INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1640

# VERIFICATION AND SYMBOLIC TEST GENERATION FOR SAFETY PROPERTIES

VLAD RUSU, HERVÉ MARCHAND, AND THIERRY JÉRON

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

# Verification and Symbolic Test Generation for Safety Properties

Vlad Rusu, Hervé Marchand, and Thierry Jéron

Systèmes communicants
Projet VerTeCs

Publication interne n1640 — août 2004 — 20 pages

**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 - an input-output automaton with variables that may range over infinite domains - is assumed. Additionally, a set of safety properties for the specification are given under the form of observers described in the same formalism. Then, each property is verified on the specification using automatic techniques (e.g., abstract interpretation) that are sound but not necessarily complete for the class of safety properties considered here. 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. If the verification step was successful, that is, it has established that the specification satisfies the property, then the test execution may detect the violation of the property by the implementation and the violation of the standard **ioco** conformance relation [18] between implementation and specification. On the other hand, if the verification step did not conclude (i.e., it did not allow to prove or to disprove the property), then the test execution may additionally detect a violation of the property by the specification.

The informations about the relative (in)consistencies between specification, implementation, and properties are reported to the user as test verdicts. The approach is illustrated on the BRP protocol [9].

**Key-words:** symbolic verification, conformance testing, symbolic test generation

*(Résumé : tsvp)*

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

# Vérification et génération de tests symboliques pour des propriétés de sûreté

**Résumé :** Cet article présente une combinaison de techniques de vérification et de test de conformité pour la vérification de systèmes réactifs. A partir d'une spécification formelle d'un système réactif, décrit sous la forme d'un automate à entrées-sorties étendu avec des variables portant sur des domaines finis ou infinis, et d'un observateur (décrit dans le même formalisme) pour une propriété de sûreté, la première étape consiste à vérifier automatiquement la propriété sur la spécification par des techniques approchées (à base d'interpretation abstraite) qui sont conservatives pour la classe de propriétés considérée. Ensuite, un cas de test est automatiquement généré et exécuté sur une implémentation boîte-noire du système. Si la vérification a permis d'établir que la propriété est satisfaite par la spécification, l'exécution du cas de test peut détecter des violations de la propriété par l'implémentation et/ou la violation de la relation de conformité **ioco** [18] entre implémentation et spécification. En revanche, si la vérification n'a pas permis d'établir si la spécification satisfait la propriété ou non (ce qui peut arriver en pratique pour des systèmes infinis), l'exécution du cas de test peut, en plus, détecter la violation de la propriété par la spécification. Les informations ainsi obtenues sur les comportements relatifs de la spécification, de l'implémentation, et de la propriété sont retournées à l'utilisateur sous la forme de verdicts de test. L'approche est illustrée sur le protocole BRP [9] de Phillips.

**Mots clés :** vérification symbolique, test de conformité, génération de tests symboliques

# 1 Introduction

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

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

Conformance testing is typically performed by automatically deriving *test cases* from the specification, and by running the test cases on the implementation to obtain *test verdicts* about the conformance between the two representations of the system [3, 6, 12, 17, 18, 19].

For example, in [12], enumerative model-checking algorithms are employed for generating test cases from an operational specification and so-called *test purposes* as guidelines for test selection. Test generation consists in exploring a product between specification and test purpose, eliminating nondeterminism, and incorporating verdicts into the result; and the generated test cases check the conformance between the implementation and the part of the specification selected by the test purposes.

An alternative approach is proposed in [15]. Rather than exploring the (large, or potentially infinite) state space of the product between test purpose and specification, the product is kept in a symbolic form. The resulting test cases are reactive programs that operate on symbolic variables, and exchange values with the implementation by means of input/output actions carrying symbolic parameters.

In [16], we propose to integrate the verification of safety properties and conformance testing in a formal validation methodology for reactive systems:

- first, the safety properties are automatically verified on the specification by model checking,
- then, test cases are automatically derived from the specification and the properties,
- finally, the test cases are executed on the black-box implementation of the system, which allows to check the satisfaction of the properties by the implementation and the conformance between implementation and specification.

The present paper proposes a symbolic approach that extends [16], very much like [15] extends [12]:

- the verification and the test generation steps are symbolic, rather than enumerative;
- safety properties, rather than test purposes, are employed for test generation.

A major difference between [16] and this paper concerns the decidability of the verification step. In the finite-state framework of [16], the verification is decidable and is performed by model checking. In the present framework that considers infinite-state systems and properties, the verification is undecidable; it is performed using automatic techniques based on abstract interpretation that are sound but, as a price to pay for automation, are not necessarily complete for the class of safety properties considered here. If the verification succeeds, then, the property is satisfied by the specification; otherwise, the status of the property is unknown, and the test generation algorithm has to take this into account. Hence, the test execution produces verdicts, which may reveal one or several of the following situations:

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

The information obtained in this manner may be employed for detecting and fixing errors in the specification, the implementation, or the property. The verification and testing steps may be repeated, until sufficient confidence is gained about the consistency of the three representations.

The rest of the paper is organised as follows. Section 2 presents the model of Input-Output Symbolic Transition Systems (IOSTS) that we use for modeling all aspects of a system (specifications, properties, implementations, and test cases). An example (the sender of the BRP protocol [9]) illustrates the model. In Section 3 we define several symbolic operations on IOSTS, together with some fairly standard definitions relative to verification and conformance testing using IOSTS as the underlying model. Section 4 is devoted to defining a symbolic test generation algorithm that takes a specification and a safety property, and produces a test case for checking the conformance of an implementation with the specification, and the satisfaction of the safety property by the implementation. As discussed above, the test case may also detect violations of the property by the specification. The test generation algorithm is proved correct, in the sense that the verdicts returned by test execution correctly characterise the relations between implementation, specification, and property. As a by-product of the correctness proofs we show that conformance to a given specification is equivalent to satisfaction of a safety property, which can be expressed as an observer built from the specification; as far as we know, this rather general result connecting verification and conformance testing is new. In Section 5 we outline a technique for statically eliminating parts of a test case from which violations of the property by the implementation (arguably, the main goal of testing) cannot be detected any more. We show that this optimisation preserve the correctness of test cases. Section 6 concludes and presents related work.

## 2 The IOSTS Model

The model of Input-Output Symbolic Transition Systems (IOSTS) is inspired from I/O automata [13]. Unlike I/O automata, IOSTS do not require *input-completeness* (all input actions 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 nonempty, finite set of typed* data, *which is the disjoint union of a set* $V$ *of* variables, *a set* $C$ *of* symbolic constants *and a set* $P$ *of* parameters. *For* $d \in D$, *type*$(d)$ *denotes the type of* $d$.
- $\Theta$ *is the* initial condition, *a Boolean expression on* $V \cup C$,
- $Q$ *is a nonempty, finite set of* locations.
- $q^0 \in Q$ *is the* initial location.
- $\Sigma$ *is a nonempty, finite* alphabet, *which is the disjoint union of a set* $\Sigma^?$ *of* input actions, *a set* $\Sigma^!$ *of* output actions, *and a set* $\Sigma^{int}$ *of* internal actions. *For each action* $a \in \Sigma$, *its* signature *sig*$(a)$ *is a tuple of types. The signature of internal actions is the empty tuple.*
- $\mathcal{T}$ *is a set of* transitions. *Each transition is a tuple* $\langle q, a, \pi, G, A, q' \rangle$ *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 tuple* $\langle \pi_i \rangle_{i=1\ldots k}$ *of parameters such that if* $sig(a) = \langle \vartheta_j \rangle_{j=1\ldots l}$ *then* $k = l$ *and* $\forall i = 1 \ldots k$, $type(\pi_i) = \vartheta_i$.
  - *a Boolean expression* $G$ *on* $V \cup C \cup \pi$, *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* $V \cup C \cup \pi$.
  - *a location* $q' \in Q$ *called the* destination *of the transition.*

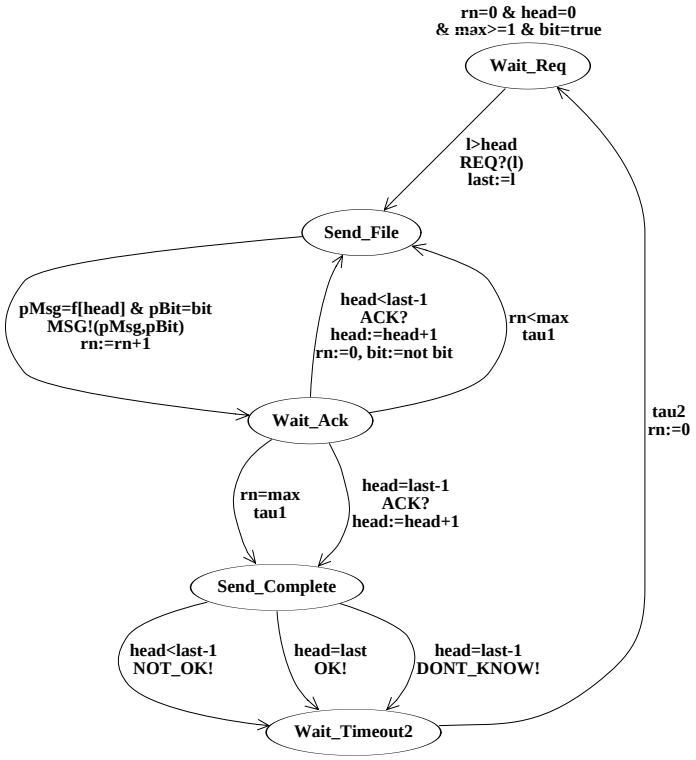



Figure 1: : the Sender of the BRP Protocol.

Our running example is the specification of the BRP protocol [9], which was designed to safely transmit files over an unsafe medium. The protocol uses a combination of alternating-bit and resend-on-timeout mechanisms. Here, we are only interested in the sender of the protocol, which is represented in Figure 1.

The corresponding IOSTS has variables $rn$, $head$, $last$, and $bit$, symbolic constants $max$ and $f$, and parameters $l$, $pMsg$, and $pBit$. Except for $bit$, $pBit$, and $f$ all the other data are of integer type; $bit$ is a Boolean modeling the alternating bit; $f$ (the file to be sent) is an array of integers; $pMsg$ is an integer that models the current message being sent, together with the Boolean parameter $pBit$.

The variable $last$ (resp. the constant $max$) stand for the number of packets in the file to be sent (resp. the maximum number of retransmissions of a packet). The variable $head$ indicates the current packet ($f(0), f(1), \ldots$ ) to be sent, and the variable $rn$ is employed for counting retransmissions.

Consider for example the transition with origin $Send\_File$ and destination $Wait\_Ack$. It has the output action $MSG$ with parameters $pMsg, pBit$, guard $(pMsg = f[head]) \wedge (pBit = bit)$, and assignment $rn := rn + 1$, $bit := bit$, $f := f$, $head := head$. In the graphical representation, the symbol $?$ (resp. $!$) denotes an input (resp. an output) action, and identity assignments ($rn := rn$) are omitted.

An IOSTS starts in the initial location with values of the variables and symbolic constants satisfying the initial condition, and proceeds by firing transitions, updating the variables according to the guards and assignments of the transitions fired, and exchanging information with the environment through input and output actions. For example, the sender of the BRP protocol (cf. Figure 1) initially waits for a $REQ$ input. On reception of this input with parameter $l > head$, the value of $l$ is saved in the variable $last$. Then, $f(0), f(1)$,*etc* are sent to the receiver (output $MSG$) together with an alternating bit. A packet may be acknowledged (input $ACK$), or there may be a timeout (internal action $\tau_1$),

after which a the packet may be resent at most $max - 1$ more times. Finally, the status of the whole transmission is reported to the environment by an output action: either $OK$ (all packets were sent and acknowledged); $DONT\_KNOW$ (all packets were sent and all but the last were acknowledged, meaning that either the last packet or the last acknowledgment were lost); or $NOT\_OK$ (some intermediary packet was not acknowledged).

**Semantics.** More formally, a *state* is a pair $\langle l, v\rangle$ where $l$ is a location and $v$ is a valuation for the variables and the symbolic constants. The set of valuations is denoted $\mathcal{V}$, and the set $Q \times \mathcal{V}$ of states is denoted by $S$. An *initial state* is a state $\langle l_0, v_0\rangle$ such that $l_0$ is the initial location and $v_0$ satisfies the initial condition. The set of initial states is denoted $S^0$. A *valued action* is a pair $\langle a, w\rangle$ where $a$ is an (input, output, or internal) action and $w$ is a valuation for the parameter(s) carried by the action. Note that the values of parameters are not contained in states, but in valued actions; to be memorised, the values of parameters have to be assigned to variables, otherwise they are lost. The set of valued actions is denoted by $\Lambda$; it is the disjoint union of the set of valued *inputs* $\Lambda^?$, valued *outputs* $\Lambda^!$, and valued internal actions $\Lambda^{int}$. The latter is identified with the set $\Sigma^{int}$ of internal actions.

For $t$ a transition of the IOSTS, the *transition relation* $\rightarrow_t$ is the set of triples $\langle s, \alpha, s'\rangle \in S \times \Lambda \times S$, where $s = \langle l, v\rangle$, $s' = \langle l', v'\rangle$ are states and $\alpha = \langle a, w\rangle$ is a valued action such that $l, l'$, and $a$ are respectively the origin, the destination, and the action labeling $t$; the values of variables, symbolic constants and actions defined by $v$, $w$ satisfy the guard $G$ of the transition $t$; and the valuation $v'$ is obtained from $v, w$ by executing the (parallel) assignments of transition $t$.

The global transition relation $\rightarrow$ is $\rightarrow = \bigcup_{t\in\mathcal{T}} \rightarrow_t$, where $\mathcal{T}$ denotes the set of transitions of the IOSTS. We write $s \xrightarrow{\alpha} s'$ for $(s, \alpha, s') \in \rightarrow$.

**Definition 2 (behavior)** *A behavior fragment is a sequence of alternating states and valued actions* $\beta: \quad s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \cdots s_{n-1} \xrightarrow{\alpha_{n-1}} s_n$. *A* behaviour *is a behavior fragment that starts in an initial state.*

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

**Definition 3 (coreachability)** *A state $s$ is* coreachable *for a set of states $S'$ if there exist $\sigma \in \Lambda^*$ such that $s \xrightarrow{\sigma} S'$. The set of states that are coreachable for $S'$ is called the* coreachable set *of $S'$.*

We often identify a location $q \in Q$ with the set of states $\{\langle q, v\rangle | v \in \mathcal{V}\}$ ($\mathcal{V}$ denotes the set of valuations of the variables and symbolic constants of the IOSTS). Then, the notation $s \xrightarrow{\sigma} q$ and the notions of reachable location and coreachable set of a given location are also defined. A trace is the subsequence of a behavior containing only what is externally visible, that is, valued inputs and valued outputs:

**Definition 4 (trace)** *The* trace *of a behavior $\beta$ is the projection of $\beta$ on $\Lambda^! \cup \Lambda^?$. The set of traces of an IOSTS $\mathcal{S}$ (i.e., the set of traces of all behaviours of $\mathcal{S}$) is denoted by Traces($\mathcal{S}$).*

## 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 (for verification) and the conformance relation (for conformance testing) between IOSTS are formally defined.

### 3.1 Parallel Product

The parallel product operation lets each system perform independently its internal actions and imposes synchronization on the shared input and output actions. The operation is defined for *compatible* IOSTS.

**Definition 5 (Compatible IOSTS)** *For $j = 1, 2$, the two IOSTS $\mathcal{S}_j = \langle D_j, \Theta_j, Q_j, q_j^0, \Sigma_j, \mathcal{T}_j \rangle$ with data $D_j$ and alphabet $\Sigma_j = \Sigma_j^? \cup \Sigma_j^! \cup \Sigma_j^{int}$ are* compatible *if $D_1 \cap D_2 = \emptyset$, $\Sigma_1^! = \Sigma_2^!$, $\Sigma_1^? = \Sigma_2^?$, $\Sigma_1^{int} \cap \Sigma_2^{int} = \emptyset$, and each $a \in \Sigma_1^? \cup \Sigma_1^!$ $(= \Sigma_2^? \cup \Sigma_2^!)$ has the same signature in both systems.*

**Definition 6 (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: $V = V_1 \cup V_2$, $C = C_1 \cup C_2$, $P = P_1 \cup P_2$, $\Theta = \Theta_1 \wedge \Theta_2$, $Q = Q_1 \times Q_2$, $q^0 = \langle q_1^0, q_2^0 \rangle$, $\Sigma^? = \Sigma_1^? = \Sigma_2^?$, $\Sigma^! = \Sigma_1^! = \Sigma_2^!$, $\Sigma^{int} = \Sigma_1^{int} \cup \Sigma_2^{int}$. The set $\mathcal{T}$ of transitions of the composed system is defined as follows:*

- *for each transition $\langle q_1, a, \pi, G, A, q'_1 \rangle \in \mathcal{T}_1$ with $a \in \Sigma_1^{int}$ and for each location of the form $\langle q_1, q_2 \rangle$ with $q_2 \in Q_2$, there is a transition of the form $\langle \langle q_1, q_2 \rangle, a, \pi, G, A, \langle q'_1, q_2 \rangle \rangle$ in $\mathcal{S}_1 || \mathcal{S}_2$ (and similarly for all $a \in \Sigma_2^{int}$),*
- *for all transitions $\langle q_1, a, \pi_1, G_1, A_1, q'_1 \rangle \in \mathcal{T}_1$, $\langle q_2, a, \pi_2, G_2, A_2, q'_2 \rangle \in \mathcal{T}_2$ with $a \in \Sigma_1^?$, let $G_{1,2}$ (resp. $A_{1,2}$) denote the expression obtained by syntactically replacing in $G_2$ (resp. $A_2$) each parameter from $\pi_2$ by the corresponding, same-position parameter from $\pi_1$. Then, in $\mathcal{S}_1 || \mathcal{S}_2$ there is a transition $\langle \langle q_1, q_2 \rangle, a, \pi_1, G_1 \wedge G_{1,2}, A_1 \cup A_{1,2}, \langle q'_1, q'_2 \rangle \rangle$ (and similarly when $a \in \Sigma_1^!$).*

**Lemma 1** ([19]) *$Traces(\mathcal{S}_1 || \mathcal{S}_2) = Traces(\mathcal{S}_1) \cap Traces(\mathcal{S}_2)$.*

### 3.2 Quiescence, and quiescent IOSTS

We here assume 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 [18]. 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. However, on a (white-box) specification such as that depicted in Figure 1, quiescence is materialized by adding a special output action $\delta$, which may occur in a location if and only if no other output or internal actions are fireable in that location.

**Definition 7 (quiescent IOSTS)** *Given an IOSTS $\mathcal{S}$ with set of variables $V$, actions $\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^{int}$, and transitions $\mathcal{T}$, the* quiescent *IOSTS $\mathcal{S}^\delta$ is obtained from $\mathcal{S}$ by adding a new output action $\delta$ to $\Sigma^!$, and, for each location $l$ of $\mathcal{S}$, a new self-looping transition, labeled with action $\delta$, with identity assignments (i.e., $x := x$ for all variables $x \in V$), and whose guard is defined by the expression*

$$\bigwedge_{a \in \Sigma^{int} \cup \Sigma^!} \neg G(a) \tag{1}$$

*where*

$$G(a) : \bigvee_{t = \langle l, a, \pi, G_t, A_t, l'_t \rangle \in \mathcal{T}} \exists \pi : sig(a).G_t(\pi) \tag{2}$$

*Interpretation:* The new output action $\delta$ may be fired in a location $l$ iff no other outputs or internal actions may be fired in $l$. Now, an action $a$ may be fired in $l$ iff there exists a transition $t = \langle l, a, \pi, G_t, A_t, l'_t \rangle \in \mathcal{T}$ with action $a$, and values for the parameters $\pi$ carried by $a$ (which can be seen as a single vector value of type $sig(a)$, cf. Definition 1) such that, by replacing the parameters $\pi$

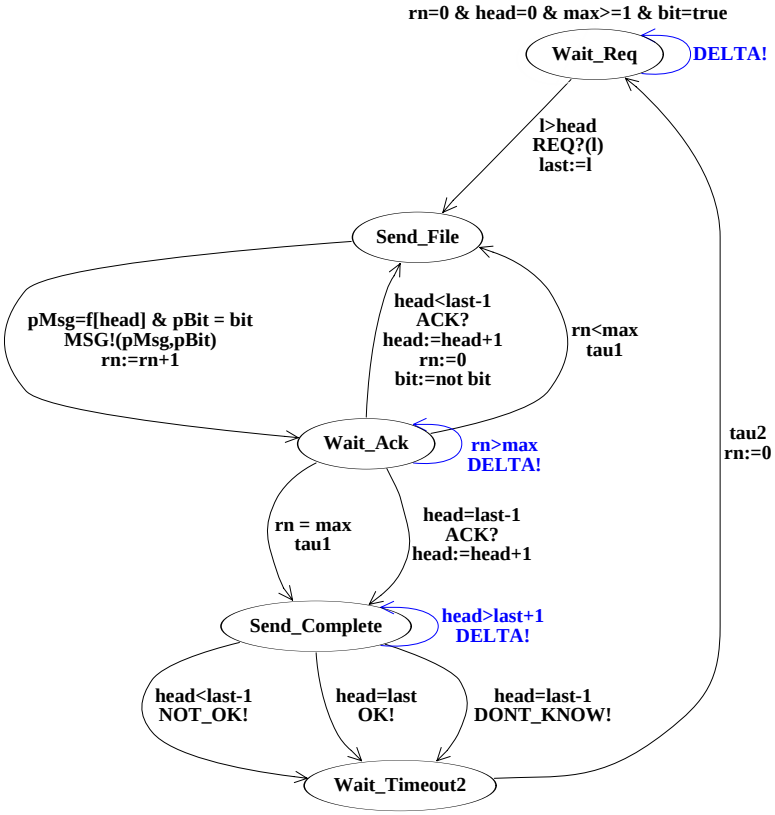



Figure 2: : the IOSTS $Sender^{\delta}$.

with their values, the resulting formula $G_t(\pi)$ is satisfiable. Hence, Formula (2) takes the disjunction, for all transitions $t = \langle l, a, \pi, G_t, A_t, l'_t \rangle \in \mathcal{T}$ and valuations of $\pi$[1], of the formulas $G_t(\pi)$.

The result, denoted $G(a)$ in Formula (2), gives the conditions on the system's variables and symbolic constants, under which the action $a$ is fireable in $l$. Finally, Formula (1) says that $\delta$ may be fired in $l$ iff no other output or internal action is fireable in $l$. Hence, it takes the *conjunction* of the *negations* of all formulas $G(a)$, for all $a \in \Sigma^{int} \cup \Sigma^{!}$.

**Example.** For the IOSTS *Sender* depicted in Fig. 1, the IOSTS $Sender^{\delta}$ is depicted in Fig. 2. In the *Send_ File* location, Formula (1) becomes $\neg\exists\langle pMsg : \mathsf{int}, pBit : \mathsf{bool}\rangle.(pMsg = f[head] \wedge pBit = bit)$, which reduces to *false*. Similarly, in *Wait_ Timeout2*, the $\tau_2$ action is always fireable, and Formula (1) evaluates to *false*. Hence, $\delta$ is not fireable in locations *Send_ File* and *Wait_ Timeout2*. On the other hand, $\delta$ might be fireable in the three remaining locations: *Wait_ Req*, *Wait_ Ack*, and *Send_ Complete*. For example, the guard of the $\delta$-labeled transition on location *Wait_ Ack* is $rn > max$, because: there are two outgoing transitions labeled by an internal action ($\tau_1$), which have the guards $rn = max$ and $rn < max$, respectively; and no outgoing transitions are labeled by outputs. (Of course, just as any other action, $\delta$ is actually fireable only if its guard actually evaluates to *true* in a reachable state.)

[1] disjunction over all valuations of $\pi$ is encoded using existential quantification.

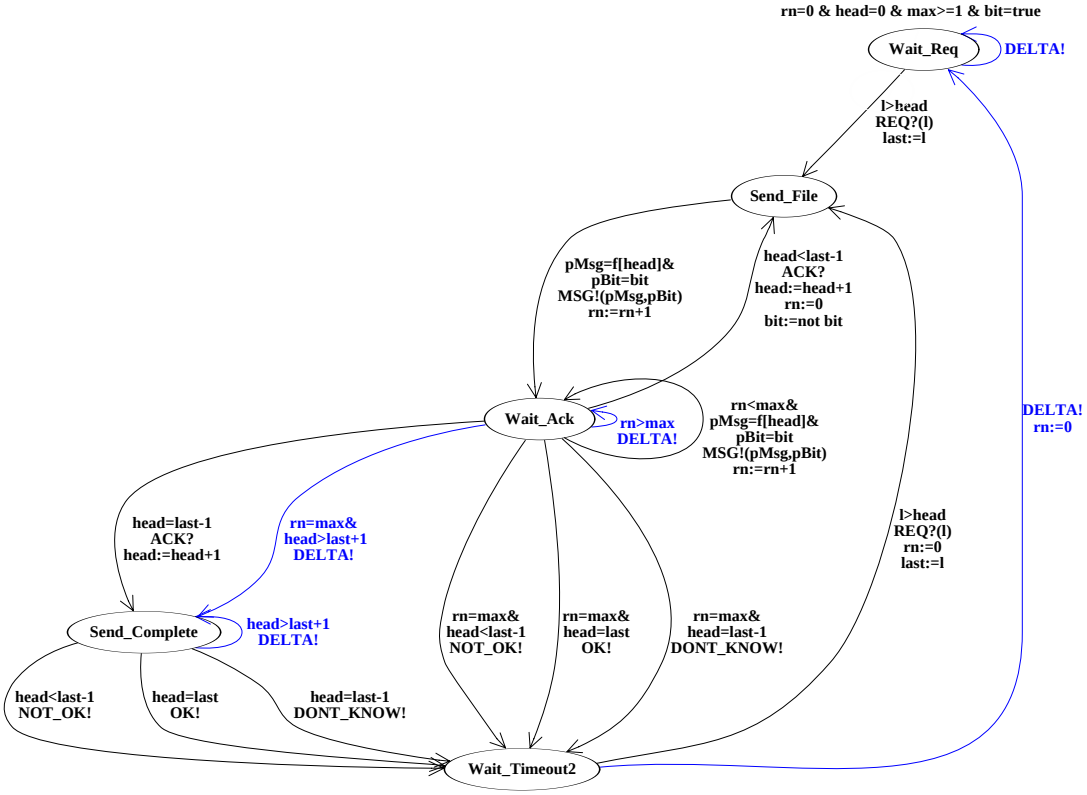



Figure 3: the IOSTS $det(Sender^{\delta})$.

## 3.3 Deterministic IOSTS, and determinisation

An IOSTS is *deterministic* if it does not have internal actions, and in each location, the guards of transitions labeled by the same action are mutually exclusive.

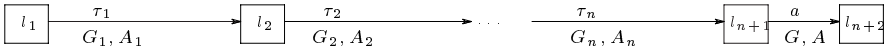
**Definition 8 (deterministic IOSTS)** *An IOSTS* $\langle D, \Theta, Q, q^0, \Sigma, \mathcal{T} \rangle$ *is deterministic if* $\Sigma^{int} = \emptyset$, *and, for all* $q \in Q$ *and for each pair of transitions with origin in* $q$ *and labeled by the same action* $a$: $\langle q, a, \pi_1, G_1, A_1, q_1 \rangle$ *and* $\langle q, a, \pi_2, G_2, A_2, q_2 \rangle$, *the conjunction* $G_1 \wedge G_2$ *is unsatisfiable.*

*Determinising* an IOSTS $\mathcal{S}$ means computing a deterministic IOSTS $det(\mathcal{S})$ with the same traces as $\mathcal{S}$, i.e., $Traces(det(\mathcal{S})) = Traces(\mathcal{S})$. This operation is an important step of conformance test generation (cf. Section 3.5). We briefly present it below; full details can be found in [19].

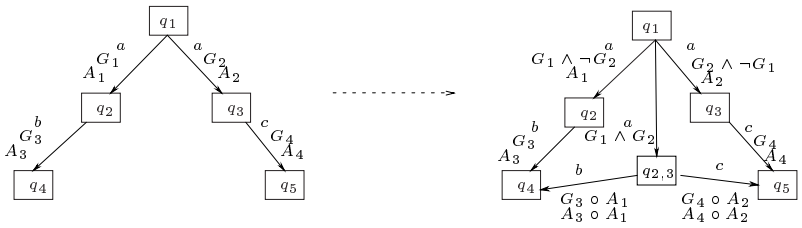
For the IOSTS $Sender^{\delta}$ (cf. Fig. 2) the IOSTS $det(Sender^{\delta})$ is shown in Fig. 3.

### Determinisation

Determinisation consists in eliminating internal actions, followed by the resolution of nondeterministic choices that remain for the observable actions. For eliminating internal actions, the idea is to compute the effect of any sequence of internal actions that leads to an input- or output-labeled transition, and to encode this effect in the guard and assignments of the last transition. Let $\tau_1$, $\tau_2$, ..., $\tau_n$ be a sequence of internal actions that leads to an input or output action $a$ (see below). The guard and assignments corresponding to $\tau_i$ are $G_i$, $A_i$, and the guard (resp. assignments) corresponding to $a$ are $G$, $A$. Then, the whole sequence is replaced by a transition with origin $l_1$, destination $l_{n+2}$, action $a$, guard $G_1 \wedge (G_2 \circ A_1) \wedge \ldots (G_n \circ A_{n-1} \circ \cdots \circ A_1) \wedge (G \circ A_n \circ A_{n-1} \circ \cdots \circ A_1)$, and assignments $A \circ A_n \circ A_{n-1} \circ \cdots \circ A_1$, where $\circ$ denotes function composition. The procedure terminates if the IOSTS does not contain cycles of internal actions, a common hypothesis in conformance testing [18], which is made consistently through this paper.



The second step consists in *postponing* the effect of a nondeterministic choice on the observable actions that follow it. This amounts to splitting, e.g., two transitions (with non-exclusive guards $G_1$, $G_2$ and assignments $A_1$, $A_2$- see below) into three: one for the case $G_1 \wedge \neg G_2$ holds, another for the case when $G_2 \wedge \neg G_1$ holds, and the last for the case $G_1 \wedge G_2$ holds. In the latter case, the choice whether to assign the variables according to $A_1$ or to $A_2$ is postponed until the observable action that follows. Thus, if $b$ is the next action, then the assignment $A_1$ should have been executed, hence, this assignment is composed with the guard and the assignment corresponding to $b$ (which produces the guard $G_3 \circ A_1$ and assignment $A_3 \circ A_1$). Similarly, if $c$ is the next action, then $A_2$ should have been executed, which produces the guard $G_4 \circ A_2$ and assignment $A_4 \circ A_2$. This procedure may not terminate (e.g., if $a = b$ and $q_4 = q_1$, the postponing goes on forever), but it does terminate for a nontrivial subclass of IOSTS defined in [19].




## 3.4 Verification of Safety Properties

The problem considered here is: given a reactive system modeled by an IOSTS $\mathcal{S}$, and a safety property $\psi$, does $\mathcal{S}$ satisfy $\psi$? We model 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 $\omega$ together with a set of dedicated locations $Violate_\omega \subseteq Q_\omega$. Its* language *is*

$$\mathcal{L}(\omega, Violate_\omega) = \{\sigma \in \Lambda_\omega^* \mid \exists s_o \in S_\omega^0, \exists q \in Violate_\omega . s_o \xrightarrow{\sigma}_\omega q\} \quad (3)$$

*We say an observer $(\omega, Violate_\omega)$ is compatible with an IOSTS $M$, or with another observer $(M, Violate_M)$, if $\omega$ is compatible with $M$. The set of observers compatible with $M$ is denoted $\Omega(M)$.*

For example, consider the following safety property of the BRP protocol: *"The sender cannot be quiescent between a request and a confirmation"*. That is, between the *REQ?* input, and the *OK!*, or *NOT_OK!*, or *DONT_KNOW!* outputs, the sender should always output something. The observer for the negation of this property is depicted in Figure 4. The set of "bad" locations is $\{Violate\}$. The self-loops "*" denote all actions that do not label other outgoing transitions.

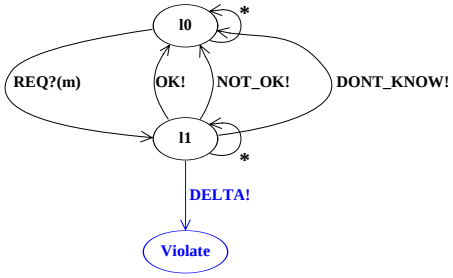



Figure 4: Sample observer.

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

For example, the IOSTS $Sender^\delta$ depicted in Figure 2 does satisfy the observer depicted in Figure 4. This can be automatically verified using abstract interpretation techniques, e.g., using the NBac tool [11], which has been incorporated in our STG (Symbolic Test Generation) tool [6]. First,

STG computes the product $Sender^{\delta}||\omega$, and then, NBac performs a symbolic reachability analysis (from the initial states) and coreachability analysis (to the violating locations) of the product. In our example, the violating locations are found unreachable, hence, the property holds. In general, the analysis might not allow to conclude, because of the over-approximations in the reachability and coreachability analyses. Note that *exact* analyses are impossible in general, because the standard state-reachability problem is undecidable for IOSTS (e.g., IOSTS subsume the class of 2-counter automata).

Finally, the parallel product of two observers can be seen as an observer as well:

**Lemma 2** *Given two compatible observers* $(\omega, Violate_{\omega})$ *and* $(\varphi, Violate_{\varphi})$ *the pair* $(\omega||\varphi, Violate_{\omega} \times Violate_{\varphi})$ *is an observer, whose language is* $\mathcal{L}(\omega||\varphi, Violate_{\omega} \times Violate_{\varphi}) = \mathcal{L}(\omega, Violate_{\omega}) \cap \mathcal{L}(\varphi, Violate_{\varphi})$.

Note that this is not the only interpretation of the parallel product as an observer; alternative choices for the violating locations are, e.g., $Violate_{\omega} \times (Q_{\varphi} \setminus Violate_{\varphi})$, which indicates the violation of the former observer, but not that of the latter; and, $(Q_{\omega} \setminus Violate_{\omega}) \times Violate_{\varphi}$, which indicates the violation of the latter, but not of the former observer. These notions are employed in Section 4 where we define test cases for detecting violations of safety properties and of conformance to a specification.

### 3.5 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 behavior of $\mathcal{I}$ can be modeled by a formal object in the same class as the specification (i.e., an IOSTS), and having a compatible interface with it. Moreover, the implementation is assumed to be *input-complete*, i.e., all its inputs are enabled all the time.

These assumptions are called *test hypothesis* in conformance testing. The central notion in conformance testing is that of *conformance relation*; we rephrase the **ioco** relation defined by Tretmans [18].

**Definition 11 (Conformance)** *For compatible IOSTS* $\mathcal{S}, \mathcal{I}$ *(cf. Def. 5),*

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

Intuitively, an implementation $\mathcal{I}$ **ioco**-conforms to its specification $\mathcal{S}$, if, after each trace of the quiescent IOSTS $\mathcal{S}^{\delta}$, the implementation only exhibits outputs and quiescences allowed by $\mathcal{S}$. 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 be accounted for in conformance testing.

**Example.** After an initial input $REQ?(-1)$, which is not accepted by the specification $Sender^{\delta}$, every trace conforms to the specification, because the conformance relation constrains only outputs.

On the other hand, assume an implementation $\mathcal{I}$ of the *Sender* process depicted in Figure 1, such that $\mathcal{I}^{\delta}$ behaves just as the IOSTS $Sender^{\delta}$ depicted in Figure 2, except for input-completeness and for the initial condition $max \geq 1$, which is omitted. Then, it is not hard to check that $REQ?(1) \cdot MSG!(f(0), true) \cdot \delta$ is a trace of $\mathcal{I}^{\delta}$. Hence, the implementation $\mathcal{I}$ does not conform to *Sender*, because $Sender^{\delta}$ does not have traces of the form $REQ? \cdot MSG! \cdot \delta$. This is because (cf. Section 3.4) $Sender^{\delta}$ satisfies the observer depicted in Fig. 4, but sequences of the form $REQ? \cdot MSG! \cdot \delta$ violate this observer.

The second part of the above example has highlighted the fact a non-conformance between implementation and specification may be indicated by the fact that the former violates a safety property that the latter satisfies. We now show how to relate conformance testing to the satisfaction of safety properties, and how to generate test cases for attempting to violate them individually or one at a time.

## 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 3.4) could not establish whether the specification satisfies the property, the test cases may also detect violations of the property by the specification as well.

All inconsistencies between implementation, specification, and property discovered in this manner are returned to the uses under the form of test verdicts. 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 any deterministic IOSTS $M$. We then show that the output-completion of any IOSTS of the form $det(\mathcal{S}^\delta)$ obtained by suspending and then determinising a given IOSTS $\mathcal{S}$ (cf. Sections 3.2, 3.3) can be seen as a *canonical tester* [4] for $\mathcal{S}$ and the **ioco** relation defined in Section 3.5. (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, proved below, that **ioco**-conformance to a specification $\mathcal{S}$ is equivalent to satisfying (a safety property described by) an observer obtained from $\Sigma^!(det(\mathcal{S}^\delta))$. Finally, by composing this observer with an arbitrary observer $(\omega, Violate_\omega)$ we obtain test cases for checking the conformance to $\mathcal{S}$ and/or the satisfaction of the property defined by $(\omega, Violate_\omega)$.

**Definition 12 (output-completion)** *Given a deterministic IOSTS $M$ with locations $L$, variables $V$, actions $\Sigma$, transitions $\mathcal{T}$, and alphabet $\Sigma = \Sigma^? \cup \Sigma^!$, the* output completion *of $M$, denoted $\Sigma^!(M)$, is an IOSTS obtained from $M$ by adding a new location $Fail_M \notin L$, and for each $l \in L$ and $a \in \Sigma^!$, a transition with origin $l$, destination $Fail_M$, action $a$, assignments $(x := x)_{x \in V}$, and guard $\bigwedge_{t=\langle l_t, a_t, \pi, G_t, A_t, l'_t\rangle \in \mathcal{T}} \neg G_t$.*

That is, any output not fireable in $M$ becomes fireable in $\Sigma^!(M)$ and leads to the new (deadlock) location $Fail_{\mathrm{M}}$. The output-completion of a deterministic IOSTS $M$ can be seen as an observer by choosing the singleton $\{Fail_{\mathrm{M}}\}$ in Definition 12 as the set of violating locations. Its language satisfies:

**Lemma 3** *For a deterministic IOSTS $M$, $\mathcal{L}(\Sigma^!(M), \{Fail_M\}) = Traces(M).\Lambda^!_M \setminus Traces(M)$.*

The following lemma says that **ioco**-conformance to a specification is basically a safety property:

**Lemma 4** *For compatible IOSTS $\mathcal{I}$, $\mathcal{S}$: $\mathcal{I}$ **ioco** $\mathcal{S}$ iff $\mathcal{I}^\delta \models (\Sigma^!(det(\mathcal{S}^\delta)), \{Fail_{det(\mathcal{S}^\delta)}\})$.*

**Proof :** By Def. 11, $\mathcal{I}$ **ioco** $\mathcal{S}$ is just $Traces(\mathcal{S}^\delta) \cdot (\Lambda_S \cup \{\delta\}) \cap Traces(\mathcal{I}^\delta) \subseteq Traces(\mathcal{S}^\delta)$, which is equivalent to $[Traces(\mathcal{S}^\delta) \cdot (\Lambda_S \cup \{\delta\}) \setminus Traces(\mathcal{S}^\delta)] \cap Traces(\mathcal{I}^\delta) = \emptyset$. Using Lemma 3, this is just $\mathcal{L}(\Sigma^!(det(\mathcal{S}^\delta)), \{Fail_{det(\mathcal{S}^\delta)}\}) \cap Traces(\mathcal{I}^\delta) = \emptyset$, which, by Def. 10, is just $\mathcal{I}^\delta \models (\Sigma^!(det(\mathcal{S}^\delta)), \{Fail_{det(\mathcal{S}^\delta)}\})$. ◇

The lemma also says that the IOSTS $\Sigma^!(det(\mathcal{S}^\delta))$ is a canonical tester for **ioco**-conformance to $\mathcal{S}$. Indeed, $\mathcal{I}^\delta \models (\Sigma^!(det(\mathcal{S}^\delta)), \{Fail_{det(\mathcal{S}^\delta)}\})$ can be interpreted as the fact that execution of $\Sigma^!(det(\mathcal{S}^\delta))$ on the implementation $\mathcal{I}$ never reaches the *Fail* location, i.e., it never leads to a "Fail" verdict; the fact that this is equivalent to $\mathcal{I}$ **ioco** $\mathcal{S}$ (as stated by Lemma 4) 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, some potential violations of some (additional) safety properties. The additional properties typically originate from the system's requirements. The observers (cf. Definition 9) employed for expressing the properties also serve as a test selection mechanism; they allow to choose among the many possible interactions between the canonical tester and an implementation.

We first note that for $M$ a deterministic IOSTS and $(\omega, Violate_\omega) \in \Omega(M)$, the IOSTS $\omega || \Sigma^!(M)$ is deterministic and compatible with $M$. It can be interpreted as an observer of $M$ by identifying its set of violating locations. Let for now this set be $\{Fail_{\mathrm{M}}\} \times Violate_\omega$, denoted by $ViolateFail_{\omega||\Sigma^!(M)}$.[2]

[2] The subscript is omitted whenever it is clear from the context.

**Lemma 5 (Violate-Fail language)**

$$\mathcal{L}(\omega||\Sigma^!(M), \mathit{ViolateFail}_{\omega||\Sigma^!(M)}) = \mathcal{L}(\omega, \mathit{Violate}_\omega) \cap [\mathit{Traces}(M) \cdot \Lambda^!_M \setminus \mathit{Traces}(M)].$$

**Proof :**

$$\begin{aligned}\mathcal{L}(\omega||\Sigma^!(M), \mathit{ViolateFail}_{\omega||\Sigma^!(M)}) &= \mathcal{L}(\omega, \mathit{Violate}_\omega) \cap \mathcal{L}(\Sigma^!(M), \{\mathit{Fail}_{\mathrm{M}}\}) \text{ (by Lemma 2)} \\ &= \mathcal{L}(\omega, \mathit{Violate}_\omega) \cap [\mathit{Traces}(M).\Lambda^!_M \setminus \mathit{Traces}(M)] \text{ (by Lemma 3)}\end{aligned}$$

◇

The IOSTS $\omega||\Sigma^!(det(\mathcal{S}^\delta))$ will play an important role and deserves a definition of its own:

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

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, by identifying the traces in which the property described by $(\omega, \mathit{Violate}_\omega)$ is violated.

**Proposition 1** *For compatible IOSTS* $\mathcal{S}$, $\mathcal{I}$,

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

**Proof :** ($\Rightarrow$)

$\mathcal{I}$ **ioco** $\mathcal{S}$ $\Rightarrow$ (cf. Lemma 4)

$\mathcal{I}^\delta \models (\Sigma^!(det(\mathcal{S}^\delta)), \{\mathit{Fail}_{det(\mathcal{S}^\delta)}\})$ $\Rightarrow$ (cf. Definition 10)

$\mathit{Traces}(\mathcal{I}^\delta) \cap \mathcal{L}(\Sigma^!(det(\mathcal{S}^\delta)), \{\mathit{Fail}_{det(\mathcal{S}^\delta)}\}) = \emptyset$ $\Rightarrow$

$\forall(\omega, \mathit{Violate}_\omega) \in \Omega(\mathcal{S}^\delta).\ \mathit{Traces}(\mathcal{I}^\delta) \cap \mathcal{L}(\omega, \mathit{Violate}_\omega) \cap \mathcal{L}(\Sigma^!(det(\mathcal{S}^\delta)), \{\mathit{Fail}_{det(\mathcal{S}^\delta)}\}) = \emptyset$

$\Rightarrow$ (cf. Definitions 10 and 13)

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

($\Leftarrow$) First, note that $(\Sigma^!(det(\mathcal{S}^\delta)), \mathit{Fail}_{det(\mathcal{S}^\delta)})$ is an observer of $\mathcal{S}^\delta$. Then,

$\forall(\omega, \mathit{Violate}_\omega) \in \Omega(\mathcal{S}^\delta).\mathcal{I}^\delta \models (\mathit{test}(\mathcal{S},\omega), \mathit{ViolateFail}_{\mathit{test}(\mathcal{S},\omega)})$ $\Rightarrow$ (cf. Def. 13 and above observation)

$\mathcal{I}^\delta \models [(\Sigma^!(det(\mathcal{S}^\delta)), \mathit{Fail}_{det(\mathcal{S}^\delta)})||(\Sigma^!(det(\mathcal{S}^\delta)), \mathit{Fail}_{det(\mathcal{S}^\delta)})]$ $\Rightarrow$

$\mathcal{I}^\delta \models (\Sigma^!(det(\mathcal{S}^\delta)), \mathit{Fail}_{det(\mathcal{S}^\delta)})$ which, by Lemma 4, is equivalent to $\mathcal{I}$ **ioco** $\mathcal{S}$. ◇

**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 $\mathit{ViolateFail}_{\mathit{test}(\mathcal{S},\omega)}$ ($= \mathit{Violate}_\omega \times \{\mathit{Fail}_{\Sigma^!(det(\mathcal{S}^\delta))}\}$), 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 |
|---|

We now consider another interpretation of the IOSTS $\omega||\Sigma^!(M)$, which leads to another test verdict. By choosing the violating locations to be $(Q_\omega \setminus \mathit{Violate}_\omega) \times \{\mathit{Fail}_{\mathrm{M}}\}$ (rather than $\mathit{Violate}_\omega \times \{\mathit{Fail}_{\mathrm{M}}\}$), we obtain a different observer. We denote by $\mathit{Fail}_{\omega||\Sigma^!(M)}$ the set $(Q_\omega \setminus \mathit{Violate}_\omega) \times \{\mathit{Fail}_{\mathrm{M}}\}$. The subscript is omitted whenever is is clear from the context. The new observer satisfies:

**Lemma 6 (Fail language)** $\mathcal{L}(\omega||\Sigma^!(M), \mathit{Fail}_{\omega||\Sigma^!(M)}) \subseteq \mathit{Traces}(M).\Lambda^!_M \setminus \mathit{Traces}(M)$.

**Proof:** By Lemma 2, $\mathcal{L}(\omega||\Sigma^!(M), \mathit{Fail}_{\omega||\Sigma^!(M)}) = \mathcal{L}(\Sigma^!(M), \mathit{Fail}_{\mathrm{M}}) \cap \mathcal{L}(\omega, (Q_\omega \setminus \mathit{Violate}_\omega)) \subseteq \mathcal{L}(\Sigma^!(M), \mathit{Fail}_{\mathrm{M}})$; and, by Lemma 3, $\mathcal{L}(\Sigma^!(M), \mathit{Fail}_{\mathrm{M}}) = \mathit{Traces}(M) \cdot \Lambda^!_{\mathrm{M}} \setminus \mathit{Traces}(M)$. ◇

**Proposition 2** *For compatible IOSTS* $\mathcal{I}$, $\mathcal{S}$ *and* $(\omega, \mathit{Violate}_\omega) \in \Omega(S^\delta)$,

$\mathcal{I}$ **ioco** $\mathcal{S} \Rightarrow \mathcal{I}^\delta \models (\mathit{test}(\mathcal{S}, \omega), \mathit{Fail}_{\mathit{test}(\mathcal{S},\omega)})$.

**Proof:** By Def. 11, $\mathcal{I}$ **ioco** $\mathcal{S}$ is $\mathit{Traces}(\mathcal{S}^\delta) \cdot (\Lambda_{\mathcal{S}} \cup \{\delta\}) \cap \mathit{Traces}(\mathcal{I}^\delta) \subseteq \mathit{Traces}(\mathcal{S}^\delta)$, which is equivalent to $[\mathit{Traces}(\mathcal{S}^\delta) \cdot (\Lambda_{\mathcal{S}} \cup \{\delta\}) \setminus \mathit{Traces}(\mathcal{S}^\delta)] \cap \mathit{Traces}(\mathcal{I}^\delta) = \emptyset$. Using Lemma 6 and Definition 13, this implies $\mathcal{L}(\mathit{test}(\mathcal{S}, \omega), \mathit{Fail}_{\mathit{test}(\mathcal{S},\omega)}) \cap \mathit{Traces}(\mathcal{I}^\delta) = \emptyset$, which, by Def. 10, is $\mathcal{I}^\delta \models (\mathit{test}(\mathcal{S}, \omega), \mathit{Fail}_{\mathit{test}(\mathcal{S},\omega)})$. ◇

Proposition 2 says that when $\mathit{test}(\mathcal{S}, \omega)$ enters a location in the set $\mathit{Fail}_{\mathit{test}(\mathcal{S},\omega)}$ $(= (Q_\omega \setminus \mathit{Violate}_\omega) \times \{\mathit{Fail}_{\Sigma^!(\mathit{det}(\mathcal{S}^\delta))}\})$ when executed on an implementation $\mathcal{I}$, then $\mathcal{I}$ violates conformance to $\mathcal{S}$. The property $\omega$ 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 $\mathit{Violate}_\omega \times Q_{\mathrm{M}}$. We denote this set by $\mathit{Violate}_{\omega||\Sigma^!(M)}$, and omit the subscript whenever it is clear from the context. The language of this observer satisfies:

**Lemma 7 (Violate language)** $\mathcal{L}(\omega||\Sigma^!(M), \mathit{Violate}_{\omega||\Sigma^!(M)}) = \mathcal{L}(\omega, \mathit{Violate}_\omega) \cap \mathit{Traces}(M)$.

**Proof :** $\mathcal{L}(M, Q_{\mathrm{M}}) = \mathit{Traces}(M)$ and by Lemma 2, $\mathcal{L}(\omega||\Sigma^!(M), \mathit{Violate}_{\omega||\Sigma^!(M)}) = \mathcal{L}(M, Q_{\mathrm{M}}) \cap \mathcal{L}(\omega, \mathit{Violate}_\omega)$. ◇

**Proposition 3** *For compatible IOSTS* $\mathcal{I}$, $\mathcal{S}$ *and observer* $(\omega, \mathit{Violate}_\omega) \in \Omega(S^\delta)$,

*(a)* $\mathcal{S}^\delta \models (\omega, \mathit{Violate}_\omega) \Rightarrow \mathcal{I}^\delta \models (\mathit{test}(\mathcal{S}, \omega), \mathit{Violate}_{\mathit{test}(\mathcal{S},\omega)})$

*(b)* $\mathcal{I}^\delta \models (\omega, \mathit{Violate}_\omega) \Rightarrow \mathcal{I}^\delta \models (\mathit{test}(\mathcal{S}, \omega), \mathit{Violate}_{\mathit{test}(\mathcal{S},\omega)})$

**Proof :** (a) By Def. 10, $\mathcal{S}^\delta \models (\omega, \mathit{Violate}_\omega)$ is $\mathit{Traces}(\mathcal{S}^\delta) \cap \mathcal{L}(\omega, \mathit{Violate}_\omega) = \emptyset$, which implies $\mathit{Traces}(\mathcal{I}^\delta) \cap \mathit{Traces}(\mathit{det}(\mathcal{S}^\delta)) \cap \mathcal{L}(\omega, \mathit{Violate}_\omega) = \emptyset$; by Lemma 7 we get $\mathcal{I}^\delta \models (\mathit{test}(\mathcal{S}, \omega), \mathit{Violate}_{\mathit{test}(\mathcal{S},\omega)})$. (b) is proved in a similar manner (replace the first two occurrences of $\mathcal{S}^\delta$ by $\mathcal{I}^\delta$ in the above proof). ◇

Proposition 3 says that when $\mathit{test}(\mathcal{S}, \omega)$ enters a location in $\mathit{Violate}_{\mathit{test}(\mathcal{S},\omega)}$ $(= \mathit{Violate}_\omega \times Q_{\mathit{det}(\mathcal{S}^\delta)})$ when executed on an implementation $\mathcal{I}$, then a violation of the property by both specification and implementation is detected. In this case, 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 $\mathit{test}(\mathcal{S}, \omega)$ is *correct*, in the sense that the verdicts **ViolateFail** (resp. **Fail**) correctly detect the violation of the property and of the conformance (resp. of the conformance only) by the implementation. This holds irrespectively of whether the specification satisfies the property or not (which may not always be possible to establish). The test case execution may detect violations of the property by the specification (using the **Violate** verdict, whose correctness is proved as well).

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 3.5, this does not compromise conformance and, of course, the specification does not violate the property on such a trace because it does not contain it. However, such traces are excluded from the generated test cases by construction.

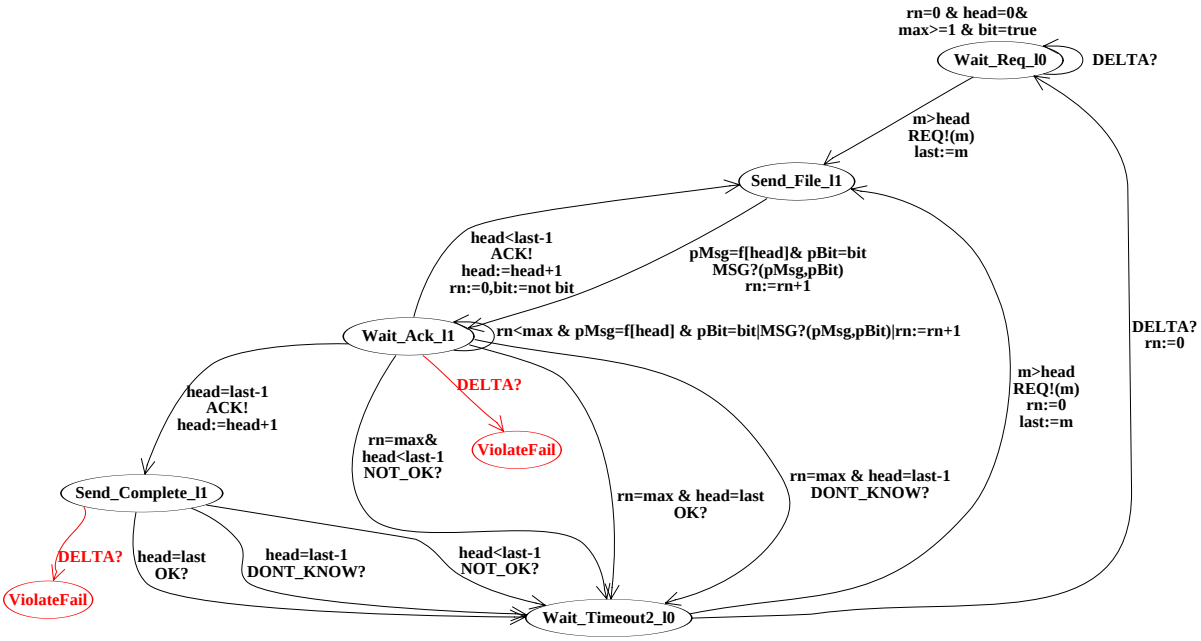



Figure 5: Test Case obtained from Sender (Figure 1) and Observer (Figure 4).

**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). Finally, the result is automatically analysed using the NBac tool [11] for statically detecting and eliminating transitions that cannot lead to the **ViolateFail** verdict any more. This operation is described in Section 5.

For the *Sender* IOSTS depicted in Figure 1 and the observer depicted in Figure 4, the corresponding test case is depicted in Figure 5. For better readability, the transitions leading to **Fail** have not been represented, and the location corresponding to the **ViolateFail** verdict has been duplicated.

For example, an implementation performing a sequence of the form $REQ? \cdot MSG! \cdot \delta$ leads to **ViolateFail**; it has already been pointed out at the end of Section 3.5 that such an implementation exhibiting violates both the property described by the observer $(\omega, Violate_\omega)$ (Fig 4) and the conformance to the specification *Sender* (Fig 1). Finally, note that the test case does not have a **Violate** verdict, because (cf. Section 3.4) $Sender^\delta$ does satisfy the property described by $(\omega, Violate_\omega)$.

## 5 Pruning a Test Case

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, i.e., from which violations of the property by the implementation cannot be detected. Hence, this operation constitutes an optimisation, in the sense that it allows to terminate test execution earlier, whenever it becomes clear that no future violations of the property may be detected. We show that the operation preserves test case correctness.

The violation of a property - described as an observer $(\omega, \mathit{Violate}_\omega)$ - by an implementation is materialised by reaching the *ViolateFail* and *Violate* sets of locations in the IOSTS $\mathit{test}(\mathcal{S}, \omega)$(cf. Section 4). We only deal here with the reachability of *ViolateFail*; the reachability of *Violate* can be dealt with in a similar manner. To simplify the presentation, we assume that there is only one *ViolateFail* location.

Then, (cf. Definition 3) we are interested in states that are not coreachable for the location *ViolateFail*. It should be quite clear that computing the exact coreachable set of a location is impossible in general, however, there exist techniques to compute over-approximations of these sets. We here use one such technique based on abstract interpretation and implemented in the NBac tool [11]. 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 and constants 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 set of states at location $l$ that are coreachable for the location $q$.

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

- *first, the IOSTS $\mathit{mirror}(\mathit{test}(\mathcal{S}, \omega))$ 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 $\Sigma^? = \Sigma^!_{\mathcal{S}} \cup \{\delta\}$,*
- *then, for each location $l \in L$, a symbolic coreachable state $\langle l, \varphi_{l\to \mathit{ViolateFail}}\rangle$ for the location ViolateFail is computed,*
- *next, for each location $l \in l$ of the IOSTS, and each transition $t \in \mathcal{T}$ with origin $l$, guard $G$, and label $a$,*
  - *if $a \in \Sigma^!$ then*
    - *if the conjunction $G \wedge \varphi_{l\to \mathit{ViolateFail}}$ is unsatisfiable, then $t$ is removed from the set $\mathcal{T}$,*
    - *otherwise, the guard of $t$ becomes $G \wedge \varphi_{l\to \mathit{ViolateFail}}$*
  - *if $a \in \Sigma^?$, then*
    - *the guard of $t$ becomes $G \wedge \varphi_{l\to \mathit{ViolateFail}}$*
    - *a new transition is added to $\mathcal{T}$, with origin $l$, destination Inconc, action $a$, guard $G \wedge \neg\varphi_{l\to \mathit{ViolateFail}}$, and identity assignments. Here, $\mathit{Inconc} \notin L$ is a new location.*

**Interpretation.** The pruning operation consists essentially in removing some transitions (and changing the destination of others) by which the *ViolateFail* location cannot be reached any more. If such a "useless" transition is labeled 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 labeled by *inputs*, by which the *ViolateFail* location cannot be reached any more, are reoriented to a new location. The new location is called *Inconc*, and it may be interpreted as a verdict:

| **Inconc**: future simultaneous violations of the property and of conformance cannot be detected |
|---|

By pruning the test case some more to eliminate states that are not coreachable to *Violate*, the **Inconc** verdict can be strengthened to "future violations of the property cannot be detected".

In the rest of this section we prove that the pruning operation preserves the correctness of test cases. The following lemma allows to define a sequence of transitions that "matches" a trace.

**Lemma 8 (support of a trace)** *If* $\sigma: \quad \alpha_1\alpha_2\cdots\alpha_n$ *is a trace of a deterministic IOSTS* $\mathcal{S}$*, then, there exist states* $s_i$ $(i = 1, n)$ *of* $\mathcal{S}$ *such that* $s_i \xrightarrow{\alpha_i} s_{i+1}$ *for* $i = 1, \ldots, n$*, and there exists a sequence of transitions* $t_i = \langle q_i, a_i, G_i, \pi_i, A_i, q_{i+1}\rangle$ $(i = 1, \ldots n)$ *of the IOSTS such that, for* $i = 1 \ldots n$*,*

- $s_i = \langle q_i, v_i\rangle$*, for some valuations* $v_i$ *of the variables and constants of the IOSTS,*
- $\alpha_i = \langle a_i, w_i\rangle$*, for some valuation* $w_i$ *of the parameters* $\pi_i$*,*
- *the valuations of variables, symbolic constants, and parameters defined by* $v_i$ *and* $w_i$ *satisfy the guard* $G_i$*, i.e.,* $\langle v_i, w_i\rangle \models G_i$*.*

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

**Proof :** Follows directly the semantics of IOSTS and the definition of deterministic IOSTS (Def. 8).$\diamond$

The next lemma gives necessary conditions under which a sequence of transitions supports a trace.

**Lemma 9 (support and symbolic coreachability)** *Let* $\sigma: \quad \alpha_1\alpha_2\cdots\alpha_n$ *be a trace of a deterministic IOSTS* $\mathcal{S}$*; let* $t_i = \langle q_i, a_i, G_i, \pi_i, A_i, q_{i+1}\rangle$ $(i = 1, \ldots, n)$ *be a sequence of transitions that supports* $\sigma$*; and let* $\langle q_i, \varphi_{q_i \to q_{n+1}}\rangle$ $(i = 1, \ldots n)$ *be symbolic coreachability sets for the location* $q_{n+1}$*. Then, for all* $i = 1, \ldots n$*, the formula* $G_i \wedge \varphi_{q_i \to q_{n+1}}$ *is satisfiable. Moreover, the sequence of modified transitions* $t'_i = \langle q_i, a_i, G_i \wedge \varphi_{q_i \to q_{n+1}}, \pi_i, A_i, q_{i+1}\rangle$ $(i = 1 \ldots n)$ *is a support for the trace* $\sigma$ *as well.*

**Proof :** For $i = 1, \ldots, n$, let $\alpha_i = \langle a_i, w_i\rangle$. By Lemma 8, there exist states $s_i = \langle q_i, v_i\rangle$ such that (1) $\langle v_i, w_i\rangle \models G_i$. Moreover, by hypothesis, $\langle q_i, \varphi_{q_i \to q_{n+1}}\rangle$ is a symbolic coreachability set for $q_{n+1}$, and it is not hard to see that $s_i \xrightarrow{\alpha_i \cdots \alpha_n} q_{n+1}$; hence, by Definition 14, (2) $v_i \models \varphi_{q_i \to q_{n+1}}$. Now, (1) and (2) imply that $\langle v_i, w_i\rangle \models G_i \wedge \varphi_{q_i \to q_{n+1}}$, and we have proved that $G_i \wedge \varphi_{q_i \to q_{n+1}}$ is satisfiable.

Finally, since $s_i = \langle q_i, v_i\rangle$, $\alpha_i = \langle a_i, w_i\rangle$, and $\langle v_i, w_i\rangle \models G_i \wedge \varphi_{q_i \to q_{n+1}}$ for $i = 1, \ldots, n$, the sequence of transitions $t'_i = \langle q_i, a_i, G_i \wedge \varphi_{q_i \to q_{n+1}}, \pi_i, A_i, q_{i+1}\rangle$ $(i = 1 \ldots n)$ satisfies all the conditions (cf. Lemma 8) for being a support for the trace $\sigma: \quad \alpha_1\alpha_2\cdots\alpha_n$, and the proof is done. $\diamond$

The following facts can be proved about the languages of $prune(\mathcal{S}, \omega)$:

**Lemma 10 (languages of $prune(\mathcal{S}, \omega)$ and languages of $test(\mathcal{S}, \omega)$)**

*(1)* $\mathcal{L}(prune(\mathcal{S}, \omega), ViolateFail) = \mathcal{L}(test(\mathcal{S}, \omega), ViolateFail)$

*(2)* $\mathcal{L}(prune(\mathcal{S}, \omega), Fail) \subseteq \mathcal{L}(test(\mathcal{S}, \omega), Fail)$

*(3)* $\mathcal{L}(prune(\mathcal{S}, \omega), Violate) \subseteq \mathcal{L}(test(\mathcal{S}, \omega), Violate)$.

**Proof :** (sketch)

• $\mathcal{L}(prune(\mathcal{S}, \omega), ViolateFail) \subseteq \mathcal{L}(test(\mathcal{S}, \omega), ViolateFail)$: let $\sigma \in \mathcal{L}(prune(\mathcal{S}, \omega), ViolateFail)$. Then, by Lemma 8, there exists a sequence of transitions of $prune(\mathcal{S}, \omega)$ that supports $\sigma$. By construction of $prune(\mathcal{S}, \omega)$ (cf. Definition 15), this sequence is of the form $t'_i = \langle q_i, a_i, G_i \wedge \varphi_{q_i \to q_{n+1}}, \pi_i, A_i, q_{i+1}\rangle$ $(i = 1 \ldots n)$, such that, for all $i = 1 \ldots n$, $\langle q_i, a_i, G_i, \pi_i, A_i, q_{i+1}\rangle$ is a transition of $test(\mathcal{S}, \omega)$, and $q_{n+1} = ViolateFail$. Indeed, all locations $q_i$ of the sequence must be locations of $test(\mathcal{S}, \omega)$ - the only location of $prune(\mathcal{S}, \omega)$ that is not a location of $test(\mathcal{S}, \omega)$ is *Inconc*, which is a deadlock, i.e., *ViolateFail* is not reachable from it, which contradicts $q_{n+1} = ViolateFail$; and $prune(\mathcal{S}, \omega)$ is obtained by strengthening the guards of transitions of $test(\mathcal{S}, \omega)$ with the corresponding predicates $\varphi_{q_i \to q_{n+1}}$. Then, the sequence of transitions of $test(\mathcal{S}, \omega)$: $t_i = \langle q_i, a_i, G_i, \pi_i, A_i, q_{i+1}\rangle$ $(i = 1 \ldots n)$ satisfies all the conditions of Lemma 8, i.e., it supports the sequence $\sigma$, which implies $\sigma \in \mathcal{L}(test(\mathcal{S}, \omega), ViolateFail)$.

• $\mathcal{L}(prune(\mathcal{S}, \omega), ViolateFail) \supseteq \mathcal{L}(test(\mathcal{S}, \omega), ViolateFail)$: Let $\sigma \in \mathcal{L}(test(\mathcal{S}, \omega), ViolateFail)$. Then, by Lemma 8, there exists a sequence of transitions of $test(\mathcal{S}, \omega)$: $t_i = \langle q_i, a_i, G_i, \pi_i, A_i, q_{i+1}\rangle$

$(i = 1 \ldots n)$ that supports the sequence $\sigma$, and $q_{n+1} = \mathit{ViolateFail}$. By Lemma 9, the sequence of modified transitions $t'_i = \langle q_i, a_i, G_i \wedge \varphi_{q_i \to q_{n+1}}, \pi_i, A_i, q_{i+1} \rangle$ $(i = 1 \ldots n)$ supports $\sigma$ as well, and, by construction, all the transitions of the sequence $t_i$ $(i = 1 \ldots n)$ also belong to $\mathit{prune}(\mathcal{S}, \omega)$. Indeed (cf. Definition 15), the only transitions of $\mathit{test}(\mathcal{S}, \omega)$ that are removed, or whose destination is changed to *Inconc* in $\mathit{prune}(\mathcal{S}, \omega)$, are those whose guard $G_i$ is inconsistent with $\varphi_{q_i \to q_{n+1}}$, and Lemma 9 guarantees that such transitions cannot belong to the sequence $t_i$ $(i = 1 \ldots n)$. Hence, $\sigma$ is supported by the sequence $t'_i$ $(i = 1 \ldots n)$, in $\mathit{prune}(\mathcal{S}, \omega)$, i.e., $\sigma \in \mathcal{L}(\mathit{prune}(\mathcal{S}, \omega), \mathit{ViolateFail})$.

• The proofs of the inclusions (2) and (3) are similar to the proof of the $\subseteq$ inclusion of (1), by considering the *Fail* and *Violate* locations, respectively, instead of *ViolateFail*. ◇.

**Corollary 1 (languages of $\mathit{prune}(\mathcal{S}, \omega)$)**

*(1)* $\mathcal{L}(\mathit{prune}(\mathcal{S}, \omega), \mathit{ViolateFail}) = \mathcal{L}(\omega, \mathit{Violate}_\omega) \cap [Traces(\mathcal{S}^\delta) \cdot (\Lambda^!_{\mathcal{S}} \cup \{\delta\}) \setminus Traces(\mathcal{S}^\delta)]$

*(2)* $\mathcal{L}(\mathit{prune}(\mathcal{S}, \omega), \mathit{Fail}) \subseteq Traces(\mathcal{S}^\delta) \cdot (\Lambda^!_{\mathcal{S}} \cup \{\delta\}) \setminus Traces(\mathcal{S}^\delta)$

*(3)* $\mathcal{L}(\mathit{prune}(\mathcal{S}, \omega), \mathit{Violate}) \subseteq \mathcal{L}(\omega, \mathit{Violate}_\omega) \cap Traces(\mathcal{S}^\delta)$.

Our final result is then:

**Theorem 1 (correctness of $\mathit{prune}(\mathcal{S}, \omega)$)** *The test case $\mathit{prune}(\mathcal{S}, \omega)$ is correct, i.e., Propositions 1, 2 and 3 (cf. Section 4) that characterise correctness still hold when $\mathit{test}(\mathcal{S}, \omega)$ is replaced with $\mathit{prune}(\mathcal{S}, \omega)$.*

**Proof :**(Sketch) For Propositions 1, 2, it is enough to note that items (1) and (2) of Corollary 1 state exactly the same properties about the languages of $\mathit{prune}(\mathcal{S}, \omega)$ as Lemmas 5, 6 about the languages of $\mathit{test}(\mathcal{S}, \omega)$, respectively. Moreover, Lemmas 5, 6 are the only facts about $\mathit{test}(\mathcal{S}, \omega)$ employed in the proofs of Propositions 1, 2 (respectively). Hence, these propositions hold for $\mathit{prune}(\mathcal{S}, \omega)$ as well.

For Proposition 3, we note that item (3) of Corollary 1 for $\mathit{prune}(\mathcal{S}, \omega)$ is weaker that the corresponding Lemma 7 for $\mathit{test}(\mathcal{S}, \omega)$: only a language inclusion holds (instead of an equality), but, trivially, the inclusion is enough for establishing that Proposition 3 holds for $\mathit{prune}(\mathcal{S}, \omega)$. ◇

## 6 Conclusion and Related Work

A system may be viewed at several levels of abstraction: high-level *properties*, operational *specification*, and final, black-box *implementation*. In the proposed framework, these views are modeled using Input-Output Symbolic Transition Systems (IOSTS), which are labeled transition systems that operate on symbolic variables and communicate with the environment through input and output actions carrying parameters. (The IOSTS for the black-box implementation is unknown, only its existence is assumed.) A conformance relation links the implementation and the specification, and a satisfaction relation links them both with the required 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, then 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 step did not conclude (i.e., it did not allow to prove or to disprove a property), then 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 the BRP protocol [9].

**Related Work.** In addition to the references given in Section 1 we list a few more related works.

In [7] an approach to generate 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 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 [8], start from a specification $S$ and a property $P$ (expressed in temporal logic) assumed to be satisfied by $S$, and use the ability of model checkers to construct counter-examples for $\neg P$ on $S$. These counter-examples can be interpreted as *witnesses* of $P$ on $S$. The paper [10] exploits this idea to describing coverage criteria in temporal logic and generating test cases using model-checking.

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

A different approach for combining model checking and black-box testing is black-box checking [14]. Under some rather strict assumptions on the implementation (among others: 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 (the size of which is exponential in $n$) for checking properties expressed by Büchi automata. The approach mixes interesting concepts from game theory and learning theory.

## 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] 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.

[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] J.C. Fernandez, L. Mounier, and C. Pachon. Property-oriented test generation. In *Formal Aspects of Software Testing Workshop (FATES'03)*, number 2391 in LNCS, 2003.

[8] A. Gargantini and C.L. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC/SIGSOFT FSE*, pages 146–162, 1999.

[9] 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.

[10] 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.

[11] B. Jeannet. Dynamic partitioning in linear relation analysis. application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, 2003.

[12] 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.

[13] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 3(2), 1999.

[14] D. Peled, M. Vardi, and M. Yannakakis. Black-box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225 – 246, 2001 2001.

[15] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *International Conference on Integrating Formal Methods (IFM'00)*, number 1945 in LNCS, pages 338–357, 2000.

[16] V. Rusu, H. Marchand, V. Tschaen, T. Jeron, and B. Jeannet. From safety verifcation to safety testing. In *Intl. Conf. on Testing of Communicating Systems (TestCom04)*, number 2978 in LNCS, 2004.

[17] 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.

[18] J. Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99*, number 1664 in LNCS, pages 46–65, 1999.

[19] E. Zinovieva. *Symbolic Test Generation for Reactive Systems*. PhD thesis, University of Rennes I, to be defended in November 2004.