Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

More Testable Properties

Yliès Falcone[1], Jean-Claude Fernandez[2], Thierry Jéron[3], Hervé Marchand[3], Laurent Mounier[2]
[1] UJF-Grenoble 1, Laboratoire dInformatique de Grenoble (LIG), Grenoble, France - e-mail: FirstName.LastName@imag.fr
[2] UJF-Grenoble 1, VERIMAG, Grenoble, France - e-mail: FirstName.LastName@imag.fr
[3] Inria, Rennes - Bretagne Atlantique, France - e-mail: FirstName.LastName@inria.fr

Received: date / Revised version: date

Abstract. Testing remains a widely used validation technique for software systems. However, recent needs in software development (e.g., in terms of security concerns) may require to extend this technique in order to address a larger set of properties.

In this article, we explore the set of testable properties within the *Safety-Progress* classification where *testability* means to establish by testing that a relation, between the tested system and the property under scrutiny, holds. We characterize testable properties w.r.t. several relations of interest. For each relation, we give a sufficient condition for a property to be testable. Then, we study and delineate a fine-grain characterization of testable properties: for each *Safety-Progress* class, we identify the subset of testable properties and their corresponding test oracle. Furthermore, we address automatic test generation for the proposed framework by providing a general synthesis technique that allows to obtain canonical testers for the testable properties in the *Safety-Progress* classification. Moreover, we show how the usual notion of quiescence can be taken into account in our general framework, and, how quiescence improves the testability results. Then, we list some existing testing approaches that could benefit from this work by addressing a wider set of properties. Finally, we propose Java-PT, a prototype Java toolbox that implements the results introduced in this article.

1 Introduction

Due to its ability to scale up well and its practical efficiency, testing remains one of the most effective and widely used validation techniques for software systems. However, due to recent needs in the software industry (for instance in terms of security), it is important to reconsider the classes of requirements this technique allows to validate or invalidate. The aim of a testing stage may be either to find defects or to witness expected behaviors on an implementation under test (IUT). From a practical point of view, a test campaign consists in producing a test suite (*test generation*) from some initial system description, and executing it on the system implementation (*test execution*). The test suite consists in a set of test cases, where each test case is a sequence of interactions to be executed by an external tester (performed on the points of control and observation, PCOs). Any execution of a test case should lead to a *test verdict*, indicating if the system succeeded or not on this particular test (or if the test was not conclusive).

Since testing is intrinsically a *partial* validation technique, an important issue raised when conducting a test campaign is to produce and select the most relevant test cases. A possible approach consists in using a *property* to drive the test generation and/or test execution steps. In this case, the property is used to generate the so-called test purposes [19,18] so as to select test cases according to some predefined abstract test scenario. A property may also represent the desired or undesired behavior of the system. As an example, such a property may formalize some security policy describing both prohibited behaviors and user expectations, as considered in [30,22]. Moreover, several approaches (e.g., [6]) combine classical testing techniques and property verification so as to improve the test activity. Most of these approaches used safety[1] and co-safety[2] properties. Then, a natural question is whether other kinds of properties can be tested, which arises in defining a precise notion of *testability*.

In [25,17], Nahm, Grabowski, and Hogrefe addressed the testability issue by discussing the set of temporal properties that can be tested on an implementation. A

[1] Let us recall that safety properties are the properties stating that "a bad thing should not happen".

[2] Let us recall that a co-safety property is a property whose negation is a safety property.

property is said to be *testable* if, from a *finite* test execution σ, it is possible to determine if a given relation $\mathcal{R}$ holds between the set of executions satisfying this property and the set of (finite and infinite) executions that could be produced by possible continuations of σ. Thus, depending on the choice of the relation $\mathcal{R}$, a test campaign can answer specific questions such as:

- $\mathcal{R}1$: Is the set of execution sequences of the IUT *included in* the set of execution sequences described by the property?
- $\mathcal{R}2$: Is the set of execution sequences described by the property *included in* the set of execution sequences of the IUT?
- $\mathcal{R}3$: Is the set of execution sequences of the IUT *equal to* the set of execution sequences described by the property?
- $\mathcal{R}4$: Do the set of execution sequences of the IUT and the set of execution sequences described by the property *intersect*?

In [25,17], this notion of testability is studied w.r.t. the *Safety-Progress* classification (see [4] and Sect. 4) for infinitary properties. The announced classes of testable properties are the safety and guarantee[3] classes. Then, it is not too surprising that most of the previously depicted approaches used safety and co-safety properties during testing. Then, an interesting question is "Are there other properties (beyond safety and guarantee properties) that are also testable?". Answering positively this question would open the way to design and extend testing frameworks so that they can validate a broader class of expected behaviors of software systems, and thus would allow to address the validation of nowadays more and more complex requirements of software systems. As we shall see, in this article we give a positive answer to this question by exhibiting properties in the *Safety-Progress* classification that are strictly more expressive than safety and guarantee properties.

Context. In this paper, we shall use the same notion of testability. We consider a generic approach, where an underlying property is compared to the possibly infinite executions of the IUT triggered by a tester. This property expresses finite and infinite[4] observable behaviors (which may be desired or not). Obviously, the challenge addressed by a tester is to be able to perform this aforementioned comparison using only a finite execution of the IUT. Note that in practice the property under scrutiny is sometimes expressed using more abstract events than the ones occurring at the IUT's execution level. However, this testability problem can still be addressed while abstracting this alphabet discrepancy. Moreover, a characteristic of this testability definition is that it does not require the existence of an executable specification to generate the test cases. As we shall see, using a property (instead of an executable specification) allows to encompass several existing conformance testing approaches.

Main contributions. This article contributes to property-oriented testing activities by leveraging the use of an extension of the *Safety-Progress* classification dedicated to runtime techniques. More specifically, the contributions of the article are as follows:

1. to propose a general approach for property-oriented software testing discussed along four relevant implementation relations, that compare the set of executions of the IUT with those described by a property (e.g., inclusion of the former in the later), and that represent a comprehensive set of property-oriented testing activities;
2. to propose a formal framework that generalizes some existing testing activities (e.g., conformance testing) and provides a formal basis for some other testing activities;
3. to give a precise characterization of testable properties that extends the initial results on testability in [25, 17] by showing that lots of interesting properties (neither safety nor guarantee) are also testable (i.e., there exist test executions allowing to produce a verdict);
4. to propose a framework that allows to easily obtain test oracles producing verdicts according to the possible test executions;
5. to show how this material can be applied to existing test generation frameworks;
6. and, finally, to present the Java toolbox Java-PT, a software implementation allowing a test designer to improve a test activity using a property, following the results proposed in the article.

A preliminary version of this paper appeared in the proceedings of ICTSS'10 [11]. This paper brings the following additional contributions. Firstly, the paper is given a more detailed theoretical treatment. Concerning the implementation relations, we explore the set of testable properties for three additional implementation relations (in [11] only relation $\mathcal{R}1$ mentioned above was studied). Moreover, we add additional examples and complete proofs for all announced results. We propose a deeper study of related work, first by describing some testing frameworks that can be leveraged by the results provided by this paper, and second by showing that our results also encompass the (usual) notion of conformance testing [31]. Finally, we introduce Java-PT by giving some implementation details and usage examples.

Paper organization. The remainder of this paper is organized as follows. First, we present in Sect. 2 a small

[3] In the *Safety-Progress* classification guarantee properties are stating that "a good thing should happen in a finite amount of time". The guarantee class corresponds to the co-safety class in the *Safety-Liveness* classification.

[4] The tester observes a finite execution of the IUT and should state a verdict about all potential continuations of this execution (finite and infinite ones).

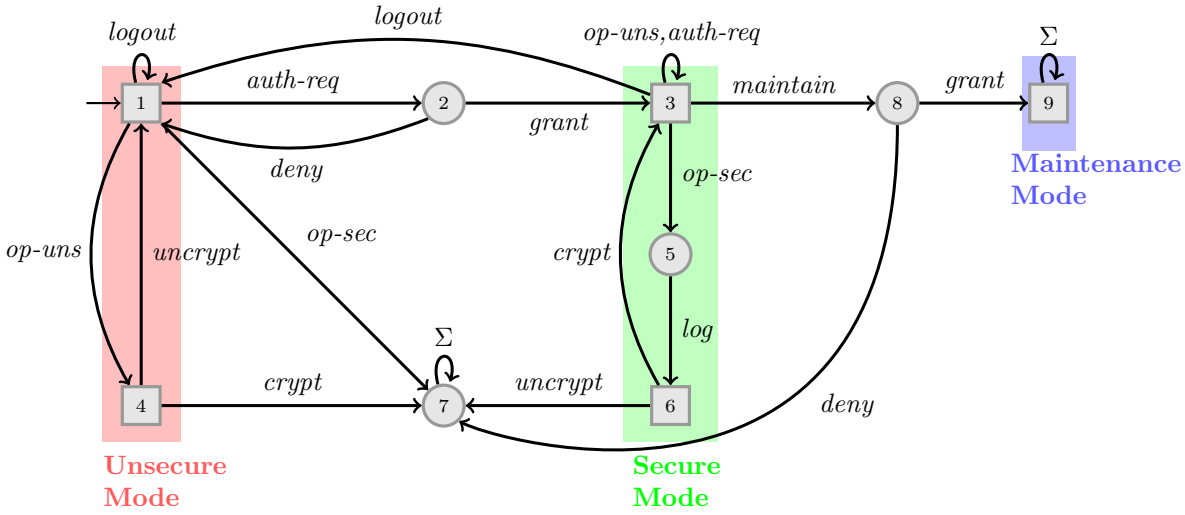


Fig. 1. A simple operating system

motivating example that will be used in the rest of the paper. In Sect. 3, some preliminary concepts and notation are introduced. A quick overview of the *Safety-Progress* classification of properties for runtime validation techniques is given in Sect. 4. Section 5 introduces the notion of testability considered in this paper. In Sect. 6, testable properties are characterized. Automatic test generation is addressed in Sect. 7, and then we show in Sect. 8 how to take into account some notion of *quiescence* from the IUT. Next, in Sect. 9, we overview related work and propose a discussion on the results provided by this paper. A presentation of Java-PT, a prototype tool implementing the results afforded by this paper, is given in Sect. 10. Finally, Sect. 11 gives some concluding remarks and raised perspectives. To lighten the presentation of the *Safety-Progress* classification some concepts are presented informally, the corresponding formal definitions can be found in Appendix A. In order to facilitate the reading of this article, some proofs are sketched, complete versions can been found in Appendix B, and, the main notations used throughout the article are summarized in Appendix C.

2 A motivating example

In this section, using an example, we illustrate informally that there exist interesting properties to be validated which do not belong to the safety and guarantee classes.

Let us consider a (very simple) operating system, providing three execution modes:

- a *non-secure* mode, in which *only* non-secure operations (*op-uns*) are allowed and their results should be not encrypted (*uncrypt*);
- a *secure* mode, in which only secure (*op-sec*) operations can be performed, but such that **(i)** every occurrence of a secure operation should be (automatically) logged (*log*) by the system, and **(ii)** results of secure operations should be encrypted (*crypt*);
- a *system maintenance* mode, in which every operation is permitted.

Switching from the non-secure mode to the secure one needs some authentication. It is achieved by emitting a request (*auth-req*). Such a request can be either granted by the system (*grant*), or denied (*deny*). A logout operation (*logout*) allows to switch back to the non-secure mode. The maintenance mode can be accessed only from the secure mode through action *maintain*. Again, this access can be either granted or denied by the system.

This very abstract system specification can be expressed by the finite-state automaton depicted in Fig. 1. Its alphabet is $\Sigma = \{$*op-uns*, *op-sec*, *auth-req*, *grant*, *deny*, *log*, *crypt*, *uncrypt*, *logout*, *maintain*$\}$. Square states are the accepting states. The non-secure mode (highlighted in red) consists in states $\{1, 4\}$, the secure mode (highlighted in green) in states $\{3, 5, 6\}$ and the maintenance mode (highlighted in blue) in state $\{9\}$. States 2 and 8 are transient states between these modes. State 7 is an "error" state, from which no accepting state is reachable. Testing this system essentially means testing several distinct properties:

- a property ψ_1^{os}, stating that "secure operations are not allowed in the non-secure mode";
- a property ψ_2^{os}, stating that "the maintenance mode is accessible from the secure mode";
- a property ψ_3^{os}, stating that "secure operations are not allowed in the non-secure mode, or, the maintenance mode is called when no operation is ongoing";
- a property ψ_4^{os}, stating that "each secure operation performed has to be logged";
- a property ψ_5^{os}, stating that "the user should be eventually permanently disconnected";
- etc.

Property ψ_1^{os} is clearly a safety property: once violated during an execution sequence, it remains false (on this execution) forever. Property ψ_2^{os} is called a *guarantee* property in the *Safety-Progress* classification: once satisfied during an execution sequence, it remains true (on this execution) forever.

Such properties are known to be testable [25] in the following sense: it is possible to produce a tester able to report a violation of ψ_1^{os} (resp., a satisfaction of ψ_2^{os}) during a finite test execution. Property ψ_4^{os} corresponds to a so-called *response property* in the *Safety-Progress* classification. In particular, valid execution sequences with respect to ψ_4^{os} may contain (possibly infinitely many) invalid prefixes. Therefore, finding, during a test campaign, a finite invalid execution sequence is not a sufficient condition to reject the IUT. We shall see however in this article, that, under certain conditions, this property can be also considered as testable, and how to produce a corresponding tester. Such a result clearly extends previous work ([25,17]), and allows to test a much larger class of properties than the ones considered so far.

3 Preliminaries and notation

In this section, we introduce some preliminary concepts and notations.

3.1 Sequences and execution sequences

The notion of sequence is used to formalize executions. Given an alphabet of actions Σ, a sequence σ on Σ is a total function $\sigma : I \to \Sigma$ where I is either the integer interval $[0, n]$ for some $n \in \mathbb{N}$, or $\mathbb{N}$ itself; where $\mathbb{N}$ is the set of non-negative integers (including 0). The empty sequence is denoted by ϵ. We denote by Σ^* the set of finite sequences over Σ, by $\Sigma^+ \stackrel{\text{def}}{=} \Sigma^* \setminus \{\epsilon\}$ the set of non-empty finite sequences over Σ, and by Σ^ω the set of infinite sequences over Σ. $\Sigma^* \cup \Sigma^\omega$ is noted Σ^∞. The length (number of elements) of a finite sequence σ is noted $|\sigma|$. For $\sigma \in \Sigma^+$ and $n \in [0, |\sigma| - 1]$, the $(n+1)$-th element of σ is noted σ_n. For $\sigma \in \Sigma^*, \sigma' \in \Sigma^\infty$, $\sigma \cdot \sigma'$ is the concatenation of σ and σ'. The sequence $\sigma \in \Sigma^*$ is a *strict prefix* of $\sigma' \in \Sigma^\infty$ (equivalently σ' is a *strict continuation* of σ), noted $\sigma \prec \sigma'$, when $\forall i \in [0, |\sigma| - 1] : \sigma_i = \sigma'_i$ and $|\sigma| < |\sigma'|$. When $\sigma' \in \Sigma^*$, we note $\sigma \preceq \sigma' \stackrel{\text{def}}{=} \sigma \prec \sigma' \vee \sigma = \sigma'$. The set of prefixes of $\sigma \in \Sigma^\infty$ is $\mathit{pref}(\sigma) \stackrel{\text{def}}{=} \{\sigma' \in \Sigma^* \mid \sigma' \preceq \sigma\}$. For a finite sequence $\sigma \in \Sigma^*$, the set of finite continuations is $\mathit{cont}(\sigma) \stackrel{\text{def}}{=} \{\sigma' \in \Sigma^* \mid \sigma \preceq \sigma'\}$. For $\sigma \in \Sigma^\infty \setminus \{\epsilon\}$ and $n < |\sigma|$, $\sigma_{\ldots n}$ is the prefix of σ containing the $n+1$ first elements.

3.2 The IUT as a generator of execution sequences

The IUT is a program $\mathcal{P}$ abstracted as a generator of execution sequences. During a program execution, we are interested in a restricted set of operations that may influence the truth value of the properties we want to test[5]. We abstract these operations by an alphabet Σ. We denote by $\mathcal{P}_\Sigma$ a program with alphabet Σ. The set of execution sequences of $\mathcal{P}_\Sigma$ is denoted by $\mathit{Exec}(\mathcal{P}_\Sigma) \subseteq \Sigma^\infty$. This set is *prefix-closed*, i.e.,

$$\forall \sigma \in \mathit{Exec}(\mathcal{P}_\Sigma) : \mathit{pref}(\sigma) \subseteq \mathit{Exec}(\mathcal{P}_\Sigma).$$

We use $\mathit{Exec}_{\text{f}}(\mathcal{P}_\Sigma)$ (resp. $\mathit{Exec}_\omega(\mathcal{P}_\Sigma)$) to refer to the finite (resp. infinite) execution sequences of $\mathcal{P}_\Sigma$, that is:

- $\mathit{Exec}_{\text{f}}(\mathcal{P}_\Sigma) \stackrel{\text{def}}{=} \mathit{Exec}(\mathcal{P}_\Sigma) \cap \Sigma^*$,
- $\mathit{Exec}_\omega(\mathcal{P}_\Sigma) \stackrel{\text{def}}{=} \mathit{Exec}(\mathcal{P}_\Sigma) \cap \Sigma^\omega$.

3.3 Labelled Transition Systems

A Labelled Transition System (LTS) defined over an alphabet Σ is a 4-tuple $G = (Q^G, q^G_{\text{init}}, \Sigma, \longrightarrow_G)$ where Q^G is a non-empty set of states and $q^G_{\text{init}} \in Q^G$ is the initial state. $\longrightarrow_G \subseteq Q^G \times \Sigma^G \times Q^G$ is the transition relation.

For $Q' \subseteq Q^G$, we note $\overline{Q'} \stackrel{\text{def}}{=} Q^G \setminus Q'$, the complement of Q' in Q^G. Moreover, for $\sigma \in \Sigma^*$ of length n and $q, q' \in Q^G$, we note $q \xrightarrow{\sigma}_G q'$ when $\exists q_1, \ldots, q_{n-1} \in Q^G : q \xrightarrow{\sigma_0}_G q_1 \wedge q_{n-1} \xrightarrow{\sigma_{n-1}}_G q' \wedge \forall i \in [1, n-2] : q_i \xrightarrow{\sigma_i}_G q_{i+1}$. For $q \in Q^G$, $\mathit{Reach}_G(q) \stackrel{\text{def}}{=} \{q' \in Q^G \mid \exists \sigma \in \Sigma^* : q \xrightarrow{\sigma}_G q'\}$ is the set of reachable states from q. For $X \subseteq Q^G$, the set of co-reachable states from X is defined as $\mathit{CoReach}_G(X) \stackrel{\text{def}}{=} \{q \in Q^G \mid \mathit{Reach}_G(q) \cap X \neq \emptyset\}$. For $\sigma \in \Sigma^\infty$, the *run* of σ on G is the sequence of states involved in the execution of σ on G. It is formally defined as $\mathit{run}(\sigma, G) \stackrel{\text{def}}{=} q_0 \cdot q_1 \cdots$ where $\forall i : q_i \xrightarrow{\sigma_i}_G q_{i+1} \wedge q_0 = q^G_{\text{init}}$. An LTS G is said to be *deterministic* if $\forall q \in Q^G$, $\forall e \in \Sigma, \forall q_1, q_2 \in Q^G : (q \xrightarrow{e}_G q_1 \wedge q \xrightarrow{e}_G q_2) \Rightarrow q_1 = q_2$. Finally, G is said to be Σ'-complete for $\Sigma' \subseteq \Sigma$ whenever $\forall q \in Q^G, \forall a \in \Sigma', \exists q' \in Q^G : q \xrightarrow{a}_G q'$. G is said to be complete if it is Σ-complete.

Previous notations transpose to Moore automata (LTSs with output function), Streett automata (the automata used to define properties), and IOLTS (Input Output LTS) that will be introduced in the remainder of this paper.

3.4 Properties as sets of execution sequences

A *finitary property* (resp. an *infinitary property*) is a subset of execution sequences of Σ^* (resp. Σ^ω) (i.e., a finitary or infinitary language). Given a finite (resp. infinite) execution sequence σ and a finitary property ϕ (resp. infinitary property φ), we say that σ *satisfies* ϕ (resp. φ) when $\sigma \in \phi$, noted $\phi(\sigma)$ (resp. $\sigma \in \varphi$, noted $\varphi(\sigma)$). A consequence of this definition is that properties

[5] Note that in practice properties are sometimes expressed using more abstract operations than the ones occurring at the IUT's execution level. However, the testability issues we address in this paper are still valid in spite of this alphabet discrepancy.

we will consider are restricted to *linear time* execution sequences, excluding properties defined on powersets of execution sequences and branching properties (cf. [10]).

3.5 Runtime properties [12]

In this paper we are interested in *runtime properties*, namely the properties that can be used in runtime-based validation techniques (e.g., runtime verification, testing). As stated in the introduction, we need to consider both finite and infinite execution sequences that a program may produce. Runtime properties should characterize satisfaction for both kinds of sequences (finite and infinite) in a uniform way. To do so, we define *r-properties* as pairs $\Pi = (\phi, \varphi) \subseteq \Sigma^* \times \Sigma^\omega$ i.e., ϕ is a finitary language and φ an infinitary language. We say that $\sigma \in \mathit{Exec}(\mathcal{P}_\Sigma)$ satisfies (ϕ, φ), noted $\Pi(\sigma)$, when $\sigma \in \Sigma^* \wedge \phi(\sigma) \vee \sigma \in \Sigma^\omega \wedge \varphi(\sigma)$. The negation of a finitary property ϕ (resp. an infinitary property φ) w.r.t. an alphabet Σ is $\Sigma^* \setminus \phi$ (resp. $\Sigma^\omega \setminus \varphi$), denoted $\overline{\phi}$ (resp. $\overline{\varphi}$) when clear from context. The definition of the negation of an *r-property* follows from the definition of the negation for finitary and infinitary properties. For an *r-property* (ϕ, φ), we define $\overline{(\phi, \varphi)}$ as $(\overline{\phi}, \overline{\varphi})$. Boolean combinations of *r-properties* are defined in a natural way. For $* \in \{\vee, \wedge\}$, $(\phi_1, \varphi_1) * (\phi_2, \varphi_2) \stackrel{\text{def}}{=} (\phi_1 * \phi_2, \varphi_1 * \varphi_2)$.

In the sequel, we will need the notion of *positive and negative determinacy* [27], first introduced by Pnueli and Zaks to define *monitorability* (i.e., to state when it is worth verifying a property at runtime). Here we rephrase this notion in our context of *r-properties*.

Definition 1 (Positive/Negative determinacy of an *r-property* [27]). An *r-property* $\Pi \subseteq \Sigma^* \times \Sigma^\omega$ is said to be:

- positively determined by $\sigma \in \Sigma^*$ if

$$\forall \mu \in \Sigma^\infty : \Pi(\sigma \cdot \mu),$$

  denoted $\oplus\text{—}\mathit{determined}(\sigma, \Pi)$;
- negatively determined by $\sigma \in \Sigma^*$ if

$$\forall \mu \in \Sigma^\infty : \neg\Pi(\sigma \cdot \mu).$$

  denoted $\ominus\text{—}\mathit{determined}(\sigma, \Pi)$.

Intuitively, an *r-property* Π is positively (resp. negatively) determined by a finite sequence σ, if σ satisfies (resp. does not satisfy) Π and every finite and infinite continuation does (resp. does not) satisfy the *r-property*.

Remark 1. One can remark that an *r-property* Π is positively determined iff $\neg\Pi$ is negatively determined, that is: $\forall \sigma \in \Sigma^*, \forall \Pi \subseteq \Sigma^* \times \Sigma^\omega : \oplus\text{—}\mathit{determined}(\sigma, \Pi) \Leftrightarrow \ominus\text{—}\mathit{determined}(\sigma, \neg\Pi)$.

4 A Safety-Progress classification for runtime techniques

The *Safety-Progress* (SP) classification of properties [23, 4] introduces a hierarchy between regular (linear time) properties[6] initially defined as sets of *infinite* execution sequences. The classification has been extended in [12] to also deal with finite-length execution sequences by revisiting it using *r-properties*. In this section, we recall the necessary concepts of the extended version of the *Safety-Progress* classification.

4.1 Informal presentation

The *Safety-Progress* classification is an alternative to the classical *Safety-Liveness* [20,1] dichotomy. Unlike this later, the *Safety-Progress* classification is a hierarchy and not a partition, and provides a finer-grain classification of properties in a uniform way according to 4 views [5]: a language-theoretic view (seeing properties as languages built using specific operators), a logical view (seeing properties as LTL formulas), a topological view (seeing properties as open or closed sets), and an automata view (seeing properties as accepted words of Streett automata [29]).

Recall that the *Safety-Liveness* classification partitions properties into two classes: safety properties (stating that "a bad thing should not happen") and liveness properties (stating that "a good thing should eventually happen"). The *Safety-Progress* classification introduces four basic classes of properties that are distinguished according to how a good thing is supposed to happen during an (infinite) execution:

- *safety* properties require that only good things happen;
- *guarantee* properties require that a good thing happens *at least once*;
- *response* properties require that a good thing happens *infinitely often*;
- *persistence* properties require that a good thing happens *persistently* (equivalently, to not occur a finite number of times).

Moreover, two composite classes are defined over basic classes: obligation (resp. reactivity) properties are obtained by finite Boolean combinations of safety and guarantee (resp. response and persistence) properties. Any linear-time property, i.e., that can be expressed as a set of sequences, belongs to the reactivity class.

A graphical and hierarchical representation of the *Safety-Progress* classification of properties is depicted in Fig. 2. A link between two classes means that the upper class strictly contains the lower one. Further details and results can be found in [13]. Here, we consider only

[6] In the remainder of this paper, the term property will stand for regular property.

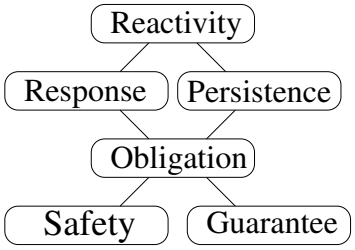


Fig. 2. The SP classification

the needed results from the language-theoretic and the automata views.

4.2 The language-theoretic view of r-properties

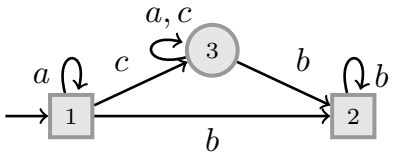
The language-theoretic view of the SP classification is based on the construction of infinitary properties and finitary properties from finitary ones. It relies on the use of four operators A, E, R, P (building infinitary properties) and four operators A_f, E_f, R_f, P_f (building finitary properties) applied to finitary properties. Formal definitions were introduced in [13] and are recalled in Appendix A. In the following $\psi \subseteq \Sigma^*$ is a finitary property.

$A(\psi)$ consists of all infinite words σ s.t. *all* prefixes of σ belong to ψ. $E(\psi)$ consists of all infinite words σ s.t. *some* prefixes of σ belong to ψ. $R(\psi)$ consists of all infinite words σ s.t. *infinitely many* prefixes of σ belong to ψ. $P(\psi)$ consists of all infinite words σ s.t. *all but finitely many* prefixes of σ belong to ψ.

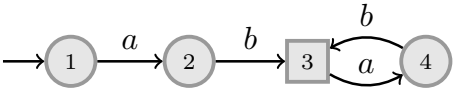
$A_f(\psi)$ consists of all finite words σ s.t. *all* prefixes of σ belong to ψ. One can observe that $A_f(\psi)$ is the largest prefix-closed subset of ψ. $E_f(\psi)$ consists of all finite words σ s.t. *some* prefixes of σ belong to ψ. One can observe that $E_f(\psi) = \psi \cdot \Sigma^*$. $R_f(\psi)$ consists of all finite words σ s.t. $\psi(\sigma)$ and there exists an infinite number of continuations σ' of σ also belonging to ψ. $P_f(\psi)$ consists of all finite words σ belonging to ψ s.t. there exists a continuation σ' of σ s.t. σ' persistently has continuations staying in ψ.

Example 1 (Language operators). Let us illustrate the application of the previously introduced language operators on some finitary properties:

- Let us consider $\Sigma_1 = \{a, b, c\}$ and the finitary property $\psi_1 = a^* \cdot \left(b^* + c \cdot (c + a)^* \cdot b^+\right)$ defined by the deterministic finite-state automaton (DFA) in Fig. 3a with accepting states $1, 2$. We have:
  - $A(\psi_1) = a^\omega + a^+ \cdot b^\omega, A_f(\psi_1) = a^* \cdot b^*$,
  - $E(\psi_1) = \Sigma_1^\omega, E_f(\psi_1) = \Sigma_1^*$,
  - $R(\psi_1) = a^\omega + a^* \cdot (b + (c \cdot (a + c)^* \cdot b) \cdot b^\omega$, $R_f(\psi_1) = a^* + a^* \cdot (b + (c \cdot (a + c)^* \cdot b) \cdot b^*$,
  - $P(\psi_1) = a^\omega + a^* \cdot (b + (c \cdot (a + c)^* \cdot b) \cdot b^\omega$, $P_f(\psi_1) = a^* + a^* \cdot (b + (c \cdot (a + c)^* \cdot b) \cdot b^*$.
- Let us consider $\Sigma_2 = \{a, b\}$, and the finitary property $\psi_2 = (a \cdot b)^+$ defined by the DFA depicted in Fig. 3b. We have:



(a) $\mathcal{A}_{\psi_1}$ defining $\psi_1 = a^* \cdot \left(b^* + c \cdot (c + a)^* \cdot b^+\right)$



(b) $\mathcal{A}_{\psi_2}$ defining $\psi_2 = (a \cdot b)^+$

Fig. 3. DFA for ψ_1 and ψ_2

  - $A(\psi_2) = A_f(\psi_2) = \emptyset$,
  - $E(\psi_2) = a \cdot b \cdot \Sigma_2^\omega, E_f(\psi_2) = a \cdot b \cdot \Sigma_2^*$,
  - $R(\psi_2) = a \cdot b \cdot (a \cdot b)^\omega, R_f(\psi_2) = a \cdot b \cdot (a \cdot b)^*$,
  - $P(\psi_2) = P_f(\psi_2) = \emptyset$.

4.3 The automata view of r-properties [12]

We define a variant of deterministic and complete Streett automata (introduced in [29] and used in [5]). We add to original Streett automata an acceptance condition for finite sequences in such a way that these automata uniformly define *r-properties*.

Definition 2 (Streett automaton). A deterministic Streett automaton $\mathcal{A}$ is a tuple $(Q^{\mathcal{A}}, q^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$. The set $Q^{\mathcal{A}}$ is the set of states, $q^{\mathcal{A}}_{\text{init}} \in Q^{\mathcal{A}}$ is the initial state. $\longrightarrow_{\mathcal{A}}$: $Q^{\mathcal{A}} \times \Sigma \to Q^{\mathcal{A}}$ is the (complete) transition function. $\{(R_1, P_1), \ldots, (R_m, P_m)\}$ is the set of accepting pairs, for $i \in [1, m]$, $R_i \subseteq Q^{\mathcal{A}}$ and $P_i \subseteq Q^{\mathcal{A}}$ are the sets of recurrent and persistent states respectively. Without loss of generality, we suppose that in the considered Streett automata, all states are reachable from the initial state: $Reach_{\mathcal{A}}(q^{\mathcal{A}}_{\text{init}}) = Q^{\mathcal{A}}$.

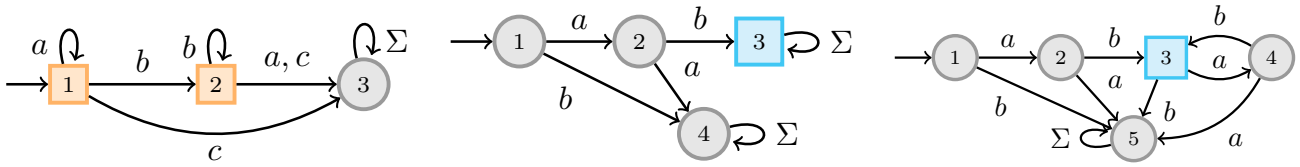
An automaton with m accepting pairs is called an m-automaton. A *plain*-automaton is a 1-automaton, and R_1 and P_1 are then referred as R and P. For an execution sequence $\sigma \in \Sigma^\omega$ on a Streett automaton $\mathcal{A}$, $vinf(\sigma, \mathcal{A})$ denotes the set of states appearing infinitely often in $run(\sigma, \mathcal{A})$.

Definition 3 (Acceptance conditions of Streett automata). Considering an m-automaton $\mathcal{A} = (Q^{\mathcal{A}}, q^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$, the acceptance conditions are defined as follows.

- For $\sigma \in \Sigma^\omega$, $\mathcal{A}$ accepts σ if

  $$\forall i \in [1, m] : vinf(\sigma, \mathcal{A}) \cap R_i \neq \emptyset \vee vinf(\sigma, \mathcal{A}) \subseteq P_i.$$

  An infinite sequence σ is accepted by $\mathcal{A}$ if, during the run of σ on $\mathcal{A}$, the set of states visited infinitely often are either all in P_i-states or contains at least one R_i-state, for each $i \in [1, m]$.



(a) $\mathcal{A}_{\Pi_1}$ safety automaton s.t. $P = \{1, 2\}$ and $R = \emptyset$

(b) $\mathcal{A}_{\Pi_2}$ guarantee automaton s.t. $R = \{3\}$ and $P = \emptyset$

(c) $\mathcal{A}_{\Pi_3}$ response automaton s.t. $R = \{3\}$ and $P = \emptyset$

Fig. 4. Streett Automata for $(A_f(\psi_1), A(\psi_1))$, $(E_f(\psi_2), E(\psi_2))$, $(R_f(\psi_2), R(\psi_2))$

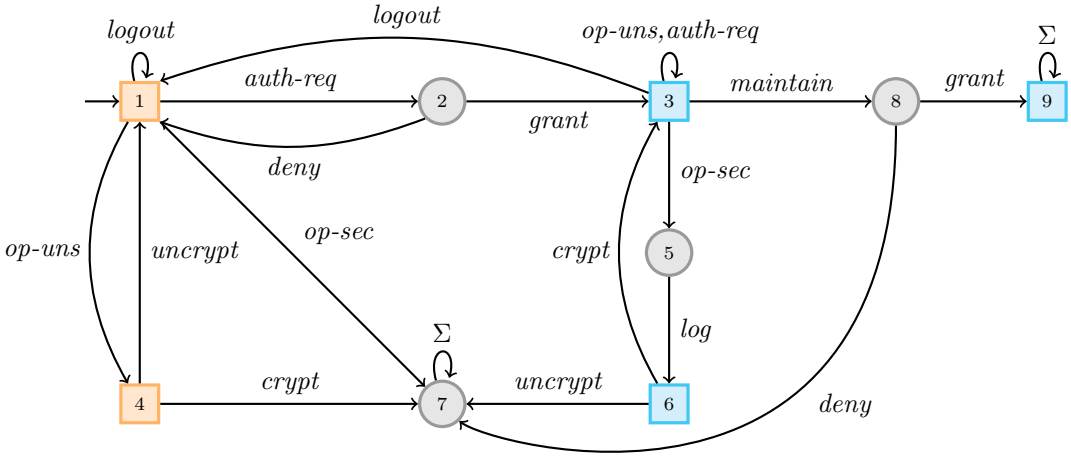


Fig. 5. $\mathcal{A}_{\Pi^{\text{OP}}}$: reactivity Streett automaton specifying the behavior of the operating system

– For $\sigma \in \Sigma^*$, $\mathcal{A}$ accepts σ if

$$q_{\text{init}}^{\mathcal{A}} \xrightarrow{\sigma}_{\mathcal{A}} q \Rightarrow \forall i \in [1, m] : q \in R_i \cup P_i.$$

A finite sequence σ is accepted by $\mathcal{A}$ if the run of σ on $\mathcal{A}$ ends in either a P_i-state or in a R_i-state, for each $i \in [1, m]$.

Example 2 (Acceptance conditions of Streett automata). Following the definition of the acceptance conditions, we can determine the accepted sequences of some Streett automata:

– The Streett automaton $\mathcal{A}_{\Pi_1}$ in Fig. 4a accepts finite sequences whose runs end either in state 1 or 2 and infinite sequences whose runs visit infinitely often only states 1 and/or 2. One can remark that the set of finite (resp. infinite) sequences accepted by $\mathcal{A}_{\Pi_1}$ is $A_f(\psi_1)$ (resp. $A(\psi_1)$), see Example 1.
– The Streett automata $\mathcal{A}_{\Pi_2}$ in Fig. 4b and $\mathcal{A}_{\Pi_3}$ in Fig. 4c accept finite sequences whose runs end in state 3, and infinite sequences whose runs visit infinitely often at least state 3. One can remark that the set of finite (resp. infinite) sequences accepted by $\mathcal{A}_{\Pi_2}$ is $E_f(\psi_2)$ (resp. $E(\psi_2)$). Similarly, the set of finite (resp. infinite) sequences accepted by $\mathcal{A}_{\Pi_3}$ is $R_f(\psi_2)$ (resp. $R(\psi_2)$).

As stated before, in the automata view, Streett automata are used to define *r-properties*:

Definition 4 (*r-property* defined by a Streett automaton). A Streett automaton $\mathcal{A}$ *defines* an *r-property* $(\phi, \varphi) \subseteq \Sigma^* \times \Sigma^\omega$ if and only if the set of finite sequences accepted by $\mathcal{A}$ equals ϕ and the set of infinite sequences accepted by $\mathcal{A}$ equals φ.

4.4 The hierarchy of r-properties

The hierarchical organization of *r-properties* can be seen in the language view using the operators and in the automata view using syntactic restrictions on Streett automata.

Definition 5 (Safety-Progress classes). An *r-property* Π, defined by $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$, is said to be:

– A *safety r-property* if $\Pi = (A_f(\psi), A(\psi))$ for some $\psi \subseteq \Sigma^*$ or equivalently $\mathcal{A}_\Pi$ is a plain-automaton s.t. $R = \emptyset$ and there is no transition from $\overline{P}$ to P.
– A *guarantee r-property* if $\Pi = (E_f(\psi), E(\psi))$ for some $\psi \subseteq \Sigma^*$ or equivalently $\mathcal{A}_\Pi$ is a plain-automaton s.t. $P = \emptyset$ and there is no transition from R to $\overline{R}$.
– An *m-obligation r-property* if $\Pi = \bigcap_{i=1}^{m}(S_i(\psi_i) \cup G_i(\psi_i'))$ or $\Pi = \bigcup_{i=1}^{m}(S_i(\psi_i) \cap G_i(\psi_i'))$ where $S(\psi_i)$ (resp. $G(\psi_i')$) are safety (resp. guarantee) *r-properties* defined over the ψ_i and the ψ_i'; or equivalently $\mathcal{A}_\Pi$ is an m-automaton s.t. for $i \in [1, m]$ there is no transition from $\overline{P_i}$ to P_i and from R_i to $\overline{R_i}$.

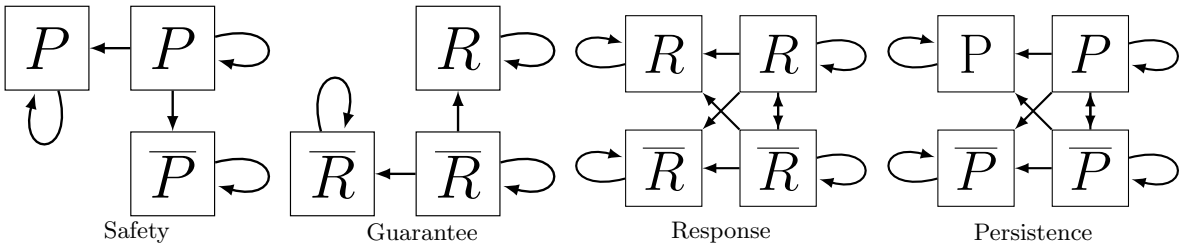


Fig. 6. Schematic illustrations of the shapes of Streett automata for basic classes

– A *response r-property* if $\Pi = (R_f(\psi), R(\psi))$ for some $\psi \subseteq \Sigma^*$ or equivalently $\mathcal{A}_\Pi$ is a plain-automaton s.t. $P = \emptyset$.
– A *persistence r-property* if $\Pi = (P_f(\psi), P(\psi))$ for some $\psi \subseteq \Sigma^*$ or equivalently $\mathcal{A}_\Pi$ is a plain-automaton s.t. $R = \emptyset$.
– A *reactivity r-property* if Π is obtained by finite Boolean combinations of response and persistence *r-properties* or equivalently $\mathcal{A}_\Pi$ is an unrestricted automaton.

An *r-property* of a given class is *pure* when not belonging to any other sub-class.

Example 3 (r-properties).

– The Streett automaton $\mathcal{A}_{\Pi_1}$ in Fig. 4a is a safety automaton and defines the safety *r-property* $(A_f(\psi_1), A(\psi_1))$ built upon ψ_1.
– The Streett automaton $\mathcal{A}_{\Pi_2}$ in Fig. 4b is a guarantee automaton and defines the guarantee *r-property* $(E_f(\psi_2), E(\psi_2))$ built upon ψ_2.
– The Streett automaton $\mathcal{A}_{\Pi_3}$ in Fig. 4c is a response automaton and defines the response *r-property* $(R_f(\psi_2), R(\psi_2))$ built upon ψ_2.

Example 4 (r-properties for the operating system). The global specification of the operating system introduced in Section 2 can be formalized as one general reactivity *r-property* Π^{op} defined by the Streett automaton $\mathcal{A}_{\Pi^{\text{op}}}$ depicted in Fig. 5. For the sake of readability, we have represented only the most interesting transitions of the automaton. In the states where transitions are omitted for some events, there is implicitly a transition starting from this state, labelled with each omitted event, and ending in state 7. Note that, using an *r-property* (here through recurrent and persistent states of a Streett automaton) allow us to more precisely formalize the specification for infinite sequences. Moreover, additionally to the properties mentioned in Section 2, the automaton formalizes additional requirements. For instance, authentication granting should be "fair": the authentication request can not be denied forever (e.g., the execution $(\mathit{auth\text{-}req} \cdot \mathit{deny})^\omega$ is not accepted by the automaton).

Moreover, from this reactivity *r-property*, one can derive several smaller *r-properties* defined using Streett automata. These *r-properties* formalize the informal properties introduced in Section 2.

– The safety *r-property* Π_1^{op} formalizes the property ψ_1^{op} and is defined by $\mathcal{A}_{\Pi_1^{\text{op}}}$ depicted in Fig. 7a.
– The guarantee *r-property* Π_2^{op} formalizes the property ψ_2^{op} and is defined by $\mathcal{A}_{\Pi_2^{\text{op}}}$ depicted in Fig. 7c.
– The obligation *r-property* Π_3^{op} formalizes the property ψ_3^{op} and is defined by $\mathcal{A}_{\Pi_3^{\text{op}}}$ depicted in Fig. 7b.
– The response *r-property* Π_4^{op} formalizes the property ψ_4^{op} and is defined by $\mathcal{A}_{\Pi_4^{\text{op}}}$ depicted in Fig. 7d.
– The persistence *r-property* Π_5^{op} formalizes the property ψ_5^{op} and is defined by $\mathcal{A}_{\Pi_5^{\text{op}}}$ depicted in Fig. 7e.

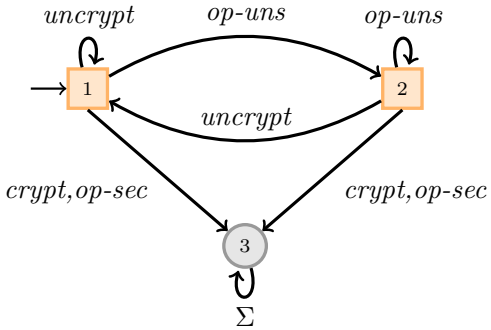
Fig. 6 illustrates the syntactic restrictions on Streett automata for each basic class. A squared box represents a group of states. States are grouped according to whether they are recurrent or not (respectively denoted by "R" and "$\overline{R}$"), and persistent or not (respectively denoted by "P" and "$\overline{P}$"). Arrows represent possible transitions between groups of states[7]. For instance, for safety automata, there are P-states, $\overline{P}$-states, and only $\overline{R}$-states. For this kind of automata, the P-states can be distinguished according to whether $\overline{P}$-states can be reached (the right-hand side group of states) or not (the left-hand side group of states).

To refine Fig. 2, a graphical representation of the *Safety-Progress* hierarchy of properties is depicted in Fig. 8: for each class of properties, characterizations are recalled in the language-theoretic and automata views, as defined in Definition 5.

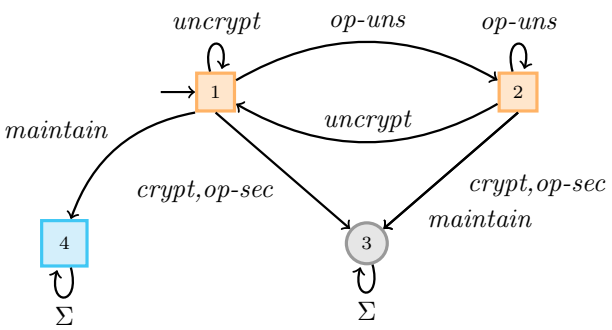
5 Some notions of testability

Recall that $\mathit{Exec}(\mathcal{P}_\Sigma) = \mathit{Exec}_{\text{f}}(\mathcal{P}_\Sigma) \cup \mathit{Exec}_\omega(\mathcal{P}_\Sigma)$. From some *finite* interaction with the underlying IUT, the tester observes a sequence of events σ in Σ^*. We study the conditions, for a such tester, using this sequence σ, to determine whether a given relation holds between the set of *all* (finite and infinite) execution sequences that can be produced by the IUT ($\mathit{Exec}(\mathcal{P}_\Sigma)$), and the set of sequences satisfying the *r-property* Π. Roughly speaking, the challenge addressed by a tester is thus to determine a

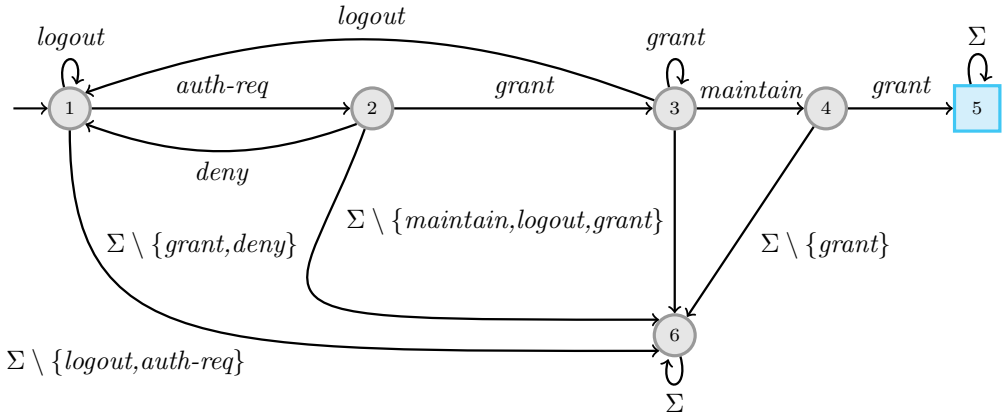
[7] This will be used later in the paper for test generation from properties (Sect. 7).



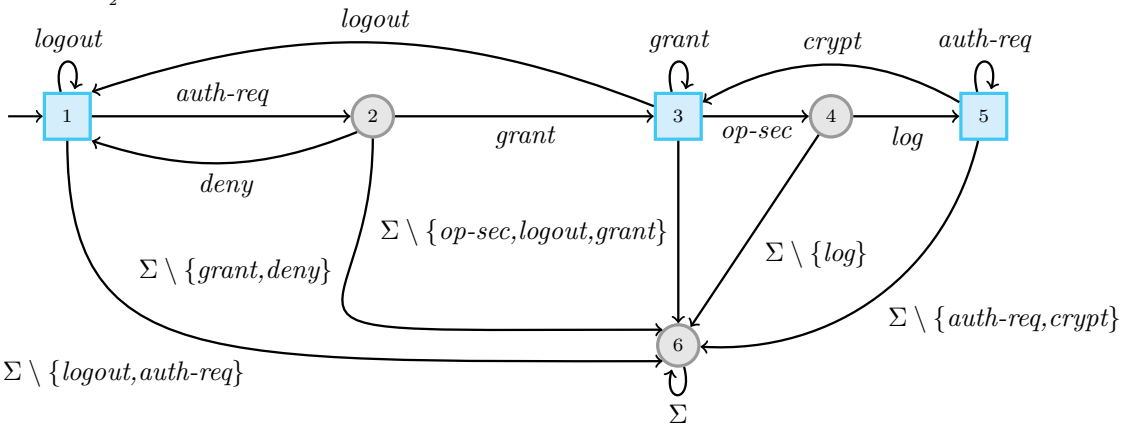
(a) $\mathcal{A}_{\Pi_1^{\mathrm{op}}}$: safety automaton s.t. $P = \{1,2\}$, $R = \emptyset$, and $\Sigma = \{uncrypt, crypt, op\text{-}uns, op\text{-}sec\}$



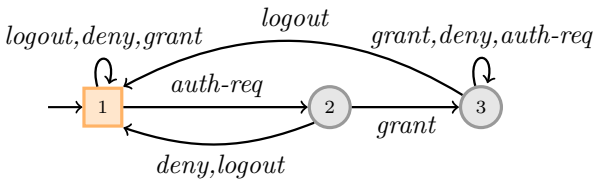
(b) $\mathcal{A}_{\Pi_3^{\mathrm{op}}}$ obligation automaton s.t. $P = \{1,2\}$, $R = \{4\}$, and $\Sigma = \{uncrypt, crypt, op\text{-}uns, op\text{-}sec, maintain\}$



(c) $\mathcal{A}_{\Pi_2^{\mathrm{op}}}$: guarantee automaton s.t. $R = \{5\}$, $P = \emptyset$, and $\Sigma = \{auth\text{-}req, grant, deny, logout, maintain\}$



(d) $\mathcal{A}_{\Pi_4^{\mathrm{op}}}$: response automaton s.t. $R = \{1,3,5\}$, $P = \emptyset$, and $\Sigma = \{auth\text{-}req, grant, deny, logout, crypt, uncrypt, op\text{-}sec\}$



(e) $\mathcal{A}_{\Pi_5^{\mathrm{op}}}$: persistence automaton s.t. $R = \emptyset$, $P = \{1\}$, and $\Sigma = \{grant, deny, logout\}$

Fig. 7. Some Streett automata for the operating system

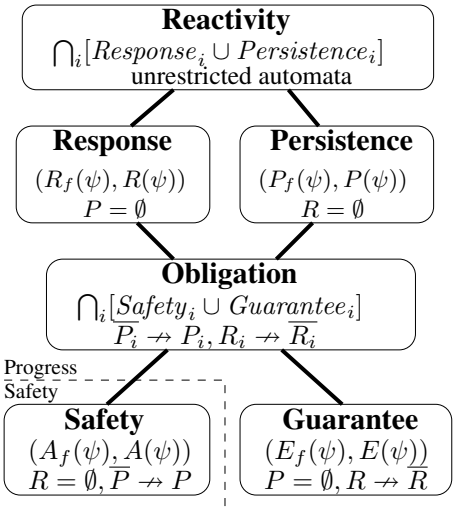


Fig. 8. The SP classification

verdict between Π and $Exec(\mathcal{P}_\Sigma)$, from a finite sequence taken from $Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$[8].

Let us recall that the *r-property* is a pair consisting of a set of finite sequences and a set of infinite sequences. In the sequel, we shall compare this pair to the set of (finite and infinite) execution sequences of the IUT. As noticed in [25], one may consider several possible relations between the execution sequences produced by the program and those described by the property. Those relations are recalled here in the context of *r-properties*.

Definition 6 (Relations between IUT sequences and an *r-property* [25]). The possible relations of interest between $Exec(\mathcal{P}_\Sigma)$ and $\Pi = (\phi, \varphi)$ are:

- $Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \subseteq \phi$ and $Exec_\omega(\mathcal{P}_\Sigma) \subseteq \varphi$: the IUT *respects* the *r-property*; all behaviors of the IUT are allowed by the *r-property* (denoted $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$).
- $Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) = \phi$ and $Exec_\omega(\mathcal{P}_\Sigma) = \varphi$: the observable behaviors of the IUT are exactly those described by the *r-property* (denoted $Exec(\mathcal{P}_\Sigma) = \Pi$).
- $Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \cap \phi \neq \emptyset$ and $Exec_\omega(\mathcal{P}_\Sigma) \cap \varphi \neq \emptyset$: the behaviors expected by the *r-property* and those of the IUT *are not disjoint* (denoted $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$).
- $\phi \subseteq Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$ and $\varphi \subseteq Exec_\omega(\mathcal{P}_\Sigma)$: the IUT *implements* the *r-property*; all behaviors described by the *r-property are feasible* by the IUT (denoted $\Pi \subseteq Exec(\mathcal{P}_\Sigma)$).

We use $\mathcal{R}$ to denote a relation ranging over the ones described in Definition 6. By $\mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi)$, we denote that the relation $\mathcal{R}$ holds between $Exec(\mathcal{P}_\Sigma)$ and Π. The test verdict is thus determined according to the conclusions that one can obtain for the considered relation. In essence, a tester can and must only determine a verdict from a *finite test execution* $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$. In Sect. 6, we will also study the conditions to state weaker verdicts on a *single* execution sequence.

Definition 7 (Verdicts [25]). Given a relation $\mathcal{R}$ between $Exec(\mathcal{P}_\Sigma)$ and Π, and a finite test execution $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$, the tester produces verdicts as follows:

- *pass* if σ allows to determine that $\mathcal{R}$ holds;
- *fail* if σ allows to determine that $\mathcal{R}$ does not hold;
- *unknown*[9] otherwise.

We note $verdict(\sigma, \mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi))$ the verdict that the observation of σ allows to determine. Let us remark the two following practical problems:

- In general, the IUT may be a program producing infinite-length execution sequences. Obviously these sequences cannot be evaluated by a tester w.r.t. Π.
- Moreover, finite execution sequences contained in the *r-property* cannot be processed easily. For instance:
  - if for example a guarantee or response *r-property* Π the test execution exhibits a sequence $\sigma \notin \Pi$, deciding to stop the test is a critical issue. Actually, nothing allows to claim that a future continuation of the test execution would not exhibit a new sequence belonging to the *r-property*, i.e., $\sigma' \in \Sigma^\infty$ s.t. $\Pi(\sigma \cdot \sigma')$.
  - conversely, for example a safety *r-property* Π, the test might exhibit $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$ s.t. $\Pi(\sigma)$, but continuing the test might also exhibit $\sigma' \in \Sigma^*$ s.t. $\sigma \cdot \sigma' \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \wedge \neg\Pi(\sigma \cdot \sigma')$.

Thus, the test should be stopped only when there is no doubt regarding the verdict to be established. Following [25], we propose a notion of testability that takes into account the aforementioned practical limitations, and that is set in the context of the Safety-Progress classification.

Definition 8 (Testability). An *r-property* Π is said to be *testable* on $\mathcal{P}_\Sigma$ w.r.t. the relation $\mathcal{R}$ if there exists a sequence $\sigma \in \Sigma^*$ s.t. $verdict(\sigma, \mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi)) \in \{pass, fail\}$.

Intuitively, this condition compels the existence of a sequence which, if played on the IUT, allows to determine *for sure*, whether the relation holds or not. Let us note that this definition entails to synthesize a test oracle which allows to determine $\mathcal{R}\big(Exec(\mathcal{P}_\Sigma), \Pi\big)$ from the observation of a sequence $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$.

A test oracle is a Moore automaton parameterized by a test relation as shown in Definition 6. It reads incrementally an execution sequence $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$ and produces verdicts in $\{pass, fail, unknown\}$.

[8] Or from a finite set of finite sequences, as a straightforward extension.

[9] In [25], this case is associated to the inconclusive verdict. Here we choose to state it as an *unknown* verdict instead. Indeed, in conformance testing, inconclusive verdicts are produced by a tester when the current test execution will not allow to reach a pass or fail verdict and is often used in association with a test purpose. Furthermore, we believe that the term "unknown" better corresponds to the fact that knowing whether the relation between $Exec(\mathcal{P}_\Sigma)$ and Π holds or not is not yet possible.

Definition 9 (Test Oracle). A *test oracle* $\mathcal{O}$ for an IUT $\mathcal{P}_\Sigma$, a relation $\mathcal{R}$ and an *r-property* Π is a deterministic Moore automaton, i.e., a 5-tuple $(Q^{\mathcal{O}}, q^{\mathcal{O}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{O}}, \Gamma^{\mathcal{O}})$. The finite set $Q^{\mathcal{O}}$ denotes the control states and $q^{\mathcal{O}}_{\text{init}} \in Q^{\mathcal{O}}$ is the initial state. The complete function $\longrightarrow_{\mathcal{O}}$: $Q^{\mathcal{O}} \times \Sigma \to Q^{\mathcal{O}}$ is the transition function. The output function $\Gamma^{\mathcal{O}} : Q^{\mathcal{O}} \to \{pass, fail, unknown\}$ produces verdicts in such a way that any state q emitting a *pass* or a *fail* verdict is final, i.e., $\forall q \in Q^{\mathcal{O}} : \Gamma^{\mathcal{O}}(q) \in \{pass, fail\} \Rightarrow q \xrightarrow{e}_{\mathcal{O}} q$, for any $e \in \Sigma$.

Definition 10 (Soundness and completeness of a test oracle). The output function $\Gamma^{\mathcal{O}}$ of a test oracle $\mathcal{O} = (Q^{\mathcal{O}}, q^{\mathcal{O}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{O}}, \Gamma^{\mathcal{O}})$ should produce verdicts in the following way:

– soundness: $\forall q \in Q^{\mathcal{O}}$:

$* \ \Gamma^{\mathcal{O}}(q) = pass \Rightarrow \big(\forall \sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) : q^{\mathcal{O}}_{\text{init}} \xrightarrow{\sigma}_{\mathcal{O}} q \Rightarrow verdict(\sigma, \mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi)) = pass\big)$

$* \ \Gamma^{\mathcal{O}}(q) = fail \Rightarrow \big(\forall \sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) : q^{\mathcal{O}}_{\text{init}} \xrightarrow{\sigma}_{\mathcal{O}} q \Rightarrow verdict(\sigma, \mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi)) = fail\big)$,

– completeness: $\forall \sigma \in \Sigma^*$:

$* \ \sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \wedge verdict(\sigma, \mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi)) = pass \Rightarrow \exists q \in Q^{\mathcal{O}} : q^{\mathcal{O}}_{\text{init}} \xrightarrow{\sigma}_{\mathcal{O}} q \wedge \Gamma^{\mathcal{O}}(q) = pass$

$* \ \sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \wedge verdict(\sigma, \mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi)) = fail \Rightarrow \exists q \in Q^{\mathcal{O}} : q^{\mathcal{O}}_{\text{init}} \xrightarrow{\sigma}_{\mathcal{O}} q \wedge \Gamma^{\mathcal{O}}(q) = fail.$

Intuitively, a test oracle is sound if the verdicts it produces are correct regarding the relation between $Exec(\mathcal{P}_\Sigma)$ and Π. It is complete if it produces the appropriate verdict for every finite sequence permitting so. Note that, implicitly, the *unknown* evaluation is never produced after a *fail* or *pass* verdict.

6 Testable properties without executable specification

In this section we shall see that the framework of *r-properties* (Sect. 4) allows to determine the testability, according to several relations between $Exec(\mathcal{P}_\Sigma)$ and Π, of the different classes of properties using positive and negative determinacy (Definition 1). Moreover, this framework also provides a computable test oracle. Furthermore, we will be able to characterize which test sequences allow to establish sought verdicts. Then, we will determine which verdict has to be produced in accordance with a given test sequence.

6.1 For the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$

6.1.1 Obtainable verdicts and sufficient conditions

For the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$, the unique verdicts that may be produced are *fail* and *unknown*. We explicit this below.

A *pass* verdict means that all execution sequences of the IUT $\mathcal{P}_\Sigma$ belong to Π. The unique case where it is possible to establish a *pass* verdict is in the trivial case where $\Pi = (\Sigma^*, \Sigma^\omega)$, i.e., the *r-property* Π is always satisfied. Obviously, any implementation with alphabet Σ satisfies this relation. In other cases, in practice it is impossible to obtain such a verdict (whatever is the property class under consideration), since the whole set $Exec(\mathcal{P}_\Sigma)$ is usually unknown from the tester. For instance, one can imagine a guarantee property Π s.t. first $\exists \sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) : \Pi(\sigma)$ (and thus $\forall \sigma' \in \Sigma^\infty : \Pi(\sigma \cdot \sigma')$) and second $\exists \sigma'' \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) : \neg\Pi(\sigma'')$ and $\forall \sigma''' \in \Sigma^\infty : \neg\Pi(\sigma'' \cdot \sigma''')$.

Remark 2. In the following, we will also study the conditions under which it is possible to state *weak pass* verdicts, when reasoning on a *single* execution sequence of the IUT. For instance, for the previously mentioned guarantee *r-property*, we will produce a *weak pass* verdict for the sequence σ.

A *fail* verdict means that there exist some sequences of the program which are not in Π. In order to produce a *fail* verdict, it is sufficient to exhibit an execution sequence of $\mathcal{P}_\Sigma$ s.t. Π is *negatively determined* by this sequence:

Property 1 (Sufficient condition to produce a *fail* verdict). Negative determinacy is a sufficient condition for a sequence to be associated to a *fail* verdict:

$$verdict(\sigma, Exec(\mathcal{P}_\Sigma) \subseteq \Pi) = \begin{cases} fail & \text{if } \ominus\text{-}determined(\sigma, \Pi) \\ unknown & \text{otherwise} \end{cases}$$

Hence, the aim of the test campaign will be to generate sequences σ of Σ^* that negatively determine Π and to play them on the implementation.

6.1.2 Testability of $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ in the *Safety-Progress* classification

For each SP class, we state the conditions under which the properties of this class are testable, i.e., the conditions to exhibit a *fail* verdict.

In the language view, the testability conditions of *r-properties* are given on the finitary properties (ψ or ψ_i for $i \in [1, n]$) over which *r-properties* are built. In the automate view, the testability conditions are given through syntactic criteria on the automata defining *r-properties*.

Theorem 1 (Testability of $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$). *Given $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q^{\mathcal{A}_\Pi}_{\text{init}}, \longrightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ defining an r-property Π built over $\psi \subseteq \Sigma^*$ or $\psi_i \subseteq \Sigma^*$ for $i \in [1, n]$, according to the class of Π, the testability conditions expressed both in the language-theoretic and automata views are given in Table 1*[10].

[10] Intuitively, the set of sequences exposed in Table 1 represents the set of sequences allowing, for each class, to negatively determine the *r-properties*.

$Exec(\mathcal{P}_{\mathbf{\Sigma}}) \subseteq \mathbf{\Pi}$	Testability Condition (language view)	Testability Condition (automata view)
Safety $(A_f(\psi), A(\psi)) \mid R = \emptyset, \overline{P} \nrightarrow P$	$\overline{\psi} \neq \emptyset$	$\overline{P} \neq \emptyset$
Guarantee $(E_f(\psi), E(\psi)) \mid P = \emptyset, R \nrightarrow \overline{R}$	$\{\sigma \in \overline{\psi} \mid pref(\sigma) \cup cont(\sigma) \subseteq \overline{\psi}\} \neq \emptyset$	$\overline{R} \setminus CoReach_{\mathcal{A}_{\Pi}}(R) \neq \emptyset$
Obligation $\bigcap_{i=1}^{k}(S_i(\psi_i) \cup G_i(\psi_i'))$ $\bigcup_{i=1}^{k}(S_i(\psi_i) \cap G_i(\psi_i'))$ $\overline{P_i} \nrightarrow P_i, R_i \nrightarrow \overline{R_i}$	$\bigcup_{i=1}^{k}\left(\overline{\psi_i} \cap \{\sigma \in \overline{\psi_i'} \mid pref(\sigma) \cup cont(\sigma) \subseteq \overline{\psi_i'}\}\right) \neq \emptyset$ $\bigcap_{i=1}^{k}\left(\overline{\psi_i} \cup \{\sigma \in \overline{\psi_i'} \mid pref(\sigma) \cup cont(\sigma) \subseteq \overline{\psi_i'}\}\right) \neq \emptyset$	$\bigcup_{i=1}^{k}\left(\overline{P_i} \cap \overline{R_i} \setminus CoReach_{\mathcal{A}_{\Pi}}(R_i)\right) \neq \emptyset$
Response $(R_f(\psi), R(\psi)) \mid P = \emptyset$	$\{\sigma \in \overline{\psi} \mid cont(\sigma) \subseteq \overline{\psi}\} \neq \emptyset$	$\overline{R} \setminus CoReach_{\mathcal{A}_{\Pi}}(R) \neq \emptyset$
Persistence $(P_f(\psi), P(\psi)) \mid R = \emptyset$	$\{\sigma \in \overline{\psi} \mid cont(\sigma) \subseteq \overline{\psi}\} \neq \emptyset$	$\overline{P} \setminus CoReach_{\mathcal{A}_{\Pi}}(P) \neq \emptyset$

Table 1. Summary of testability results w.r.t. the relation $Exec(\mathcal{P}_{\Sigma}) \subseteq \Pi$ (for a *fail* verdict)

Proof. The complete proof is given in Appendix B.1. The proof is done according to the *Safety-Progress* classes.

- In the language view, for each pair of operators X_f/X with $X \in \{A, E, R, P\}$, one can see that when a sequence is in one of the mentioned sets, the *r-property* $(X_f(\psi), X(\psi))$ is negatively determined: e.g., for a safety *r-property* $(A_f(\psi), A(\psi))$ built upon $\psi \subseteq \Sigma^*$, when $\sigma \in \overline{\psi}$, we have $\ominus\mathit{determined}(\sigma, (A_f(\psi), A(\psi)))$. Indeed, every finite (resp. infinite) continuation of σ cannot belong to $A_f(\psi)$ (resp. $A(\psi)$) because it has at least one prefix not in ψ.
- In the automata view, according to the syntactic restrictions on automata for *Safety-Progress* classes and the acceptance conditions, one can see that when a run of a sequence ends in a state in the mentioned set of states, the corresponding sequence negatively determines the underlying *r-property*[11]. This allows us, according to Property 1, to produce a *fail* verdict. For instance, for safety *r-properties*, when the run of a sequence σ ends in a $\overline{P}$ state, σ negatively determines the underlying *r-property*. Indeed, according to the acceptance criterion of Streett automata, since there is no transition from $\overline{P}$-states to P-states, every finite and infinite continuation of σ does not satisfy the underlying property.

Example 5 (Testability of some r-properties w.r.t. the relation $Exec(\mathcal{P}_{\Sigma}) \subseteq \Pi$). We present the testability of some *r-properties* introduced in Example 3.

The safety *r-property* Π_1 is testable w.r.t. the relation $Exec(\mathcal{P}_{\Sigma_1}) \subseteq \Pi_1$. Indeed in the language view, the property is built on ψ_1 and there are sequences belonging to $\Sigma_1^* \setminus \psi_1$ (the corresponding DFA has a non accepting state). In the automata view, for $\mathcal{A}_{\Pi_1}$, we have $3 \in \overline{P}$ (reachable from the initial state).

The guarantee *r-property* Π_2 is testable w.r.t. the relation $Exec(\mathcal{P}_{\Sigma_2}) \subseteq \Pi_2$. Indeed in the language view, the property is built on ψ_2 and there are sequences belonging to $\Sigma_2^* \setminus \psi_2$ s.t. all prefixes of theses sequences and all its continuations are also in $\Sigma_2^* \setminus \psi_2$. In the automata view, for $\mathcal{A}_{\Pi_2}$, there is a (reachable) state in $\overline{R}$ from which all reachable states are in $\overline{R}$: state 4.

The response *r-property* Π_3 is testable w.r.t. the relation $Exec(\mathcal{P}_{\Sigma_2}) \subseteq \Pi_3$. Indeed in the language view, the property is built on ψ_2 and there are sequences belonging to $\Sigma_2^* \setminus \psi_2$ s.t. all continuations of these sequences belong to $\Sigma_2^* \setminus \psi_2$ as well. In the automata view, for $\mathcal{A}_{\Pi_3}$, there is a (reachable) state in $\overline{R}$ from which all reachable states are in $\overline{R}$: state 5.

The response *r-property* Π_4, defined by the Streett automaton depicted in Fig. 14b, is not testable w.r.t. the relation $Exec(\mathcal{P}_{\Sigma_2}) \subseteq \Pi_4$. Indeed, in the automata view, for $\mathcal{A}_{\Pi_4}$, we have $\overline{R} \setminus CoReach_{\mathcal{A}_{\Pi_4}}(R) = \emptyset$ since $\overline{R} = \{2\}$, $1 \in Reach_{\mathcal{A}_{\Pi_4}}(2)$, and $1 \in R$. In other words, for every finite sequence σ, there exist infinite continuations satisfying Π_4 (visiting infinitely often state 1 in the automaton) and infinite continuations not satisfying Π_4 (staying persistently in state 2 in the automaton).

Example 6 (Testability of the r-properties of the operating system w.r.t. the relation $Exec(\mathcal{P}_{\Sigma}) \subseteq \Pi$). Similarly, we can present the testability of the *r-properties* for the operating system.

- The *r-property* Π^{op} is testable w.r.t. the relation $Exec(\mathcal{P}_{\Sigma}) \subseteq \Pi^{\mathrm{op}}$ where Σ is the alphabet used in the definition of Π^{op}.
- The *r-property* Π_i^{op} is testable w.r.t. the relation $Exec(\mathcal{P}_{\Sigma}) \subseteq \Pi_i^{\mathrm{op}}$, for $i \in [1, 4]$, where Σ is the appropriate alphabet used in the definition of the considered *r-property*.
- The *r-property* Π_5^{op} is not testable w.r.t. the relation $Exec(\mathcal{P}_{\Sigma}) \subseteq \Pi_5^{\mathrm{op}}$, where Σ is the alphabet used in the definition of Π_5^{op}.

6.1.3 Verdicts to deliver

We now state the verdicts that should be produced by a tester for the possibly infinite sequences of the IUT.

[11] One can remark that, for $X \subseteq Q^{\mathcal{A}_{\Pi}}$, $\overline{X} \setminus CoReach_{\mathcal{A}_{\Pi}}(X) \neq \emptyset$ can be equivalently written $\{q \in \overline{X} \mid Reach_{\mathcal{A}_{\Pi}}(q) \subseteq \overline{X}\} \neq \emptyset$.

$Exec(\mathcal{P}_{\mathbf{\Sigma}}) \subseteq \mathbf{\Pi}$	Testability Condition (language view)	Testability Condition (automata view)
Safety $(A_f(\psi), A(\psi)) \mid R = \emptyset, \overline{P} \nrightarrow P$	$\{\sigma \in \psi \mid pref(\sigma) \cup cont(\sigma) \subseteq \psi\} \neq \emptyset$	$P \setminus CoReach_{\mathcal{A}_{\Pi}}(\overline{P}) \neq \emptyset$
Guarantee $(E_f(\psi), E(\psi)) \mid P = \emptyset, R \nrightarrow \overline{R}$	$\psi \neq \emptyset$	$\overline{R} \neq \emptyset$
Obligation $\bigcap_{i=1}^{k}(S_i(\psi_i) \cup G_i(\psi_i'))$ $\bigcup_{i=1}^{k}(S_i(\psi_i) \cap G_i(\psi_i'))$ $\overline{P_i} \nrightarrow P_i, R_i \nrightarrow \overline{R_i}$	$\bigcap_{i=1}^{m} \psi_i' \neq \emptyset$ $\bigcup_{i=1}^{m}(\{\sigma \in \psi_i \mid pref(\sigma) \cup cont(\sigma) \subseteq \psi_i\} \cap \psi_i') \neq \emptyset$	$\bigcap_{i=1}^{m}(P_i \setminus CoReach_{\mathcal{A}_{\Pi}}(\overline{P_i}) \cup R_i) \neq \emptyset$
Response $(R_f(\psi), R(\psi)) \mid P = \emptyset$	$\{\sigma \in \psi \mid pref(\sigma) \cup cont(\sigma) \subseteq \psi\} \neq \emptyset$	$R \setminus CoReach_{\mathcal{A}_{\Pi}}(\overline{R}) \neq \emptyset$
Persistence $(P_f(\psi), P(\psi)) \mid R = \emptyset$	$\{\sigma \in \psi \mid pref(\sigma) \cup cont(\sigma) \subseteq \psi\} \neq \emptyset$	$P \setminus CoReach_{\mathcal{A}_{\Pi}}(\overline{P}) \neq \emptyset$

Table 2. Conditions to produce a *weak pass* verdict for the relation $Exec(\mathcal{P}_{\Sigma}) \subseteq \Pi$

In the language view, each testability condition is expressed as a composition of some ψ_i, where the $\psi_i \subseteq \Sigma^*$ ($i \in [1, n]$) are used to build the *r-property*. When σ belongs to $Exec_{\mathrm{f}}(\mathcal{P}_{\Sigma})$ and the exhibited sets, the test oracle should deliver *fail* since the underlying *r-property* is negatively determined. Conversely, when $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_{\Sigma})$ and σ is not in the exhibited sets, the test oracle can only deliver *unknown*.

In practice, those verdicts are determined by a test oracle, i.e., a computable function reading an interaction sequence. In our framework, test oracles are obtained from Streett automata. We defer the computation of the test oracle to Sect. 7, as we will generate the canonical tester which is a mechanism encompassing the test oracle.

Remark 3. The test oracle can be also obtained from the *r-properties* described in other views (language-theoretic, logical). Indeed, in [13] we describe how to express an *r-property* in the automata view from its expression in the language or the logical view.

In this part, we have clarified and extended some results of [25] about the testability of properties w.r.t. the relation $Exec(\mathcal{P}_{\Sigma}) \subseteq \Pi$. First, we have shown that the safety *r-property* $(\Sigma^*, \Sigma^{\omega})$ always lead to a *pass* verdict and is vacuously testable. Moreover, we exhibited some *r-properties* of other classes which are testable, i.e., some obligation, response, and persistence *r-properties*. The testability conditions are given in the language and automata views. We shall now go one step further in the extension of the results in [25] by introducing a finer notion of verdict.

6.1.4 Refining verdicts

Similarly to the introduction of weak truth values in runtime verification [2,12], it is possible to introduce *weak* verdicts in testing. In this respect, stopping the test and producing a weak verdict consists in stating that the test execution sequence produced so far belongs (or not) to the property. The idea of satisfaction "if the program stops here" in runtime verification [2,12] corresponds to the idea of "the test has shown enough on the implementation" in testing. In this case, testing would be similar to a kind of "active runtime verification": one is interested in the satisfaction of one execution of the program which is steered externally by a tester. Basically, it amounts to not seeing testing as a destructive activity, but as a way to also enhance confidence in the implementation compliance w.r.t. a property.

Under some conditions, it is possible to determine *weak verdicts* for some classes of properties in the following sense: the verdict is expressed on *one single execution sequence* σ, and it does not afford any conclusion on the set $Exec(\mathcal{P}_{\Sigma})$.

We have seen that, for the relation $Exec(\mathcal{P}_{\Sigma}) \subseteq \Pi$, the only verdicts that can be produced were *fail* and *unknown*. Clearly, *fail* verdicts can still be produced. Furthermore, *unknown* verdicts can be refined into *weak pass* verdicts when the sequence σ *positively determines* the *r-property*. In this case, the test can be stopped since whatever is the future behavior of the IUT, it will exhibit behaviors that will satisfy the *r-property*. In this case, it seems reasonable to produce a *weak pass* verdict and consider new test executions in order to gain in confidence.

The definition of the verdict function, as given in Property 1, can be updated:

Property 2 (Sufficient conditions to produce a *fail* or a *weak pass* verdict). Positive determinacy is a sufficient condition for a sequence to be associated to a *weak pass* verdict. Negative determinacy remains a sufficient condition for a sequence to be associated to a *fail* verdict. We have, $verdict(\sigma, Exec(\mathcal{P}_{\Sigma}) \subseteq \Pi) =$

$$\begin{cases} \mathit{fail} & \text{if } \ominus\text{-}\mathit{determined}(\sigma, \Pi), \\ \mathit{weak\ pass} & \text{if } \oplus\text{-}\mathit{determined}(\sigma, \Pi), \\ \mathit{unknown} & \text{otherwise.} \end{cases}$$

We revisit, for each *Safety-Progress* class, the situations when *weak pass* verdicts can be produced for this relation.

Theorem 2 (Producing *weak pass* verdicts for the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$). *For each Safety-Progress class, the situations when* weak pass *verdicts can be produced are given in Table 2.*

Proof. Noticing that the notions of positive and negative determinacy are dual, the proof can be conducted similarly to the proof of Theorem 1. □

Corollary 1 (Testabity of $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ with weak verdict). *Testability conditions for $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ augmented with weak verdicts are the disjunction of the conditions exposed in Tables 1 and 2.*

Proof. It is a direct consequence of Theorems 1 and 2. □

Definition 11 (Test Oracle with weak verdict). The notion of test oracle introduced in Definition 9 can be easily extended with the notion of *weak pass* verdict. Indeed, it suffices to require the following additional constraints relatively to the production of verdicts:

– soundness: $\forall q \in Q^{\mathcal{O}}$:

$$\begin{array}{l} \Gamma^{\mathcal{O}}(q) = \textit{weak pass} \\ \Rightarrow \forall \sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) : q^{\mathcal{O}}_{\mathrm{init}} \xrightarrow{\sigma}_{\mathcal{O}} q \Rightarrow \Pi(\sigma) \\ \quad \wedge \forall \sigma' \in \Sigma^* : \sigma \prec \sigma' \\ \quad\quad \Rightarrow \textit{verdict}(\sigma', Exec(\mathcal{P}_\Sigma) \subseteq \Pi) \notin \{\textit{fail}, \textit{unknown}\}, \end{array}$$

– completeness: $\forall \sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \cap \Pi$:

$$\begin{array}{l} \big(\Pi(\sigma) \wedge \forall \sigma' \in \Sigma^* : \sigma \prec \sigma' \\ \quad\quad \Rightarrow \textit{verdict}(\sigma', Exec(\mathcal{P}_\Sigma) \subseteq \Pi) \notin \{\textit{fail}, \textit{unknown}\}\big) \\ \Rightarrow \exists q \in Q^{\mathcal{O}} : q^{\mathcal{O}}_{\mathrm{init}} \xrightarrow{\sigma}_{\mathcal{O}} q \wedge \Gamma^{\mathcal{O}}(q) = \textit{weak pass}. \end{array}$$

Soundness (resp. completeness) entails a test oracle to produce a *weak pass* verdict for a sequence σ only when (resp. as soon as) Π is satisfied by σ and no future continuation of σ can lead to a *fail* or *unknown* verdict.

6.2 Testability w.r.t. the relation $Exec(\mathcal{P}_\Sigma) = \Pi$

The previous reasoning applies for the testability w.r.t. the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ apply in a similar fashion. The characterization of testable *r-properties* is thus the same. Indeed, when one finds a sequence $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$ s.t. it is possible to find a *fail* verdict for $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$, then this same verdict holds for $Exec(\mathcal{P}_\Sigma) = \Pi$, i.e., $Exec(\mathcal{P}_\Sigma) \not\subseteq \Pi \Rightarrow Exec(\mathcal{P}_\Sigma) \neq \Pi$. Note that the same reasoning applies for the *weak pass* verdict.

6.3 Testability w.r.t. the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$

Testability results for this relation can be determined using:

– the results stated for the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$,
– the duality within the *Safety-Progress* classification.

Indeed, for safety and guarantee *r-properties* (similar duality holds for response and persistence):

$$\Pi = (A_f(\psi), A(\psi)) \Rightarrow \overline{\Pi} = (E_f(\overline{\psi}), E(\overline{\psi})).$$

Furthermore, one has to notice that $\neg(Exec(\mathcal{P}_\Sigma) \subseteq \Pi) \Leftrightarrow Exec(\mathcal{P}_\Sigma) \cap \overline{\Pi} \neq \emptyset$. Consequently, testability results can be obtained in a rather straightforward manner for this relation. Indeed, the testability conditions to obtain a *pass* verdict for the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ are the same conditions to obtain a *weak pass* verdict for the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ (expressed in Theorem 2). Moreover, we can show that there exists one guarantee *r-property* for which it is not possible to obtain a *pass* verdict. Indeed, by duality with the non-testability of $(\Sigma^*, \Sigma^\omega)$ w.r.t. $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$, testing the guarantee *r-property* $(\emptyset, \emptyset)$ cannot lead to a *pass* verdict. Furthermore, we have shown that some *r-properties* of the other classes are testable[12] as well, i.e., some safety, obligation, response, and persistence *r-properties*. Finally, we provided testability conditions in the language and automata views. Thus, we have extended and clarified some results of [25].

Remark 4 (Producing a weak fail verdict for $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$). By duality, and following the reasoning used to motivate the *weak pass* verdict for the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$, it is possible to produce a *weak fail* verdict for the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$. Thus, the conditions to produce a *weak fail* verdict are the ones stated in Table 1. Thus the testability conditions for the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ augmented with the *weak fail* verdict are similarly the disjunction of the conditions expressed in Tables 1 and 2.

Example 7 (Testability of some r-properties w.r.t. the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$). We present the testability of three *r-properties* introduced in Example 3 w.r.t. $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$.

The safety *r-property* Π_1 built from ψ_1, defined by the Streett automaton depicted in Fig. 4a, is not testable w.r.t. the relation $Exec(\mathcal{P}_{\Sigma_1}) \cap \Pi_1 \neq \emptyset$. Indeed, it does not satisfy the testability condition: the automaton defining ψ_1 does not have an accepting state reachable from the initial state s.t. it is reachable only with accepting states and s.t. all reachable states are accepting ($\{\sigma \in \psi_1 \mid pref(\sigma) \cup cont(\sigma) \subseteq \psi_1\} = \emptyset$). However, this property is testable with weak verdicts. Indeed, because of state 3 in $\mathcal{A}_{\Pi_1}$, it is possible to obtain *weak fail* verdicts.

The guarantee *r-property* Π_2 built upon ψ_2, defined by the Streett automaton in Fig. 4b, is testable w.r.t. the relation $Exec(\mathcal{P}_{\Sigma_2}) \cap \Pi_2 \neq \emptyset$. Indeed, it satisfies the testability conditions for guarantee properties: the automaton defining ψ_2 has a (reachable) accepting state

[12] In [25], for this relation, only guarantee properties are declared as testable.

($\psi_2 \neq \emptyset$) and $R \neq \emptyset$ in $\mathcal{A}_{\Pi_2}$. The interesting sequences to be played in order to obtain a *pass* verdict are those leading to state 3. Moreover, it is also possible to produce *weak fail* verdicts because of state 4 in $\mathcal{A}_{\Pi_2}$.

The response *r-property* Π_3 built upon ψ_2, defined by the Streett automaton in Fig. 4c is not testable w.r.t. the relation $Exec(\mathcal{P}_{\Sigma_2}) \cap \Pi_3 \neq \emptyset$. Similarly, it does not satisfy the testability conditions for response properties.

Similarly, following the previous reasoning:

- The *r-property* Π^{op} (resp. Π_2^{op}) is testable w.r.t. the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi^{\text{op}} \neq \emptyset$ (resp. $Exec(\mathcal{P}_\Sigma) \cap \Pi_2^{\text{op}} \neq \emptyset$) where Σ is the appropriate alphabet used in the definition of the considered *r-property*.
- The *r-property* Π_i^{op} is not testable w.r.t. the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi_i^{\text{op}} \neq \emptyset$, $i \in [1, 3, 4, 5]$, where Σ is the appropriate alphabet used in the definition of the considered *r-property*.

6.4 Testability w.r.t. the relation $\Pi \subseteq Exec(\mathcal{P}_\Sigma)$

It is not possible to obtain verdicts for this relation in the general case. We explicit this below.

In order to obtain a *pass* verdict for this relation, it would require to prove that all execution sequences described by the property are sequences of the program. This is impossible as soon as the set of sequences described by the *r-property* is infinite.

In order to obtain a *fail* verdict, it would require to prove that at least one sequence described by the *r-property* cannot be played on the implementation. Even if one finds an execution sequence of the implementation not satisfying the *r-property*, it does not afford to state that the relation does not hold. Indeed, since the IUT may be non deterministic, another execution of the implementation could exhibit such a sequence. Producing a *fail* verdict would require a determinism hypothesis on the implementation.

6.5 Summary

In this section, the conditions for the testability of properties w.r.t. four relations of interest have been stated. Moreover, the use of a weaker notion of verdict has been motivated and conditions to produce them have been provided.

7 Automatic test generation w.r.t. the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$

In this section, we address test generation for the testing framework introduced in this paper. Here, test generation is based on *r-properties*, and the purpose of the test campaign is to determine verdicts for a relation between a testable *r-property* and an IUT. Before entering into the details, we first discuss informally some practical constraints that have to be taken into account for test generation. Then, we will be able to compute the canonical tester (i.e., the most general tester for the relation) (Sect. 7.1), discuss test selection (Sect. 7.2). We focus on test generation w.r.t. the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$. Automatic test generation with respect to the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ can be simply derived by duality.

Which sequences should be played? The sequences of interest to play on the IUT are naturally those leading to a *fail* or a *weak pass* verdict and these sequences can be used to generate test cases. In the language view (resp. automata view), these sequences are those belonging to the exhibited sets (resp. leading to the exhibited set of states) in testability conditions given in Tables 1 and 2. For instance, for a safety *r-property* $\Pi_S = (A_f(\psi), A(\psi))$ built upon ψ, and defined by a safety automaton $\mathcal{A}_{\Pi_S}$, one should play sequences in $\overline{\psi}$ or equivalently those leading to $\overline{P}$ in $\mathcal{A}_{\Pi_S}$.

When to stop the test? When the tested program produces an execution sequence $\sigma \in \Sigma^*$, a raised question is when to safely stop the test. Obviously, a first answer is when a *fail* or *weak pass* verdict has been issued since these verdicts are definitive. Although in other cases, when the test interactions produced some test sequences leading so far to *unknown* evaluations, the question prevails. It remains to the tester appraisal to decide when the test should be stopped (see Sect. 7.2).

Alphabet and test architecture. In order to address test generation, we will need to distinguish inputs and outputs and the alphabet of the IUT and the *r-property*. The alphabet Σ of the property is now partitioned into $\Sigma_?$ (input actions) and $\Sigma_!$ (output actions). The alphabet of the IUT becomes Σ^{IUT} and is partitioned into $\Sigma^{IUT}_?$ (input actions) and $\Sigma^{IUT}_!$ (output actions) with $\Sigma_? = \Sigma^{IUT}_?$ and $\Sigma_! = \Sigma^{IUT}_!$. As usual, we also suppose that the behavior of the IUT can be modeled by an IOLTS $\mathcal{I} = (Q^{\mathcal{I}}, q^{\mathcal{I}}_{\text{init}}, \Sigma^{IUT}, \longrightarrow_{\mathcal{I}})$. We do not require the IUT to be input-complete. If the IUT refuses an input, the test execution terminates and the associated verdict is the last produced one (before trying to emit the input), i.e., an *unknown* verdict. We do not assume neither the IUT to be deterministic.

7.1 Computation of the canonical tester

We propose a methodology to build the canonical tester for our framework. The canonical tester for a relation $\mathcal{R}$ between an IUT $\mathcal{P}_\Sigma$ and an *r-property* Π is purposed to detect all verdicts for the relation between the *r-property* and all possible test executions that can be produced with $\mathcal{P}_\Sigma$.

We define canonical testers from Streett automata. To do so, we will reuse a partition of the set of states

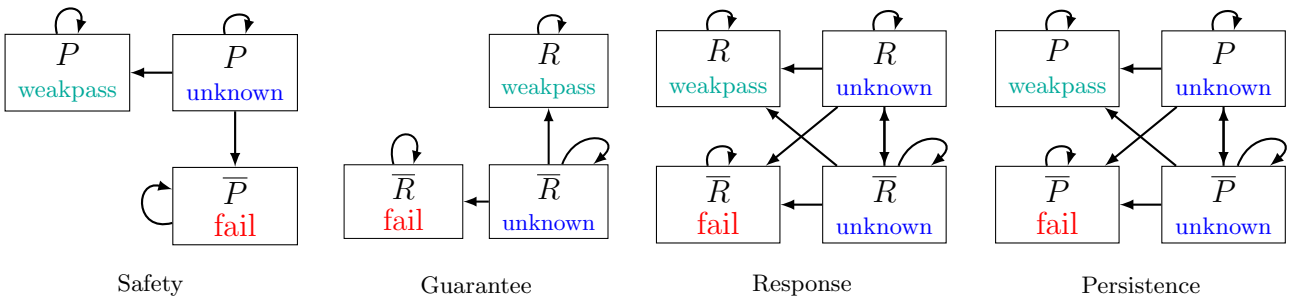


Fig. 9. Schematic illustrations of the canonical tester for basic classes

of a Streett automaton that was introduced in [12] for runtime verification.

Definition 12 (Good and bad states). For a Streett automaton $\mathcal{A}_{\Pi} = (Q^{\mathcal{A}_{\Pi}}, q_{\text{init}}^{\mathcal{A}_{\Pi}}, \Sigma, \longrightarrow_{\mathcal{A}_{\Pi}}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$, the sets $G^{\mathcal{A}_{\Pi}}, G_c^{\mathcal{A}_{\Pi}}, B_c^{\mathcal{A}_{\Pi}}, B^{\mathcal{A}_{\Pi}}$ form a partition of $Q^{\mathcal{A}_{\Pi}}$ and designate respectively the good (resp. currently good, currently bad, bad) states:

- $G^{\mathcal{A}_{\Pi}} \stackrel{\text{def}}{=} \{q \in Q^{\mathcal{A}_{\Pi}} \cap \bigcap_{i=1}^{m}(R_i \cup P_i) \mid Reach_{\mathcal{A}_{\Pi}}(q) \subseteq \bigcap_{i=1}^{m}(R_i \cup P_i)\}$;
- $G_c^{\mathcal{A}_{\Pi}} \stackrel{\text{def}}{=} \{q \in Q^{\mathcal{A}_{\Pi}} \cap \bigcap_{i=1}^{m}(R_i \cup P_i) \mid Reach_{\mathcal{A}_{\Pi}}(q) \not\subseteq \bigcap_{i=1}^{m}(R_i \cup P_i)\}$;
- $B_c^{\mathcal{A}_{\Pi}} \stackrel{\text{def}}{=} \{q \in Q^{\mathcal{A}_{\Pi}} \cap \bigcup_{i=1}^{m}(\overline{R_i} \cap \overline{P_i}) \mid Reach_{\mathcal{A}_{\Pi}}(q) \not\subseteq \bigcup_{i=1}^{m}(\overline{R_i} \cap \overline{P_i})\}$;
- $B^{\mathcal{A}_{\Pi}} \stackrel{\text{def}}{=} \{q \in Q^{\mathcal{A}_{\Pi}} \cap \bigcup_{i=1}^{m}(\overline{R_i} \cap \overline{P_i}) \mid Reach_{\mathcal{A}_{\Pi}}(q) \subseteq \bigcup_{i=1}^{m}(\overline{R_i} \cap \overline{P_i})\}$.

Informally, for an *r-property* Π defined by $\mathcal{A}_{\Pi}$, a state $q \in Q^{\mathcal{A}_{\Pi}}$ is:

- in $G^{\mathcal{A}_{\Pi}}$ (good states) if and only if it is an accepting state and all reachable states are accepting;
- in $G_c^{\mathcal{A}_{\Pi}}$ (currently good states) if and only if it is an accepting state and there is at least one reachable non-accepting state;
- in $B_c^{\mathcal{A}_{\Pi}}$ (currently bad states) if and only if it is a non-accepting state and there is at least one reachable accepting state;
- in $B^{\mathcal{A}_{\Pi}}$ (bad states) if and only if it is a non-accepting state and all reachable states are not accepting.

It is possible to show that if a sequence σ reaches a state in $B^{\mathcal{A}_{\Pi}}$ (resp. $G^{\mathcal{A}_{\Pi}}$), then the underlying property Π is negatively (resp. positively) determined by σ.

Lemma 1 (Good and Bad states vs determinacy). *Given $\sigma \in \Sigma^*$, a Streett automaton $\mathcal{A}_{\Pi}$ defining an r-property Π, and $q \in Q^{\mathcal{A}_{\Pi}}$, we have:*

$$q \in G^{\mathcal{A}_{\Pi}} \Leftrightarrow \forall \sigma \in \Sigma^* : q_{\text{init}}^{\mathcal{A}_{\Pi}} \xrightarrow{\sigma}_{\mathcal{A}_{\Pi}} q \Rightarrow \oplus\text{–}determined(\sigma, \Pi);$$

$$q \in B^{\mathcal{A}_{\Pi}} \Leftrightarrow \forall \sigma \in \Sigma^* : q_{\text{init}}^{\mathcal{A}_{\Pi}} \xrightarrow{\sigma}_{\mathcal{A}_{\Pi}} q \Rightarrow \ominus\text{–}determined(\sigma, \Pi).$$

Proof. The lemma has been proved in [13] in a different context, we propose a proof in Appendix B.2 for the sake of completeness. The proof uses the acceptance conditions of Streett automata, and the fact that we use complete and deterministic automata.

The canonical tester is defined as follows. One can note that the notion of canonical tester is a generalization of the notion of test oracle introduced in Definition 9.

Definition 13 (Canonical Tester). Given a Streett m-automaton $\mathcal{A}_{\Pi} = (Q^{\mathcal{A}_{\Pi}}, q_{\text{init}}^{\mathcal{A}_{\Pi}}, \longrightarrow_{\mathcal{A}_{\Pi}}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ defining a *testable r-property* Π, the associated canonical tester is the Moore automaton $T = (Q^T, q_{\text{init}}^T, \Sigma, \longrightarrow_T, \Gamma^T)$ where $\Gamma^T : Q^T \to \{fail, weak\ pass, unknown\}$ is the output function producing verdicts and is defined as follows:

- $Q^T = Q^{\mathcal{A}_{\Pi}}$;
- $q_{\text{init}}^T = q_{\text{init}}^{\mathcal{A}_{\Pi}}$;
- $\longrightarrow_T = \longrightarrow_{\mathcal{A}_{\Pi}}$;
- for $q \in Q^T$,

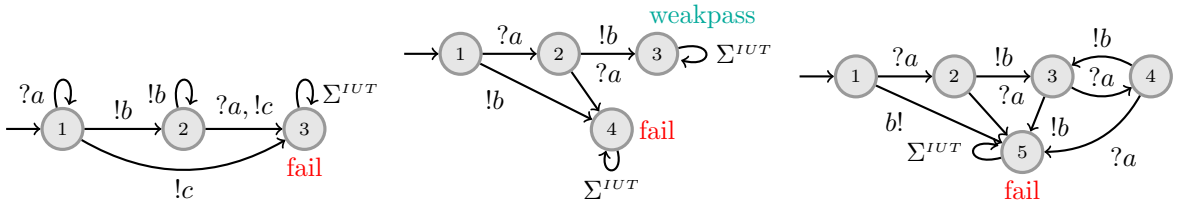
$$\Gamma^T(q) = \begin{cases} unknown & \text{when } q \in B_c^{\mathcal{A}_{\Pi}} \cup G_c^{\mathcal{A}_{\Pi}}, \\ fail & \text{when } q \in B^{\mathcal{A}_{\Pi}}, \\ weak\ pass & \text{when } q \in G^{\mathcal{A}_{\Pi}}. \end{cases}$$

A Streett automaton is simply transformed into a canonical tester by defining verdicts for its states according to the partition of bad states (in $B^{\mathcal{A}_{\Pi}}$) and good states (in $G^{\mathcal{A}_{\Pi}}$). They are assigned respectively the *fail* and *weak pass* verdicts, while all other states are given the *unknown* verdict. Note that the test can be stopped when reaching bad or good states. This construction of canonical testers is illustrated in Fig. 9 for the automata dedicated to basic classes of properties (see Fig. 6).

Theorem 3 (Soundness and completeness of canonical testers). *The canonical tester as defined in Definition 13 is a sound and complete test oracle augmented with weak verdicts as proposed in Definition 11.*

Proof. It is a direct consequence of Lemma 1. □

Example 8 (Canonical Testers). Canonical testers for *r-properties* of Example 3 are represented in Fig. 10. The alphabet partitioning is s.t. $\Sigma_?^{IUT} = \{?a\}$ and $\Sigma_!^{IUT} = \{!b, !c\}$. Assignments to *unknown* are omitted for readability.



(a) Canonical tester for Π_1 (b) Canonical tester for Π_2 (c) Canonical tester for Π_3

Fig. 10. Canonical testers built from the *r-properties* of Example 3

The canonical tester built from $\mathcal{A}_{\Pi_1}$, the Streett safety automaton defining Π_1, is s.t. the state 3 (a bad state) is assigned the *fail* verdict.

The canonical tester built from $\mathcal{A}_{\Pi_2}$, the Streett guarantee automaton defining Π_2, is s.t. the state 5 (a bad state) is assigned the *fail* verdict and state 3 (a good state) is assigned the *weak pass* verdict.

The canonical tester built from $\mathcal{A}_{\Pi_3}$, the Streett response automaton defining Π_3 is s.t. the state 5 (a bad state) is assigned the *fail* verdict.

Example 9 (Canonical Testers). The canonical tester for the global specification of the operating system defined by the Streett automaton in Fig. 5 is depicted in Fig. 11. The alphabet partitioning is $\Sigma_?^{IUT} = \{$*?maintain,?req-auth, ?op-sec,?op-uns,?logout*$\}$ and $\Sigma_!^{IUT} = \{$*!uncrypt,!crypt, !deny,!grant*$\}$. Assignments to *unknown* are omitted for readability.

Similarly, the canonical testers for the *r-properties* of the operating system are depicted in Fig 12.

7.2 Test selection

For a given *r-property*, the set of potential test sequences to be played is potentially infinite. The purpose of *test selection* is to produce a (finite) set of test cases s.t. any test case is *controllable* [18] (it has no choice between the inputs it provides to the IUT) and should produce a sound verdict. Note that a test selection algorithm usually targets a particular verdict, that could be either *fail* or *weak pass* in our context. In this case, the Streett automaton describing the property under scrutiny allows to prune the execution sequences that not do lead to the targeted verdict.

A generic test selection algorithm can be viewed as a (non-deterministic) transformation of the canonical tester $T = (Q^T, q^T_{\text{init}}, \Sigma, \longrightarrow_T, \Gamma^T)$ to produce a test case as a controllable IOLTS $TC = (Q^{TC}, q^{TC}_{\text{init}}, \Sigma, \longrightarrow_{TC}, \Gamma^{TC})$ such that each execution sequence of TC corresponds to a test execution. States of TC are some states of T. An additional verdict called *inconc* (inconclusive) replaces the *unknown* verdict in states from which the targeted verdict cannot be reached anymore. Now, the set of possible verdicts becomes $\{$*fail*, *pass*, *inconc*, *unknown*$\}$. This algorithm is informally described below (assuming that the target verdict is *fail*, thus $B^{\mathcal{A}_\Pi}$ states):

1. Define $Inconc \stackrel{\text{def}}{=} Q^T \setminus CoReach_T(B^{\mathcal{A}_\Pi})$ as the set of states from which the *fail* verdict is not reachable.
2. Define the output function of TC such that $\forall q \in Q^T \setminus Inconc : \Gamma^{TC}(q) = \Gamma^T(q)$ and $\forall q \in Inconc : \Gamma^{TC}(q) = inconc$.
3. Remove in $\longrightarrow_{TC}$ transitions from $\longrightarrow_T$ labelled by an input action of the IUT and whose target state is in *Inconc*: $\forall e \in \Sigma_?^{IUT}, \forall q, q' \in Q^{TC}$:

$$(q \xrightarrow{e}_T q' \wedge q' \in Inconc) \Rightarrow (q, e, q') \notin \longrightarrow_{TC}$$

   (while transitions of $\longrightarrow_T$ labelled by an output action of the IUT and whose target state is in *Inconc* are kept in $\longrightarrow_{TC}$).
4. Make TC controllable, i.e., ensure that it satisfies: $\forall e_1, e_2 \in \Sigma_?^{IUT}, \forall q, q_1, q_2 \in Q^{TC}$:

$$(q \xrightarrow{e_1}_{TC} q_1 \wedge q \xrightarrow{e_2}_{TC} q_2) \Rightarrow (q_1 = q_2 \wedge e_1 = e_2).$$

Roughly speaking, based on items 1. and 2., the test selection allows to stop the test and deliver an *inconc* verdict whenever it is no more possible for the tester to produce a sequence that negatively or positively determines the underlying property. Item 3. says that *Inconc* states are reached only after outputs, because the tester control inputs it provides and thus may avoid falling in *Inconc* with inputs.

Example 10 (Test sequences). Applying the test selection algorithm to the various examples we proposed is straightforward. For instance, for $\mathcal{A}_{\Pi_1^{\text{op}}}$, from the controllable canonical tester, possible test sequences (that can serve as a basis for test cases) include the sequences in $(?op\text{-}uns \cdot !uncrypt)^* \cdot ?op\text{-}uns \cdot (!crypt + ?op\text{-}sec)$. More generally, it consists in generating sequences ending in a verdict state on the controllable canonical tester.

Test selection plays also a particular role to state *weak pass* verdicts. Indeed, when dealing with sequences satisfying an *r-property so far* and not positively determining it, test selection should plan the moment for stopping the test. It can be, for instance, when the test lasted more than a given expected duration or when the number of interactions with the IUT reaches a given bound. However, one should not forget that there might exist a continuation, that can be produced by letting the test execution continue, not satisfying the *r-property* or even

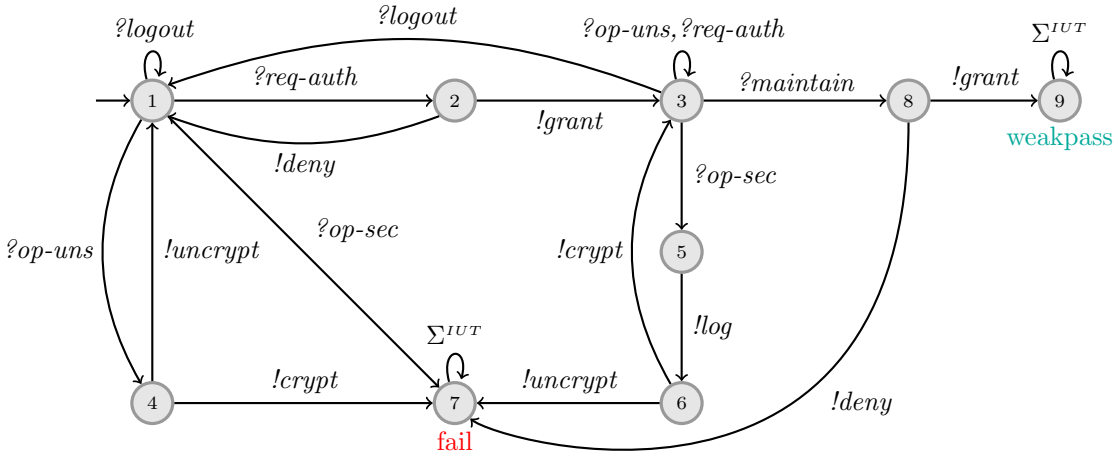


Fig. 11. Canonical tester for the global specification of the operating system

negatively determining it. Here, it thus remains to the tester expertise to state the halting criterion (possibly using quiescence, see Sect. 8).

Remark 5 (More advanced test selection). In practice, one may use the underlying Streett automaton to refine the test selection algorithm by, for instance, further constraining the states that should be visited during a test.

8 Introducing quiescence

We show in this section how the test generation technique proposed in Sect. 7 can be improved when some notion of quiescence (i.e., an explicit lack of response from the IUT) can be taken into account.

8.1 The notion of quiescence in our framework

Quiescence [32,18] was introduced in conformance testing in order to represent IUT's inactivity. In practice, several kinds of quiescence may happen (see [18] for instance). Here we distinguish two kinds of quiescence. Outputlocks (denoted δ_o) represent the situations where the IUT is waiting for an input and produces no outputs. Deadlocks (denoted δ_d) represent the situations where the IUT cannot interact anymore, e.g., its execution is terminated or it is deadlocked. In practice, the observation and the distinction of these two kinds of quiescence require the following hypothesis on the test architecture. First, we suppose the existence of a timer indicating, when not expired, that the IUT may still produce an output. Second, we suppose to be able to observe whether or not the IUT is ready to receive an input. This latter hypothesis boils down to the observation of this Boolean information on the IUT. Then, if the timer is expired and the IUT is not ready to receive an input, this means that the IUT is in deadlock or has finished its execution.

Thus, we introduce those two events in the output alphabet of the IUT. We have now the following additional alphabets: $\Sigma^{IUT}_{!,\delta} \stackrel{\text{def}}{=} \Sigma^{IUT}_{!} \cup \{\delta_o, \delta_d\}$, $\Sigma^{IUT}_{\delta} \stackrel{\text{def}}{=} \Sigma^{IUT}_{!,\delta} \cup \Sigma^{IUT}_{?}$.

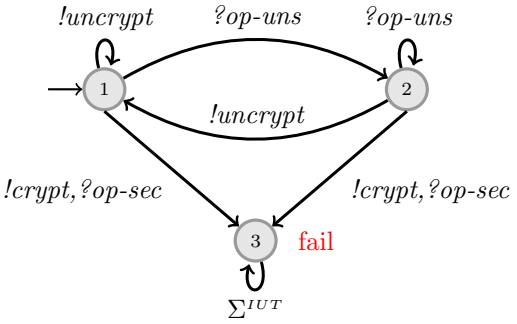
We also have to distinguish the set of traces of the IUT from the set of potential interactions with the IUT. This latest is based on the observable behavior of the IUT and potential choices of the tester. The set of executions of the IUT is now $Exec(\mathcal{P}_{\Sigma^{IUT}}) \subseteq (\Sigma^{IUT}_{\delta})^{\infty}$. However, the set of possible interactions of the tester with the IUT is

$$Inter(\mathcal{P}_{\Sigma^{IUT}}) \stackrel{\text{def}}{=} (\Sigma^{IUT} + \delta_o)^* \cdot (\delta_d + \epsilon),$$

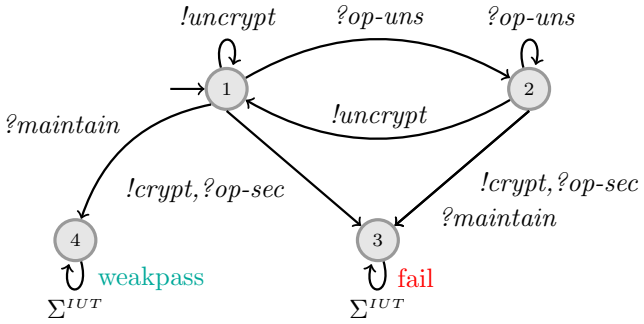
i.e., the interactions of the tester with IUT are s.t. the tester can observe IUT's outputlocks and may be ended by the observation of an outputlock, a deadlock, or program termination. When considering quiescence, characterizing testable properties now consists in comparing the set of interactions with the set of sequences described by the *r-property*. The intuitive ideas are the followings:

- The tester can observe finished executions of the IUT with δ_d. In this case, the IUT has run a finite execution. In some sense, the played sequence determines negatively or positively the *r-property* depending on whether or not it satisfied the *r-property*.
- The tester can decide to terminate its interaction with the program when observing an outputlock. When the tester played a sequence s.t. the underlying *r-property* is not satisfied and observes an outputlock, the played sequence negatively determines the *r-property*. Indeed, with no further action of the tester, the IUT is blocked in a state in which the underlying *r-property* is not satisfied.

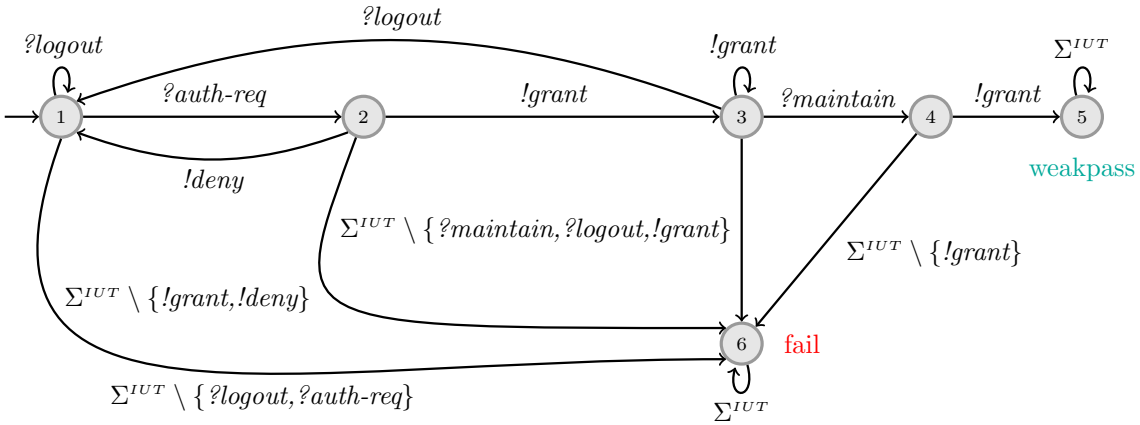
Equivalently, in a more intuitive way, quiescence is forbidden in currently bad states and allowed in currently good states. The notion of negative determinacy is now modified in the context of quiescence as follows.



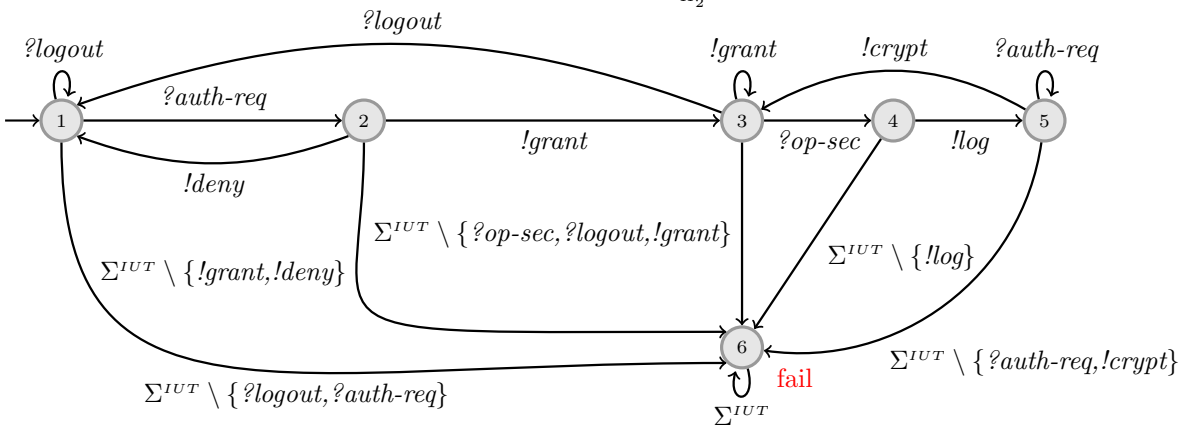
(a) Canonical tester for $\mathcal{A}_{\Pi_1^{\mathrm{op}}}$



(b) Canonical tester for $\mathcal{A}_{\Pi_3^{\mathrm{op}}}$



(c) Canonical tester for $\mathcal{A}_{\Pi_2^{\mathrm{op}}}$



(d) Canonical tester for $\mathcal{A}_{\Pi_4^{\mathrm{op}}}$

Fig. 12. Canonical tester for some *r-properties* of the operating system

$Inter(\mathcal{P}_{\mathbf{\Sigma^{IUT}}}) \subseteq \mathbf{\Pi}$	Testability Condition on the property
Safety $(A_f(\psi), A(\psi))$	$\overline{\psi} \neq \emptyset$
Guarantee $(E_f(\psi), E(\psi))$	$\{\sigma \in \overline{\psi} \mid pref(\sigma) \subseteq \overline{\psi}\} \neq \emptyset$
Obligation $\bigcap_{i=1}^{k}(S_i(\psi_i) \cup G_i(\psi_i'))$ $\bigcup_{i=1}^{k}(S_i(\psi_i) \cap G_i(\psi_i'))$	$\bigcup_{i=1}^{k}\left(\overline{\psi_i} \cap \{\sigma \in \overline{\psi_i'} \mid pref(\sigma) \subseteq \overline{\psi_i'}\right) \neq \emptyset$ $\bigcap_{i=1}^{k}\left(\overline{\psi_i} \cup \{\sigma \in \overline{\psi_i'} \mid pref(\sigma) \subseteq \overline{\psi_i'}\right\} \neq \emptyset$
Response $(R_f(\psi), R(\psi))$	$\overline{\psi} \neq \emptyset$
Persistence $(P_f(\psi), P(\psi))$	$\overline{\psi} \neq \emptyset$

Table 3. Testability w.r.t. $Inter(\mathcal{P}_{\Sigma^{IUT}}) \subseteq \Pi$ with quiescence

Definition 14 (Determinacy with quiescence). An *r-property* Π defined over Σ is said to be negatively determined upon quiescence by $\sigma \in Inter(\mathcal{P}_{\Sigma^{IUT}})$ (denoted $\ominus$*–determined–*$q(\sigma, \Pi)$) if

$$\ominus\text{–}determined(\sigma_{\downarrow_\Pi}, \Pi) \\ \vee (|\sigma| > 1 \wedge \sigma_{|\sigma|-1} \in \{\delta_d, \delta_o\} \wedge \neg \Pi(\sigma_{\downarrow_\Pi}))$$

where $\sigma_{\downarrow_\Pi}$ is the projection of σ on the vocabulary of Π, i.e., with δ_o and δ_d erased. The definition of positive determinacy is modified in the same way.

For the proposed approach, the usefulness of quiescence lies in the fact that the current test sequence does not have any continuation. Consequently, testability conditions may be weakened. Indeed, when one has determined that the current interaction with the IUT is over, it is not necessary to evaluate the satisfaction of the *r-property* anymore. In some sense, it amounts to consider that the evaluation produced by the last event before observing quiescence "terminates" the execution sequence (and there is no continuation). For instance, if the *r-property* is not satisfied after the last interaction, then the *r-property* is negatively determined by it.

8.2 Revisiting previous results for the inclusion relation

With quiescence, the purpose of the tester is now to "drive" the IUT in a state in which the underlying *r-property* is not satisfied, and then potentially observe quiescence, i.e., find a sequence that negatively determines the property upon quiescence. Informally, the testability condition relies now on the existence of a sequence s.t. the *r-property* is not satisfied. Testability results, upon the observation of quiescence and in order to produce *fail* verdicts when the tested *r-property* is not satisfied, are updated using the notion of negative determinacy with quiescence as shown in Table 3. From these testability conditions, and using the definition of negative determinacy with quiescence, we can deduce the sequences that should be played in order to obtain a *fail* verdict. Two kinds of sequences may lead to a *fail* verdict:

- the first ones, when projected on the vocabulary of the property, negatively determine it (as previously);
- the second ones are such that the prefixes of these sequences containing all events but the last one do not satisfy the property and the last event is either an outputlock or a deadlock.

It is then rather easy to give a characterization of these sequences in the language view. For instance:

- for safety *r-properties*: $\{\sigma \in Inter \mid \sigma_{\downarrow_\Pi} \in \overline{\psi}\}$,
- for guarantee *r-properties*:

$$\{\sigma \in Inter \mid \sigma_{\downarrow_\Pi} \in \overline{\psi} \wedge pref(\sigma_{\downarrow_\Pi}) \cup cont(\sigma_{\downarrow_\Pi}) \subseteq \overline{\psi}\} \\ \cup \{\sigma \cdot (\delta_o + \delta_d) \in Inter \mid \sigma_{\downarrow_\Pi} \in \overline{\psi} \wedge pref(\sigma_{\downarrow_\Pi}) \subseteq \overline{\psi}\},$$

- for response and persistence *r-properties*:

$$\{\sigma \in Inter \mid \sigma_{\downarrow_\Pi} \in \overline{\psi} \wedge cont(\sigma_{\downarrow_\Pi}) \subseteq \overline{\psi}\} \\ \cup \{\sigma \cdot (\delta_o + \delta_d) \in Inter \mid \sigma_{\downarrow_\Pi} \in \overline{\psi}\}.$$

where $Inter(\mathcal{P}_{\Sigma^{IUT}})$ is denoted *Inter* for the sake of readability. These sequences are characterized in an easier way on the canonical tester.

Now it is possible to adapt the construction of the canonical tester so as to take quiescence into consideration:

Definition 15 (Canonical Tester with quiescence). The definition of the canonical tester construction proposed in Definition 13 is updated as follows:

- Two new states q_{fail} and q_{wpass} are added to Q^T such that $\Gamma^T(q_{fail}) = fail$ and $\Gamma^T(q_{wpass}) = weak\ pass$,
- The following transitions are added to $\longrightarrow_T$:
  - $\forall q \in B_c^{\mathcal{A}_\Pi} : q \xrightarrow{\delta_o}_T q_{fail} \wedge q \xrightarrow{\delta_d}_T q_{fail}$,
  - $\forall q \in G_c^{\mathcal{A}_\Pi} : q \xrightarrow{\delta_o}_T q \wedge q \xrightarrow{\delta_d}_T q_{wpass}$,
  - $\forall e \in \Sigma^{IUT} : q_{fail} \xrightarrow{e}_T q_{fail} \wedge q_{wpass} \xrightarrow{e}_T q_{wpass}$.

In a currently bad state, the observation of quiescence produces a *fail* verdict since the underlying property is negatively determined upon quiescence. In other words, the tester can legitimately stop the interaction and let the IUT in a state in which the underlying property is not satisfied. In a currently good state, the observation of a deadlock (or program termination) produces a *weak pass* verdict since the underlying property is positively determined upon quiescence: the current interaction sequence is satisfying the underlying property and there will be no possible continuation of this interaction. However, in a currently good state, when observing an outputlock, the canonical tester stays in the same state. Indeed, stopping the tester would be a decision of the tester and not one of the implementation.
Illustrations of the construction of the canonical tester for basic classes with quiescence are given in Fig. 13, where the original (resp. modified) transitions from the Streett automaton are in plain (resp. dotted) lines.

Example 11 (Testability with quiescence). We illustrate the usefulness of quiescence by showing how using quiescence makes some properties testable, although they were not testable initially.

Consider the IUT depicted in Fig. 14a with observable actions $\Sigma_?^{IUT} = \{?a\}$ and $\Sigma_!^{IUT} = \{!b\}$. This IUT waits for an $?a$, produces a $!b$, and then non deterministically terminates or waits for an $?a$, and repeats the behavior consisting in receiving an $?a$ and producing a $!b$. The executions and possible interactions with the tester are given in Fig. 15 ("?" and "!" are not represented and $x^{\&}$ stands for $x + \epsilon$).

Now let us consider the *r-property* Π_4 defined by the Streett automaton $\mathcal{A}_{\Pi_4}$ depicted in Fig. 14b. Its vocabulary is $\{?a, !b\}$, and it has one recurrent state: $R = \{1\}$. The underlying *r-property* states that every input $?a$

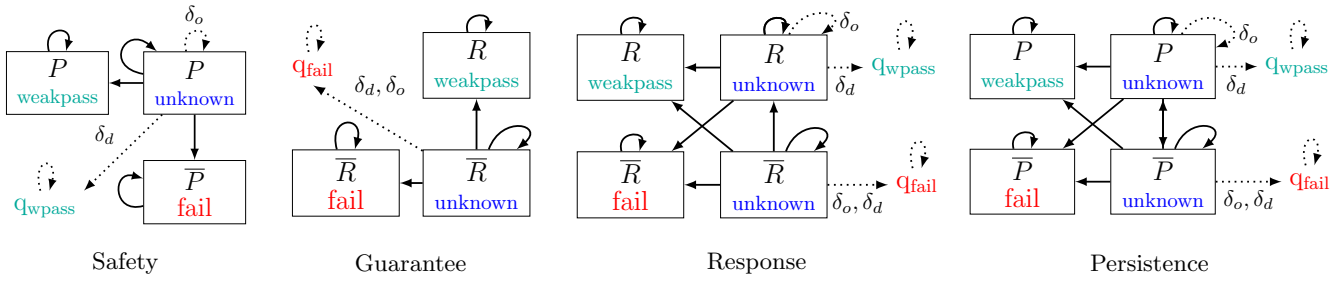
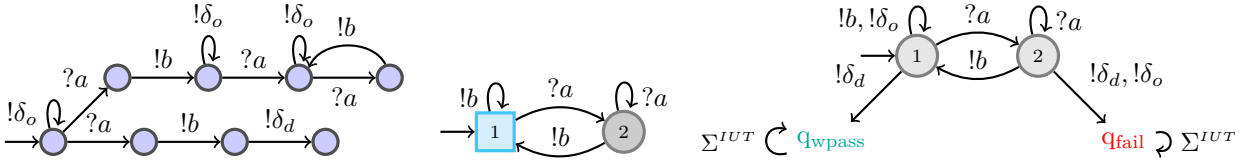


Fig. 13. Schematic illustrations of the canonical tester with quiescence for basic classes



(a) IUT with explicit quiescence (b) Π_4 defined by $\mathcal{A}_{\Pi_4}$ (c) Canonical tester for Π_4

Fig. 14. Illustrating the usefulness of quiescence

$$Exec(\mathcal{P}_{\Sigma^{IUT}}) = {\delta_o}^{\&} \cdot \left(a^{\&} + a \cdot b \cdot (\delta_d + {\delta_o}^{\&}) \cdot (a \cdot \left[{\delta_o}^{\&} \cdot ((a \cdot b)^{\&})^*\right]^* \cdot a^{\&})^{\&}\right)$$
$$Inter(\mathcal{P}_{\Sigma^{IUT}}) = {\delta_o}^{\&} \cdot \left((a \cdot b)^{\&} \cdot (\delta_d + {\delta_o}^{\&}) \cdot (a \cdot \left[{\delta_o}^{\&} \cdot ((a \cdot b)^{\&})^*\right]^* \cdot {\delta_o}^{\&})^{\&}\right)^{\&}$$

Fig. 15. Execution and interaction sequences of the IUT of Fig. 14a

should be acknowledged by an output $!b$. This property is not testable under the conditions expressed in Sect. 6, see Example 5. However, this *r-property* is testable with quiescence. One can observe that $Inter(\mathcal{P}_{\Sigma^{IUT}}) \not\subseteq \Pi_4$ because of the existence of $?a{\cdot}!b{\cdot}?a{\cdot}!\delta_o$ in $Inter(\mathcal{P}_{\Sigma^{IUT}})$. Indeed, we have:

$$\ominus\text{–}determined\text{–}q(?a{\cdot}!b{\cdot}?a{\cdot}!\delta_o, \Pi_4)$$

since the last event of $?a{\cdot}!b{\cdot}?a{\cdot}!\delta_o)$ is δ_o and $\neg\Pi_4(?a{\cdot}!b{\cdot}?a)$. The synthesized canonical tester, obtained following Definition 15, is depicted in Fig. 14c.

Example 12 (Canonical Testers with quiescence for the operating system). The canonical tester for the global specification, obtained following Definition 15, is depicted in Fig. 16a. As we can see, using quiescence allows us to test additional behaviors of the operating system. Moreover, following the same reasoning used for Π_4, using quiescence makes the *r-property* Π_5^{op} testable. The corresponding canonical tester is depicted in Fig. 16b.

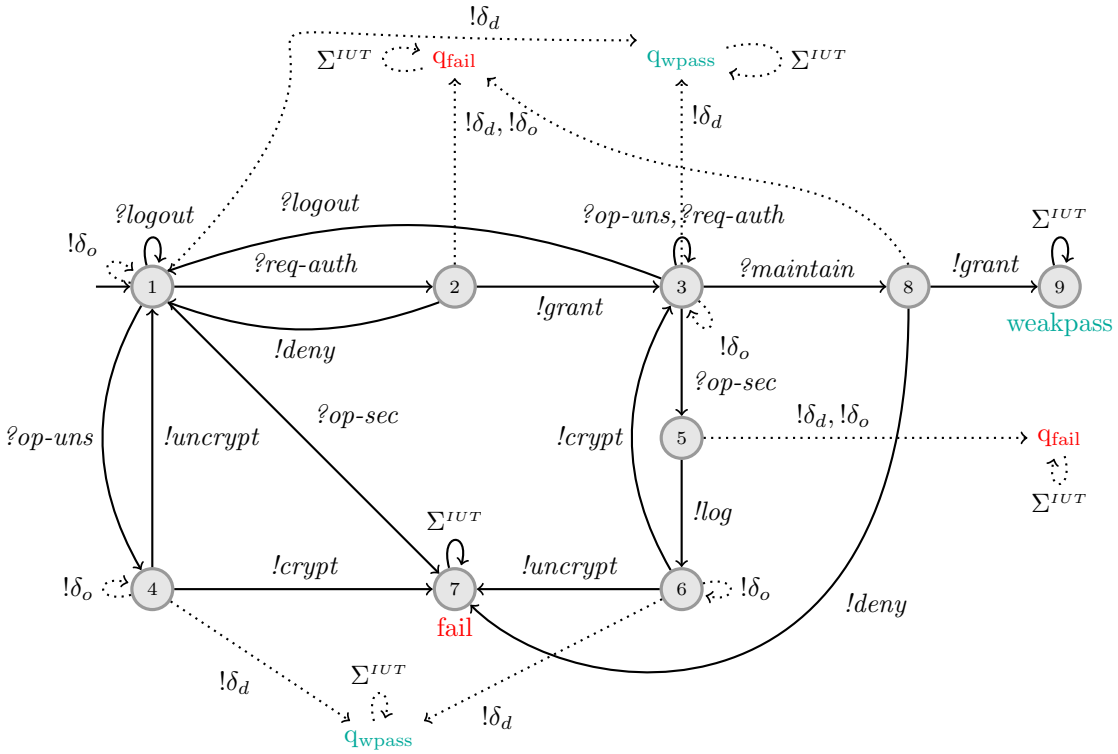
9 Related work and discussion

In this section we overview related work or work that may be leveraged by the results proposed in this paper. Then, we propose a discussion on the results afforded by this paper.

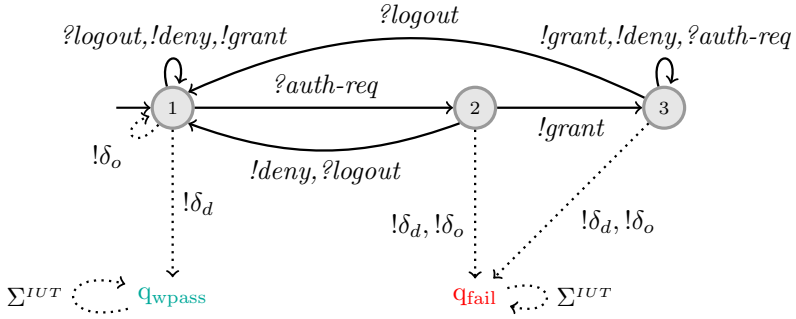
9.1 Conformance testing

The goal of conformance testing is to check that the behavior of a real software or hardware implementation is correct w.r.t. its specification. Correctness is checked through the observable actions of the implementation (its internal behavior is unknown). In formal conformance testing, one general correctness relation is the so-called *ioco* relation [31], stating that the implementation produces outputs as prescribed in the specification. In the context of ioco, there are several kinds of outputs: the implementation's outputs or *quiescence*. Quiescence models several sorts of observable inactivity of the system and is practically observed by timers. Roughly speaking, a possible approach to classical conformance testing (see e.g., [18]) proceeds as follows. Starting from the IOLTS specification $S = (Q^S, q^S_{\text{init}}, \Sigma, \longrightarrow_S)$ of the IUT, one has first to build $\Delta(S)$, the suspended IOLTS corresponding to S, obtained by adding a new observable output action δ and making IUT's inactivity explicit. This automaton is then determinized into $S' = Det(\Delta(S))$, thus representing the observable behavior of the IUT.

Classical conformance testing falls in the scope of our framework and can be easily expressed. Indeed, for a specification S, the conformance of an IUT $\mathcal{P}_\Sigma$ to S amounts to asserting the $\subseteq$ relation between $Exec(\mathcal{P}_\Sigma)$ and an *r-property* built from S. We start from the IOLTS S expressed over $\Sigma = \Sigma^S_? \cup \Sigma^S_!$ and then build $\Delta(S)$



(a) Canonical tester for the global specification Π^{op}



(b) Canonical tester for Π^{op}_5

Fig. 16. Canonical testers with quiescence for the operating system

and its determinized suspension IOLTS $S' = Det(\Delta(S))$. Then, we consider the *r-property* Π defined by the Streett safety automaton $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q^{\mathcal{A}_\Pi}_{\text{init}}, \longrightarrow_{\mathcal{A}_\Pi}, \{(\emptyset, P)\})$ s.t.

- $Q^{\mathcal{A}_\Pi} = Q^{S'} \cup \{sink\}$, with $sink \notin Q^{S'}$,
- $\longrightarrow_{\mathcal{A}_\Pi} = \longrightarrow_{S'} \cup \{(q, e, sink) \mid q \in Q^{S'} \wedge e \in \Sigma^S_! \cup \{\delta\} \wedge \neg\exists q' \in Q^{S'} : q \xrightarrow{e}_{S'} q'\}$,
- $P = Q^{S'}$.

Finally, we can build the IOLTS corresponding to the canonical tester following Definition 13. When emitting *fail* verdicts this canonical tester exactly detects violations of the conformance relation.

9.2 Testing oriented by properties for generating test purposes

One of the limits of conformance testing [31,32] lies in the size of the generated test suite which can be infinite or impracticable. Some testing approaches oriented by properties were proposed to face off this limitation by focusing on critical properties. In this case, properties are used complementary to the specification in order to generate test purposes which will be then used to conduct and select test cases. The goal of test purposes is to select a subset of test cases and their behaviors. Thus, a test purpose allows to evaluate specific features of the IUT. Once the test purposes are generated, a selection of test cases is possible using classical techniques defined on transition systems [18,9]. For instance, in [16], Fernandez *et al.* present an approach allowing to generate test

cases using LTL formula as test purposes. For a (non exhaustive) presentation of some general approaches, the reader is referred to [21].

Since these approaches are aimed to generate test purposes from properties, the testability of properties is deservedly left aside. One is interested in the satisfaction of a test case execution w.r.t. the considered property.

9.3 Combining testing and formal verification

In [6], the complementarity between verification techniques and conformance testing is studied. Notably, the authors showed that it is possible to detect (using testing) violations of safety (resp. satisfaction of co-safety) properties on the implementation and the specification. As co-safety properties in the Safety-Liveness classification are guarantee properties in the Safety-Progress classification, the framework proposed in [6] addresses only a subset of the properties stated as testable in this paper. The work proposed in [6] can then be leveraged by the work proposed in this paper.

9.4 Requirement-Based testing

In requirement-based testing, the purpose is to generate a test suite from a set of informal requirements. For instance, in [28,26], test cases are generated from LTL formula using a model-checker. Those approaches were interested in defining a syntactic test coverage for the tested requirements. Test case generation is driven by some test coverage criterion and no notion of testability is considered.

9.5 Property testing without a behavioral specification

Some previous approaches tackle the problem of testing properties on systems without behavioral specification. These approaches used the notion of tiles which are elementary test modules testing specific parts of an implementation and which can be combined to test more complex behaviors. A description of a tile-based approach was provided in [7] and formalized using a process algebra dedicated to testing [14]. Later [15], Falcone *et al.* have shown that this approach can be generalized to other formalisms (LTL and extended regular expressions) and that the test can be executed in a decentralized fashion. In [8], Darmaillacq *et al.* provided a case study dedicated to testing network security policies.

These approaches focused on deriving concrete test cases, given a property. No testability notion is associated to the considered properties. Moreover, in these approaches, the goal of a test execution is to determine the satisfaction of *one* execution w.r.t. the property without considering the whole set of execution sequences of the IUT.

9.6 Using the Safety-Progress classification in validation techniques

The *Safety-Progress* classification of properties is rarely used in validation techniques. We used (e.g., [12]) the *Safety-Progress* classification to characterize the sets of properties that can be verified and enforced at system runtime. In some sense, this previous endeavor similarly addressed the expressiveness question for runtime verification and runtime enforcement. In the light of the results afforded by [12] and this paper, we can remark that these three runtime-based validation techniques have different "expressiveness", i.e., the kind of properties that can be handled by these techniques are rather different.

In [3], Černá and Pelánek classified linear temporal properties according to the complexity of their verification. The motivation was to study the emptiness problem used in model-checking, according to the various classes. For this purpose, the authors introduced two additional views to the hierarchy. The first one is an extension of the original automata view in which temporal properties are characterized according to new acceptance conditions (Büchi, co-Büchi, weak, and terminal automata). The second one is an extension of the original logical view in which the authors organized temporal logic formula into a hierarchy according to alternation depth of temporal operators Until and Release.

9.7 Discussion

Several approaches fall in the scope of the generic one proposed in this paper. For instance, our results apply and extend the approach where verification is combined to testing as proposed in [6]. Furthermore, this approach leverages the use of test purposes [19,18] in testing to guide test selection. Indeed, the characterization of testable properties gives assets on the kind of test purposes that can be used in testing. Moreover, the properties considered in this paper are framed into the *Safety-Progress* classification of properties [23,4] which is equivalently a hierarchy of regular properties. Thus the results proposed by this paper concern previous depicted approaches in which the properties at stake can be formalized by a regular language. Furthermore, as we have seen in Sect. 9.1, classical conformance testing falls in the scope of the framework proposed by this paper.

10 Implementation: Java-PT

In this section we present the prototype tool Java-PT: Properties and their Testability with Java, an implementation of the previously described testing framework. It is mainly purposed to address testability issues w.r.t. an *r-property* under consideration. In particular, it allows to answer the following questions:

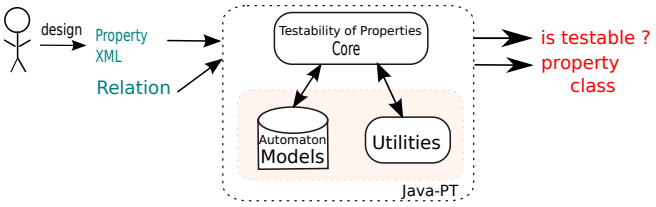


Fig. 17. Overview of Java-PT

– To which class (w.r.t. the *Safety-Progress* classification) does this property belong?
– Is this property testable w.r.t. a given testability relation?
– What is the canonical tester associated to this property for this testability relation?

This prototype can be freely downloaded at the following address:

`http://testableprops.forge.imag.fr/`

10.1 Overview

The tool is developed in the Java programming language and uses XML[13], XSLT[14], XStream[15] as underlying supporting technologies.

An overview of the architecture of Java-PT is given in Fig. 17. In the remainder of this section, we shall describe its functioning principle and its architecture.

The first step for a user before using the tool is to design an *r-property* that is purposed to be processed by the tool. The property is defined by a Streett automaton (see Sect. 10.2 for examples).

A model for automata-based objects. The tool Automaton Models of Java-PT consists of a hierarchy of classes modelling several entities (alphabet, states, and their transitions) related to the kinds of automata we consider (Streett and DFA). Those classes are used at several levels in Java-PT.

We also have implemented a component that provides means to make objects persistent in XML, i.e., being able to save and load automata from their description in a XML file. This utility consists in configuring and customizing the XStream library to realize serialization and deserialization.

Some utilities. The module *Utilities* is used by the module *Testability of Properties core* when processing properties. It contains the implementation of a set of useful operations on automata such as the computation of reachable states and the computation of the class of the automaton.

[13] Extensible Markup Language - `http://www.w3.org/XML/`
[14] The Extensible Stylesheet Language Family - `http://www.w3.org/Style/XSL/`
[15] `http://xstream.codehaus.org/`

The main module: Testability of Properties core. This modules leverages the modules *Automaton Models* and *Utilities*. It consists mainly in implementing the previous testability conditions.

10.2 Examples

In this section, we present some typical use cases of Java-PT. Examples presented in this paper (and other examples) can be found in the distribution of Java-PT, together with a complete user manual.

Let us come back to the *r-properties* Π_1 and Π_2 of Example 3. As seen in Sect. 6, these *r-properties* are testable w.r.t. the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$. Those *r-properties* are defined by their Streett automata described by the XML files in Fig. 18. Processing these properties with Java-PT is represented in Fig. 19, where:

– option `-in P.xml` provides an input property Π to be processed (in the file `P.xml`);
– option `-dc` computes the class of this property (according to the *Safety-Progress* classification);
– option `-it R` asks if this property is testable according to relation `R` (where `R` is encoded as an integer, e.g., 1 means $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$).

11 Conclusion and perspectives

Conclusion. In this paper, we study the classes of testable properties. We use a testability notion depending on a relation between the set of execution sequences that can be produced by the underlying implementation and the considered *r-property*. Leveraging the notions of positive and negative determinacy of properties, according to the relation of interest, we identify the testable fragments for each *Safety-Progress* class. Moreover we have seen that the framework of *r-properties* in the *Safety-Progress* classification provides a canonical tester able to produce a verdict depending on some finite interaction between the tester and the IUT. Furthermore, we also propose some conditions under which it makes sense for a tester to state weak verdicts. Besides, we address test generation for the general testing framework leveraged by properties. Finally, we have implemented the results of this paper in an available prototype tool.

Perspectives. A first research direction is to investigate the set of testable properties for more expressive formalisms. Indeed, the results afforded in the automata view of the *Safety-Progress* classification deal with regular properties. Classifying testable properties for more expressive specification formalisms (e.g., related to context-free properties) would be of interest. This would require to transpose the testability conditions expressed in the automata view to a more expressive recognition mechanism. Then decidability conditions of the state reachability problem need to be taken into account.

```
<StreettAutomaton>
  <Alphabet>
   <Event id="a"/>
   <Event id="b"/>
   <Event id="c"/>
  </Alphabet>
  <States class="tree-set">
    <no-comparator/>
    <State id="2">
      <Transition event="a" nextState="sink"/>
      <Transition event="b" nextState="2"/>
       <Transition event="c" nextState="sink"/>
    </State>
    <State id="1" initial="true">
      <Transition event="a" nextState="1"/>
      <Transition event="b" nextState="2"/>
      <Transition event="c" nextState="sink"/>
    </State>
     <State id="sink">
      <Transition event="a" nextState="sink"/>
      <Transition event="b" nextState="sink"/>
      <Transition event="c" nextState="sink"/>
    </State>
  </States>
  <Description>Automaton defining Pi1</Description>
  <AcceptingCondition>
    <Pair P="1,2" R=""/>
  </AcceptingCondition>
</StreettAutomaton>
```

```
<StreettAutomaton>
  <Alphabet>
   <Event id="a"/>
   <Event id="b"/>
  </Alphabet>
  <States class="tree-set">
    <no-comparator/>
    <State id="2">
      <Transition event="a" nextState="5"/>
      <Transition event="b" nextState="3"/>
    </State>
    <State id="1" initial="true">
      <Transition event="a" nextState="2"/>
      <Transition event="b" nextState="5"/>
    </State>
     <State id="5">
      <Transition event="a" nextState="5"/>
      <Transition event="b" nextState="5"/>
    </State>
    <State id="3">
      <Transition event="a" nextState="3"/>
      <Transition event="b" nextState="3"/>
    </State>
  </States>
  <Description>Automaton defining Pi2</Description>
  <AcceptingCondition>
    <Pair P="" R="3"/>
  </AcceptingCondition>
</StreettAutomaton>
```

Fig. 18. Defining Π_1 (left) and Π_2 (right) in XML

```
falcone-macbook:releases yfalcone$ java -jar java-PT.jar -in examples/Pi1.xml -dc -it 1
*****************************************************
* Java-PT: Properties and their Testability with Java
*****************************************************

Try to load file Pi1.xml...ok
Property of file Pi1.xml is a safety property.
Property of file Pi1.xml is Testable: true

falcone-macbook:releases yfalcone$ java -jar java-PT.jar -in examples/Pi2.xml -dc -it 1
*****************************************************
* Java-PT: Properties and their Testability with Java
*****************************************************

Try to load file Pi2.xml...ok
Property of file Pi2.xml is a guarantee property.
Property of file Pi2.xml is Testable: true
```

Fig. 19. Processing Π_1 and Π_2 with Java-PT

An additional perspective is to combine the proposed approach using weak verdicts with a notion of *test coverage*. Indeed, in order to bring some confidence in the fact that e.g., the implementation satisfies the property, it involves to execute the test several times to make it relevant. The various approaches [28,26] for defining test coverage for property-oriented testing could be used to reinforce a set of weak verdicts.

Another theoretical perspective would be to relax the hypothesis on alphabets made in this paper. We supposed that execution sequences of the IUT are expressed in the same alphabet as the one used to describe the property; thus avoiding the practical question of the interpretation of the IUT's sequences on the property's sequences. While this hypothesis is reasonable to study the testability of properties, in practice there might be a discrepancy between the alphabet of the property and the alphabet of actions used by the tester to interact with the IUT. This kind of situation arises for instance when testing network security policies: the policy is written at a level of abstraction different from the level of the test architecture (that may consider implementation details). It thus seems interesting to combine our results with the results in [15,7] where the alphabet discrepancy issue is specifically addressed.

Finally, it is our feeling that several previous testing frameworks using properties shall be revisited in the light of the results provided by this paper. For instance, in the security domain, testing frameworks dedicated to access-control policies [24] use rules that can be forma-

lized as safety or co-safety properties. Thus, it might be interesting to investigate whether other kinds of more evolved access-control rules, currently not tested, can be formalized as e.g., testable obligation or response properties. Consequently, those new rules could be validated using the revisited testing frameworks for access-control.

References

1. Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.
2. Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL Semantics for Runtime Verification. *Journal of Logic and Computation*, 20(3), 2010.
3. Ivana Cerná and Radek Pelánek. Relating the hierarchy of temporal properties to model checking. In *MFCS 2003: Proceedings of the* 28*th International Symposium on Mathematical Foundations of Computer Science*, volume 2747 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 2003.
4. Edward Chang, Zohar Manna, and Amir Pnueli. Characterization of Temporal Property Classes. In *Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 474–486. Springer, 1992.
5. Edward Chang, Zohar Manna, and Amir Pnueli. The Safety-Progress Classification. Technical report, Stanford University, Dept. of Computer Science, 1992.
6. Camille Constant, Thierry Jéron, Hervé Marchand, and Vlad Rusu. Integrating Formal Verification and Conformance Testing for Reactive Systems. *IEEE Trans. Software Eng.*, 33(8):558–574, 2007.
7. Vianney Darmaillacq, Jean-Claude Fernandez, Roland Groz, Laurent Mounier, and Jean-Luc Richier. Test generation for network security rules. In M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko, editors, *TestCom*, volume 3964 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2006.
8. Vianney Darmaillacq, Jean-Luc Richier, and Roland Groz. Test Generation and Execution for Security Rules in Temporal Logic. In *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 252–259. IEEE Computer Society, 2008.
9. R. G. de Vries. Towards formal test purposes. In Jan Tretmans and Ed Brinksma, editors, *FATES'01: Formal Approaches to Testing of Software*, BRICS Notes Series, pages 61–76, 2001.
10. E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. The MIT Press, 1990.
11. Yliès Falcone, Jean-Claude Fernandez, Thierry Jéron, Hervé Marchand, and Laurent Mounier. More Testable Properties. In Alexandre Petrenko, Adenilso da Silva Simão, and José Carlos Maldonado, editors, *ICTSS*, volume 6435 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2010.
12. Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime Verification of Safety-Progress Properties. In *the* 9*th Int. Workshop on Runtime Verification*, volume 5779 of *Lecture Notes in Computer Science*, pages 40–59. Springer, 2009.
13. Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *Software Tools for Technology Transfer*, 2011. DOI: 10.1007/s10009-011-0196-8. To appear.
14. Yliès Falcone, Jean-Claude Fernandez, Laurent Mounier, and Jean-Luc Richier. A test calculus framework applied to network security policies. In Klaus Havelund, Manuel Núñez, Grigore Rosu, and Burkhart Wolff, editors, *FATES/RV*, volume 4262 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2006.
15. Yliès Falcone, Jean-Claude Fernandez, Laurent Mounier, and Jean-Luc Richier. A Compositional Testing Framework Driven by Partial Specifications. In *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2007.
16. Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. Property oriented test case generation. In Alexandre Petrenko and Andreas Ulrich, editors, *FATES*, volume 2931 of *Lecture Notes in Computer Science*, pages 147–163. Springer, 2003.
17. Jens Grabowski. SDL and MSC based test case generation– an overall view of the SAMSTAG method. Technical report, University of Berne IAM-94-0005, 1994.
18. Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. *Software Tools for Technology Transfer*, 7(4):297–315, 2005.
19. Beat Koch, Jens Grabowski, Dieter Hogrefe, and Michael Schmitt II. Autolink: A tool for automatic test generation from SDL specifications. In *WIFT*, pages 114–. IEEE Computer Society, 1998.
20. Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.
21. Patricia D. L. Machado, Daniel A. Silva, and Alexandre C. Mota. Towards property oriented testing. In *Proceedings of the Second Brazilian Symposium on Formal Methods (SBMF 2005)*, volume 184 of *Electron. Notes Theor. Comput. Sci.*, pages 3–19. Elsevier Science Publishers B. V., 2007.
22. Wissam Mallouli, Jean-Marie Orset, Ana Cavalli, Nora Cuppens, and Frederic Cuppens. A Formal Approach for Testing Security Rules. In *SACMAT '07: Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 127–132. ACM, 2007.
23. Zohar Manna and Amir Pnueli. A Hierarchy of Temporal Properties (invited paper, 1989). In *PODC'90: Proceedings of the 9th symposium on Principles Of Distributed Computing*, pages 377–410. ACM, 1990.
24. Hervé Marchand, Jérémy Dubreil, and Thierry Jéron. Automatic testing of access control for security properties. In *TestCom/FATES'09*, volume 5826 of *Lecture Notes in Computer Science*, pages 113–128. Springer, 2009.
25. Robert Nahm, Jens Grabowski, and Dieter Hogrefe. Test Case Generation for Temporal Properties. Technical report, Bern University, 1993.
26. Charles Pecheur, Franco Raimondi, and Guillaume Brat. A Formal Analysis of Requirements-based Testing. In

ISSTA'09: Proceedings of the 18*th International Symposium on Software Testing and Analysis*, pages 47–56. ACM, 2009.
27. Amir Pnueli and Aleksandr Zaks. PSL Model Checking and Run-Time Verification Via Testers. In *FM06: Proceedings of the* 14*th Int Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006.
28. A. Rajan, M.W. Whalen, and M.R. Heimdahl. Model Validation using Automatically Generated Requirements-Based Tests. In *HASE '07: 10th IEEE Symposium on High Assurance Systems Engineering*, pages 95–104. IEEE computer society, 2007.
29. Robert S. Streett. Propositional Dynamic Logic of looping and converse. In *STOC '81: Proceedings of the* 13*th Symp. on Theory Of computing*, pages 375–383. ACM, 1981.
30. Yves Le Traon, Tejeddine Mouelhi, and Benoit Baudry. Testing Security Policies: Going Beyond Functional Testing. In *18th IEEE Int. Symp. on Software Reliability Engineering*, pages 93–102, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
31. Jan Tretmans. A formal approach to conformance testing. In Omar Rafiq, editor, *Protocol Test Systems*, volume C-19 of *IFIP Transactions*, pages 257–276. North-Holland, 1993.
32. Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer, 1996.

A Operators in the language-theoretic view of the *Safety-Progress* classification [13]

The language-theoretic view of the *Safety-Progress* classification is based on the construction of infinitary properties and finitary properties from finitary ones. It relies on the use of four operators A, E, R, P (building infinitary properties) and four operators A_f, E_f, R_f, P_f (building finitary properties) applying to finitary properties. In the original classification of Manna and Pnueli, the operators A, E, R, P, A_f, E_f were introduced. In this paper, we add the operators R_f and P_f and give a formal definition of all operators. In these definitions ψ is a finitary property over Σ.

Definition 16 (Operators A, E, R, P).

- $A(\psi) = \{\sigma \in \Sigma^\omega \mid \forall \sigma' \in \Sigma^*, \sigma' \prec \sigma \Rightarrow \psi(\sigma')\}$.
- $E(\psi) = \{\sigma \in \Sigma^\omega \mid \exists \sigma' \in \Sigma^*, \sigma' \prec \sigma \wedge \psi(\sigma')\}$.
- $R(\psi) = \{\sigma \in \Sigma^\omega \mid \forall \sigma' \in \Sigma^*, \sigma' \prec \sigma \Rightarrow \exists \sigma'' \in \Sigma^*, \sigma' \prec \sigma'' \prec \sigma \wedge \psi(\sigma'')\}$.
- $P(\psi) = \{\sigma \in \Sigma^\omega \mid \exists \sigma' \in \Sigma^*, \forall \sigma'' \in \Sigma^*, \sigma' \prec \sigma'' \prec \sigma \Rightarrow \psi(\sigma'')\}$.

$A(\psi)$ consists of all infinite words σ s.t. *all* prefixes of σ belong to ψ. $E(\psi)$ consists of all infinite words σ s.t. *some* prefixes of σ belong to ψ. $R(\psi)$ consists of all infinite words σ s.t. *infinitely many* prefixes of σ belong to ψ. $P(\psi)$ consists of all infinite words σ s.t. *all but finitely many* prefixes of σ belong to ψ.

The operators A_f, E_f, R_f, P_f build finitary properties from finitary ones.

Definition 17 (Operators A_f, E_f, R_f, P_f).

- $A_f(\psi) = \{\sigma \in \Sigma^* \mid \forall \sigma' \in \Sigma^*, \sigma' \preceq \sigma \Rightarrow \psi(\sigma')\}$.
- $E_f(\psi) = \{\sigma \in \Sigma^* \mid \exists \sigma' \in \Sigma^*, \sigma' \preceq \sigma \wedge \psi(\sigma')\}$.
- $R_f(\psi) = \{\sigma \in \Sigma^* \mid \psi(\sigma) \wedge \forall n \in \mathbb{N}, \exists \sigma' \in \Sigma^*, \sigma \prec \sigma' \wedge |\sigma'| \geq n \wedge \psi(\sigma')\}$.
- $P_f(\psi) = \{\sigma \in \Sigma^* \mid \psi(\sigma) \wedge \exists \sigma' \in \Sigma^*, \sigma \preceq \sigma' \wedge \forall n \in \mathbb{N}, \exists \sigma'' \in \Sigma^*, |\sigma''| = n \wedge \psi(\sigma' \cdot \sigma'')\}$.

$A_f(\psi)$ consists of all finite words σ s.t. *all* prefixes of σ belong to ψ. One can observe that $A_f(\psi)$ is the largest prefix-closed subset of ψ. $E_f(\psi)$ consists of all finite words σ s.t. *some* prefixes of σ belong to ψ. One can observe that $E_f(\psi) = \psi \cdot \Sigma^*$. $R_f(\psi)$ consists of all finite words σ s.t. $\psi(\sigma)$ and there exists an infinite number of continuations σ' of σ also belonging to ψ. $P_f(\psi)$ consists of all finite words σ belonging to ψ s.t. there exists a continuation σ' of σ s.t. σ' persistently has continuations σ'' staying in ψ (i.e., $\sigma' \cdot \sigma''$ belongs to ψ).

B Proofs

Closure of safety and guarantee r-properties. We first state the closure of safety and guarantee *r-properties* as a straightforward consequence of their definitions. Their closure will be used in the subsequent proofs.

Lemma 2 (Closure of safety and guarantee *r-properties*). *Considering an r-property $\Pi = (\phi, \varphi)$ defined over an alphabet Σ built from a finitary property ψ, the following facts hold:*

1. *Π is a safety r-property if and only if all prefixes of a sequence belonging to Π also belong to Π, i.e., Π is prefix-closed. That is:*
$$\forall \sigma \in \Sigma^\infty, \Pi(\sigma) \Rightarrow \forall \sigma' \in \Sigma^* : \sigma' \prec \sigma \Rightarrow \Pi(\sigma').$$
2. *Π is a guarantee r-property if and only if all continuations of a finite sequence belonging to Π also belong to Π, i.e., Π is extension-closed. That is:*
$$\forall \sigma \in \Sigma^* : \Pi(\sigma) \Rightarrow \forall \sigma' \in \Sigma^\infty : \Pi(\sigma \cdot \sigma').$$

Proof. We prove the two facts successively:

1. We have either $\phi(\sigma)$ or $\varphi(\sigma)$, i.e.,, all prefixes σ' of σ belong to ψ. Necessarily, all prefixes σ'' of σ' also belong to ψ, that is $\psi(\sigma'')$. By definition, that means $\sigma' \in A_f(\psi)$, i.e.,, $\psi(\sigma')$ and $\Pi(\sigma')$.
2. $\Pi(\sigma)$ implies that σ has at least one prefix $\sigma_0 \preceq \sigma$ belonging to ψ: $\sigma \in E_f(\psi)$. Then, any continuation of σ built using any finite or infinite sequence σ' has at least the same prefix belonging to ψ. If $\sigma' \in \Sigma^*$, we have $\sigma_0 \preceq \sigma \preceq \sigma \cdot \sigma'$ and $\sigma \cdot \sigma' \in E_f(\psi)$. If $\sigma' \in \Sigma^\omega$ we have $\sigma_0 \preceq \sigma \prec \sigma \cdot \sigma'$ and $\sigma \cdot \sigma' \in E(\psi)$. □

B.1 Proof of Theorem 1

The theorem states that, given a Streett m-automaton $\mathcal{A}_{\Pi} = (Q^{\mathcal{A}_{\Pi}}, q_{\text{init}}^{\mathcal{A}_{\Pi}}, \longrightarrow_{\mathcal{A}_{\Pi}}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ defining an *r-property* Π, according to the class of Π, the testability conditions expressed both in the language-theoretic and automata views are given in Table 1.

Proof. We first prove the testability conditions from the language view for each *Safety-Progress* class as given in Table 1. We have to prove that when the proposed conditions hold, the underlying property is negatively determined.

(i) For safety *r-properties*. Let Π be a safety *r-property*, then there exists $\psi \subseteq \Sigma^*$ s.t. Π can be expressed $(A_f(\psi), A(\psi))$. Let us consider $\sigma \in \overline{\psi}$. Then, according to Lemma 2 on the closure of safety *r-properties*, we can deduce that every finite (resp. infinite) continuation of σ does not belong to $A_f(\psi)$ (resp. $A(\psi)$). Thus every finite and infinite continuation of σ does not belong to $(A_f(\psi), A(\psi))$: Π is negatively determined by σ.

(ii) For guarantee *r-properties*. Let Π be a guarantee *r-property*, then there exists $\psi \subseteq \Sigma^*$ s.t. Π can be expressed $(E_f(\psi), E(\psi))$. Let σ be a sequence belonging to $\{\sigma \in \overline{\psi} \mid \mathit{pref}(\sigma) \cup \mathit{cont}(\sigma) \subseteq \overline{\psi}\}$. Then, every prefix and every continuation of σ do not belong to ψ. Consequently, these continuations cannot belong neither to $E_f(\psi)$, nor to $E(\psi)$ and (consequently) nor to Π as well: Π is negatively determined by σ.

(iii) For obligation *r-properties*. Let Π be an obligation *r-property*, then Π can be expressed (for instance) under conjunctive normal form, i.e., there exists $k \in \mathbb{N} \setminus \{0\}$, s.t.

$$\Pi = \bigcap_{i=1}^{k} (S_i(\psi_i) \cup G_i(\psi_i'))$$

where $S_i(\psi_i)$ (resp. $G_i(\psi_i')$) is a safety (resp. guarantee) *r-property* built upon ψ_i (resp. ψ_i').
Let σ_0 be a finite sequence belonging to $\overline{\psi_i} \cap \{\sigma \in \overline{\psi_i'} \mid \mathit{pref}(\sigma) \cup \mathit{cont}(\sigma) \subseteq \overline{\psi_i'}\}$ for some $i \in [1, k]$. Then, according to the proofs of (i) and (ii), each (finite or infinite) continuation σ' of σ_0 does not satisfy $(S_i(\psi_i) \cup G_i(\psi_i'))$. Hence, $\sigma' \notin \bigcap_{i=1}^{k} (S_i(\psi_i) \cup G_i(\psi_i'))$, which means $\neg\Pi(\sigma')$.

(iv) For response and persistence *r-properties*. The reasoning is similar to the one used for guarantee *r-properties*. Let Π be a response (resp. persistence) *r-property*, then there exists $\psi \subseteq \Sigma^*$ s.t. Π can be expressed $(R_f(\psi), R(\psi))$ (resp. $(P_f(\psi), P(\psi))$).
This *r-property* is testable if the set $\{\sigma \in \overline{\psi} \mid \mathit{cont}(\sigma) \subseteq \overline{\psi}\}$ is not empty. The difference with guarantee properties is that an *r-property* can be negatively determined even if it has some prefixes in ψ.

We now prove these testability conditions from the automata view. Let $\mathcal{A}_{\Pi}$ be the Streett automaton associated to Π, and let σ be a finite sequence of length n s.t. $\mathit{run}(\sigma, \mathcal{A}_{\Pi}) = q_0 \cdots q_n$.

(i) For safety *r-properties*. Let Π be a safety *r-property*. If $\forall i \in [0, n] : q_i \in \overline{P}$ then $\sigma \notin \Pi$, and, since there is no transition from $\overline{P}$ to P in $\mathcal{A}_{\Pi}$, each continuation σ' of σ is s.t. each state of $\mathit{run}(\sigma', \mathcal{A}_{\Pi})$ belongs to $\overline{P}$. Hence σ' is not accepted by $\mathcal{A}_{\Pi}$.

(ii) For guarantee *r-properties*. Let Π be a guarantee *r-property*. Let us assume that $q_n \in \overline{R} \wedge \mathit{Reach}_{\mathcal{A}_{\Pi}}(q_n) \subseteq \overline{R}$. Since there is no transition from R to $\overline{R}$ then $\sigma \notin \mathcal{A}_{\Pi}$. Moreover, each continuation σ' of σ is s.t. each state of $\mathit{run}(\sigma', \mathcal{A}_{\Pi})$ belongs to $\overline{R}$. Hence σ' is not accepted by $\mathcal{A}_{\Pi}$.

(iii) For obligation *r-properties*. Let Π be an obligation *r-property* and assume that $q_n \in (\overline{P_i} \cap \{q \in \overline{R_i} \mid \mathit{Reach}_{\mathcal{A}_{\Pi}}(q) \subseteq \overline{R_i}\}$ for some given $i \in [1, k]$. Then, according to the proofs of (i) and (ii) each continuation σ' of σ is s.t. each state of $\mathit{run}(\sigma', \mathcal{A}_{\Pi})$ belongs to $\overline{P_i} \cap \overline{R_i}$. Thus, σ' is not accepted by $\mathcal{A}_{\Pi}$.

(iv) For response *r-properties*. Let Π be a response *r-property* and assume that $q_n \in \{q \in \overline{R} \mid \mathit{Reach}_{\mathcal{A}_{\Pi}}(q) \subseteq \overline{R}\}$. Then, $\sigma \notin \Pi$ and each continuation σ' of σ is s.t. each state of $\mathit{run}(\sigma', \mathcal{A}_{\Pi})$ belongs to $\overline{R}$. Thus, σ' is not accepted by $\mathcal{A}_{\Pi}$.

(v) For persistence *r-properties*. The proof is similar to the one proposed for case *(iv)*. □

B.2 Proof of Lemma 1

Given $\sigma \in \Sigma^*$ a Streett automaton $\mathcal{A}_{\Pi}$ defining a *r-property* Π, and $q \in Q^{\mathcal{A}_{\Pi}}$ s.t. $q_{\text{init}}^{\mathcal{A}_{\Pi}} \xrightarrow{\sigma}_{\mathcal{A}_{\Pi}} q$, we have to prove:

- $q \in G^{\mathcal{A}_{\Pi}} \Leftrightarrow \oplus\text{–}\mathit{determined}(\sigma, \Pi)$,
- $q \in B^{\mathcal{A}_{\Pi}} \Leftrightarrow \ominus\text{–}\mathit{determined}(\sigma, \Pi)$.

Proof of $q \in G^{\mathcal{A}_{\Pi}} \Leftrightarrow \oplus\text{–}\mathit{determined}(\sigma, \Pi)$. We prove the implication in both ways.

- Let us suppose that $q \in G^{\mathcal{A}_{\Pi}}$. Using the acceptance criterion for finite sequences, we have that σ is accepted by $\mathcal{A}_{\Pi}$. Furthermore, as $\mathcal{A}_{\Pi}$ specifies Π, we have $\Pi(\sigma)$. Now, let us consider $\mu \in \Sigma^+$ s.t. $|\sigma| + |\mu| = n' > n$ and $\mathit{run}(\sigma \cdot \mu, \mathcal{A}_{\Pi}) = q_0 \cdots q_{n'}$. As $q \in G^{\mathcal{A}_{\Pi}}$, we deduce $\forall k \in \mathbb{N} : n \leq k \leq n' \Rightarrow q_k \in \bigcap_{i=1}^{m} R_i \cup P_i$ and consequently $\Pi(\sigma \cdot \mu)$. Let us consider $\mu \in \Sigma^{\omega}$, one may remark that $\forall i \in [1, m] : \mathit{vinf}(\sigma \cdot \mu, \mathcal{A}_{\Pi}) \cap R_i \neq \emptyset \vee \mathit{vinf}(\sigma \cdot \mu, \mathcal{A}_{\Pi}) \subseteq P_i$, which implies $\Pi(\sigma \cdot \mu)$. We have $\Pi(\sigma) \wedge \forall \mu \in \Sigma^{\infty} : \Pi(\sigma \cdot \mu)$, i.e., $\oplus\text{–}\mathit{determined}(\sigma, \Pi)$.
- Conversely, let us suppose that $\oplus\text{–}\mathit{determined}(\sigma, \Pi)$. By definition, it means $\forall \mu \in \Sigma^{\infty} : \Pi(\sigma \cdot \mu)$. According to the acceptance criterion of Streett automata, we deduce $\forall k \geq n, \forall \mu \in \Sigma^* :$

$$\mathit{run}(\sigma \cdot \mu, \mathcal{A}_{\Pi}) = q_0 \cdots q \cdots q_k \Rightarrow q_k \in \bigcap_{i=1}^{m} R_i \cup P_i.$$

That is, $\mathit{Reach}_{\mathcal{A}_{\Pi}}(q) \subseteq \bigcap_{i=1}^{m} (R_i \cup P_i)$, i.e., $q \in G^{\mathcal{A}_{\Pi}}$.

Proof of $q \in B^{\mathcal{A}_{\Pi}} \Leftrightarrow \ominus\text{–}determined(\sigma, \Pi)$. The proof can be done following the same principle as the one used to prove $q \in G^{\mathcal{A}_{\Pi}} \Leftrightarrow \oplus\text{–}determined(\sigma, \Pi)$. □

C Summary of notations

Table 3 summarizes the notations used throughout the paper.

Notation	Place Introduced	Meaning
$\mathbb{N}$	Sect. 3.1, p. 4	the set of non-negative integer (including 0)
Σ	Sect. 3.1, p. 4	the alphabet of actions
Σ^*	Sect. 3.1, p. 4	the set of finite sequences over Σ
Σ^+	Sect. 3.1, p. 4	the set of non-empty finite sequences over Σ
Σ^ω	Sect. 3.1, p. 4	the set of infinite sequences over Σ
Σ^∞	Sect. 3.1, p. 4	the set of both finite and infinite sequences over Σ
$\prec, \preceq$	Sect. 3.1, p. 4	strict and non-strict prefix notations
$pref(\sigma)$	Sect. 3.1, p. 4	the set of prefixes of $\sigma \in \Sigma^\infty$
$cont(\sigma)$	Sect. 3.1, p. 4	the set of finite continuations of $\sigma \in \Sigma^+$
σ_n	Sect. 3.1, p. 4	the $(n+1)$-th element of $\sigma \in \Sigma^+$ with $n \in [0, \lvert\sigma\rvert - 1]$
$\sigma_{\cdots n}$	Sect. 3.1, p. 4	the prefix of $\sigma \in \Sigma^\infty \setminus \{\epsilon\}$ containing the $n+1$ first elements
$\mathcal{P}_\Sigma$	Sect. 3.2, p. 4	a program with alphabet Σ
$Exec(\mathcal{P}_\Sigma)$	Sect. 3.2, p. 4	the set of all execution sequences of $\mathcal{P}_\Sigma$
$Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$	Sect. 3.2, p. 4	the set of finite execution sequences of $\mathcal{P}_\Sigma$
$Exec_\omega(\mathcal{P}_\Sigma)$	Sect. 3.2, p. 4	the set of infinite execution sequences of $\mathcal{P}_\Sigma$
$G = (Q^G, q^G_{\mathrm{init}}, \Sigma, \longrightarrow_G)$	Sect. 3.3, p. 4	IOLTS G defined on Σ
$Reach_G(q)$	Sect. 3.3, p. 4	the set of reachable states from q in an IOLTS G
$CoReach_G(X)$	Sect. 3.3, p. 4	the set of co-reachable states from $X \subseteq Q^G$ in an IOLTS G
$run(\sigma, G)$	Sect. 3.3, p. 4	the run of σ on the IOLTS G
ϕ	Sect. 3.4, p. 4	a finitary property ($\phi \subseteq \Sigma^*$)
φ	Sect. 3.4,p. 4	an infinitary property ($\varphi \subseteq \Sigma^\omega$)
Π	Sect. 3.5, p. 5	an *r-property* ($\Pi = (\phi, \varphi) \subseteq \Sigma^* \times \Sigma^\omega$)
$\oplus$*−determined*(σ, Π)	Sect. 3.5, p. 5	Π is positively determined by σ
$\ominus$*−determined*(σ, Π)	Sect. 3.5, p. 5	Π is negatively determined by σ

(a) Summary of the notations introduced in Sect. 3

Notation	Place Introduced	Meaning
A_f, E_f, R_f, P_f	Sect. 4.2, p. 6	*Safety-Progress* finitary language operators
A, E, R, P	Sect. 4.2, p. 6	*Safety-Progress* infinitary language operators
$(Q^{\mathcal{A}}, q^{\mathcal{A}}_{\mathrm{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$	Sect. 4.3, Def. 2, p. 6	Streett m-automaton
$R = \emptyset, \overline{P} \nrightarrow P$	Sect. 4.4, Def. 5, p. 7	syntactic restriction for Streett safety automata
$P = \emptyset, R \nrightarrow \overline{R}$	Sect. 4.4, Def. 5, p. 7	syntactic restriction for Streett guarantee automata
$\overline{P_i} \nrightarrow P_i, R_i \nrightarrow \overline{R_i}, i \in [1, m]$	Sect. 4.4, Def. 5, p. 7	syntactic restriction for Streett m-obligation automata
$P = \emptyset$	Sect. 4.4, Def. 5, p. 7	syntactic restriction for Streett response automata
$R = \emptyset$	Sect. 4.4, Def. 5, p. 7	syntactic restriction for Streett persistence automata
unrestricted	Sect. 4.4, Def. 5, p. 7	syntactic restriction for Streett reactivity automata
$(A_f(\psi), A(\psi))$	Sect. 4.4, Def. 5, p. 7	the safety *r-property* built from $\psi \subseteq \Sigma*$ (language view)
$(E_f(\psi), E(\psi))$	Sect. 4.4, Def. 5, p. 7	the guarantee *r-property* built from $\psi \subseteq \Sigma*$ (language view)
$\bigcap_{i=1}^m (S_i(\psi_i) \cup G_i(\psi'_i))$ or $\bigcup_{i=1}^m (S_i(\psi_i) \cap G_i(\psi'_i))$	Sect. 4.4, Def. 5, p. 7	the obligation *r-property* built from the $\psi_i \subseteq \Sigma^*$ and the $\psi'_i \subseteq \Sigma^*$, $i \in [1, m]$ (language view)
$(R_f(\psi), R(\psi))$	Sect. 4.4, Def. 5, p. 7	the response *r-property* built from $\psi \subseteq \Sigma*$ (language view)
$(P_f(\psi), P(\psi))$	Sect. 4.4, Def. 5, p. 7	the persistence *r-property* built from $\psi \subseteq \Sigma*$ (language view)

(b) Summary of the notations introduced in Sect. 4

Notation	Place Introduced	Meaning
$Exec(\mathcal{P}_\Sigma) \subseteq \Pi$	Def. 6, p. 10	testability relation stating that all behaviors of the IUT are allowed by the *r-property*
$Exec(\mathcal{P}_\Sigma) = \Pi$	Def. 6, p. 10	testability relation: "the behaviors of the IUT are those described by the *r-property*"
$Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$	Def. 6, p. 10	testability relation: "the behaviors of the IUT and those described by the *r-property* are not disjoint"
$\Pi \subseteq Exec(\mathcal{P}_\Sigma)$	Def. 6, p. 10	testability relation: "the IUT implements the *r-property*"
$verdict(\sigma, \mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi))$	Def. 7, p. 10	the verdict that the observation of σ allows to determine
$(Q^{\mathcal{O}}, q^{\mathcal{O}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{O}}, \Gamma^{\mathcal{O}})$	Def. 9, p. 11	test oracle (a Moore automaton: IOLTS with output function)

(c) Summary of the notations introduced in Sect. 5

Notation	Place Introduced	Meaning
$G^{\mathcal{A}_\Pi}, G_c^{\mathcal{A}_\Pi}, B_c^{\mathcal{A}_\Pi}, B^{\mathcal{A}_\Pi}$	Sect. 7.1, Def. 12, p. 16	the good, currently good, currently bad, and bad states of a Streett automaton $\mathcal{A}_\Pi$
$(Q^T, q^T_{\text{init}}, \Sigma, \longrightarrow_T, \Gamma^T)$	Sect. 7.1, Def. 13, p. 16	canonical tester (a Moore automaton)

(d) Summary of the notations introduced in Sect. 7

Notation	Place Introduced	Meaning
$Inter(\mathcal{P}_{\Sigma^{IUT}})$	Sect. 8.1, p. 18	the set of possible interactions of the tester with the IUT
$\ominus\text{–}determined\text{–}q(\sigma, \Pi)$	Sect. 8.1, Def. 14, p. 20	Π is negatively determined upon quiescence by σ
$\oplus\text{–}determined\text{–}q(\sigma, \Pi)$	Sect. 8.1, Def. 14, p. 20	Π is positively determined upon quiescence by σ

(e) Summary of the notations introduced in Sect. 8

Table 3. Summary of the notations used in the article