Integrating Formal Verification and Conformance Testing for Reactive Systems

Camille Constant, Thierry Jéron, Hervé Marchand, Vlad Rusu

Integrating Formal Verification and Conformance Testing for Reactive Systems

Camille Constant, Thierry Jéron, Hervé Marchand, and Vlad Rusu

Abstract—In this paper, we describe a methodology integrating verification and conformance testing. A specification of a system—an extended input-output automaton, which may be infinite-state—and a set of safety properties ("nothing bad ever happens") and possibility properties ("something good may happen") are assumed. The properties are first tentatively verified on the specification using automatic techniques based on approximated state-space exploration, which are sound, but, as a price to pay for automation, are not complete for the given class of properties. Because of this incompleteness and of state-space explosion, the verification may not succeed in proving or disproving the properties. However, even if verification did not succeed, the testing phase can proceed and provide useful information about the implementation. Test cases are automatically and symbolically generated from the specification and the properties and are executed on a black-box implementation of the system. The test execution may detect violations of conformance between implementation and specification; in addition, it may detect violation/satisfaction of the properties by the implementation and by the specification. In this sense, testing completes verification. The approach is illustrated on simple examples and on a Bounded Retransmission Protocol.

Index Terms—Software/program verification, formal methods, testing strategies.

1 Introduction

Formal verification and conformance testing are two well-established approaches for the validation of reactive systems. Both approaches consist in comparing two different views, or levels of abstraction, of a system:

- Formal verification compares a formal *specification* of the system with respect to a set of higher-level *properties* that the system should satisfy.
- Conformance testing [1], [2] compares the observable behavior of an actual black-box implementation of the system with the observable behavior described by a formal specification, according to a conformance relation. We adopt in this paper a conformance testing theory based on IOLTS (*input-output labeled transition systems*) and the **ioco** conformance relation [3].

Verification operates on formal models and allows in principle for complete, exhaustive proofs of properties on specifications. However, for expressive models such as those considered in this document (symbolic, infinite-state transition systems), verification is undecidable or too complex to be carried out completely and one has to resort to partial or approximated verification. For example, safety properties can be checked on overapproximations of the reachable states of the specification; if the property holds on the overapproximation, then it also holds on the original specification, but nothing can be concluded if the property is violated by the overapproximation.

• *The authors are with INRIA, Rennes, Campus universitaire de Beaulieu, 35042 Rennes, France.*
E-mail: {camille.constant, thierry.jeron, herve.marchand, vlad.rusu}@irisa.fr.

Manuscript received 28 Aug. 2006; revised 18 Jan. 2007; accepted 4 June 2007; published online 13 June 2007.
Recommended for acceptance by S. Uchitel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0206-0806.
Digital Object Identifier no. 10.1109/TSE.2007.70707.

Unlike verification, conformance testing is performed on the real implementation of the system by means of interactions between the implementation and test cases derived from the specification. In general, testing cannot prove that the implementation conforms to the specification: it can only detect nonconformances between the two "views" of the system. Since the specification serves as a basis for conformance testing, it is quite clear that the specification itself should be verified against expected properties.

Hence, the two techniques are complementary: Verification proves that the specification is correct with respect to the higher-level properties, and then conformance testing checks the correctness of the implementation with respect to the specification. Also, the two approaches use the same basic techniques and algorithms [4], [5], [6], [7].

However, only applying verification followed by conformance testing is not fully satisfactory. One problem is that the **ioco** conformance relation does not preserve safety properties: Even if the properties hold on the specification and the implementation **ioco**-conforms to the specification, the implementation can still violate the properties. This problem is worth mentioning although it has little practical importance, since testing can, in general, only detect nonconformances—it cannot prove conformance.

Another, more serious problem with the above "verification-then-testing" approach is that it does not guarantee that the properties are tested *at all* on the implementation. This is because the testing step only checks the conformance between implementation and *specification*, without taking the properties into account. Not testing a property that was worth verifying is a serious weakness.

Clearly, what is missing is a formal link between properties and tests. In this paper, we describe a methodology combining verification and testing that provides the

missing link, ensuring that properties verified on the specification are also tested on the implementation, and which also attempts to deal with the incompleteness of the verification step:

- A specification of a system—an extended input-output automaton, which may be infinite-state—and a set of safety properties ("nothing bad ever happens on any observable behavior") and possibility properties ("something good will eventually happen on some observable behaviors") are assumed to be given.[1] In practice, the specification could be, e.g., a UML statechart, and properties could be given by some positive and negative scenarios.
- The properties are first tentatively verified on the specification using automatic techniques based on approximated state-space exploration, which are sound, but, as a price to pay for automation, are not complete for the given class of properties. Because of this incompleteness, the verification may not succeed in proving or disproving the properties.
- However, even if verification did not fully succeed, testing can be performed and provide useful information. Test cases are automatically and symbolically generated from the specification and the properties and are executed on a black-box implementation of the system. The test execution may detect violations of conformance between implementation and specification; in addition, it may detect violation/satisfaction of the properties by the implementation *and by the specification*. In this sense, testing may complete verification.

From a methodological point of view, this means that it is not required that the (typically difficult) verification of the specification fully succeeds before testing of the final implementation can start; and the approach provides correct verdicts about the consistency between properties, specification, and implementation.

From a more theoretical point of view, a uniform presentation of verification and conformance testing is given. All properties are represented using *observers*, which are essentially automata (possibly extended with variables) with a set of accepting locations. Observers can be seen as an alternative to (temporal) logics for expressing properties; and, for some temporal logics such as LTL, formulas can be translated into "equivalent" observers; see, e.g., [8] for a transformation of *safety* LTL formulas into observers.

We also show that, under some conditions, the specification can be transformed into an observer for nonconformance. Such an observer is then a *canonical tester* [9] for **ioco**-conformance with respect to a given specification, which also proves that **ioco**-conformance to a specification is a safety property.[2] Then, *test generation* is essentially a synchronous product between the canonical tester and observers for the properties. It is followed by a *test selection* operation, which consists of extracting a "part" of the product that targets more specifically one or several of the properties. Finally, *test execution*, which consists of actually running the previously generated/selected tests, is not the main object of the present work; we just note that the symbolic test cases that we produce are reactive programs, which are executed in parallel with the implementation and attempt to "guide" the implementation toward satisfying or violating the properties chosen as the target of test selection.

1.1 Comparison to Related Work

Combining verification and testing is not a new idea. We compare here the present work with work from the literature and with our own previous work.

The approach described in [10] considers a deterministic finite-state specification S and an invariant P assumed to hold on S. Then, mutants S' of S are built using standard mutation operators and a combined machine is generated, which extends sequences of S with sequences of S'. Next, a model checker is used to generate sequences of S' that violate P, which proves that S' is a mutant of S violating P. Finally, the obtained sequences are interpreted as test cases to be executed on the implementation. In contrast, our approach is able to deal with certain classes of nondeterministic infinite-state specifications, and we perform the tests on black-box implementations, whereas [10] requires a mechanism for observing internal details (such as values of variables) of the implementation.

The approach described in [4] starts with a formal specification S and a temporal-logic property P assumed to hold on S and uses the ability of model checkers to construct counterexamples for $\neg P$ on S. These counterexamples can be interpreted as *witnesses* (and eventually transformed into test cases) for P on S. Other authors have investigated this approach, e.g., Hamon et al. [11], who propose efficient test generation techniques in this framework. Blom et al. [12] and Hong et al. [13] extend this idea by formalizing standard coverage criteria (all-definitions, all-uses, etc.) using observers (or temporal logic). Again, test cases are generated by model checking the observers (or the temporal-logic formulas) on the specification. The approaches described in these papers are also restricted to deterministic finite-state systems and do not apply to actual black-box implementations, as they require the ability to observe the values of variables in the implementation.

An approach combining model checking and black-box testing is black-box checking [14]. Under some assumptions on the implementation (the implementation is deterministic; an upper bound n on its number of states is known), the black-box checking approach constructs a complete test suite of size exponential in n for checking properties expressed by Büchi automata.

In [15], an approach for generating tests from finite-state specifications (which can be nondeterministic) and from

1. This choice of properties is not arbitrary. Safety properties on the observable behavior of the implementation are exactly the properties that can be disproved by black-box testing, and possibility properties are exactly the properties that can be established by black-box testing. We note that possibility properties ("$\exists\mathcal{U}$" properties, in branching-time temporal logic) are one class of liveness properties. Another class is that of *inevitability* properties ("$\forall\mathcal{U}$" properties), which state that something shall eventually happen on *all* observable behaviors of the implementation. Such properties cannot be proved nor disproved by black-box testing and are not considered here.

2. Intuitively, safety properties require that "nothing bad ever happens;" here, "bad" is nonconformance to the specification.

observers describing linear-time temporal logic properties is described. The generated test cases do not check for conformance to the specification but only compare the black-box implementation to the properties. The specification is only used as a guideline for test selection. In contrast, we deal with infinite-state specifications and generate tests for checking the implementation against the specification and the properties.

De Vries and Tretmans [16] describe a general framework for test generation and execution of test cases from nondeterministic specifications. The testing mechanism additionally uses so-called *observation objectives*, which are sets of traces of observable actions. The verdicts obtained by test execution describe relations between the black-box implementation and the specification, as well as the satisfaction (or not) of the objectives by the implementation. The authors instantiate their general approach in the TorX tool [17], where test purposes are finite automata and test generation and execution are performed on the fly. The work presented here can be seen as another instantiation of the same general approach, which differs from [16] with respect to the expressiveness of observers (infinite-state extended automata rather than finite) and with respect to the test generation mechanism (offline, symbolic rather than on-the-fly, enumerative). Offline test selection allows, in principle, for better selection than the on-the-fly approach, and symbolic test generation allows, in principle, a better handling of the state-space explosion problem.

Our approach can also be related to the combination of verification, testing, and monitoring proposed in [18]. In their approach, monitoring is passive (pure observation), whereas ours is interactive, guided by the choice of inputs to deliver to the system, precomputed in a test case.

We now compare this paper with some of our own previous work. In [19], we use a rudimentary combination of positive and negative observers for enumerative test generation for the SSCOP protocol, using the TGV and ObjectGode (Telelogic) tools. An observer describes an abstract view of the actions of the protocol, which over-approximates its behaviors. Test generation uses additional test purposes, which can be classified as positive observers. During test generation, violation of negative observers by the specification can be observed. However, the generated test cases are not meant to observe violations of the negative observers by the implementation.

In [20], we present an approach for combining model checking of safety properties and conformance testing for finite-state systems. That paper can be seen as a first step toward the approach presented here, which deals with infinite-state systems. In the finite-state settings of [20], verification is decidable, which influences the whole approach. For example, the generated test cases do not need to take into account the possibility that the properties might be violated by the specification—such violations are always detected by the model-checking step, in which case the test generation step is simply canceled.

In [21], we take one more step and lift the approach of [20] to infinite-state specifications and properties. However, unlike the present paper, in [21] we do not consider possibility properties. Such properties (also called "test purposes" in the literature) are a standard means for test selection; without them, e.g., a test case cannot deliver the standard "Pass" verdict.

Finally, in [22] and [23], we propose a symbolic algorithm for selecting test cases from a specification by means of test purposes. The difference with the present paper lies mainly in methodology. Test purposes in [22] and [23] are essentially a pragmatic means for test selection and are not necessarily related to properties of the specification. In contrast, test selection in the present paper uses safety or possibility properties, which are actual properties that the specification should satisfy, and the testing step is formally integrated with the verification of the properties on the specification.

Summarizing, most approaches in the literature deal with deterministic finite-state systems, whereas we deal with some classes of nondeterministic infinite-state systems; and this paper unifies and subsumes most of our works on conformance testing and its relation with verification.

1.2 Paper Organization

In Section 2, we present the model of Input-Output Symbolic Transition Systems (IOSTS). Transitions are labeled by actions partitioned into inputs, outputs, and internal actions; inputs and outputs may also carry symbolic parameters. The semantics of IOSTS in terms of (usually infinite) labeled transition systems is also given. In Section 3, we define three symbolic operations on IOSTS that are used for verification and conformance testing. Section 4 presents the notion of observer for possibility and for safety properties, the verification of such properties, and the theory of conformance testing of [3] reformulated using the notion of canonical tester, which is basically an observer for nonconformance. In Section 5, we describe test generation from a specification and a set of properties, as a product between observers for the properties and the canonical tester obtained from the specification. We prove that the resulting test cases are *correct* in the sense that the verdicts that they emit correctly characterize the mutual consistency between implementation, specification, and properties. Then, test selection amounts to choosing one or several properties of interest and targeting the test toward checking the chosen properties; in Section 6, we show how this can be done by using approximated analysis techniques based on abstract interpretation [24]. The methodology is illustrated with simple examples and also on a larger example, the BRP protocol [25].

2 The IOSTS Model

The *Input/Output Symbolic Transition Systems* (IOSTS) model is inspired by I/O automata [26]. Transitions are labeled by *actions*, partitioned into inputs, outputs, and internal actions; inputs and outputs may carry symbolic parameters. Unlike I/O automata, IOSTS do not require *input-completeness* (i.e., all input actions do not need to be enabled all the time).

2.1 Syntax of IOSTS

Definition 1 (IOSTS). *An IOSTS is a tuple* $\langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$, *where*

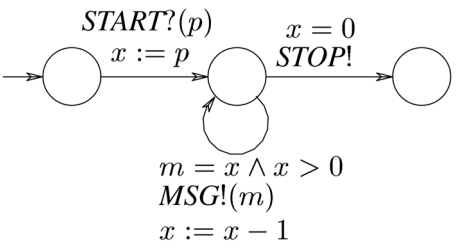


Fig. 1. Sample IOSTS S.

- V *is a finite set of typed* variables.
- P *is a finite set of* parameters. *For* $x \in V \cup P$, $type(x)$ *denotes the type of* x.
- Θ *is the* initial condition, *a predicate with variables in* V.
- L *is a nonempty, finite set of* locations *and* $l^0 \in L$ *is the initial location.*
- Σ *is a nonempty, finite* alphabet, *which is the disjoint union of a set* $\Sigma^?$ *of* input actions, *a set* $\Sigma^!$ *of* output actions, *and a set* Σ^τ *of* internal actions. *For each action* $a \in \Sigma$, *its* signature $sig(a) = \langle p_1, \ldots, p_k \rangle \in P^k$ $(k \in \mathbb{N})$ *is a tuple of distinct parameters. The signature of internal actions is the empty tuple;*
- $\mathcal{T}$ *is a set of transitions. Each transition* $t \in \mathcal{T}$ *has*
  - *a location* $l \in L$, *called the* origin *of the transition,*
  - *an action* $a \in \Sigma$, *called the* action *of the transition,*
  - *a predicate* G *with variables in* $V \cup sig(a)$, *called the* guard,
  - *an assignment* A *of the form* $(x := A^x)_{x \in V}$, *such that, for each* $x \in V$, *the right-hand side* A^x *of the assignment* $x := A^x$ *is an expression on* $V \cup sig(a)$, *and*
  - *a location* $l' \in L$, *called the destination of the transition.*

We assume that the assignments are "well typed," that is, each expression A^x in the right-hand side has a type that matches that of the variable x in the left-hand side. Such notions of well-typedness can be formalized, for example, using abstract datatypes. We do not elaborate these notions any further to avoid losing focus.

For technical reasons, we shall assume that the guards of IOSTS are expressed in a theory which is closed under quantifier elimination; that is, any predicate containing quantifiers is equivalent to one without quantifiers (and over the same set of variables). For example, Presburger arithmetic with function symbols [27] satisfies these constraints (provided that quantifiers do not bind variables within the scope of a function) and is expressive enough to express common data structures such as integers and arrays.

In graphical representations, inputs are followed by the "?" symbol and outputs are followed by "!". These symbols are not part of an action's name, but are used here only as notations.

Example 1: A simple IOSTS is depicted in Fig. 1. This system expects a $START$ input carrying an integer parameter p and saves the value of p into the variable x. Then, as long as x is strictly positive, its value is emitted to the environment via the output MSG carrying the parameter m (the guard $m = x \wedge x > 0$ means "choose a value for the parameter m that, together with the value of the variable x, satisfies the guard"). The variable x is decreased by 1 and, when it reaches 0, the $STOP$ output is emitted.

2.2 Semantics of IOSTS

The semantics of IOSTS is described in terms of input-output labeled transition systems.

Definition 2 (IOLTS semantics of an IOSTS). *An Input-Output Labeled Transition System (IOLTS) is a tuple* $\langle S, S^0, \Lambda, \rightarrow \rangle$ *where* S *is a possibly infinite set of states,* $S^0 \subseteq S$ *is the possibly infinite set of initial states,* $\Lambda = \Lambda^? \cup \Lambda^! \cup \Lambda^\tau$ *is a possibly infinite set of actions, partitioned into input, output, and internal actions, and* $\rightarrow \subseteq S \times \Lambda \times S$ *is the* transition relation.

The set $\Lambda^? \cup \Lambda^!$ is also called the set of *observable actions*. Intuitively, the IOLTS semantics of an IOSTS $\langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ explores the set of *valuations* of variables V and parameters P, where a *valuation* of the variables V is a mapping ν which maps every variable $x \in \mathcal{V}$ to a value $\nu(x)$ in the domain of x. Valuations of the parameters P are defined in a similar manner.

Let $\mathcal{V}$ denote the set of valuations of the variables V and let Π denote the set of valuations of the parameters P. Then, for an expression E involving (a subset of) $V \cup P$, and for $\nu \in \mathcal{V}$, $\pi \in \Pi$, we denote by $E(\nu, \pi)$ the value obtained by evaluating the result of substituting in E each variable by its value according to ν and each parameter by its value according to π. For $P' \subseteq P$, we denote by $\Pi_{P'}$ the restriction of the set Π of valuations to the set P' of parameters.

Definition 3 (IOLTS semantics of an IOSTS). *The semantics of an IOSTS* $S = \langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ *is an IOLTS* $[\![S]\!] = \langle S, S^0, \Lambda, \rightarrow \rangle$, *defined as follows:*

- *The set of states is* $S = L \times \mathcal{V}$.
- *The set of initial states is* $S^0 = \{\langle l^0, \nu \rangle | \Theta(\nu) = true\}$.
- *The set of actions* $\Lambda = \{\langle a, \pi \rangle | a \in \Sigma, \pi \in \Pi_{sig(a)}\}$, *hereafter called the set of valued actions, is partitioned into the sets* $\Lambda^?$ *of valued inputs,* $\Lambda^!$ *of valued outputs, and* Λ^τ *of internal actions*[3] *such that, for* $\# \in \{?, !, \tau\}$, $\Lambda^\# = \{\langle a, \pi \rangle | a \in \Sigma^\#, \pi \in \Pi_{sig(a)}\}$.
- $\rightarrow$ *is the smallest relation in* $S \times \Lambda \times S$ *defined by the following rule:*

$$\frac{\langle l, \nu \rangle, \langle l', \nu' \rangle \in S \quad \langle a, \pi \rangle \in \Lambda \quad t: \ \langle l, a, G, A, l' \rangle \in \mathcal{T} \quad G(\nu, \pi) = true \quad \nu' = A(\nu, \pi)}{\langle l, \nu \rangle \overset{\langle a, \pi \rangle}{\rightarrow} \langle l', \nu' \rangle}.$$

The rule says that the valued action $\langle a, \pi \rangle$ takes the system from a state $\langle l, \nu \rangle$ to a state $\langle l', \nu' \rangle$ if there exists a transition $t: \ \langle l, a, G, A, l' \rangle$ whose guard G evaluates to $true$

3. Since internal actions do not carry parameters, the sets Λ^τ and Σ^τ can be identified. Allowing internal actions to carry arbitrary parameters would allow for infinite branching in the semantics of IOSTS, which seriously complicates some operations on IOSTS (such as elimination of internal actions) that are essential for our purposes.

when the variables evaluate according to ν and the parameters carried by the action a evaluate according to π. Then, the assignment A maps the pair (ν, π) to ν'.

Definition 4 (Execution). *An execution fragment is a sequence of alternating states and valued actions* $s_1\alpha_1 s_2\alpha_2 ... \alpha_{n-1}s_n \in S \cdot (\Lambda \cdot S)^*$ *such that* $\forall i = 1, \ldots, n-1.\ s_i \xrightarrow{\alpha_i} s_{i+1}$. *An* execution *is an execution fragment starting in an initial state.*

We often write $s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \cdots s_n \xrightarrow{\alpha_n} s_{n+1}$ as a shortcut for $\forall i = 1, \ldots, n.\ s_i \xrightarrow{\alpha_i} s_{i+1}$, and $s \xrightarrow{\alpha}_t s'$ if s is obtained from s' by firing a certain transition $t \in \mathcal{T}$.

Definition 5 ((co)reachable states). *A state is* reachable *if it belongs to an execution. For a sequence* $\sigma = \alpha_1\alpha_2 \cdots \alpha_n$ *of valued actions, we write* $s \xrightarrow{\sigma} s'$ *for*

$$\exists s_1, \ldots s_{n+1} \in S.\ s = s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \cdots s_n \xrightarrow{\alpha_n} s_{n+1} = s'.$$

For a set $S' \subseteq S$ *of states of an IOSTS, we write* $s \xrightarrow{\sigma} S'$ *if there exists a state* $s' \in S'$ *such that* $s \xrightarrow{\sigma} s'$. *We say that* s *is coreachable for* S'.

Definition 6 (Trace). *The* trace $trace(\rho)$ *of an execution* ρ *is the projection of* ρ *on the set* $\Lambda^! \cup \Lambda^?$ *of observable actions. The set of traces of an IOSTS* $\mathcal{S}$ *is the set of all traces of all executions of* $\mathcal{S}$, *and is denoted by* $Traces(\mathcal{S})$.

Definition 7 (Recognized traces). *Let* $F \subseteq L$ *be a set of locations of an IOSTS* $\mathcal{S}$. *An execution* ρ *of* $\mathcal{S}$ *is recognized by* F *if the execution terminates in a state in* $F \times \mathcal{V}$. *A trace is recognized by* F *if it is the trace of an execution recognized by* F. *The set of traces of an IOSTS* $\mathcal{S}$ *recognized by a set* F *of locations is denoted by* $Traces(\mathcal{S}, F)$.

3 Basic Operations on IOSTS

In this section, we define a few basic operations on IOSTS that are used in verification and conformance testing based on this model. These operations are the *parallel product* of two IOSTS, as well as the *suspension* and the *determinization* of an IOSTS.

3.1 Parallel Product

The *parallel product* of two IOSTS $\mathcal{S}_1$, $\mathcal{S}_2$ will be used both in verification (for defining the set of traces of an IOSTS that are recognized by an observer) and in conformance testing (for modeling the synchronous execution of a test case on an implementation). This operation requires that $\mathcal{S}_1$, $\mathcal{S}_2$ share the same sets of input and output actions (with the same signatures), have the same set of parameters, and have no variables or internal actions in common.

Definition 8 (Compatible IOSTS). *The IOSTS* $\mathcal{S}_j = \langle V_j, P_j, \Theta_j, L_j, l_j^0, \Sigma_j, \mathcal{T}_j\rangle$ $(j = 1, 2)$ *and* $\Sigma_j = \Sigma_j^? \cup \Sigma_j^!$ *are compatible if* $V_1 \cap V_2 = \emptyset$, $P_1 = P_2$, $\Sigma_1^! = \Sigma_2^!$, $\Sigma_1^? = \Sigma_2^?$, *and* $\Sigma_1^\tau \cap \Sigma_2^\tau = \emptyset$.

Definition 9 (Parallel products). *The parallel product* $\mathcal{S} = \mathcal{S}_1||\mathcal{S}_2$ *of two compatible IOSTS* $\mathcal{S}_1, \mathcal{S}_2$ *(see Definition 8) is the IOSTS* $\langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T}\rangle$ *defined by:* $V = V_1 \cup V_2$, $P = P_1 = P_2$, $\Theta = \Theta_1 \wedge \Theta_2$, $L = L_1 \times L_2$, $l^0 = \langle l_1^0, l_2^0\rangle$, $\Sigma^? = \Sigma_1^? = \Sigma_2^?$, $\Sigma^! = \Sigma_1^! = \Sigma_2^!$, $\Sigma^\tau = \Sigma_1^\tau \cup \Sigma_2^\tau$. *The set* $\mathcal{T}$ *of symbolic transitions of the parallel product is the smallest set satisfying the following rules:*

1.
$$\frac{\langle l_1, a, G_1, A_1, l_1'\rangle \in \mathcal{T}_1, \quad a \in \Sigma_1^\tau, \quad l_2 \in L_2}{\langle\langle l_1, l_2\rangle, a, G_1, A_1 \cup (x := x)_{x \in V_2}, \langle l_1', l_2\rangle\rangle \in \mathcal{T}}$$
$$(\textit{and symmetrically for } a \in \Sigma_2^\tau),$$

2.
$$\frac{\langle l_1, a, G_1, A_1, l_1'\rangle \in \mathcal{T}_1 \quad \langle l_2, a, G_2, A_2, l_2'\rangle \in \mathcal{T}_2}{\langle\langle l_1, l_2\rangle, a, G_1 \wedge G_2, A_1 \cup A_2, \langle l_1', l_2'\rangle\rangle \in \mathcal{T}}$$
$$(\textit{for } a \in \Sigma^! \cup \Sigma^?).$$

The parallel product allows internal actions to evolve independently by Rule 1 and synchronizes on the observable actions (inputs and outputs) by Rule 2.

Lemma 1. $Traces(\mathcal{S}_1||\mathcal{S}_2) = Traces(\mathcal{S}_1) \cap Traces(\mathcal{S}_2)$, *and*

$$Traces(\mathcal{S}_1||\mathcal{S}_2, F_1 \times F_2) = Traces(\mathcal{S}_1, F_1) \cap Traces(\mathcal{S}_2, F_2).$$

3.2 Suspension

In conformance testing, it is assumed that the environment may observe not only outputs, but also *absence of outputs* (i.e., in a given state, the system is blocked, in the sense that it does not emit any output for the environment to observe). Such absence of output is called *quiescence* in the **ioco** theory of conformance testing [28].

On black-box *implementations*, quiescence is observed using timers: A timer is reset, e.g., whenever the tester sends a stimulus to the implementation. It is assumed that the duration of the timer is large enough such that, if no output occurs while the timer is running, then no output will ever occur. Then, when the timer expires, the environment "observes" quiescence.

In order to distinguish between observations of quiescence that are allowed by a specification and those that are not, quiescence is made explicit on *specifications* by means of a symbolic operation called *suspension*. This operation transforms an IOSTS $\mathcal{S}$ into an IOSTS $\mathcal{S}^\delta$, also called the *suspension IOSTS* of $\mathcal{S}$, in which quiescence is materialized by a new output action δ. For this, each location l of $\mathcal{S}^\delta$ contains a new self-looping transition, labeled with a new output action δ, which may be fired if and only if no other output or internal action may be fired in l.

Definition 10. *Given* $\mathcal{S} = \langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T}\rangle$, *an IOSTS, with* $\Sigma = \Sigma^! \cup \Sigma^? \cup \Sigma^\tau$ *and* $\delta \notin \Sigma$, *the suspension IOSTS* $\mathcal{S}^\delta$ *is the tuple* $\langle V, P, \Theta, L, l^0, \Sigma^\delta, \mathcal{T}^\delta\rangle$, *where* $\Sigma^\delta = \Sigma^{\delta!} \cup \Sigma^? \cup \Sigma^\tau$ *with* $\Sigma^{\delta!} = \Sigma^! \cup \{\delta\}$, $\mathcal{T}^\delta = \mathcal{T} \cup \{\langle l, \delta, G_{\delta,l}, (x := x)_{x \in V}, l\rangle \mid l \in L\}$, *and*

$$G_{\delta,l} \triangleq \bigwedge_{a \in \Sigma^! \cup \Sigma^\tau} \neg G_{a,l} \text{ and } G_{a,l} \triangleq \bigvee_{\langle l,a,G,A,l'\rangle \in \mathcal{T}} \exists\, sig(a).G. \quad (1)$$

We have assumed that the guards are expressed in a theory where existential quantifiers can be eliminated, such as, e.g., Presburger Arithmetic. The guard $G_{\delta,l}$ in (1) formalizes the conditions under which δ may be fired in location l, which are that no output or internal action may be fired in l. Now, for $a \in \Sigma^! \cup \Sigma^\tau$, $G_{a,l}$ gives the conditions on the system's variables under which the action a is

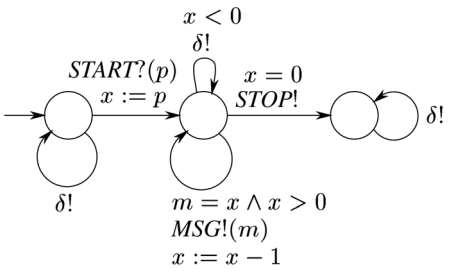


Fig. 2. Suspension IOSTS S^δ.

fireable in l. Hence, $G_{\delta,l}$ is the *conjunction* of the *negations* of all formulas $G_{a,l}$, for all $a \in \Sigma^! \cup \Sigma^\tau$.

Example 2. For the IOSTS $\mathcal{S}$ depicted in Fig. 1, the IOSTS $\mathcal{S}^\delta$ is depicted in Fig. 2. In this system, a $START$ input with a negative parameter ($p < 0$) does not allow for MSG or $STOP$ outputs, *i.e., the system is quiescent after* $START$. This is made explicit by a self-looping transition labeled δ, whose guard $x < 0$ is obtained by simplifying the expression $\neg(x = 0 \vee (\exists\, m, m = x \wedge x > 0))$, which corresponds to (1) above.

3.3 Deterministic IOSTS and Determinization

Intuitively, an IOSTS is *deterministic* if each of its traces matches exactly one execution. For example, test cases in conformance testing satisfy this property, as they must give the same verdict on the same interaction trace with an implementation.

Definition 11 (Determinism). *An IOSTS* $\langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T}\rangle$ *is deterministic if*

- *it has no internal actions:* $\Sigma^\tau = \emptyset$,
- *it has at most one initial state, i.e., the initial condition* Θ *is satisfied by at most one valuation* ν^0 *of the variables (then, the initial state is* $\langle l^0, \nu^0\rangle$*), and*
- *for all* $l \in L$ *and for each pair of distinct transitions with origin in* l *and labeled by the same action* a*, i.e.,* $t_1 : \langle l, a, G_1, A_1, l_1\rangle$ *and* $t_2 : \langle l, a, G_2, A_2, l_2\rangle$*, the conjunction of the guards* $G_1 \wedge G_2$ *is unsatisfiable.*

Since we have assumed that the guards of transitions are in a theory where existential quantifiers can be eliminated, the satisfiability of (conjuntions of) guards is decidable. The IOSTS seen so far in this paper (Fig. 1 and Fig. 2) meet the first and third conditions for being deterministic but not the second one (uniqueness of the initial state). Since their only variable x is set to the value p coming from the environment on the first input $START(p)$, the initial value is irrelevant for the rest of the behavior. A unique initial state can be obtained by letting the variable x have some arbitrary (fixed) value initially, e.g., $x = 0$. Examples of nondeterministic IOSTS are given in Section 7.

The following lemma allows us to define a sequence of transitions that "matches" a trace of an IOSTS. It will be used for proving the correctness of our test generation method.

Lemma 2 (Support of a trace). *A sequence* $\sigma : \alpha_1\alpha_2\cdots\alpha_n$ *of valued actions is a trace of a deterministic IOSTS* $\mathcal{S}$ *if and only if there exist states* s_i $(i = 1, n)$ *of* $\mathcal{S}$ *and a sequence of transitions* $t_i = \langle l_i, a_i, G_i, A_i, l_{i+1}\rangle$ $(i = 1, \ldots n)$ *of the IOSTS such that, for* $i = 1 \ldots, n$,

- l_1 *is the initial location of the IOSTS, and* ν_1 *satisfies the initial condition* Θ *of the IOSTS,*
- $s_i = \langle l_i, \nu_i\rangle$*, for some valuations* ν_i *of the variables of the IOSTS,*
- $\alpha_i = \langle a_i, \pi_i\rangle$*, for some valuation* π_i *of the parameters in* $sig(a_i)$*, and*
- *the valuations of variables and parameters defined by* ν_i *and* π_i *satisfy the guard* G_i*, i.e.,* $\langle \nu_i, \pi_i\rangle \models G_i$*, and the valuation* ν_{i+1} *is obtained from* ν_i, π_i *by applying the assignments* A_i*, i.e.,* $\nu_{i+1} = A_i(\nu_i, \pi_i)$.

In this case, the sequence of transitions $(t_i)_{i=1,\ldots,n}$ *is called a support of the trace* $sigma$*. Moreover, each trace has a unique support.*

Proof. Follows directly from the semantics of IOSTS and the definition of determinism. □

3.4 Determinizing an IOSTS $\mathcal{S}$

Determinizing an IOSTS $\mathcal{S}$ means computing a deterministic IOSTS with the same traces as $\mathcal{S}$. This operation consists of two steps: eliminating internal actions and eliminating nondeterminism between observable actions. The first step is presented below. The second one is more complex and, due to space limitations, is only sketched here; we refer the reader to [29] for details.

3.4.1 Eliminating Internal Actions

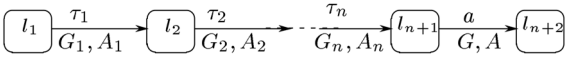
For eliminating internal actions, the idea [22], [30] is to compute the effect of any sequence of internal actions that ends with an input-labeled or output-labeled transition and to encode this effect in the guard and assignments of the last transition.

The basic operation required for eliminating internal actions is to compose guards and assignments of two consecutive transitions in which the first one, $t_1 : \langle l_1, \tau, G_1, A_1, l_1'\rangle$, is labeled by an internal action τ, and the second one, $t_2 : \langle l_1', a, G_2, A_2, l_2\rangle$, is labeled by an observable action a (input or output). Then, we denote by $A_2 \circ A_1$ the function that, to each valuation ν of the variables and π of the parameters in $sig(a)$, associates the valuation $A_2(A_1(\nu), \pi)$ of the variables, and by $G_2 \circ A_1$ the function that, to each valuation v of the variables, and π of the parameters in $sig(a)$, associates the Boolean value $G_2(A_1(\nu), \pi)$. Syntactically, the composition operation $\circ$ is implemented using substitution: If A_1 is the assignment $(x := A_x^1)_{x\in V}$ and A_2 is the assignment $(x := A_x^2)_{x\in V}$, then, $A_2 \circ A_1$ is the assignment $(x := A_x^2(x/A_x^1))_{x\in V}$, where every occurrence of x in the expression A_x^2 has been replaced by A_x^1. Similarly, the guard $G_2 \circ A_1$ is obtained by substituting, for every $x \in V$, all the occurrences of x in G_2 by the corresponding expression A_x^1. We shall also be needing the following notions before defining the elimination of internal actions:

Definition 12 (Contiguous sequence). *Let* $\eta : t_1, t_2, \ldots t_m$ $(m \geq 0)$ *be a sequence of transitions of an IOSTS. We say that the sequence is contiguous if, for* $i = 1, n-1$*, the destination of* t_i *is equal to the origin of* t_{i+1}*. In this case, we write* $\eta : l_1 \xrightarrow{t_1} l_2 \cdots l_m \xrightarrow{t_m} l_{m+1}$*, in which* l_1 *is the origin of the first transition,* l_{m+1} *is the destination of the last transition, and, for* $i = 1, m-1$*,* l_{i+1} *is both the destination of* t_i *and the origin of* t_{i+1}.

Definition 13 (Eliminating a sequence of internal actions). *Let* $\eta : l_1 \xrightarrow{t_1} l_2 \cdots l_n \xrightarrow{t_n} l_{n+1} \xrightarrow{t_{n+1}} l_{n+2}$ $(n \geq 1)$ *be a finite, contiguous sequence of transitions of an IOSTS. For* $i = 1, n$,

the action τ_i of t_i is internal, the action of t_{n+1} is an input or output action a, and for $i = 1, \ldots, n+1$, let the guard and assignments of t_i be G_i, A_i. We define the transition $t = Elim(\eta)$ as follows: The origin of t is l_1, its action is a, its guard is $G_1 \wedge (G_2 \circ A_1) \wedge \ldots (G_{n+1} \circ A_n \circ \cdots \circ A_1)$, its assignments are $A_{n+1} \circ A_n \circ \cdots \circ A_1$, and its destination is l_{n+2}.



Definition 14 (Eliminable sequence of transitions). *Let η be a sequence of transitions as in Definition 13. We say η is eliminable if either the location l_1 is initial or l_1 is the destination of some transition t labeled by an observable action (input or output).*

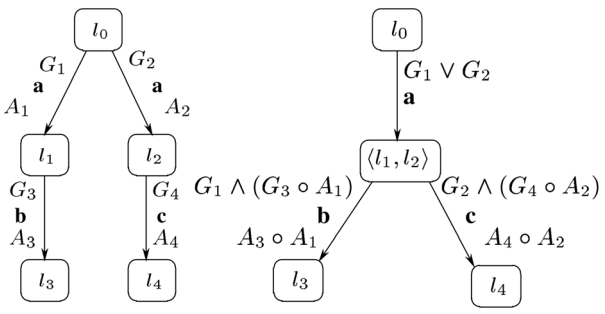
Definition 15 (Eliminating internal actions). *For $\mathcal{S}$, an IOSTS, the IOSTS $Elim(\mathcal{S})$ is built as follows:*

1. *Except for the set of transitions, $Elim(\mathcal{S})$ is identical to $\mathcal{S}$.*
2. *The set of transitions $\mathcal{T}_{Elim(\mathcal{S})}$ consists of the transitions of $\mathcal{S}$ labeled by observable actions and of the set of transitions of the form $Elim(\eta)$ for each eliminable sequence $\eta : l_1 \xrightarrow{t_1} l_2 \cdots l_n \xrightarrow{t_n} l_{n+1} \xrightarrow{t_{n+1}} l_{n+2}$ as in Definition 14.*

It can be shown that for IOSTS that do not contain *structural* cycles of internal actions (that is, contiguous sequences of the form $l_1 \xrightarrow{t_1} l_2 \cdots l_m \xrightarrow{t_m} l_{m+1}$ with $l_{m+1} = l_1$ and the action of each of the transitions t_i is internal), there are finitely many eliminable sequences in the sense of Definition 14, and the above procedure terminates and produces a trace-equivalent IOSTS.

3.5 Resolving Nondeterminism for Observable Actions

The second step of the determinization operation consists in resolving nondeterministic choices for observable actions. For this, a *symbolic determinization* procedure is defined in [29]. The procedure performs, in a certain order, a sequence of *local determinization steps*, one of which is illustrated below, where the IOSTS in the left-hand side is nondeterministic due to the two transitions labeled a with origin in l_0, with two different destinations (l_1 and l_2) and two different assignments (A_1 and A_2, respectively).



The idea for local deteriminism is to create a new location (here, $\langle l_1, l_2 \rangle$, depicted in the right-hand side of the picture) and three new transitions:

- a transition from l_0 to $\langle l_1, l_2 \rangle$, with guard $G_1 \vee G_2$, action **a**, and empty assignments; intuitively, this transition represents "both" transitions involved into nondeterminism, except for the fact that it does not "know" which assignment to perform (A_1 or A_2); hence, it performs none, but "postpones" the assignment to perform onto the following transitions,
- a transition from $\langle l_1, l_2 \rangle$ to l_3, labeled **b**, whose assignment $A_3 \circ A_1$ has "incorporated" the "postponed" assignment A_1 from the previous transition; similarly, the guard $G_1 \wedge (G_3 \circ A_1)$ of this transition has "incorporated" the guard G_1 and assignment A_1 of the previous one, and
- a transition similar to the previous one, from $\langle l_1, l_2 \rangle$ to l_4, labeled **c**.

In [29], we propose a procedure that iterates such local determinization steps and that terminates for the class of *bounded lookahead* IOSTS, for which a bounded trace allows to infer which transition involved into a nondeterministic choice has been taken.

Definition 16 (Bounded lookahead). *An IOSTS $\mathcal{S}$ has lookahead $n \in \mathbb{N}$ in a state $s \in [\![\mathcal{S}]\!]$ if $\forall t_1, t_2 \in \mathcal{T}_{\mathcal{S}}$*

$$\forall s_1, s_1', s_2, s_2' \in [\![\mathcal{S}]\!],$$

$\forall \alpha \in (\Lambda^! \cup \Lambda^?)$, and $\forall \sigma_1, \sigma_2 \in (\Lambda^! \cup \Lambda^?)^n$, $s \xrightarrow{\alpha}_{t_1} s_1'$, $s_1' \xrightarrow{\sigma_1} s_1$ and $s \xrightarrow{\alpha}_{t_2} s_2' \xrightarrow{\sigma_2} s_2$ imply $t_1 = t_2$. An IOSTS $\mathcal{S}$ has lookeahad n if it has lookahead n in all $s \in [\![\mathcal{S}]\!]$, and $\mathcal{S}$ has bounded lookahead if it has lookeahad n for some $n \in \mathbb{N}$.

Overall, the determinization operation terminates if and only if the IOSTS does not have cycles of internal actions and satisfies the bounded lookahead condition. We denote by $det(\mathcal{S})$ the deterministic IOSTS obtained from the IOSTS $\mathcal{S}$.

Lemma 3 (Traces of determinized IOSTS).

$$Traces(det(\mathcal{S})) = Traces(\mathcal{S}).$$

4 Verification and Conformance Testing with IOSTS

4.1 Verification

We consider the following standard verification problem: Given a reactive system modeled using an IOSTS $\mathcal{S}$ and a property ψ on the system's traces, does $\mathcal{S}$ satisfy ψ? We consider two kinds of properties: Safety properties and possibility properties. Both can be modeled using *observers*, which are deterministic IOSTS equipped with a set of recognizing locations.

Definition 17 (Observer). *An observer is a deterministic IOSTS ω together with a set of distinguished locations $F \subseteq L_\omega$ with no outgoing transitions, i.e., no location in F is the origin of any symbolic transition of ω. An observer (ω, F) is compatible with an IOSTS M if ω is compatible with M. The set of observers compatible with M is denoted by $\Omega(M)$.*

Let us recall the definition of the set of traces recognized by a set of locations of an IOSTS (Definition 7). For an observer (ω, F), this gives

$$Traces(\omega, F) = \{\sigma \in \Lambda_\omega^* \mid \exists s_o \in S_\omega^0,\ \exists l \in F,\ \exists \nu \in \mathcal{V}.s_o \xrightarrow{\sigma}_\omega \langle l, \nu \rangle\}. \quad (2)$$

We shall attach to observers a "sign" attribute, that is, observers can either be *positive* observers or *negative* observers. The former shall denote possibility properties, whereas the latter denote safety properties. By convention, we denote the set of recognizing locations of positive observers by $Satisfy$, and those of negative observers by $Violate$. We now define what it means for an observer to be satisfied by an IOSTS.

Definition 18. *Let M be an IOSTS and $(\omega, Satisfy) \in \Omega(M)$ a positive observer compatible with M. Then, M satisfies $(\omega, Satisfy)$, denoted by $M \models^+ (\omega, Satisfy)$, if*

$$Traces(M) \cap Traces(\omega, Satisfy) \neq \emptyset.$$

That is, a positive observer is satisfied by an IOSTS if at least one of the traces of the IOSTS is recognized by the observer. This corresponds to the intuition for possibility properties ("something good eventually happens on at least one observable trace"). In contrast, safety properties say that "nothing bad ever happens" and their satisfaction is defined as follows:

Definition 19. *Let M be an IOSTS and $(\omega, Violate) \in \Omega(M)$ a negative observer compatible with M. Then, M satisfies $(\omega, Violate)$, denoted by $M \models^- (\omega, Violate)$, if*

$$Traces(M) \cap Traces(\omega, Violate) = \emptyset.$$

Note also that satisfying a negative observer (a safety property) amounts to not satisfying, i.e., to violating a positive observer (a possibility property)—and reciprocally.

4.1.1 Verifying Safety Properties

Consider an IOSTS M and a negative observer $(\omega, Violate) \in \Omega(M)$. The safety property defined by the observer is satisfied by sequences in $(\Lambda_M^! \cup \Lambda_M^?)^* \setminus Traces(\omega, Violate)$ (and these sequences only). In particular, if M is the suspended IOSTS $\mathcal{S}^\delta$ of a given IOSTS $\mathcal{S}$, the property is satisfied by a subset of $(\Lambda_\mathcal{S}^! \cup \{\delta\} \cup \Lambda_\mathcal{S}^?)^*$.

Example 3. The observer ω_1 depicted in Fig. 3a encodes a safety property saying that between a $START$ input carrying a parameter p satisfying $p > 0$ and a $STOP$ output, the system must emit at least one MSG output. The set of recognizing locations is here the singleton $\{Violate\}$. The labels "*" are a shortcut notation for all valued actions (including quiescence, δ) that are not explicitly carried by other transitions.

The observer ω_2 depicted in Fig. 3b describes almost the same property except that the parameter p satisfies the weaker constraint $p \geq 0$.

Let L be the set of locations of the IOSTS M. Then, $Traces(M) = Traces(M, L)$ and, by Lemma 1,

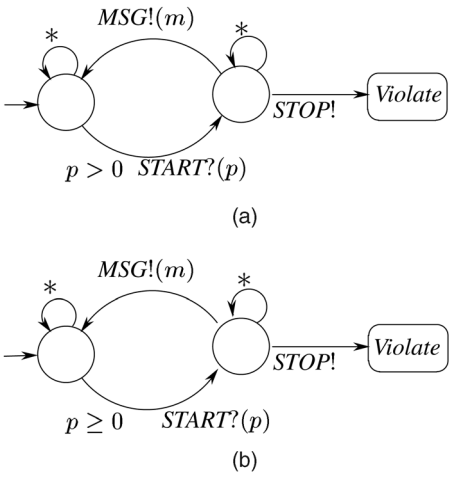


Fig. 3. Observers encoding safety properties. (a) ω_1. (b) ω_2.

$$Traces(M \| \omega, L \times Violate) = Traces(M) \cap Traces(\omega, Violate).$$

Thus, verifying $M \models^- (\omega, Violate)$ amounts to verifying whether $Traces(M \| \omega, L \times Violate)$ is empty, which can be done by establishing that the set of locations $L \times Violate$ is *unreachable* from the initial states of $M \| \omega$, or, alternatively, that the initial states of $M \| \omega$ are *not coreachable* for $L \times Violate$.

Reachability is probably the most studied verification problem and many different approaches for solving it have been proposed. Among the automatic ones, we consider here *model checking* [31], which, in general, explores a finite subset of the set of reachable states, and *abstract interpretation* [24], which symbolically explores a superset of the set of reachable states. Thus, model checking can prove reachability but, in general, cannot disprove it, and abstract interpretation can disprove reachability but, in general, cannot prove it. Indeed, reachability is undecidable for the class of models considered here as it is in general for infinite-state systems.

However, one can use abstract interpretation to *try* to prove that a safety property is satisfied, and one can use model checking to *try* to prove that it is violated. To this end, our tool STG (Symbolic Test Generation) [32] was connected with the NBac abstract interpretation tool NBac [33]. For proving a safety property represented using a negative observer $(\omega, Violate)$, STG automatically computes the product $\omega || M$; then, NBac automatically computes an overapproximation of the set of states that are both reachable from the initial states and coreachable for the locations $L \times Violate$. If the result is empty, the safety property holds, otherwise, no conclusions can be drawn.

For example, this approach allows to establish that the IOSTS $\mathcal{S}^\delta$ depicted in Fig. 2 satisfies the observer ω_1 depicted in the top of Fig. 3. However, using the same IOSTS $\mathcal{S}^\delta$ and the observer ω_2, we obtain an inconclusive result. It turns out that the safety property described by ω_2 is violated by $\mathcal{S}^\delta$, as the $START$ input carrying $p = 0$ allows a $STOP$ output before any MSG output. Violations of safety properties can be detected by model checking, but, of course, neither satisfaction nor violation can be established in general because of inherent undecidability problems.

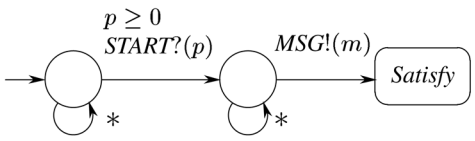


Fig. 4. Observer for possibility property ω_3.

4.1.2 Verifying Possibility Properties

In contrast to safety properties, possibility properties express that "something good eventually happens on some observable behavior." We express possibility properties using positive observers $(\omega, Satisfy)$; the property is satisfied if some location in the set $Satisfy$ is reachable. If the observer is compatible with an IOSTS M, then the property is satisfied by M if at least one trace of M is in $Traces(\omega, Satisfy)$.

Example 4. The observer ω_3 depicted in Fig. 4 describes the possibility that, after a $START$ input carrying a parameter $p \geq 0$, the system can emit the MSG output. The set of recognizing locations is the singleton $\{Satisfy\}$.

The property is satisfied by the IOSTS $\mathcal{S}^\delta$ depicted in Fig. 2, e.g, the trace $START(1) \cdot MSG(1)$ is a trace of $\mathcal{S}^\delta$ that reaches $Satisfy$, which can be established by model checking. Again, it is important to note that neither satisfaction nor violation of such properties can be decided in general, hence, no automatic tool can hope to prove or disprove all of them.

4.1.3 Combining Observers

The parallel product of two observers (ω_1, F_1) and (ω_2, F_2) can also be interpreted as an observer by choosing the set of recognising locations. Such combinations will be encountered in Section 5. A natural choice for this set is $F_1 \times F_2$, although other choices are possible. If both observers are negative, i.e., they both describe safety properties, then reaching $F_1 \times F_2$ in the observer $(\omega_1 \parallel \omega_2, F_1 \times F_2)$ expresses violation of both properties. If both observers are positive (for possibility properties), then reaching $F_1 \times F_2$ denotes satisfaction of both. If the recognising locations are, e.g., $F_1 \times (L_{\omega_2} \setminus F_2)$, then the corresponding observer denotes violation of the first safety property and not of the second one, or, for possibility properties, satisfaction of the first property but not of the second one.

It is also possible to give an interpretation of the parallel product of a positive and a negative observer. For example, if (ω_1, F_1) is positive and (ω_2, F_2) is negative, reaching $F_1 \times F_2$ in $\omega_1 \parallel \omega_2$ means that the safety property has been violated and the possibility property has been satisfied, that is, the two properties are in conflict because the "bad thing" prohibited by the safety property coincides, on some traces, with the "good thing" required by the possibility property.

4.2 Conformance Relation: ioco

A *conformance relation* formalizes the set of implementations that behave consistently with a specification. An implementation $\mathcal{I}$ is not a formal object (it is a physical system) but, in order to reason about conformance, it is necessary to assume that the semantics of $\mathcal{I}$ can be modeled by a formal object. We assume here that it is modelled by an IOLTS (see Definition 2). The notions of trace and quiescence are defined for IOLTS just as for IOSTS. The implementation is assumed to be *input-complete*, i.e., all its inputs are enabled in all states and has the same interface (inputs and output actions with their signatures) as the specification. These assumptions are called *test hypothesis* in conformance testing. The standard **ioco** relation defined by Tretmans [28] can be rephrased in the following way:

Definition 20 (ioco). *An implementation $\mathcal{I}$* **ioco***-conforms to a specification $\mathcal{S}$, denoted by $\mathcal{I}$* **ioco** *$\mathcal{S}$, if*

$$Traces(\mathcal{S}^\delta) \cdot (\Lambda^! \cup \{\delta\}) \cap Traces(\mathcal{I}^\delta) \subseteq Traces(\mathcal{S}^\delta).$$

An implementation $\mathcal{I}$ **ioco**-conforms to its specification $\mathcal{S}$, if, after each trace of the suspension IOSTS $\mathcal{S}^\delta$, the implementation only exhibits outputs and quiescences allowed by $\mathcal{S}^\delta$. In this framework, the specification may be *partial* with respect to inputs, i.e., after an input that is not described by the specification, the implementation can have any behavior without violating conformance to the specification. This corresponds to the intuition that a specification models a given set of services that must be provided by a system; a particular implementation of the system may implement more services than specified, but these additional features should not influence its conformance.

Example 5. An implementation $\mathcal{I}$ that exhibits the trace $START(1) \cdot STOP$ does not conform to the specification $\mathcal{S}$ depicted in Fig. 1—this trace is not present in the IOSTS $\mathcal{S}^\delta$ (Fig. 2). For the same reason, the trace $START(1) \cdot \delta$ reveals a nonconformance to $\mathcal{S}$. On the other hand, an implementation $\mathcal{I}'$ that performs $START(1) \cdot START(1) \cdot STOP$ (followed by a self-loop on all inputs) does conform to $\mathcal{S}$, as $\mathcal{S}^\delta$ does not constrain the traces of the implementation after the second $START$ in any way.

The implementation $\mathcal{I}'$ also demonstrates that **ioco**-conformance does not preserve safety properties. Indeed, $\mathcal{I}'$ conforms to the specification $\mathcal{S}$ depicted in Fig. 1; $\mathcal{S}$ satisfies the safety property described by the observer depicted in Fig. 1 but $\mathcal{I}'$ violates that property.

5 Test Generation

This section shows how to generate tests for checking conformance to a given specification, as well as checking other safety or possibility properties. The test cases attempt to detect violations or satisfactions of the properties by an implementation of the system and violations of the conformance between the implementation and the specification. In addition, if the verification steps described in the previous section did not completely succeed, test execution may also detect violation or satisfaction of properties by the *specification*. Hence, the testing step may sometimes complete verification. We prove that the test cases generated by our method always return correct verdicts. In this sense, the test generation method itself is correct.

Outline. We first define the *output-completion* $\Sigma^!(M)$ of a deterministic IOSTS M. We then show that the output-completion of the IOSTS $\det(\mathcal{S}^\delta)$, where $\det()$ is the determinization operation, is a *canonical tester* [9] for $\mathcal{S}$ and

the **ioco** relation defined in Section 4.2. A canonical tester for a specification with respect to a given relation makes it possible, in principle, to detect every implementation that disagrees with the specification according to the relation. This derives from the fact, stated in Proposition 1 below, that **ioco**-conformance to a specification $\mathcal{S}$ is equivalent to satisfying (a safety property described by) an observer obtained from $\Sigma^!(\det(\mathcal{S}^\delta))$. By composing this observer of nonconformance with other observers for safety or possibility properties, we obtain test cases for checking the conformance to $\mathcal{S}$ and the satisfaction/violation of the properties.

Definition 21 (Output completion). *For a deterministic IOSTS $M = \langle V, P, \Theta, L, l^0, \Sigma, \mathcal{T}\rangle$, the output completion of M is the IOSTS denoted by*

$$\Sigma^!(M) = \langle V, P, \Theta, L \cup \{Fail\}, l^0, \Sigma, \mathcal{T} \cup \{\langle l, a, \bigwedge_{t:\ \langle l,a,G_t,A_t,l'\rangle \in \mathcal{T}} \neg G_t, (x := x)_{x\in V}, Fail\rangle | l \in L, a \in \Sigma^!\}\rangle,$$

where $Fail \notin L$.

Interpretation. $\Sigma^!(M)$ is obtained from M by adding a new location $Fail \notin L$ and, for each $l \in L$ and $a \in \Sigma^!$, a transition with origin l, destination $Fail$, action a, identity assignments and guard $\bigwedge_{\langle l,a,G_t,A_t,l'_t\rangle\in\mathcal{T}} \neg G_t$. Hence, any output not fireable in M becomes fireable in $\Sigma^!(M)$ and leads to the new (deadlock) location $Fail$. The output-completion of an IOSTS M can be seen as a negative observer by choosing $\{Fail\}$ as the set of recognizing locations. The following lemma says that conformance to a specification $\mathcal{S}$ is a safety property, namely, the property represented by the negative observer $(\Sigma^!(\det(\mathcal{S}^\delta)), \{Fail\})$.

Proposition 1. $\mathcal{I}$ **ioco** $\mathcal{S}$ *iff* $\mathcal{I}^\delta \models^- (\Sigma^!(\det(\mathcal{S}^\delta)), \{Fail\})$.

We first need to introduce the following technical lemma (whose proof is omitted):

Lemma 4. *For a deterministic IOSTS M,*

$$Traces(\Sigma^!(M), \{Fail\}) = Traces(M).\Lambda^!_M \setminus Traces(M).$$

Proof of Proposition 1. By Definition 20, $\mathcal{I}$ **ioco** $\mathcal{S}$ is

$$Traces(\mathcal{S}^\delta) \cdot (\Lambda^! \cup \{\delta\}) \cap Traces(\mathcal{I}^\delta) \subseteq Traces(\mathcal{S}^\delta),$$

which is equivalent to $Traces(\mathcal{I}^\delta) \cap [Traces(\mathcal{S}^\delta) \cdot (\Lambda^! \cup \{\delta\}) \setminus Traces(\mathcal{S}^\delta)] = \emptyset$. By Lemma 4 and Lemma 3,

$$Traces(\mathcal{S}^\delta) \cdot (\Lambda^! \cup \{\delta\}) \setminus Traces(\mathcal{S}^\delta) = Traces(\Sigma^!(\det(\mathcal{S}^\delta)), \{Fail\}).$$

Hence, $\mathcal{I}$**ioco**$\mathcal{S}$ is equivalent to

$$Traces(\mathcal{S}^\delta) \cap Traces(\Sigma^!(\det(\mathcal{S}^\delta)), \{Fail\}) = \emptyset,$$

which, by Definition 19, is equivalent to

$$\mathcal{I}^\delta \models^- (\Sigma^!(\det(\mathcal{S}^\delta)), \{Fail\}).$$

□

In the rest of the paper we denote the IOSTS $\Sigma^!(\det(\mathcal{S}^\delta))$ by $canon(\mathcal{S})$. Now, the statement in Proposition 1, $\mathcal{I}^\delta \models^- (canon(\mathcal{S}), \{Fail\})$ can be interpreted as follows: The execution of $canon(\mathcal{S})$ in parallel with the implementation $\mathcal{I}$ never reaches the $Fail$ location. By Proposition 1, this is equivalent to $\mathcal{I}$ **ioco** $\mathcal{S}$; hence, $canon(\mathcal{S})$ and its $Fail$ location completely characterize **ioco**-conformance of $\mathcal{I}$ to $\mathcal{S}$, which amounts to saying that $(canon(\mathcal{S}), Fail\})$ is a *canonical tester* [9] for **ioco**-conformance to the specification $\mathcal{S}$.

A canonical tester is, in principle, enough for detecting all implementations that do not conform to a given specification. However, our goal is to detect, in addition to such nonconformances, the violations/satisfactions of other (additional) safety/possibility properties coming from, e.g., the system's requirements. The observers expressing such properties can also serve as mechanisms for test selection. Using Lemma 1, the product between an observer and the canonical tester defines a subset of "interesting" traces among all the traces generated by the canonical tester.

Definition 22 ($test(\mathcal{S}, \omega^+, \omega^-)$). *Consider a safety property defined by a negative observer $(\omega^-, Violate)$ and a possibility property defined by a positive observer $(\omega^+, Satisfy)$, where both observers are assumed to be compatible, in the sense of Definition 8, with $\mathcal{S}^\delta$.*[4] *Then, the parallel product $\omega^+||canon(\mathcal{S})||\omega^-$ is defined, and it is denoted by $test(\mathcal{S}, \omega^+, \omega^-)$.*

Then, $test(\mathcal{S}, \omega^+, \omega^-)$ can be seen as a test case that refines the canonical tester in the sense that violations/satisfactions of the safety/possibility properties by the implementation $\mathcal{I}$ can also be detected.

Definition 23 (Verdict location sets). *Consider a specification $\mathcal{S}$, and two observers: one negative, $(\omega^-, Violate)$ and one positive: $(\omega^+, Satisfy)$, that are both compatible with $\mathcal{S}^\delta$. We define seven sets of locations of the parallel product $test(\mathcal{S}, \omega^+, \omega^-)$, hereafter called verdict locations sets, or simply verdict locations, as follows:*

1. **Satisfy** $=$
$$Satisfy \times (L_{canon(\mathcal{S})} \setminus \{Fail\}) \times (L_{\omega^-} \setminus Violate).$$
2. **Violate** $=$
$$(L_{\omega^+} \setminus Satisfy \times (L_{canon(\mathcal{S})} \setminus \{Fail\}) \times Violate.$$
3. **Fail** $=$
$$(L_{\omega^+} \setminus Satisfy) \times Fail \times (L_{\omega^-} \setminus Violate).$$
4. **SatisfyFail** $=$
$$Satisfy \times Fail \times (L_{\omega^-} \setminus Violate).$$
5. **ViolateFail** $=$
$$(L_{\omega^+} \setminus Satisfy) \times Fail \times Violate.$$

4. If more than one observer of each kind is present, one "large," global negative (respectively, positive) observer can be built as the product of the "small" negative (respectively, positive) observers.

6. **SatisfyViolate** =

$$Satisfy \times L_{canon(\mathcal{S})} \setminus \{Fail\}) \times Violate.$$

7. **SatisfyViolateFail** =

$$Satisfy \times Fail \times Violate.$$

We use the terminology of "verdict locations" because of the corresponding verdicts obtained by running the observer $test(\mathcal{S}, \omega^+, \omega^-)$ in parallel with an implementation.

Definition 24 (Verdicts). *Consider an observer of the form $test(\mathcal{S}, \omega^+, \omega^-)$ as in Definition 22, equipped with the corresponding verdict location sets, and an implementation $\mathcal{I}$ compatible with $\mathcal{S}$. Then, $\mathcal{I}^\delta$ is also compatible with $\mathcal{S}^\delta$, and then the parallel composition $\mathcal{I}^\delta || test(\mathcal{S}, \omega^+, \omega^-)$ is defined. For each $sigma \in Traces(\mathcal{I}^\delta || test(\mathcal{S}, \omega^+, \omega^-))$ and each*

$$W \in \{\textbf{Satisfy}, \textbf{Violate}, \textbf{Fail}, \textbf{SatisfyFail}, \textbf{ViolateFail}, \textbf{SatisfyViolate}, \textbf{SatisfyViolateFail}\},$$

we say that the trace σ gives verdict W to $\mathcal{I}$ if $\sigma \in Traces(test(\mathcal{S}, \omega^+, \omega^-), W)$.

The following proposition characterizes the verdicts given by a trace to an implementation, from the point of view of the consistency between implementation, specification, and properties.

Proposition 2 (Correctness of test verdicts). *With the notations of Definitions 23 and 24, consider a trace $\sigma \in Traces(test(\mathcal{S}, \omega^+, \omega^-))$ that gives a certain verdict W to an implementation $\mathcal{I}$. Then,*

1. *If $W =$ **Satisfy**, then both $\mathcal{I}^\delta$ and $\mathcal{S}^\delta$ satisfy the possibility property defined by $(\omega^+, Satisfy)$. No violations of the safety property defined by $(\omega^-, Violate)$, and no nonconformances between $\mathcal{I}$ and $\mathcal{S}$ were detected.*[5]
2. *If $W =$ **Violate**, then both $\mathcal{I}^\delta$ and $\mathcal{S}^\delta$ violate the safety property defined by $(\omega^-, Violate)$. No satisfactions of the possibility property defined by $(\omega^+, Satisfy)$ and no nonconformances between $\mathcal{I}$ and $\mathcal{S}$ were detected.*[6]
3. *If $W =$ **Fail**, then there is a nonconformance between $\mathcal{I}$ and $\mathcal{S}$.*[7] *No violations/satisfaction of either of the properties by either $\mathcal{I}^\delta$ or $\mathcal{S}^\delta$ were detected.*
4. *If $W =$ **SatisfyFail**, then $\mathcal{I}^\delta$ satisfies the possibility property and there is a nonconformance between $\mathcal{I}$ and $\mathcal{S}$.*
5. *If $W =$ **ViolateFail**, then $\mathcal{I}^\delta$ violates the safety property and there is a nonconformance between $\mathcal{I}$ and $\mathcal{S}$.*
6. *If $W =$ **SatisfyViolate**, then both $\mathcal{I}^\delta$ and $\mathcal{S}^\delta$ satisfy the possibility property and violate the safety property. No nonconformances between $\mathcal{I}$ and $\mathcal{S}$ were detected.*
7. *If $W =$ **SatisfyViolateFail**, then both $\mathcal{I}^\delta$ and $\mathcal{S}^\delta$ satisfy the possibility property and violate the safety property and there is a nonconformance between $\mathcal{I}$ and $\mathcal{S}$.*

5. This verdict is usually called "Pass" in conformance testing.

6. This is a situation that may only occur if verification of the safety property on the specification did not succeed, and shows that testing can sometimes complement verification.

7. This is the standard "Fail" verdict in conformance testing.

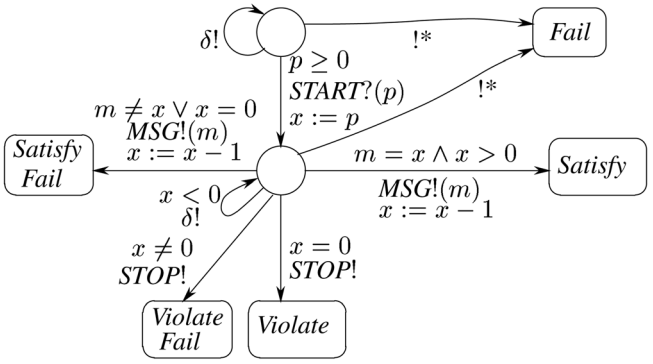


Fig. 5. Test case obtained from $\mathcal{S}$ (Fig. 1).

Proof. We here focus on the proof for the **ViolateFail** verdict (Item 5). The proofs for the other items are very similar and are omitted. We need to prove the following fact: *If a trace $\sigma \in Traces(\mathcal{I}^\delta || test(\mathcal{S}, \omega^+, \omega^-))$ gives verdict* **ViolateFail** *to an implementation $\mathcal{I}$, then $\mathcal{I}^\delta$ violates the safety property and there is a nonconformance between $\mathcal{I}$ and $\mathcal{S}$.*

To prove this fact: the hypothesis is that, in the parallel product $\mathcal{I}^\delta || test(\mathcal{S}, \omega^+, \omega^-)$, the verdict **ViolateFail** is given by some trace. This implies that the **ViolateFail** set of locations is reachable; thus, $Traces(\mathcal{I}^\delta) \cap Traces(test(\mathcal{S}, \omega^+, \omega^-), \textbf{ViolateFail}) \neq \emptyset$.

By Lemma 1 and Lemma 3,

$$Traces(test(\mathcal{S}, \omega^+, \omega^-), \textbf{ViolateFail}) = Traces(canon(\mathcal{S}), Fail) \cap Traces(\omega^-, Violate).$$

Hence, we obtain

$$Traces(\mathcal{I}^\delta) \cap Traces(canon(\mathcal{S}), Fail) \cap Traces(\omega^-, Violate) \neq \emptyset,$$

which implies both

- $Traces(\mathcal{I}^\delta) \cap Traces(canon(\mathcal{S}), Fail) \neq \emptyset$ and
- $Traces(\mathcal{I}^\delta) \cap Traces(\omega^-, Violate) \neq \emptyset$

By Definition 19, we obtain

- $\mathcal{I}^\delta \not\models (canon(\mathcal{S}), Fail)$, which, by Proposition 1, is equivalent to $\neg(\mathcal{I}$ **ioco** $\mathcal{S})$, and
- $\mathcal{I}^\delta \not\models (\omega^-, Violate)$.

That is, the implementation violates both conformance to the specification and the safety property, which establishes Item 5 of the theorem. The other items are proved similarly. □

Example 6. For the IOSTS $\mathcal{S}$ depicted in Fig. 3 and the observers ω_2 depicted in Fig. 3b and ω_3 depicted in Fig. 4, the corresponding test case is depicted in Fig. 5.

The **SatisfyFail** verdict results from the fact that the possibility property is satisfied, e.g., by the trace $START(0) \cdot MSG(0)$, which diverges from the specification. Another interesting verdict is **Violate**: If it is reached, e.g., by the trace $START(0) \cdot STOP$, it indicates

that not only the implementation but also the specification violate the safety property. This means that the verification step failed to detect the latter violation; here, the testing step has completed the information obtained by verification. Finally, the **Fail** verdict denotes nonconformance, and it is reached by all outputs (denoted "!*") that do not label other transitions.

Test execution. To conclude this section, let us imagine what it would mean to *actually* execute the test case depicted in Fig. 5 on a *real* implementation. First, this requires mapping the abstract actions $START$, $STOP$, etc., to concrete inputs and outputs of the implementation. Then, to start running the test on the implementation, the test execution mechanism has to "choose" an output p satisfying $p \geq 0$, send it to the implementation via the corresponding concrete "$START$" input, and wait for the implementation's outputs; depending on the latter, execution here goes to either of the four verdict locations and ends there.

Choosing a value satisfying $p \geq 0$ is a very simple example of *on-the-fly* constraint solving, which, in general, cannot be completely avoided at test execution time because, specifically, the parameters of inputs to be sent to the implementation need to be computed by "solving" guards that constrain them. However, some operations on constraints can be performed *statically* by the *test selection* operation defined in the next section. For example, if, in the above test case, we were "trying" to reach the **ViolateFail** verdict, this would require us to "strengthen" the guard $p \geq 0$ of the $START$-labeled transition to $p > 0$—because, if $p = 0$, the **ViolateFail** verdict is not "reachable," and this can be detected by a static analysis of the test case. Of course, this would not dispense the test execution mechanism to actually *choose* a value satisfying $p > 0$ at test execution time.

The interactions between static constraint strengthening and dynamic constraint solving are discussed in more detail in [23].

6 Test Selection

A test case like $test(\mathcal{S}, \omega^+, \omega^-)$ with so many verdicts is mostly interesting from a theoretical point of view, as all information about nonconformances and satisfaction/violation of several properties are embodied in it. In practice, more focused test cases, which target only one or a few properties at a time, are preferable. This can be performed by the *test selection* operation described below and illustrated on the case study in the next section.

Assume that we have built the test case $test(\mathcal{S}, \omega^+, \omega^-)$ as in the previous section, and we have decided on a subset of the verdicts that we want to target more specifically. These could be any of the verdicts of the test case and are encoded in a subset L_{target} of its verdict locations. For example, if we want to target the satisfaction of the possibility property, the target locations are

$$L_{target} = \mathbf{Satisfy} \cup \mathbf{SatisfyFail} \cup \mathbf{SatisfyViolateFail}.$$

Then, the test selection process consists (ideally) in selecting, from a given test case, the subset of states that are coreachable for L_{target}.

It should be quite clear that an exact computation of this set of states is impossible in general. However, there exist techniques that enable us to compute overapproximations. We here use one such technique based on abstract interpretation and implemented in the NBac tool [33]. The tool computes, for each location l, a *symbolic coreachable state*, which, intuitively, overapproximates the states with location l that are coreachable for a given set of locations L' (see Definition 5):

Definition 25 (Symbolic coreachable state). *For l a location and L' a set of locations of an IOSTS $\mathcal{S}$, we say $\langle l, \varphi_{l \to L'} \rangle$ is a symbolic coreachable state if $\varphi_{l \to L'}$ is a formula such that we have the inclusion $\{\langle l, \nu \rangle | \nu \models \varphi_{l \to L'}\} \supseteq \{\langle l, \nu \rangle | \nu \in \mathcal{V}\} \cap coreach(L')$.*

The following algorithm uses this information for performing test selection for

$$L_{target} \subseteq \mathbf{Satisfy} \cup \mathbf{Violate} \cup \mathbf{Fail} \cup \mathbf{SatisfyViolate} \cup \mathbf{SatisfyFail} \cup \mathbf{ViolateFail} \cup \mathbf{SatisfyViolateFail}:$$

Definition 26 (Test selection).

1. *The test case $test(\mathcal{S}, \omega^+, \omega^-)$ is built as described in Section 5. Let L be its set of locations, $\mathcal{T}$ its set of transitions, and $\Sigma = \Sigma^! \cup \Sigma^?$ its alphabet, where $\Sigma^? = \Sigma^?_{\mathcal{S}}$ and $\Sigma^! = \Sigma^!_{\mathcal{S}} \cup \{\delta\}$. Also let $Inconc \notin L$ be a new location.*
2. *For each location $l \in L$, a symbolic coreachable state $\langle l, \varphi_{l \to L_{target}} \rangle$ is computed.*
3. *Next, for each location $l \in L$ of the IOSTS and each transition $t \in \mathcal{T}$ of the IOSTS with origin l, guard G, and label a,*
   - *if $a \in \Sigma^?$, then*
     - *if $G \wedge \varphi_{l \to L_{target}}$ is unsatisfiable, then t is eliminated from $\mathcal{T}$,*
     - *otherwise, the guard of t becomes $G \wedge \varphi_{l \to L_{target}}$.*
   - *if $a \in \Sigma^!$, then*
     - *the guard of t becomes $G \wedge \varphi_{l \to L_{target}}$*
     - *a new transition is added to $\mathcal{T}$, with origin l, destination $Inconc$, action a, guard $G \wedge \neg \varphi_{l \to L_{target}}$, and identity assignments.*

The test selection operation consists, essentially, of detecting transitions to states that are not coreachable for the target set of locations. This is done by performing a coreachability analysis to these locations using the NBac tool [33]. If such a "useless" transition is labeled by an *input*, then it may be removed from the test case: A test case controls the inputs it provides to the implementation; hence, it may decide not to provide an input if it "knows" that the target locations are unreachable. On the other hand, *outputs* from the implementation cannot be prevented from occurring; hence, the transitions labeled by *outputs*, by which the locations in L_{target} cannot be reached, are reoriented to a new location, called $Inconc$. This location

also corresponds to an "Inconclusive" verdict, whose meaning is that the current test cannot reach the chosen target and, therefore, the current test execution can be stopped. In this way, all traces from the original test case leading to the chosen target L_{target} are preserved, and the correctness of verdicts and verdict enabledness are preserved as well.

Lemma 5. *Let $\sigma : \alpha_1\alpha_2\cdots\alpha_n$ be a trace of a deterministic IOSTS $\mathcal{S}$, let $t_i = \langle l_i, a_i, G_i, A_i, l_{i+1}\rangle$ $(i = 1,\ldots,n)$ be a sequence of transitions that supports σ, such that the last location $l_{n+1} \in L_{target}$, and let $\langle l_i, \varphi_{l_i \to L_{target}}\rangle$ $(i = 1,\ldots n)$ be symbolic coreachability sets for L_{target}. Then, for all $i = 1,\ldots n$, the formula $G_i \wedge \varphi_{l_i \to L_{target}}$ is satisfiable. Moreover, the sequence of* modified transitions *$t'_i = \langle l_i, a_i, G_i \wedge \varphi_{l_i \to L_{target}}, A_i, l_{i+1}\rangle$ $(i = 1 \ldots n)$ is a support for the trace σ as well.*

Proof (sketch). For $i = 1,\ldots,n$, let $\alpha_i = \langle a_i, \pi_i\rangle$. By Lemma 2, there exist states $s_i = \langle l_i, \nu_i\rangle$ such that 1) $\langle \nu_i, \pi_i\rangle \models G_i$. Moreover, by hypothesis, $\langle l_i, \varphi_{l_i \to L_{target}}\rangle$ is a symbolic coreachability set for L_{target} and the location $l_{n+1} \in L_{target}$ is reachable from s_i; hence, by Definition 25, 2) $\nu_i \models \varphi_{l_i \to L_{target}}$. Now, 1 and 2 imply that $\langle v_i, \pi_i\rangle \models G_i \wedge \varphi_{l \to L_{target}}$, and we have proved that $G_i \wedge \varphi_{l_i \to L_{target}}$ is satisfiable.

Then, the states $s_i = \langle l_i, \nu_i\rangle$, valued actions $\alpha_i = \langle a_i, \pi_i\rangle$, and the sequence of transitions $t'_i = \langle l_i, a_i, G_i \wedge \varphi_{l_i \to L_{target}}, A_i, l_{i+1}\rangle$ $(i = 1 \ldots n)$ satisfy all the conditions of Lemma 2, i.e., the sequence $t'_i = \langle l_i, a_i, G_i \wedge \varphi_{l_i \to L_{target}}, A_i, l_{i+1}\rangle$ $(i = 1 \ldots n)$ is a support for the trace $\alpha_1\alpha_2\cdots\alpha_n$.□

Correctness of verdicts after test selection. We denote by $select(test(\mathcal{S}, \omega^+, \omega^-), L_{target})$ the result of applying the selection operation to the test case $test(\mathcal{S}, \omega^+, \omega^-)$ for a target $L_{target} \subseteq L_{verdict}$, where $L_{verdict}$ has been defined as the union

$$\mathbf{Satisfy} \cup \mathbf{Violate} \cup \mathbf{Fail} \cup \mathbf{SatisfyViolate} \cup \mathbf{ViolateFail} \cup \mathbf{SatisfyFail} \cup \mathbf{SatisfyViolateFail}.$$

We also denote by Λ the set of valued actions (only valued inputs and valued outputs, including quiescence —internal actions have been eliminated) of the test case, and the test case itself by $TC \triangleq test(\mathcal{S}, \omega^+, \omega^-)$.

The correctness of the selection procedure then consists in establishing the following proposition:

Proposition 3 (Correctness of test cases after selection). *Let $TC = test(\mathcal{S}, \omega^+, \omega^-)$ a test case generated as described in Section 4, and let Λ be the set of valued actions (valued inputs and valued outputs, including quiescence) of the test case. Then,*

1. $$Traces(select(TC, L_{target}), L_{target}) = Traces(TC, L_{target}).$$

   That is, the traces that reach the target before the test selection still do so after test selection, and test selection does not add new traces leading to the target.

2. $$Traces(select(TC, L_{target}), L_{verdict} \setminus L_{target}) \subseteq Traces(TC, L_{verdict} \setminus L_{target}).$$

   That is, for verdicts other than the target, selection does not add new traces leading to them.

3. $$Traces(select(TC, L_{target}), Inconc) \cdot \Lambda^* \cap Traces(select(TC, L_{target})) = \emptyset.$$

   That is, if the inconclusive verdict has been reached, then the target cannot be reached anymore.

Proof (sketch). Let us first prove inclusion 1.

$$Traces(select(TC, L_{target}), L_{target}) \subseteq Traces(TC, L_{target}).$$

Let $\sigma \in Traces(select(TC, L_{target}), L_{target})$. Then, by Lemma 2, there exists a sequence of transitions of $select(TC, L_{target})$ that supports σ and, by Definition 26 of the selection operation, this sequence is of the form $t'_i = \langle l_i, a_i, G_i \wedge \varphi_{l_i \to L_{target}}, A_i, l_{i+1}\rangle$ $(i = 1 \ldots n)$, such that, for all $i = 1 \ldots n$, $\langle l_i, a_i, G_i, A_i, l_{i+1}\rangle$ is a transition of TC, and $l_{n+1} \in L_{target}$. Indeed, all locations l_i of the sequence must be locations of TC—the only location of $select(TC, L_{target})$ that is not a location of TC is $Inconc$, which is a deadlock, i.e., L_{target} is not reachable from it, which contradicts $l_{n+1} \in L_{target}$, and $select(TC, L_{target})$ is obtained by strengthening the guards of transitions of TC with the predicates $\varphi_{l_i \to L_{target}}$.

Then, the sequence of transitions of $TC : t_i = \langle l_i, a_i, G_i, A_i, l_{i+1}\rangle$ $(i = 1 \ldots n)$ satisfies all the conditions of Lemma 2, i.e., it supports the sequence σ, which implies $\sigma \in Traces(TC, L_{target})$.

$$Traces(select(TC, L_{target}), L_{target}) \supseteq Traces(TC, L_{target}).$$

Let $\sigma \in Traces(TC, L_{target})$. Then, by Lemma 2, there exists a sequence of transitions of $TC : t_i = \langle l_i, a_i, G_i, A_i, l_{i+1}\rangle$ $(i = 1 \ldots n)$ that supports the sequence σ, and $l_{n+1} \in L_{target}$. By Lemma 5, the sequence of modified transitions $t'_i = \langle l_i, a_i, G_i \wedge \varphi_{l_i \to L_{target}}, A_i, l_{i+1}\rangle$ $(i = 1 \ldots n)$ supports σ as well. All the transitions of the sequence (t'_i) $(i = 1 \ldots n)$ are transitions of $select(TC, L_{target})$; indeed (see Definition 26), the only transitions of TC that are removed, or whose destination is changed to $Inconc$ by the selection operation, are such that $G_i \wedge \varphi_{l_i \to L_{target}}$ is unsatisfiable, which contradicts Lemma 5. Hence, σ is supported by the sequence (t'_i) of transitions of $select(TC, L_{target})$ which ends in L_{target}, i.e., $\sigma \in Traces(select(TC, L_{target}), L_{target})$.

The proof of inclusion 2 is similar to the proof of the $\subseteq$ inclusion 1.

Let $\sigma \in Traces(select(TC, L_{target}), Inconc)$, and assume there exists an extension

$$\sigma \cdot \sigma' \in Traces(select(TC, L_{target}), L_{target}).$$

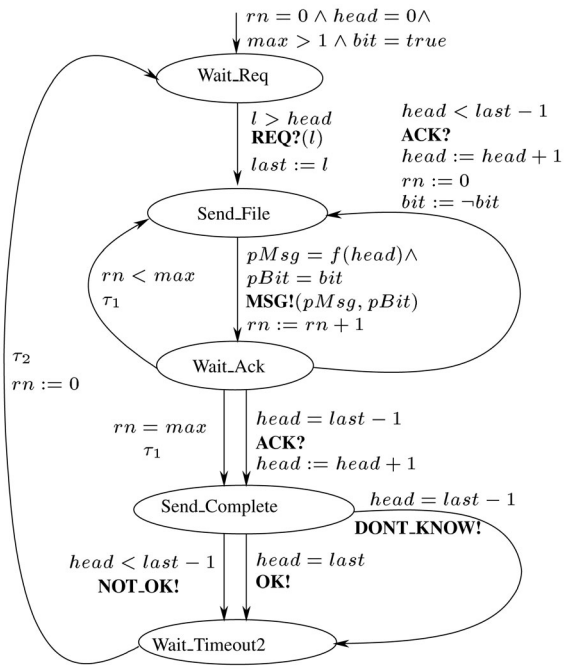


Fig. 6. Sender of the BRP.

Then, by Lemma 2, the trace $\sigma \cdot \sigma'$ has a unique support in $select(TC, L_{target})$, i.e., a sequence of contiguous transitions $(t_i)_{i=1,\ldots n}$ of $select(TC, L_{target})$, and its prefix σ also has a unique support, which is necessarily a strict prefix of the sequence $(t_i)_{i=1,\ldots n}$. Let t_j $(j < n)$ be the last transition in the support of $\sigma \in Traces(select(TC, L_{target}), Inconc)$. Then, the destination of t_j is $Inconc$, which is a deadlock; hence, the sequence (t_i) actually stops at t_j $(j < n)$, i.e., it is not a support for $\sigma \cdot \sigma'$, a contradiction. □

7 Example: The BRP Protocol

We now give a larger example, in which we present all the operations of test generation and selection defined up to this point. We consider a Bounded Retransmission Protocol [25], whose role is to transmit data in a reliable manner over an unreliable network. We focus on the sender of the protocol.

Specification. The specification of the sender is depicted in Fig. 6. Execution starts by receiving a request **REQ** with an integer parameter l. The meaning of the parameter l is that the current session must transmit $f(head), f(head + 1), \ldots f(l - 1)$, where f is the file to transmit, and the variable $head$, initialized to 0, is the index of the next "piece" of the file to be transmitted. Hence, the request makes sense only if $l > head$.

Then, the sender proceeds by transmitting mes-sages **MSG**, together with the data to transmit: $f(head)$, $f(head + 1), \ldots$, and an alternating bit allowing the receiver to distinguish between new messages and retransmissions. After each message, the sender waits for an acknowledgment **ACK** after which it proceeds by sending the next message. If the acknowledgment does not arrive, the current message is retransmitted at most $max - 1$ times,

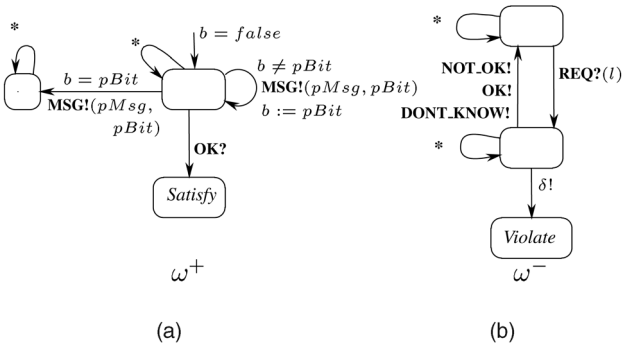


Fig. 7. Properties. (a) Possibility. (b) Safety.

where max is a symbolic constant of the protocol. The number of retransmissions is counted by the variable rn. Globally, the transmission terminates successfully when the last message has been transmitted and acknowledged. In this case, the sender confirms the success to its client by sending it an OK confirmation. The transmission may also terminate unsuccessfully (confirmation **NOT_OK**) if some message (except the last one) is not acknowledged, or the outcome may be unknown (confirmation **DONT_KNOW**) if the last message is not acknowledged—either the message or its acknowledgment could have been lost.

Properties. We consider here two properties of the sender, represented by the observers depicted in Fig. 7. The possibility property, in the left-hand side of the figure, describes an ideal scenario in which the sender transmits each message exactly once, without retransmissions. The scenario ends in the $Satisfy$ location after the **OK** confirmation. Retransmissions lead to the location represented at the extreme left of the figure, from which the $Satisfy$ location is unreachable. The observer uses a Boolean variable b, which is used to distinguish new messages from retransmissions. The safety property, depicted in the right-hand side of the figure, expresses the fact that the BRP sender is not allowed to be blocked between the request **REQ** and any of the confirmations **OK**, **NOT_OK**, or **DONT_KNOW.** If the specialy output δ, denoting blocking/quiescence, is observed between request and confirmation, the $Violate$ location is reached, which expresses violation of the safety property. All other valued actions (that do not explicilty appear in the automata representing the properties) are denoted by the *-labeled self-loops.

Verification. Both properties are satisfied by the specification. For the safety property, this is proved using the abstract interpretation-based tool NBac, and the possibility property is proved by model checking.

Test Generation. We follow here the test generation steps described in Section 5.

Suspension. The suspended IOSTS $Sender^{\delta}$ corresponding to the $Sender$ of the BRP is depicted in Fig. 8. In location *Wait_Req*, the system is blocked waiting for an input from the environment, hence, it has a self-loop labeled with the suspension action δ. The other locations where δ is potentially firable are *Wait_Ack* and *Send_Complete*. For example, in *Wait_Ack*, the guard of the δ-labeled self loop is $rn > max$, which is the complement of the two conditions $(rn = max,\ rn < max)$ labeling transitions by which the

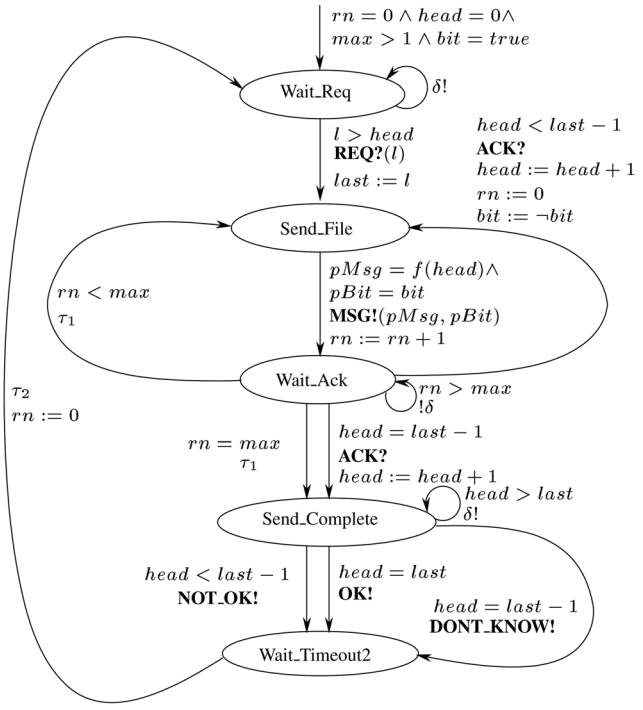


Fig. 8. Suspended BRP.

system can leave the location without intervention of the environment.

Determinization and output-completion. The determinized system $\det(Sender^{\delta})$ obtained from $Sender^{\delta}$ is depicted in Fig. 9. Here, determinization consisted in eliminating the internal actions τ_1, τ_2 as described in Section 3.3. Note that $\det(Sender^{\delta})$ is not, strictly speaking, deterministic, because its initial condition is satisfied by more than one (actually, by infinitely many) valuations of the symbolic constants max and f. However, any instance of $\det(Sender^{\delta})$ obtained by instantiating max and f to actual values is deterministic, because instantiating max and f to actual values determines unique values for all the other variables. All the IOSTS

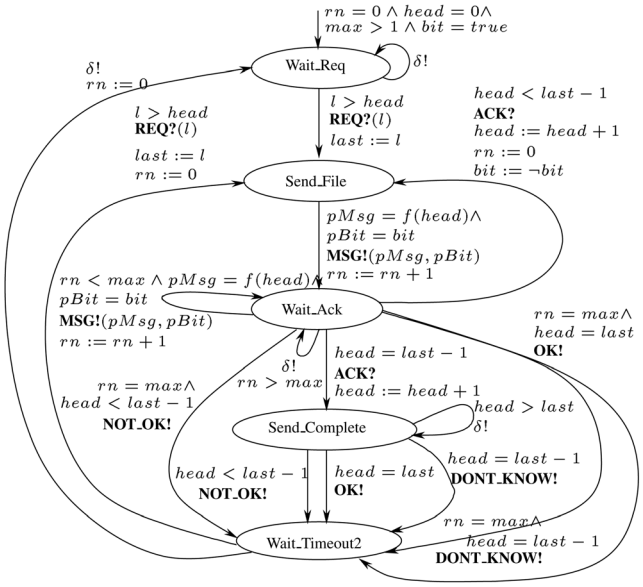


Fig. 9. Determinized BRP.

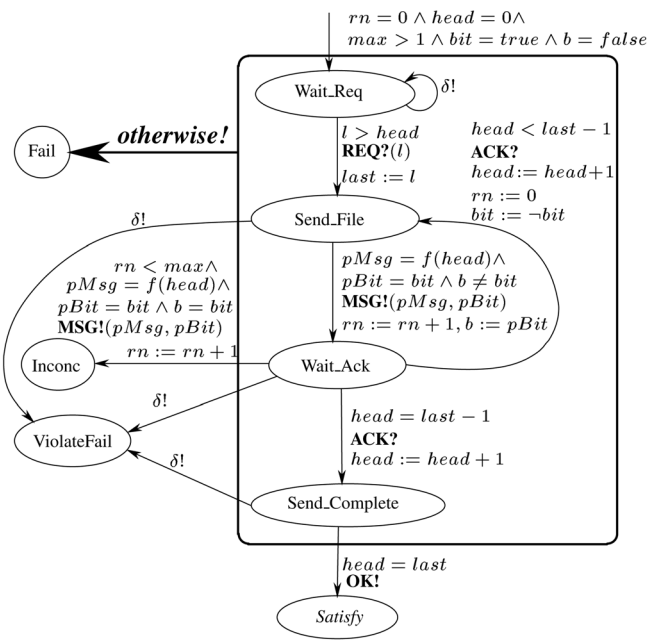


Fig. 10. Test case obtained by selection for $Satisfy$.

mentioned hereafter in this section also have this property, which is weaker than determinism but strong enough for test cases since, at test execution time, max and f will have concrete values.

The canonical tester has one more location, *Fail*, and implicit transitions from each location to it, as explained in Section 3.3 by the output-completion operation.

Test Selection. The test case obtained by specializing the canonical tester to the properties depicted in Fig. 7, followed by selection targeting the *possibility* property, is depicted in Fig. 10. Several locations are grouped into so-called "macrostates" à la Statecharts.

The "macrotransitions" leaving the macrostates represent several transitions in the sense of IOSTS. For instance, the transition labeled **otherwise** represents all transitions labeled by outputs that are not allowed by the specification, and their destination $Fail$ denotes the **Fail** verdict—nonconformance between implementation and specification. The macrotransition labeled δ represents three transitions with the same label, originating in three locations of the test case. Their destination $ViolateFail$ and the corresponding **ViolateFail** verdict indicates violation of the property by the implementation and violation of the conformance between implementation and specification. The **Satisfy** verdict is given when the positive **OK** confirmation is observed by the tester, and corresponds to the satisfaction of the possibility property. Finally, the $Inconc$ location and verdict mean that, from there on, the possibility property cannot be satisfied anymore. Since both properties (safety and possibility) are satisfied by the specification, these are the only possible verdicts here. There are no unreachable verdicts (and no unreachable locations at all), which means that, in this case, the approximations performed by NBac at test selection time were quite good. This is not always the case, but approximations can, in principle, be made "as precise as desired" by using better abstraction functions and/or

unfolding IOSTS executions up to a bounded depth before computing abstractions.

Finally, let us consider again the problem of *actually executing* such a test case on a *real* implementation. Besides mapping abstract inputs and outputs to concrete actions of the implementation, we need to instantiate the symbolic constants max and f to concrete values corresponding to those of the implementation, which are supposed to be known.[8] Then, each time the implementation needs an *input*, carrying some parameter, on-the-fly *constraint solving* is performed to compute that parameter's value, and each time the implementation provides an output, with or without a parameter, only *constraint satisfaction* (i.e., computation, which, in general, is easier than solving) is performed to compute the next transition that fires (which has been made unique by determinisation). In the above particular case, the constraint-solving is rather simple, as the only constraints to be solved are of the form $parameter = variable$, i.e., the variable's value at test execution time will be chosen as the value of the corresponding parameter. In general, on-the-fly constraint solving is not so simple, especially when complex data types are involved. The interplay between statically analyzing the test case for trying to target a certain (set of) location verdict(s), and the on-the-fly constraint solving, remains a matter for future work.

8 Conclusion and Future Work

A system may be viewed at several levels of abstraction: high-level *properties*, operational *specification*, and black-box *implementation*. In our framework, properties and specifications are described using Input-Output Symbolic Transition Systems (IOSTS), which are extended automata that operate on symbolic variables and communicate with the environment through input and output actions carrying parameters. IOSTS are given a formal semantics in terms of input-output labeled transition systems (IOLTS). The implementation is a black box, but it is assumed that its semantics can be described by an unknown IOLTS. This enables the formal linking of the implementation and the specification by a conformance relation. A satisfaction relation links them both to higher-level properties: safety properties, which express that something bad never happens, and possibility properties, which express that something good can happen.

A methodology is proposed for checking the consistency between the different views of the system: First, the properties are tentatively verified on the specification using automatic approximate analysis techniques, which are sound but, because of undecidability problems, are inherently incomplete. Then, test cases are automatically generated from the specification and the properties and are executed on the implementation of the system. If the verification step was successful, that is, it has proved or disproved each property on the specification, then, test execution may detect violation or satisfaction of the properties by the implementation and the violation of the conformance relation between implementation and specification. On the other hand, if the verification did not enable the proving or disproving of some properties, the test execution may additionally detect violation or satisfaction of the properties by the specification. In this sense, the testing step completes the verification step. The approach is illustrated on a simple example as well as on a larger example (a BRP protocol [25]).

Future Work One important issue that needs to be solved is that of *coverage*: defining a coverage measure on the specification and/or the properties and computing, e.g., at test execution time, how much of the measure has been performed. Another issue, which is solved in this paper, but not in a completely satisfying manner, is the selection of test *inputs*. We solve it here by on-the-fly constraint solving, which is computationally expensive and may, in practice, prevent actual test execution to happen (e.g., if no output value is computed in due time, some behaviors of the implementation will never be tested, especially if the implementation uses timers to control the delay during which it is willing to wait for inputs). It is an interesting question to understand how much of the test data selection can be performed offline by static analysis, and how much of it must remain on the fly.

The problem of test selection and test data selection bears a close resemblance to searching for *strategies* in *games*, in which the implementation can be seen as playing against the test case; the test case wins if it manages to reach the verdicts that it targets, and the implementation attempts to prevent the tester from reaching those verdicts.

Finally, the integration of testing systems with data and of *timed* testing, for which many interesting results have recently emerged, is another important future research topic.

8. This small breach in the black-box testing dogma is admitted in conformance testing and bears the name PIXIT (Protocol Implementation eXtra Information for Testing).

References

[1] ISO/IEC 9646, "Conformance Testing Methodology and Framework," 1992.

[2] E. Brinskma, A. Alderen, R. Langerak, J. van de Laagemat, and J. Tretmans, "A Formal Approach to Conformance Testing," *Proc. Conf. Protocol Secification, Testing, and Verification (PSTV '90)*, pp. 349-363, 1990.

[3] J. Tretmans, "Test Generation with Inputs, Outputs, and Repetitive Quiescence," *Software—Concepts and Tools*, vol. 17, no. 3, pp. 103-120, 1996.

[4] A. Gargantini and C.L. Heitmeyer, "Using Model Checking to Generate Tests from Requirements Specifications," *Proc. European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng. (ESEC/FSE '99)*, pp. 146-162, 1999.

[5] C. Jard and T. Jéron, "TGV: Theory, Principles and Algorithms, a Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems," *Software Tools for Technology Transfer*, vol. 6, Oct. 2004.

[6] J. Blom, A. Hessel, B. Jonsson, and P. Pettersson, "Specifying and Generating Test Cases Using Observer Automata," *Proc. Workshop Formal Approaches to Software Testing (FATES '04)*, J. Grabowski and B. Nielsen, eds., 2004.

[7] H.S. Hong, I. Lee, O. Sokolsky, and H. Ural, "A Temporal Logic Based Theory of Test Coverage and Generation," *Proc. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02)*, pp. 327-341, Apr. 2002.

[8] O. Kupferman and M.Y. Vardi, "Model Checking of Safety Properties," *Formal Methods in System Design*, vol. 19, no. 3, pp. 291-314, 2001.

[9] E. Brinskma, "A Theory for the Derivation of Tests," *Proc. Conf. Protocol Specification, Testing, and Verification (PSTV '88),* pp. 63-74, 1988.

[10] P. Ammann, W. Ding, and D. Xu, "Using a Model Checker to Test Safety Properties," *Proc. Int'l Conf. Eng. Complex Computer Systems,* 2001.

[11] G. Hamon, L. de Moura, and J. Rushby, "Generating Efficient Test Sets with a Model Checker," *Proc. Second Int'l Conf. Software Eng. and Formal Methods,* pp. 261-270, Sept. 2004.

[12] J. Blom, A. Hessel, B. Jonsson, and P. Pettersson, "Specifying and Generating Test Cases Using Observer Automata," *Proc. Formal Approaches to Software Testing,* J. Grabowski and B. Nielsen, eds., pp. 137-152, 2004.

[13] H. Hong, I. Lee, O. Sokolsky, and H. Ural, "A Temporal Logic Based Theory of Test Coverage and Generation," *Proc. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02),* pp. 327-341, 2002.

[14] D. Peled, M. Vardi, and M. Yannakakis, "Black-Box Checking," *J. Automata, Languages, and Combinatorics,* vol. 7, no. 2, pp. 225-246, 2001.

[15] J.C. Fernandez, L. Mounier, and C. Pachon, "Property-Oriented Test Generation," *Proc. Formal Aspects of Software Testing Workshop,* 2003.

[16] R. de Vries and J. Tretmans, "Towards Formal Test Purposes," *Formal Approaches to Testing of Software (FATES '01),* pp. 61-76, 2001.

[17] A. Belinfante, J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, and S. Mauw, "Formal Test Automation: A Simple Experiment," *Proc. Int'l Workshop Testing of Comm. Systems (IWTCS '99),* 1996. pp. 179-196

[18] K. Havelund and G. Rosu, "Synthesizing Monitors for Safety Properties," *Proc. Int'l Conf. Tools and Algorithms for Construction and Analysis of Systems (TACAS '02),* pp. 342-356, 2002.

[19] M. Bozga, J.-C. Fernandez, L. Ghirvu, C. Jard, T. Jéron, A. Kerbrat, P. Morel, and L. Mounier, "Verification and Test Generation for the SSCOP Protocol," *J. Science of Computer Programming,* special issue on formal methods in industry, vol. 36, no. 1, pp. 27-52, Jan. 2000.

[20] V. Rusu, H. Marchand, V. Tschaen, T. Jéron, and B. Jeannet, "From Safety Verifcation to Safety Testing," *Proc. Int'l Conf. Testing Comm. Systems (TestCom),* 2004.

[21] V. Rusu, H. Marchand, and T. Jéron, "Automatic Verification and Conformance Testing for Validating Safety Properties of Reactive Systems," *Proc. Symp. Formal Methods Europe (FM),* 2005.

[22] V. Rusu, L. du Bousquet, and T. Jéron, "An Approach to Symbolic Test Generation," *Proc. Int'l Conf. Integrating Formal Methods (IFM '00),* pp. 338-357 2000.

[23] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva, "Symbolic Test Selection Based on Approximate Analysis," *Proc. 11th Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05),* Apr. 2005.

[24] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," *Proc. Fourth Symp. Principles of Programming Languages (POPL),* pp. 238-252, Jan. 1977.

[25] L. Helmink, M.P.A. Sellink, and F. Vaandrager, "Proof-Checking a Data Link Protocol," *Proc. Conf. Types for Proofs and Programs (TYPES '94),* pp. 127-165, 1994.

[26] N. Lynch and M. Tuttle, "Introduction to IO Automata," *CWI Quarterly,* vol. 3, no. 2, 1999.

[27] R.E. Shostak, "A Practical Decision Procedure for Arithmetic with Function Symbols," *J. ACM* vol. 26, no. 2, pp. 351-360, 1979.

[28] J. Tretmans, "Testing Concurrent Systems: A Formal Approach," *Proc. Conf. Conccurrency Theory (CONCUR '99)* pp. 46-65, 1999.

[29] T. Jéron, H. Marchand, and V. Rusu, "Symbolic Determinisation of Extended Automata," *Proc. Fourth IFIP Int'l Conf. Theoretical Computer Science,* 2006.

[30] E. Zinovieva, "Symbolic Test Generation for Reactive Systems," PhD thesis, Univ. of Rennes I, Nov. 2004.

[31] E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking.* MIT Press, 1999.

[32] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva, "STG: A Symbolic Test Generation Tool," *Proc. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02)* pp. 470-475, 2002.

[33] B. Jeannet, "Dynamic Partitioning in Linear Relation Analysis," *Formal Methods in System Design,* vol. 23, no. 1, pp. 5-37, 2003.

Camille Constant received the master's degree in computer science from the University of Rennes 1 in 1999. Since October 1999, she has been a PhD student at IRISA in Rennes in the VerTeCs project. Her research interests include verification and automatic test generation in reactive systems.

Thierry Jéron received the PhD degree in computer science from the University of Rennes 1, in the IRISA laboratory in 1991. He then spent one year as a research engineer in the Alcatel research laboratory. He came back to IRISA in 1993 as an INRIA research scientist. He has been the scientific leader of the VerTeCS team since 2001 and has been the research director since 2006. His research topics concern the formal verification and validation of reactive systems, including model-checking, model-based testing, control synthesis and diagnosis of discrete event systems. He is author or co-author of around 50 international publications in these domains and has been involved in a number of academic and industrial collaborations.

Hervé Marchand received the master's degree in mathematics from the Universiy of Rennes 1 in 1993 and the PhD degree in computer science from the University of Rennes 1 in October 1997. From November 1997 to October 1998, he was a postdoctoral fellow at the University of Michigan, Ann Arbor. Since 1998, he has held an INRIA research position at IRISA in Rennes in the VerTeCs project. His research interests include supervisory control, automatic test generation and diagnosis of discrete events systems. He is also interested in high-level languages for reactive and real-time systems.

Vlad Rusu received the master's degree in computer science in 1993 and the PhD degree in computer science in January 1996, both from the University of Nantes, France. He was a research and teaching assistant at the University of Nantes in 1996 and 1997. He then visited SRI International in Menlo Park, California, where he was a postdoctoral fellow from 1997 to 1999. Since January 1999, he has held an INRIA research position at IRISA Rennes, France. His research interests include formal methods for verification and testing of reactive systems.