Symbolic model-based test selection

Thierry Jéron[1,2]

IRISA / INRIA Rennes - Bretagne Atlantique
Campus de Beaulieu, 35042 Rennes – France

Abstract

This paper addresses the problem of model-based off-line selection of test cases for testing the conformance of a black-box implementation with respect to a specification, in the context of reactive systems. Efficient solutions to this problem have been proposed for LTS finite-state models, based on the ioco conformance testing theory. In this paper, the approach is extended for infinite-state specifications, modelled as automata extended with variables. When considering the selection of test cases according to test purposes (abstract scenarii focused by test cases), the selection of test cases relies on approximate co-reachability analyses using abstract interpretation and syntactical transformations guided by this analysis, while test execution uses constraint solving.

Keywords: Model-based testing, conformance, test selection, model, reactive systems

1 Introduction

Many aspects of software can be tested to assess its correctness e.g., functionality, performance, timing, robustness, etc. Among these, the focus of this paper is on conformance testing of reactive systems. This consists in checking that a black-box implementation of a system, only known by its interactions with the environment through an interface, behaves correctly with respect to its specification. This relies on experimenting the system with test cases, with the objective of detecting some faults, or to improve the confidence one may have in the implementation.

Automatization of some parts of the testing activity, using models of software and formal methods is a way to improve the quality and cost of testing. For more than a decade, *model-based testing* (see e.g., [5]) advocates the use of formal models and methods to formalize this activity. The formalization relies on models of testing artifacts, relations between them including conformance, and properties they satisfy. For reactive systems, behavioral models are used for specifications, and serve both as a basis for test generation, and as an oracle for the assignment of verdicts in test cases. Testing theories based on finite-state models such as automata (see

[1] Email: `Thierry.Jeron@irisa.fr`, Web: `www.irisa.fr/prive/jeron`

[2] We would like to thank colleagues who participated in this research work, and the SBMF '08 committee for their invitation to present this work.

e.g., the survey [19]), or labelled transition systems (see e.g., [25]) are now well understood, and gave rise to automatic test generation algorithms and tools like TorX [2], TGV [14], Gotcha [3] among others, have been developed and successfully used on industrial-size systems.

Some developments are still necessary to improve the automation of test generation. In particular more powerful models taking into account some aspects of complex software must be considered such as data, time, etc. In this paper we focus on data, and models called Input/Output Symbolic Transition Systems (ioSTS) are considered. These are automata extended with variables, with distinguished input and output actions, and corresponding to reactive programs without recursion. As their semantics can be defined in terms of infinite-state Input/Output Labelled Transition Systems (ioLTS), the ioco testing theory [25], which defines conformance as a partial inclusion of external behaviours (suspension traces) of the implementation in those of the specification, can be reused. The difficulty is in the selection of test cases. Test cases are programs with variables, directly built from the ioSTS model rather than from the enumerated ioLTS semantic model. This construction relies on syntactical transformations of the specification model, guided by an approximate analysis. The models and principles of the test selection algorithms are illustrated by a simple example of a lift controller. The test selection algorithms described in this paper are implemented in the STG tool [16] (see `http://www.irisa.fr/vertecs/software.html#STG`). The approximate co-reachability and reachability analyses are provided by an interface with the NBac tool [15] (see `http://pop-art.inrialpes.fr/people/bjeannet/nbac/index.html`) using abstract interpretation [8]. This work builds on previous work by our team such as [24,17].

Related work

Several other works, done in the context of conformance testing or white bow testing can be related to our work.

Some are based on symbolic execution [4,13] and/or constraint resolution: the DART tool [11] combines symbolic execution with random testing and constraint solving, the PET tool [12] uses constraint solving to produce test cases as solutions to path conditions produced from a specification and a property, the Agatha tool [10] uses symbolic execution on a variation of the ioSTS model, the Gatel [23] and BZ-TT [20] tools rely on constraint solving. In [21] the authors use selection hypotheses combined with unfolding for algebraic data types and predicate resolution to produce test cases. Compared to our approach, these approaches are limited to deterministic systems, and consider finite unfoldings of systems by limiting the search depth in order to cope with loops, but they may produce precise test cases. Nevertheless, these are complementary methods: test selection with test purposes using approximate analyses can be seen as a front-end used to select an abstract test case, where information on non-conformance is preserved. Then constraint solving techniques can be used to search for instantiated test cases, by limiting the unfolding of remaining loops.

Other approaches are funded on abstractions: [6] uses TGV on an abstraction before concretization of test cases using constraint solving, and Ball [1] uses a combination of predicate abstraction, reachability analysis and symbolic execution. Very recently, abstract interpretation has also been proposed in the context of white box structural testing in combination with constraint solving [9].

2 Modeling reactive systems with ioSTS

In this section, a model of reactive systems, called ioSTS for *Input/Output Symbolic Transition Systems*, is proposed which will serve for specifications, test cases and test purposes. This model is a kind of extended automata model, inspired by I/O automata [22]. It is made of a set of variables (including one encoding control locations) which encode the state of the system, and transitions with guards, input and output actions with communication parameters and assignments of variables to expressions.

Definition 2.1 [ioSTS] An Input/Output Symbolic Transition System $\mathcal{M}$ is defined by a tuple (V, Θ, Σ, T) where:

- $V = V_i \cup V_x$ is the set of variables, partitioned into a set V_i of *internal* variables and a set V_x of *external* variables. $\mathcal{D}_v$ denotes the domain in which v takes its values,
- Θ is the *initial condition*. It is a predicate $\Theta \subseteq \mathcal{D}_{V_i}$ defined on internal variables. It is assumed that Θ has a unique solution in $\mathcal{D}_{V_i}$.
- $\Sigma = \Sigma_? \cup \Sigma_!$ is the finite alphabet of *actions*. Each action a has a *signature* $\mathrm{sig}(a)$, which is a tuple of types $\mathrm{sig}(a) = \langle t_1, \ldots, t_k \rangle$ specifying the types of the *communication parameters* carried by the action.
- T is a finite set of *symbolic transitions*. A symbolic transition $t = (a, \boldsymbol{p}, G, A)$, also written $[\, a(\boldsymbol{p}) \,:\, G(\boldsymbol{v}, \boldsymbol{p}) \,?\, \boldsymbol{v}_i' := A(\boldsymbol{v}, \boldsymbol{p}) \,]$, is defined by
  - · an action $a \in \Sigma$ and a tuple of *(formal) communication parameters* $\boldsymbol{p} = \langle p_1, \ldots, p_k \rangle$, which are local to a transition; without loss of generality, it is assumed that each action a always carries the same vector $\boldsymbol{p}$, which is supposed to be well-typed w.r.t. the signature $\mathrm{sig}(a) = \langle t_1, \ldots, t_k \rangle$; $\mathcal{D}_{\boldsymbol{p}}$ is denoted by $\mathcal{D}_{\mathrm{sig}(a)}$;
  - · a *guard* $G \subseteq \mathcal{D}_V \times \mathcal{D}_{\mathrm{sig}(a)}$, is a predicate on variables (internal and external) and communication parameters.
  - · an assignment $A : \mathcal{D}_V \times \mathcal{D}_{\mathrm{sig}(a)} \to \mathcal{D}_{V_i}$, which defines the evolution of the internal variables. $A_v : \mathcal{D}_V \times \mathcal{D}_{\mathrm{sig}(a)} \to \mathcal{D}_v$ denotes the function in A defining the evolution of the variable $v \in V_i$.

This model does not explicitly define control locations, but these can be encoded (as in the examples) by a variable *pc* with finite domain. One particularity of the model is the distinction made between internal and external variables. Internal variables are standard variables defining the state of the system itself, while external variables are used to observe the state of another system (thus modified by this other system). As will be seen, this allows to use the same model for specifications and test purposes. It is assumed that guards are expressed in a theory in which satisfiability is decidable. We will come back later to the hypothesis of satisfiability of guards, which will be important during test execution.

Example.

Fig. 1 describes an example of ioSTS specifying a simple lift controller (the dashed edges labelled with δ are not part of the specification and will be explained later). The integer constant h specifies the height of the building, the integer variables c and g are initialized to 0 and respectively specify the current level, and the target level. $Wait$, $Move$ and End are control locations (values of the pc variable). In location $Wait$, an input $Target?$ is read and a level is chosen by providing a value between 0 and h to the parameter p, then stored in g, and the lift controller reaches $Move$. In location $Move$, if the current level c equals the target level g, a $Stop!(p)$ is sent with $p = c$. If $c < g$, an output $Up!(p)$ is sent with $p = c$, and c is increased. If $c > g$, the output $Down!(p)$ is sent with parameter $p = c$, and c is decreased. In case of breakdown, the controller may non-dterministically send an output $Break!$ and go to state End.

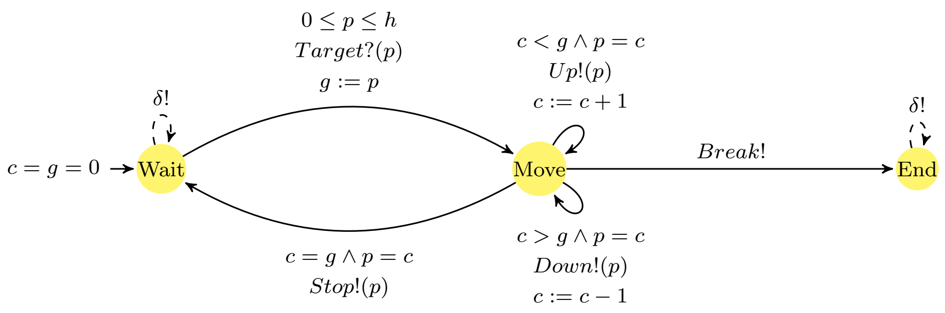

Fig. 1. ioSTS specification $\mathcal{S}$ of a simple lift controller example

The *semantics* of an ioSTS $\mathcal{M} = (V, \Theta, \Sigma, T)$ is an input/output labelled transition system (ioLTS) $[\![\mathcal{M}]\!] = (Q, Q_0, \Lambda, \rightarrow)$ where:

- $Q = \mathcal{D}_V$ is the set of states;
- $Q^0 = \{\boldsymbol{\nu} = \langle \boldsymbol{\nu}_i, \boldsymbol{\nu}_x \rangle \mid \boldsymbol{\nu}_i \in \Theta \wedge \boldsymbol{\nu}_x \in \mathcal{D}_{V_x}\}$ is the subset of initial states;
- $\Lambda = \{\langle a, \boldsymbol{\pi} \rangle \mid a \in \Sigma \wedge \boldsymbol{\pi} \in \mathcal{D}_{\mathrm{sig}(a)}\}$ is the set of *valued actions* partitioned into *valued inputs* $\Lambda_?$, and *valued outputs* $\Lambda_!$;
- the transition relation $\rightarrow$ is defined by

$$\frac{(a, \boldsymbol{p}, G, A) \in T \quad \boldsymbol{\nu} = \langle \boldsymbol{\nu}_i, \boldsymbol{\nu}_x \rangle \in \mathcal{D}_V \quad \boldsymbol{\pi} \in \mathcal{D}_{\mathrm{sig}(a)} \quad \boldsymbol{\nu}' = \langle \boldsymbol{\nu}'_i, \boldsymbol{\nu}'_x \rangle \in \mathcal{D}_V \qquad G(\boldsymbol{\nu}, \boldsymbol{\pi}) \quad \boldsymbol{\nu}'_i = A(\boldsymbol{\nu}, \boldsymbol{\pi})}{\boldsymbol{\nu} \overset{\langle a, \boldsymbol{\pi} \rangle}{\longrightarrow} \boldsymbol{\nu}'} \quad \text{(Sem)}$$

Intuitively a state is composed of the values of internal and external variables. The set of initial states is composed of states where the internal part is uniquely determined by Θ, while the external part is arbitrary. Transitions are labelled by valued actions and the set of transitions is determined by the rule (Sem): in a state $\boldsymbol{\nu} = \langle \boldsymbol{\nu}_i, \boldsymbol{\nu}_x \rangle$, a transition $(a, \boldsymbol{p}, G, A)$ can be fired if there exists a valuation $\boldsymbol{\pi}$ of the communication parameters $\boldsymbol{p}$ such that $\langle \boldsymbol{\nu}, \boldsymbol{\pi} \rangle$ satisfies the guard G; the valued action $\langle a, \boldsymbol{\pi} \rangle$ is then taken, resulting in a state $\boldsymbol{\nu}'$ where the new values of the internal variables are defined by the assignment A, while external variables take arbitrary values, which reflects the fact that their value is defined by another ioSTS. As variables may have infinite domains, the semantics of an ioSTS may be an infinite-state ioLTS.

Example.

The lift-controller example having no external variable, states are composed of values of the variables pc (locations), c (current level) and g (target level).

Notations on ioLTS

In the sequel we will use standard notations of LTS. As usual $q \overset{\alpha}{\to} q'$ is used for $(q, a, q') \in \to$ and $q \overset{\alpha}{\to}$ for $\exists q', q \overset{\alpha}{\to} q'$. For a sub-alphabet $\Lambda' \subseteq \Lambda$, a state q of M is said Λ'-*complete* if $\forall \alpha \in \Lambda' : q \overset{\alpha}{\to}$. It is *complete* if it is Λ-*complete*. The ioLTS $[\![\mathcal{M}]\!]$ is Λ'-*complete* (resp. complete) if all its states are Λ'-*complete* (resp. complete). Note that these completeness conditions can be defined on ioSTS: an ioSTS $\mathcal{M}$ is Σ'-complete for $\Sigma' \subseteq \Sigma$, if for any $a \in \Sigma'$, the predicate $\bigwedge_{(a,\boldsymbol{p},G,A)\in T} \neg G$ is unsatisfiable (otherwise said $\forall a \in \Sigma', \bigvee_{(a,\boldsymbol{p},G,A)\in T} G = \mathit{true}$).

Runs and traces.

The behavior of an ioSTS is defined by its ioLTS semantics. 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. $\forall i, q_i \overset{\alpha_i}{\to} q_{i+1}$. The run ρ is *accepted in* $F \subseteq Q$ if $q_n \in F$. $\mathit{Runs}(\mathcal{M})$ denotes the set of runs of $\mathcal{M}$ and $\mathit{Runs}_F(\mathcal{M})$ is the set of accepted runs in F. When modelling the testing activity, we consider abstractions of runs where states are abstracted away, as variables and locations cannot be observed by the environment. A *trace* of a run $\rho \in \mathit{Runs}(\mathcal{M})$ is the projection $\mathit{proj}_\Lambda(\rho)$ of ρ on actions. $\mathit{Traces}(\mathcal{M}) \triangleq \mathit{proj}_\Lambda(\mathit{Runs}(\mathcal{M}))$ denotes the set of traces of $\mathcal{M}$ and $\mathit{Traces}_F(\mathcal{M}) \triangleq \mathit{proj}_\Lambda(\mathit{Runs}_F(\mathcal{M}))$ is the set of traces of runs accepted in F.

For the sake of simplicity, we restrict to deterministic ioSTS, where an ioSTS $\mathcal{M} = (V, \Theta, \Sigma, T)$ is *deterministic* if for any action $a \in \Sigma$, and any pair of transitions $t_1 = (a, \boldsymbol{p}, G_1, A_1)$ and $t_2 = (a, \boldsymbol{p}, G_2, A_2)$ carrying the same action, the conjunction of the guards $G_1 \wedge G_2$ is unsatisfiable. In fact, it is not always possible to transform an ioSTS into a deterministic one having the same set of traces. However, a class of ioSTS as been identified which can be determinized effectively [18].

When testing for conformance, one is interested in comparing the observable behavior, thus traces of the implementation with the ones of the specification. Additionally, one can also observe quiescences of the implementation, and check that they are allowed by the specification. A quiescence occurs when no output is fireable: the system is blocked unless the environment provides an input. A particular case is a deadlock, i.e., when additionally no input is fireable. As quiescence is not preserved by determinization (the same trace can lead to a quiescent and a non-quiescent state), this information has to be computed before determinization. The corresponding transformation called *suspension* was originally defined for ioLTS [25], and here denoted Δ. It consists in adding a self loop labelled with a new output

δ in every quiescent state. Suspension is here defined directly for ioSTS with the expected effect on the ioLTS semantics (i.e. $[\![\Delta(\mathcal{M})]\!] = \Delta([\![\mathcal{M}]\!])$):

Definition 2.2 [ioSTS suspension] For an ioSTS $\mathcal{M} = (V, \Theta, \Sigma, T)$ with alphabet $\Sigma = \Sigma_! \cup \Sigma_?$, the *suspension* of $\mathcal{M}$ is the ioSTS $\Delta(\mathcal{M}) = (V, \Theta, \Sigma^\delta, T_\delta)$ where:

- the alphabet is increased by a new output: $\Sigma^\delta = \Sigma_!^\delta \cup \Sigma_?$ with $\Sigma_!^\delta = \Sigma_! \cup \{\delta\}$,
- new loop transitions labelled by δ are added: $T_\delta = T \cup \{\langle \delta, G_\delta, Id_V \rangle\}$ with

$$G_\delta = \neg \left(\bigvee_{(a,\boldsymbol{p},G,A) \in T,\ a \in \Sigma_!} \exists \boldsymbol{\pi} \in \mathcal{D}_{\mathrm{sig}(a)} : G(\boldsymbol{\nu}, \boldsymbol{\pi}) \right)$$

The guard G_δ associated to δ evaluates to `true` exactly when there is no valuation of variables and communication parameters such that an output can be fired.

The definition of suspension leads to the the observable behavior considered for testing of an ioSTS $\mathcal{M}$ can be defined as the traces of its suspension $\Delta(\mathcal{M})$. The set *Traces*$(\Delta(\mathcal{M}))$ will be denoted by *STraces*$(\mathcal{M})$. These *suspension traces* are the reference for conformance testing and will thus be central in the definition of the conformance relation.

Example

For the lift-controller example, Fig. 1 describes the suspension automaton, with δ actions with guards `true` in states $Wait$ (no output is firebale) and End (deadlock).

3 Conformance testing theory

Conformance testing consists in checking that an implementation exhibits an observable behavior consistent with its specification. In order to formalize this, one has to fix models for specifications, implementations and test cases:

- The specification is a deterministic ioSTS $\mathcal{S} = (V^{\mathcal{S}}, \Theta^{\mathcal{S}}, \Sigma, T^{\mathcal{S}})$, with $\Sigma = \Sigma_! \cup \Sigma_?$ and $V_x^{\mathcal{S}} = \emptyset$ ($\mathcal{S}$ has only internal variables). Its ioLTS semantics is $[\![\mathcal{S}]\!] = S = (Q, Q^0, \Lambda, \rightarrow)$ with $\Lambda = \Lambda_! \cup \Lambda_?$.
- Implementations are unknown, except for their interfaces and are not models but real systems. However, in order to reason about conformance, implementations are assumed to behave as models. The implementation is modelled by a (possibly non-deterministic) ioLTS $I = (Q_I, Q_I^0, \Lambda_! \cup \Lambda_?, \rightarrow_I)$ having the same interface as $\mathcal{S}$. I is assumed to be $\Lambda_?$-complete [3]. $\Delta(I)$ denotes its suspension ioLTS.
- A test case for the specification ioSTS $\mathcal{S}$ is a deterministic ioSTS $\mathcal{TC} = (V^{TC}, \Theta^{TC}, \Sigma^{TC}, T^{TC})$, where $\Sigma_?^{TC} = \Sigma_!$ and $\Sigma_!^{TC} = \Sigma_?$ (actions are mirrored w.r.t. $\mathcal{S}$), equipped with a variable $\mathsf{Verdict} \in V^{TC}$ of the enumerated type $\{$`none`, `fail`, `pass`, `inconc`$\}$. Intuitively, `fail` means rejection, `pass` means that some targeted behaviour has been reached (this will be clarified later) and `inconc` means that targeted behaviours cannot been reached anymore. $\mathcal{TC}$ is

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

assumed to be $\Sigma_?^{TC}$-complete in all states where $\mathsf{Verdict} = \mathtt{none}$. This means that $\mathcal{TC}$ is ready to react to any output of the implementation, except when a verdict is reached and the execution stops. $TC = [\![\mathcal{TC}]\!] = (Q^{TC}, q_0^{TC}, \Lambda^{TC}, \rightarrow_{TC})$ denotes the ioLTS semantics of $\mathcal{TC}$. The predicates $\mathsf{Fail} = (\mathsf{Verdict} = \mathtt{fail})$, $\mathsf{Pass} = (\mathsf{Verdict} = \mathtt{pass})$, and $\mathsf{Inconc} = (\mathsf{Verdict} = \mathtt{inconc})$ denote the subsets of Q^{TC} where verdicts are emitted.

We are now ready to formalize conformance. We consider the ioco conformance relation [25] which says that after any suspension trace of the specification, outputs (or quiescences) of the implementation must be specified. We here adopt a definition where non-conformant suspension traces are explicit:

Definition 3.1 [Conformance] Let $\mathcal{S}$ be a specification, and I be an implementation of S. The ioco conformance relation is defined as:

$$I \ \mathsf{ioco}\ \mathcal{S} \triangleq STraces(I) \cap NC\_STraces(\mathcal{S}) = \emptyset$$

where $NC\_STraces(\mathcal{S}) = STraces(\mathcal{S}) \cdot (\Lambda_! \cup \{\delta\}) \setminus STraces(\mathcal{S})$ is the set of minimal (with respect to the prefix order) non-conformant suspension traces.

Example.

For the lift-controller with constant $h = 10$, the suspension trace $\delta!.Target?(5).Up!(0).Up!(1).Up!(2).Up!(3).Up!(4).Stop!(5)$ is conformant, while $Target?(5).Up!(0).Up(1).Up(2).Down!(3).Up!(2).Up(3).Up(4).Stop!(5)$ is not as the lift is not supposed to go down if the lift is lower than the target level.

The definition of conformance exhibits the fact that conformance is a safety property: it can be violated by a finite trace. As usual, the negation of this safety property can be defined by an observer which recognizes the non-confomant suspension traces $NC\_STraces(\mathcal{S})$, giving rise to the definition of the *canonical tester*:

Definition 3.2 [Canonical Tester] Let $\mathcal{S} = (V^{\mathcal{S}}, \Theta^{\mathcal{S}}, \Sigma, T^{\mathcal{S}})$ be a deterministic ioSTS and $\Delta(S) = (V^{\mathcal{S}}, \Theta^{\mathcal{S}}, \Sigma^{\delta}, T^{\mathcal{S}}_{\delta})$ its suspension. The *canonical tester* for $\mathcal{S}$ is the (deterministic) ioSTS $Can(\mathcal{S}) = (V^{Can}, \Theta^{Can}, \Sigma^{Can}, T^{Can})$ such that:

- $V^{Can} = V^{\mathcal{S}} \cup \{\mathsf{Verdict}\}$ where $\mathsf{Verdict}$ is of the enumerated type $\{\mathtt{none}, \mathtt{fail}\}$
- $\Theta^{Can} = \Theta^{\mathcal{S}} \wedge \mathsf{Verdict} = \mathtt{none}$;
- $\Sigma_?^{Can} = \Sigma_!^{\delta}$ and $\Sigma_!^{Can} = \Sigma_?$ (the alphabet is mirrored w.r.t. $\Delta(\mathcal{S})$)
- T^{Can} is defined by the rules:

$$\frac{t \in T^{\Delta(\mathcal{S})}}{t \in T^{Can}} \qquad \text{(Keep } T^{\mathcal{S}}\text{)}$$

$$\frac{a \in \Sigma_!^{\delta} = \Sigma_?^{Can} \quad G_a = \bigwedge_{(a,\boldsymbol{p},G,A) \in T^{\Delta(\mathcal{S})}} \neg G}{\left[\, a(\boldsymbol{p}) \ : \ G_a(\boldsymbol{v}, \boldsymbol{p}) \, ? \, \mathsf{Verdict}' := \mathtt{fail} \,\right] \in T^{Can}} \qquad \text{(Input-completion)}$$

Example.

The canonical tester of the lift-controller is described in Fig. 2. First the inputs and outputs are reversed compared to $\mathcal{S}$ and new transitions to a *Fail* location have been added, labelled with actions with negations of the guards of transitions in $\mathcal{S}$, e.g., from the *Move* location, the transition carrying $Up!(p)$ is guarded by $c < g \wedge p = c$ in $\mathcal{S}$, and a new transition to Fail carrying $Up!(p)$ and guarded by $c \geq g \vee p \neq c$ is added in $Can(\mathcal{S})$.

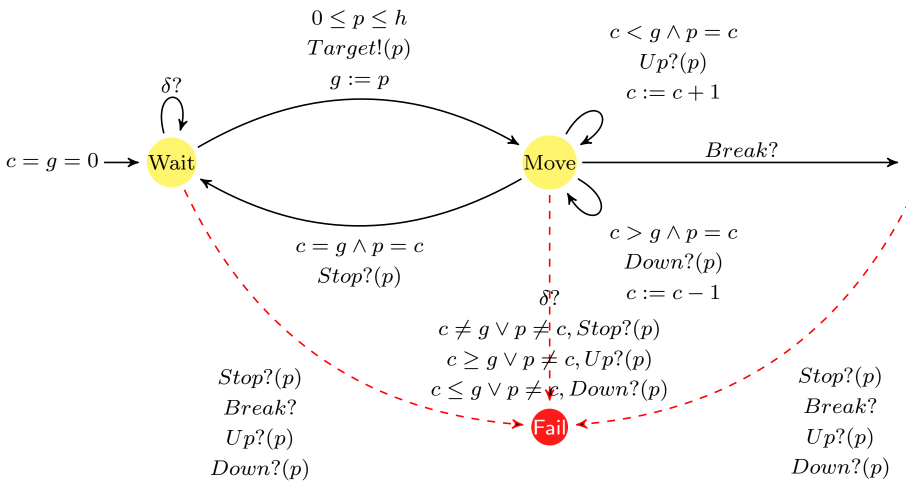

Fig. 2. Canonical tester

$Can(\mathcal{S})$ is a test case by itself: it is deterministic and $\Sigma_?^{Can}$-complete in all states where $\mathsf{Verdict} = \mathtt{none}$. Moreover $Can(\mathcal{S})$ exactly recognizes non-conformant behaviours in its Fail states, as $Traces_{\mathsf{Fail}}(Can(\mathcal{S})) = NC\_STraces(\mathcal{S})$. Conformance can then be expressed using the canonical tester:

$$I \ \mathsf{ioco}\ \mathcal{S} \iff STraces(I) \cap Traces_{\mathsf{Fail}}(Can(\mathcal{S})) = \emptyset$$

If I was known, checking non-conformance would be simple. However, as I is unknown, complete verification is impossible and one has to experiment the implementation with selected test cases. The canonical tester will play a central rôle in this selection. Test cases will have to satisfy some properties. In order to define these, the execution of a test case on an (model of) implementation must be formalized first.

Let I be an implementation with $\Delta(I) = (Q^I, Q_0^I, \Lambda_! \cup \{\delta\} \cup \Lambda_?, \rightarrow_{\Delta(I)})$ its suspension, and $TC = (Q^{TC}, q_0^{TC}, \Lambda_? \cup \Lambda_! \cup \{\delta\}, \rightarrow_{TC})$ the ioLTS semantics of an ioSTS test case $\mathcal{TC}$. The test execution of TC on I is modelled by the *parallel composition* of $\Delta(I)$ and TC, an ioLTS $\Delta(I)\|TC = (Q^I \times Q^{TC}, Q_0^I \times \{q_0^{TC}\}, \Lambda_! \cup \{\delta\} \cup \Lambda_?, \rightarrow_{\Delta(I)\|TC})$ where $\rightarrow_{\Delta(I)\|TC}$, is defined by the rule:

$$\frac{\alpha \in \Lambda_! \cup \{\delta\} \cup \Lambda_? \quad q_1 \xrightarrow{\alpha}_{\Delta(I)} q_2 \quad q_1' \xrightarrow{\alpha}_{TC} q_2'}{(q_1, q_1') \xrightarrow{\alpha}_{\Delta(I)\|TC} (q_2, q_2')}$$

It then follows that

$$\mathit{Traces}(\Delta(I)\|\mathit{TC}) = \mathit{STraces}(I) \cap \mathit{Traces}(\mathit{TC}) = \mathit{STraces}(I) \cap \mathit{Traces}(\mathcal{TC})$$

For $P \in \{\mathsf{Fail}, \mathsf{Pass}, \mathsf{Inconc}\}$, one also gets

$$\mathit{Traces}_{Q^I \times P}(\Delta(I)\|\mathit{TC}) = \mathit{STraces}(I) \cap \mathit{Traces}_P(\mathit{TC})$$

The main result of the execution of a test case on an implementation is the verdict that it emits. As this execution is not fully determined due to non-controlable choices made by the implementation, this execution may result in several traces. We then formalize the potential failure of a test case on an implementation as the fact that one of these executions may lead to a Fail verdict:

$$\mathit{TC}\ \mathit{mayfail}\ I \triangleq \mathit{Traces}_{Q^I \times \mathsf{Fail}}(\Delta(I)\|\mathit{TC}) \neq \emptyset$$

which is equivalent to $\mathit{STraces}(I) \cap \mathit{Traces}_{\mathsf{Fail}}(\mathit{TC}) \neq \emptyset$.

Test case properties:

Basic properties are essential to guarantee that selected test cases are related to the conformance relation. A set of test cases TS is said *complete* if it is both *sound* and *exhaustive* where:

- TS is *sound* $\triangleq \forall I : (I\ \mathsf{ioco}\ S \implies \forall \mathit{TC} \in \mathit{TS} : \neg(\mathit{TC}\ \mathit{mayfail}\ I))$, i.e., only non-conformant implementations may be rejected by a test case in TS.
- TS is *exhaustive* $\triangleq \forall I : (\neg(I\ \mathsf{ioco}\ S) \implies \exists \mathit{TC} \in \mathit{TS} : \mathit{TC}\ \mathit{mayfail}\ I)$, i.e., any non-conformant implementation may be rejected by a test case in TS.

As $\mathit{TC}\ \mathit{mayfail}\ I \iff \mathit{STraces}(I) \cap \mathit{Traces}_{\mathsf{Fail}}(\mathit{TC}) \neq \emptyset$ and $I\ \mathsf{ioco}\ S \iff \mathit{STraces}(I) \cap \mathit{Traces}_{\mathsf{Fail}}(\mathit{Can}(\mathcal{S})) = \emptyset$ these properties can be related to the canonical tester as follows:

$$\begin{aligned}
&\mathit{TS} \text{ is sound} && \text{iff } \textstyle\bigcup_{\mathit{TC} \in \mathit{TS}} \mathit{Traces}_{\mathsf{Fail}}(\mathit{TC}) \subseteq \mathit{Traces}_{\mathsf{Fail}}(\mathit{Can}(\mathcal{S})),\\
&\mathit{TS} \text{ is exhaustive} && \text{iff } \textstyle\bigcup_{\mathit{TC} \in \mathit{TS}} \mathit{Traces}_{\mathsf{Fail}}(\mathit{TC}) \supseteq \mathit{Traces}_{\mathsf{Fail}}(\mathit{Can}(\mathcal{S})).
\end{aligned}$$

This means that sound test cases are sub-observers of the canonical tester, and that an exhaustive test suite should capture exactly the same non-conformant behaviors as the canonical tester. Clearly, the canonical tester is by itself a complete test case. In some sense, this is the idea used in the TorX tool [2] for on-line testing for ioLTS models: the tool either non-deterministically chooses an input of the canonical tester to feed the implementation or checks that the outputs produced by the implementation does not reach Fail in the canonical tester, or stops by giving a Pass verdict. Traces of these executions form sound test cases, but exhaustiveness is in general lost when considering finite executions.

In off-line test selection, one renounces to exhaustiveness and selects test cases among all possible sound ones. The non-deterministic algorithm of [25] can then be understood as the production of a test case by an unfolding of the canonical tester. As will be seen later, one can also use test purposes to select test cases that focus on some particular behaviors. In both cases, even if exhaustiveness is lost by any finite set of test cases, one can still prove that the (inifnite) set of all test cases that could be produced is exhaustive. which guarantees that for any non-conformant implementation, there is a possibility to detect it by one test case produced by the algorithm.

4 Test selection

The test selection algorithm proposed in this paper is based on the notion of test purpose. In practice, a test purpose attached to a test case specifies the intention of the test case. The formalization of a test purpose by an ioSTS observer allows to specify abstract behaviors one is interested to test.

Definition 4.1 [Test purpose] For a specification $\mathcal{S} = (V, \Theta, \Sigma = \Sigma_! \cup \Sigma_?, T)$, a *Test Purpose* is a deterministic ioSTS $\mathcal{TP} = (V^{TP}, \Theta^{TP}, \Sigma^\delta, T^{TP})$ such that:

- $V_x^{TP} = V_i^S$: test purposes are allowed to observe the internal state of $\mathcal{S}$;
- $V_i^{TP} \cap V_i^S = \emptyset$ and V_i^{TP} contains a program counter variable pc^{TP} with $\texttt{accept} \in \mathcal{D}_{pc^{TP}}$. Its set of accepting states is $\mathsf{Accept} = (pc^{TP} = \texttt{accept})$.
- $\mathcal{TP}$ should be *complete* except when $pc^{TP} = \texttt{accept}$, which means that for any action $a \in \Sigma^\delta$, $pc^{TP} \neq \texttt{accept} \Rightarrow \bigvee_{(a,\mathbf{p},G,A) \in T^{TP}} G = \mathit{true}$. This ensures that $\mathcal{TP}$ does not restrict the runs of $\mathcal{S}$ before they are accepted (if ever).

Example.

A test purpose for the lift-controller is described by Fig. 3. The dashed transitions are those that are automatically added by the STG tool, e.g., loops in locations labelled by actions that do not appear as outgoing transitions, and transitions to a *Sink* location guarded by the negation of the union of guards of the same action in outgoing transitions. The intention of this test purpose is to select behaviors where a first $\mathit{Stop}(p)$ occurs at a certain level l storing the value of p, and a second $\mathit{Stop}(p)$ occurs at a level p which is exactly the half of l $(2p = l)$, and less than the third of h $(3p \geq h)$. One immediately notice that l should be even.

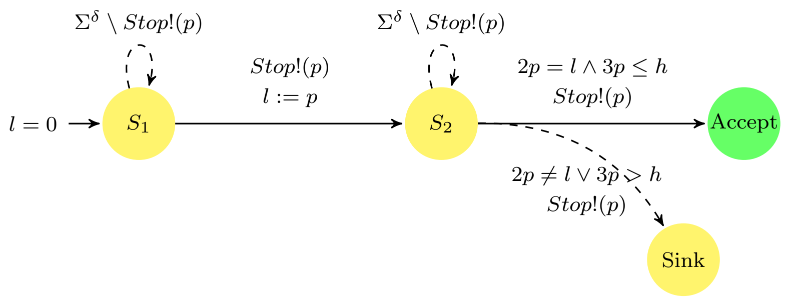

Fig. 3. A test purpose

A test purpose, with its set of accepting states allows to recognize behaviors of the specification. More precisely, it allows to recognize runs. In the case of

automata or ioLTS, a synchronous product is used to capture this idea. Similarly, a synchronous product for ioSTS can be defined. This synchronous product is here specialized for the canonical tester $Can(\mathcal{S}) = (V^{Can}, \Theta^{Can}, \Sigma^{\delta}, T^{Can})$ and a test purpose $\mathcal{TP} = (V^{TP}, \Theta^{TP}, \Sigma^{\delta}, T^{TP})$ with $V_x^{TP} = V_i^{Can}$:
The *synchronous product* of $Can(\mathcal{S})$ and $\mathcal{TP}$ is the ioSTS $\mathcal{P} = Can(\mathcal{S}) \times \mathcal{TP} = (V^P, \Theta^P, \Sigma^{Can}, T^P)$ where:

- $V^P = V_i^P \cup V_x^P$, with $V_i^P = V_i^{Can} \cup V_i^{TP}$ and $V_x^P = \emptyset$;
- $\Theta^P(\langle \boldsymbol{v}^{Can}, \boldsymbol{v}^{TP} \rangle) = \Theta^{Can}(\boldsymbol{v}^{Can}) \wedge \Theta^{TP}(\boldsymbol{v}^{TP})$;
- T^P is defined by the following inference rule:

$$\frac{\begin{array}{c}[\,a(\boldsymbol{p}) \,:\, G^c(\boldsymbol{v}^c, \boldsymbol{p})\,?\,(\boldsymbol{v}_i^c)' := A^c(\boldsymbol{v}^c, \boldsymbol{p})\,] \in T^{Can} \\ \left[\,a(\boldsymbol{p}) \,:\, G^t(\boldsymbol{v}^t, \boldsymbol{p})\,?\,(\boldsymbol{v}_i^t)' := A^t(\boldsymbol{v}^t, \boldsymbol{p})\,\right] \in T^{TP}\end{array}}{\begin{array}{c}\left[a(\boldsymbol{p}) \,:\, G^c(\boldsymbol{v}^c, \boldsymbol{p}) \wedge G^t(\boldsymbol{v}^t, \boldsymbol{p})\,? \right. \\ \left. (\boldsymbol{v}_i^c)' := A^c(\boldsymbol{v}^c, \boldsymbol{p}), (\boldsymbol{v}_i^t)' := A^t(\boldsymbol{v}^t, \boldsymbol{p})\right] \in T^P\end{array}}$$

We denote $\mathcal{P}'$ the ioSTS obtained by adding the assignment $\mathsf{Verdict} := \mathtt{pass}$ to all transitions with assignment $pc' := \mathtt{accept}$.

Example.

The synchronous product $\mathcal{S} \times \mathcal{TP}$ for the lift controller is represented in Fig. 4. For simplicity, we did not represent the ioSTS for $\mathcal{P}' = Can(\mathcal{S}) \times \mathcal{TP}$. $\mathcal{P}'$ would have the same structure, plus transitions to a Fail location labelled with actions with negations of guards of outgoing transitions in $\mathcal{S} \times \mathcal{TP}$ as in the canonical tester.

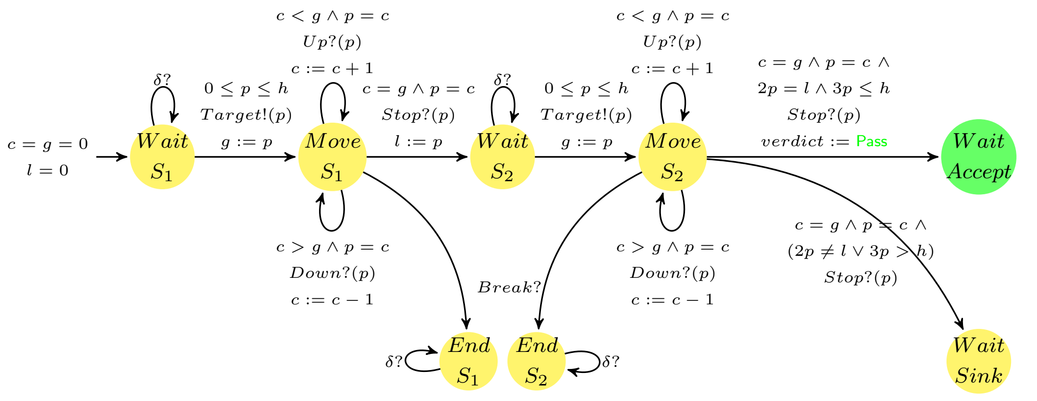

Fig. 4. Synchronous product $\mathcal{S} \times \mathcal{TP}$

As $\mathcal{TP}$ is non-intrusive (it observes but does not modify variables of $\mathcal{S}$), one gets

$$Traces(\mathcal{P}') \subseteq Traces(Can(\mathcal{S})) \text{ and } Traces_{\mathsf{Fail}}(\mathcal{P}') = Traces(\mathcal{P}') \cap Traces_{\mathsf{Fail}}(Can(\mathcal{S}))$$

This means that $\mathcal{P}'$ detects every non-conformance along its traces. It is thus a sound test case. One also gets

$$Traces_{\mathsf{Pass}}(\mathcal{P}') = Traces_{\mathsf{Accept}}(\mathcal{P}) \subseteq STraces(\mathcal{S}) \cap Traces_{\mathsf{Accept}}(\mathcal{TP})$$

The inclusion becomes an equality if $\mathcal{TP}$ does not observe variables of $\mathcal{S}$.

Note that $\mathcal{P}'$ has two distinguished rôles: it is both an observer of non-conformant suspension traces with its Fail states, and an observer of the abstract behaviors one wants to test with its Pass states. But no test selection has been performed yet. An ideal selection would mean to select $Traces_{\mathsf{Pass}}(\mathcal{P}')$, and, along these traces, unspecified outputs leading to Fail. However, systems are not fully controllable: outputs are not fully determined by inputs because of choices made by the system. This entails that all specified outputs have to be considered after a trace: outputs from which Pass is reachable or Fail is reached, but also outputs after which Pass is not reachable anymore, which is denoted by the Inconc verdict And in this last case, one would like to detect the divergence as soon as possible, i.e., on the first output where this occurs.

This amounts to computing the set of states from which Pass is reachable, denoted $coreach(\mathsf{Pass})$, which is the least fix-point of an monotonic function in the lattice 2^Q of subsets of Q: $coreach(\mathsf{Pass}) = \mathrm{lfp}(\lambda X.\mathsf{Pass} \cup pre(X))$ where $pre(X) = \{q \mid \exists q' \in X, \exists \alpha \in \Lambda : q \xrightarrow{\alpha} q'\}$ is the set of states from which X can be reached in one transition.

In the case of finite ioLTS, $coreach(\mathsf{Pass})$ is easily computed with graph algorithms (computation of strongly connected components), as implemented in the TGV tool [14]. However the set $coreach(\mathsf{Pass})$ is not computable for ioSTS. A solution then consists in computing an over-approximation. This is the idea used in the STG tool, with the help of the NBAC tool, a verification tool based on abstract interpretation. How this computation is performed by NBAC is not detailled (other techniques and tools could compute such an approximation) but we assume that an over-approximation has been computed, and explain how this approximation is used in the selection of test cases. The main idea of test selection is to transform $\mathcal{P}' = (V^{P'}, \Theta^{P'}, \Sigma^{Can}, T^{P'})$ together with its set of Fail and Pass states into a test case $\mathcal{TC}$, by reinforcing the guards of transitions of $\mathcal{P}'$ in order to try to stay in $coreach(\mathsf{Pass})$. As this set is not computable, this set and thus the guards are over-approximated.

Given a set of states $X \in \mathcal{D}_{\boldsymbol{v}}$ represented by a formula $X(\boldsymbol{v})$, let $pre(A)(X)(\boldsymbol{v}, \boldsymbol{p})$ denote the precondition of X by an assignment $A : \mathcal{D}_{\boldsymbol{v}} \times \mathcal{D}_{\boldsymbol{p}} \to \mathcal{D}_{\boldsymbol{v}}$:
$pre(A)(X)(\boldsymbol{v}, \boldsymbol{p}) \;=\; \exists \boldsymbol{v'} : X(\boldsymbol{v'}) \wedge \boldsymbol{v'} = A(\boldsymbol{v}, \boldsymbol{p}) \;=\; X(A(\boldsymbol{v}, \boldsymbol{p}))$

In other words, $pre(A)(X)(\boldsymbol{v}, \boldsymbol{p})$ represents the set of values of variables $\boldsymbol{v}$ and parameters $\boldsymbol{p}$ from which X is reached after the assignment A. Note that the operator $pre(A)$ is monotone. Assume that one can compute a monotone over-approximation$pre^{\alpha}(A)(X) \supseteq pre(A)(X)$ of $pre(A)(X)$.

Assume now that an over-approximation $coreach^{\alpha}$ of $coreach(\mathsf{Pass})$ has been computed. This computation typicaly uses the operation $pre^{\alpha}(A)(X)$, together with a widening operator which guarantees the termination of fix-point iterations. We assume that $coreach^{\alpha}$ is represented by a predicate.

The predicate $pre^{\alpha}(A)(coreach^{\alpha})$ is then an over-approximation of the set of values for variables and parameters which allow to stay in $coreach(\mathsf{Pass})$ when taking a transition $(a, \boldsymbol{p}, G, A)$. In other words it is a *necessary condition* to stay in $coreach(\mathsf{Pass})$. Its negation is thus a *sufficient condition* to leave $coreach(\mathsf{Pass})$. We use this to reinforce the guards and compute a test case from $\mathcal{P}'$.

The test case for $\mathcal{S}$ and $\mathcal{TP}$ is the ioSTS $\mathcal{TC} = (V^{P'}, \Theta^{P'}, \Sigma^{Can}, T^{\mathcal{TC}})$ where $T^{\mathcal{TC}}$ is defined from $\mathcal{P}'$ by the three rules:

$$\frac{(a, \boldsymbol{p}, G, A) \in T^{P'} \quad a \in \Sigma_!^{Can} \qquad G' = pre^{\alpha}(A)(coreach^{\alpha})}{(a, \boldsymbol{p}, G \wedge G', A) \in T^{\mathcal{TC}}} \quad \text{(Select)}$$

$$\frac{(a, \boldsymbol{p}, G, A) \in T^{P'} \quad a \in \Sigma_?^{Can} \quad A_{\mathsf{Verdict}} = \mathsf{Verdict}' := \mathtt{fail}}{(a, \boldsymbol{p}, G, A) \in T^{\mathcal{TC}}} \quad \text{(Fail)}$$

$$\frac{(a, \boldsymbol{p}, G, A) \in T^{P'} \quad a \in \Sigma_?^{Can} \quad A_{\mathsf{Verdict}} \neq \mathsf{Verdict}' := \mathtt{fail} \qquad G' = pre^{\alpha}(A)(coreach^{\alpha})}{(a, \boldsymbol{p}, G \wedge G', A), (a, \boldsymbol{p}, G \wedge \neg G', A') \in T^{\mathcal{TC}}} \quad \text{(Split)}$$

$$\text{where } A' \text{ is defined by } \begin{cases} A'_{\mathsf{Verdict}} = \mathsf{Verdict}' := \mathtt{inconc}, \\ A'_v = A_v \text{ for } v \neq \mathsf{Verdict}, \end{cases}$$

The transformation treats inputs and outputs differently, because the tester controls its outputs, but only observes its inputs. The first rule (Select) is for outputs. It consists in reinforcing the guard with $G' = pre^{\alpha}(A)(coreach^{\alpha})$ in order to keep a chance to stay in $coreach(\mathsf{Pass})$ after the transition. Implicitly this rule forbids all outputs that would certainly leave $coreach(\mathsf{Pass})$. The two other rules are concerned with non-controlable inputs. The first one (Fail) keeps all transitions with Fail verdict. The second splits transitions labelled with inputs into two new transitions: one by which we may stay in $coreach(\mathsf{Pass})$ by reinforcing the guard with G', the other one by which we are sure to leave $coreach(\mathsf{Pass})$ by reinforcing the guard with the negation of G'. In this second case an Inconc verdict is set.

The test case can be further simplified with an over-approximation $reach^{\alpha}(\Theta^{P'})$ of its reachable states $reach(\Theta^{P'})$ where $reach(\Theta^{P'}) = \mathrm{lfp}(\lambda X.\Theta^{P'} \cup post(X))$ with $post(X) = \{q' \mid \exists q \in X, \exists \alpha \in \Lambda : q \xrightarrow{\alpha} q'\}$ being the set of states reachable from X in one transition. The simplification consists in removing transitions of which the guards are unsatisfiable in the over-approximation $reach^{\alpha}(\Theta^{P'})$ of the set of reachable states i.e., transitions $(a, \boldsymbol{p}, G, A)$ where $G \wedge reach^{\alpha}(\Theta^{P'})$ simplifies to `false`. Note that, this transformation is purely syntactic: it does not modify the semantics of TC as it only removes infeasible transitions.

Example.

The results of the backward and forward analysis computed by the NBAC tool are described in the following table:

location	$coreach^\alpha$	$reach^\alpha$
$Wait, S_1$	$h \geq 0$	$c = g = 0$
$Move, S_1$	$3g \leq 2h$	$0 \leq c \leq g \wedge 3g \leq 2h$
End, S_1	$false$	$0 \leq c \leq h \wedge c \leq g + l \leq h + c \wedge 0 \leq l \leq h \wedge 0 \leq g \leq h$
$Wait, S_2$	$3c \leq 2h$	$0 \leq c = g = l \wedge 3c \leq 2h$
$Move, S_2$	$3g \leq h$	$g \leq c \leq 2g = l \wedge 3g \leq h$
End, S_2	$false$	$0 \leq c \leq 2g \wedge g \leq 2h \wedge 3c \leq 2h$
$Wait, Sink$	$false$	$false$
$Wait, Accept$	$true$	$0 \leq c = g \wedge 2c = l \wedge 3c \leq h$

Notice that the analysis is not exact as the fact that the first target level should be even is lost: NBAC is not able to infer this as it restricts its analysis to polyhedra, and parity is not a linear constraint.

As a consequence of these analyses, the resulting test case is computed, as shown in Fig. 5 (for clarity, transitions to Fail are not represented but can be obtained by complementation of guards). The coreachability analysis allows to constraint the guards of transitions. In particular for outputs, the first transition carrying $Target(p)$ is enforced by $3p \leq 2h$ as $pre^\alpha(g := p)(3g \leq 2h) = 3p \leq 2h$, and the second one by $3p \leq h \wedge 2p = l$ as $pre^\alpha(g := p)(g \leq c \leq 2g = l \wedge 3g \leq h \wedge) = 3p \leq h \wedge 2p = l$.

The reachability analysis then allows to cut non-fireable transitions. In location $Move, S_1$ (resp. $Move, S_2$) the transition carrying $Down?(p)$ (resp. $Up?(p)$) is not fireable as its guard $c > g \wedge p = c$ (resp. $c < g \wedge p = c$) is unsatisfiable when $0 \leq c \leq g$ (resp. $g \leq c \leq 2g$). Similarly the guard $c = g \wedge p = c \wedge 2p \neq l \vee 3p > h$ of the transition from $Move, S_2$ to $Wait, Sink$ is unsatisfiable when $g \leq c \leq 2g \wedge 3g \leq h$.

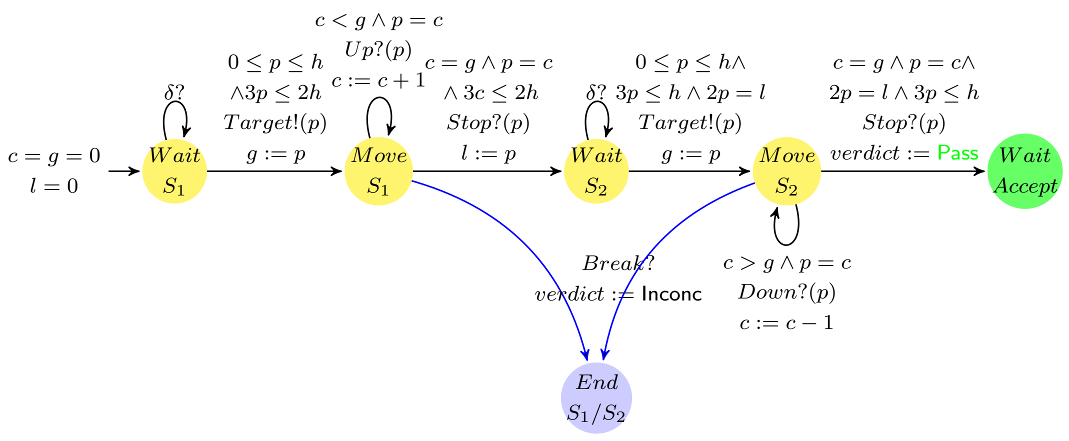

Fig. 5. Selected test case

Properties of selected test cases

As expected, it can be proved that the (infinite) set of test cases that can be produced by the test selection algorithm is sound and exhaustive. Soundness directly comes from the soundness of $Can(S)$, which is preserved by synchronous product and selection (these transformation do not add any trace leading to Fail). For exhaustiveness, the idea is, for any non-conformant trace $\sigma.a \in NC\_STraces(()S) = Traces_{\mathsf{Fail}}(Can(S))$, to identify a test purpose $\mathcal{TP}$ for which the selection produces a test case $\mathcal{TC}$ such that $\sigma.a \in Traces_{\mathsf{Fail}}(\mathcal{TC})$. One chooses an output b such that $\sigma.b \notin Traces_{\mathsf{Fail}}(Can(S))$ (the existence of b is ensured as $Can(S)$ has no quiescence), and builds an test purpose accepting $\sigma.b$. By construction the test case $\mathcal{TC}$ built from $\mathcal{TP}$ then reaches Fail for the trace $\sigma.a$.

Apart soundness and exhaustiveness which relate Fail verdicts to conformance, other properties related to verdicts Pass and Inconc are also of importance as they relate test cases to the selection means, namely test purposes. It can be proved that Pass verdicts are always exact: a Pass is always emitted when the current trace is accepted by the test purpose. The only verdict where the over-approximation may cause a lost in precision is Inconc. In fact, it may be the case that some traces of $\mathcal{TC}$ are not anymore a prefix of an accepted trace, but is not detected as such. It will either be detected by continuation of the trace, or will loop. It is easy to see that the precision of the approximation directly influences the ability of test cases to emit Inconc verdicts accurately. For the reader familiar with structural testing, the problem of detecting Inconc verdicts is similar to the classical problem of the undecidability of the feasability of a path in structural testing, which is known to be undecidable.

5 Test execution

In most test generation techniques, generated test cases are completely instanciated test cases in the form of sequences (in the deterministic case), trees or graphs where actions carry some valued parameters. In our case, generated test cases are kinds of programs described as ioSTS, thus have variables, guards, actions and assignments. During test execution, the test harness will start the test case with its unique initial state. Then the system will progress with transitions carrying either inputs or outputs. The tester can choose a transition carrying an output if one is fireable. In this case, as values of variables are known, it should choose a value of parameters such that the guard is satisfiable. This is where satisfiability of guards should be decidable. If there is none, an input (or quiescence should be observed). The choice of parameters values should be done by constraint solving. An input is fireable only if it is sent by the implementation under test. The tester then has to check, according to the current value of variables and the value of the input parameters, which guard is satisfiable among transitions carrying this input. As the test case is input-complete in any state with no verdict, exactly one transition is fireable.

Example.

In the lift-controller example, the execution proceeds as follows. Let us fix the constant $h = 10$. The tester chooses a value for p for Target such that $0 \leq p \leq$

$10 \wedge 3p \leq 20$. Assume that 4 is chosen, then the tester will observe the trace $Up?(0).Up?(1).Up?(2).Up?(3).Stop?(4)$ if the implementation conforms to $\mathcal{S}$ but does not emit $Break$. The tester will then choose 2 as the unique solution of the guard $0 \leq p \leq 10 \wedge 2p = 4 \wedge 3p \leq 20$. The lift will then go down and the tester is supposed to observe $Down?(4).Down?(3).Stop?(2)$.

If 3 was chosen as a solution to $0 \leq p \leq 10 \wedge 3p \leq 20$ for the first level to reach, the tester would observe $Up?(0).Up?(1).Up?(2).Stop?(3)$. After that, there is no solution to $0 \leq p \leq 10 \wedge 2p = 5 \wedge 3p \leq 20$ and the tester can only observe a quiescence $?\delta$. This situation is due to the approximation, as the constraint on the first $Target$ does not ensure that p is even.

6 Conclusion and perspectives

This paper proposes an approach to the off-line selection of test cases from specification models with control and data (ioSTS) using test purposes. This technique avoids the state explosion problem due to the enumeration of data values and produces test cases on the form of programs. Test selection reduces to syntactical operations on these models and relies on an over-approximate analysis of the co-reachable states to a target location. 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 restricted to deterministic specifications. However, non-determnistic specifications can be taken into account with some restrictions [18].

In our perspectives of this work, more powerful models of systems with features such as time, recursion and concurrency should be considered. A similar approach has been developed for stack automata modeling recursive programs and test purposes specified as automata [7]. In this case, the analyses are exact, but cannot be fully used if test cases cannot observe their own stack, thus also inducing an approximation. For test generation, one problem to address in these models is partial observability, which, as for ioSTS, entails the identification of determinizable sub-classes corresponding to applications.

Other challenges are the combination of these techniques with coverage-based test selection. One direction should be to use the dynamic partitioning facility provided by the tool Nbac used by STG as an aid for test selection with respect to coverage criteria with a deeper semantic meaning. More generally, conformance testing appeals for more semantic based coverage criteria.

References

[1] T. Ball. A theory of predicate-complete test coverage and generation. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Third International Symposium on Formal Methods for Components and Objects (FMCO'04), Leiden, The Netherlands, November 2 - 5, 2004, Revised Lectures*, volume 3657 of *LNCS*. Springer, 2005.

[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 study in coverage-driven test generation. In *Proceedings of the 36th ACM/IEEE Conference on Design Automation (DAC'99)*, 1999.

[4] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT: a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM Press.

[5] 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.

[6] J.R. Calamé, N. Ioustinova, J. van de Pol, and N. Sidorova. Data abstraction and constraint solving for conformance testing. In *Proc. of 12th Asia-Pacific Software Engineering Conference (APSEC'05), Taipei, Taiwan*, pages 541–548, 2005.

[7] C. Constant, B. Jeannet, and T. Jéron. Automatic test generation from interprocedural specifications. In *TestCom/Fates07*, number 4581 in LNCS, pages 41–57, Tallinn, Estonia, June 2007.

[8] 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 (POPL'77), Los Angeles, CA*, pages 238–252, 1977.

[9] T. Denmat. *Contraintes et abstractions pour la génération automatique de données de test.* PhD thesis, INSA Rennes, June 2008.

[10] C. Gaston, P. Le Gall, N. Rapin, and A. Touil. Symbolic execution techniques for test purpose definition. In *The 18th IFIP International Conference on Testing Communicating Systems (TestCom'06)*, 2006.

[11] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI'05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 213–223, New York, NY, USA, 2005. ACM Press.

[12] E. Gunter and D. Peled. Model checking, testing and verification working together. *Formal Aspects of Computing*, 17(2):201–221, August 2005.

[13] W. E. Howden. Theoretical and empirical studies of program testing. In *Proceedings of the 3rd international conference on Software engineering (ICSE '78)*, pages 305–311, Piscataway, NJ, USA, 1978. IEEE Press.

[14] C. Jard and T. Jéron. TGV: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)*, 6, octobre 2004.

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

[16] B. Jeannet, T. Jéron, and F. Ployette. STG: a symbolic test generation tool for reactive systems. In *TestCom/Fates07, Tool paper*, Tallinn, Estonia, June 2007.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] N. Lynch and M. Tuttle. Introduction to IO automata. *CWI Quarterly*, 3(2), 1999.

[23] B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions: GATEL. In *15th IEEE International Conference on Automated Software Engineering (ASE'00), Los Alamitos, CA, USA*, page 229. IEEE Computer Society, 2000.

[24] V. Rusu, L. du Bousquet, and T. Jéron. An approach to symbolic test generation. In *Integrated Formal Methods (IFM'00), Dagstuhl, Allemagne, LNCS 1945*, volume 1945 of *LNCS*, pages 338–357. Springer Verlag, Novembre 2000.

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