An Approach to Symbolic Test Generation

Vlad Rusu, Lydie du Bousquet, and Thierry Jéron
{rusu|jeron|ldubousq}@irisa.fr *

IRISA/INRIA, Rennes, France

Abstract. Test generation is a program-synthesis problem: starting from the formal specification of a system under test, and from a test purpose describing a set of behaviours to be tested, compute a reactive program that observes an implementation of the system to detect non-conformant behaviour, while trying to control it towards satisfying the test purpose. In this paper we describe an approach for generating symbolic test cases, in the form of input-output automata with variables and parameters.

1 Introduction

It is widely recognized that testing is an essential component of the full lifecycle of software systems. Among the many different testing techniques, *conformance testing* [11] is one of the most rigorous. The usual theoretical approach [5, 16] is to consider a formal specification of the intended behaviour of the Implementation Under Test (IUT). It allows to define the notion of *conformance relation*, which defines the correct implementations with respect to the specification. It also allows to formally define *test cases*, their execution on an IUT, and the notion of *verdict* associated to the execution. However, the process of writing test cases for large specifications is complicated, error-prone and expensive. For example, the paper [12] identifies various errors in about 15% of the test cases from a commercially available test suite for the SSCOP protocol [21]. Another difficulty comes from the black-box nature of the implementation, whose behaviour is only observable and controlable at the interfaces. In this context, a formal framework is a prerequisite for giving precise and consistent meanings to test cases.

During the last decade, testing theories and algorithms for the automatic generation of tests have been developed from specifications modelled by variants of the Labeled Transition System model (LTS). Some efficient algorithms are based on adaptations of on-the-fly model-checking algorithms [13]. Academic tools such as TorX [1], TGV [6] and industrial tools such as TestComposer (Verilog) already exist, which implement these algorithms and produce correct test cases in a formal framework. However, these theories and tools do not explicitly take into account the program *data*, as the underlying model of LTS implies that values of variables are expanded during state exploration. This may result in the classical state-explosion problem, but also has the effect of obtaining test

* The specifications and proofs for the example treated in the paper can be found at http://www.irisa.fr/pampa/perso/rusu/IFM00/.

cases where all variables are instantiated. This is in contradiction with industrial practice, where test cases (written, for example, in the TTCN language [11]) are real programs with parameters and variables. Generating such test cases requires new models and techniques. The models should explicitly include parameters and variables, and the techniques should treat them symbolically by combining model checking with constraint propagation, static analysis, and theorem proving. In this paper, we present some steps towards such an approach.

The rest of the paper is organized as follows. In Section 2 we define an extension of the LTS model (called IOSTS) to include parameters and variables. In Section 3 we define subclasses of the IOSTS model that are used for specifications, test purposes and test cases. We adapt a conformance relation from [16] to the model of IOSTS. In Section 4 we define a notion of correctness of test cases with respect to specifications and test purposes. In Section 5 we describe how to generate test cases by successively computing a synchronous product between a specification and a test purpose, eliminating internal actions and nondeterminism, and selecting the behaviours that are accepted by the test purpose. (Currently, the method works only for specifications for which internal actions and non-determinism can be eliminated using a given set of heuristics.) We also prove the correctness of the generated test cases. In Section 6 we show how to simplify the test case using static analysis and theorem proving to remove irrelevant control and data. The approach is demonstrated on a simple example: the BRP protocol [8].

2 Model: Input-Output Symbolic Transition Systems

We define a model of extended transition systems called Input-Output Symbolic Transition Systems (IOSTS), inspired from I/O automata [15] and CSP [10]. The IOSTS model was designed to be easily translatable into the input languages of tools such as the PVS theorem prover [17] and the HyTech model checker [9].

Syntax. Let D be a set of typed data. We denote by *type*(d) the type of element $d \in D$, by $\mathcal{E}(D)$ the set of type-correct expressions on the data D, and by $\mathcal{B}(D)$ the subset of boolean expressions.

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* P *of* parameters *and a set* M *of* messages*,*
- $\Theta \in \mathcal{B}(V \cup P)$ *is the* initial condition*,*
- Q *is a nonempty, finite set of* locations*,*
- $q^0 \in Q$ *is the* initial location*,*
- Σ *is a nonempty, finite* alphabet*, which is the disjoint union of a set* Σ^i *of* input actions*, a set* Σ^o *of* output actions*, and a set* Σ^{int} *of* internal actions*. For each action* $a \in \Sigma^i \cup \Sigma^o \cup \Sigma^{int}$*, there is a (possibly empty) tuple of types* $sig(a) = \langle \vartheta_1, \ldots, \vartheta_k \rangle$*, called the* signature *of the action (the signature of an internal action* $a \in \Sigma^{int}$ *is the empty tuple),*

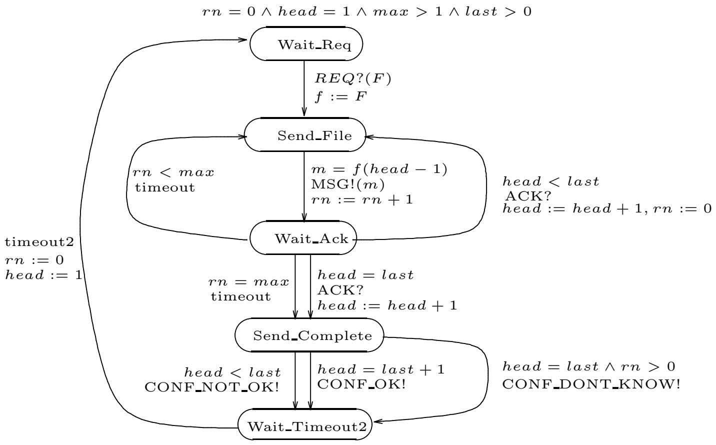

Fig. 1. Example of IOSTS: the Sender of the BRP Protocol.

– $\mathcal{T}$ *is a set of* transitions. *Each transition consists 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 of messages $\mu = \langle m_1, \dots, m_k\rangle$ such that if $sig(a) = \langle \vartheta_1 \dots \vartheta_k\rangle$ is the signature of action a, then, for all $i \in [1,k]$, $type(m_i) = \vartheta_i$,*
  - *a boolean expression $G \in \mathcal{B}(V \cup P \cup \mu)$, called the* guard *of the transition,*
  - *a set of expressions A, called the* assignments *of the transition, such that for each variable $x \in V$ there is exactly one assignment in A, of the form $x := A^x$, where A^x is an expression on $V \cup P \cup \mu$,*
  - *a location $q' \in Q$ called the* destination *of the transition.*

Figure 1 illustrates an example of IOSTS, which has variables f, rn, $head$, parameters max and $last$, and messages F and m. We assume that rn, $head$, max, and $last$ are natural numbers (with $max > 1, last > 0$), variable f and message F are functions from natural numbers to some unintepreted type (which is also the type of message m), and for all $i \in \mathbb{N}, F(i) \neq F(i+1)$.

Consider for example the transition with origin *Send_File* and destination *Wait_Ack*. It has the action MSG with message m, guard $m = f(head - 1)$, and assignment $rn := rn + 1$, $f := f$, $head := head$. In the graphical representation, the symbol *?* (resp. *!*) denotes an input (resp. an output) action, and the assignments without effect (such as $f := f$) are omitted.

Semantics. For $D' \subseteq D$ a subset of the data, we denote by $Val(D')$ the set of (type-consistent) valuations of D', which associate to each element of $d \in D'$ a value of type $type(d)$. A *state* is a pair $\langle l, v\rangle$ where $l \in Q$ is a location and $v \in Val(V \cup P)$ is a valuation for the variables and the parameters. We denote the set of states by S. An *initial state* is a state $\langle l_0, v_0\rangle$ such that $l_0 = q^0$ (the initial location) and $v_0 \models \Theta$. We denote by S^0 the set of initial states. A *valued input* (respectively a *valued output*) is a tuple consisting of an input action $a \in \Sigma^i$ (resp. an output action $a \in \Sigma^o$) and a tuple of values of the types given by the action's signature. That is, if $a \in \Sigma^i$ and $sig(a) = \langle \vartheta_1, \dots \vartheta_l \rangle$, then valued inputs are of the form $\langle a, v_1, \dots, v_l \rangle$ where for all $i \in [1, l]$, v_i is a value of type ϑ_i (and similarly for valued outputs). We denote by Υ (resp. Ω) the set of valued inputs (resp. of valued outputs.)

Let $D', D'' \subseteq D$ denote two disjoint subsets of D. For $v \in Val(D')$ and $w \in Val(D'')$, we denote by $v \cdot w$ the valuation of $D' \cup D''$ such that $v \cdot w(d) = v(d)$ if $d \in D'$ and $v \cdot w(d) = w(d)$ if $d \in D''$. Given an expression $e \in \mathcal{E}(D)$ and a valuation $v \in Val(D)$, we denote by $v(e)$ the value obtained by replacing in e each element $d \in D$ by its value $v(d)$. The semantics of an IOSTS is given by a labeled transition system with set of states S, initial states S^0, labels $\Upsilon \cup \Omega \cup \Sigma^{int}$, and the transition relation $\longrightarrow \subseteq S \times (\Upsilon \cup \Omega \cup \Sigma^{int}) \times S$, which is the smallest relation defined by the following rule:

$$\frac{\begin{array}{l}\langle q, a, \mu, G, A, q' \rangle \in \mathcal{T} \\ v, v' \in Val(V \cup P),\ w \in Val(\mu),\ v \cdot w(G) = true \\ \forall x \in V : v'(x) = v \cdot w(A^x) \quad \forall x \in P : v'(x) = v(x)\end{array}}{< q, v > \xrightarrow{<a,w>} < q', v' >}$$

The relation $\longrightarrow$ induces the relation $\Longrightarrow \subseteq S \times (\Omega \cup \Upsilon)^* \times S$ by dropping all internal actions:

- $s \overset{\epsilon}{\Longrightarrow} s' \overset{def}{=} \exists \tau_1, \dots \tau_n \in \Sigma^{int}, \exists s_1, \dots s_{n-1} \in S, s \xrightarrow{\tau_1} s_1 \cdots s_{n-1} \xrightarrow{\tau_n} s'$,
- for $\alpha \in \Omega \cup \Upsilon$, $s \overset{\alpha}{\Longrightarrow} s' \overset{def}{=} \exists s_1, s_2 \in S, s \overset{\epsilon}{\Longrightarrow} s_1 \xrightarrow{\alpha} s_2 \overset{\epsilon}{\Longrightarrow} s'$
- for $\sigma = \alpha_1 \cdots \alpha_n \in (\Omega \cup \Upsilon)^n$ with $n > 1$, $s \overset{\sigma}{\Longrightarrow} s' \overset{def}{=} \exists s_1, \dots s_{n-1} \in S$, $s \overset{\alpha_1}{\Longrightarrow} s_1 \cdots s_{n-1} \overset{\alpha_n}{\Longrightarrow} s'$.

For an IOSTS $\mathcal{S}$ we denote by *traces*$(\mathcal{S})$ the set $\{\sigma \in (\Omega \cup \Upsilon)^* | \exists s_0 \in S^0, \exists s \in S, s_0 \overset{\sigma}{\Longrightarrow} s\}$. For $\sigma \in (\Omega \cup \Upsilon)^*$, we denote by $\mathcal{S}$ *after* σ the set of states $\{s \in S | \exists s_0 \in S^0, s_0 \overset{\sigma}{\Longrightarrow} s\}$. For $S' \subseteq S$ a set of states, we denote by $out(S')$ the set of valued outputs that can be "observed" in states $s' \in S'$ possibly after internal actions, that is, $out(S') = \{\alpha \in \Omega | \exists s' \in S', \exists s \in S, s' \overset{\alpha}{\Longrightarrow} s\}$. For a set of traces L, we denote by *pref*(L) the set of strict prefixes of sequences in L.

Subclasses of IOSTS. Let $\mathcal{S}$ be an IOSTS, P its set of parameters, and $\pi \in Val(P)$ a valuation of the parameters. By replacing each parameter $p \in P$ by its value $\pi(p)$, we obtain an IOSTS denoted $\mathcal{S}(\pi)$, called an *instance* of $\mathcal{S}$. In this case, we also say that $\mathcal{S}(\pi)$ is *instantiated.* An instantiated IOSTS is *initialized* if the initial condition Θ assigns to each variable $v \in V$ exactly one value. An arbitrary IOSTS is initialized if all its instances are initialized. An IOSTS $\mathcal{S}$ is *deterministic* if for all $s \in S$: $card(\cup_{\tau \in \Sigma^{int}}\{s' \in S | s \xrightarrow{\tau} s'\}) \leq 1$ and for all $\alpha \in \Omega \cup \Upsilon$, $card(\{s' \in S | s \xrightarrow{\alpha} s'\}) \leq 1$. That is, for each state $s \in S$, if an internal action is executed, then the next state depends only on s, and if a valued input (or valued output) α is executed, then the next state depends only on s and α. We then have the following proposition:

Proposition 1. *If $\mathcal{TC}$ is an initialized, deterministic IOSTS without internal actions, then for all $\sigma \in traces(\mathcal{TC})$, the set of states $\mathcal{TC}$ $after\,\sigma$ is a singleton.*

A location q of an IOSTS is *input-complete* if for each state $s =< q, v >\in S$ and each valued input $\alpha \in \Upsilon$, the set $\{s'|s \overset{\alpha}{\Longrightarrow} s'\}$ is nonempty. An IOSTS is *complete* if for each $s \in S$, $\alpha \in \Omega \cup \Upsilon \cup \Sigma^{int}$, the set $\{s'|s \xrightarrow{\alpha} s'\}$ is nonempty.

Operations on IOSTS. For an IOSTS $\mathcal{S}$ with alphabet $\Sigma_{\mathcal{S}} = \Sigma^i_{\mathcal{S}} \cup \Sigma^o_{\mathcal{S}} \cup \Sigma^{int}_{\mathcal{S}}$ and two disjoint sets of actions Σ^i, Σ^o with $\Sigma^i \cup \Sigma^o \subseteq \Sigma^i_{\mathcal{S}} \cup \Sigma^o_{\mathcal{S}}$, we denote by $mirror(\mathcal{S}, \Sigma^i, \Sigma^o)$ the IOSTS which is the same as $\mathcal{S}$ except for its input and output actions, which are $(\Sigma^i_{\mathcal{S}} \setminus \Sigma^i) \cup \Sigma^o$ and $(\Sigma^o_{\mathcal{S}} \setminus \Sigma^o) \cup \Sigma^i$, respectively. We denote by $mirror(\mathcal{S})$ the operation $mirror(\mathcal{S}, \Sigma^i_{\mathcal{S}}, \Sigma^o_{\mathcal{S}})$. Intuitively, $mirror(\mathcal{S}, \Sigma^i, \Sigma^o)$ replaces in the IOSTS $\mathcal{S}$ all the actions from Σ^i with actions from Σ^o, and reciprocally, while $mirror(\mathcal{S})$ transforms all inputs of $\mathcal{S}$ into outputs and reciprocally.

We now define a generic composition operation for IOSTS, which we then specialize to define two operations (parallel and product) that are needed in the sequel. For two IOSTS $\mathcal{S}_1 = \langle D_1, \Theta_1, Q_1, q^0_1, \Sigma_1, \mathcal{T}_1\rangle$ with data $D_1 = V_1 \cup P_1 \cup M_1$ and alphabet $\Sigma_1 = \Sigma^i_1 \cup \Sigma^o_1 \cup \Sigma^{int}_1$ (resp. $\mathcal{S}_2 = \langle D_2, \Theta_2, Q_2, q^0_2, \Sigma_2, \mathcal{T}_2\rangle$ with data $D_2 = V_2 \cup P_2 \cup M_2$ and alphabet $\Sigma_2 = \Sigma^i_2 \cup \Sigma^o_2 \cup \Sigma^{int}_2$), we say $\mathcal{S}_1, \mathcal{S}_2$ are *compatible for composition* if $(V_1 \cup M_1) \cap (V_2 \cup M_2) = \emptyset$, $(P_1 \cup M_1) \cap M_2 = \emptyset$, $(P_2 \cup M_2) \cap M_1 = \emptyset$, $\Sigma^i_1 = \Sigma^o_2$, $\Sigma^o_1 = \Sigma^i_2$, and each action $a \in \Sigma^i_1 \cup \Sigma^i_2$ (which also means $a \in \Sigma^o_1 \cup \Sigma^o_2$) has the same signature in both systems. Note that the two systems may share some parameters, and that a variable of one system may be a parameter of the other (meaning that it can be read, but not modified).

The composition operation lets each system perform independently its internal actions (that are not in the alphabet of the other), and imposes synchronization on the shared actions. Formally, the composition $\mathcal{S} = \mathcal{S}_1 | \mathcal{S}_2$ of compatible IOSTS $\mathcal{S}_1, \mathcal{S}_2$ is the IOSTS $\langle D, P, \Theta, Q, q^0, \Sigma, \mathcal{T}\rangle$ that consists of the following elements: $V = V_1 \cup V_2$, $P = (P_1 \cup P_2) \setminus (V_1 \cup V_2)$, $M = M_1 \cup M_2$, $\Theta = \Theta_1 \wedge \Theta_2$, $Q = Q_1 \times Q_2$, $q^0 = \langle q^0_1, q^0_2\rangle$, $\Sigma^i = \emptyset$, $\Sigma^o = \Sigma^o_1 \cup \Sigma^o_2$, $\Sigma^{int} = \Sigma^{int}_1 \cup \Sigma^{int}_2$. The set $\mathcal{T}$ of transitions of the composed system is defined as follows:

- for each transition $\langle q_1, a, \mu, G, A, q'_1\rangle \in \mathcal{T}_1$ with $a \in \Sigma^{int}_1 \setminus \Sigma^{int}_2$ and for each location of the form $\langle q_1, q_2\rangle$ with $q_2 \in Q_2$, there is in $\mathcal{S}_1|\mathcal{S}_2$ a transition $\langle\langle q_1, q_2\rangle, a, \mu, G, A \cup \bigcup_{y \in V_2}\{y := y\}, \langle q'_1, q_2\rangle\rangle$ (and similarly for all actions $a \in \Sigma^{int}_2 \setminus \Sigma^{int}_1$),
- for each transitions $\langle q_1, a, \mu_1, G_1, A_1, q'_1\rangle \in \mathcal{T}_1$, $\langle q_2, a, \mu_2, G_2, A_2, q'_2\rangle \in \mathcal{T}_2$ with $a \in \Sigma^i_1$, let $G_{1,2}$ (resp. $A_{1,2}$) denote the expression obtained by replacing in G_2 (resp. A_2) each message from μ_2 by the corresponding, same-position message from μ_1. Then, in $\mathcal{S}_1|\mathcal{S}_2$ there is a transition $\langle\langle q_1, q_2\rangle, a, \mu_1, G_1 \wedge G_{1,2}, A_1 \cup A_{1,2}, \langle q'_1, q'_2\rangle\rangle$ (and similarly when $a \in \Sigma^i_2$ or $a \in \Sigma^{int}_1 \cap \Sigma^{int}_2$).

Parallel operation. The parallel operation $||$ is the same as the composition $|$, except that it is defined only for IOSTS $\mathcal{S}_1, \mathcal{S}_2$ that satisfy the stronger compatibility requirements $D_1 \cap D_2 = \emptyset$, $\Sigma_1^{int} \cap \Sigma_2^{int} = \emptyset$. We use this operation to model the execution of a test case on a black-box implementation.

Product operation. The product operation $\times$ of IOSTS $\mathcal{S}_1$, $\mathcal{S}_2$ is defined as $\mathcal{S}_1 \times \mathcal{S}_2 = mirror(\mathcal{S}_1 | mirror(\mathcal{S}_2), \Sigma_1^i, \Sigma_1^o)$. The compatibility requirement for the product operation (in addition to the fact that $\mathcal{S}_1$ and $mirror(\mathcal{S}_2)$ must be compatible for composition) is $\Sigma_1^{int} = \Sigma_2^{int}$. We use this operation during test generation to select a part of a specification, by computing the product with another IOSTS called a *test purpose*. For precise selection, we allow the test purpose to have access to the specification's internal actions and data.

We are interested in the following relationships between the traces of the parallel and product-composed systems and the traces of their components.

Proposition 2 (Traces of the Parallel and Product Compositions). *For IOSTS $\mathcal{S}_1$, $\mathcal{S}_2$ that are compatible for the parallel (resp. product) composition:*

1. $traces(\mathcal{S}_1 || \mathcal{S}_2) = traces(\mathcal{S}_1) \cap traces(\mathcal{S}_2)$
2. *if $\mathcal{S}_2$ is complete, then $traces(\mathcal{S}_1 \times \mathcal{S}_2) = traces(\mathcal{S}_1)$.*

Sketch of Proof. To prove the first item, we show that for all states s_1, s_1' of $\mathcal{S}_1$ (resp. s_2, s_2' of $\mathcal{S}_2$), and for all valued input or output $\alpha \in \Omega_1 \cup \Upsilon_1$ (which also means $\alpha \in \Upsilon_2 \cup \Omega_2$), the relation $\langle s_1, s_2 \rangle \overset{\alpha}{\Longrightarrow} \langle s_1', s_2' \rangle$ holds in $\mathcal{S}_1 || \mathcal{S}_2$ iff $s_1 \overset{\alpha}{\Longrightarrow} s_1'$ holds in $\mathcal{S}_1$ and $s_2 \overset{\alpha}{\Longrightarrow} s_2'$ holds in $\mathcal{S}_2$. For this, we first prove that internal actions of each system leave the state of the other unchanged (which is true essentially because of the hypothesis that the data of the two systems are disjoint), and that, for a valued input (or output) α, a transition relation involving α exists in the composition if and only if it exists in both components. For the second item, we first note that the traces of an IOSTS $\mathcal{S}_2$, and those of $\mathcal{S}_2$ modified by a *mirror* operation, are the same. Thus, it is enough to show that, for IOSTS $\mathcal{S}_1$ and $\mathcal{S}_2'$ that are compatible for the $|$-composition, which have the same internal actions, and such that $\mathcal{S}_2'$ is complete, the equality $traces(\mathcal{S}_1 | \mathcal{S}_2') = traces(\mathcal{S}_1) \cap traces(\mathcal{S}_2')$ holds. For the $\subseteq$ inclusion, we proceed in the same manner as for the corresponding inclusion in item 1 (here, the data independence hypothesis is not used) and we use the fact that (since $\mathcal{S}_2'$ is complete), $traces(\mathcal{S}_2') = (\Upsilon_2 \cup \Omega_2)^* = (\Omega_1 \cup \Upsilon_1)^*$. For the $\supseteq$ inclusion of item 2, we use again the fact that $\mathcal{S}_2'$ is complete: by composing it with $\mathcal{S}_1$, any sequence of valued actions which is present in the latter is also allowed by the former, therefore it is also present in the composition $\mathcal{S}_1 | \mathcal{S}_2'$. $\square$

3 Conformance Testing with IOSTS

Conformance testing [11] is a methodology for validating reactive systems. It consists in testing a *conformance relation* between a *specification*, which is a formal model of a system, and an *implementation* of the system, which is a physical process (typically, a program running on a machine) that can only be controlled and observed at its interfaces. This is done by running *test cases* on the implementation and obtaining *verdicts* about the conformance. In general, three kinds of verdicts are distinguished. Informally, *Fail* means that non-conformance was detected, *Pass* means that the implementation behaved in conformance with the specification and the goal of the test experiment (described by the a so-called *test purpose*) has been reached, and *Inconclusive* means that the implementation behaved correctly but, due to the lack of control on the implementation, it did not allow to reach the expected goal. We model specifications, implementations, test purposes and test cases as IOSTS, with some restrictions for each category.

Specifications. A specification is an initialized IOSTS. In this paper we consider (for test generation) specifications without cycles of internal actions. We then indicate how to extend the approach to allow cycles of internal actions that perform internal, deterministic, computations, during which the system does not communicate with its environment.

As a running example, we use the specification of the BRP protocol [8], 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 parameters *last* (resp. *max*) stand for the number of packets in the file being sent (resp. the maximum number of retransmissions of a packet). The variable *head* is used to count the packets (which are $f(0)$ to $f(last-1)$) and variable rn is for counting retransmissions. On reception of a REQ input with messsage F from its client, the sender saves it in variable f, then iteratively sends the packets using the output action *MSG* with message m. A packet can be acknowledged (input ACK), or there may be a *timeout* (an internal action), after which the sender decides to resend the same packet. The status of the whole transmission is described by either *CONF_OK*, meaning that all the packets were sent and acknowledged, *CONF_DONT_KNOW*, meaning that all the packets were sent and all but the last were acknowledged, or *CONF_NOT_OK*, meaning that some intermediary packet was not acknowledged.

Implementations. An implementation is a physical object. However, in order to be able to formally reason with it, we assume (this is a usual "testing hypothesis") that the implementation can be modelled by an IOSTS, which is unknown except for its input and output alphabet and the signatures of its actions, which are supposed to be the same as those of the specification.

Test Purposes. A test purpose is used to select a part of the specification, for which a test case will be generated. A "good" test purpose should be simple (typically, much simpler than the specification) and should select exactly the scenarios that the user has in mind. In practice, as our experience with the tool TGV [13] , the process is iterative: a first test purpose only grossly selects a part

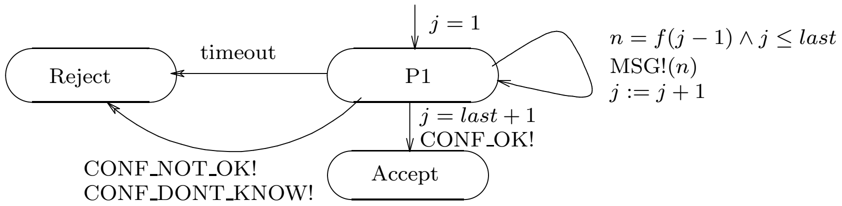

Fig. 2. A Test Purpose for the BRP Sender.

of the specification, then the user has to examine the result and modify the test purpose, and repeatedly so until a satisfactory result is obtained. Here, we use IOSTS as test purposes and the product operation as the selection mechanism. This gives enough expressiveness for describing, e.g., parameterized scenarios that would be hard to describe with, e.g., finite automata or temporal logic.

Formally, let $\mathcal{S} = \langle D_\mathcal{S}, \Theta_\mathcal{S}, Q_\mathcal{S}, q^0_\mathcal{S}, \Sigma_\mathcal{S}, \mathcal{T}_\mathcal{S} \rangle$ be a specification IOSTS. A *test purpose* of $\mathcal{S}$ is an IOSTS $\mathcal{TP} = \langle D_{\mathcal{TP}}, \Theta_{\mathcal{TP}}, Q_{\mathcal{TP}}, q^0_{\mathcal{TP}}, \Sigma_{\mathcal{TP}}, \mathcal{T}_{\mathcal{TP}} \rangle$ together with a set of locations $Accept_{\mathcal{TP}} \subseteq Q_{\mathcal{TP}}$ such that $\mathcal{TP}$ is initialized, complete, $\times$-compatible with $\mathcal{S}$, and all the transitions with origin in the set $Accept_{\mathcal{TP}}$ are self-loops.

The IOSTS represented in Figure 2 is a test purpose for the specification of the BRP sender. It is used to select the scenarios in which the latter sends all messages exactly once (without retransmissions) and exits successfully with a *CONF_OK* confirmation. It has the variable j, parameters f and $last$ (which are also a variable, respectively a parameter of the specification), and one *Accept* location. The location *Reject* is used to discard executions in which the BRP sender does not behave as intended (i.e., it executes the internal action *timeout*, known to be followed by a retransmission, or it exits with a wrong confirmation).

This test purpose is not complete but, by convention, it is implicitly completed as follows: in each location, any action that does not (syntactically) label an outgoing transition, will produce a self-loop; and any action that labels some outgoing transition with a guard G, will generate a transition to *Reject* with the guard $\neg G$. This simplifies the writing of test purposes by allowing to focus on the intended behaviour for testing, leaving practically irrelevant details (such as completeness) to the test generation tool.

Given a specification $\mathcal{S}$ and a test purpose $\mathcal{TP}$ of $\mathcal{S}$, we define the product $\mathcal{P} = \mathcal{S} \times \mathcal{TP}$, the set of locations $Accept_\mathcal{P} = Q_\mathcal{S} \times Accept_{\mathcal{TP}}$, and the set of traces of the product $Atraces(\mathcal{P}) = \{\sigma \in (\Omega_\mathcal{P} \cup \Upsilon_\mathcal{P})^* | \exists s_0 \in S^0_\mathcal{P}, \exists s = \langle l, v\rangle \in S_\mathcal{P} : l \in Accept_\mathcal{P}, s_0 \stackrel{\sigma}{\Longrightarrow}_\mathcal{P} s\}$. By Proposition 2 item 2, we know that $Atraces(\mathcal{P}) \subseteq traces(\mathcal{S})$. Intutively, $Atraces(\mathcal{P})$ is the set of traces of the specification that are selected (through the product operation) by the test purpose.

The product between the BRP sender and its test purpose is partially represented in Figure 3.

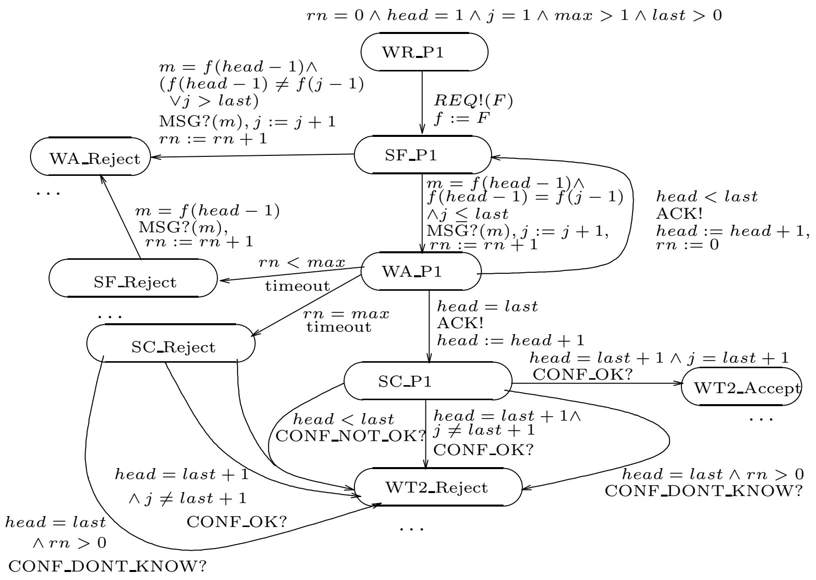

Fig. 3. Product of Sender and Test Purpose (partially represented)

Test Cases. Test cases are used to assign verdicts to implementations. This is done by running the two systems in parallel, and by observing the state of the test case at the end of the interaction. Formally, a test case is an initialized, deterministic IOSTS, together with three disjoint sets of locations *Pass*, *Inconclusive* and *Fail*. Moreover, a test case should react promptly to any input from the implementation, thus, it should not have internal actions, and all its locations (except those in the set $Fail \cup Pass \cup Inconclusive$) should be input-complete.

An example of test case is represented in Figure 4. It starts by giving to the sender's implementation a file F to transmit. Then it expects to receive the successive packets in the file, and tries to acknowledge each packet. If it succeeds and receives the *CONF_OK* confirmation, the verdict is *Pass*, as the implementation behaved in conformance with the specification and the test purpose is satisfied: all packets were sent without retransmission. If, however, the test case gets another copy of the packet that it has just received, this means that the sender performed a retransmission, and the verdict is *Inconclusive*: the implementation behaved in conformance with the specification, but it did not allow to satisfy the test purpose. Finally, if some other input is received, the verdict is *Fail*, as the implementation emitted an output that is not allowed by the specification. For simplicity, the locations *Fail* are not represented in Figure 4.

Verdicts. We formalize the notion of verdict. Let $\mathcal{TC}$ be a test case. We denote by ***Pass*** (respectively ***Inconclusive***, ***Fail***) the set of states of $\mathcal{TC}$ whose locations are in the set *Pass* (resp. *Inconclusive*, *Fail*). Let I be the implementation

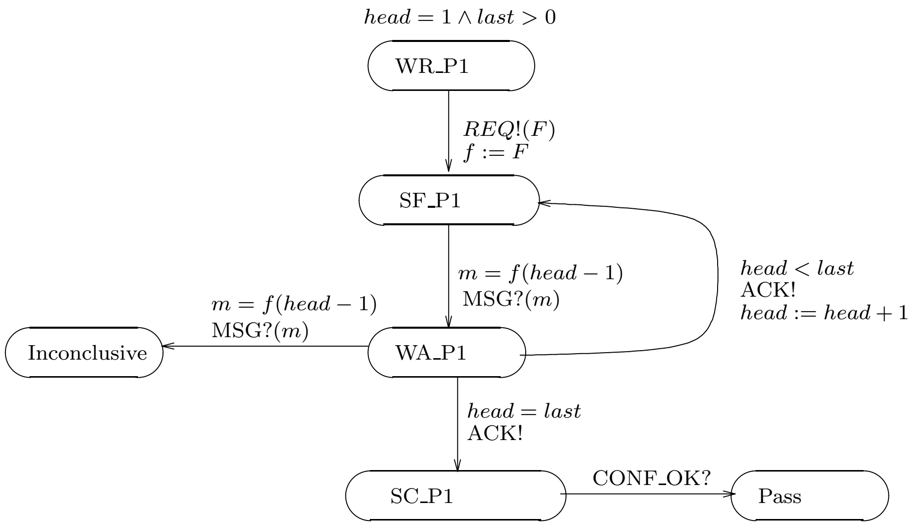

Fig. 4. Example of Test Case for the BRP Sender.

under test. For a trace $\sigma \in traces(I||\mathcal{TC})$, we write $verdict(\mathcal{TC}, I, \sigma) =$ Pass if $\mathcal{TC}\,after\ \sigma \subseteq \boldsymbol{Pass}$. We introduce the notations $verdict(\mathcal{TC}, I, \sigma) =$ Fail and $verdict(\mathcal{TC}, I, \sigma) =$ Inconclusive in a similar way.

Conformance. The conformance relation links an implementation with *instances* of the specification and the test purpose. Indeed, in practice, implementations are compared against specifications with known values of the parameters.

Definition 2 (Conformance Relations). *Let $\mathcal{S}(\pi)$ be an instance of IOSTS $\mathcal{S}$, $\mathcal{TP}$ be a test purpose for $\mathcal{S}$, $\mathcal{P} = \mathcal{S} \times \mathcal{TP}$ their product, and $\mathcal{P}(\pi) = (\mathcal{S} \times \mathcal{TP})(\pi)$ the corresponding instance of the product.*

1. *An implementation I is* conformant *to the instance $\mathcal{S}(\pi)$ of the specification $\mathcal{S}$, denoted $I\,conf\,\mathcal{S}(\pi)$, if for all traces $\sigma \in traces(\mathcal{S}(\pi)) : out(I\ after\ \sigma) \subseteq out(\mathcal{S}(\pi)\ after\ \sigma)$.*
2. *An implementation I is conformant to the instance $\mathcal{S}(\pi)$ of the specification $\mathcal{S}$ and relative to the test purpose $\mathcal{TP}$, denoted $I\,conf_{\mathcal{TP}}\ \mathcal{S}(\pi)$, if for all $\sigma \in pref(Atraces((\mathcal{S} \times \mathcal{TP})(\pi))) : out(I\ after\,\sigma) \subseteq out(\mathcal{S}(\pi)\ after\,\sigma)$.*

That is, after each trace of the specification, the possible outputs of the implementation are included in those of the specification. In the second case, the inclusion should hold after each prefix of a trace of the specification that is selected (through the product mechanism) by the test purpose.

4 Correctness of Test Cases

We formalize what it means for a test case to be correct, relative to a given specification and test purpose and for a given class of implementations. The intuition is that any instance of the test case should always give the right verdict when executed on an implementation in the class.

Definition 3 (Correctness of Test Cases). *Let $\mathcal{S}$ be a specification, $\mathcal{TP}$ a test purpose of $\mathcal{S}$, and $\mathcal{I}$ a set of implementations. Let $\mathcal{TC}$ be a test case such that the parameter sets of $\mathcal{TC}$ and $\mathcal{S}$ are the same, and π a valuation of the parameters.*

- *$\mathcal{TC}$ is* sound *for $\mathcal{S},\mathcal{I}$ if for every instance $\mathcal{TC}(\pi)$ and implementation $I \in \mathcal{I}$, if there exists $\sigma \in traces(I||\mathcal{TC}(\pi))$ such that $verdict(\mathcal{TC}(\pi), I, \sigma) =$* Fail*, then I is not conformant to $\mathcal{S}(\pi)$.*
- *$\mathcal{TC}$ is* relatively complete *for $\mathcal{S}, \mathcal{TP}, \mathcal{I}$ if for every instance $\mathcal{TC}(\pi)$ and implementation $I \in \mathcal{I}$: if I is not conformant to $\mathcal{S}(\pi)$ relative to $\mathcal{TP}$, then there exists $\sigma \in traces(I||\mathcal{TC}(\pi))$ such that $verdict(\mathcal{TC}(\pi), I, \sigma) =$* Fail.
- *$\mathcal{TC}$ is* accurate *for $\mathcal{S}, \mathcal{TP}, \mathcal{I}$ if for every instance $\mathcal{TC}(\pi)$, implementation $I \in \mathcal{I}$, trace $\sigma \in traces(I||\mathcal{TC}(\pi))$: $verdict(\mathcal{TC}(\pi), I, \sigma) =$* Pass *iff $\sigma \in Atraces((\mathcal{S} \times \mathcal{TP})(\pi))$.*
- *$\mathcal{TC}$ is* conclusive *for $\mathcal{S}, \mathcal{TP}, \mathcal{I}$ if for every instance $\mathcal{TC}(\pi)$, implementation $I \in \mathcal{I}$, trace $\sigma \in traces(I||\mathcal{TC}(\pi))$: $verdict(\mathcal{TC}(\pi), I, \sigma) =$* Inconclusive *implies $\sigma \in traces(\mathcal{S}(\pi))$ and $\sigma \notin pref(Atraces((\mathcal{S} \times \mathcal{TP})(\pi)))$.*

A test case $\mathcal{TC}$ is correct *for $\mathcal{S}, \mathcal{TP}, \mathcal{I}$ if it is sound for $\mathcal{S}, \mathcal{I}$ and relatively complete, accurate and conclusive for $\mathcal{S}, \mathcal{TP}, \mathcal{I}$.*

Intuitively, *soundness* means that the test case rejects only non-conformant implementations (in the given class). *Relative completeness* means that the test case can detect all implementations in the given class, which are non-conformant with the specification relative to the test purpose. (It does not mean *all* these implementations will actually be detected, because of nondeterminism of the implementation and because there may be an infinity of traces to consider.) *Accuracy* means that the verdict Pass is given when the observed trace of the implementation is a trace of the specification that is selected by the test purpose. Finally, *conclusiveness* means that the verdict Inconclusive is given when the observed trace of the implementation is a trace of the specification, but it cannot be extended into a trace that eventually produces the verdict Pass.

5 Test Case Generation

We present a procedure for generating symbolic test cases that are correct in the sense of Definition 3. The first step is computing the product between the test purpose and the specification, which was defined in Section 3. The following steps consist in transforming and simplifying the product (e.g., Figure 3) to obtain a test case (e.g., Figure 4) with all the required properties.

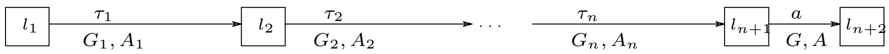

Fig. 5. Sequence of Internal Actions.

Closure: Eliminating Internal Actions. A test case should react promptly to inputs from the implementation. A natural way to obtain this is by requiring that every location of a test case (except the verdict locations) is input-complete. However, the possible inputs in a location may be hidden by internal actions. For example, consider the location *WA_P1* of the IOSTS represented in Figure 3. Here, the system can only send an output *ACK!* or execute the internal action *timeout*, but the inputs (*MSG?*, *CONF_OK?*, etc) can only be executed after a *timeout*. To make the location input-complete, we first have to eliminate the latter actions. Thus, to obtain test cases where all locations are input-complete, all internal actions have to be eliminated from the product $\mathcal{P} = \mathcal{S} \times \mathcal{TP}$. For this, the idea is to compute the effect of any *sequence* of internal actions that leads to an input-labeled transition, and to encode it in the last transition. This gives a simple syntactical procedure, which works if the specification does not contain cycles of internal actions. (A way to handle cycles is proposed at the end of this section.)

More formally, let $\tau_1, \tau_2, \ldots, \tau_n$ be a sequence of internal actions that leads to an input action a (cf. Figure 5). The guard and assignments corresponding to τ_i are G_i, A_i, and the guard (resp. assignments) corresponding to a are G, A. Then, it is not hard to see that a trace-equivalent system is obtained by replacing the whole sequence 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.

We obtain an IOSTS $\overline{\mathcal{P}}$ with the same traces as $\mathcal{P}$, but without internal actions. The set of accepting locations $Accept_{\overline{\mathcal{P}}}$ is defined by $Accept_{\overline{\mathcal{P}}} = Q_{\overline{\mathcal{P}}} \cap Accept_{\mathcal{P}}$. For the product IOSTS represented in Figure 3, the corresponding IOSTS without internal actions is represented in Figure 6. The differences come from the transitions labeled *timeout*, which have disappeared, and whose guards ($rn < max$, resp. $rn = max$) have been propagated to the nearest observable transitions.

Determinization. Nondeterminism is prohibited in test cases, because the verdicts should not depend on internal choices of the tester. Thus, the following step is to eliminate nondeterminism from the IOSTS $\overline{\mathcal{P}}$. This means computing another IOSTS denoted $[\overline{\mathcal{P}}]$, which has the same traces as $\overline{\mathcal{P}}$ (and thus, the same traces as the product $\mathcal{P} = \mathcal{S} \times \mathcal{TP}$), but without nondeterministic choices. The typical case of nondeterministic choice is represented in Figure 7: an (input or output) action a leading to two different effects on the variables and/or the control.

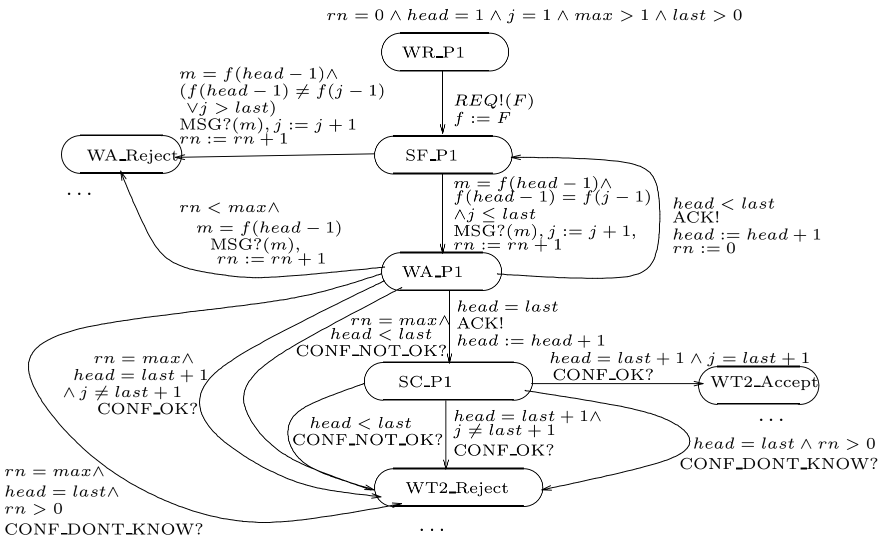

Fig. 6. IOSTS of Figure 3, after Elimination of Internal Actions.

Eliminating nondeterminism in a symbolic transition system is a difficult problem in general. Here, we propose a simple heuristic that takes care of common situations, such as the one represented in Figure 7. The idea is to *postpone* the effect of a nondeterministic choice on the observable actions that follow it. In the situation represented below, this amounts to splitting the two transitions (with guards G_1, G_2 and assignments A_1, A_2) 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. A new location $q_{2,3}$ is introduced, which we call a *copy* of locations q_2, q_3. Thus, if b is the next action, then the assignment A_1 should have been executed, and it is composed with the guard and the assignment corresponding to b (which produces the guard $G_3 \circ A_1$ and assignement $A_3 \circ A_1$). Similarly, if c is the next action, then A_2 should have been executed, which produces the guard $G_4 \circ A_2$ and assignement $A_4 \circ A_2$. Of course, if b and c are actually the same action, then the same scheme has to be applied again, and the whole procedure might not terminate. It can be shown that it does terminate if the maximal sequence of actions involved in nondeterministic choices (such as the sequence composed of action a in Figure 7) does not produce cycles in the graph of the IOSTS. Also, knowing that some guards are mutually exclusive can help the determinization procedure to terminate. For example, determinization leaves the IOSTS represented in Figure 6 unchanged, because wherever there are transitions with the same origin and labeled with the same action (*MSG?*, *ACK!*) and with several possible outcomes, the corresponding guards are mutually exclusive.

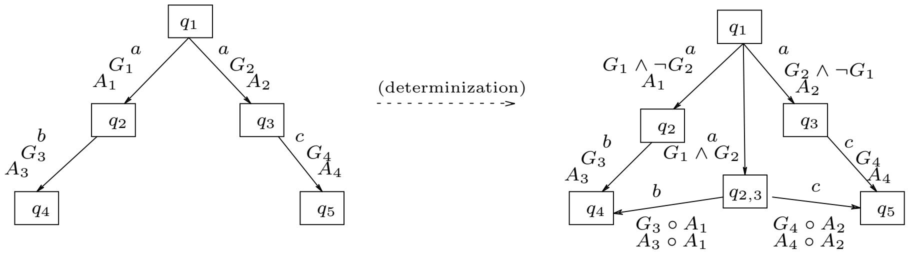

Fig. 7. Determinization.

We associate to the IOSTS $[\overline{\mathcal{P}}]$ (obtained after product, closure and determinization) a set of accepting locations $Accept_{[\overline{\mathcal{P}}]}$, which is the union of $Accept_{\overline{\mathcal{P}}}$ and of the set of new locations produced by determinization, which are copies of some location in $Accept_{\overline{\mathcal{P}}}$.

Selection. The next step consists in selecting a part of the control graph of the IOSTS $[\overline{\mathcal{P}}]$ that leads to the set of locations $Accept_{[\overline{\mathcal{P}}]}$, and in adding transitions to a new location *fail*, such as to make the IOSTS input-complete. We define three subsets of the set of locations $Q_{[\overline{\mathcal{P}}]}$ as follows:

- let *Pass* denote the set $Accept_{[\overline{\mathcal{P}}]}$
- let $Leads2Pass$ denote the set of locations $q \in Q_{[\overline{\mathcal{P}}]} \setminus Pass$ such that, in the graph $\langle Q_{[\overline{\mathcal{P}}]}, \mathcal{T}_{[\overline{\mathcal{P}}]} \rangle$, there exists a path (i.e., a sequence of contiguous locations and transitions) from the initial location q^0 to q and from q to *Pass*
- let *Inconclusive* denote the set of locations $q' \in Q_{[\overline{\mathcal{P}}]} \setminus Pass \setminus Leads2Pass$ such that there is a transition in $\mathcal{T}_{[\overline{\mathcal{P}}]}$ with origin in $Leads2Pass$, destination q', labeled by an input $a \in \Sigma^i_{[\overline{\mathcal{P}}]}$.

Let also *fail* denote a new location ($fail \notin Q_{[\overline{\mathcal{P}}]}$). We define the test case as an IOSTS $\mathcal{TC} = \langle D_{\mathcal{TC}}, \Theta_{\mathcal{TC}}, Q_{\mathcal{TC}}, q^0_{\mathcal{TC}}, \Sigma_{\mathcal{TC}}, \mathcal{T}_{\mathcal{TC}} \rangle$ as follows: $D_{\mathcal{TC}} = D_{[\overline{\mathcal{P}}]}$, $\Theta_{\mathcal{TC}} = \Theta_{[\overline{\mathcal{P}}]}$, $Q_{\mathcal{TC}} = Pass \cup Leads2Pass \cup Inconclusive \cup \{fail\}$, $q^0_{\mathcal{TC}} = q^0_{[\overline{\mathcal{P}}]}$ (if $q^0_{[\overline{\mathcal{P}}]} \notin Pass \cup Leads2Pass$ then the test case is not defined). The alphabet of the test case is $\Sigma^i_{\mathcal{TC}} = \Sigma^i_{[\overline{\mathcal{P}}]}$, $\Sigma^o_{\mathcal{TC}} = \Sigma^o_{[\overline{\mathcal{P}}]}$, $\Sigma^{int}_{\mathcal{TC}} = \emptyset$. The set of transitions $\mathcal{T}_{\mathcal{TC}}$ consists of the transitions of $\mathcal{T}_{[\overline{\mathcal{P}}]}$ with origin and destination in $Pass \cup Leads2Pass \cup Inconclusive$, and of a set of new transitions, with origin in every $q \in Leads2Pass$ and for each input $a \in \Sigma^i_{\mathcal{TC}}$. Every such new transition leads to *fail*, and, for a given origin q and action a, its guard is the negation of the disjunction of all the guards of transitions in $\mathcal{T}_{[\overline{\mathcal{P}}]}$, with same origin q and action a. The assignments of the new transitions are empty. They are used to make the test case input-complete, by letting go to *fail*, from any state, any valued input that is not allowed by the specification.

For the IOSTS partially represented in Figure 6, the selection operation eliminates precisely the parts that are not represented. The set of locations *Pass* consists of only one element (*WT2_Accept*). The set *Inconclusive* consists of locations

WA_Reject, $WT2\_Reject$. For simplicity, the location *fail*, and the transitions leading to it, are not represented. By comparing the IOSTS in Figures 6 and 4, it is clear that the former still needs considerable simplification. This is the role of the simplification operations, which we describe in more detail in the next section.

However, the IOSTS $\mathcal{TC}$ obtained so far has all the properties required from a test case: it is input-complete (this is obtained by adding the new transitions to *fail*), it is initialized (this is inherited from the specification and the test purpose, and is preserved by subsequent transformations), it is deterministic (this is obtained by closure and determinization), and furthermore:

Proposition 3 (Correctness of Test Generation). *For any specification $\mathcal{S}$ and test purpose $\mathcal{TP}$ of $\mathcal{S}$, the test case $\mathcal{TC}$ obtained from $\mathcal{S}$ and $\mathcal{TP}$ by test generation is correct for $\mathcal{S}$, $\mathcal{TP}$, and the class of implementations that have the same inputs, outputs, and signatures as $\mathcal{S}$.*

Proof outline. We have to prove that the four properties of definition 3 (soundness, relative completeness, accuracy and conclusiveness) hold. First, it is easy to see that, by construction, $\mathcal{TC}$ has the same parameter set as $\mathcal{S}$ and that, for an arbitrary implementation I which has the same inputs, outputs, and signatures as $\mathcal{S}$, I is compatible with $\mathcal{TC}$ for parallel composition. (Thus, the valued inputs of I are the valued outputs of $\mathcal{TC}$ and reciprocally.) We consider an arbitrary instance $\mathcal{TC}(\pi)$ of $\mathcal{TC}$, and the corresponding instances $\mathcal{S}(\pi)$ of the specification $\mathcal{S}$ and $(\mathcal{S} \times \mathcal{TP})(\pi)$ of the synchrounous product $\mathcal{S} \times \mathcal{TP}$. As in Section 3, we denote by ***Pass*** (respectively ***Inconclusive*** and ***Fail***) the set of states of $\mathcal{TC}(\pi)$ whose locations are in the set *Pass* (resp. *Inconclusive* and $\{fail\}$). In the following proofs, we implicitly make use of Proposition 1 (i.e., when we say a trace σ brings the $\mathcal{TC}(\pi)$ in a given location, we implicitly assume that $\mathcal{TC}(\pi)\,\mathit{after}\,\sigma$ is a singleton).

<u>Soundness :</u> Let $\sigma \in traces(I||\mathcal{TC}(\pi))$ such that $verdict(\mathcal{TC}(\pi), I, \sigma) = \text{Fail}$. By the first item of Proposition 2, σ is a trace of I and a trace of $\mathcal{TC}(\pi)$. By definition of *verdict*, we obtain $\mathcal{TC}(\pi)\,\mathit{after}\,\sigma \subseteq \textbf{\textit{Fail}}$, meaning that σ leads the test case to the location *fail*. Consider the prefix σ' of σ obtained by removing its last valued action α. By construction of the test case, we obtain that α is a valued input of $\mathcal{TC}(\pi)$ (thus, a valued output of I), $\sigma' \in traces((\mathcal{S} \times \mathcal{TP})(\pi))$, and $\sigma = \sigma' \cdot \alpha \notin traces((\mathcal{S} \times \mathcal{TP})(\pi))$. Using the second item of Proposition 2, we obtain that $\sigma' \in traces(\mathcal{S}(\pi))$ and $\sigma \notin traces(\mathcal{S}(\pi))$, thus, $\alpha \notin out(\mathcal{S}(\pi)\ after\ \sigma')$. But $\sigma = \sigma' \cdot \alpha$ is a trace of I and α is a valued output of I, thus $\alpha \in out(I\ after\ \sigma')$. Hence, I does not conform to $\mathcal{S}(\pi)$.

<u>Relative completeness :</u> Suppose that I does not conform to $\mathcal{S}(\pi)$ relative to $\mathcal{TP}$. By definition of *conf*, there exists a trace $\sigma' \in pref(Atraces((\mathcal{S} \times \mathcal{TP})(\pi)))$ and a valued output α of the implementation such that $\alpha \in out(I\ after\ \sigma')$ but $\alpha \notin out(\mathcal{S}(\pi)\ after\ \sigma')$. We choose a trace σ' which is minimal with this property, and let $\sigma = \sigma' \cdot \alpha$. Thus, $\sigma \in traces(I)$ and $\sigma \notin traces(\mathcal{S}(\pi))$. From $\sigma' \in pref(Atraces((\mathcal{S} \times \mathcal{TP})(\pi)))$ and the minimality of σ, we obtain (by construction of the test case) also $\sigma' \in traces(\mathcal{TC}(\pi))$ and $(\mathcal{TC}(\pi)\ after\ \sigma') \cap$

$(\boldsymbol{Fail} \cup \boldsymbol{Pass} \cup \boldsymbol{Incon}) = \emptyset$; thus, σ' brings the test case in a location q which is not in $Pass \cup Inconclusive \cup \{fail\}$, thus q is input-complete. But α is also an input of the test case, thus, the trace $\sigma = \sigma' \cdot \alpha$ is also in $traces(\mathcal{TC}(\pi))$. Now, $\sigma \in traces(\mathcal{TC}(\pi))$ and $\sigma \notin traces(\mathcal{S}(\pi))$ implies, by construction of the test case, that $\mathcal{TC}(\pi)\ after\ \sigma \subseteq \boldsymbol{Fail}$. As $\sigma \in traces(I)$ and $\sigma \in traces(\mathcal{TC}(\pi))$, we obtain, by the first item of Proposition 2, that $\sigma \in traces(I||\mathcal{TC}(\pi))$. The latter together with $\mathcal{TC}(\pi)\ after\ \sigma \subseteq \boldsymbol{Fail}$ implies $verdict(\mathcal{TC}(\pi), I, \sigma) =$ Fail.

Accuracy : By definition, for a trace $\sigma \in traces(I||\mathcal{TC}(\pi))$, $verdict(\mathcal{TC}(\pi), I, \sigma) =$ Pass iff $\mathcal{TC}(\pi)\ after\ \sigma \subseteq \boldsymbol{Pass}$. By construction of the test case, the traces of $\mathcal{TC}(\pi)$ leading to a state in the set $\boldsymbol{Pass}$ are exactly those in $Atraces((\mathcal{S} \times \mathcal{TP})(\pi))$, which proves the property.

Conclusiveness : By definition, for $\sigma \in traces(I||\mathcal{TC}(\pi))$, $verdict(\mathcal{TC}(\pi), I, \sigma) =$ Inconclusive iff $\mathcal{TC}(\pi)\ after\ \sigma \subseteq \boldsymbol{Inconc}$. If σ leads to a state in the set $\boldsymbol{Inconc}$, by construction of the test case, σ is a trace of $(\mathcal{S} \times \mathcal{TP})(\pi)$, thus (by the second item of Proposition 2), $\sigma \in traces(\mathcal{S}(\pi))$. We prove the second part of the property by contradiction. If we had $\sigma \in pref(Atraces((\mathcal{S} \times \mathcal{TP})(\pi)))$, then, by construction of the test case, the location of $\mathcal{TC}(\pi)\ after\ \sigma$ would be in the set $Leads2Accept$, not in $Inconclusive$. This concludes the proof. $\square$

Cycles of Internal Actions. We consider a class of specifications that contain cycles of internal actions, and for which it is still possible to generate test cases with all the correctness properties except relative completeness (cf. Definition 3). This class includes specifications described, e.g., in an imperative programming language, where cycles of internal actions correspond to deterministic control structures like iterations or recursions. For a specification $\mathcal{S}$ with locations Q and transitions $\mathcal{T}$, let $\mathcal{T}^{int} \subseteq \mathcal{T}$ denote the subset of transitions labeled by internal actions (which we call for short *internal transitions*). Let $Q^{int} \subseteq Q$ denote the set of locations such that all transitions with origin in Q^{int} are internal. We call Q^{int} the set of *internal locations*.

The test generation process is redefined in the following way. The product operation is unchanged. The elimination of internal actions is applied only to internal transitions whose origins are in $Q \setminus Q^{int}$. (Thus, if these transitions do not form cycles, the operation terminates, as for example in the case of the IOSTS represented in Figure 3). The determinization operation is the same, and the selection operation is unchanged, except that it does not add transitions leading to *fail* from the internal locations.

To show that the obtained test case $\mathcal{TC}$ is correct (expect for relative completeness), we prove the soundness, accuracy and conclusiveness like in the proof of Proposition 3. (However, we cannot use Proposition 1 any more, since $\mathcal{TC}$ now has internal actions). The proof of relative completeness does not work, because of the internal locations, which are not input-complete. Intuitively, the test case can miss some inputs while it is executing its internal actions. In practice, this problem can be fixed by buffering the inputs for later observation.

6 Simplifying the Test Cases

The test generation procedure defined in the previous section consists essentially of syntactical transformations (which extend the algorithms implemented in the tool TGV [13] to handle symbolic data). Although it does produce correct test cases (e.g., the one represented in Figure 6), the latter are almost never what the user would expect (e.g., something like Figure 4). Indeed, there are unreachable parts, redundant variables, guards and assignments can be simplified, etc. Thus, we need semantic-based simplification techniques that preserve the correctness of test cases. We present an experiment with the HyTech model checker [9] and the PVS theorem prover [17] for simplifying the test case of the BRP, and indicate other potentially useful techniques.

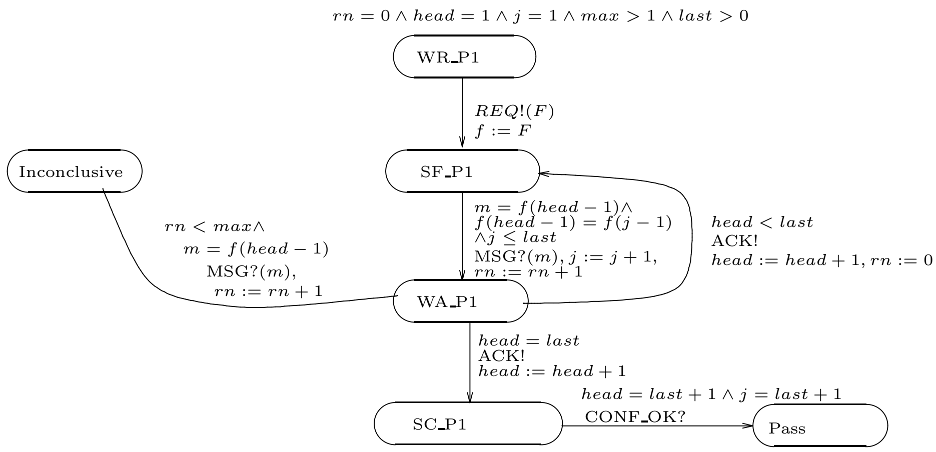

Fig. 8. Test Case after Elimination of Irrelevant Control using HyTech and PVS

We define translations of IOSTS into the HyTech and PVS input languages. All the corresponding files (PVS specifications and proofs, and HyTech scripts) can be downloaded from `http://www.irisa.fr/pampa/perso/rusu/IFM00/`.

Translation to HyTech. HyTech is a model checker originally designed for hybrid systems verification. It handles discrete variables (integers) through a polyhedral library [7]. We had to modify the tool to re-include a widening operator that forces convergence of fixpoints. As HyTech does not handle uninterpreted function symbols, the translation of the test case represented in Figure 6 involves replacing every conjunct containing the uninterpreted function symbol f with *true*. Also, the actions become uninterpreted labels of the HyTech transitions.

Translation to PVS. PVS is a general-purpose theorem prover with a rich input language (typed higher-order logic). Consequently, the translation of IOSTS into PVS preserves all the information and is straightforward.

Simplifying the Test Case. The simplification of the test case represented in Figure 6 is done in two steps. First, HyTech (with widening) is used to automatically generate a set of linear invariants[1]. This gives the following output:

```
Location: Pass
     rn = 1   & head = last + 1   & head = j   & head >= 2   & max >= 2
Location: Inconclusive
     j = head + 1   & head >= 1   & rn >= 1   & max >= 2   & head <= last
Location: SC_P1
     rn = 1   & head = last + 1   & head = j   & head >= 2   & max >= 2
Location: WA_P1
     rn = 1   & j = head + 1   & head >= 1   & max >= 2   & head <= last
Location: SF_P1
     rn = 0   & head = j   & head >= 1   & max >= 2   & head <= last
Location: WR_P1
     rn = 0   & head = 1   & j = 1   & max >= 2   & last >= 1
```

Thus, e.g., whenever control is at location SF_P1, the relation $rn = 0 \wedge head = j \wedge head \leq last$ holds between the variables of the IOSTS. This is useful information: in particular, it shows that location $WT2\_Reject$ is not reachable, thus, it can be eliminated, together with all six transitions leading to it. This also indicates another transition that could be eliminated: the transition from SF_P1 to WA_Reject. This is because its guard $f(head-1) \neq f(j-1) \vee j > last$, in conjunction with the invariant produced by HyTech, evaluates to *false*. However, HyTech cannot evaluate conditions with uninterpreted function symbols, thus, the simplified IOSTS, together with the generated invariants, are translated to PVS, which automatically proves the following invariant:

```
%proved (transition from SF_P1 to Inconclusive is never fireable)
trans1_never_fireable: LEMMA invariant(NOT LAMBDA(s:State):EXISTS (m:Msg):
control(s)=SF_P1 AND m=f(s)(head(s)-1) AND (j(s)>last OR m/=f(s)(j(s)-1)))
```

This allows to eliminate the transition from SF_P1 to WA_Reject, which produces the IOSTS represented in Fig. 8. Other simplifications are then made: given the invariant relations between *head* and j in locations SF_P1, WA_P1, and SC_P1, it is clear that j can be replaced by *head* and eliminated altogether from the IOSTS, since it has no more influence on the variables. The same holds for rn, as the only test $rn < max$ on this variable always evaluates to *true*. Finally, we obtain the test case represented in Figure 4.

Clearly, other program analysis and verification techniques can be useful for symbolic test generation. We are investigating slicing [22] (see [4] for a use of slicing techniques in the context of conformance test generation), and automatic theorem-proving in rich, yet decidable theories such as WS1S [14] or

[1] `Inconclusive` in the HyTech output corresponds to WA_Reject in Figure 6.

the quantifier-free fragment of Presburger arithmetic with uninterpreted function symbols [20]. Integrating these algorithmic and deductive techniques into an environment for symbolic test generation is a longer-term project.

7 Conclusion and Future Work

In this paper we describe some basic steps towards the generation of symbolic test cases in the form of extended transition systems with parameters and variables. The approach generalizes existing work on test generation using enumerative methods [1, 6]. The generated test cases satisfy some correctness properties, which means essentially that they always emit the right verdict. To be useful in practice, the test cases have to be simplified using automated static analysis and proof strategies. This is a first direction for continuing work.

Another future work direction (with the same goal of obtaining simpler test cases) is to generate test cases from an abstraction of the specification, rather than from the (concrete) specification itself. The usual notion of abstraction (over-approximation, cf. [2]) cannot be applied here, because of the asymetry between inputs and outputs in the conformance relation. Over-approximating the outputs of the specification is acceptable (the test cases remain sound, although they lose completeness and accuracy), but over-approximating the inputs may lead to unsoundness (the test case explores behaviours that are not in the specification, which may produce false *Fails*). We need abstractions that over-approximate the outputs and under-approximate the inputs.

Third, symbolic test cases need to be instantiated before they can be executed. An analysis of their behaviours, depending on their parameters and messages, may be useful to find domains of equivalent values. Then, under uniformity hypotheses [3] it is enough to instantiate the parameters and messages to one value in each domain. Also, test generation needs two inputs: specifications and test purposes. Designing test purposes by hand from the specification can be difficult, in particular if a good coverage of the specification is targetted. We believe that classical structural coverage criteria [19], combined with symbolic analysis, may yield interesting test purposes.

Finally, the techniques described in this paper can be used in (or fertilized by) techniques of other domains. In particular, there are obvious similarities between test generation and controler synthesis [18], where a controler has to be derived from the specification of a program with respect to a control plan.

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)*, pp 179-196, 1999.
2. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems sompositionally and automatically. In *Computer-Aided Verification (CAV'98)*, LNCS 1427, pp 319-331, 1998.

3. G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal* 6(6), pp 387-405, 1991.
4. M. Bozga, J-C. Fernandez, and L. Ghirvu. Using static analysis to improve automatic test generation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, LNCS 1785, pp 235-250, 2000.
5. E. Brinskma. A theory for the derivation of tests. In *Protocol Specification, Testing, and Verification (FORTE/PSTV'98)*, pp 63-74, 1998.
6. J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29 (1-2): 123-146, 1997.
7. N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2): 157-185, 1997.
8. L. Helmink, M. P. A. Sellink, and F. Vaandrager. Proof-checking a data link protocol. In *Types for Proofs and Programs, International Workshop (TYPES'94)*, LNCS 806, pp 127-165, 1994.
9. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer* 1:110-122, 1997.
10. C.A.R. Hoare. Communicating sequential processes. *Prentice Hall International Series in Computer Science*, 1985.
11. ISO/IEC. International Standard 9646-1/2/3, OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework, 1992.
12. C. Jard, T. Jéron and P. Morel. Verification of test suites. To appear in *International Conference on the Testing of Communicating Systems (TestCom'00)*, 2000.
13. T. Jéron and P. Morel. Test generation derived from model-checking. In *Computer-Aided Verification (CAV'99)*, LNCS 1633, pp 108-122, 1999.
14. N. Klarlund and M. Schwartzbach. A Domain-Specific Language for Regular Sets of Strings and Trees. *IEEE Transactions On Software Engineering* 25(3):378-386, 1999.
15. N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 3(2), 1999.
16. J. Tretmans. Test generation with inputs, outputs and quiescence. *Software - Concepts and Tools*, 17(3):103-120, 1996.
17. S. Owre, J. Rusby, N. Shankar, and F. von Henke. Formal verification of fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2): 107-125, 1995.
18. P.J. Ramadge, and W.M. Wonham. Supervisory Control of a Class of Discrete-Event Processes. *SIAM Journal of Control and Optimization*, 25(1):206-230, 1987.
19. S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985.
20. R. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the ACM*, 26(2): 351-360, 1979.
21. International Telecommunications Union. B-ISDN SAAL Service-Specific Connection-Oriented Protocol. *ITU-T Recommendation Q.2110*, 1994.
22. F. Tip. A survey of program slicing techniques. Technical Report CS-R9438, Centrum voor Wiskunde en InformatIca, 1994.