PUBLICATION
INTERNE
N° 1649

SYMBOLIC TEST SELECTION BASED ON APPROXIMATE ANALYSIS

BERTRAND JEANNET , THIERRY JÉRON , VLAD RUSU , ELENA ZINOVIEVA

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

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

Symbolic Test Selection based on Approximate Analysis

Bertrand Jeannet * , Thierry Jéron ** , Vlad Rusu *** , Elena Zinovieva ****

Systèmes communicants
Projet Vertecs

Publication interne n1649 — Octobre 2004 — 18 pages

Abstract: This paper addresses the problem of generating symbolic test cases for testing the conformance of a black-box implementation with respect to a specification, in the context of reactive systems. The challenge we consider is the selection of test cases according to a test purpose, which is here a set of scenarii of interest that one wants to observe during test execution. Because of the interactions that occur between the test case and the implementation, test execution can be seen as a game involving two players, in which the test case attempts to satisfy the test purpose.

Efficient solutions to this problem have been proposed in the context of finite-state models, based on the use of fixpoint computations. We extend them in the context of infinite-state symbolic models, by showing how approximate fixpoint computations can be used in a conservative way. The second contribution we provide is the formalization of a quality criterium for test cases, and a result relating the quality of a generated test case to the approximations used in the selection algorithm.

Key-words: conformance testing, symbolic test selection, reactive systems, abstract interpretation

(Résumé : tsvp)

* Bertrand.jeannet@irisa.fr
** Thierry.Jeron@irisa.fr
*** Vlad.Rusu@irisa.fr
**** leroux_elena@yahoo.fr

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

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

Génération de test symbolique fondée sur des analyses approchées

Résumé : Cet article traite de la génération de cas de test symboliques pour tester la conformité d'une implémentation boîte-noire vis-à-vis d'une spécification, dans le contexte des systèmes réactifs. Le problème que nous considérons est la sélection de cas de tests à partir d'un objectif de test, qui est ici un ensemble de scénarios intéressants que l'on veut observer au cours de l'exécution du test. En raison des interactions qui se produisent entre le cas de test et l'implémentation, l'exécution du test peut être vu comme un jeu impliquant deux joueurs, dans lequel le cas de test tente de satisfaire l'objectif de test.

Des solutions efficaces à ce problème ont été proposées dans le cadre de modèles à états finis, fondé sur l'utilisation de calculs de point fixe. Nous les étendons au cas de modèles à états infinis et symboliques, en montrant comment des calculs de point fixe approchés peuvent être utilisés de manière conservative. La seconde contribution que nous proposons est la formalisation d'un critère de qualité pour les cas de test, et un résultat reliant la qualité d'un cas de test généré par notre méthode aux approximations utilisées dans l'algorithme de sélection.

Mots clés : test de conformité, sélection de tests symboliques, systèmes réactifs, interprétation abstraite

1 Introduction

We address the generation of test cases in the framework of conformance testing of reactive systems [1]. In this context, a Test Case (TC) is a program run in parallel with a black-box Implementation Under Test (IUT), that stimulates the IUT by repeatedly sending inputs and checking that the observed outputs of the IUT are in conformance with a given specification S. In case the IUT exhibits a conformance error, the execution is immediately interrupted. However, in addition to checking the conformance of the IUT, the goal of the test case is to guide the parallel execution towards the satisfaction of a test purpose, typically a set of scenarii of interest. Because of this second feature, test execution can be seen as a game between two programs, the test case and the IUT. The test case wins the game if it succeeds to make the parallel execution realize one of the interesting scenarii specified by the test purpose; the IUT wins if the execution cannot be extended any more to one that realizes an interesting scenario, or in other words if the execution does not belong any more to the prefixes of the interesting scenarii. If a conformance error is detected, the game terminates with a tie. Note that in present discussion, we separate the two aspects of detecting conformance error and of guiding the IUT toward the satisfaction of the test purpose, and focus on the latter. If a conformance error is detected, the game is interrupted with the corresponding *Fail* verdict.

In this context, The *test selection* problem consists in finding a strategy that minimizes the likehood for the test case to lose the game. Indeed, it is generally not possible to ensure that the test case wins, because IUT is unknown: it is a black-box program that may behave in a non-controllable way, and which is only assumed to be conformant for the game aspect. Focusing only on the cases where there exists a winning strategy would be too restrictive. This problem has been previously adressed in a context where the specifications, the test cases and the test purposes are modeled with finite Labelled Transition Systems (LTS) [11]. Finding a suitable strategy for the test case is decomposed in two steps:

1. One first performs an *off-line* selection of a Test Case that detects when the game is lost by the tester and stops the execution in this case. This is done by static analysis of the specification S and the test purpose TP.

2. Then, during the execution of the obtained test case in parallel with the IUT, one performs an *on-line* selection of the inputs that the test case sends to the IUT. This on-line selection is based on the history of the current execution.

A previous paper [18] extends these principles and algorithmic methods to the case where specifications, test purposes and test cases are modeled with Input-Output Symbolic Transition Systems (ioSTS), which are automata that operate on variables (integers, booleans, aggregate types, ...) and communicate with the environment by means of input and output actions carrying parameters. However, the off-line test selection algorithm presented in [18] takes into account only the finite control structure of ioSTS and ignores the behavior of the variables: hence, the static analysis used for the off-line selection (Step 1) is not very precise. It cannot be made exact either, because of undecidability problems, but its precision can be improved.

Contributions. The contributions of this paper are twofold. First we describe a general test selection method parameterized by an approximate analysis, in the context of Input-Output Symbolic Transition Systems. We suggest a more precise approximate analysis than the one proposed in [18], based on Abstract Interpretation techniques [6]. We show that the test cases obtained by this method are sound and emit only correct verdicts.

Second, we investigate the effect of the approximations of the analysis from the point of view of test execution as a game: in which way do they degrade the winning capabilities of the obtained test case? This leads us to define an accuracy ordering between test cases, to formalize the notion of optimal test case, and to compare the test cases generated by our method using these notions.

Context and Related work.

Conformance testing. Testing is the most used validation technique to assess the quality of software systems. Among the aspects of software that can be tested, *e.g.*, functionality, performance, timing, robustness, etc, we focus here on conformance testing and specialize it to reactive systems. In this approach, the software is specified in a behavioral model which serves both as a basis for test generation and for verdicts assignment. Testing theories based on models such as automata associated to fault models (see *e.g.* the survey [15]), or labelled transition systems with conformance relations (see *e.g.* [20]) are now well understood. Test generation algorithms have been designed based on these theories, and tools like TorX [2], TGV [11], Gotcha [3], have been developed and used on industrial-size systems.

Test selection. The test selection problem consits in choosing some test cases among many possible, according to a given criterion. Most approaches are based on variants of classical control and data-flow coverage criteria [10, 3], while others focus on specific functionalities using test purposes [8]. Although this is not always made explicit, test generation typically relies on reachability and coreachability analyses [11] based on post- and pre- predicate transformers.

Symbolic models. Many of the existing test generation algorithms and tools operate on variants of labeled transition systems (LTS). High-level specifications (written in languages such as SDL, Lotos, or UML) can be treated as well, via a translation (by state-space exploration) to the more basic labeled transition systems. More recently, attempts have been made in the direction of *symbolic* test generation [18, 4] which works directly on the higher-level specifications without enumerating their state-space, thus avoiding the state-space explosion problem. These models, whether they are called extended finite states machines or symbolic transition systems s are essentialy automata manipulating variables (integers, booleans, aggregate types, ...) which correspond to programs without recursive procedure calls and without memory allocation. Testing theories stay unchanged, being based on the semantics of the models in terms of (infinite) transition systems. But the algorithms should be adapted to cope with symbolic automata. In most approaches, one tries to generate instanciated test cases, *i.e.* with fixed values for input and output parameters, that exercise particular executions of the system.

Outline. In Section 2, we recall ioLTSmodel, the corresponding testing theory and the principles of test generation using test purposes. In Section 3, we define the syntax of the symbolic model of ioSTSand its ioLTS semantics. In Section 4, we propose an off-line test selection algorithm for ioSTS based on syntactical transformations and parameterized by an approximate fixpoint analysis. Section 5 describes the on-line test selection that occurs during test execution. Section 6 defines qualitative properties on tests cases concerning their ability to satisfy the test purpose, and shows how the approximations used in the off-line test generation step influence those qualities. Finally Section 7 addresses implementation issues

2 Testing with Input/Output Labeled Transition Systems

2.1 The ioLTS model

Specification languages for reactive systems can often be given a semantics in terms of labelled transition systems. For test generation, we use the following version where actions are explictly partitioned into *inputs*, which are controlled by the environment, and *outputs*, which the environment may only observe. This model also serves as a semantic model for our symbolic automata (cf. Section 3).

Definition 1 (ioLTS) *An Input/Output Labelled Transition System is a tuple* $(Q, Q_0, \Lambda, \rightarrow)$ *where* Q *is a set of states,* Q_0 *the set of initial states,* $\Lambda = \Lambda_? \cup \Lambda_!$ *is a set of actions partitionned into inputs* ($\Lambda_?$) *and outputs* ($\Lambda_!$) *and* $\rightarrow \subseteq Q \times \Lambda \times Q$ *is the transition relation.*

We write $q \xrightarrow{\alpha} q'$ in place of $(q, \alpha, q') \in \rightarrow$ and note $q \xrightarrow{\alpha}$ when $\exists q' : q \xrightarrow{\alpha} q'$.

For the sake of simplicity, we consider only *deterministic* ioLTS: the alphabet does not contain internal actions and $\forall q \in Q, q \xrightarrow{\alpha} q' \wedge q \xrightarrow{\alpha} q'' \Rightarrow q' = q''$. This is not a strong restriction: finite ioLTS can always can be transformed into deterministic ioLTS with same traces using the classical $\tau - closure$ and subset constructions.

The behaviours of ioLTS can be seen at different semantic levels: in *runs*, one considers alternate sequences of states and actions, while in *traces* the states are abstracted away. A *run* is a finite sequence $\rho = q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} q_n$ such that $q_0 \in Q_0$ and $\forall i < n, (q_i, \alpha_i, q_{i+1}) \in \rightarrow$. Its projection onto actions is the *trace* $\sigma = trace(\rho) = \alpha_0 \ldots \alpha_{n-1}$. We denote by $Runs(M) \subseteq Q_0 \cdot (\Lambda \cdot Q)^*$ the set of runs of M and by $Traces(M) \subseteq \Lambda^*$ the set of traces of M. An ioLTS M can be seen as an automaton if it is equipped with a set of marked states $X \subseteq Q$. A run $\rho = q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} q_n$ is *accepted in* X iff $q_n \in X$. We denote $Runs_X(M) \subseteq Runs(M)$ the set of runs accepted by X. Similarly, the set of accepted traces $Traces_X(M) \subseteq Traces(M)$ is obtained by projecting $Runs_X(M)$ on Λ^*. An ioLTS M is *complete in a state* q if $\forall \alpha \in \Lambda : q \xrightarrow{\alpha}$; it is *complete* if it is complete in all states. Similarly, the notion of *input*-completeness is defined by replacing Λ with $\Lambda_?$.

The *synchronous product* of two ioLTS $M = (Q, Q_0, \Lambda, \rightarrow^M)$ and $M' = (Q', Q'_0, \Lambda, \rightarrow^{M'})$ with same alphabet is the ioLTS $M \times M' = (Q \times Q', Q_0 \times Q'_0, \Lambda, \rightarrow^{M \times M'})$ where $\rightarrow^{M \times M'}$ is defined by the inference rule:

$$\frac{q_1 \xrightarrow{\alpha}^M q_2 \quad q'_1 \xrightarrow{\alpha}^{M'} q'_2}{(q_1, q'_1) \xrightarrow{\alpha}^{M \times M'} (q_2, q'_2)} \qquad \text{(Sync)}$$

As usual, we get $Traces(M \times M') = Traces(M) \cap Traces(M')$ and $Traces_{X \times X'}(M \times M') = Traces_X(M) \cap Traces_{X'}(M')$ for $X \subset Q$ and $X' \subset Q'$.

For a set of traces $W \subseteq \Lambda^*$, we denote respectively by $pref_{\leq}(W)$ and $pref_{<}(W)$ the set of prefixes (resp. strict prefixes) of W. For $X \subseteq Q$, we denote by $post(X) = \{q' \mid \exists q \in X, \exists \alpha \in \Lambda : q \xrightarrow{\alpha} q'\}$ and $pre(X) = \{q \mid \exists q' \in X, \exists \alpha \in \Lambda : q \xrightarrow{\alpha} q'\}$ the pre- and post-condition operators. The set of states *reachable* from a subset Q' of Q in M may then be defined by $reach(Q') = \text{lfp}(\lambda X.Q' \cup post(X))$ where lfp is the least fixpoint operator. Similarly, the set of states *coreachable* from a set of states Q' may be defined as $coreach(Q') = \text{lfp}(\lambda X.Q' \cup pre(X))$.

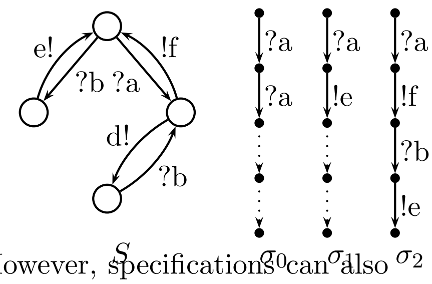
2.2 Testing Theory

The testing theory we consider is based on the notions of *specification*, *implementation*, and *conformance relation* between them [20]. Since testing is based on visible behaviors, the conformance relation is defined in terms of traces, more precisely as a partial trace inclusion. Testing is then achieved by test cases that control the implementation's inputs and observe its outputs, emitting verdicts according to the conformance relation. Moreover, we want to select interesting tests using *test purposes*, which also indicate when the test terminates.

The specification is an ioLTS $S = (Q^S, Q^S_0, \Lambda, \rightarrow^S)$. The Implementation Under Test (IUT) is also assumed to be an ioLTS $IUT = (Q^{IUT}, Q^{IUT}_0, \Lambda, \rightarrow^{IUT})$ which is unknown except for its alphabet, which is assumed to be the same as that of the specification. Moreover, it is assumed that the IUT is *input-complete*, which reflects the hypothesis that the implementation cannot refuse an input from its environment.

Definition 2 (Conformance relation) *A* trace σ is conformant to S, *denoted by* $\sigma\ conf\ S$, *iff* $pref_{\leq}(\sigma) \cap (Traces(S) \cdot \Lambda_!) \subseteq Traces(S)$. IUT is conformant to S, *denoted by* $IUT\ conf\ S$, *iff* $Traces(IUT) \cap Traces(S) \cdot \Lambda_! \subseteq Traces(S)$.

Notice that any trace of S is conformant to S.



Intuitively, IUT *conf* S if after each trace of S, IUT may emit only outputs that S can emit as well while its inputs are unconstrained. For readers familiar with ioco [20], note that *conf* can be interpreted as ioco if S makes *quiescence* (absence of output) explicit. For instance on the figure $\neg(\sigma_1 \textit{ conf } S)$, $\sigma_2 \textit{ conf } S$, and $\sigma_0 \textit{ conf } S$ as σ_0 diverges from S by an *input*.

If S is input-complete, *conf* is just the trace inclusion relation. However, specifications can also be partial: a trace belonging to S and thus conformant can be extended into one that leaves S *by an input* not expected by S at the current state; then all its extensions will be conformant. This is why we have Intuitively S specifies what is to be expected in some situations, but does not tell anything for other situations. Typically, this allows to add a new service in an implementation conformant to S without making it not conformant any more.

2.3 Test Cases and Test Purposes

A test case is an ioLTS able to interact with an implementation and to emit verdicts about the implementation's conformance with respect to the specification. In our approach, a test case is an ioLTS implementing a strategy for satisfying a given *test purpose* (typically, staying within a (finite or infinite) set of traces). The test case takes into account output choices of the specification (observable non-determinism) and anticipates incorrect outputs of the implementation.

Definition 3 (Test Case) *A* test case *for a specification S is an ioLTS $TC = (Q^{TC}, Q_0^{TC}, \Lambda^{TC}, \rightarrow^{TC})$ equipped with 3 disjoint subsets of sink states* Pass, Fail, Inconc $\subseteq Q^{TC}$ *(corresponding to the verdicts it may emit) such that*

- *its alphabet is the mirror of that of S ($\Lambda_?^{TC} = \Lambda_!^S$ and $\Lambda_!^{TC} = \Lambda_?^S$) and it is input-complete (outputs of IUT are not refused) except in verdict states;*
- *$Traces(TC) \subseteq Traces(S) \cdot (\{\epsilon\} \cup \Lambda_!)$: as soon as a conformance error cannot occur any more the test case stops (cf. Definition 2).*

The specification S contains all the information relevant to conformance, and its mirror followed by *input-completion* constitutes a test case by itself. However, such a test case is typically too large and is not focused on any part of the system. It is more interesting in practice to test what happens in the course of a given scenario (or set thereof), and if no error has been detected, to end the test successfully when the scenario is completed. This is precisely the reason for introducing *test purposes*, as in TorX [2])

Definition 4 (Test Purpose) *A* test purpose *TP for a specification S is an ioLTS $TP = (Q^{TP}, Q_0^{TP}, \Lambda, \rightarrow^{TP})$ equipped with a subset* Accept $\subseteq Q$ *of accepting states, which are sink. TP is complete except in* Accept *states. TP defines a set $ATraces(S, TP)$ of accepted traces of S which induces a set $RTraces(S, TP)$ of refused traces (traces of S that cannot be extended to accepted traces):*

$$ATraces(S, TP) = Traces_{Q^S \times \mathsf{Accept}}(S \times TP) \tag{1}$$

$$RTraces(S, TP) = Traces(S) \setminus pref_{\leq}(ATraces(S, TP)) \tag{2}$$

The completeness assumption allows not to constraint S during the parallel execution of S and TP , unless the execution trace is accepted. Observe that *both accepted and refused traces are conformant.*

More elaborate test purposes can be defined which may choose runs (thus observing internal actions and states), as in the Agedis version of TGV, or for ioSTS test purposes considered in section 4. These extensions are implemented in the product operation and do not modify the definition of $ATraces(S, TP)$. based on internal actions and states.

Property	Explanations
(1) TC is output-complete, verdict states are sink	always, def 3
(2) $Traces(TC) \subseteq Traces(S)\cdot(\{\epsilon\} \cup \Lambda_!)$	always, def 3
(3) $Traces_{\mathsf{Fail}}(TC) = Traces(TC) \cap ((Traces(S)\cdot\Lambda_!) \setminus Traces(S))$	soundness, def 5
(4) $Traces_{\mathsf{Pass}}(TC) = Traces(TC) \cap ATraces(S, TP)$	soundness, def 5
(5) $Traces_{\mathsf{Inconc}}(TC) \subseteq RTraces(S, TP)$	soundness, def 5
(6) $Traces_{\mathsf{Inconc}}(TC) = Traces(TC) \cap RTraces(S, TP) \cap pref_{<}(ATraces(S, TP))\cdot\Lambda_!$	optimality, def 10
(7) $Traces(TC) \cap \Lambda^*\cdot\Lambda_? \subseteq pref_{\leq}(ATraces(S, TP))$	optimality, def 10

Table 1: Properties of Test Cases

A test case TC should emit the appropriate verdicts in the appropriate situations. Fail verdicts should be emitted if and only if a non conformance is observed. This requirement depends only on S. Additionally, since a test purpose TP is used for selection, TC and IUT can be viewed as players in a game. In this context, Pass verdicts should reflect success of the game for the test case, while Inconc verdicts should reflect defeat for the test case. These requirements are made explicit in the following definition:

Definition 5 (Soundness of test case verdicts) *The verdicts of TC are* sound *w.r.t. S and TP whenever the following properties of Table 1 are satisfied:*

- *(3):* Fail *is emitted* iff *TC observes an unspecified output after a trace of S.*
- *(4):* Pass *is emitted* iff *TC observes a trace of S accepted by TP.*
- *(5):* Inconc *may be emitted* only if *the trace observed by TC belongs to S (thus, it is conformant) but is refused by TP. In this case, the test execution can be interrupted, as* Pass *cannot be emitted any more.*

Notice that for test cases satisfying Definition 5, Fail and Pass verdicts are uniquely defined, so that they are emitted appropriately and as soon as possible. In particular for Fail, the requirement is stronger than the usual notion of soundness [20] which says that only non conformant IUTs can be rejected. On the other hand, Definition 5 does not uniquely define the Inconc verdict. We have adopted this definition in anticipation of the general (symbolic) test selection algorithm (addressed in Section 4) where checking whether a trace is refused is undecidable. Finally, note that our requirements allows Inconc verdicts on system inputs, which is unusual but simplifies the discussion. Those inputs may be pruned during the off-line test selection phase.

2.4 Off-line test selection algorithm

For finite ioLTS, the principles of test generation using test purposes [14] are described by Table 2, assuming a specification S and a test purpose TP. Explanations and sketch of proof are given below.

1. After Step 1, by properties of $\times$, $Traces_{\mathsf{Pass}}(P) = ATraces(S, TP)$, implying Property (4) of Table 1, and $Traces(P) \subseteq Traces(S)$, implying Property (2). Intuitively, the product P combines information about conformance, coming from S, and information about the game with the IUT, coming from TP.

2. After Step 2, we have $Traces_{\mathsf{Inconc}}(P') \subseteq RTraces(S, TP)$, implying Property (5) of Table 1. Properties (2) and (4) from the previous step are preserved. This *selection* step is based on the definition of $RTraces(S, TP)$ and the following property: $pref_{\leq}(Traces_{\mathsf{Pass}}(P)) = Traces_{coreach(\mathsf{Pass})}(P)$.

```
{ 1.  Product and Pass verdict }
```
P := $S \times TP$; Pass := $Q^S \times \mathit{Accept}^{TP}$;
```
{ 2.  Selection and Inconc verdict }
```
$P' = (Q^P, Q_0^P, \Lambda, \rightarrow^{P'})$ is equipped with $\mathsf{Inconc} \subseteq Q^P$ and $\rightarrow^{P'}$ is defined by:

$$\frac{q, q' \in \mathit{coreach}(\mathsf{Pass}) \quad q \xrightarrow{\alpha}^{P} q' \quad \alpha \in \Lambda_? \cup \Lambda_!}{q \xrightarrow{\alpha}^{P'} q'} \text{(KeepI)} \qquad \frac{q \in \mathit{coreach}(\mathsf{Pass}), q' \notin \mathit{coreach}(\mathsf{Pass}) \quad q \xrightarrow{\alpha}^{P} q' \quad \alpha \in \Lambda_!}{q \xrightarrow{\alpha}^{P'} q' \quad q' \in \mathsf{Inconc}} \text{(Inconc)}$$

```
{ 3.  Input-completion and Fail verdict }
```
$TC = (Q^{P'} \cup \{\mathsf{Fail}\}, Q_0^P, \Lambda^{TC}, \rightarrow^{TC})$ with $\Lambda_?^{TC} = \Lambda_!$ and $\Lambda_!^{TC} = \Lambda_?$, and $\rightarrow^{TC}$ is defined by:

$$\frac{q \xrightarrow{\alpha}^{P'} q'}{q \xrightarrow{\alpha}^{TC} q'} \text{(KeepF)} \qquad \frac{\neg(q \xrightarrow{\alpha}^{P'}) \quad \alpha \in \Lambda_!}{q \xrightarrow{\alpha}^{TC} \mathsf{Fail} \qquad (\alpha \in \Lambda_?^{TC})} \text{(Fail)}$$

Table 2: Off-line test selection algorithm

The exact knowledge of $\mathit{coreach}(\mathsf{Pass})$ allows to detect when an action extends a trace and causes it to be refused (*i.e.*, not a prefix any more of accepted traces). In such a case, if the action is an output of the IUT, thus not controllable by the TC, we enter the Inconc state, rule (Inconc). Otherwise, if the action is an input of the IUT, thus controlled by the TC, we forbid its emission, rule (Inconc). In contrast, if a transition makes P' stay in $\mathit{coreach}(\mathsf{Pass})$, rule (KeepI), we keep it.

3. After Step 3, we have $\mathit{Traces}_{\mathsf{Fail}}(TC) = \mathit{Traces}(TC) \cap (\mathit{Traces}(S) \cdot \Lambda_! \setminus \mathit{Traces}(S))$, which is Property (3) of Table 1. Moreover, Property (1) becomes true (rule (Fail)). Properties (2), (4), (5) are preserved by the transformation.

We sketched the proof that generated test cases have sound verdicts, thus ensuring soundness (only non conformant implementations can be rejected). Conversely, we should prove exhaustiveness, *i.e.*, for a given specification S, any non-conformant IUT may be detected by a test case generated as above. This is true if we assume that any trace of S can be prolongated, which is the case if quiescence has been explicited by δ loops. By definition of *conf* , if $\neg(IUT \mathit{ conf } S)$, there exists a trace $\sigma.a \in (\mathit{Traces}(IUT) \cap \mathit{Traces}(S) \cdot \Lambda_!) \setminus \mathit{Traces}(S)$. It then suffices to choose TP such that $\sigma.b$ leads to Accept^{TP}, for an chosen b such that $\sigma.b \in \mathit{Traces}(S)$.

The TGV tool [11] is based on the above algorithm. The main optimization, consists in performing these operations on the fly. This means that S, P, P', and TC are built in a lazy way, from a high level specification, thus avoiding the state explosion problem. This involves both a reachability and coreachability analysis of P that do not modify the soudness of the test case. A test case generated as above is called a *complete test graph* as it contains all traces accepted by TP. This notion will be formalized in Section 6. TGV also allows to generate other test cases that are less complete, by pruning some outputs and the corresponding subgraphs.

We do not describe the *on-line* selection phase, that occurs during the parallel execution of the test case with the IUT. It is described later for ioSTS.

3 ioSTS: Input/Output Symbolic Transition Systems

In this section, we introduce a model of symbolic automata with a finite set of locations and typed variables, which communicates with its environment through actions carrying values. We call it ioSTS for Input/Output Symbolic Transition Systems. Figure 1 gives an example of such an ioSTS.

Variables, Predicates, Assignments.

In the sequel we shall assume a set of typed variables. We note $\mathcal{D}_v$ the domain in which a variable v takes its values. For a set of variables $V = \{v_1, \dots, v_n\}$, we note $\mathcal{D}_V$ the product domain $\mathcal{D}_{v_1} \times \ldots \times \mathcal{D}_{v_n}$. An element of $\mathcal{D}_V$ is thus a vector of values for the variables in V. We use also the notation $\mathcal{D}_{\vec{v}}$ for a vector $\vec{v}$ of variables. Depending on the context, a predicate $P(V)$ on a set of variables V may be considered either as a set $P \subseteq \mathcal{D}_V$, or as a logical formula, the semantics of which is a function $\mathcal{D}_V \to \{true, false\}$. An assignment for a variable v depending on the set of variables V is a function of type $\mathcal{D}_V \to \mathcal{D}_v$. An assignment for a set X of variables is then a function of type $\mathcal{D}_V \to \mathcal{D}_X$. We do not specify the syntactical constructions used for building predicates and assignments. They are discussed in the full paper.

3.1 Syntax of ioSTS

Definition 6 (IOSTS) *An Input/Output Symbolic Transition System $\mathcal{M}$ is defined by a tuple (V, Θ, Σ, T) where:*

- $V = V_p \cup V_o$ *is the set of variables, partitioned into a set V_p of* proper *variables and a set V_o of* observed *variables.*
- Θ *is the* initial condition*: a predicate $\Theta \subseteq \mathcal{D}_{V_p}$ defined on proper variables.*
- $\Sigma = \Sigma_? \cup \Sigma_!$ *is the finite alphabet of* actions*. Each action a has a* signature $\mathrm{sig}(a)$*, which is a tuple of types* $\mathrm{sig}(a) = \langle t_1, \dots, t_k \rangle$ *specifying the types of the* communication parameters *carried by the action.*
- T *is a finite set of* symbolic transitions*. A symbolic transition $t = (a, \vec{p}, G, A)$, also noted $\big[\, a(\vec{p}) \,:\, G(\vec{v}, \vec{p}) \,?\, \vec{v}_p' := A(\vec{v}, \vec{p}) \,\big]$, is defined by (i) an action $a \in \Sigma$ and a tuple of* (formal) communication parameters $\vec{p} = \langle p_1, \dots, p_k \rangle$*, which are local to a transition; without loss of generality, we assume that each action a always carries the same vector $\vec{p}$, which is supposed to be well-typed w.r.t. the signature* $\mathrm{sig}(a) = \langle t_1, \dots, t_k \rangle$*; $\mathcal{D}_{\vec{p}}$ is denoted by $\mathcal{D}_{\mathrm{sig}(a)}$; (ii) a* guard $G \subseteq \mathcal{D}_V \times \mathcal{D}_{\mathrm{sig}(a)}$*, which is a predicate on the variables and the communication parameters, and (iii) an assignment $A : \mathcal{D}_V \times \mathcal{D}_{\mathrm{sig}(a)} \to \mathcal{D}_{V_p}$, which defines the evolution of the proper variables.*

This model is rather standard, except for the distinction between proper and observed variables. The observed variables allow an observer ioSTS $\mathcal{M}_1$ to inspect the variables of another ioSTS $\mathcal{M}_2$ when composed together with it. Note also that there is no explicit notion of control location, since the control structure of an automaton can be encoded by a specific program counter variable.

3.2 Semantics of ioSTS

The s*emantics* of an ioSTS $\mathcal{M} = (V, \Theta, \Sigma, T)$ is an ioLTS $[\![\mathcal{M}]\!] = (Q, Q_0, \Lambda, \to)$:

- $Q = \mathcal{D}_V$, $Q_0 = \{\vec{\nu} = \langle \vec{\nu}_p, \vec{\nu}_o \rangle \mid \vec{\nu}_p \in \Theta \wedge \vec{\nu}_o \in \mathcal{D}_{V_o}\}$;
- $\Lambda = \{\langle a, \vec{\pi} \rangle \mid a \in \Sigma \wedge \vec{\pi} \in \mathcal{D}_{\mathrm{sig}(a)}\}$;

- $\rightarrow$ is defined by

$$\frac{(a,\vec{p},G,A) \in T \quad \vec{\nu} = \langle \vec{\nu}_p, \vec{\nu}_o \rangle \in \mathcal{D}_V \quad \vec{\pi} \in \mathcal{D}_{\mathrm{sig}(a)} \quad G(\vec{\nu},\vec{\pi}) \\ \vec{\nu}' = \langle \vec{\nu}'_p, \vec{\nu}'_o \rangle \in \mathcal{D}_V \quad \vec{\nu}'_p = A(\vec{\nu},\vec{\pi})}{\vec{\nu} \overset{\langle a,\vec{\pi}\rangle}{\rightarrow} \vec{\nu}'} \tag{Sem}$$

The rule says that a transition $(a,\vec{p},G,A)$ of an ioSTS is fireable in the current state $\vec{\nu} = \langle \vec{\nu}_p, \vec{\nu}_o \rangle$, if their exists a valuation $\vec{\pi}$ of the communication parameters $\vec{p}$ such that $\langle \vec{\nu}, \vec{\pi} \rangle$ satisfies the guard G; in such a case, the valued action $\langle a, \vec{\pi} \rangle$ is taken, the proper variables are assigned new values as specified by the assignment A, whereas observed variables take arbitrary values. Such a behaviour for observed variables reflects the fact that their value is defined by another ioSTS.

Given this semantics, most notions and properties of ioSTS are defined in terms of their underlying ioLTS semantics. For example, a run (resp. a trace) of an ioSTS $\mathcal{M}$ is a run (resp. a trace) of its ioLTS semantics $[\![\mathcal{M}]\!]$. An ioSTS $\mathcal{M}$ is *deterministic* if $[\![\mathcal{M}]\!]$ is deterministic. Whether an ioSTS is deterministic or not cannot be decided for ioSTS in the general case, as it implies the knowledge of reachable states. Sufficient conditions for an ioSTS to be deterministic exist (mutual exclusion of guards of all transtions labeled by the same action).

3.3 Synchronous product of ioSTS

The *product* of ioSTS is more complex than that of ioLTS: ioSTS synchronize on actions, but also via observed variables, which observe *runs* (not only traces). This allows to describe observers for *runs* (and not only for traces). This product is thus very similar to the one used in tools like ObjectGeode where a (non intrusive) observer is synchronized with an SDL specification for verification purpose.

Definition 7 (Product) *Two ioSTS $\mathcal{M}^i = (V^i, \Theta^i, \Sigma, T^i), i = 1,2$ with the same alphabet are* compatible *for product if $V_p^1 \cap V_p^2 = \emptyset$ (proper variables are disjoint). In this case, their product $\mathcal{M}^1 \times \mathcal{M}^2 = \mathcal{M} = (V, \Theta, \Sigma, T)$ is defined by*

- *$V = V_p \cup V_o$, with $V_p = V_p^1 \cup V_p^2$ and $V_o = (V_o^1 \cup V_o^2) \setminus V_p$;*
- *$\Theta(\langle \vec{v}^1, \vec{v}^2 \rangle) = \Theta^1(\vec{v}^1) \wedge \Theta^2(\vec{v}^2)$;*
- *T is defined by the following inference rule:*

$$\frac{\left[\, a(\vec{p}) \,:\, G^1(\vec{v}^1,\vec{p}) \,?\, (\vec{v}_p^1)' := A^1(\vec{v}^1,\vec{p}) \,\right] \in T^1 \\ \left[\, a(\vec{p}) \,:\, G^2(\vec{v}^2,\vec{p}) \,?\, (\vec{v}_p^2)' := A^2(\vec{v}^2,\vec{p}) \,\right] \in T^2}{\left[\, a(\vec{p}) \,:\, G^1(\vec{v}^1,\vec{p}) \wedge G^2(\vec{v}^2,\vec{p}) \,?\, (\vec{v}_p^1)' := A^1(\vec{v}^1,\vec{p}), (\vec{v}_p^2)' := A^2(\vec{v}^2,\vec{p}) \,\right]}$$

If $V_o^1 \cap V_p^2 \neq \emptyset$, G^1 and A^1 may depend on proper variables of $\mathcal{M}_2$ (cf. Fig. 3).

Let $\mathcal{M}^1$ and $\mathcal{M}^2$ be two ioSTS compatible for product, and $\mathcal{M} = \mathcal{M}^1 \times \mathcal{M}^2$. Then $\mathit{Traces}(\mathcal{M}) \subseteq \mathit{Traces}(\mathcal{M}^1) \cap \mathit{Traces}(\mathcal{M}^2)$. Let also $F^i = X^i \times \mathcal{D}_{V_o^i}$, where $i = 1,2$ and $X^i \subseteq \mathcal{D}_{V_p^i}$, be sets of accepting states of ioSTS $\mathcal{M}^i$. By taking as set of accepting states $F = X^1 \times X^2 \times \mathcal{D}_{V_o}$ for $\mathcal{M}$, we have $\mathit{Traces}_F(\mathcal{M}) \subseteq \mathit{Traces}_{F^1}(\mathcal{M}^1) \cap \mathit{Traces}_{F^2}(\mathcal{M}^2)$. It is not hard to see that the two trace inclusions are obtained from corresponding equalities for runs and accepting runs, which become inclusions by projection on observable actions.

3.4 Extending test theory to ioSTS

The testing theory for ioLTS developped in Section 2.2 also applies to ioSTS. Specifications, test purposes and test cases are assumed to be ioSTS; moreover:

- A specification is supposed to be an ioSTS $S = (V^S, \Theta^S, \Sigma, T^S)$ with only proper variables and no observed variable ($V^S = V_p^S$);
- A test purpose for S is an ioSTS $TP = (V^{TP}, \Theta^{TP}, \Sigma, T^{TP})$ such that $V_o^{TP} = V_p^S$ (symbolic test purposes are allowed to observe the internal state of S). The set of accepting states is defined by the truth value of a Boolean variable $\mathsf{Accept} \in V_p^{TP}$. TP should be *complete* except when $\mathsf{Accept} = \mathtt{true}$, which means that for any action a, $\bigcup_{(a,\vec{p},G,A)\in T^{TP}} G \Leftrightarrow \neg\mathsf{Accept}$. This condition can be enforced syntactically by completion of TP. It ensures that TP does not restrict the runs of S before they are accepted (if ever).
- A test case is an ioSTS $TC = (V^{TC}, \Theta^{TC}, \Sigma, T^{TC})$ with a variable $\mathsf{Verdict} \in V^{TC}$ of the enumerated type $\{\mathtt{none}, \mathtt{fail}, \mathtt{pass}, \mathtt{inconc}\}$.

The set of accepted traces is defined as $ATraces(S, TP) = Traces_{\mathsf{Accept}}(S \times TP)$ (as in Definition 4, except that the product is now the ioSTS product).

4 Off-line Test Selection for ioSTS

The aim of this section is to extend the test generation principles of ioLTS to *symbolic* test generation, taking into account the following difficulties:

1. Ensuring semantic transformations through syntactic operations of ioSTS;
2. Relying on approximate coreachability analysis instead of exact analysis, due to undecidability issues in the (infinite-state) symbolic case.

We consider again the simple case where the specification S and the test purpose TP do not contain internal actions or non-determinism. Our running example is depicted on Figures 1–5. The selection algorithm is given in Table 3.

1. Step 1 is the symbolic version of Step 1 for ioLTS (cf. Table 2). The same invariants hold. The transformation from P to P' consists in specifying the behavior of the $\mathsf{Verdict}$ variable and making states with $\mathsf{Verdict} \neq \mathtt{none}$ sink.
2. Step 2 is the main step of the selection. As the coreachability problem is now undecidable, coreachability analysis should be approximated. Fixpoint computations on ioSTS or similar models can indeed be overapproximated by classical Abstract Interpretation techniques [6, 9, 13]. We consider here an overapproximation $coreach^\alpha \supseteq coreach(\mathsf{Pass})$ of the exact set of coreachable states (see Figure 6(b)). It can be represented by a logical formula to be used in syntaxical operations on ioSTS. Moreover, $pre^\alpha(A)(X)$ denotes a formula representing an overapproximation of the precondition $pre(A)(X)$. In this context, $pre^\alpha(A)(coreach^\alpha)$ is an overapproximation of the set of values for variables and parameters which allow to stay in $coreach(\mathsf{Pass})$ when taking the transition, or in other words it is a *necessary condition.* Its negation is thus a *sufficient condition* to leave $coreach(\mathsf{Pass})$, and to lose the game for the test case. Hence, rule (KeepI) discards all (semantic) transitions labeled by a (controllable) input that *certainly* exit $coreach(\mathsf{Pass})$, and rule (KeepI) "redirects" to Inconc all transitions labelled by an (uncontrollable) output that *certainly* exit $coreach(\mathsf{Pass})$.
3. Finally, Step 3 is the symbolic version of the corresponding step in Table 2.

Intuitively, the main effect of approximations for the generated test case will be to miss the emission of the Inconc verdict, or to provide inadequate inputs to the IUT with the effect that the test case loses the game. In the worst case, when the approximation does not deliver any information, the Inconc verdict will never be emitted and the test case will not guide at all the parallel execution. This will be formalized in Section 6.

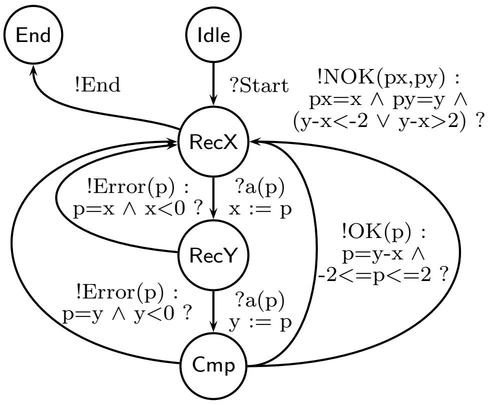


Figure 1: Specification

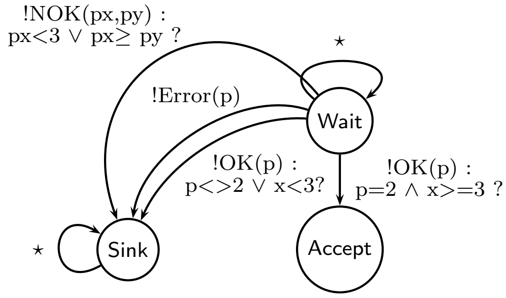


Figure 2: Test Purpose

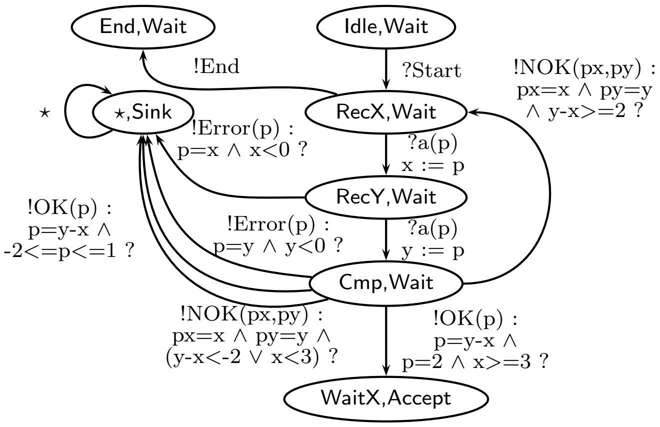


Figure 3: Product IOSTS (after Step 1)

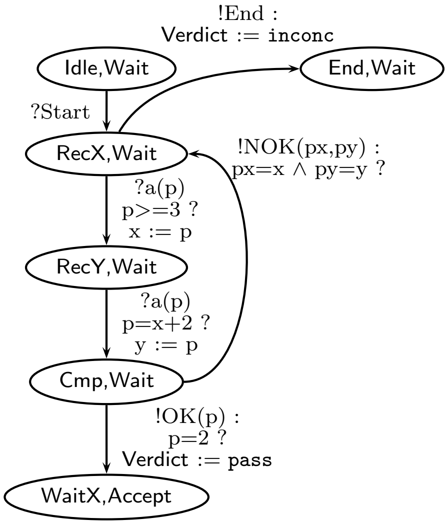


Figure 4: Product IOSTS modified using coreachability analysis and Inconc verdict (Step 2), and simplified with reachability analysis

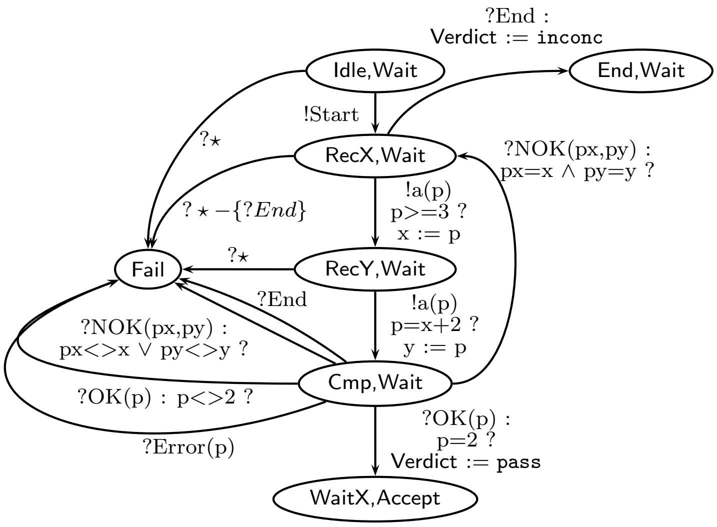


Figure 5: Resulting Test Case

The specification describes a program which is waiting for two successive inputs $?a(p_1)$ and $?a(p_2)$, and emits $!OK(p_2 - p_1)$ when their difference is less than 2 in absolute value, and $!NOK$ otherwise.

If the value held by the channel $?a$ is negative, the message $!Error$ is emitted. The program may also emits $!End$ and ends its execution.

The test purpose specifies that a test of interest is one where $!OK(2)$ is emitted, from a state of the specification where $x >= 3$. This implies that $p_1 \geq 3$, fact which is discovered by a coreachability analysis and taken into account in Figure 4.

`{ 1. Product and Pass verdict }`

P `:=` $S \times TP$

$P' = (V^P \cup \{\mathsf{Verdict}\}, \Theta^P \wedge \mathsf{Verdict} = \mathtt{none}, \Sigma, T^{P'})$ `is defined by`

$$\frac{[\, a(\vec{p}) \,:\, G(\vec{v},\vec{p}) \,?\, \vec{v}' = A(\vec{v},\vec{p}) \,] \in T^P}{\left[\begin{array}{c} a(\vec{p}) \,:\, G(\vec{v},\vec{p}) \wedge \mathsf{Verdict} = \mathtt{none} \,? \\ \vec{v}' := A(\vec{v},\vec{p}), \mathsf{Verdict}' := \mathtt{if}\ A_{\mathsf{Accept}}\ \mathtt{then\ pass\ else}\ \mathsf{Verdict} \end{array}\right] \in T^{P'}} \tag{3}$$

`{ 2. Selection and Inconc verdict }`

$P'' = (V^{P'}, \Theta^{P'}, \Sigma, \rightarrow^{P''})$ `is defined by`

$$\frac{[\, a(\vec{p}) \,:\, G(\vec{v},\vec{p}) \,?\, \vec{v}' = A(\vec{v},\vec{p}) \,] \in T^{P'}}{[\, a(\vec{p}) \,:\, G(\vec{v},\vec{p}) \wedge pre^{\alpha}(A)(coreach^{\alpha}) \,?\, \vec{v}' = A(\vec{v},\vec{p}) \,] \in T^{P''}} \quad \text{(KeepI)}$$

$$\frac{\big[\, a(\vec{p}) \,:\, G(\vec{v},\vec{p}) \,?\, \vec{v}' = A(\vec{v},\vec{p}); \mathsf{Verdict}' := A_{\mathsf{Verdict}} \,\big] \in T^{P'} \quad a \in \Sigma_!}{\big[\, a(\vec{p}) \,:\, G(\vec{v},\vec{p}) \wedge \neg pre^{\alpha}(A)(coreach^{\alpha}) \,?\, \vec{v}' = A(\vec{v},\vec{p}); \mathsf{Verdict}' := \mathtt{inconc} \,\big] \in T^{P''}} \quad \text{(Inconc)}$$

`{ 3. Input-completion and Fail verdict }`

$TC = (V^{P'}, \Theta^{P'}, \Sigma, T^{TC})$ `is defined by`

$$\frac{t \in T^{P''}}{t \in T^{TC}} \quad \text{(KeepF)} \qquad \frac{a \in \Sigma_! \quad G_a = \bigwedge\{\neg G(\vec{v},\vec{p}) \mid (a,\vec{p},G,A) \in T^{P''}\}}{\big[\, a(\vec{p}) \,:\, G_a(\vec{v},\vec{p}) \,?\, \mathsf{Verdict}' := \mathtt{fail} \,\big] \in T^{TC}} \quad \text{(Fail)}$$

Table 3: Off-line symbolic test selection algorithm

5 On-line Test Selection and Test Case Execution

The off-line test selection phase produces an ioSTS equipped with verdicts, but it does not implement a single strategy, in terms of its game against the IUT. Indeed, it may contain choices between several outputs that may be sent to the IUT. For instance, in Fig. 5, from the location RecX,Wait, the action $a(p)$ may be emitted for any $p \geq 3$. This illustrates the fact that the test case has to assign values to the formal communication parameters carried by the actions. As a consequence, test case execution implies *on-line constraint solving*.

In that respect, our test framework differs greatly from most test generation techniques based on off-line constraint solving [17, 16], where a test case is just a sequence and executing it implies only on-line constraint evaluation. The advantage of our framework is the ability to deal with non-deterministic systems, for which one cannot fully anticipate a (conformant) behavior.

A test case is thus executed as follows. At any point of the parallel execution of TC with IUT, TC is in a known state $\vec{\nu} \in \mathcal{D}_V$. It may then have the choice between observing an output of the IUT, or controling an input of the IUT:

- TC observes an output a of the IUT with actual values of parameters $\vec{\pi} \in \mathcal{D}_{sig(a)}$. As TC is input-complete and deterministic, exactly one transition $(a, \vec{p}, G, A) \in T^{TC}$ satisfies $\langle \vec{\nu}, \vec{\pi} \rangle \in G$. It then performs the assignments $\vec{\nu}' = A(\vec{\nu}, \vec{\pi})$, and checks the value of Verdict.

- TC controls an input a in a transition $(a, \vec{p}, G, A)$. By constraint solving, it chooses $\vec{\pi}$ such that $\langle \vec{\nu}, \vec{\pi} \rangle \in G$, sends $a(\vec{\pi})$ to IUT, performs $\vec{\nu}' = A(\vec{\nu}, \vec{\pi})$, and finally checks the verdict.

A test execution driver thus needs to implement the choice between observing or controlling, the evaluation of guards on outputs of the IUT, constraint solving for the choice of values of parameters on inputs of the IUT, and evaluation of assignments. Evaluation of a formula is never a problem, however the use of constraint solving techniques to instanciate input parameters imposes restrictions into a decidable theory, such as Presburger arithmetic.

6 Quality of test cases

As sketched during the off-line test selection algorithm of the previous section, test case verdicts are sound, which implies the usual soundness property of test cases — only non-conformant IUT can be rejected, like for ioLTS. Exhaustiveness — every non conformant IUT may be rejected, can also be proved, in a way similar to the proof for ioLTS.

These properties are related to conformance or to soundness of verdicts. They do not say whether test cases are good players in the game against the IUT. In this section we formalize qualitative properties of test cases relative to their ability to satisfy the test purpose during test execution, and show how the precision of the approximate analysis during the off-line selection algorithm influences them. We consider a fixed specification S and test purpose TP.

We can first compare two test cases in terms of their sets of traces leading to Pass. The requirement (4) of Table 1 only relates $Traces_{\mathsf{Pass}}(TC)$ and $Traces(TC)$. However, a test case can be pruned in any state where there exists a choice between several outputs (inputs of IUT). Such an operation may reduce the sets $Traces(TC)$ and $Traces_{\mathsf{Pass}}(TC)$.

Definition 8 (Completeness ordering; completeness of a test case) *Let TC and TC' be two test cases with sound verdicts (Definition 5), both generated from same S and TP. TC' is* less complete *than TC, denoted by $TC' \preceq^{\text{comp}} TC$, if $Traces_{\mathsf{Pass}}(TC') \subseteq Traces_{\mathsf{Pass}}(TC)$. TC is a* complete test case *if $Traces_{\mathsf{Pass}}(TC) = ATraces(S, TP)$.*

The test cases produced by the off-line test selection algorithms of Sections 2.4 and 4 are complete. In the TGV tool however [11], they are pruned to remove the choices between inputs to be sent to the IUT. Thus, in this case the on-line test selection (described in section 5 for ioSTS) is partly performed off-line.

The (partial) completeness of a test case is not directly related to its quality as a player in the game for satisfying the test purpose. However, we are only able to compare the quality of test cases when they are equivalent with respect to the completeness ordering. Otherwise, two test cases may have disjoint sets of traces, which makes their comparison as players difficult. We now define an accuracy ordering between test cases that are equivalent with respect to the completeness ordering. The definition seems simplistic, however it makes sense when examining its consequences, by taking the properties of Table 1 into account.

Definition 9 (Accuracy ordering) *Let TC and TC' be two test cases with sound verdicts that are equivalent for the completeness ordering.* TC is more accurate than TC', *denoted by $TC \preceq^{\text{acc}} TC'$, if $Traces(TC) \subseteq Traces(TC')$.*

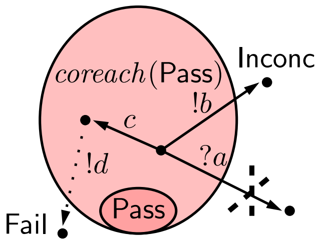
In particular, using properties of Table 1, $TC \preceq^{\text{acc}} TC'$ implies

1. $Traces_{\mathsf{Inconc}}(TC) \subseteq Traces(TC')$: TC' detects inconclusives later than TC, if ever;
2. $Traces_{\mathsf{Fail}}(TC) \subseteq Traces_{\mathsf{Fail}}(TC')$: TC' emits more Fail verdicts than TC,

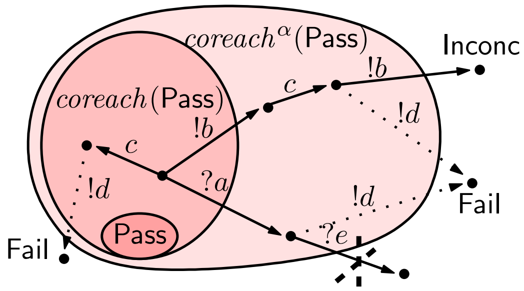
The second consequence may seem paradoxal: an accurate test case detects less conformance errors than a less accurate one! In fact, two test cases that are equivalent for the completeness ordering have same accepted traces and detect exactly the same errors along prefixes of those accepted traces. But a less accurate test case controls less the inputs of the IUT and emits later Inconc verdicts, which gives more opportunity to the IUT to exhibit non-conformance.

Now, among sound and completeness-equivalent test cases, the optimal test case can be defined as the test case where refused traces of S w.r.t. TP are never entered by a controllable input of the IUT and where Inconc verdict is emitted as soon as an output of IUT enters these refused traces.

Definition 10 (Optimal test case) *TC is* optimal *w.r.t. S and TP if:*



(a) Exact analysis



(b) Approximate analysis

Figure 6: Control and inconclusiveness: Step 2 of off-line selection algorithms

1. *$Traces(TC) \cap \Lambda^* \cdot \Lambda_? \subseteq pref_{\leq}(ATraces(S, TP))$: TC does not lose the game on a controllable action;*

2. *$Traces_{\mathsf{Inconc}}(TC) = Traces(TC) \cap RTraces(S, TP) \cap pref_{<}(ATraces(S, TP)) \cdot \Lambda_!$: TC immediately detects refused traces.*

These conditions correspond to properties (7) and (6) of Table 1, respectively.

In the case of finite ioLTS it is not hard to see that the algorithm in Section 2.4 builds sound, complete and optimal test cases. Completeness is obtained at Step 1 of the algorithm. Optimality is obtained at Step 2, as Inconc is reached exactly when leaving $coreach(\mathsf{Pass})$. Both properties are preserved by Step 3. In the case of ioSTS, we also build sound and complete test cases. However, the effect of the approximate coreachability analysis is to relax the optimality (see also Figure 6). The following theorem confirms the relevance of the accuracy ordering of test cases (Definition 9) and identifies the consequences of an approximate analysis in the off-line test selection algorithm.

Theorem 1 (Relating accuracy to precision) *Let TC and TC′ be two test cases generated by the algorithm described in Table 3, where TC was generated using a more precise approximation α than the approximation α' used for generating TC′, i.e., $pre^{\alpha}(A)(coreach^{\alpha}) \subseteq pre^{\alpha'}(A)(coreach^{\alpha'})$. Then $TC \preceq^{\text{acc}} TC'$.*

Proof. We need only to consider Step 2. For inputs, only the rule (KeepI) of Step 2 of the algorithm applies. In this case, a better precision for $pre^{\alpha}(A)(coreach^{\alpha})$ strengthens the guards of the symbolic transitions, implying that fewer semantic transitions will be inferred in the underlying ioLTS. For outputs, both inference rules apply. For any state q of the underlying ioLTS, a better precision for $pre^{\alpha}(A)(coreach^{\alpha})$ means that more semantic transitions from q leading to Inconc (which is sink) will be inferred by the rule (Inconc) while less transitions from q to its "normal" successors (which are generally not sink) will be inferred by the rule (KeepI). This implies that $Traces(TC^1) \subseteq Traces(TC^2)$. Moreover, $Traces_{\mathsf{Pass}}(TC^1) = Traces_{\mathsf{Pass}}(TC^2)$, hence the conclusion of the theorem. □

The two extreme cases are actually the following:

- The computation is exact; an *optimal* test case is obtained, as in Section 2.4;
- The approximation is maximal ($pre^{\alpha}(A)(coreach^{\alpha}) = true$), thus delivers no information: Step 2 of the algorithm has no effect and the test case is unable to control the implementation to satisfy the test purpose, nor to detect when Inconc should be emitted.

7 Implementation Issues

So far we have characterized the properties of test cases in term of different types of traces. However there are also other interesting practical issues, namely their size and optimality in term of execution speed.

Simplification of symbolic transitions. A reachability analysis of the generated test case allows to simplify it:

1. by removing dead branches (corresponding to dead code in the programming language community);
2. by simplifying expressions in symbolic transitions, using knowledge about reachable states. For instance, if a transition is guarded by the condition $G = (x \geq 0) \vee (0 \geq y \geq x)$ and if its origin location satisfies the invariant $(x \geq 1)$, we can replace G by `true` without modifying the semantics of the tets case in terms of runs.

The removal of dead branches reduces the size of the test case, whereas the simplification of expressions is likely to speed up the evaluation of expressions and simplifies the constraint solving problems occuring during the execution of the test case.

We describe now precisely the framework for which we have applied the principles of symbolic test generation and which have been implemented in the STG tool.

IOSTS. They are specified in a variant of the IF language (Intermediate Format) [], which is basically a textual language for symbolic automata. IF is connected to higher-level languages such as UML or SDL. The datatypes that are taken into account are the same as in [12]: Booleans, enumerated types and integers[1], as well as their combination in aggregate types and arrays. The logical atoms in formulas are either Boolean variables, equalities between enumerated variables/values, and linear constraints on integers. Expressions of non-Boolean types are enumerated variables/values or linear expressions, possibly guarded by formulas. Depending on the context, formulas and expressions are represented by Abstract Syntax Trees or by BDDs and MTBDDs, as described in [12].

Simplification of expressions. When an expressions e is represented with a BDD/MTBDD and the reachability invariant f with a BDD, a well-known and very efficient operator is the generalized cofactor defined in [5]. Its use in such a context of hybrid binary diagrams is also described in [12].

Satisfiability and constraints solving. The satisfiability test on formula is decidable with the above-mentioned assumption. Indeed, such formulas can be encoded with Presburger arithmetic formulas, which is a decidable theory. Actually, one one hand we use the Omega library to simplify the formulas and to solve constraints during the test case execution, on the other hand fixpoint analysis (reachability and coreachability) are performed by the NBac tool, based oon the use of BDDs and convex polyhedra [12, 13].

8 Conclusion

In this paper, we have presented a symbolic test generation algorithm for specifications and test purposes given as symbolic automata. Test generation has been decomposed into an off-line selection of test cases and an on-line execution on the *IUT*. The off-line selection is based on syntactical transformations and is parameterized by an approximate fixpoint analysis. We have showed how

[1]which could easily be generalized to rationals

the precision of the analysis influences the accuracy of selected test cases. The on-line execution is based on constraint solving. These algorithms have been implemented in our tool STG, STG relies on NBac (`http://www.irisa.fr/prive/bjeannet/nbac/nbac.html`) for the approximate fix-point analysis and Omega (`http://www.cs.umd.edu/projects/omega/`) for constraint resolution in Presburger arithmetic. STG uses a more general model than the one presented here, in particular admitting non-deterministic specifications under some conditions. While most other works are limited to controlable (and deterministic) systems, we are able to generate test cases for non-controllable (and non-deterministic) specifications. Moreover, we believe that our framework can be generalized to other models that cannot be analyzed in an exact way.

In a sense, our approach is an improvement of strictly on-line approaches (as e.g. TorX [2, 7]), which lack control of the test cases on the *IUT*. Off-line symbolic selection seriously improves this feature. In fact, off-line selection based on test purposes can be seen as a syntactic slicing of the specification w.r.t. particular scenarios, preserving the capability to generate sound test cases on line.

Several extensions of our work can be investigated. We have presented this work in the context of conformance testing, but similar techniques could be used in structural testing for the selection of test cases based on the source code. Also, other models can be considered, such as programs with recursive calls modelled as pushdown automata. One problem is then to decide what is observable and controlable by a tester. Finally, other selection criteria can benefit from our techniques, like safety properties [19] or standard structural coverage criteria.

References

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

[2] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In 12^{th} *Int. Workshop on Testing of Communicating Systems*. Kluwer Academic Publishers, 1999.

[3] M. Benjamin, D. Geist, A. Hartman, G. Mas, R. Smeets, and Y. Wolfsthal. A feasibility study in formal coverage driven test generation. In *DAC99*, 1999.

[4] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: a symbolic test generation tool. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS2002, Grenoble, France, LNCS 2280*. Springer-Verlag, April 2002.

[5] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*, volume 407 of *LNCS*, 1989.

[6] 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, Los Angeles, CA*, 1977.

[7] L. Frantzen, J. Tretmans, and T. Willemse. Test generation based on symbolic specifications. In *FATES 2004, Linz, Austria*, 2004.

[8] J. Grabowski, D. Hogrefe, and R. Nahm. Test Case Generation with Test Purpose Specification by MSCs. In *6^{th} SDL Forum, Darmstadt (Germany)*. North-Holland, 1993.

[9] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2), 1997.

[10] H.S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *TACAS '02, Grenoble, France, LNCS 2280*. Springer-Verlag, 2002.

[11] C. Jard and T. Jéron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 6, 2004.

[12] B. Jeannet. Representing and approximating transfer functions in abstract interpretation of hetereogeneous datatypes. In *Static Analysis Symposium, SAS'02, Madrid (Spain), LNCS 2477*, 2002.

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

[14] T. Jéron and P. Morel. Test generation derived from model-checking. In *CAV'99, Trento, Italy, LNCS 1633*. Springer-Verlag, 1999.

[15] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - A Survey. *Proceedings of the IEEE*, 84(8), 1996.

[16] B. legeard, F. Peureux, and M. Utting. Automated boundary testing from z and b. In *Proc. of Formal Methods Europe (FME'02)*, volume 2391 of *LNCS*, July 2002.

[17] B. Marre and A. Arnould. Test sequences generation from lustre descriptions: Gatel. In *IEEE Int. Conf. on Automated Software Engineering (ASE'0)*. IEEE Computer Society Press, September 2000.

[18] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *Integrated Formal Methods (IFM'00), Dagstuhl, Allemagne, LNCS 1945*. Springer Verlag, 2000.

[19] V. Rusu, H. Marchand, and T. Jéron. Verification and symbolic test generation for safety properties. Technical Report 5285, INRIA, 2004. Available at `http://www.inria.fr/rrrt/rr-5285.html`.

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