Automatic Verification and Conformance Testing for Validating Safety Properties of Reactive Systems*

Vlad Rusu, Hervé Marchand, and Thierry Jéron
First.Last@irisa.fr

IRISA/INRIA, Campus de Beaulieu, Rennes, France

Abstract This paper presents a combination of verification and conformance testing techniques for the formal validation of reactive systems. A formal specification of a system, which may be infinite-state, and a set of safety properties are assumed. Each property is verified on the specification using automatic techniques based on abstract interpretation, which are sound, but, as a price to pay for automation, are not necessarily complete. Next, for each property, a test case is automatically generated from the specification and the property, and is executed on a black-box implementation of the system to detect violations of the property by the implementation and non-conformances between implementation and specification. If the verification step did not conclude, the test execution may also detect violations of the property by the specification.

Keywords: verification, conformance testing, symbolic test generation

1 Introduction

Formal verification and conformance testing are two well-established approaches for validating reactive systems. Both approaches consist in checking the consistency between two representations of a system:

- formal verification typically compares a formal *specification* of the system with respect to some higher-level required *properties*;
- conformance testing [1,5] compares the observable behaviour of a black-box *implementation* of the system with that described by the specification.

A formal validation chain for reactive systems, combining verification and conformance testing, may naturally consist of the following steps:

1. the properties are automatically verified on the specification;
2. test cases are automatically derived from the specification and the properties;
3. the test cases are executed on the black-box implementation of the system, to check the satisfaction of the properties by the implementation and the conformance between implementation and specification.

* The full version of this paper is available as IRISA report [17].

In this paper we formally define and study such a validation chain. We consider a general class of specifications which may be infinite-state (automata extended with variables, which communicate with the environment by means of inputs and outputs carrying parameters). In this setting, the verification step (in particular, for safety properties) is undecidable. In order to keep it automatic and ensure that it always terminates, we adopt approximate, conservative verification techniques based on abstract interpretation [7], which may either prove the property, or terminate with a "don't know" answer.

The main contribution of the paper lies in the second step of the proposed validation chain. It is a test generation algorithm that takes into account the infinite-state nature of the specifications and the incompleteness of the verification step. The algorithm takes as inputs a specification and a safety property, and produces a test case for checking the conformance between a given implementation and the specification, and the satisfaction of the safety property by the implementation. To deal with infinite-state specifications and properties, the algorithm is *symbolic*: it does not attempt to enumerate the (potentially infinite) domain of the specification's variables, but deals with the variables by means of symbolic computations. As a consequence of the incompleteness of the verification step, the test cases generated by our algorithm may also detect violations of the property by the *specification* when executed on the *implementation*. Hence, test execution may detect one or several of the following inconsistencies:

- violation of the property by the specification,
- violation of the property by the implementation,
- violation of conformance between implementation and specification.

These results are returned to the user in the form of test verdicts, and may be employed to fix errors in the implementation, specification, or the properties.

The rest of the paper is organised as follows. Section 2 presents the model of Input-Output Symbolic Transition Systems (IOSTS) and, in Section 3 we set the framework for verification and testing using IOSTS as the underlying model.

Section 4 defines our symbolic test generation algorithm. The algorithm is proved correct, in the sense that the verdicts returned by test execution correctly characterise the relations between implementation, specification, and property. Moreover, the (infinite) set of all test cases generated in this manner may, in principle, discover all implementations that do not conform to a given specification according to the standard **ioco** relation [19]. As a by-product of the correctness proofs, we show that **ioco**-conformance with respect to a given specification is a safety property. We also provide a symbolic construction of the canonical tester [4] for **ioco**-conformance with respect to a given specification.

Section 5 outlines a technique for optimising test cases towards detecting the violation of the property. We show that this optimisation preserves the correctness of the test verdicts. The overall approach is illustrated on a simple example. The full version of this paper [17] contains a larger example (the Bounded Retransmission Protocol [11]) and provides proofs of the results.

2 The IOSTS Model

The model of Input-Output Symbolic Transition Systems (IOSTS) is inspired from I/O automata [15]. Unlike I/O automata, IOSTS do not require *input-completeness* (i.e., all input actions do not need to be enabled all the time).

Definition 1 (IOSTS). *An IOSTS is a tuple* $\langle D, \Theta, Q, q^0, \Sigma, \mathcal{T} \rangle$ *where*

- D *is a finite set of typed* Data, *partitioned into a set* V *of* variables *and a set* P *of* parameters. *For* $d \in D$, $type(d)$ *denotes the type of* d.
- Θ *is the* initial condition, *a Boolean expression on* V,
- Q *is a nonempty, finite set of* locations *and* $q^0 \in Q$ *is the* initial location.
- Σ *is a nonempty, finite* alphabet, *which is the disjoint union of a set* $\Sigma^?$ *of* input actions *and a set* $\Sigma^!$ *of* output actions[1]. *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 parameters.*
- $\mathcal{T}$ *is a set of* transitions. *Each transition is a tuple* $\langle q, a, G, A, q' \rangle$ *made of:*
  - *a location* $q \in Q$, *called the* origin *of the transition.*
  - *an action* $a \in \Sigma$ *called the* action *of the transition.*
  - *a Boolean expression* G *on* $V \cup sig(a)$, *called the* guard.
  - *an* assignment A, *which is a set of expressions 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* $V \cup sig(a)$.
  - *a location* $q' \in Q$ *called the* destination *of the transition.*

A simple example of IOSTS is depicted in Figure 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 variable x is decreased by 1, and when it reaches 0, the $STOP$ output is emitted.

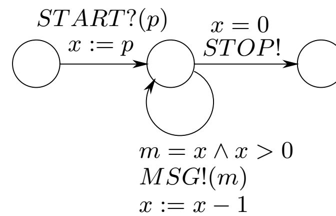

Figure1. Sample IOSTS $\mathcal{S}$

[1] For simplicity, only input and output actions are considered here. A more detailed model, which also contains internal actions, is defined in the full paper.

Semantics. The semantics of IOSTS is described in terms of input-output labelled transitions systems (IOLTS).

Definition 2. *An IOLTS is a tuple* $\langle S, S^0, \Lambda, \rightarrow \rangle$ *where* S *is a set of* states, *which may be infinite,* $S^0 \subseteq S$ *is the set of* initial states, $\Lambda = \Lambda^? \cup \Lambda^!$ *is a set of (input or output)* actions, *and* $\rightarrow \subseteq S \times \Lambda \times S$ *is the* transition relation.

Intuitively, the IOLTS semantics of an IOSTS $\langle D = V \cup P, \Theta, Q, q^0, \Sigma, \mathcal{T} \rangle$ enumerates of the possible tuples of values (hereafter called *valuations*) of parameters P and variables V. Let $\mathcal{V}$ denote the set of valuations of the variables V, and 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 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. *The semantics of an IOSTS* $\mathcal{S} = \langle D, \Theta, Q, q^0, \Sigma, \mathcal{T} \rangle$ *is an IOLTS* $[\![\mathcal{S}]\!] = \langle S, S^0, \Lambda, \rightarrow \rangle$, *defined as follows:*

- *the set of states is* $S = Q \times \mathcal{V}$,
- *the set of initial states is* $S^0 = q^0 \times \mathcal{V}^0$, *with* $\mathcal{V}^0 = \{\nu \in \mathcal{V} |\ \Theta(\nu) = true\}$
- *the set of actions* $\Lambda = \{\langle a, \pi \rangle | a \in \Sigma, \pi \in \Pi_{sig(a)}\}$, *also called the set of* valued actions, *is partitioned into the sets* $\Lambda^?$ *of* valued inputs *and* $\Lambda^!$ *of* valued outputs, *such that for* $\# \in \{?, !\}$, $\Lambda^\# = \{\langle a, \pi \rangle | a \in \Sigma^\#, \pi \in \Pi_{sig(a)}\}$.
- $\rightarrow$ *is the smallest relation in* $S \times \Lambda \times S$ *defined by the following rule:*

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

The rule says that the valued action $\langle a, \pi \rangle$ takes the system from a state $\langle q, \nu \rangle$ to a state $\langle q', \nu' \rangle$ if there exists a transition $t = \langle q, a, G, A, q' \rangle$ whose guard G evaluates to $true$ when the variables evaluate according to ν and the parameters carried by the action a evaluate according to π. Then, the assignment A of the transition maps the pair (ν, π) to ν'.

Definition 4 (run). *A run fragment is a sequence of alternating states and valued actions* $s_1 \stackrel{\alpha_1}{\rightarrow} s_2 \stackrel{\alpha_2}{\rightarrow} \cdots s_{n-1} \stackrel{\alpha_{n-1}}{\rightarrow} s_n$. *A* run *is a run fragment that starts in an initial state.*

A state is *reachable* if it is the last state of a run. For a sequence $\sigma = \alpha_1 \alpha_2 \cdots \alpha_n$ of valued actions, we sometimes write $s \stackrel{\sigma}{\rightarrow} s'$ for $\exists s_1, \ldots s_{n+1} \in S.\ s = s_1 \stackrel{\alpha_1}{\rightarrow} s_2 \stackrel{\alpha_2}{\rightarrow} \cdots s_n \stackrel{\alpha_n}{\rightarrow} s_{n+1} = s'$. For a set of states $S' \subseteq S$ of the IOSTS we write $s \stackrel{\sigma}{\rightarrow} S'$ if there exists a state $s' \in S'$ such that $s \stackrel{\sigma}{\rightarrow} s'$.

Definition 5 (trace). *The* trace *of a run* ρ *is the projection of* ρ *on* $\Lambda^! \cup \Lambda^?$. *The set of traces of an IOSTS* $\mathcal{S}$ *is denoted by* $Traces(\mathcal{S})$.

Let $F \subseteq Q$ be a set of locations of an IOSTS $\mathcal{S}$. A run ρ is *recognised by* F if it ends in a state in $F \times \mathcal{V}$. A trace is *recognised by* F if it is the projection on $\Lambda^! \cup \Lambda^?$ of a recognised run. The set of recognised traces is denoted by $RTraces(\mathcal{S}, F)$.

An IOSTS is *deterministic* if in each location, the guards of the transitions labelled by the same action are mutually exclusive. All the IOSTS considered in this paper are deterministic. In the full version [17], more general IOSTS are also considered (nondeterministic IOSTS with internal actions). A symbolic *determinisation* operation, which consists in transforming a nondeterministic IOSTS into a deterministic one having the same set of traces, is also presented. The operation is proved correct and terminates for a subclass of IOSTS [20].

3 Verification and conformance testing with IOSTS

This section sets the framework for verification and conformance testing with IOSTS. First, we present a few operations on IOSTS, and then the satisfaction relation and the conformance relation between IOSTS are formally defined.

3.1 Parallel Product

The *parallel product* of two IOSTS is an IOSTS whose set of traces (resp. recognised traces) are the intersection of the set of traces (resp. recognised traces) of the operands. This operation imposes that the IOSTS have no shared variables, but are defined on the same alphabets of actions and same parameters.

Definition 6 (Compatible IOSTS). *For $j = 1, 2$, the two IOSTS $\mathcal{S}_j = \langle D_j = V_j \cup P_j, \Theta_j, Q_j, q^0_j, \Sigma_j, \mathcal{T}_j \rangle$ with data D_j and alphabet $\Sigma_j = \Sigma^?_j \cup \Sigma^!_j$ are com*patible *if $V_1 \cap V_2 = \emptyset$, $P_1 = P_2$, $\Sigma^!_1 = \Sigma^!_2$, $\Sigma^?_1 = \Sigma^?_2$.*

Definition 7 (Parallel Product). *The parallel product $\mathcal{S} = \mathcal{S}_1 || \mathcal{S}_2$ of two compatible IOSTS $\mathcal{S}_1, \mathcal{S}_2$ is the IOSTS $\langle D, P, \Theta, Q, q^0, \Sigma, \mathcal{T} \rangle$ that consists of the following elements: $V = V_1 \cup V_2$, $P = P_1 = P_2$, $\Theta = \Theta_1 \wedge \Theta_2$, $Q = Q_1 \times Q_2$, $q^0 = \langle q^0_1, q^0_2 \rangle$, $\Sigma^? = \Sigma^?_1 = \Sigma^?_2$, $\Sigma^! = \Sigma^!_1 = \Sigma^!_2$ The set $\mathcal{T}$ of transitions of the composed system is the smallest set defined by the rule:*

$$\frac{\langle q_1, a, G_1, A_1, q'_1 \rangle \in \mathcal{T}_1 \quad \langle q_2, a, G_2, A_2, q'_2 \rangle \in \mathcal{T}_2}{\langle \langle q_1, q_2 \rangle, a, G_1 \wedge G_2, A_1 \cup A_2, \langle q'_1, q'_2 \rangle \rangle \in \mathcal{T}}$$

Lemma 1 (traces of the parallel product).
$Traces(\mathcal{S}_1 || \mathcal{S}_2) = Traces(\mathcal{S}_1) \cap Traces(\mathcal{S}_2)$.
$RTraces(\mathcal{S}_1 || \mathcal{S}_2, F_1 \times F_2) = RTraces(\mathcal{S}_1, F_1) \cap RTraces(\mathcal{S}_2, F2)$.

3.2 Quiescence and suspension IOSTS

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 does

not emit any output for the environment to observe). This is called *quiescence* in conformance testing [19]. On a black-box implementation, quiescence is observed using timers: a timer is reset whenever the environment sends a stimulus to the implementation; when the timer expires, the environment observes quiescence.

In order to distinguish a quiescence that is also present in a specification from one that is not, quiescence can be made explicit on a specification by a symbolic operation called *suspension*. This operation transforms an IOSTS $\mathcal{S}$ into an IOSTS $\mathcal{S}^\delta$, also called the *suspension IOSTS* of $\mathcal{S}$. Each location q of $\mathcal{S}^\delta$ contains a new self-looping transition, labelled with a new output action δ, which may be fired if and only if no other output action may be fired in q. Formally,

Definition 8 (Suspension). *Given* $\mathcal{S} = \langle D = V \cup P, \Theta, Q, q^0, \Sigma = \Sigma^! \cup \Sigma^?, \mathcal{T}\rangle$ *an IOSTS, the* suspension *IOSTS* $\mathcal{S}^\delta$ *is the tuple* $\langle D = V \cup P, \Theta, Q, q^0, (\Sigma^! \cup \{\delta\}) \cup \Sigma^?, \mathcal{T} \cup \bigcup_{q \in Q} \langle q, \delta, G_{\delta,q}, (v := v)_{v\in V}, q\rangle\rangle$ *where*

$$G_{\delta,q} : \bigwedge_{a\in\Sigma^!} \neg G_{a,q} \text{ where } G_{a,q} : \bigvee_{t=\langle q,a,G,A,q'\rangle\in\mathcal{T}} \exists sig(a).G. \qquad (1)$$

For the IOSTS $\mathcal{S}$ depicted in Figure 1, the IOSTS $\mathcal{S}^\delta$ is depicted in Figure 2. The guard $x < 0$ of the transition labeled δ is obtained by simplifying the expression $\neg(x = 0 \vee \exists m, m = x \wedge x > 0)$, which corresponds to Formula (1) above.

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 the special output $\delta!$ after *START*.

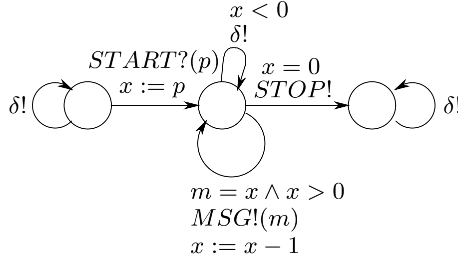

Figure2. Suspension IOSTS $\mathcal{S}^\delta$

3.3 Verification of Safety Properties

The problem considered here is: given a reactive system modelled by an IOSTS $\mathcal{S}$, and a safety property ψ defined on its traces, does $\mathcal{S}$ satisfy ψ? We model safety properties using *observers*, which are deterministic IOSTS equipped with a set of "bad" locations; the property is violated when a "bad" location is reached.

Definition 9 (Observer). *An observer is a deterministic IOSTS ω together with a set of dedicated locations $Violate_\omega \subseteq Q_\omega$, which are deadlocks (no outgoing transitions). An observer $(\omega, Violate_\omega)$ is compatible with an IOSTS M if ω is compatible with M. The set of observers compatible with M is denoted $\Omega(M)$.*

An observer $\omega \in \Omega(M)$ defines a safety property on $(\Lambda^!_M \cup \Lambda^?_M)^*$, namely, the property that is satisfied by all sequences in $(\Lambda^!_M \cup \Lambda^?_M)^* \setminus RTraces(\omega, Violate_\omega)$ (and those sequences only). In particular, if M is the suspension IOSTS $\mathcal{S}^\delta$ of a given IOSTS $\mathcal{S}$, then the property is satisfied by a subset of $(\Lambda^!_S \cup \{\delta\} \cup \Lambda^?_S)^*$.

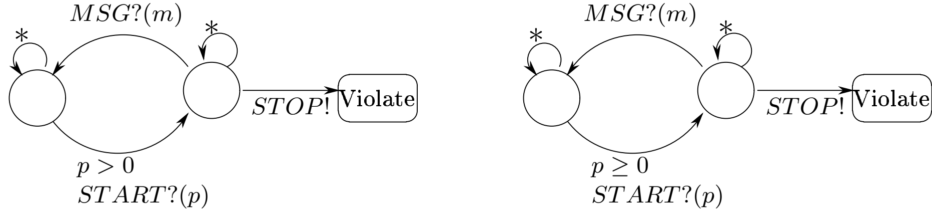

Figure3. Sample observers : ω_1 (left), ω_2 (right).

For example the observer ω_1 depicted in Figure 3 describes the safety property which says that between *START* input carrying a parameter $p > 0$, and a *STOP* output, the system must exhibit at least one *MSG* output. The set of "bad" locations is $\{Violate\}$. The self-loops "*" denote all actions (including the quiescence δ) that do not label other outgoing transitions. The observer ω_2 depicted on the right-hand side of Figure 3 describes almost the same property (except for the fact that *START* input carries a parameter $p \geq 0$). An IOSTS satisfies an observer if no trace of the IOSTS is recognised by the observer:

Definition 10 (IOSTS Satisfies Observer). *For an IOSTS $\mathcal{S}$ and an observer $(\omega, Violate_\omega) \in \Omega(\mathcal{S})$, we say that $\mathcal{S}$ satisfies $(\omega, Violate_\omega)$, denoted by $\mathcal{S} \models (\omega, Violate_\omega)$, if $Traces(\mathcal{S}) \cap RTraces(\omega, Violate_\omega) = \emptyset$.*

Let Q denote the set of locations of $\mathcal{S}$. Then, $Traces(\mathcal{S}) = RTraces(\mathcal{S}, Q)$ and $RTraces(\mathcal{S}\|\omega, Q \times Violate_\omega) = RTraces(\mathcal{S}, Q) \cap RTraces(\omega, Violate_\omega)$ (cf. Lemma 1). Hence, checking $\mathcal{S} \models (\omega, Violate_\omega)$ amounts to checking the emptyness of the set $RTraces(\mathcal{S}\|\omega, Q \times Violate_\omega)$. This can be done checking that the intersection between the set of *reachable states* of $\mathcal{S}\|\omega$, and the set of states whose locations lie in $Q \times Violate_\omega$, is empty. Alternatively, the intersection between the set of states from which $Q \times Violate_\omega$ is reachable (also called the *coreachable set* of $Q \times Violate_\omega$), and the set of initial states, can be checked for emptyness.

However, reachable and coreachable sets are not computable in general because of undecidability problems. Approximate analysis techniques such as abstract interpretation [7], can be used to compute over-approximations of them.

Our tool STG (Symbolic Test Generation) [6] is interfaced with a tool called NBac [13] for this purpose. First, STG automatically computes the product $\omega\|\mathcal{S}$, and then, NBac automatically performs an approximate reachability analysis (from the initial states) and approximate coreachability analysis (to the violating locations) of the product. These tools can be employed to prove, e.g., that the IOSTS $\mathcal{S}^\delta$ depicted in Figure 2 does satisfy the observer ω_1 depicted in Figure 3. (The violating locations are found unreachable, hence, the property holds).

On the other hand, it is impossible *in general* to prove automatically that an IOSTS does *not* satisfy an observer. Such a situation occurs with the IOSTS $\mathcal{S}^\delta$ in Figure 2 and the observer ω_2 depicted in the right-hand side of Figure 3: $\mathcal{S}^\delta$ does not satisfy ω_2, because a $START$ input carrying the parameter $p = 0$ allows for a $STOP$ output to be emitted (without any MSG inputs in between), which violates the property of interest (the *Violate* location is reached).

Combining observers. The parallel product of two observers $(\omega, \mathit{Violate}_\omega)$ and $(\varphi, \mathit{Violate}_\varphi)$ can be also interpreted in terms of safety properties. We use these properties in Section 4. A natural choice is to equip the product $\omega || \varphi$ with the set of locations $\mathit{Violate}_\omega \times \mathit{Violate}_\varphi$; by Lemma 1, $\mathit{RTraces}(\omega || \varphi, \mathit{Violate}_\omega \times \mathit{Violate}_\varphi) = \mathit{RTraces}(\omega, \mathit{Violate}_\omega) \cap \mathit{RTraces}(\varphi, \mathit{Violate}_\varphi)$; hence, we obtain a safety property which is violated whenever both safety properties described by $(\omega, \mathit{Violate}_\omega)$ and $(\varphi, \mathit{Violate}_\varphi)$ are violated. Alternative choices for the violating locations are, e.g., $\mathit{Violate}_\omega \times (Q_\varphi \setminus \mathit{Violate}_\varphi)$, which indicates the violation of the former property, but not that of the latter; and, $(Q_\omega \setminus \mathit{Violate}_\omega) \times \mathit{Violate}_\varphi$, which indicates the violation of the latter, but not of the former property.

3.4 Conformance Testing

A *conformance relation* formalises 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 modelled by a formal object. We assume here that it is modelled by an IOLTS (cf. 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.

These assumptions are called *test hypothesis* in conformance testing. The central notion in conformance testing is that of *conformance relation*; the standard **ioco** relation defined by Tretmans [19] can be rephrased as

Definition 11 (ioco). *An inplementation $\mathcal{I}$* **ioco***-conforms to a specification $\mathcal{S}$, denoted by $\mathcal{I}$* **ioco** *$\mathcal{S}$, if $\mathit{Traces}(\mathcal{S}^\delta) \cdot (\Lambda^! \cup \{\delta\}) \cap \mathit{Traces}(\mathcal{I}^\delta) \subseteq \mathit{Traces}(\mathcal{S}^\delta)$.*

Intuitively, 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 $\mathcal{S}^\delta$. Hence, in this framework, the specification is *partial* with respect to inputs, i.e., after an input that is not described by the specification, the implementation may have any behaviour, 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. An implementation that exhibits the trace $START?(1) \cdot STOP!$ does not conform to the specification $\mathcal{S}$ depicted in Figure 1 - this trace is not present

in the IOSTS $\mathcal{S}^\delta$ (Figure 2). For the same reason, the trace $START?(1) \cdot \delta!$ reveals a non-conformance to $\mathcal{S}$. On the other hand, a trace such as $START?(1) \cdot START?(1) \cdot STOP!$ does not pose problems for conformance, as $\mathcal{S}^\delta$ does not constrain the traces of the system after the second *START?* in any way.

4 Test Generation for Safety and Conformance

This section shows how to generate a test case from a specification using a safety property as a guide. The test case attempts to detect violations of the property by an implementation of the system and violations of the conformance between the implementation and the specification. Moreover, if the verification step (Section 3.3) could not establish the fact that the specification satisfies the property, the generated test cases may also detect violations of the property by the specification when executed on the implementation.

We show 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 an IOSTS M. We then show that the output-completion of the IOSTS of $\mathcal{S}^\delta$ is a *canonical tester* [4] for $\mathcal{S}$ and the **ioco** relation defined in Section 3.4 (a canonical tester for a specification with respect to a given relation allows, in principle, to detect every implementation that disagrees with the specification according to the relation). This derives from the fact, stated in Lemma 2 below, that **ioco**-conformance to a specification $\mathcal{S}$ is equivalent to satisfying (a safety property described by) an observer obtained from $\Sigma^!(\mathcal{S}^\delta)$. By composing this observer with another observer $(\omega, \mathit{Violate}_\omega)$ we obtain test cases for checking the conformance to $\mathcal{S}$ and the satisfaction of $(\omega, \mathit{Violate}_\omega)$.

Definition 12 (output-completion). *Given* $M = \langle D, \Theta, Q, q^0, \Sigma, \mathcal{T}\rangle$ *a deterministic IOSTS, the* output completion *of* M *is the IOSTS* $\Sigma^!(M) = \langle D, \Theta, Q \cup \{\mathit{Fail}_M\}, q^0, \Sigma, \mathcal{T} \cup \bigcup_{q \in Q, a \in \Sigma^!} \langle q, a, \bigwedge_{t=\langle q,a,G_t,A_t,q'_t\rangle \in \mathcal{T}} \neg G_t, (x := x)_{x \in V}, \mathit{Fail}_M\rangle\rangle$.

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

Lemma 2. $\mathcal{I}$ **ioco** $\mathcal{S}$ *iff* $\mathcal{I}^\delta \models (\Sigma^!(\mathcal{S}^\delta), \{Fail_{\mathcal{S}^\delta}\}))$.

The lemma also says that the IOSTS $\Sigma^!(\mathcal{S}^\delta)$ is a canonical tester for **ioco**-conformance to $\mathcal{S}$. Indeed, $\mathcal{I}^\delta \models (\Sigma^!(\mathcal{S}^\delta), \{Fail_{\mathcal{S}^\delta)}\}$ can be interpreted as the fact that execution of $\Sigma^!(\mathcal{S}^\delta)$ on the implementation $\mathcal{I}$

never leads to a "Fail" verdict; the fact that this is equivalent to $\mathcal{I}$ **ioco** $\mathcal{S}$ (as stated by Lemma 2) amounts to having a canonical tester [4].

A canonical tester is, in principle, enough for detecting all implementations that do not conform to a given specification. However, our goal in this paper is to detect, in addition to such non-conformances, other potential violations of other (additional) safety properties coming from, e.g., the system's requirements.

The observers (cf. Definition 9) employed for expressing such properties also serve as a test selection mechanism; by Lemma 1, the product between an observer and the canonical tester can be used to define a subset of traces of interest among the many possible traces of the canonical tester.

We first note that for an IOSTS M and an observer $(\omega, \mathit{Violate}_\omega) \in \Omega(M)$, the IOSTS $\omega||\Sigma^!(M)$ can be interpreted as an observer of M by choosing its set of violating locations. Let for now this set be $\mathit{Violate}_\omega \times \{\mathit{Fail}_M\}$, denoted by $\mathit{ViolateFail}_{\omega||\Sigma^!(M)}$.The subscript is omitted whenever it is clear from the context.

Definition 13. *For* $(\omega, \mathit{Violate}_\omega) \in \Omega(S^\delta)$, $\mathit{test}(\mathcal{S}, \omega) \triangleq \omega||\Sigma^!(\mathcal{S}^\delta)$.

In the rest of the section we show that every $\mathit{test}(\mathcal{S}, \omega)$ can be seen as a test case that refines the canonical tester, as violations of $(\omega, \mathit{Violate}_\omega)$ are also checked.

Proposition 1. $\mathcal{I}$ **ioco** $\mathcal{S}$ *iff*
$\forall(\omega, \mathit{Violate}_\omega) \in \Omega(\mathcal{S}^\delta).\ \mathcal{I}^\delta \models (\mathit{test}(\mathcal{S}, \omega), \mathit{ViolateFail}_{\mathit{test}(\mathcal{S},\omega)})$.

Interpretation. The IOSTS $\mathit{test}(\mathcal{S}, \omega)$ can be seen as a test case to be executed in parallel with an implementation $\mathcal{I}$. Proposition 1 says that if this execution enters a location in $\mathit{ViolateFail}_{\mathit{test}(\mathcal{S},\omega)}$ $(= \mathit{Violate}_\omega \times \{\mathit{Fail}_{\mathcal{S}^\delta}\})$, then the implementation violates both the property defined by $(\omega, \mathit{Violate}_\omega)$ and the conformance to specification $\mathcal{S}$. In this situation, the **ViolateFail** verdict is given:

ViolateFail: the implementation violates the property *and* the conformance

The proposition also says that the (infnite) set $\{\mathit{test}(\mathcal{S}, \omega) | (\omega, \mathit{Violate}_\omega) \in \Omega(\mathcal{S}^\delta)\}$ of test cases is "exhaustive" for checking **ioco**-conformance to a given specification $\mathcal{S}$, meaning that all non-conformances may, in principle, be detected.

We now consider another interpretation of the IOSTS $\omega||\Sigma^!(M)$, which leads to another test verdict. Choosing the violating locations to be $(Q_\omega \setminus \mathit{Violate}_\omega) \times \{\mathit{Fail}_M\}$ results in a different observer. We denote by $\mathit{Fail}_{\omega||\Sigma^!(M)}$ the set $(Q_\omega \setminus \mathit{Violate}_\omega) \times \{\mathit{Fail}_M\}$. The subscript is omitted whenever the context is clear.

Proposition 2. *For an IOSTS* $\mathcal{S}$ *and* $(\omega, \mathit{Violate}_\omega) \in \Omega(S^\delta)$,
$\mathcal{I}^\delta \not\models (\mathit{test}(\mathcal{S}, \omega), \mathit{Fail}_{\mathit{test}(\mathcal{S},\omega)}) \Rightarrow \neg(\mathcal{I}$ **ioco** $\mathcal{S})$

Proposition 2 says that when $\mathit{test}(\mathcal{S}, \omega)$ enters a location in the set $\mathit{Fail}_{\mathit{test}(\mathcal{S},\omega)}$ $(= (Q_\omega \setminus \mathit{Violate}_\omega) \times \{\mathit{Fail}_{\mathcal{S}^\delta}\}$) when executed on an implementation $\mathcal{I}$, then $\mathcal{I}$ violates conformance to $\mathcal{S}$. The property ω is not violated (the $\mathit{Violate}_\omega$ set is not entered). In this case, the **Fail** verdict is given:

Fail: the implementation violates the conformance, but not the property

A third interpretation of the IOSTS $\omega || \Sigma^!(M)$ as an observer can be given, by choosing the set of violating locations to be $Violate_\omega \times Q_M$. We denote this set by $Violate_{\omega||\Sigma^!(M)}$, and omit the subscript whenever it is clear from the context.

Proposition 3. *For an IOSTS $\mathcal{S}$ and observer $(\omega, Violate_\omega) \in \Omega(S^\delta)$, $\mathcal{I}^\delta \not\models (test(\mathcal{S},\omega), Violate_{test(\mathcal{S},\omega)}) \Rightarrow \mathcal{I}^\delta \not\models (\omega, Violate_\omega) \wedge \mathcal{S}^\delta \not\models (\omega, Violate_\omega)$.*

Proposition 3 says that when $test(\mathcal{S},\omega)$ enters a location in $Violate_{test(\mathcal{S},\omega)}$ when executed on an implementation $\mathcal{I}$, then a violation of the property by both specification and implementation is detected. Hence, the **Violate** verdict is given:

Violate: the specification and the implementation violate the property

Discussion. Propositions 1, 2, and 3 show that the test generation algorithm, i.e., the construction of the IOSTS $test(\mathcal{S},\omega)$ and of its three verdicts, are *correct*, in the sense that verdicts correctly describe the relations between specification, implementation, and property. The verdict **ViolateFail** (resp. **Fail**) detects the violation of the property and of the conformance (resp. of the conformance only) by the implementation. This holds independently of whether the specification satisfies the property or not; indeed, the execution of the test case on the implementation may detect violations of the property by the specification using the **Violate** verdict. The ability to generate test cases from a property and a specification which may or may not satisfy the property is important, because verification is undecidable for the infinite-state systems considered in this paper.

A natural question that arises is why a violation of the property by the implementation is always detected simultaneously with either (1) a violation of the property by the specification or (2) a violation of the conformance between implementation and specification. The reason is that our test cases are extracted from the specification, i.e., they only contain traces of the specification. An implementation may only violate a property without (1) or (2) occurring when it executes a trace that diverges at some point from the specification by an *input*; indeed, as seen in Section 3.4, this does not compromise conformance and, of course, the specification cannot violate the property on a trace that it does not contain. Such traces are excluded from the generated test cases by construction.

Alternatively, these traces could be included in the test cases, but this implies to perform an *input-completion* of the specification (similar to Definition 12) first, and could lead to test cases that are typically too large for use in practice.

Building an actual test case. To build an actual test case from $test(\mathcal{S},\omega)$, all inputs are transformed into outputs and reciprocally (this operation is called *mirror* ; in the test execution process, the actions of the implementation and those of the test case must complement each other). For the IOSTS $\mathcal{S}$ depicted in Figure 1 and the observer ω_2 depicted in Figure 3, the corresponding test case (before simplification) is depicted in Figure 4. Finally, the result is automatically analysed and simplified using the NBac tool [13] for statically eliminating transitions that cannot lead to the violation of the property any more (cf. Section 5).

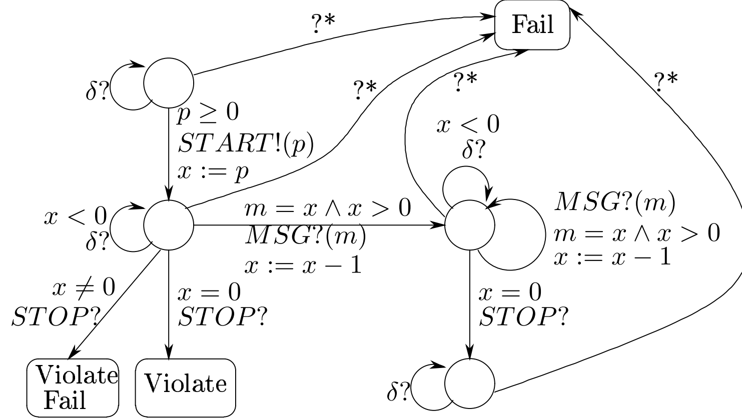

Figure4. Before selection: test case obtained from $\mathcal{S}$ (Figure 1) and ω_2 (Figure 3).

5 Test selection

The main goal of the testing process is to detect violations of the system's required properties by the system's implementation. In this section we outline a technique for statically detecting and eliminating locations and transitions of a test case (generated from a specification and a property as described in Section 4) from which this goal cannot be achieved any more; the resulting test case attempts to keep the implementation in states where it may still violate the property. We show that this optimisation preserves correctness of test verdicts.

The violation of a property - described as an observer (ω, $Violate_\omega$) - by an implementation is materialised by reaching the *ViolateFail* and *Violate* sets of locations in the IOSTS $test(\mathcal{S}, \omega)$(cf. Section 4). For a state s of an IOSTS and a location q of the IOSTS, we say that s is *coreachable* for the location q if there exists a valuation v of the variables such that $s \xrightarrow{\sigma} \langle q, v\rangle$. Then, the test selection process consists (ideally) in selecting, from a given test case, the subset of states that are coreachable for the locations in $Violate \cup ViolateFail$.

It should be quite clear that an exact computation of this set of states is impossible in general. However, there exist techniques that allow to compute an over-approximation of it. We here use one such technique based on abstract interpretation and implemented in the NBac tool [13]. Given a location q of an IOSTS, the tool computes, for each location l, a *symbolic coreachable state* for q:

Definition 14 (symbolic coreachable state). *For l, q two locations of an IOSTS $\mathcal{S}$, we say $\langle l, \varphi_{l \to q}\rangle$ is a* symbolic coreachable state *for q if $\varphi_{l \to q}$ is a formula on the variables of the IOSTS such that, if a state of the form $\langle l, v\rangle$ is coreachable for q, then $v \models \varphi_{l \to q}$ holds.*

I.e., $\langle l, \varphi_{l \to q}\rangle$ over-approximates the states with location l that are coreachable for q. The following algorithm uses this information for *pruning* a test case.

Definition 15 (pruning). *For an IOSTS $\mathcal{S}$ and an observer $(\omega, Violate_\omega)$ from the set $\Omega(\mathcal{S}^\delta)$, let $prune(\mathcal{S}, \omega)$ be the IOSTS computed as follows.*

- *first, the IOSTS $mirror(test(\mathcal{S}, \omega))$ is computed as in Section 4. 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 $\Sigma^? = \Sigma^!_{\mathcal{S}} \cup \{\delta\}$. Let also $Inconc \notin L$ be a new location.*
- *then, for each location $l \in L$, a symbolic coreachable state $\langle l, \varphi_{l \to q}\rangle$, for each location $q \in Violate \cup ViolateFail$ is computed. Let φ_l denote the formula $\bigvee_{q \in Violate \cup ViolateFail} \varphi_{l \to q}$*
- *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 $G \wedge \varphi_l$ is unsatisfiable, then t is eliminated from $\mathcal{T}$,*
    - *otherwise, the guard of t becomes $G \wedge \varphi_l$*
  - *if $a \in \Sigma^?$, then*
    - *the guard of t becomes $G \wedge \varphi_l$*
    - *a new transition is added to $\mathcal{T}$, with origin l, destination $Inconc$, action a, guard $G \wedge \neg\varphi_l$, and identity assignments.*

The *pruning* operation consists in detecting transitions whose firing leads to states where the *Violate* and *ViolateFail* sets of locations are unreachable. This is done by performing a coreachability analysis to these locations using the NBac tool [13]. If such a "useless" transition is labelled by an *output*, then it may be removed from the test case (a test case controls its outputs, hence, it may decide not to perform an output if violations of the property cannot be detected afterwards). On the other hand, *inputs* cannot be prevented from occurring, hence, the transitions labelled by *inputs*, by which the *Violate* and *ViolateFail* sets of locations cannot be reached any more, are reoriented to a new location, called *Inconc*. Reaching *Inconc* during test execution is interpreted as a verdict:

Inconc: violations of the property cannot be detected any more

Proposition 4. *The test case obtained by after pruning is correct, i.e., Propositions 1, 2 and 3 still hold when $test(\mathcal{S}, \omega)$ is replaced with $prune(\mathcal{S}, \omega)$.*

The test case obtained after pruning $test(\mathcal{S}, \omega_2)$ is depicted in Figure 5. It starts by sending a $START$ with a positive parameter p to the implementation, and then waits for inputs. If the implementation replies with $STOP$, the test execution terminates with a verdict, which depends on whether the parameter p was strictly positive or was equal to zero:

- If $p > 0$, the sequence $START(p) \cdot STOP$ exhibits a non-conformance between implementation (which accepts this sequence) and specification (which does not accept it). This sequence is also a witness for the violation of the property by the implementation: the verdict is *ViolateFail*;
- If $p = 0$, $START(p) \cdot STOP$ is a witness for violation of the property defined by ω_2 by both implementation and specification: the verdict is *Violate*.

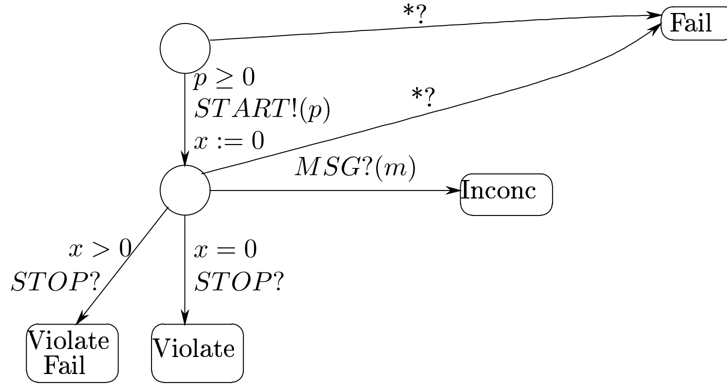

Figure5. After selection: test case obtained from $\mathcal{S}$ (Figure 1) and ω_2 (Figure 3).

Finally, if the implementation replies with *MSG* after *START*, the current test case cannot detect violations of the property any more, and the verdict is *Inconc*.

6 Conclusion and Related 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 labelled transition systems (IOLTS). The implementation is a black box, but it is assumed that its semantics can be described by an unknown IOLTS. This allows to formally link the implementation and the specification by a conformance relation. A satisfaction relation links them both to higher-level properties.

A validation methodology is proposed for checking these relations, i.e., for detecting inconsistencies between the different views of the system: First, the properties are automatically verified on the specification using abstract interpretation techniques. 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 established that the specification satisfies a property, the test execution may detect the violation of the property by the implementation and the violation of the conformance relation between implementation and specification. On the other hand, if the verification did not allow to prove a property, the test execution may additionally detect a violation of the property by the specification. Any inconsistencies obtained in this manner are reported to the user in the form of test verdicts. The approach is proved correct and is illustrated on a simple example. The full version of this paper [17] illustrates the approach on a larger example (the BRP protocol [11]).

Related Work. In [8] an approach for generating tests from a specification and from observers describing linear-time temporal logic requirements is described. The generated test cases do not check for conformance, they only check the fact that the implementation does not violate the requirements.

The approach described in [2] considers a 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 that violate P, which prove that S' is a mutant of S violating P. Finally, the obtained sequences are interpreted as test cases to be executed on the implementation.

The authors of [9] start from a specification S and a temporal-logic property P assumed to hold on S, and use the ability of model checkers to construct counter-examples for $\neg P$ on S. These counter-examples can be interpreted as *witnesses* (i.e., test cases) for P on S. The papers [3,12] extend this idea by formalising standard coverage criteria (all-definitions, all-uses, *etc*) using observers (resp. in temporal logic). Again, test cases are generated by model checking the observers (or the temporal-logic formulas) on the specification.

The approaches described in all these papers rely on model checking, hence, they only work for finite-state systems; moreover, they do not formally relate satisfaction of properties to conformance testing, and, except for [8], they do not formally define a conformance relation.

In [18] we present an approach for combining model checking and conformance testing for finite-state systems, which can be seen as a first step of the approach presented here, which deals with infinite-state systems. In the finite-state framework of [18] verification is decidable, which heavily influences the whole approach: for example the test generation algorithm (based on enumerative model checking) does not need to take into account the possibility that the property might be violated by the specification.

A different approach for combining model checking and black-box testing is black-box checking [16]. 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.

Our approach can also be related to the combination of verification, testing and monitoring proposed in [10]. In their approach, monitoring is passive (pure observation), whereas ours is reactive and adaptative, guided by the choice of inputs to deliver to the system as pre-computed in a test case.

Finally, in [14] we propose a symbolic algorithm for selecting test cases from a specification be means of so-called *test purposes.* The difference with the present paper lies mainly in methodology. Test purposes in [14] are essentially a pragmatic means for test selection - they have to be provided by the user. In contrast, test selection in the present paper consists in automatically attempting to violate a safety property that was automatically verified (succesfully or not) on the specification. Moreover, test purposes can be classified as *reachability* properties, which have an exactly opposite semantics to the safety properties considered here (reachability properties are negations of safety properties).

References

1. ISO/IEC 9646. Conformance Testing Methodology and Framework, 1992.
2. P. Ammann, W. Ding, and D. Xu. Using a model checker to test safety properties. In *International Conference on Engineering of Complex Computer Systems*. IEEE Computer Society, 2001.
3. J. Blom, A. Hessel, B. Jonnson, and P. Pettersson. Specifying and generating test cases using observer automata. In *Workshop on Formal Approaches to Software Testing (Fates'04)*, pages 137–152, 2004.
4. E. Brinskma. A theory for the derivation of tests. In *Protocol Specification, Testing and Verification (PSTV'88)*, pages 63–74, 1988.
5. E. Brinskma, A. Alderen, R. Langerak, J. van de Laagemat, and J. Tretmans. A formal approach to conformance testing. In *Protocol Secification, Testing and Verification (PSTV'90)*, pages 349–363, 1990.
6. D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: a symbolic test generation tool. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, number 2280 in LNCS, pages 470–475, 2002.
7. P. Cousot and R. Cousot. Abstract intrepretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
8. J.C. Fernandez, L. Mounier, and C. Pachon. Property-oriented test generation. In *Formal Aspects of Software Testing Workshop*, number 2931 in LNCS, 2003.
9. A. Gargantini and C.L. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC/SIGSOFT FSE*, pages 146–162, 1999.
10. K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Int. Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'02), Grenoble, France*, number 2280 in LNCS, pages 342–356, 2002.
11. L. Helmink, M. P. A. Sellink, and F. Vaandrager. Proof-checking a data link protocol. In *Types for Proofs and Programs (TYPES'94)*, number 806 in LNCS, pages 127–165, 1994.
12. H. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, number 2280 in LNCS, pages 327–341, 2002.
13. B. Jeannet. Dynamic partitioning in linear relation analysis. *Formal Methods in System Design*, 23(1):5–37, 2003.
14. B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *Int. Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'05), Grenoble, France (to appear)*, 2005.
15. N. Lynch and M. Tuttle. Introduction to IO automata. *CWI Quarterly*, 3(2), 1999.
16. D. Peled, M. Vardi, and M. Yannakakis. Black-box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225 – 246, 2001 2001.
17. V. Rusu, H. Marchand, and T. Jéron. Verification and symbolic test generation for safety properties. Technical Report 1640, IRISA, august 2004. Available at `http://www.irisa.fr/vertecs/Publis/Ps/PI-1640.pdf`.
18. V. Rusu, H. Marchand, V. Tschaen, T. Jéron, and B. Jeannet. From safety verifcation to safety testing. In *Intl. Conf. on Testing of Communicating Systems (TestCom04)*, number 2978 in LNCS, 2004.
19. J. Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99*, number 1664 in LNCS, pages 46–65, 1999.
20. E. Zinovieva. *Symbolic Test Generation for Reactive Systems*. PhD thesis, University of Rennes I, November 2004.