From Safety Verification to Safety Testing

Vlad Rusu, Hervé Marchand, Valéry Tschaen, Thierry Jéron, and Bertrand Jeannet

IRISA/INRIA Rennes, France First.Last@irisa.fr

Abstract A methodology that combines verification and conformance testing for validating safety requirements of reactive systems is presented. The requirements are first automatically verified on the system's specification. Then, test cases are automatically derived from the specification and the requirements, and executed on a black-box implementation of the system. The test cases attempt to push the implementation into violating a requirement. We show that an implementation conforms to its specification if and only if it passes all the test cases generated in this way.

Keywords: verification, conformance testing, safety properties.

1 Introduction

Formal verification and conformance testing are two well-established methods for validating software systems. In *verification* [14], a formal *specification* of the system is proved correct with respect to some higher-level *requirements*. In *conformance testing* [10] the external, observable traces of a black-box *implementation* of the system are compared to those of its formal specification, according to a *conformance relation*. For validating reactive systems (such as communication protocols) the two methods play complementary roles: the former ensures that the operational specification $\mathcal{S}$ meets its requirements $\mathcal{R}$, while the latter checks that the implementation $\mathcal{I}$ of the system conforms to its specification $\mathcal{S}$. Thus, through verification and testing, a connection between a system's final implementation and its initial requirements can be established:

1. first, satisfaction of the requirements by the specification is automatically verified, e.g., by model checking;
2. then, the user (or a test coverage tool, e.g., the TestComposer module of ObjectGeode [16]) produces *test purposes*, which are abstract scenarios to be tested on the implementation $\mathcal{I}$;
3. next, a test generation tool, e.g., Autolink [18], TorX [1] or TGV [11] uses the test purposes to generate test cases from the specification;
4. finally, the test cases are executed on the implementation $\mathcal{I}$, and verdicts are issued regarding its conformance with the specification.

This validation process corresponds to the current state-of-the-art use of formal methods in the telecom world [2]. The main problem with this process is that it does not guarantee that what is being tested on the implementation of the system (at Step 4) are the same requirements that have been verified to hold on the specification (at Step 1).

This is because the test generation step (Step 3) uses *test purposes*, which are a pragmatic means (actually, an essential one) to achieve test generation; but test purposes are typically written (at Step 2) independently of the requirements, that is, there is no *formal* connection between the test purposes and the requirements. If some crucial safety requirement is missed by all the test purposes, the final implementation may violate that requirement, and this violation remains undetected.

In this paper we propose a methodology to integrate verification and conformance testing into a seamless, sound validation process between *safety* requirements, specification, and implementation. The above validation process is then reformulated as follows: Step 1 is standard verification; Step 2 may be skipped (there is no need here to write test purposes by hand, but, of course, this is not forbidden either); Step 3 is a test generation algorithm that takes the specification and a safety requirement, and produces a test case for checking the requirement on the implementation; and Step 4 is standard conformance test execution.

Framework. The specification is given by IOLTS (Input-Output Labeled Transition Systems, i.e., finite, labeled transition systems with inputs, outputs, and internal actions). The requirements express safety properties on observable behaviors of the specification and are described by means of a particular class of IOLTS: *observers*, which enter a dedicated "*Violate*" location when the property is violated. Finally, the conformance relation between a black-box implementation and a specification is **ioco**, a standard relation used in conformance testing [19], which requires that after each visible trace of the specification, the observed outputs and blockings of the implementation are among those allowed by the specification.

Results. The meaning of requirement *relevant* for a specification is formally defined in the paper. We prove that an implementation conforms to a specification if and only if it satisfies all the relevant safety requirements that are also satisfied by the specification. This result is interesting because it establishes a formal connection between conformance and property satisfaction. However, it does not say how to actually check the safety requirements on the implementation. Moreover, the result is restricted to *relevant* requirements.

Hence, we propose a test generation algorithm, which takes a specification and a safety requirement (relevant or not), and produces a test case that, when executed on an implementation, attempts to push the implementation into violating the requirement. It is shown that an implementation conforms to a specification if and only if it passes all the test cases generated by the proposed algorithm.

The rest of the paper is organized as follows. In Section 2, the main concepts from verification and conformance testing are recalled. In Section 3, the notion of a safety property *relevant* to a specification is defined, and a result that connects conformance testing and satisfaction of relevant safety properties is proved. In Section 4, the previous result is extended to take into account arbitrary safety properties. This directly induces a sound and complete test generation algorithm for checking safety properties on the implementation. We conclude in Section 5.

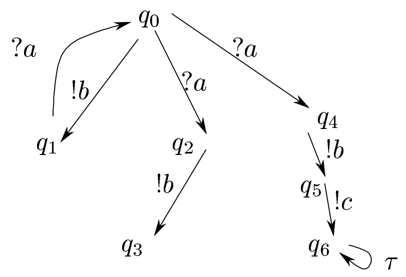

Figure1. Example of IOLTS $\mathcal{S}$

2 Verification and Conformance Testing

Definition 1 (IOLTS). *An IOLTS is a tuple $M = (Q, A, \rightarrow, q_o)$ where Q is a finite, non-empty set of states; $q_o \in Q$ is the initial state; $\rightarrow \subseteq Q \times A \times Q$ is the transition relation; and A is a finite alphabet of actions, partitioned into three sets $A = A_? \cup A_! \cup I$, where $A_?$ is the set of* input actions*, $A_!$ is the set of* output actions*, and I is the set of* internal *actions.*

The actions in $A_? \cup A_!$ are also called *visible actions*. In the examples, a question mark placed before the name of an action (as in "$?a$") denotes the fact that a is an input $a \in A_?$. An exclamation mark (as in "$!b$") denotes an output $b \in A_!$. Internal actions are generically denoted by τ.

Example 1. The IOLTS depicted in Figure 1 describes a simple nondeterministic system. In the initial state q_0, the system can either spontaneously emit $!b$ and block itself waiting in state q_1 for an $?a$; or directly wait for an $?a$ in q_0 and then, nondeterministically go to either q_2 or q_4. In q_2, the system may only emit $!b$ and deadlock in q_3, while in q_4, $!b$ then $!c$ are emitted, followed by a loop of internal actions (a livelock).

2.1 Notations and Basic Definitions

Let $M = (Q, A, \rightarrow, q_o)$ be an IOLTS. The notation $q \xrightarrow{a} q'$ stands for $(q, a, q') \in \rightarrow$ and $q \xrightarrow{a}$ for $\exists q' : q \xrightarrow{a} q'$. An IOLTS is sometimes identified with its initial state, i.e., we write $M \rightarrow$ for $q_o \rightarrow$. Let $\mu_i \in A$ denote some actions, $a_i \in A \setminus I$ some visible actions, $\tau_i \in I$ some internal actions, $\sigma \in (A \setminus I)^*$ a sequence of visible actions, $q, q' \in Q$ some states.

- We write $q \xrightarrow{\mu_1 \cdots \mu_n} q'$ for $\exists q_0, \ldots, q_n : q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_n} q_n = q'$.
- The notation $\Gamma(q)$ stands for $\{\mu \in A \mid q \xrightarrow{\mu}\}$, i.e., the set of fireable actions in q. Similarly $\Gamma^{-1}(q)$ denotes the set $\{\mu \in A | \exists q'.q' \xrightarrow{\mu} q\}$.
- Visible behaviors are described by the $\Rightarrow$ relation, defined by $q \overset{\varepsilon}{\Rightarrow} q' \triangleq q = q'$ or $q \xrightarrow{\tau_1 \cdot \tau_2 \cdots \tau_n{}^*} q'$ and $q \overset{a}{\Rightarrow} q' \triangleq \exists q_1, q_2 : q \overset{\varepsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \overset{\varepsilon}{\Rightarrow} q'$. We also write $q \overset{a_1 \cdots a_n}{\Rightarrow} q'$ for $\exists q_0, \ldots, q_n : q = q_0 \overset{a_1}{\Rightarrow} q_1 \cdots \overset{a_n}{\Rightarrow} q_n = q'$

- $Traces(q) \triangleq \{\sigma \in (A \setminus I)^* \mid q \overset{\sigma}{\Rightarrow}\}$ (resp. $Traces(M) \triangleq Traces(q_0)$) denotes the sequences of visible actions that are fireable from a state q (resp. from the initial state of the IOLTS M).
- For $q \in Q$ and a trace $\sigma \in Traces(q)$, we denote by q *after* $\sigma \triangleq \{q' \in Q \mid q \overset{\sigma}{\Rightarrow} q'\}$ (resp. P*after* $\sigma \triangleq \bigcup_{q \in P} q$ *after* σ) the set of states that are reachable from the state q (resp. from the set of states P) by sequences of actions whose projection onto visible actions is σ.
- For $q \in Q$, $Out(q) \triangleq \Gamma(q) \cap A_!$ is the set of fireable outputs in q. This notion is naturally extended to sets of states: for $P \subseteq Q$, $Out(P) \triangleq \bigcup_{q \in P} Out(q)$. Likewise, $In(q)$ (resp. $In(P)$) denote the set of fireable inputs in $q \in Q$ (resp. in $P \subseteq Q$).
- Finally, an IOLTS M is *input-complete* whenever $\forall q \in Q$, $In(q \textit{ after } \epsilon) = A_?$, that is, in each state, all inputs are accepted, possibly after a sequence of internal actions (here, ϵ denotes the empty trace).

Based on the previous notations we introduce some common definitions.

Definition 2 (Deterministic IOLTS). *An IOLTS M is* deterministic *if $I^M = \emptyset$, and for all $q, q', q'' \in Q^M$ and $a \in A^M$, $q \xrightarrow{a} q'$ and $q \xrightarrow{a} q''$ imply $q' = q''$.*

Given an arbitrary IOLTS M, one can construct a deterministic IOLTS $det(M)$ with the same visible behavior, i.e., $Traces(M) = Traces(det(M))$.

Definition 3 (Determinization). *The* deterministic IOLTS *of an IOLTS $M = (Q, A, \rightarrow, q_o)$ is $det(M) = (2^Q, A \setminus I, \rightarrow_d, q_0 \textit{ after } \varepsilon)$, whose transition relation $\rightarrow_d$ is the smallest relation defined by: $P \xrightarrow{a}_d P'$ if $P' = P$ after a.*

The *synchronous composition* of two IOLTS performs synchronization on their common visible actions and lets them evolve independently by internal actions:

Definition 4 (Synchronous composition). *Let $M_i = (Q_i, A^i, \rightarrow_i, {q^i}_o)$, $i = 1, 2$ be two IOLTS, with I_i the internal actions of M_i. The* synchronous composition *$M_1 \parallel M_2$ of M_1 and M_2 is an IOLTS $(Q, A, \rightarrow, q_o)$ such that:*

- $Q = Q^1 \times Q^2,\ q_o = (q_o^1, q_o^2),\ A = A^1 \cap A^2 \cup (I^1 \cup I^2)$
- *$\rightarrow$ is the smallest relation in $Q \times A \times Q$ satisfying*

$$(q_1, q_2) \xrightarrow{a} \begin{cases} (q_1', q_2') \textit{ if } a \in A \setminus (I^1 \cup I^2) \ \wedge\ q_1 \xrightarrow{a}_1 q_1' \ \wedge\ q_2 \xrightarrow{a}_2 q_2' \\ (q_1', q_2) \textit{ if } a \in I^1 \ \wedge\ q_1 \xrightarrow{a}_1 q_1' \\ (q_1, q_2') \textit{ if } a \in I^2 \ \wedge\ q_2 \xrightarrow{a}_2 q_2' \end{cases}$$

It is not hard to show that $Traces(M_1 \parallel M_2) = Traces(M_1) \cap Traces(M_2)$.

2.2 Verification of safety properties

The verification problem considered here is : given a reactive system M and a property ψ, does M satisfy ψ ($M \models \psi$)? We model properties using observers, which are a particular class of IOLTS.

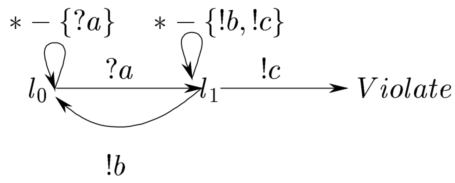

Figure2. Sample observer. For $A \subseteq A^\omega$, the notation $* - A$ is a shortcut for $A^\omega \setminus A$.

Definition 5 (Observer). *An observer for an IOLTS M is a deterministic IOLTS $\omega = (Q^\omega, A^\omega, \rightarrow_\omega, q_0^\omega)$ such that $A^\omega = A^M \setminus I^M$, and there exists a unique state $Violate_\omega \in Q^\omega \setminus \{q_0^\omega\}$ such that $\Gamma^{-1}(Violate_\omega) \subseteq A_!^M$ and $\Gamma(Violate_\omega) = \emptyset$. Its language is $\mathcal{L}(\omega) = \{\sigma \in (A^\omega)^* \mid q_0^\omega \xrightarrow{\sigma}_\omega Violate_\omega\}$.*

An observer expresses the negation of a safety property on the visible behavior of a system. The *Violate* state is entered when the system emits an undesired output. We note that Definition 5 matches the class of Büchi automata obtained from negations of LTL safety formulas [13], except for the self-loop on the *Violate* accepting state, and for the propositions labelling transitions (rather than states).

Example 2. Consider the property: between each $?a$ and $!c$, there must be at least one $!b$. The negation of this property is expressed by the observer depicted in Figure 2. Here, an action $?a$ followed by a $!c$ goes to the Violate location, meaning that the property was violated as there has been no $!b$ between $?a$ and $!c$. However, if $!b$ occurs after $?a$, the property cannot be violated unless another $?a$ occurs, hence, the observer goes back to its initial location to wait for another $?a$.

Let $\Omega(M)$ denote the (infinite) set of observers for an IOLTS M. The following definition formalizes the *satisfaction* of $\omega \in \Omega(M)$ by M:

Definition 6 (Satisfaction relation). *An IOLTS M satisfies an observer $\omega \in \Omega(M)$, denoted $\mathcal{S} \models \omega$, if and only if $Traces(M) \cap \mathcal{L}(\omega) = \emptyset$.*

$M \models \omega$ holds whenever no state $(q, Violate_\omega)$ with $q \in Q^M$ is reachable in $M \parallel \omega$.

Example 3. The IOLTS $\mathcal{S}$ depicted in Figure 1 satisfies the observer ω depicted in Figure 2. Indeed, there is no way that in $\mathcal{S}$ an $?a$ can be directly followed by $!c$ without $!b$ occurring in between. As these are the only traces that may lead to Violate in ω, we conclude that $\mathcal{S} \models \omega$.

2.3 Conformance Testing

The goal of conformance testing is to establish whether a black-box implementation $\mathcal{I}$ conforms to its formal specification $\mathcal{S}$. In our framework, the specification is given by an IOLTS $\mathcal{S} = (Q^\mathcal{S}, A^\mathcal{S}, \rightarrow_\mathcal{S}, q_0^\mathcal{S})$. The 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 behavior of $\mathcal{I}$ can be modeled by a formal object in the same class as the specification and having a *compatible interface* with it, i.e.,

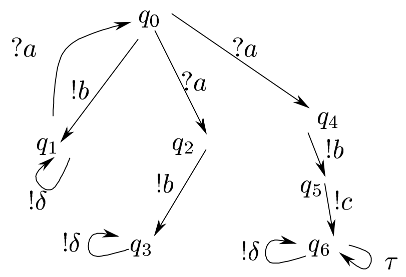

Figure3. The suspension IOLTS $\mathcal{S}^\delta$ for the IOLTS $\mathcal{S}$ depicted in Figure 1

having the same set of visible actions. Moreover, the implementation can never refuse an input from the environment (it is *input-complete*).

These assumptions are called *test hypothesis* in conformance testing [19]. Thus, the implementation is modeled by an input-complete IOLTS $\mathcal{I} = (Q^{\mathcal{I}}, A^{\mathcal{I}}, \rightarrow_{\mathcal{I}}, q_0^{\mathcal{I}})$ with $A^{\mathcal{I}} = A^{\mathcal{I}}_? \cup A^{\mathcal{I}}_! \cup I^{\mathcal{I}}$, $A^{\mathcal{S}}_? = A^{\mathcal{I}}_?$, and $A^{\mathcal{S}}_! = A^{\mathcal{I}}_!$.

Quiescence: The tester observes not only responses of the implementation, but also *absence of response* (i.e., in a given state, the implementation does not emit any output for the tester to observe). This is called *quiescence* in conformance testing. There are three possible reasons for quiescence:

- A *deadlock* state is a state where the system cannot evolve: $\Gamma(q) = \emptyset$.
- An *output quiescent* state is a state where the system is waiting only for an input from the environment, i.e. $\Gamma(q) \subseteq A_?$.
- A *livelock* state is a state from which the system diverges by an infinite sequence of internal actions. In the case of finite state systems that we consider, a livelock is a loop of internal actions, i.e., $\exists \tau_1, \ldots \tau_n, q \overset{\tau_1 \cdots \tau_n}{\rightarrow} q$.

In practice, quiescence is observed using timers : a timer is reset whenever the tester sends a stimulus to the implementation; when the timer expires, the tester observes quiescence. It is assumed that the timer is set to a value large enough such it only expires only when no response will ever occur.

At the model level, however, quiescence is materialized by adding a special output action δ that manifests itself in quiescent states [19].

Definition 7 (Suspension IOLTS). *The* suspension IOLTS *of an IOLTS* $\mathcal{S} = (Q, A, \rightarrow, q_o)$ *is an IOLTS* $\mathcal{S}^\delta = (Q^\delta, A^\delta, \rightarrow_\delta, q_0^\delta)$ *with* $Q^\delta = Q$, $q_o^\delta = q_o$, $A^\delta = A \cup \{\delta\}$ *and* $\delta \in A^\delta_!$ *(δ is an output of $\mathcal{S}^\delta$). The transition relation of $\mathcal{S}^\delta$ is* $\rightarrow_\delta = \rightarrow \cup \{q \xrightarrow{\delta} q \mid q \text{ is quiescent}\}$. *The traces of $\mathcal{S}^\delta$ are the* suspension traces *of* $\mathcal{S}$.

Example 4. For $\mathcal{S}$ depicted in Fig. 1, q_3 is a deadlock, q_1 is an output-lock, and q_6 is a livelock. $\mathcal{S}^\delta$ (cf. Fig. 3) is obtained by adding a δ-labeled self-loop to them.

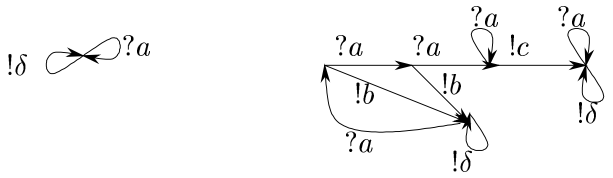

Figure4. Suspension IOLTS of Implementations $\mathcal{I}_1$ and $\mathcal{I}_2$ from Example 5.

Conformance relation A *conformance relation* formalizes the set of implementations that behave consistently with a specification. Here, we use the classical **ioco** relation defined by Tretmans [19]. Intuitively, an implementation $\mathcal{I}$ **ioco**-conforms to its specification $\mathcal{S}$, if, after each suspension trace of $\mathcal{S}$, the implementation only exhibits outputs and quiescences that are possible in $\mathcal{S}$. Formally:

Definition 8 (Conformance). *For IOLTS $\mathcal{S}, \mathcal{I}$ such that $\mathcal{I}$ is interface compatible with $\mathcal{S}$, $\mathcal{I}$ **ioco** $\mathcal{S} \triangleq \forall \sigma \in Traces(\mathcal{S}^\delta), Out(\mathcal{I}^\delta\, after\ \sigma) \subseteq Out(\mathcal{S}^\delta\, after\ \sigma)$.*

Remember that the black-box implementation $\mathcal{I}$ is assumed to be input-complete, but the specification $\mathcal{S}$ is not necessarily so. 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 behavior, without violating conformance to the specification. This corresponds to the intuition that a specification models a minimal set of services that must be provided by a system. A particular implementation of the system may include more services than what is specified, but these services should not influence conformance to the specification.

Example 5. First, consider the implementation $\mathcal{I}_1$ whose suspension IOLTS is depicted in Figure 4 (left), which only accepts inputs ?a and emits no outputs but quiescence. For $\mathcal{S}$ depicted in Figure 1, $\mathcal{I}_1$ **ioco** *$\mathcal{S}$ does not hold, because quiescence is not allowed initially in the suspension IOLTS $\mathcal{S}^\delta$ (Figure 3). On the other hand, the implementation $\mathcal{I}_2$ whose suspension IOLTS is depicted in Figure 4 (right) does conform to $\mathcal{S}$: the first divergence between $\mathcal{I}_2^\delta$ and $\mathcal{S}^\delta$ is on the second* input *?a in line, which does not violate conformance.*

The mechanism for testing the conformance of an implementation with its specification consists in generating *test cases* from the specification, and running them in parallel with the implementation to detect non-conformances between the two representations of the system. Test cases are essentially deterministic, input-complete IOLTS. In Section 4 we show how to generate test cases from the specification using observers for safety properties, in order to test both the satisfaction of the properties by the implementation and its conformance to the specification.

Example 6. To see why it is important to extract test cases from the specification, consider a tester that initially stimulates the implementation $\mathcal{I}_1$ from Example 5 using two *consecutive ?a actions. This behavior does not belong to the specification $\mathcal{S}$, which only allows one ?a initially (cf. Fig 1). Although $\mathcal{I}_1$* **ioco** *$\mathcal{S}$ does not hold, this particular tester cannot observe it, because it has forced the implementation to diverge from the specification by the second ?a. When this happens, the implementation is free to do anything without violating the conformance.*

3 Connecting Conformance and Property Satisfaction

In this section we establish a relation between conformance ($\mathcal{I}$ **ioco** $\mathcal{S}$) and the satisfaction of safety properties by the specification and the implementation ($\mathcal{S} \models \omega$, $\mathcal{I} \models \omega$). Properties are expressed using observers $\omega \in \Omega(\mathcal{S})$. Remember that a specification $\mathcal{S}$ is a *partial* specification in the sense defined in Section 2.3: the specification does not say what happens after an unspecified input. Hence, an observer is *relevant* for a specification $\mathcal{S}$ if it may only diverge from $\mathcal{S}$ by outputs:

Definition 9 (Relevant observer). *An observer $\omega \in \Omega(M)$ is* relevant *for M if $\mathcal{L}(\omega) \subseteq Traces(M) \cdot A_!^M$. The set of relevant observers for M is denoted $\varrho(M)$.*

Example 7. The observer ω depicted in Figure 2 is not relevant for the specification $\mathcal{S}$ depicted in Figure 1. This is because ω accepts arbitrarily many ?a's initially, while $\mathcal{S}$ accepts only one. Intuitively, ω is not relevant because it says something about behaviors that diverge from $\mathcal{S}$ through an unspecified input.

Relevant observers play an essential role in Theorem 1 below, which establishes a relation between conformance testing and property satisfaction.

To prove Theorem 1 the following technical lemma will be employed.

Lemma 1. *For A, B, C arbitrary sets, $\forall x \in A.(x \notin B \Rightarrow x \notin C)$ implies $(A \cap B = \emptyset \Rightarrow A \cap C = \emptyset)$.*

Theorem 1. *For all IOLTS $\mathcal{I}, \mathcal{S}$: $\mathcal{I}$* **ioco** *$\mathcal{S} \Leftrightarrow \forall \omega \in \varrho(\mathcal{S}^\delta).\ \mathcal{S}^\delta \models \omega \Rightarrow \mathcal{I}^\delta \models \omega$.*

Proof : ($\Rightarrow$): $\mathcal{I}$ **ioco** $\mathcal{S}$ is $\forall \sigma \in Traces(\mathcal{S}^\delta).Out(\mathcal{I}^\delta\, after\ \sigma) \subseteq Out(\mathcal{S}^\delta\, after\ \sigma)$ (Definition 8). By definition of $Out(\mathcal{S}^\delta\, after\ \sigma)$, $Out(\mathcal{I}^\delta\, after\ \sigma)$ (cf. Section 2.1):

$$\forall \sigma \in Traces(\mathcal{S}^\delta), \forall a \in A_!^{\mathcal{S}^\delta}.\ \sigma \cdot a \in Traces(\mathcal{I}^\delta) \Longrightarrow \sigma \cdot a \in Traces(\mathcal{S}^\delta)$$

which is clearly equivalent to

$$\forall \sigma \in Traces(\mathcal{S}^\delta), \forall a \in A_!^{\mathcal{S}^\delta}.\ \sigma \cdot a \notin Traces(\mathcal{S}^\delta) \Longrightarrow \sigma \cdot a \notin Traces(\mathcal{I}^\delta) \qquad (1)$$

Now, consider an arbitrary relevant observer $\omega \in \varrho(\mathcal{S}^\delta)$. By Definition 9, all the sequences $\sigma' \in \mathcal{L}(\omega)$ are of the form $\sigma' = \sigma \cdot a$, where $\sigma \in Traces(\mathcal{S}^\delta)$ and $a \in A_!^{\mathcal{S}^\delta}$. Hence, the implication (1) can be rewritten equivalently as

$$\forall \omega \in \varrho(\mathcal{S}^\delta).\forall \sigma' \in \mathcal{L}(\omega).\ \sigma' \notin Traces(\mathcal{S}^\delta) \Longrightarrow \sigma' \notin Traces(\mathcal{I}^\delta) \qquad (2)$$

Using Lemma 1 with $A = \mathcal{L}(\omega)$, $B = Traces(\mathcal{S}^\delta)$, $C = Traces(\mathcal{I}^\delta)$, we obtain

$$\forall \omega \in \varrho(\mathcal{S}^\delta).\mathcal{L}(\omega) \cap Traces(\mathcal{S}^\delta) = \emptyset \Longrightarrow \mathcal{L}(\omega) \cap Traces(\mathcal{I}^\delta) = \emptyset \qquad (3)$$

which, by Definition 6 is $\forall \omega \in \varrho(\mathcal{S}^\delta).\ \mathcal{S}^\delta \models \omega \Rightarrow \mathcal{I}^\delta \models \omega$: this direction is done. ($\Leftarrow$) Assume $\neg(\mathcal{I}$ **ioco** $\mathcal{S})$. We prove that there exists a relevant observer $\omega \in \varrho(\mathcal{S}^\delta)$ such that $\mathcal{S}^\delta \models \omega$ but $\mathcal{I}^\delta \not\models \omega$. This leads to a contradiction and completes the proof. To build ω, from $\neg(\mathcal{I}$ **ioco** $\mathcal{S})$ we obtain that there exists a sequence of the form $\sigma \cdot a$ with $a \in A_!^{\mathcal{S}^\delta}$ such that $\sigma \in Traces(\mathcal{S}^\delta)$, $a \in Out(\mathcal{I}^\delta \ \mathit{after} \ \sigma)$ but $a \notin Out(\mathcal{S}^\delta \ \mathit{after} \ \sigma)$. Let ω be an observer such that $\mathcal{L}(\omega) = \{\sigma \cdot a\}$.

Then, clearly, ω is relevant for $\mathcal{S}^\delta$ as $\mathcal{L}(\omega) \subseteq Traces(\mathcal{S}^\delta) \cdot A_!^{\mathcal{S}^\delta}$. Also, $\mathcal{S}^\delta \models \omega$ as $a \notin Out(\mathcal{S}^\delta \ \mathit{after} \ \sigma)$, therefore, $\sigma \cdot a \notin Traces(\mathcal{S}^\delta)$, i.e., $\mathcal{L}(\omega) \cap Traces(\mathcal{S}^\delta) = \emptyset$; and $\mathcal{I}^\delta \not\models \omega$ as $a \in Out(\mathcal{I}^\delta \ \mathit{after} \ \sigma)$ and therefore $\sigma \cdot a \in Traces(\mathcal{I}^\delta) \cap \mathcal{L}(\omega) \neq \emptyset$. Hence, the observer ω is relevant for $\mathcal{S}^\delta$, $\mathcal{S}^\delta \models \omega$, and $\mathcal{I}^\delta \not\models \omega$: the proof is done.□

Interpretation. Theorem 1 can be interpreted as follows: an implementation $\mathcal{I}$ **ioco**-conforms to its specification $\mathcal{S}$ if, whenever $\mathcal{S}^\delta$ satisfies a relevant safety property, $\mathcal{I}^\delta$ satisfies it as well. Hence, in order to establish conformance, it is enough to prove that all relevant safety properties satisfied by the specification are also satisfied by the implementation. This is a completeness result, which is impossible to achieve in practice because there may be infinitely many relevant properties that hold on a specification.

On the other hand, Theorem 1 also says that, to detect conformance violation, it is enough to exhibit one relevant property that is satisfied by the specification, but violated by the implementation. This a soundness result and is achievable in practice. However, it does not say how to actually check the violation of the property by the implementation (the observer is not a test case, for example, it is not necessarily input-complete) and, more importantly, it is limited to properties expressed by observers that are relevant to the specification.

These limitations are raised in Section 4. We conclude this section by an example showing that the *relevance* hypothesis is essential for Theorem 1 to hold.

Example 8. Consider the observer ω, which was shown in Example 7 to be irrelevant for $\mathcal{S}$ (Fig. 1). For the same reason, ω is irrelevant for $\mathcal{S}^\delta$ (Fig.3). Consider now implementation $\mathcal{I}_2$ whose suspension IOLTS is depicted in Figure 4. We have shown in Example 5 that $\mathcal{I}_2$ **ioco** *$\mathcal{S}$, and in Example 1 that $\mathcal{S}^\delta \models \omega$, but clearly, $\mathcal{I}_2^\delta \not\models \omega$ because $\mathcal{I}_2^\delta$ admits a $!c$ directly after an $?a$. That is, except for its irrelevance, the observer ω falsifies Theorem 1.*

4 Test Generation from Safety Requirements

This section shows how to generate test cases from a specification using a safety requirement as a guide. Intuitively, such a test case guides the implementation, and attempts to "push" it into violating the requirement.

It should be clear from the previous examples that writing a relevant requirement for a given specification (in the sense of Definition 9) is not always easy. For example, the requirement expressed by the observer ω from Examples 2 to 8 is a natural (and true) property of $\mathcal{S}$, but it is nevertheless irrelevant for $\mathcal{S}$.

However, when an observer as a whole is irrelevant for a given specification, a subset the observer of it may still be relevant. For example, the sequence $?a{\cdot}!c \in \mathcal{L}(\omega)$ is in $Traces(\mathcal{S}^\delta) \cdot A_!^{\mathcal{S}^\delta}$, i.e., it is relevant for $\mathcal{S}^\delta$ in the sense of Definition 9.

Hence, we need a test generation algorithm that takes a specification $\mathcal{S}$ and an *arbitrary* requirement $\omega \in \Omega(\mathcal{S})$, and automatically sorts out from ω what is relevant for $\mathcal{S}$ and what is not. This is done by the following operation.

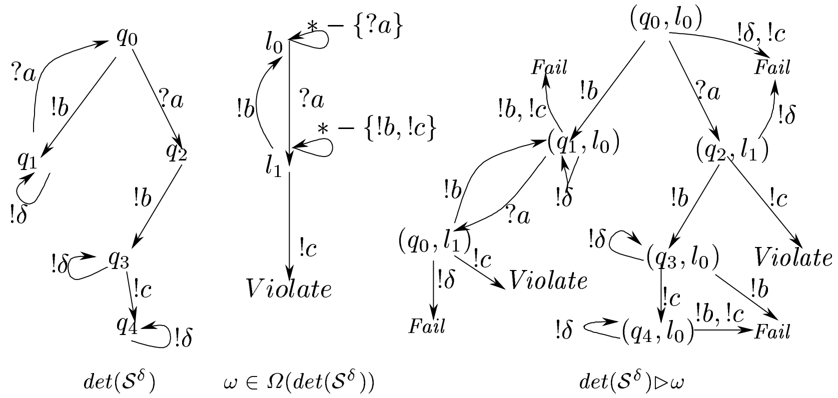

Figure5. The $\rhd$ operation between $det(\mathcal{S}^\delta)$ and ω

Definition 10 (Forcing). *Let* $M = (Q^M, A^M, \rightarrow_M, q_0^M)$ *be a deterministic IOLTS and* $\omega = (Q^\omega, A^\omega, \rightarrow_\omega, q_0^\omega)$ *an observer for* M. *The forcing of* M *by* ω, *denoted* $M \rhd \omega$, *is an IOLTS* $(Q^\rhd, A^\rhd, \rightarrow_\rhd, q_o^\rhd)$ *such that*

- *the set of states* $Q^\rhd$ *is* $(Q^M \times Q^\omega) \cup \{\mathit{Violate}, \mathit{Fail}\}$, *where* $\mathit{Violate}, \mathit{Fail} \notin Q^M \times Q^\omega$
- *the initial state* ${q^\rhd}_o$ *is* (q_o^M, q_o^ω)
- *the alphabet* $A^\rhd$ *is* A^M *(same partitioning between inputs and outputs)*
- *the transition relation* $\rightarrow_\rhd$ *of* $Q^\rhd$ *is the smallest relation defined as follows. For all states* $(p, q) \in Q^\rhd$ *and action* $a \in A^\rhd$:
  - *if* $p \xrightarrow{a}_M p'$, $q \xrightarrow{a}_\omega q'$ *then* $(p, q) \xrightarrow{a}_\rhd (p', q')$ $[\alpha]$
  - *if* $a \in A_!$, $p \not\xrightarrow{a}_M$, $q \xrightarrow{a}_\omega \mathit{Violate}_\omega$ *then* $(p, q) \xrightarrow{a}_\rhd \mathit{Violate}$ $[\beta]$
  - *if* $a \in A_!$, $p \not\xrightarrow{a}_M$, $\neg(q \xrightarrow{a}_\omega \mathit{Violate}_\omega)$, *then* $(p, q) \xrightarrow{a}_\rhd \mathit{Fail}$. $[\gamma]$

Example 9. For the IOLTS $\mathcal{S}^\delta$ *depicted in Figure 1 and* ω *depicted in Fig. 4, the IOLTS* $det(\mathcal{S}^\delta) \rhd \omega$ *is depicted in Figure 4 (note that* $\omega \in \Omega(det(\mathcal{S}^\delta))$*). For better readability, the Violate and Fail locations have been duplicated.*

Definition 10 deserves some comments. The forcing operation performs synchronization on visible actions whenever it is possible (line $[\alpha]$). However, for each *output* a that is not allowed by the specification M, $M \rhd \omega$ performs this output anyway. Intuitively, this is because the forcing operation is the first step towards test case generation, and the test cases are executed in parallel with an implementation $\mathcal{I}$ of the system to detect property violation and/or non-conformance:

- If the output a is not allowed by the specification M and leads the observer ω into its $\mathit{Violate}_\omega$ state, then $M \rhd \omega$ goes into its *Violate* state as well - cf. line $[\beta]$. If this happens when running $M \rhd \omega$ in parallel with an implementation $\mathcal{I}$, this means that $\mathcal{I}$ violates both the conformance to M and the property defined by ω. This is formalized by Theorem 2 below.

– if the output a is not allowed by M but does not lead the observer into its $Violate_\omega$ state (line $[\gamma]$), then $M \rhd \omega$ goes into its *Fail* state. If this happens when running $M \rhd \omega$ in parallel with an implementation $\mathcal{I}$, then $\mathcal{I}$ violates the conformance to M (but does not necessarily violate the property defined by ω). This is formalized by Theorem 3.

Note that if M is a deterministic IOLTS and $\omega \in \Omega(M)$, then $M \rhd \omega$ is an observer for M as well, i.e., $M \rhd \omega$ satisfies all the conditions of Definition 5. In particular, its language $\mathcal{L}(M \rhd \omega)$ is defined. The following lemma characterizes this language.

Lemma 2. *For IOLTS M and $\omega \in \Omega(M)$, $\mathcal{L}(M \rhd \omega) = \mathcal{L}(\omega) \cap [Traces(M) \cdot A_!^M]$.*

Proof : We prove for an arbitrary sequence $\sigma \in (A^{\mathrm{M}})^*$ that $\sigma \in \mathcal{L}(M \rhd \omega)$ iff $\sigma \in \mathcal{L}(\omega) \cap [Traces(M) \cdot A_!^{\mathrm{M}}]$.

First, by Definition 5, a sequence σ that belongs to either of these sets cannot be empty, because *Violate* cannot be the initial state of an observer.

Then, $\sigma = \sigma' \cdot a \in \mathcal{L}(M \rhd \omega)$ iff a is an output that has been forced to go to *Violate* in $M \rhd \omega$ by rule $[\beta]$ of Definition 10. But, by the same rule, this happens if and only if $\sigma \in \mathcal{L}(\omega)$ and $\sigma' \in Traces(M)$, i.e., $\sigma \in \mathcal{L}(\omega) \cap [Traces(M) \cdot A_!^{\mathrm{M}}]$. □

Lemma 2 says that $M \rhd \omega$ is a relevant observer for M (it actually says that $M \rhd \omega$ defines the strongest safety requirement weaker than ω and relevant for M).

Theorem 2 below is a refinement of Theorem 1, obtained by dropping the *relevance* hypothesis for the observer. Its proof invokes Theorem 1 and Lemma 2.

Theorem 2. *Let $\mathcal{I}$ and $\mathcal{S}$ be two IOLTS, then*

$$\mathcal{I} \ \mathbf{ioco} \ \mathcal{S} \ \Leftrightarrow \ \forall \omega \in \Omega(\mathcal{S}^\delta). \mathcal{S}^\delta \models \omega \ \Rightarrow \ \mathcal{I}^\delta \models (det(\mathcal{S}^\delta) \rhd \omega).$$

Proof :($\Rightarrow$) Assume $\mathcal{I}$ **ioco** $\mathcal{S}$ and let ω be an arbitrary observer in $\Omega(\mathcal{S})$ such that $\mathcal{S}^\delta \models \omega$. Thus, $Traces(\mathcal{S}^\delta) \cap \mathcal{L}(\omega) = \emptyset$, which implies $Traces(\mathcal{S}^\delta) \cap \mathcal{L}(\omega) \cap [Traces(\mathcal{S}^\delta) \cdot A_!^{\mathcal{S}^\delta}] = \emptyset$, which, by Lemma 2 is just $\mathcal{S}^\delta \models (det(\mathcal{S}^\delta) \rhd \omega)$.

Still by Lemma 2 we know that $det(\mathcal{S}^\delta) \rhd \omega$ is a relevant observer for $\mathcal{S}^\delta$. Then, using Theorem 1 we obtain: $\mathcal{S}^\delta \models (det(\mathcal{S}^\delta) \rhd \omega) \Rightarrow \mathcal{I}^\delta \models (det(\mathcal{S}^\delta) \rhd \omega)$.

By transitivity of $\Rightarrow$, we have $\mathcal{I}^\delta \models (det(\mathcal{S}^\delta) \rhd \omega)$, and this direction is done.

($\Leftarrow$): Assume $\forall \omega \in \Omega(\mathcal{S}^\delta). \mathcal{S}^\delta \models \omega \ \Rightarrow \ \mathcal{I}^\delta \models (det(\mathcal{S}^\delta) \rhd \omega)$. Then, in particular, we have that this implication is true for all *relevant* observers $\omega \in \varrho(\mathcal{S}^\delta)$, that is, $\forall \omega \in \varrho(\mathcal{S}^\delta). \mathcal{S}^\delta \models \omega \ \Rightarrow \ \mathcal{I}^\delta \models (det(\mathcal{S}^\delta) \rhd \omega)$. However, by Lemma 2, $\mathcal{I}^\delta \models (det(\mathcal{S}^\delta) \rhd \omega)$ implies $Traces(\mathcal{I}^\delta) \cap \mathcal{L}(\omega) \cap [Traces(\mathcal{S}^\delta) \cdot A_!^{\mathcal{S}^\delta}] = \emptyset$, and, by Definition 9, $\mathcal{L}(\omega) \subseteq [Traces(\mathcal{S}^\delta) \cdot A_!^{\mathcal{S}^\delta}]$. Hence, we have $Traces(\mathcal{I}^\delta) \cap \mathcal{L}(\omega) = \emptyset$, which implies $\mathcal{I}^\delta \models \omega$.

We have obtained that, for all *relevant* observers $\omega \in \varrho(\mathcal{S}^\delta)$, $\mathcal{S}^\delta \models \omega \ \Rightarrow \ \mathcal{I}^\delta \models \omega$ holds. By Theorem 1, we obtain $\mathcal{I}$ **ioco** $\mathcal{S}$, and the proof is done. □

Interpretation. For observers ω that represent *true* safety properties of a specification $\mathcal{S}$, whenever $det(\mathcal{S}^\delta) \rhd \omega$ enters its *Violate* state when executed in parallel with an implementation $\mathcal{I}$, then $\mathcal{I}$ violates both the safety property defined by ω (cf. Lemma 2) and the conformance to the specification (cf. Theorem 2).

Hence, $det(\mathcal{S}^\delta)\rhd\omega$ is the basis for a potentially interesting test case. When it enters its *Violate* state, the implementation will be assigned the **Violate** verdict:

Violate: The implementation violates both the property and the conformance

We now consider the situation when $det(\mathcal{S}^\delta)\rhd\omega$ enters its *Fail* state.

Theorem 3. *For IOLTS $\mathcal{I}$, $\mathcal{S}$ and $\omega \in \Omega(\mathcal{S}^\delta)$, if there exists $\sigma \in Traces(\mathcal{I}^\delta)$ $\cap Traces(det(\mathcal{S}^\delta)\rhd\omega)$ such that $Fail \in (det(\mathcal{S}^\delta)\rhd\omega)$ after σ, then $\neg(\mathcal{I}$ **ioco** $\mathcal{S})$.*

Proof : By Definition 10 line $[\gamma]$, $Fail \in (det(\mathcal{S}^\delta)\rhd\omega)$ *after* σ means that $\sigma = \sigma' a$, where a is an output ($a \in A_!$) which is not fireable in $det(\mathcal{S}^\delta)$ after the sequence σ', i.e., $a \notin Out(\mathcal{S}^\delta$ *after* $\sigma')$. However, by $\sigma \in Traces(\mathcal{I}^\delta)$ we have that $a \in Out(\mathcal{I}^\delta$ *after* $\sigma')$. Then, by Definition 8, $\neg(\mathcal{I}$ **ioco** $\mathcal{S})$. □

Interpretation. Theorem 3 says that when $det(\mathcal{S}^\delta)\rhd\omega$ enters *Fail* when run on an implementation, the latter violates conformance to the specification (but not necessarily the property ω). In this case, the **Fail** verdict is given:

Fail: The implementation violates the conformance but not necessarily the property

What remains to do is to build from $det(\mathcal{S}^\delta)\rhd\omega$ an actual test case.

Mirror. The next step consists in transforming all inputs of $det(\mathcal{S}^\delta)\rhd\omega$ into outputs and reciprocally. This is called the mirror operation. It is necessary because, in the test execution process, the actions of the implementation and those of the test case must complement each other.

Pruning. This operation consists in suppressing from $det(\mathcal{S}^\delta)\rhd\omega$ the subgraphs that cannot lead to *Violate*. Here, the main goal of testing is to check the violation of the requirement after a trace of the specification, and, if an implementation leads a tester (extracted from the specification) into a subgraph that cannot lead to *Violate*, the current test experiment will never be able to achieve this goal.

There are two situations, depending on whether the subgraph (from which *Violate* is unreachable) was entered through an input or an output:

- the subgraph has been entered by an *output* of the tester. In this case, the transition labeled by that output (together with the whole subgraph), are removed. Intuitively, the tester has control over its outputs, thus, it may decide not to stimulate the implementation with an output if it knows that this will never lead to a **Violate** verdict.
- the subgraph has been entered by an *input* of the tester (that does not directly lead to *Fail*). In this case, only the transition labeled by that input is kept (the rest of the graph is removed). The destination of the transition is set to a new state called *Inconc*, which means that no *Violate* verdict can be given any more (but the conformance was not violated). Hence, for completeness, in this situation the verdict will be **Inconc** (inconclusive).

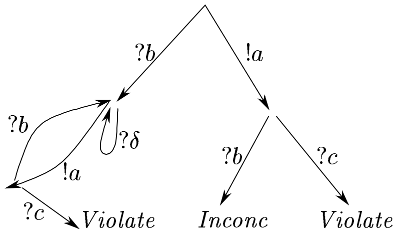

Figure6. Test case generated from $\mathcal{S}$ and ω (except the *Fail* location).

Inconc: neither **Fail** nor **Violate** have occurred and **Violate** cannot occur any more

Let $test(S, \omega) = prune(mirror(det(\mathcal{S}^{\delta}) \rhd \omega))$ denote the IOLTS obtained after these operations. $test(\mathcal{S}, \omega)$ is the test case generated from specification $\mathcal{S}$ and observer ω. It is not hard to see that by replacing $det(\mathcal{S}^{\delta}) \rhd \omega$ by $test(\mathcal{S}, \omega)$ in the statements of Theorems 2, 3 the proofs still hold. This is because $test(\mathcal{S}, \omega)$ satisfies Lemma 2 as well, i.e., $\mathcal{L}(test(S, \omega)) = \mathcal{L}(det(\mathcal{S}^{\delta}) \rhd \omega) = \mathcal{L}(\omega) \cap [Traces(\mathcal{S}^{\delta}) \cdot A_!^{\mathcal{S}^{\delta}}]$, as only subgraphs that *cannot lead to Violate* have been suppressed by pruning.

The above property of the language of $test(\mathcal{S}, \omega)$ is enough to establish Theorem 2.

On the other hand, Theorem 3 is concerned with traces of $det(\mathcal{S}^{\delta}) \rhd \omega$ that lead to *Fail*, and a trace that leads to *Fail* in $test(\mathcal{S}, \omega)$ also leads to *Fail* in $det(\mathcal{S}^{\delta}) \rhd \omega$.

This establishes that Theorem 3 still holds when $det(\mathcal{S}^{\delta}) \rhd \omega$ is replaced by $test(\mathcal{S}, \omega)$.

Example 10. The test case depicted in Figure 6 checks the property: "between each $?a$ and $!c$, there is at least one $!b$" (cf. Example 2). The Fail location has not been represented; there is an implicit transition to Fail from each state, labeled by all the input actions that do not go anywhere else. Implementation $\mathcal{I}_2$ from Example 5, which initially receives an $?a$ and then emits $!b$ (cf. right of Figure 4) violates neither the property nor conformance to $\mathcal{S}$ on this trace, and our test case will never be able to detect violation of the requirement in the future, hence, the **Inconc** *verdict is assigned. An implementation that receives an $?a$ initially and then directly emits $!c$ violates both the property and the conformance, hence, the verdict is* **Violate**. *Finally, an implementation that emits $!\delta$ initially, violates the conformance to the specification (but not the property): the verdict is* **Fail**.

5 Conclusion, Related Work, and Future Work

We now recall the framework proposed in the paper and the main results obtained. A system is viewed at three different levels of abstraction: high-level *requirements* $\mathcal{R}$, operational *specification* $\mathcal{S}$, and final, black-box *implementation* $\mathcal{I}$. In the proposed framework, all three views are modeled using Input-Output Labeled transition systems, which are labeled transition systems whose actions are partitioned into inputs, outputs, and internal actions.

The conformance relation **ioco** [19] links $\mathcal{I}$ and $\mathcal{S}$, and the satisfaction relation $\models$ links $\mathcal{S}$ (or $\mathcal{I}$) and requirements $\mathcal{R}$. A notion of requirement that is *relevant* for a specification is defined, which essentially means that the requirement does not refer to features that are not incorporated into the specification.

Our first result says intuitively that $\mathcal{I}$ **ioco**-conforms to $\mathcal{S}$ if and only if $\mathcal{I}$ satisfies all relevant requirements that are satisfied by $\mathcal{S}$. While it is interesting from a theoretical point of view, because it gives an alternative definition of **ioco**-conformance, this result is not practical as it is restricted to *relevant* requirements (which are not always easy to come up with) and it does not say how to actually check a requirement on a black-box implementation.

This can be done by testing, and therefore we propose a test generation algorithm that takes an IOLTS specification $\mathcal{S}$, and an arbitrary requirement ω (relevant or not) expressed using specific IOLTS called *observers*, and produces an IOLTS test case, denoted $test(\mathcal{S}, \omega)$. The tester emits verdicts: **Violate**, **Fail**, and **Inconc**, which express relations between $\mathcal{I}$, $\mathcal{S}$, and $\mathcal{R}$.

Our second result says intuitively that $\mathcal{I}$ **ioco** $\mathcal{S}$ holds if and only if, for all observers ω that express *true* properties of the specification $\mathcal{S}$, by executing $test(\mathcal{S}, \omega)$ in parallel with the implementation $\mathcal{I}$, the **Violate** verdict is not obtained.

The "only if" part of this result is a theoretical completeness result: in order to establish conformance, an infinite number of test cases should be executed, and each execution is a potentially infinite process as well.

More interesting in practice is the "if" part of the result, which is a soundness property. Going back to the validation approach proposed in Section 1, it implies that every requirement that holds on the specification allows to automatically generate a test case to check the requirement on the implementation as well.

A similar soundness result holds for the **Fail** verdict, which says that, if the **Fail** verdict is obtained by executing $test(\mathcal{S}, \omega)$ in parallel with the implementation $\mathcal{I}$, then $\mathcal{I}$ **ioco** $\mathcal{S}$ does not hold. The **Fail** verdict plays the role of a warning in the proposed validation process: when it is issued, the implementation has violated conformance to the specification, and the trace that led to this violation can be examined to check whether this corresponds to a serious problem or not.

Finally, an **Inconc** (inconclusive) verdict means that the current test experiment will never be able to detect the violation of a requirement in the future, therefore, the user may stop the current test experiment and start another one.

Related Work. There exists a lot of interest in formal verification from researchers and, recently, formal verification has started to penetrate the industry. More recently, conformance testing (and other forms of testing) have become a topic of interest to the verification community. This has resulted in new algorithms and tools for testing based on verification technology (mainly model checking) [1,4,11,18].

In [6] the authors describe an approach to generate tests from observers describing linear-time temporal logic requirements and to execute the tests directly on the implementation. This is similar to what we do, except for the logic and one more important point: [6] does not require verification of the property on the specification prior to conformance testing, and the test cases do not check conformance, but only that the implementation does not violate the requirements.

The authors of [15] have ideas similar to ours. Given a specification S, and an invariant P assumed to be satisfied by S, mutants S' of S are built using standard mutation operators. Then, a combined machine is generated, which extends sequences of S with sequences of S'. Next, the SMV model-checker is used to generate sequences that violate P, i.e., sequences that prove that S' is a mutant of S violating P. Finally, the sequences are interpreted as test cases. The construction of the combined machine is quite similar to our forcing operation (with S' interpreted as ω). Several other papers, like [7,5], start from a specification S and a property P in a temporal logic (CTL or LTL) satisfied by S, and use the counter-example facility of a model checker (SMV, SPIN) to generate counter-examples of $\neg P$, thus, witnesses of P in S. Some papers like [8] extend the idea to describe coverage criteria in temporal logic and generate test cases using model-checking.

However, all these papers suffer from the same drawbacks. They do not take nondeterminism into account, do not differentiate between inputs and outputs, and do not formally define conformance testing. Moreover, except [15], they do not relate satisfaction of properties to conformance testing.

In [17] another approach to combine test selection and verification is presented. The idea is to use symbolic test selection techniques to extract test cases from a specification, which, under some sufficient conditions, can be used to perform a compositional verification of the requirements. However, the test selection mechanism is not related to the requirements in a formal way, as it is in this paper.

Future work. In the near future we are planning to implement the test generation method for safety properties presented here in the TGV tool [11]. TGV uses *test purposes* as test selection mechanisms, which express *reachability* properties of the specification; a test case is generated for every witness (trace) showing that the specification satisfies a given reachability property [11]. Thus, to perform test generation from safety properties, it is *not* enough to take the negation of a safety property, and use TGV with the resulting reachability property as a test purpose: if the specification satisfies the safety property (as assumed everywhere in this paper), its negation has no witnesses, thus, TGV produces an empty test case.

Our symbolic test generation tool STG [4], based on abstract interpretation rather than enumeration, is another target for the new test generation algorithm.

We are also planning to extend this framework to LTL safety formulas [13] by a translation of LTL formulas on observable events into observers, and connect this work with that of [9,3]. The problem addressed by these papers is to check whether temporal logic formulas expressing safety requirements have a sufficient coverage of the specification. Here, coverage is defined by the ability of a formula to distinguish mutants. Thus, if a set of requirements has a good coverage of the specification, the test cases obtained by our method may have a good coverage on implementations, thus, a good chance of finding bugs during test execution.

Finally, in this paper the situation where the specification does *not* satisfy the requirements has not been considered. We are currently investigating an approach based on previous work [12] for automatically computing the largest specification contained in the original specification, which satisfies the requirements and does not change the set of implementations that conform to the original specification.

References

1. A. Belinfante, J. Feenstra, R. de Vries, J. Tretmans, N. Goga, L. Feijs, and S. Mauw. Formal test automation: a simple experiment. In *International Workshop on the Testing of Communicating Systems (IWTCS'99)*, pages 179–196, 1996.
2. 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. *Science of Computer Programming*, 36(1):27–52, 2000.
3. H. Chockler, O. Kupferman, R.P. Kurshan, and M.Y. Vardi. A practical approach to coverage in model checking. In *Computer-Aided Verification (CAV'01)*, number 2102 in LNCS, pages 66–78, 2001.
4. 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.
5. A. Engels, L.M.G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, number 1217 in LNCS, pages 384–398, 1997.
6. J.C. Fernandez, L. Mounier, and C. Pachon. Property-oriented test generation. In *Formal Aspects of Software Testing Wokrshop (FATES'03)*, 2003.
7. A. Gargantini and C.L. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC / SIGSOFT FSE*, pages 146–162, 1999.
8. 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.
9. Y.V. Hoskote, T. Kam, P.-H. Ho, and X. Zhao. Coverage estimation for symbolic model checking. In *Design Automation Conference*, pages 300–305, 1999.
10. ISO/IEC. International Standard 9646, OSI-Open Systems Interconnection, Information Technology - Conformance Testing Methodology and Framework, 1992.
11. T. Jéron and P. Morel. Test generation derived from model-checking. In *Computer-Aided Verification (CAV'99)*, number 1633 in LNCS, pages 108–122, 1999.
12. T. Jéron, H. Marchand, V. Rusu, and V. Tschaen. Synthèse de contrôleurs pour une relation de conformité. In *Modélisation des systèmes réactifs (MSR'03)*, 2003.
13. O. Kupferman and M.Y. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
14. Z. Manna and A. Pnueli. *Temporal verification of reactive systems. Vol. 1: Specification, Vol. 2: Safety.* Springer-Verlag, 1991 and 1995.
15. P.Ammann, W. Ding, and D. Xu. Using a model checker to test safety properties. In *International Conference on Engineering of Complex Computer Systems (ICECCS'01)*. IEEE Computer Society, 2001.
16. Telelogic SDL products. `http://www.telelogic.com/products/sdl`.
17. V. Rusu. Combining formal verification and conformance testing for validating reactive systems. *Software Testing, Verification, and Reliability*, 13(3):157–180, 2003.
18. M Schmitt, A. Ek, J. Grabowski, D. Hogrefe, and B. Koch. Autolink - putting SDL-based test generation into practice. In *International Workshop on the Testing of Communicating Systems (IWTCS'97)*, pages 227–244, 1997.
19. J. Tretmans. Testing concurrent systems: A formal approach. In *Concurrency Theory (CONCUR'99)*, number 1664 in LNCS, pages 46–65, 1999.