\mathcal{OT}$ can be linearly computed from $\mathcal{DP}$, the construction of a strategy relies on the construction of the W_i^j, and is hence exponential in the size of $X^{\mathcal{DP}}$ and linear in the size of $L^{\mathcal{DP}}$.*

Observe that if $\mathcal{DP}$ is obtained from $\mathcal{P}$ by the game presented in [BSJK15], then $L^{\mathcal{DP}}$ is doubly-exponential in the size of $X^{\mathcal{S}} \uplus X^{\mathcal{TP}} \uplus X^{\mathcal{DP}}$ (notice that in the setting of [BSJK15], $X^{\mathcal{DP}}$ is a parameter of the algorithm, since determinization depends on fixed ressources, namely the number of clocks of the resulting TA and their maximal constants).

4.4 Properties of the test cases

Having constructed strategies for the tester, and identified a scope of implementation behaviours that allows these strategies to enforce a conclusive verdict, we now study the properties obtained by the test generation method presented above. In fact the first properties do not depend on the strategy and were already valid in the context of [BJSK12]. Their proofs are based on the exact correspondence between ***Fail*** in $\mathcal{OT}$ and the non-conformant traces of $\mathcal{S}$, and use the trace equivalence of the different models ($\mathcal{DP}$, $\mathcal{P}$ and $\mathcal{S}$) to conclude. As they exploit mainly the game structure $\mathcal{A}_g^{\mathcal{OT}}$, fairness is not used, and they do not rely on the strategy being played. They are thus valid for any strategy in $\mathcal{A}_g^{\mathcal{OT}}$, not only rank-lowering strategies. Only Proposition 34, the exhaustiveness property, uses rank-lowering strategies and requires fairness.

Recall that a *test case* is a pair $(\mathcal{A}_g^{\mathcal{OT}}, f)$ where $\mathcal{A}_g^{\mathcal{OT}}$ is the game corresponding to the objective-centered tester $\mathcal{OT}$, and f is a strategy on $\mathcal{A}_g^{\mathcal{OT}}$. We will precise when f is rank-lowering. We denote by $\mathcal{TC}(\mathcal{S}, \mathcal{TP})$ the set of possible test cases generated from a specification $\mathcal{S}$ and a test purpose $\mathcal{TP}$, and $\mathcal{TC}(\mathcal{S})$ the set of test cases for any test purpose. Recall that it is assumed that the test purposes associated with a specification are restricted to those leading to a determinizable product.

We first define *parallel runs* as the possible outcomes of a test case combined with an implementation, which thus model their parallel composition.

Definition 30. Given a test case $(\mathcal{A}_g^{\mathcal{OT}}, f)$ and an implementation $\mathcal{I}$, their *parallel runs* are the sequences $((s_i, s'_i), (\gamma_i, \gamma'_i), (s_{i+1}, s'_{i+1}))_i$ such that $(s_i, \gamma_i, s_{i+1})_i$ is an outcome of f in $\mathcal{A}_g^{\mathcal{OT}}$, $((s'_i, \gamma'_i, s'_{i+1}))_i$ is a run of $\mathcal{I}$, and for all i, either $\gamma_i = \gamma'_i$ if $\gamma_i \in \mathbb{R}_{\geq 0}$ or $\mathsf{act}(\gamma_i) = \mathsf{act}(\gamma'_i)$ otherwise. We write $\mathsf{ParRun}((\mathcal{A}_g^{\mathcal{OT}}, f), \mathcal{I})$ for the set of parallel runs of the test case $(\mathcal{A}_g^{\mathcal{OT}}, f)$ and of the implementation $\mathcal{I}$.

We say that an implementation $\mathcal{I}$ fails a test case $(\mathcal{A}_g^{\mathcal{OT}}, f)$, and write $\mathcal{I}$ fails $(\mathcal{A}_g^{\mathcal{OT}}, f)$, when there exists a run in $\mathsf{ParRun}((\mathcal{A}_g^{\mathcal{OT}}, f), \mathcal{I})$ that reaches $\boldsymbol{Fail}$. Our method is sound, that is, a conformant implementation cannot be detected as failing a test case.

Proposition 31. *The test-case generation method is* sound*: for any specification $\mathcal{S}$, it holds*

$$\forall \mathcal{I} \in \mathcal{I}(\mathcal{S}),\ \forall (\mathcal{A}_g^{\mathcal{OT}}, f) \in \mathcal{TC}(\mathcal{S}),\ (\mathcal{I}\ \mathit{fails}\ (\mathcal{A}_g^{\mathcal{OT}}, f) \Rightarrow \neg(\mathcal{I}\ \mathit{tioco}\ \mathcal{S})).$$

Proof. Let $\mathcal{S}$ be a specification, $\mathcal{I} \in \mathcal{I}(\mathcal{S})$ and $(\mathcal{A}_g^{\mathcal{OT}}, f) \in \mathcal{TC}(\mathcal{S})$. Suppose that $\mathcal{I}$ fails $(\mathcal{A}_g^{\mathcal{OT}}, f)$, we will prove that $\neg(\mathcal{I}\ \mathsf{tioco}\ \mathcal{S})$.

Since $\mathcal{I}$ fails $(\mathcal{A}_g^{\mathcal{OT}}, f)$, there is a finite run ρ of $\mathsf{ParRun}((\mathcal{A}_g^{\mathcal{OT}}, f), I)$ such that $last(\rho) \in \boldsymbol{Fail} \times S^{\mathcal{I}}$ and it is the first configuration of ρ in this set. Let $\sigma = \mathsf{trace}(\rho)$. By construction of $\boldsymbol{Fail}$, either $\sigma = \sigma' \cdot t$ (if the configuration of $\boldsymbol{Fail}$ reached corresponds to a faulty invariant) or $\sigma = \sigma' \cdot a$ with $a \in \Sigma_!$ (and Fail is reached). In both cases $\mathsf{out}(\mathcal{I}\ \mathsf{after}\ \sigma') \nsubseteq \mathsf{out}(\mathcal{DP}\ \mathsf{after}\ \sigma')$, and by definition $\neg(I\ \mathsf{tioco}\ \mathcal{DP})$.

As $\mathsf{Traces}(\mathcal{P}) = \mathsf{Traces}(\mathcal{DP})$ by exact-determinizability hypothesis, $\neg(I\ \mathsf{tioco}\ \mathcal{P})$. Finally, as $\mathsf{Traces}(\mathcal{P}) = \mathsf{Traces}(\mathcal{S})$, we have $\neg(I\ \mathsf{tioco}\ \mathcal{S})$, which concludes the proof. □

The next property says that, under the assumption that $\mathcal{DP}$ is an exact determinization of $\mathcal{P}$, for any generated test case and any implementation, if their parallel execution follows a non-conformant trace, it raises a fail verdict. This means that there is no approximation in the detection of non-conformances.

Proposition 32. *The test generation method is* strict*: given a specification $\mathcal{S}$,*

$$\forall \mathcal{I} \in \mathcal{I}(\mathcal{S}),\ \forall (\mathcal{A}_g^{\mathcal{OT}}, f) \in \mathcal{TC}(\mathcal{S}),\ \neg(\mathit{ParRun}((\mathcal{A}_g^{\mathcal{OT}}, f), \mathcal{I})\ \mathit{tioco}\ \mathcal{S}) \Rightarrow \mathcal{I}\ \mathit{fails}\ (\mathcal{A}_g^{\mathcal{OT}}, f)$$

Proof. Let $\mathcal{S}$ be a specification, $\mathcal{I} \in \mathcal{I}(\mathcal{S})$ and $(\mathcal{A}_g^{\mathcal{OT}}, f) \in \mathcal{TC}(\mathcal{S})$. Suppose that $\neg(\mathsf{ParRun}((\mathcal{A}_g^{\mathcal{OT}}, f), \mathcal{I})\ \mathsf{tioco}\ \mathcal{S})$. We want to show that $\mathcal{I}$ fails $(\mathcal{A}_g^{\mathcal{OT}}, f)$. By definition of $\neg(\mathsf{ParRun}((\mathcal{A}_g^{\mathcal{OT}}, f), \mathcal{I})\ \mathsf{tioco}\ \mathcal{S})$, there exist $\sigma \in \mathsf{Traces}(\mathcal{S})$ and $a \in \mathsf{out}(\mathsf{ParRun}((\mathcal{A}_g^{\mathcal{OT}}, f), \mathcal{I})\ \mathsf{after}\ \sigma) \setminus \mathsf{out}(\mathcal{S}\ \mathsf{after}\ \sigma)$ with

$$\mathsf{out}(\mathsf{ParRun}((\mathcal{A}_g^{\mathcal{OT}}, f), \mathcal{I})\ \mathsf{after}\ \sigma) = \{a \in \Sigma_! \cup \mathbb{R}_{\geq 0} \mid \exists \rho \in \mathsf{ParRun}((\mathcal{A}_g^{\mathcal{OT}}, f), \mathcal{I}),\ \mathsf{trace}(\rho) = \sigma \cdot a\}$$

the extension of outputs after a trace to parallel runs. Since $\mathcal{DP}$ is an exact determinization of $\mathcal{P}$ we have the following equalities: $\mathsf{Traces}(\mathcal{S}) = \mathsf{Traces}(\mathcal{P}) = \mathsf{Traces}(\mathcal{DP}) = \mathsf{Traces}_{\mathsf{conf}}(\mathcal{OT})$. Since $a \in \mathbb{R}_{\geq 0} \cup \Sigma_!$, $\sigma \cdot a \in \mathsf{Traces}(\mathcal{OT})$ as invariants have been removed, and the automaton has been completed on $\Sigma_!$ with transitions to Fail). Hence $\sigma \cdot a \in \mathsf{Traces}(\mathsf{Run}_{\mathsf{fail}}(\mathcal{OT}))$. Thus, for $\rho \in \mathsf{ParRun}((\mathcal{A}_g^{\mathcal{OT}}, f), \mathcal{I})$ such that $\mathsf{trace}(\rho) = \sigma \cdot a$, $\mathsf{last}(\rho) \in \boldsymbol{Fail}$ and $\mathcal{I}$ fails $(\mathcal{A}_g^{\mathcal{OT}}, f)$. □

This method also enjoys a precision property: traces leading the test case to $\boldsymbol{Pass}$ are exactly traces conforming to the specification and accepted by the test purpose. The proof uses only properties of the game, the exact encoding of the Accept states into the definition of $\boldsymbol{Pass}$ by propagation through the different test artifact, and once more is valid for any strategy used.

Proposition 33. *The test case generation method is* precise*: for any specification $\mathcal{S}$ and test purpose $\mathcal{TP}$ it can be stated that*

$$\forall(\mathcal{A}_g^{\mathcal{OT}}, f) \in \mathcal{TC}(\mathcal{S}, \mathcal{TP}), \forall\sigma \in \mathit{Traces}(\mathit{Outcome}(f)),$$
$$\mathcal{A}_g^{\mathcal{OT}} \ \mathit{after}\ \sigma \in \boldsymbol{Pass} \iff (\sigma \in \mathit{Traces}(\mathcal{S}) \wedge \mathcal{TP}\ \mathit{after}\ \sigma \cap \mathit{Accept}^{\mathcal{TP}} \neq \emptyset)$$

Proof. Let σ be in $\mathsf{Traces}(\mathsf{Outcome}(f))$. Then $\mathcal{A}_g^{\mathcal{OT}}$ after $\sigma \in \boldsymbol{Pass}$ if, and only if, the run ρ such that $\mathsf{trace}(\rho) = \sigma$ (which is unique by determinism of $\mathcal{A}_g^{\mathcal{OT}}$ outside $\boldsymbol{Fail}$) is such that $\mathsf{last}(\rho) \in \boldsymbol{Pass}$, *i.e.*, $\rho \in \mathsf{Run}(\mathcal{DP})$ and $\mathsf{last}(\rho) \in \mathsf{Accept}^{\mathcal{DP}}$. Hence $\mathcal{DP}$ after $\sigma \in \mathsf{Accept}^{\mathcal{DP}}$ and as the determinization is exact, $\sigma \in \mathsf{Traces}(\mathcal{P})$ and $\mathcal{P}$ after $\sigma \in \mathsf{Accept}^{\mathcal{P}}$, which gives by definition $\sigma \in \mathsf{Traces}(\mathcal{S}) \wedge \mathcal{TP}$ after $\sigma \cap \mathsf{Accept}^{\mathcal{TP}} \neq \emptyset$. □

Lastly, the test generation method when generating rank lowering strategies is exhaustive in the sense that for any non-conformance, there exists a test case using a rank lowering strategy that allows to detect it, under fairness assumption.

Proposition 34. *The test generation method based on rank-lowering strategies is* exhaustive*: for any exactly determinizable specification $\mathcal{S}$ and any implementation $\mathcal{I} \in \mathcal{I}(\mathcal{S})$ making fair runs*

$$\neg(\mathcal{I}\ \mathit{tioco}\ \ \mathcal{S}) \Rightarrow \exists(\mathcal{A}_g^{\mathcal{OT}}, f) \in \mathcal{TC}(\mathcal{S}), \mathcal{I}\ \mathit{fails}\ (\mathcal{A}_g^{\mathcal{OT}}, f)$$

f where f is a rank lowering strategy.

Proof. To demonstrate this property, a test purpose is tailored to detect a given non-conformance, by targeting a related conformant trace. Fairness ensures that the tester will eventually hit this trace and then the particular non-conformance.

Let $\mathcal{S}$ be a specification, and $\mathcal{I} \in \mathcal{I}(\mathcal{S})$ a non-conformant implementation. By definition of $\neg(\mathcal{I}\ \mathsf{tioco}\ \mathcal{S})$, there exists $\sigma \in \mathsf{Traces}(\mathcal{S})$ and $a \in \mathbb{R}_{\geq 0} \cup \Sigma_!$ such that $a \in \mathsf{out}(\mathcal{I}\ \mathsf{after}\ \sigma)$ and $a \notin \mathsf{out}(\mathcal{S}\ \mathsf{after}\ \sigma)$. As $\mathcal{S}$ is repeatedly-observable, there exists $t \in \mathbb{R}_{\geq 0}$ and $b \in \Sigma_{\mathsf{obs}}^{\mathcal{S}}$ such that $\sigma \cdot t \cdot b \in \mathsf{Traces}(\mathcal{S})$. Because $\mathcal{S}$ is also non-blocking, if a is a delay, we can take $b \in \Sigma_!^{\mathcal{S}}$. Indeed, otherwise there would be no trace controlled by the implementation for any finite time (say, for time a).

It is possible to build a test purpose $\mathcal{TP}$ that accepts exactly the trace $\sigma \cdot t \cdot b$. It suffices to direct every transition that is not part of this trace to a sink location. As $\sigma \cdot t \cdot b \in \mathsf{Traces}(\mathcal{S})$, it is also a trace of the product $\mathcal{P} = \mathcal{S} \times \mathcal{TP}$. As $\mathcal{S}$ is exactly determinizable and $\mathcal{TP}$ is deterministic, $\mathcal{P}$ is exactly determinizable by allowing enough resources (number of clocks and maximal constant) to $\mathcal{DP}$. We thus obtain $\mathsf{Traces}(\mathcal{DP}) = \mathsf{Traces}(\mathcal{P})$ and $\sigma \cdot t \cdot b \in \mathsf{Traces}(\mathcal{DP})$. Hence, the minimal elements of $\boldsymbol{Pass}$ are $\mathcal{OT}$ after $\sigma \cdot t \cdot b$.

From $\mathcal{OT}$ a test case $(\mathcal{A}_g^{\mathcal{OT}}, f)$ can be built, with f a rank-lowering strategy. By assumption, the implementation is playing fair runs, hence f is winning. So there exists $\rho \in \mathsf{Outcome}(f)$ such that $\mathsf{trace}(\rho) = \sigma \cdot t \cdot b$, and thus there exists $\rho' \in \mathsf{Outcome}(f)$ such that $\mathsf{trace}(\rho') = \sigma$. By assumption, $\sigma \cdot a \in \mathsf{Traces}(\mathcal{I})$, and depending on the nature of a:

– if $a \in \Sigma_!$ then $\sigma \cdot a \in \mathsf{Outcome}(s_0^{\mathcal{OT}}, f)$ as $\mathcal{A}_g^{\mathcal{OT}}$ is complete on $\Sigma_!$. Hence $\sigma \cdot a \in \mathsf{ParRun}((\mathcal{A}_g^{\mathcal{OT}}, f), \mathcal{I})$ and as $\sigma \cdot a \notin \mathsf{Traces}(\mathcal{S})$ and the determinization is exact, $\sigma \cdot a \notin \mathsf{Traces}_{\mathsf{conf}}(\mathcal{OT})$ and $\mathcal{A}_g^{\mathcal{OT}}$ after $\sigma \cdot a \in \boldsymbol{Fail}$. Hence $\mathcal{I}$ fails $(\mathcal{A}_g^{\mathcal{OT}}, f)$;

– if a is a delay, then $a > t$, and $b \in \Sigma_!$. As b is controlled by the implementation, and there is no invariant in $\mathcal{A}_g^{\mathcal{OT}}$, $\sigma \cdot a \in \mathsf{Outcome}(f)$. Hence $\sigma \cdot a \in \mathsf{ParRun}((\mathcal{A}_g^{\mathcal{OT}}, f), \mathcal{I})$ and as $\sigma \cdot a \notin \mathsf{Traces}(\mathcal{S})$ and the determinization is exact, $\sigma \cdot a \notin \mathsf{Traces}_{\mathsf{conf}}(\mathcal{OT})$ and $\mathcal{A}_g^{\mathcal{OT}}$ after $\sigma \cdot a \in \boldsymbol{Fail}$. Hence $\mathcal{I}$ fails $(\mathcal{A}_g^{\mathcal{OT}}, f)$. □

The properties obtained in this subsection correspond exactly to the properties already obtained in [BJSK12] in the determinizable case, our results only adapt the formalism to the game formulation, making it more precise, especially on the interaction between the test strategy and the implementation, as expected from the game formulation. The construction of rank-lowering strategies allows to keep the interesting properties of the test cases, while adding an equally important information about the control of the test cases: the rank-lowering strategies minimize the reliance on the implementation and are moreover winning in finite time (*i.e.*, an outcome of the strategy can not loop infinitely without reaching a winning configuration) against all fair implementations.

5 Implementing rank-lowering strategies

The construction of a practical algorithm to implement a rank-lowering strategy mainly boils down to the computation of a zone-based representation of the $(W_i^j)_{j,i}$ hierarchy.

Although the definition points toward a backward implementation, we follow [CDF+05] and prefer a forward approach. Using this, while the exact ranks are computed online, intermediary results provide an (over-)approximation of the ranks.

5.1 Algorithm

We extend the zone-based on-the-fly algorithm for constructing winning strategies in timed games proposed in [CDF+05]. Our algorithm, presented in Algorithm 1, uses at its core a rank-updating function detailed in Algorithm 2. In order to present these functions, we first define $(j,i)^-$ as the maximal rank strictly lower than (j,i), so as to avoid the distinction between $(j,i-1)$ and $(j-1,\infty)$ in the algorithms. We also define, for a zone Z, a location l, an action $\alpha \in \Sigma$, the set of states reached from (l,Z) after α and some delay:

$$\mathsf{Post}_\alpha((l,Z)) = \{s' \mid \exists s \in (l,Z).\ \exists t \in \mathbb{R}_{\geq 0}.\ s \xrightarrow{\alpha.t} s'\}.$$

As $\mathcal{A}_g^{\mathcal{OT}}$ is deterministic, this set of states corresponds to a zone in a unique location.

In the following, we heavily rely on the properties of π_d and π_{cl}, the two functions used to compute the sets $(W_i^j)_{j,i}$ (defined in Def. 20). We quickly formalize them bellow.

Proposition 35. *For π a function in $\{\pi_d, \pi_{cl}\}$ and S' a set of configurations, $\pi(S') \supseteq S'$ and π is increasing.*

Proof. For the first property, $\pi(S') \supseteq S'$ is implied by the inclusion of null delays in tPred. For the second property, notice that Pred_Σ and tPred are increasing, and for π_d that $\overline{S'}$ decreases when S' increases. □

These properties transfer to the computation of $(W_i^j)_{j,i}$ and are used to ground the algorithms. Although it was not formally stated before, it was already highlighted in Fig. 14.

Our algorithm is based on a data structure approximating the W_i^j. It takes the form of a dictionary $W(S)$ for each symbolic state S. It associates with each rank (j, i) a set of configurations representing a union of zones in S denoted $W(S)[j, i]$. In order to reduce the memory cost of $W(S)$, we chose to keep in memory only the ranks (j, i) that increase strictly the corresponding zone, *i.e.*, such that $W(S)[j, i] \supsetneq W(S)[j, i-1]$. The ranks that do not have their zone explicitly stored hence refer to the greatest zone of lower rank. If no such zone exists, then the zone is empty. During the execution of the algorithm, the dictionary W will store under-approximations of the W_i^j that will gradually converge to the target sets.

The computation of an RLS realized by Algorithm 1 is based on a set Visited of encountered symbolic states representing sets of configurations inside a given location, and a queue Waiting of transitions to be processed[12]. For each symbolic state, two informations are stored: a list Depend of incoming transitions, and a dictionary W of the sets of configurations associated with the different ranks in the state. The algorithm is based on a method updateRank that will be presented later, and that correctly updates the estimates $W(S)$ according to the available information. After an initialization taking care of the timed successors of the initial configuration, denoted by $\mathbf{0}^{\nearrow}$ (lines 1 to 8), the main loop starts in line 9 and iterates until the reachable part of the automaton is explored, unless the initial configuration is a final one. In this loop, at each iteration, a transition is taken out of Waiting and processed. There are two cases to consider, presented in Fig. 15, where the snake arrow represents the call to an edge, forward or backward. If $S' \notin$ Visited the transition was stored to explore and the first branch of the **if** executes (line 12). It records the new S' and analyses the local information as presented on the left side of Fig. 15. If new information is acquired, its backpropagation is planed (line 18). The second situation corresponds to a backpropagation triggered by a previous call to line 18. In this case we are mostly interested in updating the ranks in the source location of the transition, and backpropagating again if anything has been updated. This corresponds to the right side of Fig. 15. Observe that line 23 is there to account for an exploration that leads to an already explored state[13].

Algorithm 2 describes the updateRank function. Its purpose is, for a given state S, to update $W(S)[(j, i)]$ according to the values of W at its successor locations. The general structure of the algorithm is a **while**-loop on the ranks. In order to compute more efficiently, and to avoid the complex discussion about the unbounded range of i for a given j (and of j in general), it relies on the identification of *active* ranks through *active*(j, i), *i.e.*, ranks in which $W(S')[(j, i)]$ strictly increases in a state S' of interest (either S or a successor). We distinguish

[12] The specific order of computation of the transitions does not impact termination or correctness, but can be of great importance for efficiency.

[13] It corresponds notably to the case where this transition is not triggered by a backpropagation call.

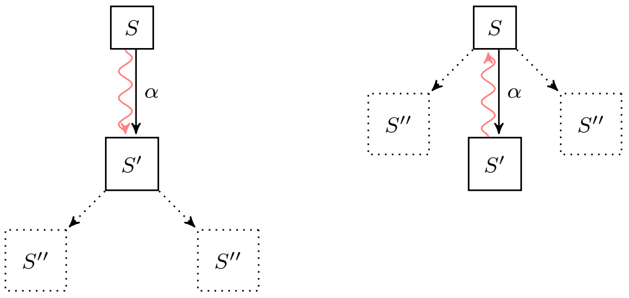


Fig. 15 The two kind of edges popped from Waiting.

```
Computation of an RLS
Input: an observable DTA $\mathcal{A}_g$
Output: A symbolic transition system augmented with information about ranks
(*Initialization*)
let $S_0 = (l_0, \mathbf{0}^{\nearrow})$ in
Visited $\leftarrow \{S_0\}$
Waiting $\leftarrow \{(S_0, \alpha, S') \mid S' = \mathsf{Post}_\alpha(S_0)\}$
Depend$(S_0) = \emptyset$
let $S = S_0 \cap \textbf{\textit{Pass}}$ in
if $S \neq \emptyset$ then
    $W(S_0)[(0,0)] \leftarrow S$
    updateRank$(S_0)$

(*Main*)
while $\mathit{Waiting} \neq \emptyset \wedge (s_0 \notin W(S_0)[(0,0)])$ do
    $e = (S, \alpha, S') \leftarrow \mathit{pop}(\mathsf{Waiting})$
    if $S' \notin \mathit{Visited}$ then
        Visited := Visited $\cup \{S'\}$
        Depend$(S') \leftarrow \{(S, \alpha, S')\}$
        let $S = S' \cap \textbf{\textit{Pass}}$ in
        if $S \neq \emptyset$ then
            $W(S')[(0,0)] \leftarrow Z$
            updateRank$(S')$
            Waiting := Waiting $\cup \{e\}$
        Waiting := Waiting $\cup \{(S', \alpha, S'') \mid S'' = \mathsf{Post}_\alpha(S')\}$
    else
        $\mathit{improved} =$ updateRank$(S)$
        if $\mathit{improved}$ then
            Waiting := Waiting $\cup$ Depend$(S)$
        Depend$(S')$ := Depend$(S') \cup \{e\}$
```

Algorithm 1: Computation of an RLS.

between local activity (in $active_{loc}$,) which has to be updated directly, and activity in a successor (in $active_{suc}$), which is of interest for the next rank. Furthermore, we identify the maximal active rank in $r_{\max}$[14]. This is realized in the initialization (until line 7). In the following, we use the notations $r.i$ and $r.j$ to denote the components of a rank. The rest of the algorithm is a loop in the ranks, which uses the jumpNext method to update the current rank according to *active*. The loop terminates when we went far enough to ensure that no more updates can be performed. Lines 9

[14] The maximal rank may evolve during the algorithm execution as ranks are added to or removed from *active*.

to 11 compute the current estimation of W_i^j according to $W(S')$. Then, if this set is strictly greater than the current approximate, it is updated (starting line 16). Otherwise, the data structure is cleared for this rank (lines 12 to 15). In both cases, $active_{loc}$ is updated.

```
function updateRank
Input: a symbolic state $S$
Output: a boolean denoting whether an improvement has been performed
1  let $active_{suc}((j,i))$ be true iff $W(S')[(j,i)] \neq W(S')[(j,i)^-]$ for some successor $S'$ of $S$;
2  let $active_{loc}((j,i))$ be true iff $W(S)[(j,i)] \neq W(S)[(j,i)^-]$;
3  $active((j,i)) = active_{suc}((j,i)) \vee active_{loc}((j,i))$;
4  $r_{\max} = max_{(j,i)}\, active((j,i))$;
5  $Temp_j = -1$ (* the last j for which a set has been improved *);
6  $Temp_W = W(S)[(0,0)]$ (* the current set *);
7  $j = 0$; $i = 1$;
8  while $(j,i) \leq (r_{\max}.j + 1, 0)$ do
       (* We try to improve the set of rank j,i *);
9      if $i = 0$ then
           (*then we add a control loss*);
10         $W \leftarrow \pi_{cl}(\cup_{S' \in \mathsf{Visited}} W(S')[(j,i)^-]) \cap S$;
       else
11         $W \leftarrow \pi_d(\cup_{S' \in \mathsf{Visited}} W(S')[(j,i)^-]) \cap S$;
12     if $W \subseteq Temp_W$ then
           (* this set does not need to be explicitly stored *);
13         erase $W(S)[(j,i)]$ (*handles the data structure*);
14         $active_{loc}(j,i) \leftarrow false$;
15         jumpNext;
       else
16         if $W(S)[(j,i)] \subsetneq W$ then
               (* the set was improved *);
17             $W(S)[(j,i)] \leftarrow W$;
18             $active_{loc}((j,i)) \leftarrow true$;
19             $Temp_j = j$;
20             $Temp_W = W$;
21         $i := i + 1$;
22 return $Temp_j \geq 0$
```

Algorithm 2: Function updateRank.

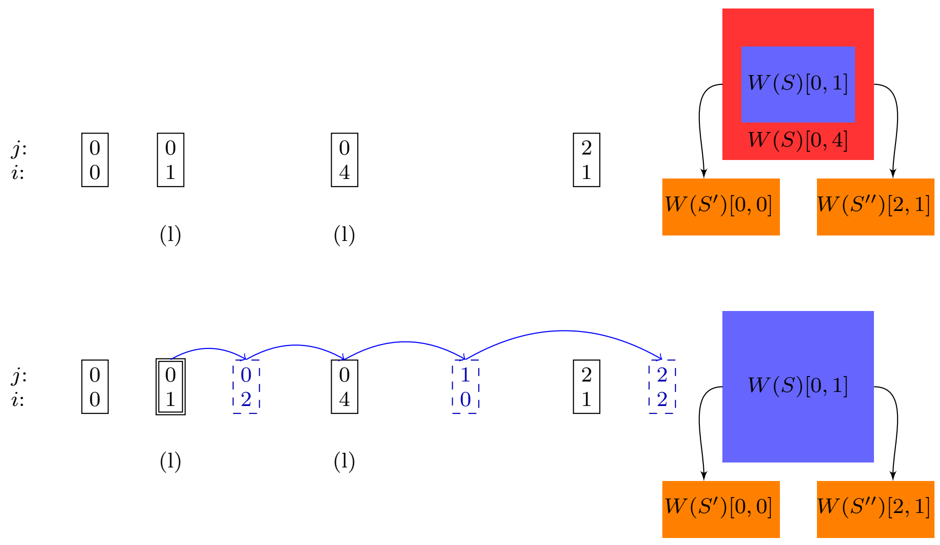
The jumpNext method is used in updateRank to correctly select the next rank of interest. Two main discussions are handled by Algorithm 3. First, local and distant ranks have a different behaviour. Indeed, if something is stored in $W(S)[(j,i)]$, it has to be updated and thus we target (j,i). Else, we go to $(j,i+1)$ as the construction of a set only depends on the sets of strictly lesser rank. This distinction is performed using the $active_{loc}$ subset of *active*. Second, when the next active rank is after a control loss (*i.e.*, if j increases), we have to know if that control loss can increase the set. If the control loss operator π_{cl} has not been applied to the current set yet, it could lead to an increase of the set and we go to $(j+1,0)$ to test it. Else we can go directly to the next active rank (with the discussion according to its location).

Example 11. Consider a symbolic state S such that $\emptyset \subsetneq W(S)[0,1] \subsetneq W(S)[0,4]$ and no other ranks are stored locally. We suppose that it has two successors S' and S'' that only store one rank each: $W(S')[0,0]$ and $W(S)[2,1]$. This situation and the result of a call to updateRank(S) are displayed in Fig. 16. The upper part

```
jumpNext;
(* we jump to the next possible improvement*);
1  let $n_r$ be the next active rank, including the current one.;
2  if $j = n_r.j$ then
3      if $active_{loc}(n_r)$ then
4          $i := n_r.i$
       else
5          $i := n_r.i + 1$;
   else
6      if $Temp_j = j$ then
           (* we might win something more after a control loss *);
7          $j := j + 1$;
8          $i := 0$;
       else
9          $j := n_r.j$;
10         if $active_{loc}(n_r)$ then
11             $i := n_r.i$
           else
12             $i := n_r.i + 1$;
```

Algorithm 3: jumpNext.

Fig. 16 An execution of updateRank(S). On the left side are the active ranks of S, and on the right side the sets stored by W in S and its successors. Both sides are represented before (above) and after (below) the algorithm execution.



of the figure represents the situation before the execution of the algorithm, with the active ranks being $(0,1)$, $(0,4)$–active locally–, $(0,0)$ and $(2,1)$–active in the successors. The lower part of the figure represents the situation after the algorithm execution when we suppose that $W(S)[0,1] = W(S)[0,4]$ with the update. The arrows and the dashed ranks represent the calls to jumpNext and the ranks that are tested in updateRank.

One benefit of using a forward algorithm is that it can provide us with sub-optimal strategies before completion (as soon as the initial state is in some $W(S_0)[(j,i)]$). The forward approach also avoids exploring unreachable states, and can be made more efficient by first pruning the sets of losing states (*e.g.*, inconclusive states).

Observe that such sub-optimal strategies obtained before completion of updateRank correspond to under-approximated $(W_i^j)_{j,i}$, *i.e.*, to over-approximated ranks; they do not prevent the implementation to send the test case to unexplored parts of the model (but can be improved by running updateRank further).

Finally, notice that updateRank is the core of our algorithm, and that it can actually be used both in forward and backward exploration algorithms.

5.2 Properties

We prove some interesting properties of the algorithm by constructing them from properties of updateRank and invariants of the main-algorithm's **while** loop.

The following lemma ensures that updateRank preserves the soundness of the approximation of the $(W_i^j)_{j,i}$. Informally, it states that from an under-approximation of $(W_i^j)_{j,i}$, updateRank cannot overestimate it (*i.e.*, overestimate the rank).

Lemma 36. *The following property is an invariant of updateRank:*

$$\forall\ i,j \in \mathbb{N},\ \forall S \in \textit{Visited},\ W(S)[(j,i)] \subseteq W_i^j.$$

Proof. For two integers i_m and j_m, we write $P(j_m, i_m)$ for the property:

$$\forall j \le j_m,\ i \le i_m,\ \forall S \in \mathsf{Visited}.\ W(S)[(j,i)] \subseteq W_i^j.$$

We show that property $P(j_m, i_m)$ is preserved by induction on (j_m, i_m):

- for $(j,i) = (0,0)$, updateRank does not modify W, so $P(0,0)$ remains is preserved;
- fix $j_m, i_m \in \mathbb{N}$ and suppose that $P(j_m, i_m)$ holds true. For a given S, either $W(S)[(j_m, i_m+1)]$ is not updated, and the property is trivially preserved , or
$$W(S)[(j_m, i_m+1)] = \pi_d\Big(\bigcup_{S' \in \mathsf{Visited}} W(S')[(j_m, i_m)]\Big) \cap S.$$
By hypothesis, $\bigcup_{S' \in \mathsf{Visited}} W(S')[(j_m, i_m)] \subseteq W_{i_m}^{j_m}$. Furthermore, π_d is an increasing function. Hence $W(S)[(j_m, i_m+1)] \subseteq \pi_d(W_{i_m}^{j_m}) \cap S = W_{i_m+1}^{j_m} \cap S$. It comes that $P(j_m, i_m+1)$ is satisfied after the call to updateRank;
- fix $j_m \in \mathbb{N}$. We denote by i_{j_m} the minimal i_m such that $(\forall i'_m > i_m,\ W_{i_m}^{j_m} = W_{i'_m}^{j_m})$ holds. Suppose $P(j_m, i_{j_m})$. For a given S, if $W(S)[(j_m+1, 0)]$ is not updated then the property is trivially preserved. Otherwise, with the same arguments as in the previous case,
$$W(S)[(j_m+1, 0)] = \pi_{cl}\Big(\bigcup_{S' \in \mathsf{Visited}} W(S')[(j_m, i_{j_m})]\Big) \cap S \subseteq W_0^{j_m+1}.$$
It comes that after updateRank, $P(j_m+1, 0)$ still holds. □

The following lemma states that $W(S)$ is increasing as a function of the rank, when it corresponds to a coherent approximation.

Lemma 37. *For $S \in \mathsf{Visited}$, $(j,i) > (j',i')$, when $\mathsf{updateRank}$ is not processing a rank between these two, we have that $W(S)[(j,i)] \supseteq W(S)[(j',i')]$.*

Proof. The proof can be done by a direct induction using the fact that either $W(S)[(j,i)]$ is not explicitly stored, which means that $W(S)[(j,i)] = W(S)[(j,i)^-]$ or $W(S)[(j,i)] = \pi_d(\bigcup_{S' \in \mathsf{Visited}} W(S)[(j,i)^-]) \cap S \supseteq \bigcup_{S' \in \mathsf{Visited}} W(S)[(j,i)^-] \cap S = W(S)[(j,i)^-]$ as for any set of configuration S', $\pi_d(S') \supseteq S'$, or the same with π_{cl}. □

We can now use the previous result to state that $\mathsf{updateRank}$ correctly updates the approximation.

Lemma 38. *The application of $\mathsf{updateRank}$ ensures the following two properties:*

- *if before the call,*
$$\forall S \in \mathsf{Visited},\ \forall i,j \in \mathbb{N},\ W(S)[(j,i+1)] \subseteq \pi_d\Big(\bigcup_{S' \in \mathsf{Visited}} W(S')[(j,i)]\Big) \cap S$$
then, after the call,
$$\forall S \in \mathsf{Visited},\ \forall i,j \in \mathbb{N},\ W(S)[(j,i+1)] = \pi_d\Big(\bigcup_{S' \in \mathsf{Visited}} W(S')[(j,i)]\Big) \cap S;$$
- *if before the call,*
$$\forall S \in \mathsf{Visited},\ \forall i,j \in \mathbb{N},\ W(S)[(j+1,0)] \subseteq \pi_{cl}\Big(\bigcup_{S' \in \mathsf{Visited}} W(S')[(j+1,0)^-]\Big) \cap S$$
then, after the call,
$$\forall S \in \mathsf{Visited},\ \forall j \in \mathbb{N}, W(S)[(j+1,0)] = \pi_{cl}\Big(\bigcup_{S' \in \mathsf{Visited}} W(S')[(j+1,0)^-]\Big) \cap S.$$

Proof. We prove the first property. The same proof can be used for the second one with π_{cl} instead of π_d. Fix $j,i \in \mathbb{N}$ and $S \in \mathsf{Visited}$. Suppose that before the call to $\mathsf{updateRank}$, it holds

$$\forall S \in \mathsf{Visited},\ \forall i,j \in \mathbb{N},\ W(S)[(j,i+1)] \subseteq \pi_d\Big(\bigcup_{S' \in \mathsf{Visited}} W(S')[(j,i)]\Big) \cap S.$$

- if (j,i) is processed, then when the counters of the function are equal to (j,i) we discuss according to the condition $W \subseteq \mathit{Temp}_W$. If it is satisfied, $W(S)[(j,i)]$ is suppressed from the data structure, meaning that $W(S)[(j,i+1)] = \mathit{Temp}_W$. We furthermore have $W \supseteq \mathit{Temp}_W$ as $W \supseteq W(S)[(j,i)]$ as $\pi_d(S) \supseteq S$ and same for π_{cl}, and $W(S)[(j,i)] \supseteq \mathit{Temp}_W$ by Lemma 37. It comes that $W(S)[(j,i+1)] = W$ and the property holds. If the condition $W \subseteq \mathit{Temp}_W$ is not met, either the test $W(S)[(j,i+1)] \subsetneq W$ is satisfied, and in this case the value of $W(S)[(j,i+1)]$ is updated to the correct value, or it is not, and by hypothesis, we already have $W(S)[(j,i+1)] = W = \pi_d(\bigcup_{S' \in \mathsf{Visited}} W(S')[(j,i)]) \cap S$;

– if (j,i) is not processed by $\mathsf{updateRank}$, then it means that

$$\pi_d\Big(\bigcup_{S'\in\mathsf{Visited}} W(S')[(j,i)]\Big)\cap S=\pi_d\Big(\bigcup_{S'\in\mathsf{Visited}} W(S')[(j,i)^-]\Big)\cap s.$$

Indeed it means that for all S' that matter for S, there is no increase in the region between these two ranks. In this case, we know that (j,i) is not a key of $W(S)$ as the rank is not local (else it would have been updated). It comes that $W(S)[(j,i)] = W(S)[(j,i)^-]$. By induction $W(S)[(j,i)^-] = \pi_d\big(\bigcup_{s'\in\mathsf{Visited}} W(S')[(j,i)^-]\big)\cap S$ (the induction is correctly initialized as $(0,1)$ is processed). It comes that $W(S)[(j,i)] = \pi_d\big(\bigcup_{S'\in\mathsf{Visited}} W(S')[(j,i)]\big)\cap S$. $\square$

Now that the properties of the auxiliary function are stated, we discuss of the **while** loop invariants in the main algorithm.

Lemma 39. *The following properties hold at the beginning and end of every iteration of the* **while** *loop of Algorithm 1:*

– *for any $S\in\mathsf{Visited}$ and any $\alpha\in\Sigma$, writing $S'=\mathsf{Post}_\alpha(S)$, we have:*

$$(S,\alpha,S')\in\mathsf{Waiting}\vee\big(S'\in\mathsf{Visited}\wedge(S,\alpha,S')\in\mathsf{Depend}(S')\big)\,;$$

– *for any $S\in\mathsf{Visited}$ and any $i,j\in\mathbb{N}$, we have $W(S)[(j,i)]\subseteq W_i^j$;*
– *for any $S\in\mathsf{Visited}$ and any $i,j\in\mathbb{N}$,*

$$\begin{aligned}W(S)[(j,i+1)]\subseteq\pi_d\Big(\bigcup_{S'\in\mathsf{Visited}} W(S')[(j,i)]\Big)\cap S\wedge&\\ \Big(W(S)[(j,i+1)]=\pi_d\Big(\bigcup_{S'\in\mathsf{Visited}} W(S')[(j,i)]\Big)\cap S\vee&\\ \exists(S,\alpha,S')\in\mathsf{Waiting},\ s.t.\ S'\in\mathsf{Visited}\Big);&\end{aligned}$$

– *for any $S\in\mathsf{Visited}$ and any $j\in\mathbb{N}$,*

$$\begin{aligned}W(S)[(j+1,0)]\subseteq\pi_{cl}(\cup_{S'\in\mathsf{Visited}}W(S')[(j+1,0)^-])\cap S\wedge&\\ \Big(W(S)[(j+1,0)]=\pi_{cl}(\cup_{S'\in\mathsf{Visited}}W(S')[(j+1,0)^-])\cap S\vee&\\ \exists(S,\alpha,S')\in\mathsf{Waiting},\ s.t.\ S'\in\mathsf{Visited}\Big).&\end{aligned}$$

Proof. We prove each point independently.

– We prove the first invariant by induction. Before the loop execution, $\mathsf{Visited}=\{S_0\}$ and $\mathsf{Waiting}=\{(S_0,\alpha,S')\mid S'=\mathsf{Post}_\alpha(S_0)\}$. Hence the property is satisfied. During the loop execution, the property is conserved. Indeed, if $e=(S,\alpha,S')$ is the considered edge, if $S'\in\mathsf{Visited}$ the only transition taken out of $\mathsf{Waiting}$ is e and it is added to $\mathsf{Depend}(S')$. Furthermore no state is added to $\mathsf{Visited}$, hence the property is preserved. Else, $S'\notin\mathsf{Visited}$ and (1) e is added to $\mathsf{Depend}(S')$ which ensures the property for $\mathsf{Visited}\setminus\{S'\}$; (2) $\{(S',\alpha,S'')\mid S''=\mathsf{Post}_\alpha(S')\}$ is added to $\mathsf{Waiting}$, ensuring the property of S'. In both cases, the property is preserved;
– We show the property by induction:

  - before the loop starts, the property holds. Indeed, either $W(S_0)[(0,0)]$ is empty, and in this case all the $W(S_0)[(j,i)]$ are empty, or $W(S_0)[(j,i)] = S_0 \cap \boldsymbol{Pass} \subseteq \boldsymbol{Pass} = W_0^0$. In this case, observe that this satisfies the property, and we know by Lemma 36 that updateRank preserves it;
  - with the same argument as in the base case, we know that $W(S)[(0,0)] \subseteq W_0^0$. Hence, by induction hypothesis, the property is always satisfied before the call to updateRank, and this call preserves it by Lemma 36.
- The property is holds when first entering the loop. Indeed, if $S_0 \cap \boldsymbol{Pass} = \emptyset$, $W(S_0)$ is empty and the property trivially holds. Else, notice that $W(S_0)[(j,i+1)] = W(S_0)[(0,0)] \subseteq \pi_d(W(S_0)[(j,i)]) \cap s$ before the call to updateRank as only $W(S_0)[(0,0)]$ adds some states (*i.e.*, implicitly, all other indices have equal sets). By Lemma 38 the property is ensured upon entering the loop.
  During the loop execution, when S' is a new state, for all states previously in Visited the property is ensured by induction hypothesis. For S', if there is no successor of S' in Visited then the same proof as for S_0 ensures that the property holds at the end of the loop iteration. Else, such a (S', α, S'') is in Waiting with $S'' \in \text{Visited}$, and $W(S')$ is empty, and thus $W(s')[j, i+1]$ is included in every set, and in $\pi_d\big(\bigcup_{S' \in \text{Visited}} W(S')[(j,i)]\big) \cap S$. This is enough to ensure the property.
  When S' is not a new state, the property is ensured for $\text{Visited} \setminus \{S\}$ by induction hypothesis. For S, we know that $W(S)[(j,i+1)] \subseteq \pi_d\big(\bigcup_{S' \in \text{Visited}} W(S')[(j,i)]\big)$. Hence by Lemma 38 the call to updateRank ensures the property;
- this property is proved as the previous one, using the second case of the same lemma. ◻

Finally, we prove that when the algorithm terminates the (W_i^j) sets are correctly computed on the reachable part of the game arena. The property further elaborates that stopping the algorithm early (*i.e.*, when Waiting still contains transitions to be processed) leads to an under approximation of the sets. Early termination does not appear in the code proposed in this paper, but could take place based on any criterion (*e.g.*, number of loops, finite rank in the initial configuration...) added to the while loop condition.

Theorem 40. *Upon termination of the algorithm, the ranks are correctly computed. Early termination leads to an under-approximation is the* $(W_i^j)_{j,i}$.

Proof. This is a corollary of the next—more technical—proposition.

For the following property, we restrict $\mathcal{A}$ to its reachable part. This allows to state the third equation without intersecting W_i^j with $Reach(\mathcal{A})$ or $\bigcup_{S \in \text{Visited}} S$.

Proposition 41. *The following properties are invariants of the while loop of updateRank:*

$$\forall S \in \mathit{Visited},\ \forall i,j \in \mathbb{N}.\ W(S)[(j,i)] \subseteq W_i^j(\mathcal{A}) \tag{3}$$

$$\mathit{Waiting} = \emptyset \Rightarrow \forall q \in \mathit{Reach}_{\mathcal{A}}((l_0, \mathbf{0})), \exists S \in \mathit{Visited}.\ q \in S \tag{4}$$

$$\mathit{Waiting} = \emptyset \Rightarrow \Big(\forall S \in \mathit{Visited}, \forall, j, i.\ W(S)[(j,i)] \supseteq S \cap W_i^j(\mathcal{A})\Big) \tag{5}$$

Equation 3 gives the under approximation that holds during the algorithm, and ensure the properties for early termination. Equations 4 and 5 ensure the correct computation in the case of complete termination (as in this case Waiting is empty).

Proof. The first property is a direct consequence of the second point of Lemma 39.

For the second property, we reason by induction on an execution. Consider $q = (l, v) \in \mathsf{Reach}((l_0, \mathbf{0}))$. There exists a path $q_i \xrightarrow{a_i} q_{i+1}$ in $\mathcal{A}$ with $q_0 = (l_0, \mathbf{0})$ and $q_n = q$. We associate an element of $\mathsf{Visited}$ to each q_i by induction as follow: $q_0 \in S_0$ by construction. If $a_i \in \mathbb{R}$, $S_{i+1} = S_i$. As the elements of $\mathsf{Visited}$ are closed by delays, we have $q_{i+1} \in S_{i+1}$. Else $S_{i+1} = \mathsf{Post}_{a_i}(S_i)$. Observe that S_{i+1} is guaranteed to be in $\mathsf{Visited}$ by the first point of Lemma 39 as $\mathsf{Waiting} = \emptyset$. By induction, we have a $S_n \in \mathsf{Visited}$ such that $q \in S_n$.

In order to prove the third property, we suppose that $\mathsf{Waiting} = \emptyset$ and prove the property by induction:

- for $(0, 0)$ we have that $W(S)[(0, 0)] = S \cap \boldsymbol{Pass} = S \cap W_0^0$;
- for $(j, i+1)$ suppose that the property is verified for (j, i) for any S'. $W(S)[(j, i+1)] = \pi_d\big(\bigcup_{S' \in \mathsf{Visited}} W(S')[(j, i)]\big) \cap S$ by the third point of Lemma 39 (as $\mathsf{Waiting} = \emptyset$). By induction hypothesis $W(S)[(j, i+1)] \supseteq \pi_d\big(\bigcup_{S' \in \mathsf{Visited}} W_i^j \cap S'\big) \cap S$ as π_d is an increasing function. As $\mathsf{Waiting} = \emptyset$, we have Equation 4 and $\bigcup_{S' \in \mathsf{Visited}} S' = \mathsf{Reach}((l_0, \mathbf{0}))$ and as we restricted $\mathcal{A}$ to its reachable part we have $W(S)[(j, i+1)] \supseteq \pi_d\big(\bigcup_{S' \in \mathsf{Visited}} W_i^j\big) \cap S = W_{i+1}^j \cap S$. We thus have our result;
- for $(j+1, i)$ we follow the same ideas using the fourth point of Lemma 39 instead of the third, and the properties of π_{cl} instead of those of π_d. □

6 Conclusion

This paper proposes a game approach to solve a controllability problem in conformance testing of real-time systems specified as timed automata with inputs and outputs (TAIO). Given a specification and a test purpose targetting some behaviours, test synthesis consists in producing test cases that control system inputs and observe outputs and delays. Test cases can then be defined as strategies of a game between the tester and the system, where the satisfaction of the test purpose should be reached, while non-conformance is detected if it occurs. Unfortunately winning strategies do not always exist, even under the simplifying assumption of strong connectivity that we consider, and the tester needs to rely on the system cooperation to reach its goal. Our aim is then to minimize those cooperative outputs and delays, named *control losses*. We define rank-lowering strategies that minimize both the distance to the satisfaction of a test purpose, and the control losses and the distance to the next one. We also exhibit fairness assumptions that are sufficient to make those strategies winning. In terms of testing this means that if the system obeys those assumptions, exhaustiveness of the test synthesis method is ensured, together with soundness, strictness and precision. A symbolic algorithm is proposed to effectively compute these strategies, paving the way to an implementation.

This paper opens numerous directions for future work. First, we intend to tackle partial observation in a more complete and practical way. One direction consists in finding weaker conditions under which approximate determinization [BSJK15] preserves strong connectivity, a condition for the existence of winning strategies. Another possibility would be to suppress the restart transition and the need for exact determinization, and try to make strategies as robust as possible in this very general setting where victory can not be ensured even with a fairness condition. Quantitative aspects could also better meet practical needs. The distance to the goal could also include the time distance or costs of transitions, in particular to avoid restarts when they induce heavy costs but longer and cheaper paths are possible. The fairness assumption could also be refined. For now it is assumed on both the specification and the implementation. If the implementation does not implement some outputs, a tester could detect it with a bounded fairness assumption [Ram98], adapted to the timed context (after sufficiently many experiments traversing some region all outputs have been observed), thus allowing a stronger conformance relation with equality of output sets. A natural extension could also be to complete the approach in a stochastic view, for example by constructing a probabilistic approximation of the implementation behaviour during the test execution, allowing to use this information to improve efficiency or to gradually adapt our fairness expectation and thus the test strategy. Finally, we plan to implement the results of this work in an open tool for the analysis of timed automata (*e.g.*, TChecker, `https://www.labri.fr/perso/herbrete/tchecker/index.html`), experiment on real examples and check the scalability of the method.

Declaration

Partial financial support was received from French ANR project TickTac (ANR-18-CE40-0015). The authors have no competing interests to declare that are relevant to the content of this article.

References

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

[AFH94] Rajeev Alur, Limor Fix, and Thomas A Henzinger. A determinizable class of timed automata. In *International Conference on Computer Aided Verification*, number 818 in Lecture Notes in Computer Science, pages 1–13. Springer, 1994.

[AMPS98a] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. *IFAC Proceedings Volumes*, 31(18):447–452, 1998.

[AMPS98b] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. In *Proceedings of the 5th IFAC Cconference on System Structure and Control (SSC'98)*, pages 469–474. Elsevier, July 1998.

[AMPS98c] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata1. *IFAC Proceedings Volumes*, 31(18):447–452, 1998. 5th IFAC Conference on System Structure and Control 1998 (SSC'98), Nantes, France, 8-10 July.

[BB04] Laura Brandán Briones and Ed Brinksma. A test generation framework for quiescent real-time systems. In Jens Grabowski and Brian Nielsen, editors, *Proceedings of the 4th International Workshop on Formal Approaches to Software Testing (FATES'04)*, volume 3395 of *Lecture Notes in Computer Science*, pages 64–78. Springer-Verlag, September 2004.

[BJSK12] Nathalie Bertrand, Thierry Jéron, Amélie Stainer, and Moez Krichen. Off-line test selection with test purposes for non-deterministic timed automata. *Logical Methods in Computer Science*, 8(4), 2012.

[BSJK15] Nathalie Bertrand, Amélie Stainer, Thierry Jéron, and Moez Krichen. A game approach to determinize timed automata. *Formal Methods in System Design*, 46(1):42–80, February 2015.

[BY04] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 2098 of *Lecture Notes in Computer Science*, pages 87–124. Springer-Verlag, 2004.

[CDF$^+$05] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In Martín Abadi and Luca de Alfaro, editors, *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR'05)*, volume 3653 of *Lecture Notes in Computer Science*, pages 66–80. Springer-Verlag, August 2005.

[CKL98] Richard Castanet, Ousmane Koné, and Patrice Laurençot. On-the-fly test generation for real time protocols. In *Proceedings of the International Conference On Computer Communications and Networks (ICCCN'98)*, pages 378–387. IEEE Comp. Soc. Press, October 1998.

[COG98] Rachel Cardell-Oliver and Tim Glover. A practical and complete algorithm for testing real-time systems. In *Proceedings of the 5th Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98)*, volume 1486 of *Lecture Notes in Computer Science*, pages 251–261. Springer-Verlag, September 1998.

[DLLN08a] Alexandre David, Kim G. Larsen, Shuhao Li, and Brian Nielsen. Cooperative testing of timed systems. In *Proceedings of the 4th Workshop on Model Based Testing (MBT'08)*, volume 220 of *Electronic Notes in Theoretical Computer Science*, pages 79–92, 2008.

[DLLN08b] Alexandre David, Kim G. Larsen, Shuhao Li, and Brian Nielsen. A game-theoretic approach to real-time system testing. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'08)*, pages 486–491, March 2008.

[DLLN09] Alexandre David, KimG. Larsen, Shuhao Li, and Brian Nielsen. Timed testing under partial observability. In *International Conference on Software Testing Verification and Validation (ICST09)*, pages 61–70. IEEE, 2009.

[DT98] Conrado Daws and Stavros Tripakis. Model checking of real-time reachability properties using abstractions. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 313–329, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[ENDK02] Abdeslam En-Nouaary, Radhida Dssouli, and Ferhat Khendek. Timed WP-method: Testing real-time systems. *IEEE Transactions on Software Engineering*, 28(11):1023–1038, November 2002.

[Fin06] Olivier Finkel. Undecidable problems about timed automata. In Eugene Asarin and Patricia Bouyer, editors, *International Conferences on Formal Modelling and Analysis of Timed Systems (FORMATS'06)*, volume 4202 of *Lecture Notes in Computer Science*, pages 187–199. Springer-Verlag, September 2006.

[Hen21] Léo Henry. *There and back again : formal methods and model learning for real-time systems.* PhD thesis, University of Rennes 1, France, December 2021.

[HJM18] Léo Henry, Thierry Jéron, and Nicolas Markey. Control strategies for off-line testing of timed systems. In *SPIN 2018 - International Symposium on Model Checking Software*, pages 171–189, 06 2018.

[HMN16] Robert M Hierons, Mercedes G Merayo, and Manuel Núñez. Controllability through nondeterminism in distributed testing. In *IFIP International Conference on Testing Software and Systems*, pages 89–105. Springer, 2016.

[KLSV03] Dilsun Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems. In *Real-Time Systems Symposium*, pages 166–177, January 2003.

[KT04] Moez Krichen and Stavros Tripakis. Black-box conformance testing for real-time systems. In Susanne Graf and Laurent Mounier, editors, *International Workshop on Model Checking Software (SPIN2004)*, number 2989 in Lecture Notes in Computer Science, pages 109–126, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[KT09] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, June 2009.

[LMN04] Kim Guldstrand Larsen, Marius Mikučionis, and Brian Nielsen. Online testing of real-time systems using Uppaal. In Jens Grabowski and Brian Nielsen, editors, *Proceedings of the 4th International Workshop on Formal Approaches to Software Testing (FATES'04)*, volume 3395 of *Lecture Notes in Computer Science*, pages 79–94. Springer-Verlag, September 2004.

[Min17] Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages*, 4(3-4):120–372, 2017.

[NS03] Brian Nielsen and Arne Skou. Automated test generation from timed automata. *International Journal on Software Tools for Technology Transfer*, 5(1):59–77, November 2003.

[Ram98] Solofo Ramangalahy. Strategies for comformance testing. Research Report 98-010, Max-Planck Institut für Informatik, May 1998.

[RW89] P. J. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, 77(1):81–98, 1989.

[SPKM08] P Vijay Suman, Paritosh K Pandya, Shankara Narayanan Krishna, and Lakshmi Manasa. Timed automata with integer resets: Language inclusion and expressiveness. In *International Conference on Formal Modeling and Analysis of Timed Systems*, pages 78–92. Springer, 2008.

[SVD01] Jan Springintveld, Frits Vaandrager, and Pedro R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.

[Tre96] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996.

[Tri04] Stavros Tripakis. Folk theorems on the determinization and minimization of timed automata. In Kim Guldstrand Larsen and Peter Niebert, editors, *International Conferences on Formal Modelling and Analysis of Timed Systems (FORMATS'03)*, volume 2791 of *Lecture Notes in Computer Science*, pages 182–188, Berlin, Heidelberg, 2004. Springer-Verlag.

[vdB20] Petra van den Boss. *Coverage and Games in Model-Based Testing.* PhD thesis, Radboud University Nijmegen, 2020.

[Yan04] Mihalis Yannakakis. Testing, optimization, and games. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP'04)*, Lecture Notes in Computer Science, pages 28–45. Springer-Verlag, 2004.