MODEL-BASED TEST SELECTION FOR INFINITE STATE REACTIVE SYSTEMS [*]

Thierry Jéron
Irisa/Inria Rennes, Campus de Beaulieu, 35042 Rennes, France
Thierry.Jeron@irisa.fr

Abstract We address the problem of off-line selection of test cases for testing the conformance of a black-box implementation with respect to a specification of a reactive systems. Efficient solutions to this problem have been proposed in the context of finite-state models, based on the **ioco** conformance testing theory. We extend them in the context of infinite state specifications, modelled as automata extended with variables. We consider the selection of test cases according to test purposes describing abstract scenarios that one wants to test. The selection of program test cases then consists in syntactical transformations of the specification model, using approximate analysis.

1. INTRODUCTION AND MOTIVATION

Testing is the most used validation technique to assess the correctness of reactive systems. For more than a decade, *model-based testing* (see e.g. [1]) advocates the use of models to formalize this validation activity. The formalization relies on precise models of specifications, implementations and test cases, a formal definition of correctness, required properties of test cases with respect to correctness, and test generation algorithms.

In this paper we address the generation of test cases in the framework of conformance testing of reactive systems [6]. Conformance testing consists in checking that a black-box implementation of a system, only known by its interactions with the environment, behaves correctly with respect to its specification. Conformance testing then relies on experimenting the system with test cases, with the objective of detecting some faults with respect to the specification's external behaviour.

We consider models of reactive systems, called Input/Output Symbolic Transition Systems (ioSTS), which are automata extended with variables, with dis-

[*] This paper is partly based on [11, 8, 12].

tinguished input and output actions, and corresponding to reactive programs without recursion. Their semantics can be defined in terms of infinite state Input/Output Labelled Transition Systems (ioLTS). For ioLTS, the **ioco** testing theory [13] defines conformance as a partial inclusion of external behaviours (suspension traces) of the implementation in those of the specification. Several research works have considered this testing theory and propose test generation algorithms. We focus on off-line test selection where a test case is built from a specification and a test purpose (representing abstract behaviours one wants to test), and further executed on the implementation. Test cases are built directly from the ioSTS model rather than constructing test cases from the enumerated ioLTS semantic model. This construction relies on syntactic transformations of the specification model, guided by an approximate analysis of the set of states co-reachable from a set of final state.

2. ioSTS: A MODEL OF REACTIVE SYSTEMS

Syntax of the ioSTS model. We propose a model called ioSTS for *Input/Output Symbolic Transition Systems*. It extends labelled transition systems for modelling imperative programs without recursion and communicating with their environment. An ioSTS is made of variables, input and output actions carrying communication parameters carried by actions, guards and assignments. In ioSTS, a data d (variable or communication parameter) has a type $type(d)$, with values in $Dom(d)$. For a data set $B = \{d_1, \ldots, d_n\}$, we note $Dom(B) = Dom(d_1) \times \cdots Dom(d_n)$. A predicate ϕ (e.g. a guard) on a data set B defines the subset of vectors in $Dom(B)$ satisfying ϕ.

DEFINITION 1 (ioSTS) *An input/output symbolic transition system (*ioSTS*) is a tuple $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ where*

- $D = V \cup P$ *is a finite set of* data *partitionned into* variables V *and* communication parameters P. *We note* $\mathcal{V} = Dom(V)$ *and* $\Pi = Dom(P)$.
- Θ *called the* initial condition *is a predicate on variables* V.
- L *is a finite set of* locations, *with* $l^0 \in L$ *the* initial location.
- $\Sigma = \Sigma^? \cup \Sigma^!$ *is the* finite alphabet of actions *partitionned into* inputs $\Sigma^?$ *and* outputs $\Sigma^!$ [1] *. An action* $a \in \Sigma$ *is characterized by its* signature *sig*$(a) = \langle p_1, \ldots, p_k \rangle \in P^k$ *specifying types of communication parameters carried by the action* a. *We note* $\Pi_a = Dom($*sig*$(a))$.
- $\mathcal{T}$ *is a finite set of* symbolic transitions. *A transition is a tuple* $t = \langle l, a, G, A, l' \rangle$ *defined by: its* origin *and* destination *locations* l *and* $l' \in L$*; an* action $a \in \Sigma$*; a* guard G *is a predicate on* $V \cup$ *sig*(a)*; an* assignment A*, of the form* $(x := A^x)_{x \in V}$ *such that, for each* $x \in V$*,* A^x *is an*

[1] The general model also considers internal actions

expression on $V \cup sig(a)$ defining the evolution of variables[2]*. We note Id_V the identity assignment $(x := x)_{x \in V}$.*

Semantics of ioSTS. The semantics of $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma, \mathcal{T} \rangle$ is an input/ouput labelled transition system (ioLTS) $[\![\mathcal{M}]\!] = \langle Q, Q^0, \Lambda, \rightarrow \rangle$, where:

- $Q = L \times \mathcal{V}$ is the set of states and $Q^0 = l^0 \times \Theta$ its subset of initial states;
- $\Lambda = \Lambda^? \cup \Lambda^!$ s.t. for $\# \in \{?, !\}$, $\Lambda^\# = \{\langle a, \pi \rangle | a \in \Sigma^\#, \pi \in \Pi_a\}$ is the set of *valued actions* partitionned into *valued inputs* $\Lambda^?$, and *outputs* $\Lambda^!$,
- $\rightarrow \subseteq Q \times \Lambda \times Q$ is the smallest relation defined by the following rule:

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

Intuitively, the ioLTS semantics of an ioSTS enumerates all possible *states* (pairs $q = \langle l, \nu \rangle$ composed of a location and the vector of values of variables) and *valued actions* (pairs $\alpha = \langle a, \pi \rangle$ composed of an action and the vector of values of its communication parameters) between states. The rule means that in a state $\langle l, \nu \rangle$, a transition $t = \langle l, a, G, A, l' \rangle$ is fireable if there exists a valuation π of $sig(a)$ such that G evaluates to $true$ for ν and π. The system then moves from with the action $\langle a, \pi \rangle$ to a state $\langle l', \nu' \rangle$ where ν' is the new valuation of variables obtained from ν and π by the assignment A.

As usual for ioLTS, we note $q \xrightarrow{\alpha} q'$ for $(q, a, q') \in \rightarrow$. For a sub-alphabet $\Lambda' \subseteq \Lambda$, we say that M is Λ'*-complete* in a state q if $\forall \alpha \in \Lambda' : q \xrightarrow{\alpha}$. An ioSTS is *deterministic* if Θ has a unique solution and in each location l, for all action a, for all pairs of transitions starting in l and carrying a, the conjonction of their guards is empty.

A *run* of an ioSTS $\mathcal{M}$ is an alternate sequence of states and valued actions $\rho = q_0 \alpha_0 q_1 \ldots \alpha_{n-1} q_n \in Q^0.(\Lambda.Q)^*$ s. t. $\forall i, q_i \xrightarrow{\alpha_i} q_{i+1}$. ρ is *accepted in* $F \subseteq Q$ if $q_n \in F$. *Runs*$(\mathcal{M})$ (resp. *Runs*$_F(\mathcal{M})$) denotes the set of runs (resp. accepted runs in F) of $\mathcal{M}$. When modelling the testing activity, we need to abstract away states which are not observable from the environment. A *trace* of a run $\rho \in$ *Runs*$(\mathcal{M})$ is the projection $proj_\Lambda(\rho)$ of ρ on actions.b *Traces*$(\mathcal{M}) \triangleq proj_\Lambda(runs\mathcal{M})$ denotes the set of traces of $\mathcal{M}$ and *Traces*$_F(\mathcal{M}) \triangleq proj_\Lambda($*Runs*$_F(\mathcal{M}))$ is the set of traces of runs accepted in F.

Visble behaviour for testing. During conformance testing, the tester stimulates inputs of the system under test, and observes not only its outputs, but also its *quiescences* (absence of output) using timers, assuming that timeout values are large enough such that, if a timeout occurs, the system is indeed

[2] The scope of parameters is limited to one transition

quiescent. The tester should be able to distinguish between specified and unspecified quiescence. But as trace semantics does not preserve quiescence in general, possible quiescence should be made explicit on the specification by a transformation called *suspension* [13]. This consists in adding a self-loop labelled with a new output δ in each quiescent state. We define suspension for ioSTS as follows. For an ioSTS $\mathcal{M} = \langle D, \Theta, L, l^0, \Sigma = \Sigma^! \cup \Sigma^?, \mathcal{T} \rangle$, the *suspension* of $\mathcal{M}$ is the ioSTS $\Delta(\mathcal{M}) = \langle D, \Theta, L, l^0, \Sigma^\delta = (\Sigma^! \cup \{\delta\}) \cup \Sigma^?, \mathcal{T}_\delta, \rangle$ with $\mathcal{T}_\delta = \mathcal{T} \cup \{\langle l, \delta, G_{\delta,l} Id_V, l \rangle \mid l \in L\}$ and

$$G_{\delta,l} = \neg \bigvee_{\langle l,a,G,A,l' \rangle \in \mathcal{T},\ a \in \Sigma^!} \exists \pi \in \Pi_a . G(a, \pi)$$

For an ioSTS $\mathcal{M}$ modelling a system, the behaviour considered for testing is then *STraces*$(\mathcal{M}) \triangleq$ *Traces*$(\Delta(\mathcal{M}))$.

3. CONFORMANCE TESTING THEORY

We now reformulate the **ioco** testing theory from [13]. It mainly consits in defining models for specifications, implementations and test cases, defining conformance, test executions and verdicts.

Conformance relation. We assume that the specification is an ioSTS $\mathcal{S}$, and that the behaviour of the unknown implementation could be modelled by an (non-deterministic) ioLTS $I = \langle Q_I, Q_I^0, \Lambda^! \cup \Lambda^?, \rightarrow_I \rangle$ with same interface. We also assume that I is $\Lambda^?$-complete [3]. The conformance relation then defines the set of correct implementation models:

I **ioco** $\mathcal{S} \triangleq$ *STraces*$_{\neg ioco}(\mathcal{S}) \cap$ *STraces*$(I) = \emptyset$
where *STraces*$_{\neg ioco}(\mathcal{S}) =$ *STraces*$(\mathcal{S}) \cdot (\Lambda^! \cup \{\delta\}) \setminus$ *STraces*$(\mathcal{S})$.

STraces$_{\neg ioco}(\mathcal{S})$ exactly represents the set of non-conformant behaviours: I is non-conformant as soon as it may exhibit a suspension trace of $\mathcal{S}$ prolongated with an unspecified outputs or quiescence. Interestingly, our formulation of **ioco** explicits the fact that conformance is a safety property of I: conformance is violated if one exhibits a finite trace of I in *STraces*$_{\neg ioco}(\mathcal{S})$. If I was known, verifying conformance would then amount to building a deterministic *non-conformance observer can*$(\mathcal{S})$ equiped with a set of states **Fail** such that *Traces*$_{\textbf{Fail}}$(*can*$(\mathcal{S})) =$ *STraces*$_{\neg ioco}(\mathcal{S})$, computing the synchronous product of I and *can*$(\mathcal{S})$ and checking whether **Fail** is reachable.

However, as I is unknown, one can only experiment it with selected test cases, providing inputs and checking that outputs and quiescences of I are specified in $\mathcal{S}$. This entails that, except in simple cases, conformance cannot be proved by testing, only non-conformance witnesses can be exhibited.

[3] This ensures that the composition of I with a test case TC never blocks because of non-implemented inputs.

Test cases, test executions and verdicts. In our modelling framework, we aim at building test cases in the form of ioSTSȦ test case for the specification ioSTS $\mathcal{S}$ is a deterministic ioSTS $\mathcal{TC} = \langle D = V \cup P, \Theta_{TC}, L_{TC}, l^0_{TC}, \Sigma_{TC} = \Sigma^!_{TC} \cup \Sigma^?_{TC}, \mathcal{T}_{TC}\rangle$ with $\Sigma^!_{TC} = \Sigma^?$ and $\Sigma^?_{TC} = \Sigma^! \cup \{\delta\}$ (actions are mirrored w.r.t. $\mathcal{S}$) with semantics $[\![TC]\!] = TC$. $\mathcal{TC}$ is equipped with a collection of sets of verdict locations $\mathsf{Verdict}$ partitionned into **Fail** (meaning rejection), **Pass** (meaning that targetted behaviours have been reached) and **Inconc** (meaning that targetted behaviours cannot been reached anymore). We also call $\mathsf{Verdict}$ the collection of sets of states of TC where the location is in $\mathsf{Verdict}$. We assume that $\mathsf{Verdict}$ states are trap states (with no transitions) and that all states except $\mathsf{Verdict}$ ones are $\Lambda^?_{TC}$-complete.

We model the execution of a test case $\mathcal{TC}$ on an implementation I by the *parallel composition* of $TC = [\![\mathcal{TC}]\!]$ with $\Delta(I)$ (quiescences of I are observed) with synchronization on common actions. Let $\Delta(I) = \langle Q_{\text{I}}, Q^0_{\text{I}}, \Lambda^! \cup \{\delta\} \cup \Lambda^?, \rightarrow_{\Delta(I)}\rangle$ and $TC = \langle Q_{\text{TC}}, q^0_{\text{TC}}, \Lambda^? \cup \Lambda^! \cup \{\delta\}, \rightarrow_{\text{TC}}\rangle$, $\Delta(I) \| TC$ is the ioLTS $(Q_{\text{I}} \times Q_{\text{TC}}, Q^0_{\text{I}} \times \{q^0_{\text{TC}}\}, \Lambda^! \cup \{\delta\} \cup \Lambda^?, \rightarrow_{\Delta(I)\|TC})$ where, for $\alpha \in \Lambda^! \cup \{\delta\} \cup \Lambda^?$, $(q_1, q'_1) \xrightarrow{\alpha}_{\Delta(I)\|TC} (q_2, q'_2)$ iff $q_1 \xrightarrow{\alpha}_{\Delta(I)} q_2$ and $q'_1 \xrightarrow{\alpha}_{\text{TC}} q'_2$.

The possible rejection of I by TC is defined by the fact that $\Delta(I)\|TC$ may lead to **Fail** in TC: TC *mayfail* $I \triangleq$ *Traces*$_{Q_{\text{I}} \times \textbf{Fail}}(\Delta(I)\|TC) \neq \emptyset$ which is equivalent to *Traces*$(\Delta(I)) \cap$ *Traces*$_{\textbf{Fail}}(TC) \neq \emptyset$.

Now, test generation algorithms should produce test cases with properties relating rejection with non-conformance. Formally, let TS be a set of test cases. We say that TS is *complete* if it is both *correct* and *exhaustive* where:

TS is *correct* $\triangleq \forall I, (I$ **ioco** $S \implies \forall TC \in TS, \neg TC$ *mayfail* $I)$.

i.e. only non-conformant implementations can be rejected by a test case in TS.

TS is *exhaustive* $\triangleq \forall I, (\neg(I$ **ioco** $S) \implies \exists TC \in TS, TC$ *mayfail* $I)$.

i.e. every non-conformant implementation can be rejected by a test case in TS.

Using the definitions of I **ioco** S and TC *mayfail* I, one can now prove:

TS is correct $\iff \bigcup_{TC \in TS}$ *Traces*$_{\textbf{Fail}}(TC) \subseteq$ *STraces*$_{\neg ioco}(S)$ and
TS is exhaustive $\iff \bigcup_{TC \in TS}$ *Traces*$_{\textbf{Fail}}(TC) \supseteq$ *STraces*$_{\neg ioco}(S)$.

Interestingly, if one considers the non-conformance observer *can*(S) as a test case (by mirroring its actions), as *Traces*$_{\textbf{Fail}}($*can*$(S)) =$ *STraces*$_{\neg ioco}(S)$, it immediately follows that the singleton $\{$*can*$(S)\}$ is a complete test suite, in some sense the most general testing process for conformance w.r.t. S. Moreover, all correct test cases should be sub-observers of *can*(S), while an exhaustive test suite must reject all implementations rejected by *can*(S). In fact, all test generation algorithms for **ioco** producing complete test suites can be understood as producing an infinite number of unfoldings of *can*(S). But in practice, *can*(S) cannot be used directly as a test case. One wants to select individual test cases focussed on some particular behaviour. Selection of a sound test suite will then be based on the selection of sub-behaviours of *can*(S). The selection algorithm should remain *limit exhaustive*: for any non-conformant implementation, one could generate a test case that could reject it.

4. TEST SELECTION FOR ioSTS

At the end of the section an example illustrates the principles of test selection. As explained previously, test selection consists in extracting a sub-observer of the non-conformance observer *can*$(\mathcal{S})$. The first operation consists in constructing the ioSTS *can*$(\mathcal{S})$ such that $[\![\textit{can}(\mathcal{S})]\!] = \textit{can}([\![S]\!])$. When $\mathcal{S}$ is deterministic (or determinized) [4], this is easily done by adding, in every location l and for all output a, a new transition $\langle l, a, G_{\textbf{Fail}}, Id_V, \textbf{Fail}\rangle$ where $G_{\textbf{Fail}} = \neg \bigvee_{\langle l,a,G,A,l'\rangle \in \mathcal{T}} G$ and **Fail** is a new location.

In this paper we focus on the selection of test cases by test purposes describing some abstract behaviour one wants to test. We define test purposes as ioSTS equipped with a set of accepting locations playing the role of a non intrusive observer. Its set of variables consists of its set of *proper* variables and the set of variables of the specification that it may observe, but cannot modify.

A *Test Purpose* for a specification ioSTS $\mathcal{S} = \langle V \cup P, \Theta, L, l^0, \Sigma, \mathcal{T}\rangle$ is an ioSTS $\mathcal{TP} = \langle V_p \cup V \cup P, \Theta_{TP}, L_{TP}, l^0_{TP}, \Sigma \cup \{\delta\}, \mathcal{T}_{TP}\rangle$ equipped with a distinguished set of locations *Accept* $\subseteq L_{TP}$. We assume that $\mathcal{TP}$ is complete in all locations except *Accept* (for each action a the conjunction of guards of all transitions carrying a is $true$) and cannot modify variables in V (assignments to these variables are the identity assignment).

The role of the test purpose is to select suspension traces of *can*$(\mathcal{S})$ accepted by $\mathcal{TP}$. The usual way to define this intersection for ioLTS is to perform a *synchronous product*. We define a corresponding syntactic operation on ioSTS where transitions with same actions synchronize on the conjunction of their guards. Formally, the *synchronous product* of *can*$(\mathcal{S}) = \langle V \cup P, \Theta, L \cup \{\textbf{Fail}\}, l^0, \Sigma, \mathcal{T}_c\rangle$ and $\mathcal{TP} = \langle V \cup V_p \cup P, \Theta_{TP}, L_{TP}, q^0_{TP}, \Sigma \cup \{\delta\}, \mathcal{T}_{TP}\rangle$ is the ioSTS *can*$(\mathcal{S}) \times \mathcal{TP} = \langle V \cup V_p \cup P, \Theta \wedge \Theta_{TP}, L \cup \{\textbf{Fail}\} \times L_{TP}, (l^0, l^0_{TP}), \Sigma, \mathcal{T}'\rangle$ where $\langle (l_1, l_2), a, G_1 \wedge G_2, A_1; A_2, (l'_1, l'_2)\rangle \in T'$ if and only if $\langle l_1, a, G_1, A_1, l'_1\rangle \in T_c \wedge \langle l_2, a, G_2, A_2, l'_2\rangle \in T_{TP}$ and $A_1; A_2$ is the sequential composition of assignments affecting disjoint sets of variables.

$\mathcal{TP}$ is non-intrusive, thus *Traces*(*can*$(\mathcal{S}) \times \mathcal{TP}$) = *Traces*(*can*$(\mathcal{S})$) and *Traces*$_{\textbf{Fail} \times L_{TP}}$(*can*$(\mathcal{S}) \times \mathcal{TP}$) = *Traces*$_{\textbf{Fail}}$(*can*$(\mathcal{S})$) = *STraces*$_{\neg ioco}(\mathcal{S})$ meaning that *can*$(\mathcal{S}) \times \mathcal{TP}$ is a non-conformance observer. We also have *Traces*$_{L \times \textit{Accept}}$(*can*$(\mathcal{S}) \times \mathcal{TP}$) = *STraces*$(\mathcal{S}) \cap$ *Traces*$_{\textit{Accept}}(\mathcal{TP})$ meaning that *can*$(\mathcal{S}) \times \mathcal{TP}$ is an observer of traces accepted by $\mathcal{TP}$, restricted to

[4] For the sake of simplicity, we restrict here to deterministic ioSTSspecifications. Non-deterministic ioSTScan be handled at least for a sub-class of ioSTS where non-determinism can be solved with bounded lookahead [9].

suspension traces of $\mathcal{S}$. Thus, depending on the considered distinguished locations **Fail** × L_{TP} or $L \times$ *Accept*, the ioSTS observer *can*$(\mathcal{S}) \times \mathcal{TP}$ can play two different roles.

But *can*$(\mathcal{S}) \times \mathcal{TP}$ is just an unfolding of *can*$(\mathcal{S})$ from which we now need to select traces by focussing on traces accepted in *Accept*. Ideally, we want to select exactly $\mathit{STraces}(S) \cap \mathit{Traces}_{\mathit{Accept}}(\mathcal{TP})$, plus unspecified outputs prolongating these traces in $\mathit{STraces}_{\neg ioco}(S)$. However we consider *non-controllable* system models, for which an input does not determine an unique output. After a trace, the tester should then consider all possible outputs: those from which *Accept* is reachable or **Fail** is reached, but also those after which *Accept* is not reachable anymore. In this last case, we want to detect this divergence as soon as possible, and set the *Inconc* verdict. This reduces to the problem of computing the set $coreach(\mathit{Accept})$ of states co-reachable from $L \times$ *Accept*. This is easy for finite state systems and solved with graph algorithms. However, this problem is undecidable for $ioSTS$ models.

Our solution, implemented in the STG tool [3], consists in computing an over-approximation $coreach^{\alpha} \supseteq coreach(\mathit{Accept})$ represented by a predicate. This is provided by an interface with the NBac tool [7] using abstract interpretation [4]. For any assignment A of a transition $t \in \mathcal{T}'$, we also compute an over-approximation of the set of values for variables and parameters allowing to stay in $coreach^{\alpha}$ when firing t, noted $pre^{\alpha}(A)(coreach^{\alpha})$. In other words it is a *necessary condition* to go in $coreach(\mathit{Accept})$ by t. Its negation is thus a *sufficient condition* to leave $coreach(\mathit{Accept})$. The selection of a test case $\mathcal{TC}$ from *can*$(\mathcal{S}) \times \mathcal{TP}$ then consists in mirroring actions, transforming *Accept* locations into **Pass** and modifying the transitions in $\mathcal{T}'$ into $\mathcal{T}_{TC}$ with the two following rules:

$$\text{Keep: } \frac{\langle l, a, G, A, l' \rangle \in \mathcal{T}'}{\langle l, a, G \wedge pre^{\alpha}(A)(coreach^{\alpha}), A, l' \rangle \in \mathcal{T}_{TC}}$$

$$\text{Inconc: } \frac{\langle l, a, G, A, l' \rangle \in \mathcal{T}' \quad a \in \Sigma_!}{\langle l, a, G \wedge \neg pre^{\alpha}(A)(coreach^{\alpha}), A, \mathit{Inconc} \rangle \in \mathcal{T}_{TC}}$$

The effect of rule (Keep) is to discards all (semantic) transitions labeled by a (controllable) input that *certainly* exit $coreach(\mathit{Accept})$, and rule (Inconc) "redirects" to a new location *Inconc* all transitions labelled by an (uncontrollable) output that *certainly* exit $coreach(\mathit{Accept})$. The test case can be further simplified (without modifying its semantics) with an over-approximation of its reachable states $reach^{\alpha}(\Theta \wedge \Theta_{TP})$. Notice that these analysis can be improved using the dynamic partitionning facility of NBac, allowing to separate locations with respect to the analysis.

Test case properties. As *can*(S) is sound and is not modified by the synchronous product and selection, all test cases are sound. Limit exhaustiveness comes from the following construction: for any non-conformant implementation, there exists a trace $\sigma.a$ in $TracesI \cap STraces_{\neg ioco}(\mathcal{S})$. It then suffices to construct a test purpose $\mathcal{TP}$ such that the trace $\sigma.a$ leads to *Accept*. The test case obtained from $\mathcal{S}$ and $\mathcal{TP}$ then may reject I.

What is lost by the over-approximation of $coreach(Accept)$, compared with an (hypothetical) exact computation, is the hability to detect infeasible traces to *Accept* as soon as this happens. Of course, the more precise is the approximation, the sooner is the detection [8].

Simple example.

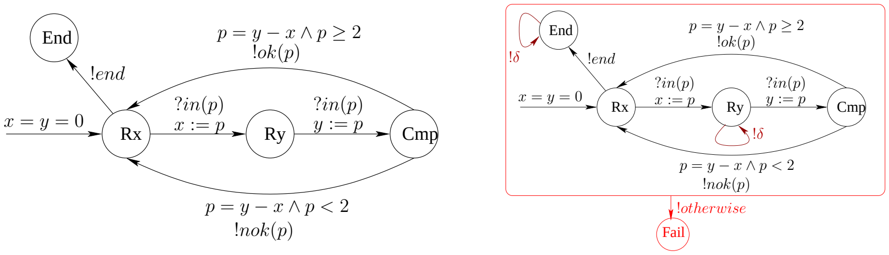

Figure 1. (Left) ioSTS $\mathcal{S}$ reading and comparing two values. (Right) canonical tester *can*$(\mathcal{S})$.

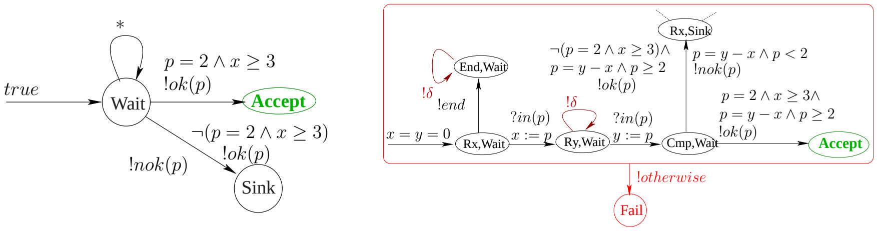

Figure 2. (Left) ioSTS test purpose $\mathcal{TP}$. (Right) Synchronous product *can*$(\mathcal{S}) \times \mathcal{TP}$.

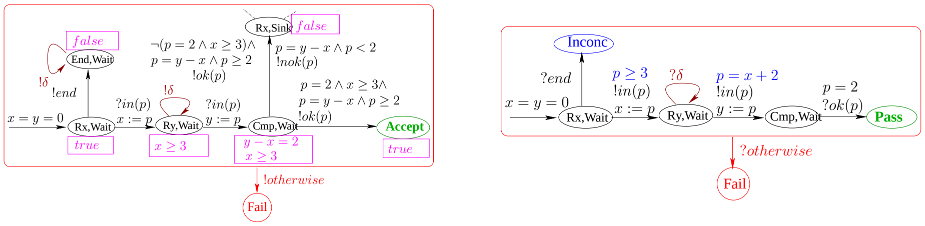

Figure 3. (Left) Computation of $coreach^{\alpha}$. (Right) Resulting test case $\mathcal{TC}$.

Test execution:. Test cases produced so far are ioSTS. In particular the values of communication parameters of test cases are not instanciated. During test execution, values of communication parameters have to be chosen for outputs of the test cases, among values satisfying the guard (e.g. $p = 5$ for $p \geq 3$ in the example). This is simply done by a constraint solver. Conversely, when receiving an input from the implementation, as the test case is input complete and deterministic, one has to check which transition can be fired, by checking the guard with the value of the received communication parameter (e.g. go to **Pass** is $p = 2$, and $Fail$ otherwise).

5. CONCLUSION AND PERSPECTIVES

There is still very few research work on model-based test generation which are able to cope with models containing both control and data without enumerating data values. Some exist however in the context of the **ioco** testing theory. In [10] the authors use selection hypotheses combined with operation unfolding for algebraic data types and predicate resolution to produce test cases from Lotos specifications. The paper [5] lifts the **ioco** theory from LTS to STS (Symbolic Transition Systems) but addresses the on-line test generation problem where next actions of test cases are computed during execution. In [2] the authors start with a specification model similar to ioSTS, abstract the model in a finite state one, use our TGV tool to generate test cases in the abstract domain, and then solve a constraint programming problem in the concrete model.

In the present paper, we have presented an approach to the off-line generation of test cases from specification models with control and data (ioSTS) and test purposes in the same model. The main advantage of this test generation technique is to avoid the state explosion problem due to the enumeration of data values. Test generation reduces to syntactic operations on these models and an over-approximate analysis of the co-reachable states to a target location. Test cases are generated in the form of ioSTS, thus representing uninstanciated test programs. During execution of test cases on the implementation, constraint solving is used to choose output data values. For simplicity, the theory exposed in this paper is retricted to deterministic specifications. However non-determnistic specifications can be taken into account if ioSTS have no loops of internal actions and have bounded lookahead.

Among the perspectives of this work, we expect to consider more powerful models of systems with features such as time, recursion and concurrency. For test generation, one problems to address in these models is partial observability, which entails the identification of determinizable sub-classes corresponding to applications. We also think that the ideas of this technique can also be used in other contexts, in particular for structural white box testing where test cases are generated from the source code of the system. One of the main problems of these techniques which is to avoid infeasible paths, could be partly solved by techniques similar to ours.

ACKNOWLEDGMENTS

I wish to thank the Organizing Committee of DIPES for this invitation, as well as all my colleagues who participated in this work.

REFERENCES

[1] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems: Advanced Lectures*, volume 3472 of *LNCS*. Springer, 2005.

[2] J. R. Calamé, N. Ioustinova, J. van de Pol, and N. Sidorova. Data abstraction and constraint solving for conformance testing. In *Proc. of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005), Taipei, Taiwan*, pages 541–548. IEEE Computer Society, December 2005.

[3] 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)*, volume 2280 of *LNCS*, pages 470–475, Grenoble, France, avril 2002.

[4] P. Cousot and R. Cousot. Abstract intrepretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[5] L. Frantzen, J. Tretmans, and T. Willemse. Test generation based on symbolic specifications. In *4th International Workshop on Formal Approaches to Testing of Software (FATES 2004), Linz, Austria*, volume 3395 of *LNCS*. Springer-Verlag, 2004.

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

[7] B. Jeannet. Dynamic partitioning in linear relation analysis. *Formal Methods in System Design*, 23(1):5–37, 2003.

[8] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *11th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05), Edinburgh, Scottland*, volume 3440 of *LNCS*. Springer, april 2005.

[9] T. Jéron, H. Marchand, and V. Rusu. Symbolic determinisation of extended automata. In *4th IFIP International Conference on Theoretical Computer Science, 2006, Santiago, Chile*. SSBM (Springer Science and Business Media), August 2006.

[10] G. Lestiennes and M.-C. Gaudel. Testing processes from formal specifications with inputs, outputs and data types. In *13th International Symposium on Software Reliability Engineering (ISSRE'02), Annapolis, Maryland*. IEEE Computer Society Press, 2002.

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

[12] V. Rusu, H. Marchand, and T. Jéron. Automatic verification and conformance testing for validating safety properties of reactive systems. In John Fitzgerald, Andrzej Tarlecki, and Ian Hayes, editors, *Formal Methods 2005 (FM05)*, volume 3582 of *LNCS*. Springer, juillet 2005.

[13] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.