HAL
open science

More Testable Properties

Yliès Falcone, Jean-Claude Fernandez, Thierry Jéron, Hervé Marchand, Laurent Mounier

To cite this version:

Yliès Falcone, Jean-Claude Fernandez, Thierry Jéron, Hervé Marchand, Laurent Mounier. More Testable Properties. [Research Report] RR-7279, INRIA. 2010, pp.45. inria-00484297

HAL Id: inria-00484297

https://inria.hal.science/inria-00484297

Submitted on 18 May 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

More Testable Properties

Yliès Falcone — Jean-Claude Fernandez — Thierry Jéron — Hervé Marchand — Laurent Mounier

N° 7279

April 2010

Domaine 2

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7279--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

More Testable Properties

Yliès Falcone* , Jean-Claude Fernandez†, Thierry Jéron * ,
Hervé Marchand * , Laurent Mounier †

Domaine : Algorithmique, programmation, logiciels et architectures
Équipe-Projet Vertecs

Rapport de recherche n° 7279 — April 2010 — 42 pages

Abstract: In this paper, 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 wrt. several relations of interest. For each relation, we give a sufficient condition for a property to be testable. Then, we study and delineate, for each Safety-Progress class, the subset of testable properties and their corresponding test oracle producing verdicts for the possible test executions. Furthermore, we address automatic test generation for the proposed framework. Finally, we present a tool implementing the results proposed by this paper.

Key-words: testability, property, Safety-Progress classification, test oracle, negative/positive determinacy

* INRIA Bretagne Atlantique, Rennes, France. Email: FirstName.LastName@inria.fr. Homepage: `http://www.irisa.fr/prive/{yfalcone,jeron,hmarchand}`

† Vérimag, Grenoble, France. Email: FirstName.LastName@imag.fr. Homepage: `http://www-verimag.fr/~{fernand,mounier}`

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Plus de propriétés testables

Résumé : Nous explorons l'espace des propriétés testables dans la classification Safety-Progress. La testabilité des propriétés est définie selon l'existence d'une relation entre le système et la propriété examinée. Nous caractérisons l'ensemble des propriétés testables par rapport à plusieurs relations d'interets. Pour chaque relation, nous donnons une condition suffisante pour que la propriété soit testable. Ensuite, nous étudions et délimitons, pour chaque classe de propriétés, le sous-ensemble de propriétés testables. De plus, pour chaque relation considérée, nous définissons une notion d'oracle de test produisant des verdicts pour les différentes exécutions possibles du test. Finalement, nous présentons un outil prototype implémentant les résultats proposés.

Mots-clés : testabilité, propriété classification Safety-Progress, oracle de test, determination négative/positive

1 Introduction

Due to its ability to scale up well and its practical aspect, testing remains one of the most effective and widely used validation technique for software systems. Even if lots of work have already been carried out on this topic, improving the effectiveness of a testing phase while reducing its cost and time consumption remains a very important challenge, sustained by a strong industrial demand. 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 set of interaction sequences 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).

One way to improve the practical feasibility of a test campaign is to use a property. A property may be used for instance to drive the test execution. In this case, the property is used to generate the so-called test purposes [KGHS98, JJ05] which will select, among the possible test case behaviors, the most relevant ones. A property may also represent the desired behavior of the tested system. In this setting, the property may be for instance a formalization of a security policy describing prohibited behaviors and expectations from the users, as considered for instance in [TMB07, MOC+07]. Several testing approaches (*e.g.*, [CJMR07]) combine classical testing techniques and property verification so as to improve the test activity. Most of these approaches used safety and accessibility (co-safety) properties. A natural question is the existence of other kinds of properties that can be "tested", *i.e.,* to define a precise notion of *testability*.

The considered notion of testability. In [NGH93, Gra94], Nahm, Grabowski, and Hogrefe addressed this issue by discussing the set of temporal properties that can be tested on an implementation. A property is said to be *testable* if it is possible to determine if a given relation (*e.g.*, inclusion) holds between the sequences described by a property and the set of execution sequences that can be produced by interacting with the IUT, after the execution of a finite sequence on the IUT. In their work, testability of properties is studied wrt. the *Safety-Progress* classification ([CMP92a] and Section 3) for infinitary properties. The announced classes of testable properties are the safety and guarantee[1] classes. Then, it is not too surprising that most of the previously depicted approaches used safety and co-safety properties during testing.

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 execution sequences of the IUT. This property expresses observable behaviors,

[1] In the *Safety-Progress* classification the guarantee class is the co-safety class in the *Safety-Liveness* classification.

which may be desired or not. Usually, IUT's execution sequences are expressed in a different alphabet than the one used to describe the property and have thus to be interpreted. However, testability and the test oracle problem (*i.e.,* the problem of deciding verdicts) can be studied while abstracting this alphabet discrepancy. A second characteristic is that we do not require the existence of a an executable specification to generate the test cases. This allows to encompass several conformance testing approaches by viewing the specification as a special property.

Motivations and contributions. The main motivation of this paper is to leverage the use of an extended version of the *Safety-Progress* classification of properties dedicated to runtime techniques. We give a precise characterization of testable properties and provide a formal basis for several previous testing activities. We extend the results of [NGH93] by showing that lots of interesting properties (not safety nor guarantee) are also testable. Moreover, this framework allow to simply obtain test oracles producing verdicts according to the test execution.

Paper organization. The remainder of this paper is organized as follows. In Section 2, some preliminary concepts and notations are introduced. A quick overview of the *Safety-Progress* classification of properties for runtime validation techniques is given in Section 3. Section 4 introduces the notion of testability considered in this paper. In Section 5, testable properties are characterized. In Section 6, we study the conditions under which verdicts may be refined. Automatic test generation is addressed in Section 7. A presentation of a prototype tool implementing the proposed features is given in Section 8. Next, in Section 9, we overview the related work and propose a discussion on the results provided by this paper. Finally, Section 10 gives some concluding remarks and raised perspectives.

2 Preliminaries and notations

This section introduces some background, namely the notions of *program execution sequences*, IOLTSs, and *properties*.

2.1 Sequences, and execution sequences

Given an alphabet of actions Σ, a sequence σ on Σ is a total function $\sigma : I \to \Sigma$ where I is either the interval $[0, n]$ for some $n \in \mathbb{N}$, or $\mathbb{N}$ itself. The empty sequence is denoted by ϵ. We denote by Σ^* the set of 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 $|\sigma|$ and the $(i+1)$-th element of σ is denoted by σ_i. 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 $\sigma \preceq \sigma' \stackrel{\text{def}}{=} \sigma \prec \sigma' \vee \sigma = \sigma'$. For $\sigma \in \Sigma^\infty$ and $n \in \mathbb{N}$, $\sigma_{\dots n}$ is the sub-sequence containing the $n + 1$ first elements of σ. 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 $\mathit{cont}^*(\sigma) \stackrel{\text{def}}{=} \{\sigma' \in \Sigma^* \mid \exists \sigma'' \in \Sigma^* : \sigma' = \sigma \cdot \sigma''\}$.

The implementation that we are aiming to test is here a program $\mathcal{P}$ abstracted as a generator of execution sequences. We are interested in a restricted set of operations the program can perform. These operations influence the truth value of properties that we want to test. Such execution sequences can be made of access events on a secure system to its resources, or kernel operations on an operating system. In the testing context of this paper, those operations are made on PCOs. We abstract these operations by a finite set of *events*, namely a vocabulary Σ. We denote by $\mathcal{P}_\Sigma$ a program for which the vocabulary is Σ. The set of execution sequences of $\mathcal{P}_\Sigma$ is denoted by $Exec(\mathcal{P}_\Sigma) \subseteq \Sigma^\infty$. This set is *prefix-closed*, that is $\forall \sigma \in Exec(\mathcal{P}_\Sigma) : pref(\sigma) \subseteq Exec(\mathcal{P}_\Sigma)$. We will use $Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$ (resp. $Exec_\omega(\mathcal{P}_\Sigma)$) to refer to the finite (resp. infinite) execution sequences of $\mathcal{P}_\Sigma$, that is $Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \stackrel{\text{def}}{=} Exec(\mathcal{P}_\Sigma) \cap \Sigma^*$ and $Exec_\omega(\mathcal{P}_\Sigma) \stackrel{\text{def}}{=} Exec(\mathcal{P}_\Sigma) \cap \Sigma^\omega$. In the remainder of this article, we consider a vocabulary Σ.

2.2 IOLTSs

An Input Output Labelled Transition System (IOLTS) without internal actions and defined over an alphabet Σ is a 4-tuple $\mathcal{I} = (Q^\mathcal{I}, q^\mathcal{I}_{\text{init}}, \Sigma, \longrightarrow_\mathcal{I})$ where $Q^\mathcal{I}$ is a non-empty set of states where $q^\mathcal{I}_{\text{init}}$ the initial state. For IOLTSs the alphabet of actions $\Sigma = \Sigma^\mathcal{I}_! \cup \Sigma^\mathcal{I}_?$ is partitioned in output and input alphabets ($\Sigma^\mathcal{I}_!$ and $\Sigma^\mathcal{I}_?$). The relation $\longrightarrow_\mathcal{I} \subseteq Q^\mathcal{I} \times \Sigma^\mathcal{I} \times Q^\mathcal{I}$ is the transition relation. The (Σ', q_{new})-completion of the IOLTS $\mathcal{I}$ defined over Σ, where $\Sigma' \subseteq \Sigma$ and q_{new} is a fresh state, is the IOLTS $Comp^{q_{new}}_{\Sigma'}(\mathcal{I}) = (Q^\mathcal{I} \cup \{q_{new}\}, \Sigma, \longrightarrow'_\mathcal{I}, q^\mathcal{I}_{\text{init}})$ where $\longrightarrow'_\mathcal{I} = \longrightarrow_\mathcal{I} \cup \{q \xrightarrow{a}{}'_\mathcal{I} q_{new} \mid q \in Q^\mathcal{I}, a \in \Sigma', \nexists q' \in Q^\mathcal{I} : q \xrightarrow{a}_\mathcal{I} q'\}$.

2.3 Properties

Properties as sets of execution sequences. A *finitary property* (resp. an *infinitary property*, a *property*) is a subset of execution sequences of Σ^* (resp. Σ^ω, Σ^∞). Given a finite (resp. infinite) execution sequence σ and a property ϕ (resp. 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 we will consider are restricted to *linear time* execution sequences, excluding specific properties defined on powersets of execution sequences and branching properties.

Runtime properties [FFM09b]. In this paper we are interested in *runtime properties*, namely the properties that can be used in runtime-based validation techniques. As stated in the introduction, we consider finite and infinite execution sequences (that a program may produce). Runtime properties should characterize satisfaction for both kinds of sequences in a uniform way. To do so, we use *r-properties* (runtime properties) (that we introduced in [FFM09b] in the context of runtime verification) as pairs $(\phi, \varphi) \subseteq \Sigma^* \times \Sigma^\omega$. Intuitively, the finitary property ϕ represents the desirable property that finite execution sequences should fulfill, whereas the infinitary property φ is the expected property for infinite execution sequences. The definition of negation of an *r-property* follows from definition of 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 $* \in \{\cup, \cap\}$,

$(\phi_1, \varphi_1) * (\phi_2, \varphi_2) \stackrel{\text{def}}{=} (\phi_1 * \phi_2, \varphi_1 * \varphi_2)$. Considering an execution sequence $\sigma \in Exec(\mathcal{P}_\Sigma)$, we say that σ satisfies (ϕ, φ) when $\sigma \in \Sigma^* \wedge \phi(\sigma) \vee \sigma \in \Sigma^\omega \wedge \varphi(\sigma)$. For an *r-property* $\Pi = (\phi, \varphi)$, we note $\Pi(\sigma)$ (resp. $\neg\Pi(\sigma)$) when σ satisfies (resp. does not satisfy) (ϕ, φ).

In the sequel, we will need the notion of positive and negative determinacy [PZ06] introduced by Pnueli. This notion was introduced in the field of runtime verification in order to represent the situations when it is worth verifying a property at runtime [PZ06, FFM10].

Definition 1 (Positive/Negative determinacy [PZ06]). An *r-property* $\Pi \subseteq \Sigma^* \times \Sigma^\omega$ is said to be:

- negatively determined by $\sigma \in \Sigma^*$ if $\neg\Pi(\sigma) \wedge \forall \mu \in \Sigma^\infty : \neg\Pi(\sigma \cdot \mu)$, denoted $\ominus\text{–}determined(\sigma, \Pi)$;
- positively determined by $\sigma \in \Sigma^*$ if $\Pi(\sigma) \wedge \forall \mu \in \Sigma^\infty : \Pi(\sigma \cdot \mu)$, denoted $\oplus\text{–}determined(\sigma, \Pi)$.

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*.

3 A SP classification for runtime techniques

This section presents minimal theoretical background on the *Safety-Progress* (SP) classification of properties, introduced by Manna and Pnueli in [MP90, CMP92a], in a runtime context. This classification originally introduced a hierarchy between regular properties[2] defined as sets of *infinite* execution sequences. In [FFM09b], we extended the classification to deal with finite-length execution sequences by revisiting it using *r-properties* (in the automata view only, Section 3.2).

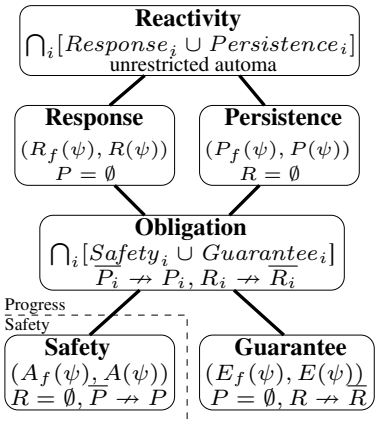


Figure 1: The SP classification

The *Safety-Progress* classification is an alternative to the classical *Safety-Liveness* [Lam77, AS84] 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 and according to 4 views [CMP92b]: a language-theoretic view (seeing properties as sets of sequences), 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 [Str81]).

A graphical representation of the *Safety-Progress* classification of properties is depicted in Fig. 1. Further details and results can be found in [FFM10]. Here, we will consider only the language-theoretic and the automata views dedicated to *r-properties* in which the properties will be defined relatively to a set of events Σ.

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

3.1 The language-theoretic view of *r-properties*

In the language-theoretic view of the *Safety-Progress* classification, properties of each classes are built using special operators.

3.1.1 Construction of *r-properties*

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) applied to finitary properties. In the following ψ is a finitary property over Σ.

$A(\psi)$ 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* (at least one) 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 extensions σ'' staying in ψ (*i.e.,* $\sigma' \cdot \sigma''$ belongs to ψ).

The formal definitions of these operators can be found in [FFM10]. One may remark the duality between those operators A and E, and operators R and P, *i.e.,* $\overline{A(\psi)} = E(\overline{\psi})$, and $\overline{R(\psi)} = P(\overline{\psi})$.

Based on these operators, each class can be seen from the language-theoretic view.

Definition 2 (*Safety-Progress* classes, language view). An *r-property* $\Pi = (\phi, \varphi)$ is defined to be

- A *safety r-property* if $\Pi = (A_f(\psi), A(\psi))$ for some finitary property ψ. That is, all prefixes of a finite word $\sigma \in \phi$ or of an infinite word $\sigma \in \varphi$ belong to ψ.
- A *guarantee r-property* if $\Pi = (E_f(\psi), E(\psi))$ for some finitary property ψ. That is, each finite word $\sigma \in \phi$ or infinite word $\sigma \in \varphi$ is guaranteed to have some prefixes (at least one) belonging to ψ.
- A *response r-property* if $\Pi = (R_f(\psi), R(\psi))$ for some finitary property ψ. That is, each finite word $\sigma \in \phi$ belongs to ψ and has an infinite extension with an infinite number of prefixes belonging to ψ. And, each infinite word $\sigma \in \varphi$ recurrently has (infinitely many) prefixes belonging to ψ.
- A *persistence r-property* if $\Pi = (P_f(\psi), P(\psi))$ for some finitary property ψ. That is, each finite word belongs to ψ and has an infinite extension with an finite number of prefixes not belonging to ψ (the prefixes "persistently" belongs to ψ). And, each infinite word $\sigma \in \varphi$ persistently has (continuously from a certain point on) prefixes belonging to ψ.

In all cases, we say that Π is built over ψ. Furthermore, obligation (resp. reactivity) *r-properties* are obtained by Boolean combinations of safety and guarantee (resp. response and persistence) *r-properties*.

Given a set of events Σ, we note $\text{Safety}(\Sigma)$ (resp. $\text{Guarantee}(\Sigma)$, $\text{Obligation}(\Sigma)$, $\text{Response}(\Sigma)$, $\text{Persistence}(\Sigma)$) the set of safety (resp. guarantee, obligation, response, persistence) *r-properties* defined over Σ.

In Section 3.3, we illustrate the construction of infinitary properties from finitary ones for each of the four operators.

3.1.2 Some useful facts in the language view

Now, we give some useful facts about *r-properties* in the language view. Those facts will be used in the remainder when characterizing the set of testable properties.

Let us first note the duality between some classes coming from the duality of the operators:

- Π is a safety *r-property* iff $\overline{\Pi}$ is a guarantee *r-property*.
- Π is a response *r-property* iff $\overline{\Pi}$ is a persistence *r-property*.

We now state the closure of safety and guarantee *r-properties* as a straightforward consequence of their definitions.

Remark 1 (Closure of *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 iff* all prefixes of a sequence belonging to Π also belong to Π, *i.e.,* Π is prefix-closed.

2 Π is a guarantee *r-property iff* all continuations of a finite sequence belonging to Π also belong to Π, *i.e.,* Π is extension-closed. $*$

The following lemma (inspired from [CMP92b]) provides a decomposition of each obligation property in a normal form. The proof of this lemma can be found in [FMFR10].

Lemma 1 : *Any obligation r-property can be represented as the intersection*

$$\bigcap_{i=1}^{k} (\text{Safety}_i \cup \text{Guarantee}_i)$$

for some $k > 0$, where Safety_i *and* Guarantee_i *are respectively safety and guarantee r-properties. We refer to this presentation as the conjunctive normal form of obligation r-properties.*

Similarly, any obligation *r-property* can be expressed in disjunctive normal form: $\bigcup_{i=1}^{k} (\text{Safety}'_i \cap \text{Guarantee}'_i)$.

When an *r-property* Π is expressed as $\cap_{i=1}^{k} (\text{Safety}_i \cup \text{Guarantee}_i)$ or $\cup_{i=1}^{k} (\text{Safety}_i \cap \text{Guarantee}_i)$, Π is said to be a k-obligation *r-property*. Similar definitions and properties hold for reactivity *r-properties* which are expressed by combination of response and persistence *r-properties*.

3.2 The automata view of *r-properties* [FFM09b]

For the automata view of the *Safety-Progress* classification, we follow [CMP92b] and define *r-properties* using Streett automata. Furthermore, for each class of the *Safety-Progress* classification it is possible to syntactically characterize a recognizing finite-state automaton.

3.2.1 Streett automata

We define[3] a variant of deterministic and complete Streett automata (introduced in [Str81] and used in [CMP92b]) for property recognition. These automata process events and decide properties of interest. We add to original Streett automata a finite-sequence recognizing criterion in such a way that these automata uniformly recognize *r-properties*.

Definition 3 (Streett automaton). A deterministic finite-state Streett automaton is a tuple $(Q, q_{\text{init}}, \Sigma, \longrightarrow, \{(R_1, P_1), \ldots, (R_m, P_m)\})$. The set Q is the set of states, $q_{\text{init}} \in Q$ is the initial state. The function $\longrightarrow : Q \times \Sigma \to Q$ is the (complete) transition function. In the following, for $q, q' \in Q, e \in \Sigma$ we abbreviate $\longrightarrow (q, e) = q'$ by $q \xrightarrow{e} q'$. The set $\{(R_1, P_1), \ldots, (R_m, P_m)\}$ is the set of accepting pairs, for all $i \leq n$, $R_i \subseteq Q$ are the sets of recurrent states, and $P_i \subseteq Q$ are the sets of persistent states.

We refer to an automaton with m accepting pairs as an m-automaton. When $m = 1$, a 1-automaton is also called a *plain*-automaton, and we refer to R_1 and P_1 as R and P. Moreover, for $\sigma = \sigma_0 \cdots \sigma_{n-1}$ a word of Σ^* of length n and $q, q' \in Q^{\mathcal{A}}$ two states, we note $q \xrightarrow{\sigma} q'$ when $\exists q_1, \ldots, q_{n-2} \in Q^{\mathcal{A}} : q \xrightarrow{\sigma_0} q_1 \wedge \ldots \wedge q_{n-2} \xrightarrow{\sigma_{n-2}} q'$. In the following $\mathcal{A} = (Q^{\mathcal{A}}, q^{\mathcal{A}}_{\text{init}}, \Sigma, \longrightarrow_{\mathcal{A}}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ designates a deterministic finite-state Streett m-automaton.

For $q \in Q^{\mathcal{A}}$, $Reach_{\mathcal{A}}(q)$ is the set of reachable states from q with at least one transition in $\mathcal{A}$ and q itself, that is $Reach_{\mathcal{A}}(q) = \{q' \in Q^{\mathcal{A}} \mid \exists \sigma \in \Sigma^+ : q \xrightarrow{\sigma}_{\mathcal{A}} q'\} \cup \{q\}$. For $\sigma \in \Sigma^\infty$, the *run* of σ on $\mathcal{A}$ is the sequence of states involved by the execution of σ on $\mathcal{A}$. It is formally defined as $run(\sigma, \mathcal{A}) = q_0 \cdot q_1 \cdots$ where $\forall i : (q_i \in Q^{\mathcal{A}} \cap Reach_{\mathcal{A}}(q^{\mathcal{A}}_{\text{init}}) \wedge q_i \xrightarrow{\sigma_i}_{\mathcal{A}} q_{i+1}) \wedge q_0 = q^{\mathcal{A}}_{\text{init}}$.

For an execution sequence $\sigma \in \Sigma^\omega$ on a Streett automaton $\mathcal{A}$, we define $vinf(\sigma, \mathcal{A})$, as the set of states appearing infinitely often in $run(\sigma, \mathcal{A})$. It is formally defined as follows: $vinf(\sigma, \mathcal{A}) = \{q \in Q^{\mathcal{A}} \mid \forall n \in \mathbb{N}, \exists m \in \mathbb{N} : m > n \wedge q = q_m \in run(\sigma, \mathcal{A})\}$.

For a Streett automaton, the notion of acceptance condition is defined using the accepting pairs.

Definition 4 (Acceptance condition for infinite sequences). For $\sigma \in \Sigma^\omega$, we say that $\mathcal{A}$ 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 is accepted by a Streett m-automaton, if for each $i \in [1, m]$, the set of states visited infinitely often are all in the persistent states of the pair i, or the states visited infinitely often contains at least one recurrent state of the pair i.

[3] There exist several equivalent definitions of Streett automata dedicated to infinite sequences recognition. We choose here to follow the definition used in [CMP92b].

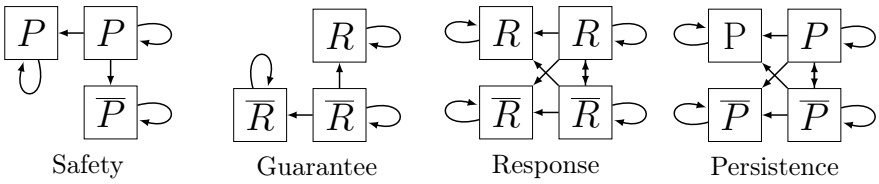


Figure 2: Illustrations of the shapes of Streett automata for basic classes

To deal with *r-properties* we need to define also an acceptance criterion for *finite* sequences:

Definition 5 (Acceptance condition for finite sequences). For a finite-length execution sequence $\sigma \in \Sigma^*$ s.t. $|\sigma| = n$, we say that the m-automaton $\mathcal{A}$ accepts σ if $(\exists q_0, \ldots, q_{n-1} \in Q^{\mathcal{A}} : run(\sigma, \mathcal{A}) = q_0 \cdots q_{n-1} \wedge q_0 = q^{\mathcal{A}}_{\text{init}}$ and $\forall i \in [1, m] : q_{n-1} \in P_i \cup R_i)$.

A finite sequence is accepted by a Streett automaton if and only if it terminates on a distinguished state R_i or P_i for each accepting pair i.

3.2.2 The hierarchy of automata.

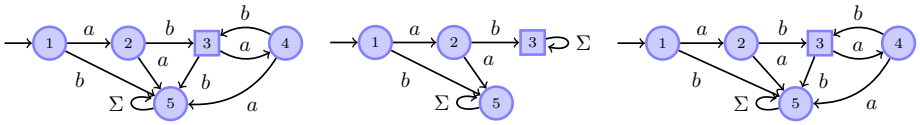
An interesting feature of Streett automata is that the class of property they recognize can be easily characterized by some syntactic considerations.

- A *safety automaton* is a plain automaton s.t. $R = \emptyset$ and there is no transition from a state $q \in \overline{P}$ to a state $q' \in P$.
- A *guarantee automaton* is a plain automaton s.t. $P = \emptyset$ and there is no transition from a state $q \in R$ to a state $q' \in \overline{R}$.
- An *m-obligation automaton* is an m-automaton s.t. for each i in $[1, m]$:
  - there is no transition from $q \in \overline{P_i}$ to $q' \in P_i$,
  - there is no transition from $q \in R_i$ to $q' \in \overline{R_i}$.
- A *response automaton* is a plain automaton s.t. $P = \emptyset$.
- A *persistence automaton* is a plain automaton s.t. $R = \emptyset$.
- A *reactivity automaton* is any unrestricted automaton.

The syntactic restrictions are illustrated in Fig. 2: shapes of Streett automata for basic classes are depicted. In these illustrations, we distinguish "terminal" recurrent and persistence states. For instance, for safety automata, we made the distinction between persistent states for which it is not possible to reach non persistent states and the other persistent states. One may remark that these syntactic restrictions hold for the automata represented in Figs. 3b, 4b and 4c. One may also notice that the duality between the classes, expressed in the language-theoretic view, holds for the hierarchy of automata as well.

(a) $\psi_1 = a^* \cdot \left(b^* + c \cdot (c+a)^* \cdot b^+\right)$ (b) Streett safety, $P = \{1, 2\}$

Figure 3: DFA for ψ_1 and Streett for $(A_f(\psi_1), A(\psi_1))$



(a) $\psi_2 = (a \cdot b)^+$ (b) Streett guarantee (c) Streett response

Figure 4: DFA for ψ_2 and Streett for $(E_f(\psi_2), E(\psi_2))$, $(R_f(\psi_2), R(\psi_2))$, $R = \{3\}$

Automata and properties. We now link Streett automata to *r-properties*.

Definition 6 (Automata and *r-properties*). We say that a Streett automaton $\mathcal{A}$ *defines* an *r-property* $(\phi, \varphi) \in 2^{\Sigma^* \times \Sigma^\omega}$ if and only if the set of finite (resp. infinite) execution sequences accepted by $\mathcal{A}$ is equal to ϕ (resp. φ), which is noted $\mathcal{L}(\mathcal{A}) = (\phi, \varphi)$.

3.3 Examples and summary

The following example illustrates the concepts introduced in this section.

Example 1 (*r-properties*[4]) Let us consider $\Sigma_1 = \{a, b, c\}$ and $\psi_1 = a^* \cdot \left(b^* + c \cdot (c+a)^* \cdot b^+\right)$ defined by the deterministic finite-state automaton (DFA) in Fig. 3a with accepting states $1, 2$. The Streett automaton in Fig. 3b defines $\Pi_1 = (A_f(\psi_1), A(\psi_1))$.

Let us consider $\Sigma_2 = \{a, b\}$, and the finitary property $\psi_2 = (a \cdot b)^+$ recognized by the DFA depicted in Fig. 4a. The Streett automaton in Fig. 4b (resp. Fig. 4c) represents the guarantee (resp. response) *r-property* $\Pi_2 = (E_f(\psi_2), E(\psi_2))$ (resp. $(R_f(\psi_2), R(\psi_2))$ built upon ψ_2 with $R = \{3\}$.

A graphical representation of the *Safety-Progress* hierarchy of properties is depicted in Fig. 1. A link between two classes means that the higher class contains strictly the lower one. Furthermore, for each class, we have recalled and uniformly extended the characterizations in the language-theoretic and automata views.

Remark 2 It is worth noticing that property interpretation of finite sequences extends to infinite sequences in a consistent way, depending on the class of properties under consideration. Considering $\sigma \in \Sigma^\omega$, we have:

- for a safety property Π, $\forall i \in \mathbb{N} : \Pi(\sigma_{\dots i}) \Leftrightarrow \Pi(\sigma)$

- for a guarantee property Π, $\exists i \in \mathbb{N} : \Pi(\sigma_{\dots i}) \Leftrightarrow \Pi(\sigma)$
- for a response property Π, $\overset{\infty}{\exists}\, i \in \mathbb{N} : \Pi(\sigma_{\dots i}) \Leftrightarrow \Pi(\sigma)$
- for a persistence property Π, $\overset{\infty}{\nexists}\, i \in \mathbb{N} : \neg\Pi(\sigma_{\dots i}) \Leftrightarrow \Pi(\sigma)$

where $\overset{\infty}{\exists}$ means "there exists an infinite number of". $*$

4 Some notions of testability

From its *finite* interaction with the underlying implementation under test, the tester produces an interpretation. This interpretation is a sequence of events in Σ^*. We study the conditions for a tester, using the produced sequence of events, to determine whether a given relation holds between:

- the set of *all* (finite and infinite) execution sequences that can be produced by the IUT: $Exec(\mathcal{P}_\Sigma)$,
- the set of sequences described by the *r-property* Π.

Roughly speaking, the challenge addressed by a tester is thus to determine a verdict, *i.e.,* to state whether a relation holds between Π and $Exec(\mathcal{P}_\Sigma)$, from a finite sequence extracted from $Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$.

Let us recall that the *r-property* is a pair made of two sets: a set of finite sequences and a set of infinite sequences (see Section 2.3). In the following, we shall compare this pair to the set of execution sequences of the IUT which is a set constituted of finite and infinite sequences. In the remainder, in order to simplify notations, when we compare those pairs, we are making two comparisons: we compare the finitary (resp. infinitary) part of the *r-property* to the restriction to finite (resp. infinite) sequences of the program. We explicit this comparison in Definition 7, then we use a simplified notation.

As noticed in [NGH93], 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 7 (Relations between an IUT and an *r-property*). The possible relations of interest between $Exec(\mathcal{P}_\Sigma)$ and Π are:

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

Then, the test verdict is 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 interaction* $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$. The following verdicts are produced when the test activity and the considered verdicts deal with *all* executions of the IUT. In Section 6, we will study the conditions under which one can state weaker verdicts on a *single* execution sequence.

Definition 8 (Verdicts [NGH93]). Given a relation $\mathcal{R}$ between $Exec(\mathcal{P}_\Sigma)$ and Π, defined in the sense of Definition 7, the tester produces:

- *pass* if the test execution allows to determine that $\mathcal{R}$ holds;
- *fail* if the test execution allows to determine that $\mathcal{R}$ does not hold;
- *unknown*[5] if the test execution does not allow yet to determine $\mathcal{R}$.

From a test sequence $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$ produced by the tester and executed on the IUT, we note $verdict(\sigma, \mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi))$ the verdict that the observation of σ allows to determine. Let us remark that the two following problems may occur in practice:

- In the general case, the IUT may be a program exhibiting infinite-length execution sequences. Evaluating those sequences wrt. an *r-property* Π is thus not realizable by a tester.
- Moreover, finite execution sequences contained in the *r-property* cannot be processed easily. For instance, if 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 continuation of the test execution would not exhibit a new sequence belonging to the *r-property*, *i.e.,* $\sigma' \in \Sigma^\infty$ s.t. $\sigma \cdot \sigma' \in \Pi$.

Thus, the test should stop only when there is no doubt regarding the verdict to be established. That is to say, the moment when it is not worth letting the execution continue since no future possible continuation can question the produced verdict. Following [NGH93], 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. To do so, we will suppose the existence of a tester that can interpret the execution sequences of the IUT on $Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$.

Definition 9 (Testability). An *r-property* Π is said to be *testable* on $\mathcal{P}_\Sigma$ wrt. the relation $\mathcal{R}$ if there exists an execution sequence $\sigma \in \Sigma^*$ s.t.:

$$\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \Rightarrow verdict(\sigma, \mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi)) \in \{pass, fail\}$$

[5] In [NGH93], 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.

Intuitively, this condition compels the existence of a sequence which, if one can play it on the IUT, allows to determine for sure, if the relation holds or not. Let us note that this definition entails to synthesize a test oracle, *i.e.,* in our case a computable[6] function which allows to determine $\mathcal{R}\big(Exec(\mathcal{P}_\Sigma), \Pi\big)$ from the observation of a sequence $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$.

Test oracles. In the remainder, we will be interested in defining test oracles for the various test relations we consider. A test oracle, is finite state machine (FSM) parametrized by a test relation as shown in Definition 7. It reads incrementally an interaction sequence $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma)$ and produces verdicts in $\{pass, fail, unknown\}$.

Definition 10 (Test Oracle). A *test oracle* $\mathcal{O}$ for an IUT $\mathcal{P}_\Sigma$, a relation $\mathcal{R}$ and an *r-property* Π is a 4-tuple $(Q^{\mathcal{O}}, q^{\mathcal{O}}_{\mathrm{init}}, \longrightarrow_{\mathcal{O}}, \Gamma^{\mathcal{O}})$. The finite set $Q^{\mathcal{O}}$ denotes the control states and $q^{\mathcal{O}}_{\mathrm{init}} \in Q^{\mathcal{O}}$ 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 with the following constraints:

- all states emitting a *pass* or a *fail* verdict are final (sink states),
- $\exists \sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) : q^{\mathcal{O}}_{\mathrm{init}} \xrightarrow{\sigma}_{\mathcal{O}} q \wedge \Gamma(q) = pass \Rightarrow \mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi)$,
- $\exists \sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) : q^{\mathcal{O}}_{\mathrm{init}} \xrightarrow{\sigma}_{\mathcal{O}} q \wedge \Gamma(q) = fail \Rightarrow \neg\mathcal{R}(Exec(\mathcal{P}_\Sigma), \Pi)$.

5 Characterizing testable properties without executable specification

The framework of *r-properties* (Section 3) allows to determine, according to the considered relation, the testability of the different kinds of properties. Moreover, this framework provides a computable oracle, which is a sufficient condition for testing. 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 the played test sequence.

We now characterize the set of testable properties according to the tested relation between $Exec(\mathcal{P}_\Sigma)$ and Π. To do so, we will use the notion of positive and negative determinacy applied to *r-properties* (see Section 2.3).

5.1 Testability wrt. the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$

Obtainable verdicts and sufficient conditions. For this relation, the unique verdicts that may be produced are *fail* and *unknown*. We explicit this below.

A *pass* verdict means that all execution sequences of $\mathcal{P}_\Sigma$ 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 verified. Obviously, every implementation with alphabet Σ satisfies this relation. In other cases,

[6] It will be indeed the case since we are working in the *Safety-Progress* classification which is dedicated to regular properties. The oracle will consist in a composition of decidable questions on regular sets.

it is impossible to obtain such a verdict (whatever is the property class under consideration), since the whole set $\mathcal{P}_\Sigma$ is usually unknown from the tester. In Section 6, we will study the conditions under which it is possible to state *weak pass* verdicts, when reasoning on a *single* execution sequence of the IUT.

A *fail* verdict means that there are some sequences produced by the program which are not in Π. In order to produce such a verdict, we need to observe a *finite* sequence σ that is a prefix of a program's execution sequence. Moreover, in order to stop the test, we need to be sure that there is no σ-continuation which belongs to Π. That is to say, one needs to exhibit an execution sequence of $\mathcal{P}_\Sigma$ s.t. Π is *negatively determined* by this sequence.

For this relation, we will be thus only interested in seeking *fail* verdicts under the sufficient condition expressed by the following theorem:

Theorem 1 (Condition to determine a *fail* verdict for $\boldsymbol{Exec(\mathcal{P}_\Sigma) \subseteq \Pi}$): *It is possible to state that the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ is not verified, if there exists an execution sequence of the program s.t. the r-property Π is negatively determined. More formally:*

$$\exists \sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) : \ominus\text{–}determined(\sigma, \Pi) \;\Rightarrow\; verdict(\sigma, Exec(\mathcal{P}_\Sigma) \subseteq \Pi) = fail \quad (1)$$

Proof : The proof of this theorem is straightforward. Indeed, if the program produces an execution sequence σ which cannot be continued in a finite nor infinite execution sequence belonging to the *r-property* Π, then the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ is necessarily not verified. Indeed, we know for sure that there is at least one sequence of the program not belonging to the *r-property*. ∎

Testability of this relation in the *Safety-Progress* classification. For each of the following classes, we state the conditions under which the properties of this class are testable. Moreover, we exhibit the test sequences that are possible to test on the implementation in order to determine a *fail* verdict for the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$.

Theorem 2 (Language-view of the testability of $\boldsymbol{Exec(\mathcal{P}_\Sigma) \subseteq \Pi}$): *For each Safety-Progress class, the testability conditions in order to obtain a fail verdict and expressed in the language-theoretic view are the following:*

- *a safety r-property $(A_f(\psi), A(\psi))$ is testable if $\overline{\psi} \neq \emptyset$;*
- *a guarantee r-property $(E_f(\psi), E(\psi))$ is testable if $\{\sigma \in \overline{\psi} \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \overline{\psi}\} \neq \emptyset$;*
- *a k-obligation r-property built over ψ_i and ψ'_i, $i \in [1, k]$,*
  - *and expressed in conjunctive normal form is testable if $\bigcup_{i=1}^{k} \left(\overline{\psi_i} \cap \{\sigma \in \overline{\psi'_i} \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \overline{\psi'_i}\}\right) \neq \emptyset$,*
  - *and expressed in disjunctive normal form is testable if $\bigcap_{i=1}^{k} (\overline{\psi_i} \cup \{\sigma \in \overline{\psi'_i} \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \overline{\psi_i}\}) \neq \emptyset$;*
- *a response (resp. persistence) r-property $(R_f(\psi), R(\psi))$ (resp. $(P_f(\psi), P(\psi))$) is testable if $\{\sigma \in \overline{\psi} \mid cont^*(\sigma) \subseteq \overline{\psi}\} \neq \emptyset$.*

PROOF : We prove the testability conditions for each *Safety-Progress* class.

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 $\sigma \in \Sigma^* \cap \overline{\psi}$. Then, according to Remark 1 on the closure of safety *r-properties* we have 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. Let us remark that the only safety *r-property* which is not testable under these conditions is the *r-property* always true: $(\Sigma^*, \Sigma^\omega)$. As stated before, it is straightforward that every implementation, for which the observable vocabulary is Σ, satisfies this property.

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 of σ and every continuation do not belong to ψ. Consequently, these continuations cannot belong to $E_f(\psi)$ nor $E(\psi)$ nor Π as well; Π is negatively determined by σ.

For obligation *r-properties*. Let Π be an obligation *r-property*, then Π can be expressed in a conjunctive normal form and in a disjunctive normal form (cf. Lemma 1).

- When Π is expressed in conjunctive normal form, then there exists $k \in \mathbb{N}^*$, s.t. Π is expressed $\bigcap_{i=1}^{k}(S_i(\psi_i) \cup G_i(\psi'_i))$ where $S_i(\psi_i)$ (resp. $G_i(\psi'_i)$) is a safety (resp. guarantee) *r-property* built upon ψ_i (resp. ψ'_i).
- When Π is expressed in a disjunctive normal form, then there exists $k \in \mathbb{N}^*$, s.t. Π is expressed $\bigcup_{i=1}^{k}(S_i(\psi_i) \cap G_i(\psi'_i))$ where $S_i(\psi_i)$ (resp. $G_i(\psi'_i)$) is a safety (resp. guarantee) *r-property* built upon ψ_i (resp. ψ'_i).

In order to show that a sequence, belonging to one of this predefined set, has all its finite and infinite continuations not satisfying the *r-property*, it suffices to realize an induction on k and use the previous reasoning applied to safety and guarantee *r-properties*.

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 is here that in order for the *r-property* to be negatively determined, it can have some prefixes in ψ. ∎

Theorem 3 (Automata-view of the testability of $\boldsymbol{Exec(\mathcal{P}_\Sigma) \subseteq \Pi}$): *For a Streett automaton $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ recognizing an r-property Π, according to the class of $\mathcal{A}_\Pi$, the testability conditions expressed in the automata view are given below. We say that the property recognized by*

- *a safety automaton is testable if $\overline{P} \neq \emptyset$;*
- *a guarantee automaton is testable if $\{q \in \overline{R} \mid \mathit{Reach}_{\mathcal{A}_\Pi}(q) \subseteq \overline{R}\} \neq \emptyset$;*
- *a k-obligation automaton is testable if $\bigcup_{i=1}^{k}(\overline{P_i} \cap \{q \in \overline{R_i} \mid \mathit{Reach}_{\mathcal{A}_\Pi}(q) \subseteq \overline{R_i}\}) \neq \emptyset$;*
- *a response automaton is testable if $\{q \in \overline{R} \mid \mathit{Reach}_{\mathcal{A}_\Pi}(q) \subseteq \overline{R}\} \neq \emptyset$;*

- *a persistence automaton is testable if* $\{q \in \overline{P} \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq \overline{P}\} \neq \emptyset$.

PROOF : Let first note that all these conditions are of the form $S \neq \emptyset$. The proof follows from the facts that we are considering complete and deterministic Streett automata and all states are reachable from the initial state. Then using the finite sequence acceptance criterion of Streett automata and the syntactic restrictions applying for automata in each *Safety-Progress* class, one can see that when a run of a sequence reaches a state in S, the property is negatively determined by the involved sequence. ■

Property 1 (Testability in language and automata view are equivalent) : *For an r-property* Π *recognized by a Streett automaton* $\mathcal{A}_\Pi$*, according to the class of* Π*, the testability conditions for* $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ *expressed in the language and automata views are equivalent.* ♢

Verdicts to deliver. We now state the verdicts that should be produced by a tester for the potential sequences of the IUT. Each testability condition in the language view is in the form $f(\{\psi_i\}_i) \neq \emptyset$ where the $\psi_i \subseteq \Sigma^*$ $(i \in [1, n])$ are used to build the *r-property* and f is a composition of set operations on ψ_i. For instance, for obligation *r-properties* expressed in conjunctive normal form, the testability condition is expressed $\bigcup_{i=1}^{k} \left(\overline{\psi_i} \cap \{\sigma \in \overline{\psi'_i} \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \overline{\psi'_i}\}\right) \neq \emptyset$. When $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \cap f(\{\psi_i\}_i)$, the test oracle should deliver *fail* since the underlying *r-property* is negatively determined. Conversely, when $\sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) \setminus f(\{\psi_i\}_i)$, the test oracle can deliver *unknown*.

In practice, those verdicts should be determined by a computable function, reading an interaction sequence, *i.e.,* a test oracle. In our framework, the test oracle is obtained from a Streett automaton[7] and is formally defined in the following property.

Property 2 (Test oracle for the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$) : *Given* $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q^{\mathcal{A}_\Pi}_{\mathrm{init}}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ *defining* Π*, the test oracle* $(Q^{\mathcal{O}}, q^{\mathcal{O}}_{\mathrm{init}}, \longrightarrow_{\mathcal{O}}, \Gamma^{\mathcal{O}})$ *for the relation* $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ *is defined as follows.* $Q^{\mathcal{O}}$ *is the smallest subset of* $Q^{\mathcal{A}_\Pi}$*, reachable from* $q^{\mathcal{O}}_{\mathrm{init}}$ *by* $\longrightarrow_{\mathcal{O}}$ *(defined below) with* $q^{\mathcal{O}}_{\mathrm{init}} = q^{\mathcal{A}_\Pi}_{\mathrm{init}}$.

- $\Gamma^{\mathcal{O}}$ *is defined as follows:*
  - *If* Π *is a pure safety, guarantee, obligation, or response property* $\Gamma^{\mathcal{O}}(q) =$ *fail if* $q \in \bigcup_{i=1}^{k}(\overline{P_i} \cap \{q \in \overline{R_i} \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq \overline{R_i}\}$ *and unknown otherwise,*
  - *If* Π *is a pure persistence property* $\Gamma^{\mathcal{O}}(q) =$ *fail if* $q \in \{q \in \overline{P} \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq \overline{P}\}$ *and unknown otherwise;*
- $\longrightarrow_{\mathcal{O}}$ *is defined as the smallest relation verifying:*
  - $q \xrightarrow{e}_{\mathcal{O}} q$ *if* $\exists e \in \Sigma, \exists q' \in Q^{\mathcal{O}} : q \xrightarrow{e}_{\mathcal{A}_\Pi} q'$ *and* $\Gamma^{\mathcal{O}}(q) =$ *fail,*
  - $\longrightarrow_{\mathcal{O}} = \longrightarrow_{\mathcal{A}_\Pi}$ *otherwise.* ♢

The proof of this property follows from Theorem 3 and Definition 10.

[7] The test oracle can be also obtained from the *r-properties* described in others views (language, logic). Indeed, in [FFM10] we describe how to express an *r-property* in the automata view from its expression in the language or the logic view.

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

Table 1: Summary of testability results wrt. the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$

Example and summary. We present an example illustrating the results presented so far for the testability of *r-properties* wrt. $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$.

Example 2 (Testability of some *r-properties* wrt. $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$) *We present the testability of three r-properties introduced in Example 1. The safety r-property Π_1 is testable wrt. the relation $Exec(\mathcal{P}_{\Sigma_1}) \subseteq \Pi_1$. Indeed in the language view, there are sequences belonging to $\overline{\psi_1}$ (the corresponding DFA has a non accepting state). In the automata view, we have sink $\in \overline{P}$ (reachable from the initial state). The guarantee r-property Π_2 is testable wrt. the relation $Exec(\mathcal{P}_{\Sigma_2}) \subseteq \Pi_2$. Indeed, there are sequences belonging to $\overline{\psi_2}$ s.t. all prefixes of theses sequences and all its continuations are also in $\overline{\psi_2}$. In the automata view, there is a (reachable) state in $\overline{R}$ from which all reachable states are in $\overline{R}$. The response r-property Π_3 is testable wrt. the relation $Exec(\mathcal{P}_{\Sigma_2}) \subseteq \Pi_3$. Indeed, there are sequences belonging to $\overline{\psi_2}$ s.t. all continuations of these sequences belong to $\overline{\psi_2}$. In the automata view, there is a (reachable) state in $\overline{R}$ from which all reachable states are in $\overline{R}$.*

Thus, we have clarified and extended some results of [NGH93]. 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[8], *i.e.,* some guarantee, obligation, response, and persistence *r-properties*. Finally, we provided testability conditions in the language and automata views.

5.2 Testability wrt. 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* can not 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 *fail* verdict would require determinism hypothesis on the implementation.

5.3 Testability wrt. the relation $Exec(\mathcal{P}_\Sigma) = \Pi$

Previous reasonings applied for the testability wrt. 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.,* $Exec(\mathcal{P}_\Sigma) \not\subseteq \Pi \Rightarrow Exec(\mathcal{P}_\Sigma) \neq \Pi$.

[8] In [NGH93], for this relation, only safety properties are declared as testable.

5.4 Testability wrt. 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), we have that if $\Pi = (A_f(\psi), A(\psi))$ then $\overline{\Pi} = (E_f(\overline{\psi}), E(\overline{\psi}))$. Furthermore, one have to notice that $\neg(Exec(\mathcal{P}_\Sigma) \subseteq \Pi) \Leftrightarrow Exec(\mathcal{P}_\Sigma) \cap \overline{\Pi} \neq \emptyset$.

As a consequence, testability results can be obtained in a rather straightforward manner for this relation. Full treatment can be found alternatively in Appendix A. Results are summarized in Table 2. Thus, we have extended and clarified some results of [NGH93]. Notably, we have shown that there exists one guarantee *r-property* for which it is not possible to obtain a *pass* verdict. Indeed, by duality, testing the guarantee *r-property* $(\emptyset, \emptyset)$ cannot lead to a *pass* verdict. Moreover, we have shown that some *r-properties* of the others classes are testable[9] as well, *i.e.,* some safety, obligation, response, and persistence *r-properties*. Finally, we provided testability conditions in the language and automata views.

Example 3 (Testability of some *r-properties* wrt. $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$) We present the testability of three *r-properties* introduced in Example 1 wrt. $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$.

The safety *r-property* Π_1 built from ψ_1, recognized by the Streett automaton depicted in Fig. 3b, is not testable wrt. the relation $Exec(\mathcal{P}_{\Sigma_1}) \subseteq \Pi_1$. Indeed, it does not satisfy the testability condition: the automaton recognizing ψ_1 does not have an accepting state reachable from the initial state s.t. it is reachable with accepting state and all reachable states are accepting ($\{\sigma \in \psi_1 \mid pref(\sigma) \cup cont(\sigma) \subseteq \psi_1\} = \emptyset$).

The guarantee *r-property* Π_2 built upon ψ_2, and represented on the Streett automaton in Fig. 4b, is testable wrt. the relation $Exec(\mathcal{P}_{\Sigma_2}) \cap \Pi_2 \neq \emptyset$. Indeed, it satisfies the testability conditions for guarantee properties: the automaton recognizing ψ_2 has a (reachable) accepting state ($\psi_2 \neq \emptyset$) and $R \neq \emptyset$ in $\mathcal{A}_{\Pi 2}$. The interesting sequences to be played to obtain a *pass* verdict are those leading to state 3.

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

6 Refining verdicts ?

Similarly to the introduction of weak truth values in runtime verification [BLS09, FFM09b, FFM10], 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 interaction sequence produced so far belongs (or not) to the property. The idea of satisfaction "if the program stops here" in runtime verification [BLS09, FFM09b] corresponds to the idea of "the test has shown enough on the implementation" in testing. In this case, testing would be similar to a

[9] In [NGH93], for this relation, only guarantee properties are declared as testable.

$Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$	Obtainable Verdicts	Testability condition (language view)	Testability condition (automata view)
Safety $(A_f(\psi), A(\psi))\ \lvert R = \emptyset, \overline{P} \nrightarrow P$	*pass*, *unknown*	$\{\sigma \in \psi \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \psi\}$	$\{q \in P \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq P\} \neq \emptyset$
Guarantee $(E_f(\psi), E(\psi))$	*pass*, *unknown*	$\psi \neq \emptyset$	$R \neq \emptyset$
Obligation $\bigcap_{i=1}^k (S_i(\psi_i) \cup G_i(\psi'_i))$ $\bigcup_{i=1}^k (S_i(\psi_i) \cap G_i(\psi'_i))$ $\overline{P_i} \nrightarrow P_i, R_i \nrightarrow \overline{R_i}$	*pass*, *unknown*	$\bigcap_{i=1}^k (\{\sigma \in \psi_i \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \psi_i\} \cup \psi'_i) \neq \emptyset$ $\bigcup_{i=1}^k (\{\sigma \in \psi_i \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \psi_i\} \cap \psi'_i) \neq \emptyset$	$\bigcup_{i=1}^k (\{q \in P_i \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq P_i\} \cap R_i) \neq \emptyset$
Response $(R_f(\psi), R(\psi))\ \lvert P = \emptyset$	*pass*, *unknown*	$\{\sigma \in \psi \mid cont^*(\sigma) \subseteq \psi\} \neq \emptyset$	$\{q \in R \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq R\} \neq \emptyset$
Persistence $(P_f(\psi), P(\psi))\ \lvert R = \emptyset$	*pass*, *unknown*	$\{\sigma \in \psi \mid cont^*(\sigma) \subseteq \psi\} \neq \emptyset$	$\{q \in P \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq P\} \neq \emptyset$

Table 2: Summary of testability results wrt. the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$

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 enhance confidence in the implementation compliance wrt. 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 $Exec(\mathcal{P}_\Sigma)$.

6.1 Revisiting testability wrt. the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$.

We have seen that, for $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$, 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.

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

For safety *r-properties*. Let Π be a safety *r-property*, then there exists $\psi \subseteq \Sigma^*$ s.t. Π can be expressed $(A_f(\psi), A(\psi))$. When the produced sequence belongs to $\{\sigma \in \psi \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \psi\}$, the tester can produce a weak *pass* verdict.

For guarantee *r-properties*. Let Π be a guarantee *r-property*, then there exists $\psi \subseteq \Sigma^*$ s.t. Π can be expressed $(E_f(\psi), E(\psi))$. It is possible to produce a weak *pass* verdict if the set ψ is not empty: guarantee *r-properties* are always positively determined when they are satisfied.

For obligation *r-properties*. Let Π be an m-obligation *r-property*.

- If for $m \in \mathbb{N}^*$, Π is expressed $\bigcap_{i=1}^{m}(S_i(\psi_i) \cup G_i(\psi'_i))$ where $S_i(\psi_i)$ (resp. $G_i(\psi'_i)$) is a safety (resp. guarantee) *r-property* built upon ψ_i (resp. ψ'_i), $i \in [1, m]$. The tester can produce a weak *pass* verdict when the interaction sequence belongs to $\bigcap_{i=1}^{m} \psi'_i$.

- If for $m \in \mathbb{N}^*$, Π is expressed $\bigcup_{i=1}^{m}(S_i(\psi_i) \cap G_i(\psi'_i))$ where $S_i(\psi_i)$ (resp. $G_i(\psi'_i)$) is a safety (resp. guarantee) *r-property* built upon ψ_i (resp. ψ'_i), $i \in [1, m]$. The tester can produce a weak *pass* verdict when the interaction sequence produced by the program belongs to $\bigcup_{i=1}^{m}(\{\sigma \in \psi_i \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \psi_i\} \cap \psi'_i)$.

For response and persistence *r-properties*. The reasoning is similar to the one used for safety *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. $(P_f(\psi), P(\psi))$). When the interaction sequence belongs to $\{\sigma \in \psi \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \psi\}$, the tester can produce a weak *pass* verdict.

Remark 3 When seeking *weak pass* verdicts, we try to determine positively the underlying *r-property*. Then the testability conditions for weaker verdicts exactly corresponds to the testability conditions for the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ without weak verdicts. $*$

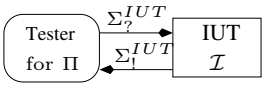


Figure 5: A generic test architecture

6.2 Revisit of testability wrt. the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$.

Similarly, for this relation, we use the duality inside the *Safety-Progress* classification. We have seen that the sole verdicts that can be produced were *pass* and *unknown*. The *fail* verdict that can be produced for this relation is still possible under the same conditions as for the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$. Indeed, in this case, one aims to find a sequence that negatively determines the *r-property*, *i.e.,* to play test sequences s.t. whatever the possible future continuation of the test would be, the *r-property* will not be satisfied.

7 Automatic test generation

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 detect verdicts for a relation between an *r-property* and an IUT. Before entering into the details of test generation, we first discuss informally some practical constraints that have to be taken into account for test generation. After that, we compute the canonical tester, discuss test selection, and show how quiescence can be taken into account in our framework.

7.1 General principles

We describe on the vocabulary Σ, according to the relation of interest, the sequences that should be played on the implementation in order to obtain the appropriate verdict and we discuss the test halting question.

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

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 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. 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 this verdict is 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 Section 7.3).

Vocabularies and test architecture. In order to address test generation, we will need to distinguish inputs and outputs and the vocabularies of the IUT and the *r-property*. The generic test architecture that we consider is depicted in Fig 5.

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 $\mathcal{I} = (Q^{\mathcal{I}}, q_{\text{init}}^{\mathcal{I}}, \Sigma^{IUT}, \longrightarrow_{\mathcal{I}})$.

7.1.2 For the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$

Similar arguments apply for this relation in order to find the sequences to play. The difference is here that we are seeking *pass* verdicts. In a similar way to the previous relation, it is possible to stop the test in the following cases. First, when a *pass* verdict is delivered. And second, when the *r-property* is negatively determined by the execution produced on the implementation or when there does not exist a continuation of this execution sequence s.t. the *r-property* is negatively determined.

7.2 Computation of the canonical tester.

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

We define canonical testers from Streett automata. To do so, we will use a set of subsets of Streett automaton states that we introduced in [FFM09b] for run-time verification. For a Streett m-automaton $\mathcal{A}_\Pi$, 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:

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

It is possible to show [FFM09b] 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.

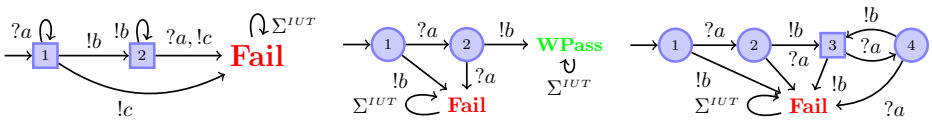
The canonical tester is $Test(\Pi) = \text{GenCan}(\mathcal{A}_\Pi)$ where the GenCan function is defined in Definition 11.

Definition 11 (Function GenCan). From a Streett m-automaton $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ defining a *testable r-property* Π, the function GenCan generates the IOLTS $T = (Q^T, q_{\text{init}}^T, \Sigma, \longrightarrow_T)$ defined as follows:

- $Q^T = B_c^{\mathcal{A}_\Pi} \cup G_c^{\mathcal{A}_\Pi} \cup \{Fail\} \cup \{WPass\}$ with $q_{\text{init}}^T = q_{\text{init}}^{\mathcal{A}_\Pi}$;
- $\longrightarrow_T$ is defined as follows:
  - $\forall e \in \Sigma : Fail \xrightarrow{e}_T Fail \wedge WPass \xrightarrow{e}_T WPass$,

- $q \xrightarrow{e}_T$ *Fail* if $q \xrightarrow{e}_{\mathcal{A}_\Pi} q' \wedge q' \in B^{\mathcal{A}_\Pi}$, for any $e \in \Sigma$,
- $q \xrightarrow{e}_T$ *WPass* if $q \xrightarrow{e}_{\mathcal{A}_\Pi} q' \wedge q' \in G^{\mathcal{A}_\Pi}$, for any $e \in \Sigma$,
- $q \xrightarrow{e}_T q'$ if $q \xrightarrow{e}_{\mathcal{A}_\Pi} q' \wedge q, q' \in G_c^{\mathcal{A}_\Pi} \cup B_c^{\mathcal{A}_\Pi}$, for any $e \in \Sigma$.

The GenCan function transforms a Streett automaton as follows. Transitions leading to a bad (resp. good) state are redirected to *Fail* (resp. *WPass*). Those latest states are terminal: the test can be stopped and the verdict produced.



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

Figure 6: Canonical testers built from the *r-properties* of Example 1

Example 4 (Canonical Testers) Canonical testers for *r-properties* of Example 1 are represented in Fig. 6. Canonical testers are built using the GenCan function. The alphabet partitioning is s.t. $\Sigma_?^{IUT} = \{?a\}$ and $\Sigma_!^{IUT} = \{!b, !c\}$.

The canonical tester built from the Streett safety automaton defining Π_1 is s.t. the state 3 (a bad state) is replaced by the state *Fail*.

The canonical tester built from the Streett guarantee automaton defining Π_2 is s.t. the state 5 (a bad state) is replaced by the state *Fail* and state 3 (a good state) is replaced with *WPass* (weak pass).

The canonical tester built from the Streett response automaton defining Π_3 is s.t. the state 5 (a bad state) is replaced by the state *Fail*.

7.3 Test selection

For a given *r-property*, the set of potential sequences to be played is infinite. In practice, one may use the underlying Streett automaton to constrain the states that should be visited during a test. Furthermore, as usual, one needs to select a test case that is *controllable* [JJ05]. It can be done on the canonical tester by first disabling input actions that do not permit to reach sought verdicts. Second, for a state in which several input actions are possible, one needs to generate different test cases with one input per state.

Test selection plays also a role to state *weak pass* verdicts. Indeed, when dealing with sequences satisfying a *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 is greater or equal than an expected number. 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 negatively determining it. Here, it thus remains to the tester expertise to state the halting criterion (possibly using quiescence, see Section 7.4).

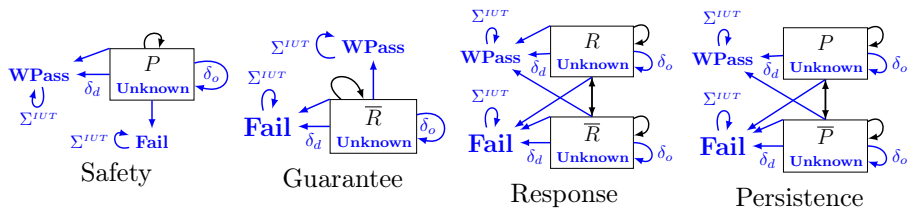


Figure 7: Schematic illustrations of the canonical tester for basic classes

7.4 Introducing quiescence ?

Quiescence [Tre96, JJ05] was introduced in conformance testing in order to represent IUT's inactivity. In practice, several kinds of quiescence may happen (see [JJ05] 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 finished or it is deadlocked. Thus, we introduce those two events in the output alphabet of the IUT. We have now the following additional alphabets: $\Sigma^{IUT}_{!,\delta} = \Sigma^{IUT}_{!} \cup \{\delta_o, \delta_d\}$, $\Sigma^{IUT}_{\delta} = \Sigma^{IUT}_{!,\delta} \cup \Sigma^{IUT}_{?}$.

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 $Exec(\mathcal{P}_{\Sigma^{IUT}}) \subseteq (\Sigma^{IUT}_{\delta})^{\infty}$. The set of interactions of the tester with the IUT is $Inter(\Sigma^{IUT}) \subseteq (\Sigma^{IUT} + \delta_o)^* \cdot (\delta_d + \epsilon)$. The interactions are sequences in which the tester can observe IUT's outputlocks and finishes by the observation of a deadlock or program termination. When considering quiescence, characterizing testable properties now consists in comparing the set of interactions to the set of sequences described by the *r-property*. The intuitive ideas are the following:

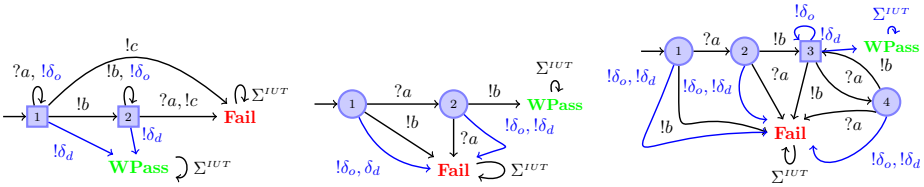
- The tester can observe finished executions of the IUT with δ_d. In this case, the IUT has played a finite sequence. In some sense, the played sequence determines negatively or positively the *r-property* depending on whether or not it satisfied the *r-property*.
- The tester can decide to terminate 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 determine negatively in some sense 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.

The notion of positive and negative determinacy is now modified in the context of quiescence as follows. We say that the *r-property* Π is negatively determined upon quiescence by the sequence $\sigma \in Inter(\mathcal{P}_{\Sigma^{IUT}})$ (denoted $\ominus$–*determined*–$q(\sigma, \Pi)$) if $\ominus$–*determined*$(\sigma_{\downarrow\Sigma^{IUT}}, \Pi) \vee (|\sigma| > 1 \wedge last(\sigma) \in \{\delta_d, \delta_o\} \wedge \neg\Pi((\sigma_{\ldots|\sigma|-2})_{\downarrow\Sigma^{IUT}})$, where $\sigma_{\downarrow\Sigma^{IUT}}$ is the projection of σ on Σ^{IUT}.

For the proposed approach, the usefulness of quiescence lies in the fact that the current test sequence does not have any continuation. Consequently the testability conditions may be weakened. Indeed, when one has determined that the current interaction with the IUT is over, it is not necessary to require that

$Exec(\mathcal{P}_\Sigma) \subseteq \Pi$	Possible Verdicts	Testability Condition
Safety $(A_f(\psi), A(\psi))$	*fail*, *unknown*	$\overline{\psi} \neq \emptyset$
Guarantee $(E_f(\psi), E(\psi))$	*fail*, *unknown*	$\{\sigma \in \overline{\psi} \mid pref(\sigma) \subseteq \overline{\psi}\}\} \neq \emptyset$
Obligation $\bigcap_{i=1}^{k}(S_i(\psi_i) \cup G_i(\psi'_i))$ $\bigcup_{i=1}^{k}(S_i(\psi_i) \cap G_i(\psi'_i))$	*fail*, *unknown*	$\bigcup_{i=1}^{k}\left(\overline{\psi_i} \cap \{\sigma \in \overline{\psi'_i} \mid pref(\sigma) \subseteq \overline{\psi'_i}\right) \neq \emptyset$ $\bigcap_{i=1}^{k}\left(\overline{\psi_i} \cup \{\sigma \in \overline{\psi'_i} \mid pref(\sigma) \subseteq \overline{\psi'_i}\} \neq \emptyset\right.$
Response $(R_f(\psi), R(\psi))$	*fail*, *unknown*	$\overline{\psi} \neq \emptyset$
Persistence $(P_f(\psi), P(\psi))$	*fail*, *unknown*	$\overline{\psi} \neq \emptyset$

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



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

Figure 8: Canonical testers for the *r-properties* of Example 1 with quiescence

the *r-property* should be evaluated in the same way. In some sense, it amounts to consider that the evaluation produced by the last event before observing quiescence "finishes" the execution sequence. That is to say, if the *r-property* was satisfied (resp. not satisfied) by the last observed sequence, then the *r-property* is positively (resp. negatively) determined by the observed sequence.

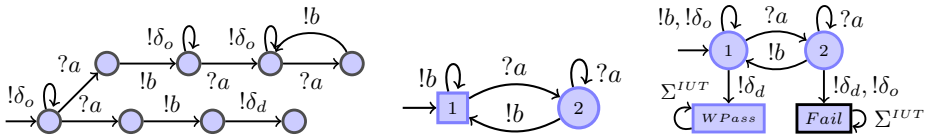
Revisiting previous results. With quiescence, the purpose of the tester is now to "drive" the IUT in a state in which the underlying *r-property* is not satisfied, and then observe quiescence. Informally, the testability condition relies now on the existence of a sequence s.t. the *r-property* is not 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 as shown in Table 3.

The canonical tester construction is also updated by adding the following rules for $\longrightarrow_T$: $\forall q \in B_c^{\mathcal{A}_\Pi} : q \xrightarrow{\delta_o,\delta_d}_T Fail$, $\forall q \in G_c^{\mathcal{A}_\Pi} : q \xrightarrow{\delta_o}_T q \wedge q \xrightarrow{\delta_d}_T WPass$. Illustrations of the construction of the canonical tester for basic classes with quiescence is given in Fig. 7.

Example 5 (Canonical Testers with quiescence) Canonical testers with quiescence for *r-properties* of Example 1 are represented in Fig. 8. Canonical testers are built using the updated version of the GenCan function. The IOLTs have now two new elements in their alphabet representing quiescence: δ_o and δ_d.

From the canonical testers depicted in Fig. 6, additional transitions regarding quiescence are added following the rules described previously.

We illustrate in the following example the usefulness of quiescence. This example shows how weaker testability conditions with quiescence can leverage the testability of properties and detect additional misbehaviors of implementations.



(a) An IUT (b) An *r-property* (c) Canonical tester

Figure 9: Illustrating the usefulness of quiescence

Example 6 (Usefulness of Testability with quiescence) *Consider the IUT depicted in Fig. 9a with observable actions* $\Sigma_?^{IUT} = \{?a\}$ *and* $\Sigma_!^{IUT} = \{!b\}$*. This IUT waits for an* $?a$*, produces a* $!b$*, and then non deterministically finishes 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 ("?" and "!" are not represented and* $x^{\&}$ *stands for* $x + \epsilon$*):*

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

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

Now let us consider the response r-property defined by the Streett automaton depicted in Fig. 9b. Its vocabulary is $\{?a, !b\}$*, it has one recurrent state (*$R = \{1\}$*) and no persistent state (*$P = \emptyset$*). The underlying r-property states that "every input* $?a$ *should be acknowledged by an output* $!b$*". This r-property is not testable under the conditions expressed in Section 5 (i.e., with "strong" verdicts): the response automaton defining it does not have a (reachable) state* q *in* $\overline{R}$ *s.t. all states that can be reached from* q *are in* $\overline{R}$*. However, this r-property is testable with quiescence. Furthermore, one can observe that* $Inter(\mathcal{P}_{\Sigma^{IUT}}) \not\subseteq \Pi$ *because the existence of* $?a{\cdot}!b{\cdot}?a{\cdot}!\delta_o$ *in* $Inter(\mathcal{P}_{\Sigma^{IUT}})$*. The synthesized canonical tester is depicted in Fig. 9c.*

Revisiting testability wrt. $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$**.** Testability conditions for this relation can be updated in a similar way. In this case, one is interested in seeking sequences satisfying the *r-property*.

8 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 help test designers to enhance testing stages with the use of properties. This prototype can be freely downloaded at the following address: http://www.irisa.fr/prive/yfalcone/software.php.

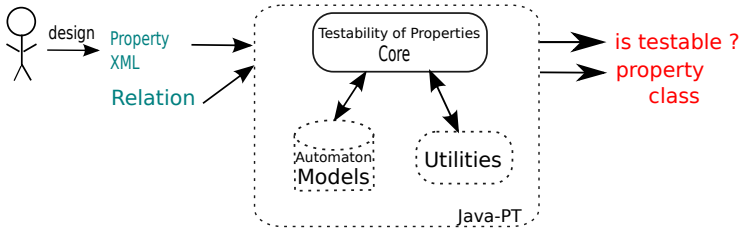


Figure 10: Overview of Java-PT

8.1 Overview

The tool is developed in the Java programming language and uses XML[10], XSLT[11], XStream[12] as underlying supporting technologies.

An overview of the architecture of Java-PTis given in Fig. 10. In the following 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 Section 8.2 for examples).

A model for automata-based objects. The tool Automaton Models of Java-PT consists of a hierarchy of classes modelling several entities based on automata and which are used by the various tools of Java-PT. We provide only an abstract description. We have modelled the various notions of an automaton using a set of Java classes: alphabet, states, and their transitions. Those classes are used at several levels in the tool.

We also have implemented a component which provides means to make objects persistent in XML. This utility consists in configuring and customizing the XStream library to realize serialization and deserialization.

A set of 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 in an automaton.

The main module: Testability of Properties core. The main module is the "Testability of Properties core". This modules leverages the modules *Automaton Models* and *Utilities*. It consists mainly in implementing the testability conditions given in Tables 1 and 2.

8.2 Examples

In this section, we present some examples of properties processed with Java-PT. Examples of this paper and more examples can be found in the distribution of Java-PT.

[10]Extensible Markup Language - http://www.w3.org/XML/
[11]The Extensible Stylesheet Language Family - http://www.w3.org/Style/XSL/
[12]http://xstream.codehaus.org/

```
<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>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>Pi2</Description>
  <AcceptingCondition>
    <Pair P="" R="3"/>
  </AcceptingCondition>
</StreettAutomaton>
```

Figure 11: Defining Π_1 (left) and Π_2 (right) in XML

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

Figure 12: Processing Π_1 and Π_2 with Java-PT

Let us come back on the *r-properties* Π_1 and Π_2 of Example 1. As seen in Section 5, these property are testable wrt. the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$. Those *r-properties* are defined by their Streett automata described by the XML files in Fig. 11. Processing these properties with Java-PT is as represented in Fig. 12.

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. More specifically, we first relate some testing approaches in which a property plays a major role. Then, since the testing approach considered here does not use an executable specification, we recall some previous related approaches. Then, since the major originality of this paper is the use of the *Safety-Progress* classification, we overview the previous uses of this classification in validation techniques. Finally, we propose a discussion on the results afforded by this paper.

9.1 Testing with properties

We overview some approaches related to the topic of this paper and dedicated to testing in which the behavior described by the property is tested or in which a property is used to help the test selection.

Testing oriented by properties for generating test purposes. One of the limits of conformance testing [Tre96] 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 as a complement 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 [JJ05, dV01]. For instance, in [FMP03], 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 [MSM07].

Combining testing and formal verification. In [CJMR07], the complementarity between verification techniques and conformance testing is studied. Notably, the authors shown that it is possible to detect (using testing) violations of safety (resp. satisfaction of co-safety) properties on the implementation and the specification.

Test of security policies. Last years, several approaches were proposed for testing security policies by extending classical conformance testing approaches.

For instance, in [TMB07, PMT08], the functional model of the system is defined using contracts which are use-cases enhanced with pre and post conditions. The security policy is expressed in an access control model. Test cases are derived in order to test the security and the functional features of the system in a complementary fashion.

In [MOC+07], the authors propose to integrate an access control policy to a specification expressed by an extended finite-state machine (EFSM).

In [MMC08b], the authors are interested in the test of security policies for Web services expressed in temporal logic with deontic features. The approach consists in integrating logical formulae in the specifications of Web services based on communicating automata.

Some approaches of passive testing (*e.g.*, [MMC08a, MWC08]), similar to offline runtime verification were used to verify the conformance of network traffic wrt. functional or security requirements.

In [MDJ09], the authors are interested in testing opacity properties which are aiming to improve the confidentiality of systems. In the context of enforcing opacity via access control mechanisms, the authors show how to derive tests in order to detect violation of the conformance of an access control implementation to its specification.

Requirement-Based testing. In requirement-based testing, the purpose is to generate a test suite from a set of informal requirements. For instance,

in [RWH07, WRHM06, PRB09], 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.

9.2 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 [DFG+06] and formalized using a process algebra dedicated to testing [FFMR06]. Later [FFMR07], Falcone et al. have shown that this approach can be generalized to general formalisms (LTL and extended regular expressions) and that the test can be executed in a decentralized fashion. In [DRG08], Darmaillacq et al. provided a case study dedicated to test security policies of networks.

9.3 Using the *Safety-Progress* classification in validation techniques

The *Safety-Progress* classification of properties is rarely used in validation techniques. To the best of the authors knowledge, this classification was used in two kind of approaches: runtime validation techniques and model-checking.

In our previous work [FFM09b, FFM08, FFM09a], we used the *Safety-Progress* classification properties to characterize the sets of properties that can be verified and enforced during the runtime of systems. In some sense, this previous endeavor similarly addressed the expressiveness question for runtime verification and runtime enforcement. For runtime verification, relying on a notion of property monitorability (*i.e.,* the capability of being verified at runtime) parametrized by a truth-domain, we determined [FFM09b] the classes of monitorable properties within the *Safety-Progress* classification for several truth-domains of interest. For runtime enforcement, we characterized the set of enforceable properties independently from any enforcement mechanism. Our results [FFM08, FFM09a] generalized and extended previous ones in the field of runtime enforcement.

In [CP03], 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. To 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.

9.4 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 [CJMR07]. Furthermore, this approach leverages the use of test purposes [KGHS98, JJ05] 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 [MP90, CMP92a] 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, classical conformance testing fall in the scope of the proposed framework. Indeed, suspended traces of an implementation preserving the *ioco* relation wrt. a given specification can be expressed as a safety property [CJMR07].

10 Conclusion and perspectives

Conclusion. In this paper, we study the space 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 *r-property*. Leveraging the notions of positive and negative determinacy of properties, we have identified for each *Safety-Progress* class and according to the relation of interest, the testable fragment. Moreover we have seen that the framework of *r-properties* in the *Safety-Progress* classification provides a decidable test oracle in order to produce a verdict depending on the 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. Finally, all results of this paper are implemented in a prototype tool for which a description is given.

Perspectives. A first research direction is to investigate the set of testable properties for more expressive formalisms. Indeed, the *Safety-Progress* classification is concerned with regular properties, and classifying testable properties for *e.g.*, context-free properties would be of interest.

Another perspective is to combine the approach proposed with weak verdicts to a notion of *test coverage*. The various approaches [RWH07, WRHM06, PRB09] for defining test coverage for property-oriented testing could be used to reinforce a set of weak verdicts.

References

[AS84] Bowen Alpern and Fred B. Schneider. Defining Liveness. Technical report, Cornell University, Ithaca, NY, USA, 1984.

[BLS09] Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing LTL Semantics for Runtime Verification. *Journal of Logic and Computation*, 2009.

[CJMR07] 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.

[CMP92a] Edward Chang, Zohar Manna, and Amir Pnueli. Characterization of Temporal Property Classes. In *Automata, Languages and Programming*, pages 474–486, 1992.

[CMP92b] Edward Chang, Zohar Manna, and Amir Pnueli. The Safety-Progress Classification. Technical report, Stanford University, Dept. of Computer Science, 1992.

[CP03] Ivana Cerná and Radek Pelánek. Relating Hierarchy of Temporal Properties to Model Checking. In Branislav Rovan and Peter Vojtás, editors, *MFCS*, volume 2747 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 2003.

[DFG+06] Vianney Darmaillacq, Jean-Claude Fernandez, Roland Groz, Laurent Mounier, and Jean-Luc Richier. Test Generation for Network Security Rules. In *Testing of Communicating Systems, 18th IFIP TC6/WG6.1 International Conference, TestCom 2006*, volume 3964 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2006.

[DRG08] 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.

[dV01] R. G. de Vries. Towards formal test purposes. In Jan Tretmans and Ed Brinksma, editors, *Formal Approaches to Testing of Software 2001 (FATES'01), Aarhus, Denmark*, volume NS-01-4 of *BRICS Notes Series*, pages 61–76, Aarhus, Denkmark, August 2001.

[FFM08] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Synthesizing Enforcement Monitors wrt. the Safety-Progress Classification of Properties. In *ICISS '08: Proceedings of the 4th International Conference on Information Systems Security*, pages 41–55, Berlin, Heidelberg, 2008. Springer-Verlag.

[FFM09a] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Enforcement Monitoring wrt. the Safety-Progress Classification of Properties. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 593–600. ACM, 2009.

[FFM09b] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime Verification of Safety-Progress Properties. In Saddek Bensalem and Doron Peled, editors, *RV*, volume 5779 of *Lecture Notes in Computer Science*, pages 40–59. Springer, 2009.

[FFM10] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can You Verify and Enforce at Runtime? Technical Report TR-2010-5, Verimag Research Report, 2010.

[FFMR06] Yliès Falcone, Jean-Claude Fernandez, Laurent Mounier, and Jean-Luc Richier. A Test Calculus Framework Applied to Network Security Policies. In *FATES 2006 and RV 2006: First Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification, Revised Selected Papers*, pages 55–69, 2006.

[FFMR07] Yliès Falcone, Jean-Claude Fernandez, Laurent Mounier, and Jean-Luc Richier. A Compositional Testing Framework Driven by Partial Specifications. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 2007.

[FMFR10] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez, and Jean-Luc Richier. Runtime Enforcement Monitors: composition, synthesis, and enforcement abilities, 2010. Under revision at FMSD: Formal Methods in System Design. A preliminary version version of this paper appeared as Verimag Technical Report TR-2008-7.

[FMP03] Jean-Claude Fernandez, Laurent Mounier, and Cyril Pachon. Property Oriented Test Case Generation. In *FATES'03: Proceedings of the Third International Workshop on Formal Approaches to Testing of Software*, pages 147–163, 2003.

[Gra94] 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.

[JJ05] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(4):297–315, 2005.

[KGHS98] Beat Koch, Jens Grabowski, Dieter Hogrefe, and Michael Schmitt. Autolink: A Tool for Automatic Test Generation from SDL Specifications. *Industrial-Strength Formal Specification Techniques*, 0:114, 1998.

[Lam77] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, 1977.

[MDJ09] Hervé Marchand, Jérémy Dubreil, and Thierry Jéron. Automatic testing of access control for security properties. In *TESTCOM '09/FATES '09: Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop*, pages 113–128. Springer-Verlag, 2009.

[MMC08a] Wissam Mallouli, Amel Mammar, and Ana R. Cavalli. Modeling System Security Rules with Time Constraints Using Timed Extended Finite State Machines. In David Roberts, Abdulmotaleb El-Saddik, and Alois Ferscha, editors, *DS-RT*, pages 173–180. IEEE Computer Society, 2008.

[MMC08b] Wissam Mallouli, Gerardo Morales, and Ana Cavalli. Testing Security Policies for Web Applications. In *ICSTW '08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 269–270. IEEE Computer Society, 2008.

[MOC+07] 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.

[MP90] Zohar Manna and Amir Pnueli. A Hierarchy of Temporal Properties (invited paper, 1989). In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 377–410. ACM, 1990.

[MSM07] Patricia D. L. Machado, Daniel A. Silva, and Alexandre C. Mota. Towards Property Oriented Testing. *Electron. Notes Theor. Comput. Sci.*, 184:3–19, 2007.

[MWC08] Wissam Mallouli, Bachar Wehbi, and Ana R. Cavalli. Distributed Monitoring in Ad Hoc Networks: Conformance and Security Checking. In *ADHOC-NOW*, volume 5198 of *Lecture Notes in Computer Science*, pages 345–356. Springer, 2008.

[NGH93] Robert Nahm, Jens Grabowski, and Dieter Hogrefe. Test Case Generation for Temporal Properties. Technical report, Bern University, 1993.

[PMT08] Alexander Pretschner, Tejeddine Mouelhi, and Yves Le Traon. Model-Based Tests for Access Control Policies. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 338–347. IEEE Computer Society, 2008.

[PRB09] Charles Pecheur, Franco Raimondi, and Guillaume Brat. A Formal Analysis of Requirements-based Testing. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 47–56, New York, NY, USA, 2009. ACM.

[PZ06] Amir Pnueli and Aleksandr Zaks. PSL Model Checking and Run-Time Verification Via Testers. In *FM06: Proceedings of the* 14th *Int Symp. on Formal Methods*, pages 573–586, 2006.

[RWH07] Ajita Rajan, Michael Whalen, and Mats P.E. Heimdahl. Model Validation using Automatically Generated Requirements-Based Tests. In *High Assurance Systems Engineering Symposium, 2007. HASE '07. 10th IEEE*, pages 95–104, Nov. 2007.

[Str81] Robert S. Streett. Propositional Dynamic Logic of looping and converse. In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 375–383, New York, NY, USA, 1981. ACM.

[TMB07] Yves Le Traon, Tejeddine Mouelhi, and Benoit Baudry. Testing Security Policies: Going Beyond Functional Testing. *Software Reliability Engineering, International Symposium on*, 0:93–102, 2007.

[Tre96] Jan Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer-Verlag, 1996.

[WRHM06] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage Metrics for Requirements-based Testing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 25–36. ACM, 2006.

A Testability wrt. the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$

We study the testability of *r-properties* wrt. the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$.

Obtainable verdicts and sufficient condition. The only verdicts that can be possibly obtained for this relation are *pass* and *unknown*.

A *fail* verdict would mean that $\Pi \cap Exec(\mathcal{P}_\Sigma) = \emptyset$. It is impossible to obtain such a verdict in the general case[13]. Even if one finds a set of execution sequences in $\mathcal{P}_\Sigma$ which does not belong to Π, one cannot safely state that there does not exist another execution of the implementation exhibiting an execution sequence belonging to Π. Note that, similarly to the testability of $Exec(\mathcal{P}_\Sigma) \subseteq Pi$ and dually, the *r-property false*, *i.e.,* $(\emptyset, \emptyset)$ is vacuously testable.

A *pass* verdict means that $\Pi \cap Exec(\mathcal{P}_\Sigma) \neq \emptyset$. In order to produce such a verdict, one needs to find an execution sequence $\sigma \in \Sigma^*$ which belongs to Π, and s.t. all extensions belong also to Π. In others words, one needs to exhibit an execution sequence of $\mathcal{P}_\Sigma$ s.t. Π is positively determined by this sequence.

For this relation, we will be thus only interested in seeking *pass* verdicts under the condition expressed by the following theorem:

Theorem 4 (Condition to determine a *pass* verdict for $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$): *The relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ is verified if there exists a sequence s.t. the r-property Π is positively determined. More formally:*

$$\exists \sigma \in Exec_{\mathrm{f}}(\mathcal{P}_\Sigma) : \oplus\text{–}determined(\sigma, \Pi) \Rightarrow verdict(\sigma, Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset) = pass \tag{2}$$

Proof : The proof of this theorem is similar to the proof of Theorem 1. Indeed, if we can find an execution sequence of the implementation which has all continuations satisfying the *r-property* Π, then this sequence shows that the implementation and the property have a common sequence. ∎

[13] Let us recall that we set ourselves in the case in which we do not have the source code of the tested implementation nor specification.

Testability of this relation in the *Safety-Progress* classification. We now establish for each class of properties the conditions under which the left part of the implication (in Theorem 4) holds. Using the previous theorem, it gives the existence of an execution sequence giving a *pass* verdict for the relation.

Theorem 5 (Language-view of the testability of $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$): *For each Safety-Progress class, the testability conditions in order to obtain a pass verdict and expressed in the language view are the following:*

- *a safety r-property* $(A_f(\psi), A(\psi))$ *is testable if* $\{\sigma \in \psi \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \psi\}$*;*
- *a guarantee r-property* $(E_f(\psi), E(\psi))$ *is testable if* $\psi \neq \emptyset$*;*
- *an k-obligation r-property built over* ψ_i *(resp.* ψ'_i*)*
  - *and expressed in conjunctive normal form is testable if* $\bigcap_{i=1}^{k}(\{\sigma \in \psi_i \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \psi_i\} \cup \psi'_i) \neq \emptyset$*,*
  - *and expressed in disjunctive normal form is testable if* $\bigcup_{i=1}^{k}(\{\sigma \in \psi_i \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \psi_i\} \cap \psi'_i) \neq \emptyset$*;*
- *a response (resp. persistence) r-property* $(R_f(\psi), R(\psi))$ *(resp.* $(P_f(\psi), P(\psi))$*) is testable if* $\{\sigma \in \psi \mid cont^*(\sigma) \subseteq \psi\} \neq \emptyset$*.*

Proof : We prove the testability conditions for each *Safety-Progress* class.

- For safety *r-properties*. Let Π be a safety *r-property*, then there exists $\psi \subseteq \Sigma^*$ s.t. Π can be expressed $(A_f(\psi), A(\psi))$. This *r-property* is testable if the set $\{\sigma \in \psi \mid pref(\sigma) \cup cont^*(\sigma) \subseteq \psi\}$ is not empty. The sequences belonging to this set are in ψ and all its extensions also are. According to the definition of safety *r-properties*, those sequences and all their extensions belong to Π.
- For guarantee *r-properties*. Let Π be a guarantee *r-property*, then there exists $\psi \subseteq \Sigma^*$ s.t. Π can be expressed $(E_f(\psi), E(\psi))$. This *r-property* is testable if the set ψ is not empty. Indeed, let $\sigma \in \psi$, then according to the definition of guarantee *r-properties*, a sequence belonging to ï¿½ ψ has all its finite (resp. infinite) extensions belonging to $E_f(\psi)$ (resp. $E(\psi)$).
- For obligation *r-properties*. Let Π an obligation *r-property*, then Π can be expressed in conjunctive or disjunctive normal form (cf. Lemma 1).
  - If Π is expressed in a conjunctive normal form, then there exists $k \in \mathbb{N}$, s.t. Π is expressed $\bigcap_{i=1}^{k}(S_i(\psi_i) \cup G_i(\psi'_i))$ where $S_i(\psi_i)$ (resp. $G_i(\psi'_i)$) is a safety (resp. guarantee) *r-property* built upon ψ_i (resp. ψ'_i). This *r-property* is testable if the set $\bigcap_{i=1}^{k}(\{\sigma \in \psi_i \mid cont^*(\sigma) \subseteq \psi_i\} \cup \psi'_i)$ is not empty.
  - If Π is expressed in a disjunctive normal form, then there exists $k \in \mathbb{N}$, s.t. Π is expressed $\bigcup_{i=1}^{k}(S_i(\psi_i) \cap G_i(\psi'_i))$ where $S_i(\psi_i)$ (resp. $G_i(\psi'_i)$) is a safety (resp. guarantee) *r-property* built upon ψ_i (resp. ψ'_i). This *r-property* is testable if the set $\bigcup_{i=1}^{k}(\{\sigma \in \psi_i \mid cont^*(\sigma) \subseteq \psi_i\} \cap \psi'_i)$ is not empty.

In order to prove that a sequence belonging to one these defined sets has all its finite and infinite continuations satisfying a k-obligation *r-property*, it suffices to make an induction on k and use the reasoning used for safety and guarantee *r-properties*.

- For response and persistence *r-properties*. The reasoning is similar to the one used for safety *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 $\{\sigma \in \psi \mid cont^*(\sigma) \subseteq \psi\}$ is not empty. Although, the difference is the following: for the *r-property* to be positively determined, it can have sequences which are not in ψ. ∎

For each class of properties, under the expressed conditions, if one manages to play a sequence of one of these sets, it is possible to state a *pass* verdict for the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$.

Theorem 6 (Automata-view of the testability of $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$): *Given a Streett automaton $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\text{init}}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$ recognizing an r-property Π, according to the class of $\mathcal{A}_\Pi$, the testability conditions expressed in the automata view are given below. We say that the property recognized by*

- *a safety automaton is testable if $\{q \in P \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq P\} \neq \emptyset$;*
- *a guarantee automaton is testable if $R \neq \emptyset$;*
- *a k-obligation obligation automaton is testable if $\bigcup_{i=1}^{k}(\{q \in P_i \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq P_i\} \cap R_i) \neq \emptyset$;*
- *a response automaton is testable if $\{q \in R \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq R\} \neq \emptyset$;*
- *a persistence automaton is testable if $\{q \in P \mid Reach_{\mathcal{A}_\Pi}(q) \subseteq P\} \neq \emptyset$.*

Proof : The proof is similar to the proof of Theorem 3, except that when a run of a sequence reaches one of the exhibited set, the underlying *r-property* is positively determined. ∎

Property 3 (Testability in language and automata view are equivalent) : *For an r-property Π recognized by a Streett automaton $\mathcal{A}_\Pi$, according to the class of Π, the testability conditions for $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ expressed in the language and automata view are equivalent.* ♢

Verdicts to deliver. In a similar way to the previous relation, the testability conditions expressed above are of the form

$$f(\psi_1, \ldots, \psi_n) \neq \emptyset$$

where the ψ_i are finitary properties used to build the *r-property*. The membership test for a sequence of $f(\psi_1, \ldots, \psi_n)$ is decidable. Consequently, from these finitary properties ψ_i used to define an *r-property*, it is possible to define a computable oracle allowing to deliver a verdict for the execution sequences of an implementation:

When an execution sequence $\sigma \in \mathit{Exec}_{\mathrm{f}}(\mathcal{P}_\Sigma)$ belongs to $f(\psi_1, \ldots, \psi_n)$, the test oracle delivers a *pass* verdict for this relation.

Conversely, when an execution sequence $\sigma \in \mathit{Exec}_{\mathrm{f}}(\mathcal{P}_\Sigma)$ does not belong to $f(\psi_1, \ldots, \psi_n)$, the test oracle can deliver an *unknown* verdict for this relation. A remaining question is when the tester should decide to deliver the definitive verdict and stop the test.

Those verdicts are determined by the test oracle which is obtained from a Streett automaton and is formally defined as follows:

Definition 12 (Test oracle for the relation $\mathit{Exec}(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$). Given an *r-property* Π recognized by a Streett automaton $\mathcal{A}_\Pi = (Q^{\mathcal{A}_\Pi}, q_{\mathrm{init}}^{\mathcal{A}_\Pi}, \Sigma, \longrightarrow_{\mathcal{A}_\Pi}, \{(R_1, P_1), \ldots, (R_m, P_m)\})$, the test oracle $(Q^{\mathcal{O}}, q_{\mathrm{init}}^{\mathcal{O}}, \longrightarrow_{\mathcal{O}}, \Gamma^{\mathcal{O}})$ for the relation $\mathit{Exec}(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ is defined as follows:

- $Q^{\mathcal{O}}$ is the smallest subset of $Q^{\mathcal{A}_\Pi}$, reachable from $\longrightarrow_{\mathcal{O}}$ (defined below);
- $q_{\mathrm{init}}^{\mathcal{O}} = q_{\mathrm{init}}^{\mathcal{A}_\Pi}$;
- $\Gamma^{\mathcal{O}}$ is defined as follows:
  - If Π is a pure safety, guarantee, obligation, or persistence property: $\Gamma^{\mathcal{O}}(q) = \mathit{pass}$ if $q \in \bigcup_{i=1}^{k}(\{q \in P_i \mid \mathit{Reach}_{\mathcal{A}_\Pi}(q) \subseteq P_i\} \cap R_i) \neq \emptyset$ and *unknown* otherwise;
  - If Π is a pure response property $\Gamma^{\mathcal{O}}(q) = \mathit{pass}$ if $q \in \{q \in R \mid \mathit{Reach}_{\mathcal{A}_\Pi}(q) \subseteq R\}$ and *unknown* otherwise;
- $\longrightarrow_{\mathcal{O}}$ is defined as the smallest relation verifying:
  - $q \xrightarrow{e}_{\mathcal{O}} q$ if $\exists e \in \Sigma, q' \in Q^{\mathcal{O}} : q \xrightarrow{e}_{\mathcal{A}_\Pi} q'$ and $\Gamma^{\mathcal{O}}(q) = \mathit{pass}$,
  - $\longrightarrow_{\mathcal{O}} = \longrightarrow_{\mathcal{A}_\Pi}$ else.

Summary. Testability results for the relation $\mathit{Exec}(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ are summarized in Table 2. Thus, we have extended and clarified some results of [NGH93]. Notably, we have shown that there exists one guarantee *r-property* which is not testable. Moreover, we have shown that some *r-properties* of the others classes are testable as well.

Contents

1 Introduction **3**

2 Preliminaries and notations **4**
2.1 Sequences, and execution sequences 4
2.2 IOLTSs 5
2.3 Properties 5

3 A SP classification for runtime techniques **6**
3.1 The language-theoretic view of *r-properties* 7
3.1.1 Construction of *r-properties* 7
3.1.2 Some useful facts in the language view 8
3.2 The automata view of *r-properties* [FFM09b] 9
3.2.1 Streett automata 9
3.2.2 The hierarchy of automata. 10
3.3 Examples and summary 11

4 Some notions of testability **12**

5 Characterizing testable properties without executable specification **14**
5.1 Testability wrt. the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ 14
5.2 Testability wrt. the relation $\Pi \subseteq Exec(\mathcal{P}_\Sigma)$ 19
5.3 Testability wrt. the relation $Exec(\mathcal{P}_\Sigma) = \Pi$ 19
5.4 Testability wrt. the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ 20

6 Refining verdicts ? **20**
6.1 Revisiting testability wrt. the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$. 22
6.2 Revisit of testability wrt. the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$. 23

7 Automatic test generation **23**
7.1 General principles 23
7.1.1 For the relation $Exec(\mathcal{P}_\Sigma) \subseteq \Pi$ 23
7.1.2 For the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ 24
7.2 Computation of the canonical tester. 24
7.3 Test selection 25
7.4 Introducing quiescence ? 26

8 Implementation: Java-PT **28**
8.1 Overview 29
8.2 Examples 29

9 Related work and discussion **30**
9.1 Testing with properties 31
9.2 Property testing without a behavioral specification 32
9.3 Using the *Safety-Progress* classification in validation techniques . 32
9.4 Discussion 32

10 Conclusion and perspectives **33**

A Testability wrt. the relation $Exec(\mathcal{P}_\Sigma) \cap \Pi \neq \emptyset$ 37

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399