Automatic Test Generation from Interprocedural Specifications

Camille Constant[1], Bertrand Jeannet[2] and Thierry Jéron[1] ⋆

[1] IRISA/INRIA, Campus de Beaulieu, Rennes, France,
`{constant, jeron}@irisa.fr`
[2] INRIA Rhône-Alpes, Saint Ismier, France,
`Bertrand.Jeannet@inrialpes.fr`

Abstract. This paper adresses the generation of test cases for testing the conformance of a reactive black-box implementation with respect to its specification. We aim at extending the principles and algorithms of model-based testing for recursive interprocedural specifications that can be modeled by Push-Down Systems (PDS). Such specifications may be more compact than non-recursive ones and are more expressive.
The generated test cases are selected according to a test purpose, a (set of) scenario of interest that one wants to observe during test execution. The test generation method we propose in this paper is based on program transformations and a coreachability analysis, which allows to decide whether and how the test purpose can still be satisfied. However, despite the possibility to perform an exact analysis, the inability of test cases to inspect their own stack prevents it from using fully the coreachability information. We discuss this partial observation problem, its consequences, and how to minimize its impact.

1 Introduction

Testing is the most used validation technique to assess the correctness of reactive 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 [1]. Conformance testing compares the observable behaviour of an actual black-box implementation of the system with the observable behaviour described by a formal specification, according to a conformance relation. It is an instance of *model-based testing* where specifications, implementations, and the conformance relation between them are formalised. Test cases are automatically derived from specifications, and the verdicts resulting from test execution on an implementation are proved to be consistent with respect to the conformance relation. Moreover, in addition to checking the conformance of the implementation, the goal of the test case is also to guide the test execution towards the satisfaction of a test purpose, typically a set of scenarii of interest. The *test selection* problem consists in finding a strategy that maximizes the likehood for the test case to realize the test purpose.

⋆ This work was partly supported by France Telecom R&D, contract 46132862

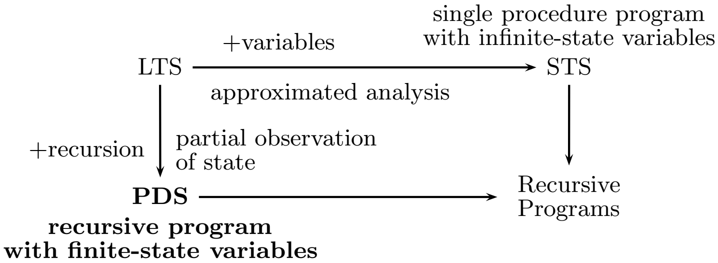


Fig. 1. Test selection on various models

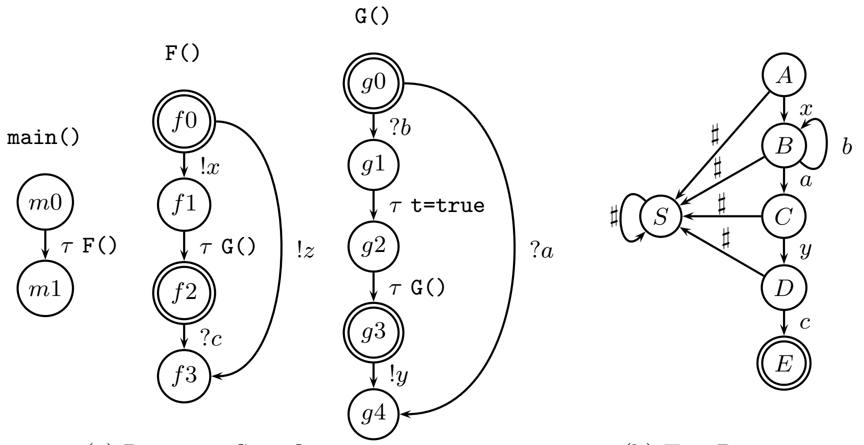
This problem has been previously addressed in the case where the specifications, the test cases and the test purposes are modeled with finite Labelled Transition Systems (LTS) [2, 3]. It was more recently addressed in the case where the same objects are modeled with Symbolic Transition Systems (STS), which extend LTS with infinite datatypes and can model non-recursive imperative programs [4]. The aim of this paper is to address the test selection problem in the case where the specification is modeled as a Push-Down System (PDS), which extends LTS with a stack over a finite alphabet and can model recursive programs manipulating finite datatypes. Such specifications may be more compact that non-recursive ones and are more expressive than single procedure programs. Fig. 1 summarizes the different models.

Outline and Contributions. We first illustrate our test selection methodology on an example in Sect. 2. Then, we recall in Sect. 3 the testing theory framework we use. Next, we present our contribution which is twofold:

– First, we describe in Sect. 4 a test generation method, that takes as input a recursive specification and a non-recursive test purpose and that returns a recursive test case. This method is based on program transformations. Technical choices are guided by theoretical properties of the underlying PDS and LTS models, but the generation is defined in terms of programming language concepts.
– Second, we present in Sect. 5 a selection algorithm which takes as input the previously generated test case and specializes it. This algorithm is based on a (co)reachability analysis of the test case. We formalize a partial observation problem, due to the inability of test cases to inspect their own stack. We compare its consequences on the generated test cases with the impact of using a non-exact, over-approximated coreachability analysis as done for test selection based on symbolic STS models [4]. We also propose an improvement of the selection algorithm to minimize the negative impact of this partial observation aspect on test selection.

2 Introductive Example

We illustrate in this section the concepts we will develop and our testing methodology on a running example, before formalizing it in the next sections.



(a) Recursive Specification, recognizing the traces $xb^n ay^n c + z$

(b) Test Purpose, recognizing the traces $xb^* ayc$

Fig. 2. Control-flow Graphs of Specification and Test Purpose

Specification. In our testing theory, the *IUT* is considered as a black box reactive system, and its observations are the messages exchanged with its environment. The specification we consider as an example is the small recursive program of Fig. 4. Its control flow graph is given on Fig. 2(a). Double circles denote the observation points (see Sect. 4). Inputs and outputs are distinguished by the symbols ? and ! (inputs and outputs alphabets are disjoints). The behavior of this specification program is the following: the main function calls the function $F()$, which either emits the output $!z$ and returns, or emits the output $!x$, calls the function $G()$, then receives the input c and returns to its caller. According to its first input, the function $G()$ has two different behaviors: if the input is $?a$, it returns to the caller, whereas if the input is $?b$, the function $G()$ is called again (recursively) and after it returns, the output $!y$ is emitted.

What is really important is the traces recognized (or generated) by the specification, which is here $\mathit{Traces}(S) = \{!z\} \cup \{!x \cdot (?b)^n \cdot ?a \cdot (!y)^n \cdot c? \mid n \geq 0\}$, a context free language. The (non-)conformance of an *IUT* w.r.t. a specification S will be only based on $\mathit{Traces}(S)$. Intuitively, an *IUT* will be defined as *conformant* to S if after any execution trace which is a prefix of $\mathit{Traces}(S)$, it emits only outputs that S can emit as well. Note that the global variable t has no influence on $\mathit{Traces}(S)$, its usefulness will be explained later (see Sect. 5).

```
void main(){
   emit(x) [] receive(p);
   emit(y) [] receive(p);
}
```

Fig. 3. Non-conformant *IUT*

For instance, the *IUT* besides (where `[]` stands for the non-deterministic choice operator) is not conformant to S. After the execution trace $!x$, it may emit $!y$, whereas S specifies that no output may be emitted at this point: one or more $?b$ and then one $?a$ should be received first). Note that on all states, *IUT* is ready to receive any input on the parameter p.

```
enum out_t { x,y,z };
enum inp_t { a,b,c };


bool t = false;
void main(){
  f();
}

void f()
{
  f0: emit(p) when (p == x || p == z){


        if (p==z) goto f3;
      }
  f1: g();
  f2: receive(p) when (p == c) {};



  f3:
}

void g()
{
  g0: receive(p) when (p == a || p == b){
        if (p == a) goto g4;
      };



  g1: t = true;
  g2: g();
  g3: emit(p) when (p == y) {}


  g4:
}
```

Fig. 4. Specification corresponding to Fig. 2

```
enum out_t { a,b,c };
enum inp_t { x,y,z };
enum verdict_t { none, fail, pass, inconc };
enum verdict_t verdict = none;
bool t = false;
void main(){
  f();
}

void f()
{
  f0: receive (p) when true {
        if (p != x && p != z)
          { verdict = fail; abort(); }
        if (p == z) goto f3;
      };
  f1: g();
  f2: emit(p) when (p == c) {}
      []
      receive(p) when true
        { verdict = fail; abort() };
  f3:
}

void g()
{
  g0: emit(p) when (p == a || p == b){
        if (p == a) goto g4;
      }
      []
      receive(p) when true
        { verdict = fail; abort() };
  g1: t = true;
  g2: g();
  g3: receive(p) when true {
        if (p != y){ verdict = fail; abort(); }
      }
  g4:
}
```

Fig. 5. Canonical tester associated to the specification of Fig. 4

Canonical Tester. A tester, called *canonical tester* can be generated from S very straightforwardly, according to the (yet intuitive) definition of conformance. The name *canonical tester* stems from the fact that it can detect any non-conformant execution of the implementation. It is actually the most general tester, from which any sound test case can be derived. The program transformation consists in mirroring inputs into outputs and *vice-versa*, and to emit a failure verdict when the transformed program receives an unexpected input at an observation point. Fig. 5 gives the canonical tester $\mathrm{Can}(S)$ associated to S. A new type and a global variable `verdict` have been introduced for storing the verdict. $\mathrm{Can}(S)$ stimulates the IUT by sending to it input messages, and checks that the outputs of the IUT, which correspond to its own inputs, are conformant w.r.t. S.

If this canonical tester is run in parallel with the non-conformant program of Fig 3, and if the conformant program chooses to emit $!x$ and then $!y$, the tester will perform the execution $m_0 \xrightarrow{\tau} f_0 \xrightarrow{?x} f_1 \xrightarrow{\tau} g_0$ and will reach location g_0, where it will receive an unexpected $?y$ input and will abort.

Notice that not only the (canonical) tester, but also the example IUT accept any input at an observation point. These are usual asumptions in the ioco theory. For the tester, the reason is that it should check any output from the IUT for conformance. For the IUT, this allows to prevent deadlocks.

Test purpose. For large specifications, the canonical tester is too general. It tests the IUT in a completely random way. One is often more interested in guiding the execution of the IUT so as to realize a specific scenario that may reveal an error, and to stop the test execution successfully when the scenario has been completed without conformance error.

In this context, a test purpose is a (set of) scenario one wants to observe during a *conformant* test execution. The test purpose depicted as an automaton on Fig. 2 specifies that one is interested in detecting conformance errors occuring along the traces in $Traces_E(TP) = xb^*ayc$. The symbol $\sharp$ means "all other elements in the alphabet" and the double circle denotes the final state E. This test purpose indicates that we want to test the case where the IUT emits $!x$ at control point f_0 and performs one recursive call of G from G.

The aim of test selection is to transform the canonical tester so that it is more likely to produce the execution trace xb^*ayc until completion when executed in parallel with the IUT. If a conformance error occurs, the tester aborts immediately with a `fail` verdict. For instance, the first time the tester enters in function `G()`, it should first emit a $!b$, because a matching $?y$ should be later received to realize the scenario. Moreover, the second time it enters (recursively) in function `G()`, it should emit an $!a$, because only one $?y$ message should be observed before $!c$.

On the other hand, if an IUT starts its execution by emitting one $!z$ (which is conformant to S), the scenario cannot be completed. The tester should detect such a case and abort gracefully with an `inconclusive` verdict.

Selected test case. Fig. 8 depicts the test case we obtain with the method we will develop in the paper. Compared to the canonical tester of Fig. 5, we have first inserted at each observation point a call to the function `TP()` (after having checked the absence of conformance error at this point). The function `TP()` defined on Fig. 6 takes as input the last message exchanged and implements the automaton of Fig. 2(b). If the final state is reached, it emits the `pass` verdict.

There are two other modifications to the canonical tester. At control point f_0, when a $?z$ is received, the `inconclusive` verdict is emitted. Last, at control point g_0, the condition for emitting a message has been enforced: $!a$ is emitted iff the variable `t` is true. Indeed, `t` allows to distinguish if `G()` is called for the first time from f_1, in which case `t` is false, or if it is called recursively from g_2, in which case `t` is true. Hence, the knowledge of the value of `t` allows the test case to realize exactly the scenario defined by the test purpose (once $?x$ has been received from the IUT).

The next sections describe the theoretical fundations of this test selection scheme sketched on the running example. After having presented the testing theory on LTS models in Sect. 3, we adapt it on recursive specifications in Sect 4.

```
enum pc_t { A,B,C,D,E,S };
enum pc_t pc = A;
void TP(enum msg_t p)
{
  if    (pc == A && p == x ) pc = B;
  elsif (pc == B && p == b)  pc = B;
  elsif (pc == B && p == a)  pc = C;
  elsif (pc == C && p == y)  pc = D;
  elsif (pc == D && p == c ){
      pc = E;
      verdict = pass;
      abort();
  }
  else pc = S;
}
```

Fig. 6. Test Purpose corresp. to Fig. 2.(b)

```
// Type and global variables Declarations
// ...
void main(){
  m0: f();
  m1:
}

void f()
{
  f0: receive (p) when true {
  f0r:  if (p != x && p != z)
          { verdict = fail; abort(); }
        TP(p);


      };
  f1: g();
  f2: emit(p) when (p == c) {
  f2e:  TP(p)
      }
      []
      receive(p) when true
  f2r:  { verdict = fail; abort(); };
  f3:
}

void g()
{
  g0: emit(p) when (p == a || p == b){

  g0e:  TP(p);
        if (p == a) goto g4;
      }
      []
      receive(p) when true
  g0r:  { verdict = fail; abort(); };
  g1: t = true;
  g2: g();
  g3: receive(p) when true {
  g3r:  if (p != y)
          { verdict = fail; abort(); }
        TP(p);
      }
  g4:
}
```

Fig. 7. Product

```
// Type and global variables Declarations
// ...
void main(){
  m0: f();
  m1:
}

void f()
{
  f0: receive (p) when true {
  f0r:  if (p != x && p != z)
          { verdict = fail; abort(); }
        TP(p);
        if (p == z)
          { verdict = inconc; abort(); }
      };
  f1: g();
  f2: emit(p) when (p == c) {
  f2e:  TP(p)
      }
      []
      receive(p) when true
  f2r:  { verdict = fail; abort(); };
  f3:
}

void g()
{
  g0: emit(p) when ((p == a && t == true)
                 || (p == b && t == false)){
  g0e:  TP(p);
        if (p == a) goto g4;
      }
      []
      receive(p) when true
  g0r:  { verdict = fail; abort(); };
  g1: t = true;
  g2: g();
  g3: receive(p) when true {
  g3r:  if (p != y)
          { verdict = fail; abort(); }
        TP(p);
      }
  g4:
}
```

Fig. 8. Test Case after selection

3 Testing Theory

The testing theory we consider is based on the notions of *specification*, *implementation*, and *conformance relation* between them [2] and on the model of *Labelled Transition Systems* (LTS). An LTS is defined by a tuple $M = (Q, Q_0, \Lambda, \rightarrow)$ where Q is a set of states, Q_0 is the set of initial states, $\Lambda = \Lambda_v \cup \{\tau\}$ is an alphabet of visible (Λ_v) and internal ($\{\tau\}$) actions and $\rightarrow \subseteq Q \times \Lambda \times Q$ is a set of labelled transitions. The notation $p \xrightarrow{a} q$ stands for $(p, a, q) \in \rightarrow$, and $p \xrightarrow{a}$ for $\exists q : p \xrightarrow{a} q$. An execution is a sequence $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \ldots q_{n+1}$ with $q_0 \in Q_0$. $\mathit{Traces}(M) \subseteq \Lambda_v^*$ denotes the projection of the set of executions of M onto visible actions. For a subset $X \subseteq Q$ of states, $\mathit{Traces}_X(M)$ denotes the projection of the set of executions of M ending in a state $q' \in X$ onto visible actions. It is also named the set of traces *accepted* by X. The set of prefixes (resp. strict prefixes) of a set of traces Y is denoted by $\mathit{pref}_{\leq}(Y)$ (resp. $\mathit{pref}_{<}(Y)$). M is called *deterministic* if Q_0 has a single element q_0, if $p \xrightarrow{\alpha} q \wedge p \xrightarrow{\alpha} q' \implies q = q'$ and if $p \xrightarrow{\tau} q \implies \neg(\exists \alpha \in \Lambda_v : p \xrightarrow{\alpha})$. Note that this definition is not the standard definition of a deterministic LTS because transitions with internal actions are possible. M is *complete* for $A \subseteq \Lambda$, if $\forall a \in A, \forall p \in Q,\ p \xrightarrow{a}$.

The specification is a deterministic LTS $S = (Q^S, Q_0^S, \Lambda, \rightarrow_S)$, and the Implementation Under Test (IUT) is assumed to be an LTS $IUT = (Q^{IUT}, Q_0^{IUT}, \Lambda, \rightarrow_{IUT}$) 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 IUT cannot refuse an input from its environment.

In this context, a *test case* for the specification S is a deterministic LTS $TC = (Q^{TC}, Q_0^{TC},\ \Lambda, \rightarrow_{TC})$ which is able to interact with an implementation and to emit verdicts:

- its alphabet is the mirror of that of S ($\Lambda_?^{TC} = \Lambda_!^S$ and $\Lambda_!^{TC} = \Lambda_?^S$)
- it is input-complete (outputs of IUT are not refused) except in verdict states;
- it is equipped with 3 disjoint subsets of sink, *verdict* states $\mathsf{Pass}, \mathsf{Fail}, \mathsf{Inconc} \subseteq Q^{TC}$. Intuitively, Fail means rejection, Pass that some targetted behavior has been realized (this will be clarified later), and Inconc that a targetted behavior cannot be realized any more.

The conformance relation defines which implementations are considered correct w.r.t. the specification. We will consider the following conformance relation:

Definition 1 (Conformance relation). *Let $S = (Q^S, Q_0^S, \Lambda, \rightarrow_S)$ and $IUT = (Q^{IUT}, Q_0^{IUT}, \Lambda, \rightarrow_{IUT})$ be two LTS with same alphabet. A* trace σ of IUT conforms to S*, denoted by $\sigma\ \mathit{conf}\ S$, iff*

$$\mathit{pref}_{\leq}(\sigma) \cap [\mathit{Traces}(S) \cdot \Lambda_! \setminus \mathit{Traces}(S)] = \emptyset$$

IUT conforms to S*, denoted by $IUT\ \mathit{conf}\ S$, iff all its traces are conformant: $\mathit{Traces}(IUT) \cap [\mathit{Traces}(S) \cdot \Lambda_! \setminus \mathit{Traces}(S)] = \emptyset$.*

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. Except for the notion of *quiescence* (absence of outputs), *conf* corresponds to the *ioco* relation of [2].

The set of traces $Traces(S){\cdot}\Lambda_! \setminus Traces(S)$ is the set of minimal non-conformant traces. It is characterized by a test case called the *canonical tester*, which is obtained from the specification S by inversion of inputs and outputs, followed by an *input-completion*, where each unspecified input leads to Fail.

Definition 2 (Canonical Tester). *Let* $S = (Q^S, Q_0^S, \Lambda, \rightarrow_S)$ *be the deterministic LTS of the specification. The* canonical tester *of* S *for* *conf* *is the deterministic LTS* $\mathrm{Can}(S) = (Q^S \cup \mathsf{Fail}, Q_0^S, \Lambda^{\mathrm{Can}}, \rightarrow_{\mathrm{Can}})$ *such that*

- $\mathsf{Fail} = \{q_{\mathsf{Fail}}\}$*, with* $q_{\mathsf{Fail}} \notin Q^S$ *a new state;*
- *its alphabet is the mirror of that of* S ($\Lambda_?^{\mathrm{Can}} = \Lambda_!^S$ *and* $\Lambda_!^{\mathrm{Can}} = \Lambda_?^S$)
- $\rightarrow_{\mathrm{Can}}$ *is defined by the rules:*

$$\frac{q, q' \in Q^S \quad q \xrightarrow{\alpha}_S q'}{q \xrightarrow{\alpha}_{\mathrm{Can}} q'} \qquad \frac{q \in Q^S \quad \alpha \in \Lambda_!^S = \Lambda_?^{\mathrm{Can}} \qquad \neg(q \xrightarrow{\alpha}_S)}{q \xrightarrow{\alpha}_{\mathrm{Can}} q_{\mathsf{Fail}}}$$

We have
$$Traces(\mathrm{Can}(S)) = pref_{\leq}(Traces(S)\cdot\Lambda_!)$$
$$Traces_{\mathsf{Fail}}(\mathrm{Can}(S)) = Traces(S)\cdot\Lambda_! \setminus Traces(S)$$

$\mathrm{Can}(S)$ is already a test case. However, it 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.

Definition 3 (Test Purpose). *A* test purpose *for a specification* S *is a deterministic LTS* $TP = (Q^{TP}, Q_0^{TP}, \Lambda, \rightarrow_{TP})$ *equipped with a subset* $\mathsf{Accept} \subseteq Q^{TP}$ *of accepting sink states.* TP *is complete except in these* Accept *states.*

The completeness assumption allows not to constrain S in the product $S \times TP$ (unless the execution trace is accepted).

The test case should now not only detect conformance errors, but also try to satisfy the test purpose. For this, it has to take into account output choices of the specification (observable non-determinism) and to detect incorrect outputs of the IUT w.r.t. the test purpose.

The verdicts and their meanings are summarised as follows. The Fail verdict is emitted if the implementation does not conform to the specification, so *iff* TC observes an unspecified output after a trace of S. Pass means that the behavior wanted by the test purpose has been realized by the implementation. The verdict Pass is thus emitted *iff* TC observes a trace of S accepted by TP. The Fail and Pass verdicts are uniquely defined, so that they are emitted appropriately and as soon as possible, whereas the Inconc verdict is not uniquely defined. Indeed, Inconc, which means that the behavior wanted by TP cannot be realized any more, may be emitted *only if* the trace observed by TC belongs to S (it is conformant) but is refused by TP. The test execution can thus be interrupted, as Pass cannot be emitted any more. We have adopted this definition because checking whether a trace is refused is not always possible, either because it is undecidable, for instance with infinite-state symbolic model [4], or because of partial observation issues as discussed in Sect. 5. We refer to [4] for how *optimal* test cases are defined.

Test selection for LTS. We briefly recall how to generate an optimal test case from a specification S and a test purpose TP given as finite LTS [3]. One first builds the canonical tester $\mathrm{Can}(S)$ using Def. 2. One then builds the product $P = \mathrm{Can}(S) \times TP$ combining the information about conformance given by $\mathrm{Can}(S)$ and the information about the wanted scenario given by TP. One defines the set Pass of verdict states as $\mathsf{Pass} = Q^S \times \mathsf{Accept}^{TP}$. Adding the Inconc verdict is done by observing that

$$pref_{\leq}(Traces_{\mathsf{Pass}}(P)) = Traces_{coreach(\mathsf{Pass})}(P)$$

where $coreach(\mathsf{Pass}) = \{q \in Q^P \mid \exists q' \in \mathsf{Pass} : q \rightarrow^* q'\}$ denotes the set of states that may reach a state in Pass. A valid test case TC is obtained from P by adding a new state Inconc and by modifying $\rightarrow_P$ as follows:

$$\frac{q \xrightarrow{\alpha}_P q' \quad q' \in coreach(\mathsf{Pass})}{q \xrightarrow{\alpha}_{TC} q'} \qquad \frac{q \xrightarrow{\alpha}_P q' \quad \alpha \in \Lambda_?^{\mathrm{Can}(S)} \quad q' \notin coreach(\mathsf{Pass})}{q \xrightarrow{\alpha}_{TC} \mathsf{Inconc}}$$

The first rule keeps only transitions maintaining the execution in $coreach(\mathsf{Pass})$. In particular, it selects the appropriate outputs w.r.t. TP that should be sent to the IUT. As TC should remain input-complete, the second rule redirects the input transitions not selected by the first rule to the Inconc verdict, which is thus emitted as soon as the execution leaves the prefixes of accepted traces by a conformant input.

4 Modeling Recursive Specifications and Test Purposes

The previous section recalled our framework for model-based testing, based on the low-level semantics model of LTS. We already extended these principles and designed sound algorithms for infinite-state symbolic transition systems in [4]. Our aim here is to do the same for recursive specifications which can be compiled into (input/output) pushdown automata, PDS. Such specifications manipulate finite data but may have an infinite control due to the recursion, hence they are more expressive than finite LTS. In terms of traces, which is a relevant notion for the conformance relation, they generate context-free languages instead of regular languages. Moreover, even if there are cases where the recursion is bounded and the specification may be flattened into a LTS (by inlining), such a process may result in a huge LTS.

Push-Down Systems. A labelled *Push-Down System* (PDS) is defined by a tuple $\mathcal{P} = (G, \Gamma, \Lambda, c_0, \hookrightarrow)$ where G is a finite set of locations, Γ is a finite stack alphabet, $c_0 \in G \times \Gamma^*$ is the inital configuration, $\Lambda = \Lambda_v \cup \{\tau\}$ is a finite set of visible (Λ_v) and internal ($\{\tau\}$) actions, and $\hookrightarrow \subseteq (G \times \Gamma) \times \Lambda \times (G \times \Gamma^*)$ is a finite

Expressions	*expr*		
Atomic Instructions	*atom*	$::=$	$var = expr \mid if\ (expr)$ goto $label$
Interproc. Instructions	*callret*	$::=$	$proc() \mid$ return
Communications	*com*	$::=$	emit(p) when $expr\ \{block\}$
		$\mid$	receive(p) when $expr\ \{block\}$
		$\mid$	$com\ []\ com$
Instructions	*instr*	$::=$	$atom \mid callret \mid com$
Sequences	*block*	$::=$	$\epsilon \mid instr; block$

Fig. 9. Language Syntax

Control point	$k \in K$
Global environment	$g \in GEnv = GVar \to Val$
Local environment	$l \in LEnv = LVar \to Val$
Configuration	$(g, \sigma) \in C = GEnv \times (K \times LEnv)^+$

Fig. 10. Language Semantic domains

set of labelled transitions. Such a labelled PDS $\mathcal{P}$ generates an infinite LTS $M = (Q^M, Q_0^M, \Lambda, \to_M)$ where $Q^M = G \times \Gamma^*$, $Q_0^M = \{c_0\}$, and $\to$is defined by the rule:

$$(g, \gamma) \overset{\alpha}{\hookrightarrow} (g, \gamma') \wedge \omega \in \Gamma^* \implies (g, \omega \cdot \gamma) \xrightarrow{\alpha} (g, \omega \cdot \gamma')$$

The notions of deterministic and complete PDS are defined in terms of LTS.

A small programming language. The syntax and semantics of the small language we used in the example of Sect. 2 is inspired by Bebop [5], an input language of the Moped tool, which is a model-checker for linear-time temporal logic on pushdown systems [6].

Bebop uses a classical imperative language syntax. We assume for the sake of simplicity that control structures have been transformed into test and branch intructions, and that parameter passing and returns for procedures are emulated by using dedicated global variables. This results in the syntax given in Fig. 9.

The features added to Bebop are the communication instructions, and the non deterministic choice operator between them. Emission and reception instructions use a special global variable p which contains the message, and which may be used only in the condition and in the block associated to these instructions. We assume that emissions and receptions are not nested. The operator [] is the non-deterministic choice operator. It may be used only for communication instructions. The reason is that while we allow non-determinism, it should remain observable, so that to any trace of the program corresponds an unique execution.

Its semantics as a Push-Down System (PDS). We assume that the special variable p takes its values in the alphabet Λ. The semantics of this language is defined using the domains defined on Fig. 10. It is given as a labelled PDS $\mathcal{P} = (G, \Gamma, c_0, \Lambda, \hookrightarrow)$ where $G = GEnv$, $\Gamma = K \times LEnv$, $c_0 = (g_0, (k_0, l_0))$ is the initial configuration, and $\hookrightarrow$ is defined by the following inference rules, using the control flow graph associated to the program. We just sketch the standard inference rules and we refer to [6] for more details, as we focus more precisely on the semantics of the emission and reception instructions.

– An atomic instruction generates a rule of the form $\dfrac{k \xrightarrow{atom} k'}{(g,(k,l)) \overset{\tau}{\hookrightarrow} (g',(k',l'))}$ with a condition on (g,l) in the case of a test and branch instruction.

– A procedure call generates a rule $\dfrac{k \xrightarrow{proc()} k'}{(g,(k,l)) \overset{\tau}{\hookrightarrow} (g,(k',l)\cdot(s_{proc},l'_0))}$ where s_{proc} is the start point of the caller. Such a transition means that a new activation record is pushed onto the stack, with an initial local environment l'_0, which reflects the assumption that the variables are uninitialized. A procedure return generates a rule $\dfrac{k \xrightarrow{proc()} k' \quad e_{proc} \xrightarrow{\mathsf{return}} \dots}{(g,(k',l')\cdot(e_{proc},l)) \overset{\tau}{\hookrightarrow} (g,(k',l'))}$ where the activation record is popped and the control goes back to the caller.

– An emission instruction generates a rule

$$\dfrac{k \xrightarrow{\mathsf{emit}(p)\ \mathsf{when}\ expr\ \{k':block\}} k'' \qquad \forall v \neq p : g'(v) = g(v)}{\begin{array}{c}(g,(k,l)) \overset{p}{\hookrightarrow} (g',(k',l)) \text{ if } [\![expr]\!](g',l) = \mathsf{true} \\ (g,(k,l)) \overset{\tau}{\hookrightarrow} (g',(k'',l)) \text{ if } [\![expr]\!](g',l) = \mathsf{false}\end{array}}$$

One first forgets the previous value of p when introducing g', in order to make it uninitialized, as its real scope is the condition and the block associated to the emission. Then, if the current environment (g,l) satisfies the condition, p is emitted and the control passes to the beginning of the block k'. Otherwise, the control passes to k''. Notice that an non-deterministic choice is performed here: the instruction may emit any message p which satisfies the condition.

The semantics of the reception is identical to the emission. Emissions and receptions need to be distinguished only w.r.t. the conformance relation.

All instructions generate internal transitions labelled by τ, except emission and reception instructions. The *observation points* of a program are defined as the control points at the beginning of communication instructions. They are the only control points from which a message may be exchanged. Such observation points may be separated by (sequences of) ordinary control points linked by internal τ-transitions. Notice that we do not use the term "observation point" in the sense given to it in the testing community, when refering to the testing architecture.

Interprocedural specification and its canonical tester. An interprocedural specification S (*c.f.* Fig. 4) is a program defined with the language of Fig. 9, which is deterministic, in the sense that the allowed non-determinism should be observable, so that to a trace corresponds a unique possible execution ending in an observation point. A choice can still exist between two emissions and/or receptions, but we cannot have a choice between two internal instructions (generating τ-transitions).

This deterministic assumption allows to build easily the canonical tester of S, which is an executable, hence deterministic observer of $Traces(S)\cdot\Lambda^S_? \setminus Traces(S)$. The canonical tester $\mathrm{Can}(S)$ is obtained from S using the following program transformation at each observation point:

$$
\begin{array}{|l c |l}
\mathsf{emit}(p)\ \mathsf{when}\ expr_e\ \{ & & \mathsf{receive}(p)\ \mathsf{when}\ true\ \{ \\
 & & \quad \mathsf{if}(\mathsf{not}\ expr_e)\{verdict = fail; abort()\ \} \\
\quad block_e & \Rightarrow & \quad block_e \\
\} & & \} \\
\ [] & & \ [] \\
\mathsf{receive}(p)\ \mathsf{when}\ expr_r\ \{block_r\} & & \mathsf{emit}(p)\ \mathsf{when}\ expr_r\ \{block_r\}
\end{array}
$$

This operation mimics the corresponding operation defined for LTS in Sect. 3. Here, it could be done on the PDS generated by the program, but we prefer to proceed directly by program transformations.

Test purpose. When performing test generations from LTS, the test purpose is an LTS that is taken into account by computing the product $\mathrm{Can}(S) \times TP$ (*c.f.* Sect. 3). Now, $\mathrm{Can}(S)$ is a PDS. It is known that the product of two PDS is not a PDS, hence we cannot specify test purposes using PDS if we do not want to manipulate more expressive computational models. However, as the product of a PDS with an LTS is still a PDS, we can consider test purposes defined by finite LTS. We can compute the synchronous product of $\mathrm{Can}(S)$ with TP to add the Pass verdict to the canonical tester.

However, our goal is to proceed by program transformations. This excludes to work directly on the underlying LTS and PDS models. The solution consists:

- in implementing the LTS TP (which should satisfy Def. 3) by a procedure `TP(p)` that takes as input the last exchanged message and implements the LTS, *c.f.* Figs. 2(b) and 6;
- and in instrumenting $\mathrm{Can}(S)$ by inserting calls to `TP` at observation points:

$$
\begin{array}{|l c |l}
\mathsf{receive}(p)\ \mathsf{when}\ true\ \{ & & \mathsf{receive}(p)\ \mathsf{when}\ true\{ \\
\quad \mathsf{if}(\mathsf{not}\ expr_r) & & \quad \mathsf{if}(\mathsf{not}\ expr_r) \\
\qquad \{\ verdict = fail; abort()\ \} & & \qquad \{\ verdict = fail; abort()\ \} \\
\quad block_r & \Rightarrow & \quad TP(p); block_r \\
\} & & \} \\
\ [] & & \ [] \\
\mathsf{emit}(p)\ \mathsf{when}\ expr_e\ \{block_e\} & & \mathsf{emit}(p)\ \mathsf{when}\ expr_e\ \{TP(p); block_e\}
\end{array}
$$

The call to `TP` is performed after having checked the conformance, because accepted traces are conformant. The procedure `TP` is in charge of emitting the Pass verdict. This transformed canonical tester will be denoted by P. Fig. 7 depicts the obtained program for our running example.

5 Test Selection on the Recursive Canonical Tester

Test selection is based on the same principle as for LTS, *c.f.* Sect. 3. In particular we will exploit the identity $pref_{\leq}(Traces_{\mathsf{Pass}}(P)) = Traces_{coreach(\mathsf{Pass})}(P)$ to recognize (conformant) traces that may be accepted in the future by the test purpose. However, the inability of test cases to inspect their own stack prevents it from using fully the coreachability information. We analyse this partial observation problem in this section.

Location	Coreachable states from $\langle(-, \mathsf{pass}, E), \omega\rangle$
m_0	$\langle(\mathsf{ff}, -, A), \omega.m_0\rangle$ $\langle(\mathsf{tt}, -, A), \omega.m_0\rangle$
m_1	$\langle(-, \mathsf{pass}, E), \omega.m_1\rangle$
f_0	$\langle(\mathsf{ff}, -, A), \omega.f_0\rangle$ $\langle(\mathsf{tt}, -, A), \omega.f_0\rangle$
f_{0r}	$\langle(\mathsf{ff}, -, A, x), \omega.f_{0r}\rangle$ $\langle(\mathsf{tt}, -, A, x), \omega.f_{0r}\rangle$
f_1	$\langle(\mathsf{ff}, -, B), \omega.f_1\rangle$ $\langle(\mathsf{tt}, -, B), \omega.f_1\rangle$
f_2	$\langle(\mathsf{ff}, -, D), \omega.f_2\rangle$ $\langle(\mathsf{tt}, -, D), \omega.f_2\rangle$
f_{2e}	$\langle(\mathsf{ff}, -, D, c), \omega.f_{2e}\rangle$ $\langle(\mathsf{tt}, -, D, c), \omega.f_{2e}\rangle$
f_{2r}	$\perp$
f_3	$\langle(-, \mathsf{pass}, E), \omega.f_3\rangle$
g_0	$\langle(\mathsf{ff}, -, B), \omega.(f_1 g_0 + f_1 g_2 g_0)\rangle$ $\langle(\mathsf{tt}, -, B), \omega.(f_1 g_0 + f_1 g_2 g_0)\rangle$
g_{0e}	$\langle(\mathsf{ff}, -, B, a), \omega.f_1 g_2 g_{0e}\rangle$ $\langle(\mathsf{ff}, -, B, b), \omega.f_1 g_{0e}\rangle$ $\langle(\mathsf{tt}, -, B, a), \omega.f_1 g_2 g_{0e}\rangle$ $\langle(\mathsf{tt}, -, B, b), \omega.f_1 g_{0e}\rangle$
g_{0r}	$\perp$
g_1	$\langle(\mathsf{ff}, -, B), \omega.f_1 g_1\rangle$ $\langle(\mathsf{tt}, -, B), \omega.f_1 g_1\rangle$
g_2	$\langle(\mathsf{ff}, -, B), \omega.f_1 g_2\rangle$ $\langle(\mathsf{tt}, -, B), \omega.f_1 g_2\rangle$
g_3	$\langle(\mathsf{ff}, -, C), \omega.f_1 g_3\rangle$ $\langle(\mathsf{tt}, -, C), \omega.f_1 g_3\rangle$
g_{3r}	$\langle(\mathsf{ff}, -, C, y), \omega.f_1 g_{3r}\rangle$ $\langle(\mathsf{tt}, -, C, y), \omega.f_1 g_{3r}\rangle$
g_4	$\langle(\mathsf{ff}, -, C), \omega.f_1 g_2 g_4\rangle$ $\langle(\mathsf{ff}, -, D), \omega.f_1 g_4\rangle$ $\langle(\mathsf{tt}, -, C), \omega.f_1 g_2 g_4\rangle$ $\langle(\mathsf{tt}, -, D), \omega.f_1 g_4\rangle$

(a) Coreachable states

Location	Reachable states from $\langle(\mathsf{ff}, \mathsf{none}, A), m_0\rangle$
f_{0r}	$\langle(\mathsf{ff}, \mathsf{none}, A, x), m_1 f_{0r}\rangle$ $\langle(\mathsf{ff}, \mathsf{none}, A, z), m_1 f_{0r}\rangle$
f_{2e}	$\langle(\mathsf{ff}, \mathsf{none}, C, c), m_1 f_{2e}\rangle$ $\langle(\mathsf{tt}, \mathsf{none}, D, c), m_1 f_{2e}\rangle$ $\langle(\mathsf{tt}, \mathsf{none}, S, c), m_1 f_{2e}\rangle$
f_{2r}	$\langle(\mathsf{ff}, \mathsf{none}, C, -), m_1 f_{2r}\rangle$ $\langle(\mathsf{tt}, \mathsf{none}, D, -), m_1 f_{2r}\rangle$ $\langle(\mathsf{tt}, \mathsf{none}, S, -), m_1 f_{2r}\rangle$
g_{0e}	$\langle(\mathsf{ff}, \mathsf{none}, B, a), m_1 f_2 g_{0e}\rangle$ $\langle(\mathsf{ff}, \mathsf{none}, B, b), m_1 f_2 g_{0e}\rangle$ $\langle(\mathsf{tt}, \mathsf{none}, B, a), m_1 f_2 g_3^+ g_{0e}\rangle$ $\langle(\mathsf{tt}, \mathsf{none}, B, b), m_1 f_2 g_3^+ g_{0e}\rangle$
g_{0r}	$\langle(\mathsf{ff}, \mathsf{none}, B, -), m_1 f_2 g_{0r}\rangle$ $\langle(\mathsf{tt}, \mathsf{none}, B, -), m_1 f_2 g_3^+ g_{0r}\rangle$
g_{3r}	$\langle(\mathsf{tt}, \mathsf{none}, C, y), m_1 f_2 g_{3r}^+\rangle$ $\langle(\mathsf{tt}, \mathsf{none}, D, y), m_1 f_2 g_{3r}^+\rangle$ $\langle(\mathsf{tt}, \mathsf{none}, S, y), m_1 f_2 g_{3r}^+\rangle$

(b) Reachable states in observation points

Location	Intersection reachable and coreachable states
f_{0r}	$\langle(\mathsf{ff}, \mathsf{none}, A, x,), m_1 f_{0r}\rangle$
f_{2e}	$\langle(\mathsf{ff}, \mathsf{none}, C, c), m_1 f_{2e}\rangle$
f_{2r}	$\perp$
g_{0e}	$\langle(\mathsf{ff}, \mathsf{none}, B, a), m_1 f_2 g_3 g_{0e}\rangle$ $\langle(\mathsf{tt}, \mathsf{none}, B, b), m_1 f_2 g_{0e}\rangle$
g_{0r}	$\perp$
g_{3r}	$\langle(\mathsf{tt}, \mathsf{none}, C, y), m_1 f_2 g_{3r}\rangle$

(c) Intersection between reachable and coreachable states

Fig. 11. Analysis of the program of Fig. 7. *The configurations are composed of the values of global variables* $(t, verdict, pc, p)$ *and the stack* ($-$ *means any value, and* $\omega = K^*$). *As* p *is "active" only at observation points, its value is not precised elsewhere.*

Coreachability Analysis. In the PDS generated by the semantics of our programming language, a configuration is a pair $(g, \sigma) \in C$ of a global environment and a call-stack. The set of configurations corresponding to the Pass verdict is $\mathsf{Pass} = \{(g, \sigma) \mid g(verdict) = \mathsf{Pass})\}$. The wanted coreachable set is $coreach = \{c \in C \mid \exists c' \in \mathsf{Pass} : c \rightarrow^* c'\}$.

We will exploit nice theoretical properties of PDS for computing *coreach*. These properties justify the choice of PDS as the semantic model of our language, and the restriction to finite-state variables. Given a PDS $\mathcal{P} = (G, \Gamma, c_0, \Lambda, \hookrightarrow)$, a set of configurations $X \in \wp(G \times \Gamma^*) = G \rightarrow \wp(\Gamma^*)$ is *regular* if it associates to each global state a regular language. The first result is that the coreachability (resp. reachability) set of a PDS is regular if the final (resp. initial) set of configurations is regular [7]. The second result is that in this case, the coreachability (resp. reachability) set is computable with polynomial complexity [8, 9]. The MOPED tool implements efficient symbolic algorithms to compute these sets, using a model of symbolic PDS where the transition relation $\hookrightarrow$ is represented with BDDs [6].

As the set Pass is regular, we can provide to MOPED the PDS generated by our recursive program and the set Pass, and we obtain the regular set of coreachable configurations. Coming back to our running example, the table of Fig. 11(a) indicates, for every location, the configurations from which we can reach the final configuration $\langle(-, \mathsf{pass}, E), \omega\rangle$. As there are no local variables in the example, the stacks contain only control points.

The problem of partial observation. The selection consists in adding tests in the program P, using coreachability information, for selecting the outputs to emit, and for detecting the inputs which make P leave the set of accepted traces. However, in an usual imperative language like ours, a program can only observe the top of the stack, whereas deciding whether the current configuration is coreachable or not may require the inspection of the full stack.

Let us define the observation function $\alpha : \begin{array}{ccc} C & \to & \mathit{GEnv} \times K \times \mathit{LEnv} \\ (g, \omega\cdot(k,l)) & \mapsto & (g,k,l) \end{array}$ extended to sets, and $\gamma = \alpha^{-1}$ the corresponding inverse function. (α, γ) forms a Galois connection. At some location k of the program, given a set of configurations X, and $X(c) = \{(c \in X \mid c = (g, \omega\cdot(k,l))\}$ its projection on location k, the program can only decide if the current valuation of variables (g,l) is included in $\alpha(X(c))$. This means that *in term of configurations*, one can only test inclusionship in $\gamma\circ\alpha(X) \supseteq X$. In particular one may be in a case with

$$\gamma\circ\alpha(\mathit{coreach}(k)) \cap \gamma\circ\alpha(\overline{\mathit{coreach}}(k)) \neq \emptyset \tag{1}$$

where one cannot decide, using only the observable part of the configuration, whether the configuration is coreachable or not. For instance, in Fig. 11(a), in location g_{0e}, $\alpha(\mathit{coreach}(g_{0e})) = (p \in \{a,b\} \wedge pc = B)$ and $\alpha(\overline{\mathit{coreach}}(g_{0e})) = \mathsf{tt}$.

Selection rules. Because of the partial obervation phenomenon, we have to be conservative in the selection. Let $\mathit{cond}_{co(k)}(g,l)$ be the logical characterization of $\alpha(\mathit{coreach}(k))$. We transform the program P as follows:

$\mathsf{receive}(p)$ when $\mathit{true}\{$
  if (not expr_r)
    $\{\ \mathit{verdict} = \mathit{fail};\ \mathit{abort}()\ \}$
k_r :
 	
  $\mathit{TP}(p); \mathit{block}_r$
$\}$
[]
$\mathsf{emit}(p)$ when expr_e
  $\{k_e : \mathit{TP}(p); \mathit{block}_e\}$
	$\Rightarrow$	$\mathsf{receive}(p)$ when $\mathit{true}\{$
		  if (not expr_r)
		    $\{\ \mathit{verdict} = \mathit{fail};\ \mathit{abort}()\ \}$
		k_r : if not $(\mathit{cond}_{co(k_r)})$
		    $\{\ \mathit{verdict} = \mathit{inconc};\ \mathit{abort}()\ \}$
		  $\mathit{TP}(p); \mathit{block}_r$
		$\}$
		[]
		$\mathsf{emit}(p)$ when expr_e and $\mathit{cond}_{co(k_e)}$
		$\{k_e : \mathit{TP}(p); \mathit{block}_r\}$

For receptions, at location k_r, after having checked the conformance, $\neg\mathit{cond}_{co(k)}$ is a *sufficient* condition to leave the prefixes of accepted traces $(\gamma(\neg\mathit{cond}_{co(k)}) \subseteq$

$\overline{coreach}(k)$). So if it is satisfied we emit Inconc. For emissions, $cond_{co(k)}$ is a *necessary* condition to stay in prefixes of accepted traces ($\gamma(cond_{co(k)}) \supseteq coreach(k)$).

The obtained program is a sound test case. There is a strong similarity between this test selection algorithm and the test selection algorithm for symbolic infinite-state transition systems defined in [4]. Here partial observation may prevent us to perform an optimal selection. In [4], it is the impossibility to compute the exact coreachability set, and the need to resort to an overapproximation.

Improving selection with reachability information. One can improve the selection algorithm using reachability information. Let *reach* denote the set of reachable configurations of the program P. At a point k, we can exploit the knowledge that the current configuration is anyway included in $reach(k)$, and testing the inclusion in $\gamma\circ\alpha(reach(k)\cap coreach(k))$ instead of $\gamma\circ\alpha(coreach(k))$.[3] The problematic case identified by Eqn (1) becomes

$$\gamma \circ \alpha(reach(k) \cap coreach(k)) \cap \gamma \circ \alpha(reach(k) \cap \overline{coreach}(k)) \neq \emptyset \tag{2}$$

It is clear that Eqn. (2) implies Eqn. (1) but that the converse if false.

Coming back to our example, Fig. 11(b) gives the reachability set of P projected on observation points, and Fig. 11(c) the intersection $reach(k)\cap coreach(k)$ for these points. If $cond_{co(k)}(g,l)$ denotes now a formula characterizing $\alpha(reach(k)\cap coreach(k))$, we now have $cond_{co(g_0 a)} = (pc = B)\wedge(t\wedge p = b\vee\neg t\wedge p = a)$ instead of just $(pc = B)$. One can check that Eqn. (2) is not true for $k = g_{0e}$, thus the selection is optimal at this point. Fig. 8 depicts the test case obtained by this improved selection algorithm. It should be noted that the presence of the variable `t` helps to perform an accurate selection at location g_0, because it allows to distinguish whether `G()` has been called from f_1 or from g_2. If we remove this variable, which does not change the semantics of S w.r.t. the conformance relation, one could not select optimally the output a or b to send to the IUT.

6 Concluding Remarks

The selection algorithm of Sect. 5 is currently under implementation by the extension of the Moped tool. Moped acts as a model-checker returning a Boolean answer, possibly with a counter-example. For our application, we need to get the sets of configurations computed by Moped, to intersect reachability and coreachability sets, to project this intersection on the visible part, and to convert the result in terms of a programming language expression.

It is interesting to note the similarity of the two combinations: partial observation and exact analysis w.r.t. full observation and approximated analysis. In case of partial observation, the observation function α we introduced acts exactly as an abstraction (approximation) function. This means that one could apply our method to general recursive programs, on which the analysis would be in general approximated. The non-optimality of the selection would then be a consequence of the combination of partial observation and inexact analysis. One gets the diagram of Fig. 1.

[3] As reachability and coreachability sets are regular, so is their intersection.

Alternative methods. Our selection method described in Sect. 5 is based on (i) an exact analysis computing full configurations (instead of just visible parts of configurations), and (ii) on pure program transformations. These two choices could be revised. Concerning (i), one could use a less precise, classical interprocedural analysis method, which could still be exact for the observable part of the stack (for instance using the Bebop tool [5]). However it would lead to a less precise selection scheme. In particular, intersecting the coreachabe set with reachable set would filter out less values. Concerning (ii), one could instrument the program so as to get more knowledge about the invisible part of the configuration. For instance, one could add a data-structure maintaining a stack of procedure return points, and using it when testing if one is still in a coreachable configuration. Although the resulting test case could not be any more transformed into a PDS, the analysis would still be performed on the same intermediate program P as in Sect. 5. Test case execution would be however slower, as testing for coreachability would involve more complex datatypes.

References

1. ISO/IEC 9646: Conformance Testing Methodology and Framework (1992)
2. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software—Concepts and Tools **17**(3) (1996)
3. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. Int. Journal on Software Tools for Technology Transfer **6** (2004)
4. Jeannet, B., Jéron, T., Rusu, V., Zinovieva, E.: Symbolic test selection based on approximate analysis. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS'05. Volume 3440 of LNCS. (2005)
5. Ball, T., Rajamani, S.: Bebop: A symbolic model checker for boolean programs. In: Workshop SPIN'00. Volume 1885 of LNCS. (2000)
6. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Computer Aided Verification, CAV'01. Volume 2102 of LNCS. (2001)
7. Caucal, D.: On the regular structure of prefix rewriting. Theoretical Computer Science **106** (1992)
8. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. Electronic Notes on Theoretical Computer Science **9** (1997)
9. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model checking. In: Int. Conf. on Concurrency Theory, CONCUR'97. Volume 1243 of LNCS. (1997)