Concolic Execution and Test Case Generation in Prolog*

Germán Vidal

MiST, DSIC, Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain
gvidal@dsic.upv.es

Abstract. Symbolic execution extends concrete execution by allowing symbolic input data and then exploring all feasible execution paths. It has been defined and used in the context of many different programming languages and paradigms. A symbolic execution engine is at the heart of many program analysis and transformation techniques, like partial evaluation, test case generation or model checking, to name a few. Despite its relevance, traditional symbolic execution also suffers from several drawbacks. For instance, the search space is usually huge (often infinite) even for the simplest programs. Also, symbolic execution generally computes an overapproximation of the concrete execution space, so that false positives may occur. In this paper, we propose the use of a variant of symbolic execution, called concolic execution, for test case generation in Prolog. Our technique aims at full statement coverage. We argue that this technique computes an underapproximation of the concrete execution space (thus avoiding false positives) and scales up better to medium and large Prolog applications.

1 Introduction

There is a renewed interest in *symbolic execution* [9, 3], a well-known technique for program verification, testing, debugging, etc. In contrast to concrete execution, symbolic execution considers that the values of some input data are unknown, i.e., some input parameters $x, y, \ldots$ take *symbolic values* $X, Y, \ldots$ Because of this, symbolic execution is often non-deterministic: at some control statements, we need to follow more than one execution path because the available information does not suffice to determine the validity of a control expression, e.g., symbolic execution may follow both branches of the conditional "`if` $(x{>}0)$ `then` $exp1$ `else` $exp2$" when the symbolic value X of variable x is not constrained enough to imply neither $x{>}0$ nor $\neg(x{>}0)$. Symbolic states include a *path condition* that stores the current constraints on symbolic values, i.e., the

* This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2013-44742-C4-1-R and by the *Generalitat Valenciana* under grant PROMETEO/2011/052.

conditions that must hold to reach a particular execution state. E.g., after symbolically executing the above conditional, the derived states for $exp1$ and $exp2$ would add the conditions $X>0$ and $X\leq 0$, respectively, to their path conditions.

Traditionally, formal techniques based on symbolic execution have enforced *soundness*: if a symbolic state is reached and its path condition is satisfiable, there must be a concrete execution path that reaches the corresponding concrete state. In contrast, we say that symbolic execution is *complete* when every reachable state in a concrete execution is "covered" by some symbolic state. For the general case of infinite state systems, completeness usually requires some kind of *abstraction* (as in infinite state model checking).

In the context of logic programming, we can find many techniques that use some form of *complete* symbolic execution, like partial evaluation [12, 13, 4]. However, these overapproximations of the concrete semantics may have a number of drawbacks in the context of testing and debugging. On the one hand, one should define complex subsumption and abstraction operators since the symbolic search space is usually infinite. These abstraction operators, may introduce *false positives*, which is often not acceptable when debugging large applications. On the other hand, because of the complexity of these operators, the associated methods usually do not scale to medium and large applications.

In imperative programming, an alternative approach, called *concolic execution* [6, 16], has become popular in the last years. Basically, concolic execution proceeds as follows: first, a concrete execution using random input data is performed. In parallel to the concrete execution, a symbolic execution is also performed, but restricted to the same conditional choices of the concrete execution. Then, by negating one of the constraints in the symbolic execution, new input data are obtained, and the process starts again. Here, only concrete executions are considered and, thus, no false positives are produced. This approach has given rise to a number of powerful and scalable tools in the context of imperative and concurrent programming, like Java Pathfinder [14] and SAGE [7].

In this paper, we present a novel scheme for testing pure Prolog (without negation) based on a notion of concolic execution. To the best of our knowledge, this is the first approach to concolic execution in the context of a declarative programming paradigm.

2 Preliminaries

We assume some familiarity with the standard definitions and notations for logic programs [11]. Nevertheless, in order to make the paper as self-contained as possible, we present in this section the main concepts which are needed to understand our development.

In this work, we consider a first-order language with a fixed vocabulary of predicate symbols, function symbols, and variables denoted by Π, Σ and $\mathcal{V}$, respectively. In the following, we let $\overline{o_n}$ denote the sequence of syntactic objects $o_1, \ldots, o_n$, and we let $\overline{o_{n,m}}$ denote the sequence $o_n, o_{n+1}, \ldots, o_m$. Also, we often use $\overline{o}$ when the number of elements in the sequence is irrelevant. We let $\mathcal{T}(\Sigma, \mathcal{V})$

denote the set of *terms* constructed using symbols from Σ and variables from $\mathcal{V}$. An *atom* has the form $p(\overline{t_n})$ with $p/n \in \Pi$ and $t_i \in \mathcal{T}(\Sigma, \mathcal{V})$ for $i = 1, \ldots, n$. A *goal* is a finite sequence of atoms $A_1, \ldots, A_n$, where the *empty goal* is denoted by *true*. A *clause* has the form $H \rightarrow \mathcal{B}$ where H is an atom and $\mathcal{B}$ is a goal (note that we only consider *definite* programs). A logic *program* is a finite sequence of clauses. $\mathcal{V}ar(s)$ denotes the set of variables in the syntactic object s (i.e., s can be a term, an atom, a query, or a clause). A syntactic object s is *ground* if $\mathcal{V}ar(s) = \emptyset$. In this work, we only consider *finite* ground terms.

Substitutions and their operations are defined as usual. In particular, the set $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$ is called the *domain* of a substitution σ. We let *id* denote the empty substitution. The application of a substitution θ to a syntactic object s is usually denoted by juxtaposition, i.e., we write $s\theta$ rather than $\theta(s)$. A syntactic object s_1 is *more general* than a syntactic object s_2, denoted $s_1 \leqslant s_2$, if there exists a substitution θ such that $s_2 = s_1\theta$. A *variable renaming* is a substitution that is a bijection on $\mathcal{V}$. Two syntactic objects t_1 and t_2 are *variants* (or equal up to variable renaming), denoted $t_1 \approx t_2$, if $t_1 = t_2\rho$ for some variable renaming ρ. A substitution θ is a unifier of two syntactic objects t_1 and t_2 iff $t_1\theta = t_2\theta$; furthermore, θ is the *most general unifier* of t_1 and t_2, denoted by $mgu(t_1, t_2)$ if, for every other unifier σ of t_1 and t_2, we have that $\theta \leqslant \sigma$.

The notion of *computation rule* $\mathcal{R}$ is used to select an atom within a goal for its evaluation. Given a program P, a goal $\mathcal{G} \equiv A_1, \ldots, A_n$, and a computation rule $\mathcal{R}$, we say that $\mathcal{G} \leadsto_{P,\mathcal{R},\sigma} \mathcal{G}''$ is an *SLD resolution step* for $\mathcal{G}$ with P and $\mathcal{R}$ if

- $\mathcal{R}(\mathcal{G}) = A_i$, $1 \leqslant i \leqslant n$, is the selected atom,
- $H \rightarrow \mathcal{B}$ is a renamed apart clause of P (in symbols $H \rightarrow \mathcal{B} \ll P$),
- $\sigma = mgu(A, H)$, and
- $\mathcal{G}' \equiv (A_1, \ldots, A_{i-1}, \mathcal{B}, A_{i+1}, \ldots, A_n)\sigma$.

We often omit P, $\mathcal{R}$ and/or σ in the notation of an SLD resolution step when they are clear from the context. An *SLD derivation* is a (finite or infinite) sequence of SLD resolution steps. We often use $\mathcal{G}_0 \leadsto^*_\theta \mathcal{G}_n$ as a shorthand for $\mathcal{G}_0 \leadsto_{\theta_1} \mathcal{G}_1 \leadsto_{\theta_2} \ldots \leadsto_{\theta_n} \mathcal{G}_n$ with $\theta = \theta_n \circ \cdots \circ \theta_1$ (where $\theta = \{\}$ if $n = 0$). An SLD derivation $\mathcal{G} \leadsto^*_\theta \mathcal{G}'$ is *successful* when $\mathcal{G}' = true$; in this case, we say that θ is the *computed answer substitution*. SLD derivations are represented by a (possibly infinite) finitely branching tree.

3 A Deterministic Semantics

In this section, we introduce a deterministic small-step semantics for pure Prolog (without negation). Basically, as we will see in the next section, we need to keep some information through the complete Prolog computation, and the usual semantics based on non-determinism and backtracking is not adequate for this purpose. Therefore, we propose the use of a stack to store alternative execution paths that are tried when a failure is reached. The resulting small-step semantics

$$\text{(unfolding)}\ \frac{\text{let } \overline{H_n \overset{\ell_n}{\leftarrow} \mathcal{B}_n} \ll P \text{ be all the clauses such that } \mathsf{mgu}(A_1, H_i) = \sigma_i \neq \mathit{fail}, i = 1, \ldots, n}{\langle \overline{A_m}; \sigma; \mathcal{S} \rangle \overset{\mathsf{u}(\overline{\ell_n})}{\rightarrow} \langle (\mathcal{B}_1, \overline{A_{2,m}})\sigma_1; \sigma\sigma_1; [(\ell_2; \sigma\sigma_2; (\mathcal{B}_2, \overline{A_{2,m}})\sigma_2), \ldots, (\ell_n; \sigma\sigma_n; (\mathcal{B}_n, \overline{A_{2,m}})\sigma_n)] {+}{+} \mathcal{S} \rangle}$$

$$\text{(backtracking)}\ \frac{\text{there is no clause } H \overset{\ell'}{\leftarrow} \mathcal{B} \ll P \text{ such that } \mathsf{mgu}(A_1, H) \neq \mathit{fail}}{\langle \overline{A_m}; \sigma'; [(\ell; \mathcal{G}; \sigma) | \mathcal{S}] \rangle \overset{\mathsf{b}(\ell)}{\rightarrow} \langle \mathcal{G}; \sigma; \mathcal{S} \rangle} \quad (m \geqslant 1)$$

$$\text{(failure)}\ \frac{\text{there is no clause } H \overset{\ell}{\leftarrow} \mathcal{B} \ll P \text{ such that } \mathsf{mgu}(A_1, H) \neq \mathit{fail}}{\langle \overline{A_m}; \sigma; [\,] \rangle \overset{\mathsf{f}}{\rightarrow} \langle \mathsf{fail}; \sigma; [\,] \rangle} \quad (m \geqslant 1)$$

Fig. 1. Deterministic small-step semantics

is clearly equivalent to the original one when a depth-first search is considered. Actually, our deterministic semantics is essentially equivalent to (a subset of) the *linear* semantics presented in [17].

In the following, we assume that the program clauses are labeled. In particular, given a program P, we use the notation $H \overset{\ell}{\leftarrow} \mathcal{B} \ll P$ to refer to a (renamed apart) labeled clause $H \leftarrow \mathcal{B}$ in P. Labels must be unique. Moreover, we only consider Prolog's left-to-right computation rule, and assume that only the computation of the first answer for the initial goal is relevant (as it is common in practical Prolog applications).

Our semantics deals with *states*, which are defined as follows:

Definition 1 (state). *A* state *is a tuple $\langle \mathcal{G}; \sigma; \mathcal{S} \rangle$, where $\mathcal{G}$ is a goal, σ is a substitution—the (partial) answer computed so far—and $\mathcal{S}$, the* stack*, is a (possibly empty) list of tuples $(\ell; \mathcal{G}'; \sigma')$ with ℓ a clause label, $\mathcal{G}'$ a goal and σ' a substitution.*

The small-step deterministic semantics is defined as the smallest relation that obeys the labeled transition rules shown in Figure 1, where $[H|R]$ denotes a list with head H and tail R, and "++" denotes list concatenation.

Given a goal $\mathcal{G}_0$, the initial state has the form $\langle \mathcal{G}_0; id; [\,] \rangle$. The transition relation is labeled with $\mathsf{u}(\overline{\ell_n})$, denoting an unfolding step with the clauses labeled with $\overline{\ell_n}$, $\mathsf{b}(\ell)$, denoting a backtracking step that tries a clause labeled with ℓ, or f, denoting a failing derivation.

Let us briefly explain the rules of the small-step semantics:

- The unfolding rule proceeds as in standard SLD resolution, but considers all matching clauses, so that all SLD resolution steps are performed in one go. The first unfolding step is used to replace the goal component of the state,

while the remaining ones (if any) are added on top of the stack (thus we mimic the usual depth-first search of Prolog). Here, the labels of the clauses and the partial computed answers are also stored in the stack in order to recover this information when a backtracking step is performed.
- The backtracking rule applies when no further unfolding is possible and the goal component is not *true* (the empty goal). In this case, we discard the current goal and consider the first goal in the stack, extracting the clause label and the partial answer that are needed for the transition step.
- Finally, the failure rule is used to terminate a computation that reaches a goal in which the selected atom does not match any rule and, moreover, there are no alternatives in the stack.

A *successful* computation has the form $\langle \mathcal{G}_0; id; [\,] \rangle \stackrel{s_1}{\rightarrow} \langle \mathcal{G}_1; \sigma_1; \mathcal{S}_1 \rangle \stackrel{s_2}{\rightarrow} \ldots \stackrel{s_n}{\rightarrow} \langle true; \sigma_n; \mathcal{S}_n \rangle$, where σ_n (restricted to the variables of $\mathcal{G}_0$) is the computed answer substitution. A *failing* computation has the form $\langle \mathcal{G}_0; id; [\,] \rangle \stackrel{s_1}{\rightarrow} \langle \mathcal{G}_1; \sigma_1; \mathcal{S}_1 \rangle \stackrel{s_2}{\rightarrow} \ldots \stackrel{s_n}{\rightarrow} \langle \mathcal{G}_n; \sigma_n; \mathcal{S}_n \rangle \stackrel{\mathsf{f}}{\rightarrow}_{id} \langle \mathsf{fail}; \sigma_n; [\,] \rangle$; we keep σ_n in the last state since it might be useful for analyzing finite failure derivations.

Now, we introduce the following notion of execution *trace*, that will be used in the next section to steer the symbolic execution.

Definition 2 (trace). *Let* $\langle \mathcal{G}_0; id; \mathcal{S}_0 \rangle \stackrel{s_1}{\rightarrow} \langle \mathcal{G}_1; \sigma_1; \mathcal{S}_1 \rangle \stackrel{s_2}{\rightarrow} \ldots \stackrel{s_n}{\rightarrow} \langle \mathcal{G}_n; \sigma_n; \mathcal{S}_n \rangle$ *be a computation. The associated* trace *is the list* $[s_1, s_2, \ldots, s_n]$*, where each* s_i *is either of the form* $\mathsf{u}(\overline{\ell_m})$*,* $\mathsf{b}(\ell)$ *or* f*.*

Example 1. Consider the rev_acc_type program to reverse a list using an accumulator and also checking the type of the input parameter (from the DPPD library [10]), extended with predicates `main`, `length`, and `foo`:

```
(1) main(L,N,R) :-
      length(L,N),
      rev(L,[],R),
      foo(a).
(2) main(_L,_N,error).

(3) rev([],A,A).
(4) rev([H|T],Acc,Res) :-
      is_list(Acc),
      rev(T,[H|Acc],Res).

(5) is_list([]).
(6) is_list([_H|T]) :-
       is_list(T).

(7) length([],0).
(8) length([_H|R],s(N)) :-
      length(R,N).

(9) foo(b).
```

Here, we use natural numbers as clause labels. Predicate `main` considers two cases: if the input list `L` has length `N` (the length is represented using natural numbers built from `0` and `s(_)` to avoid the use of built-ins), the reverse of `L` is computed; otherwise, we assume that an error occurs. The computation for the initial goal $\mathtt{main}([\mathtt{a},\mathtt{b}], \mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{R})$ is shown in Figure 2, where only the relevant computed substitutions are shown. The trace associated to the computation is

$$[\mathsf{u}(1,2), \mathsf{u}(8), \mathsf{u}(8), \mathsf{u}(7), \mathsf{u}(4), \mathsf{u}(5), \mathsf{u}(4), \mathsf{u}(6), \mathsf{u}(5), \mathsf{u}(3), \mathsf{b}(2)]$$

$\langle \texttt{main}([\texttt{a},\texttt{b}], \texttt{s}(\texttt{s}(0)), \texttt{R});\ id;\ [\,]\rangle$

$\overset{\texttt{u}(1,2)}{\rightarrow} \langle \texttt{length}([\texttt{a},\texttt{b}], \texttt{s}(\texttt{s}(0))), \texttt{rev}([\texttt{a},\texttt{b}], [\,], \texttt{R}), \texttt{foo}(\texttt{a});\ id;\ [(2; \{\texttt{R}/\texttt{error}\}; \texttt{true})]\rangle$

$\overset{\texttt{u}(8)}{\rightarrow} \langle \texttt{length}([\texttt{b}], \texttt{s}(0)), \texttt{rev}([\texttt{a},\texttt{b}], [\,], \texttt{R}), \texttt{foo}(\texttt{a});\ id;\ [(2; \{\texttt{R}/\texttt{error}\}; \texttt{true})]\rangle$

$\overset{\texttt{u}(8)}{\rightarrow} \langle \texttt{length}([\,], 0), \texttt{rev}([\texttt{a},\texttt{b}], [\,], \texttt{R}), \texttt{foo}(\texttt{a});\ id;\ [(2; \{\texttt{R}/\texttt{error}\}; \texttt{true})]\rangle$

$\overset{\texttt{u}(7)}{\rightarrow} \langle \texttt{rev}([\texttt{a},\texttt{b}], [\,], \texttt{R}), \texttt{foo}(\texttt{a});\ id;\ [2; (\{\texttt{R}/\texttt{error}\}; \texttt{true})]\rangle$

$\overset{\texttt{u}(4)}{\rightarrow} \langle \texttt{is\_list}([\,]), \texttt{rev}([\texttt{b}], [\texttt{a}], \texttt{R}), \texttt{foo}(\texttt{a});\ id;\ [(2; \{\texttt{R}/\texttt{error}\}; \texttt{true})]\rangle$

$\overset{\texttt{u}(5)}{\rightarrow} \langle \texttt{rev}([\texttt{b}], [\texttt{a}], \texttt{R}), \texttt{foo}(\texttt{a});\ id;\ [(2; \{\texttt{R}/\texttt{error}\}; \texttt{true})]\rangle$

$\overset{\texttt{u}(4)}{\rightarrow} \langle \texttt{is\_list}([\texttt{a}]), \texttt{rev}([\,], [\texttt{b},\texttt{a}], \texttt{R}), \texttt{foo}(\texttt{a});\ id;\ [(2; \{\texttt{R}/\texttt{error}\}; \texttt{true})]\rangle$

$\overset{\texttt{u}(6)}{\rightarrow} \langle \texttt{is\_list}([\,]), \texttt{rev}([\,], [\texttt{b},\texttt{a}], \texttt{R}), \texttt{foo}(\texttt{a});\ id;\ [(2; \{\texttt{R}/\texttt{error}\}; \texttt{true})]\rangle$

$\overset{\texttt{u}(5)}{\rightarrow} \langle \texttt{rev}([\,], [\texttt{b},\texttt{a}], \texttt{R}), \texttt{foo}(\texttt{a});\ id;\ [(2; \{\texttt{R}/\texttt{error}\}; \texttt{true})]\rangle$

$\overset{\texttt{u}(3)}{\rightarrow} \langle \texttt{foo}(\texttt{a});\ \{\texttt{R}/[\texttt{b},\texttt{a}]\};\ [(2; \{\texttt{R}/\texttt{error}\}; \texttt{true})]\rangle$

$\overset{\texttt{b}(2)}{\rightarrow} \langle \texttt{true};\ \{\texttt{R}/\texttt{error}\};\ [\,]\rangle$

Fig. 2. Successful computation for $\texttt{main}([\texttt{a},\texttt{b}], \texttt{s}(\texttt{s}(0)), \texttt{R})$.

4 Concolic Execution

In this section, we introduce the semantics of concolic execution. Essentially, it deals with symbolic input data (free variables in our context), as in standard symbolic execution, but is driven by a concrete execution. Often, a single algorithm mixing both concrete and symbolic execution is introduced. In contrast, for clarity, we prefer to keep both calculi independent: the concrete semantics produces a trace, which is then used to steer the symbolic execution.[1]

The symbolic states for concolic execution are defined as follows:

Definition 3 (symbolic state). *A* symbolic state *is a tuple* $\langle \tau; \mathcal{L}; \mathcal{G}; \sigma; \mathcal{S}; T \rangle$, *where*

- τ *is a computation trace,*
- $\mathcal{L}$ *is a list of clause labels (namely, a stack that keeps track of the current clause environment),*[2]
- $\mathcal{G}$ *is a goal,*
- σ *is the partial answer computed so far (a substitution),*
- $\mathcal{S}$ *is a (possibly empty) list of tuples,* $(\ell; \mathcal{L}'; \sigma; \mathcal{G}')$, *where* $\mathcal{L}'$ *is also a list of clause labels, and*
- T *is a set of clause labels (the labels of those clauses not yet completely evaluated).*

[1] Nevertheless, an implementation of this technique may as well combine both calculi into a single algorithm to improve efficiency.

[2] The usefulness of keeping the clause stack will become clear in the next section. Basically, it is needed to know which other clauses can be completely evaluated when a given clause—the one that is on top of the stack—is completely evaluated.

$$\text{(unfolding)}\ \frac{\begin{array}{l}\text{if } \overline{H_n \overset{\ell_n}{\leftarrow} \mathcal{B}_n} \ll P \text{ with } \mathsf{mgu}(A_1, H_i) = \sigma_i, i = 1, \ldots, n \text{ and} \\ \overline{H'_k \overset{\ell'_k}{\leftarrow} \mathcal{B}'_k} \ll P \text{ with } \mathsf{mgu}(A_1, H'_j) = \sigma'_j,\ j = 1, \ldots, k \\ \text{and } \{\overline{\ell_n}\} \cap \{\overline{\ell'_k}\} = \{\,\}\end{array}}{\langle [\mathsf{u}(\overline{\ell_n})|\tau]; \mathcal{L}; \overline{A_m}; \sigma; \mathcal{S}; T\rangle \overset{\{\overline{\mathsf{c}(\ell'_k, \sigma'_k)}\}}{\leadsto} \begin{array}{l}\langle \tau; [\ell_1|\mathcal{L}]; \\ (\mathcal{B}_1, \mathsf{e}(\ell_1), \overline{A_{2,m}})\sigma_1;\ \sigma\sigma_1; \\ [(\ell_2; [\ell_2|\mathcal{L}]; \sigma\sigma_2; (\mathcal{B}_2, \mathsf{e}(\ell_2), \overline{A_{2,m}})\sigma_2), \\ \ldots, \\ (\ell_n; [\ell_n|\mathcal{L}]; \sigma\sigma_n; (\mathcal{B}_n, \mathsf{e}(\ell_n), \overline{A_{2,m}})\sigma_n)] +\!\!+ \mathcal{S}; T\rangle\end{array}}$$

$$\text{(exit)}\ \frac{A_1 = \mathsf{e}(\ell)}{\langle \tau; [\ell|\mathcal{L}]; \overline{A_m}; \sigma; \mathcal{S}; T\rangle \overset{\{\}}{\leadsto} \langle \tau; \mathcal{L}; \overline{A_{2,m}}; \sigma; \mathcal{S}; T \setminus \{\ell\}\rangle}$$

$$\text{(backtracking)}\ \frac{A_1 \neq \mathsf{e}(\_),\ \overline{H_n \overset{\ell_n}{\leftarrow} \mathcal{B}_n} \ll P \text{ s.t. } \mathsf{mgu}(A_1, H_i) = \sigma \neq \mathit{fail},\ i = 1, \ldots, n}{\langle [\mathsf{b}(\ell)|\tau]; \mathcal{L}; \overline{A_m}; \sigma; [(\ell; \mathcal{L}'; \sigma'; \mathcal{G})|\mathcal{S}]; T\rangle \overset{\{\overline{\mathsf{c}(\ell_n, \sigma_n)}\}}{\leadsto} \langle \tau; \mathcal{L}'; \mathcal{G}; \sigma'; \mathcal{S}; T\rangle} \quad (m \geqslant 1)$$

$$\text{(failure)}\ \frac{\text{if } \overline{H_n \overset{\ell_n}{\leftarrow} \mathcal{B}_n} \ll P \text{ such that } \mathsf{mgu}(A_1, H_i) = \sigma \neq \mathit{fail},\ i = 1, \ldots, n}{\langle [\mathsf{f}|\tau]; \mathcal{L}; \overline{A_m}; \sigma; [\,]; T\rangle \overset{\{\overline{\mathsf{c}(\ell_n, \sigma_n)}\}}{\leadsto} \langle \tau; \mathcal{L}; \mathsf{fail}; \sigma; [\,]; T\rangle} \quad (m \geqslant 1)$$

Fig. 3. Concolic execution semantics

The concolic execution semantics is defined as the smallest relation that obeys the labeled transition rules shown in Figure 3. Given a trace τ, the initial symbolic state has the form $\langle \tau; [\,]; \mathcal{G}_0; id; [\,]; T\rangle$, where $\mathcal{G}_0$ is a goal with the same predicates as in the concrete execution, but with fresh variables as arguments, and T is a set with the labels of all program clauses. The transition relation is labeled with a (possibly empty) list of terms of the form $\mathsf{c}(\ell, \theta)$, which denote possible alternatives for unfolding that concrete execution did not consider. Missing alternatives will be used to generate new input data that explore different execution paths.

Let us briefly explain the rules of concolic execution semantics:

– The unfolding rule follows the trace of the concrete execution and applies the same unfolding step. Here, a call of the form $\mathsf{e}(\ell)$ is added to the end of the clause bodies to mark when the clauses are completely evaluated. This is required in our context since we only consider that a clause is *covered* when

all body atoms are successfully executed.[3] This will be a useful information for test case generation, as we will see in the next section. Moreover, in this rule, we label the step with the information regarding the remaining clauses whose head unifies with the selected atom (and did not match with it in the concrete execution). Finally, we add ℓ_1 to the stack of clause labels (the current environment).
- The exit rule applies when the selected atom has the form $\mathsf{e}(\ell)$. In this case, we remove ℓ from the top of the environment stack, and also delete ℓ from the set of clause labels T (i.e., clause ℓ has been completely evaluated).
- The backtracking and failure rules proceed analogously to the unfolding rule by labeling the step with the information regarding the additional clauses whose head unify with the selected atom (if any).

When the set labeling rules unfolding, backtracking and failure is not empty, we have identified situations in which the symbolic state can follow an execution path that is not possible with the concrete goal. Therefore, they allow us to construct new input data for the initial goal so that a different execution path is followed.

Let us now show a simple computation with the concolic execution semantics. We postpone to the next section the algorithm for test case generation.

Example 2. Consider the following simple program:

```
(1)  p(X) :- q(X),r(X).
(2)  q(X) :- s(X).
(3)  s(a).
(4)  s(b).
(5)  r(b).
```

where we again consider natural numbers as clause labels. The concrete execution for the initial goal $\mathtt{p(a)}$ is as follows:[4]

$$\langle \mathtt{p(a)}; id; [] \rangle \overset{\mathsf{u(1)}}{\rightarrow} \langle \mathtt{q(a)}, \mathtt{r(a)}; id; [] \rangle \overset{\mathsf{u(2)}}{\rightarrow} \langle \mathtt{s(a)}, \mathtt{r(a)}; id; [] \rangle \overset{\mathsf{u(3)}}{\rightarrow} \langle \mathtt{r(a)}; id; [] \rangle \overset{\mathsf{f}}{\rightarrow} \langle \mathsf{fail}; id; [] \rangle$$

Therefore, its associated trace is $\tau = [\mathsf{u(1)}, \mathsf{u(2)}, \mathsf{u(3)}, \mathsf{f}]$. Now, for concolic execution, we consider the trace τ and the initial goal $\mathtt{p(X)}$. The concolic execution is shown in Figure 4. As can be seen, the execution of clauses 1, 4 and 5 has not been completed. Moreover, we can observe that there was only one missing alternative when unfolding $\mathtt{s(X)}$. In the next section, we show how this information can be used for test case generation.

[3] Observe that other, more relaxed, notions of clause covering are possible; e.g., consider that a clause is covered as soon as the clause is used in an unfolding step. Also, see [2] for a more declarative notion of test coverage.

[4] In the examples, we restrict the (partial) computed answers to the variables of the initial goal.

$$
\begin{array}{rrlll}
 & \langle [\mathtt{u}(1),\mathtt{u}(2),\mathtt{u}(3),\mathtt{f}];\,[\,];\; & \mathtt{p}(\mathtt{X}); & id; & [\,];\,[1,2,3,4,5]\rangle \\
\overset{\{\}}{\leadsto} & \langle [\mathtt{u}(2),\mathtt{u}(3),\mathtt{f}];\,[1];\; & \mathtt{q}(\mathtt{X}),\mathtt{r}(\mathtt{X}),\mathtt{e}(1); & id; & [\,];\,[1,2,3,4,5]\rangle \\
\overset{\{\}}{\leadsto} & \langle [\mathtt{u}(3),\mathtt{f}];\,[2,1];\; & \mathtt{s}(\mathtt{X}),\mathtt{e}(2),\mathtt{r}(\mathtt{X}),\mathtt{e}(1); & id; & [\,];\,[1,2,3,4,5]\rangle \\
\overset{\{\mathtt{c}(4,\{\mathtt{X}/\mathtt{b}\})\}}{\leadsto} & \langle [\mathtt{f}];\,[3,2,1];\; & \mathtt{e}(3),\mathtt{e}(2),\mathtt{r}(\mathtt{a}),\mathtt{e}(1); & \{\mathtt{X}/\mathtt{a}\}; & [\,];\,[1,2,3,4,5]\rangle \\
\overset{\{\}}{\leadsto} & \langle [\mathtt{f}];\,[2,1];\; & \mathtt{e}(2),\mathtt{r}(\mathtt{a}),\mathtt{e}(1); & \{\mathtt{X}/\mathtt{a}\}; & [\,];\,[1,2,4,5]\rangle \\
\overset{\{\}}{\leadsto} & \langle [\mathtt{f}];\,[1];\; & \mathtt{r}(\mathtt{a}),\mathtt{e}(1); & \{\mathtt{X}/\mathtt{a}\}; & [\,];\,[1,4,5]\rangle \\
\overset{\{\}}{\leadsto} & \langle [\,];\,[1];\; & \mathtt{fail}; & \{\mathtt{X}/\mathtt{a}\}; & [\,];\,[1,4,5]\rangle
\end{array}
$$

Fig. 4. Concolic execution for $[\mathtt{u}(1),\mathtt{u}(2),\mathtt{u}(3),\mathtt{f}]$ and $\mathtt{p}(\mathtt{X})$

5 Test Case Generation

In this section, we present an algorithm for test case generation using concolic execution. In contrast to previous approaches for Prolog testing, our technique considers an underapproximation, i.e., only actual executions are considered (since there is no abstraction involved). Therefore, no false positives may occur. If a test case shows an error, this is an actual error in the considered program.

5.1 The Algorithm

In this section, we assume that the program contains a single predicate that starts the execution, which we denote with *main*. This is not unusual for real applications. Moreover, we consider a particular *mode* for *main*,[5] where $in(main/n) = \{i_1,\ldots,i_m\}$ denotes the set of input parameters of *main*.

The algorithm for test case generation proceeds as follows:

1. First, a random goal of the form $main(\overline{t_n})$ is produced, where at least the input arguments (according to $in(main/n)$) must be ground.
2. Now, we use the concrete semantics to execute the goal $main(\overline{t_n})$, thus obtaining an associated trace τ. We assume that this execution terminates, which is reasonable since the input arguments are ground. In practice, one can use a timeout and report a warning when the execution takes more time.
3. Then, we use concolic execution to run an initial symbolic state of the form

   $$\langle \tau; [\,]; main(\overline{X_n}); id; [\,]; T\rangle$$

   where T is a set with the labels of all program clauses. Since the concrete execution was finite, so is the concolic execution (since it performs exactly the same steps). Let us consider that it has the following form:

   $$\langle \tau_0;\mathcal{L}_0;\mathcal{G}_0;\sigma_0;\mathcal{S}_0;T_0\rangle \overset{c_1}{\leadsto} \ldots \overset{c_m}{\leadsto} \langle \tau_m;\mathcal{L}_m;\mathcal{G}_m;\sigma_m;\mathcal{S}_m;T_m\rangle$$

[5] Extending our approach to multiple modes would not be difficult, but would introduce another source of nondeterminism when grounding an input goal.

where $\tau_0 = \tau$, $\mathcal{L}_0 = [\,]$, $\mathcal{G}_0 = main(\overline{X_n})$, $\sigma_0 = id$, $\mathcal{S}_0 = [\,]$, $T_0 = T$, and $\mathcal{G}_m$ is either *true* or fail.

4. Now, we check the value of T_m. If $T_m = \{\,\}$, the algorithm terminates since all clauses have been completely executed. Otherwise, we identify the *last* state $\langle \tau_i; \mathcal{L}_i; \mathcal{G}_i; \sigma_i; \mathcal{S}_i; T_i \rangle$ in the above concolic execution such that
   - the previous transition $\overset{c_i}{\leadsto}$ is labeled with $c_i = \{\overline{\mathsf{c}(\ell_k, \theta_k)}\}$, $k > 0$, and
   - there exists $j \in \{1, \ldots, k\}$ such that either $\ell_j \in T_m$ or $\mathcal{L}_i$ contains (not necessarily in a top position) some labels from T_m; the reason to also consider the labels from $\mathcal{L}_i$ is that considering an alternative clause may help to complete the execution of *all* the clauses in the current clause stack. Either way, we choose a clause j in a non-deterministic way.

   Therefore, we have a prefix of the complete concolic execution of the form:

$$\langle \tau_0; \mathcal{L}_0; \mathcal{G}_0; \sigma_0; \mathcal{S}_0; T_0 \rangle \overset{c_1}{\leadsto} \ldots \overset{c_{i-1}}{\leadsto} \langle \tau_{i-1}; \mathcal{L}_{i-1}; \mathcal{G}_{i-1}; \sigma_{i-1}; \mathcal{S}_{i-1}; T_{i-1} \rangle \overset{c_i}{\leadsto} \langle \tau_i; \mathcal{L}_i; \mathcal{G}_i; \sigma_i; \mathcal{S}_i; T_i \rangle$$

   and we are interested in the (possibly) partial computed answer $\sigma_{i-1}\theta_j$, since it will allow us to explore a different execution path, possibly covering some more program clauses.
   Hence, we have a second test case: $\mathcal{G}'_0 = main(\overline{X_n})\sigma_{i-1}\theta_j\gamma$, where γ is a substitution that is only aimed at grounding the input parameters $in(main/n)$ of *main* using arbitrary values (of the right type, preferably minimal ones).
5. Finally, we consider the initial state $\langle \mathcal{G}'_0; id; [\,] \rangle$ and obtain a new trace using the concrete execution semantics τ', so that a new initial symbolic state is defined as follows: $\langle \tau'; [\,]; main(\overline{X_n}); id; [\,]; T_m \rangle$ and the process starts again (i.e., we jump again to step 3). Observe that the initial state includes the set of clause labels T_m obtained in the last state of the previous concolic execution, in order to avoid producing new tests for clauses that are already covered by some previous test case.

Let us now illustrate the complete test case generation process with an example.

5.2 Test Case Generation in Practice

In this section, we illustrate the generation of test cases using a slight modification of the program in Example 1:

```
(1) main(L,N,R) :-
      length(L,N),
      rev(L,[],R).
(2) main(_L,_N,error).

(3) rev([],A,A).
(4) rev([H|T],Acc,Res) :-
      is_list(Acc),
      rev(T,[H|Acc],Res).
(5) is_list([]).
(6) is_list([_H|T]) :-
       is_list(T).

(7) length([],0).
(8) length([_H|R],s(N)) :-
      length(R,N).
```

$$
\begin{array}{lllll}
 & \langle [\mathtt{u}(1,2),\mathtt{u}(8),\mathtt{b}(2)];\,[\,];\ \mathtt{main}(\mathtt{L},\mathtt{N},\mathtt{R});\ & id;\ & [\,];\,\{1,\ldots,8\}\rangle \\
\xrightarrow{\{\}} & \langle [\mathtt{u}(8),\mathtt{b}(2)];\,[1];\ \begin{pmatrix}\mathtt{length}(\mathtt{L},\mathtt{N}),\\ \mathtt{rev}(\mathtt{L},[\,],\mathtt{R}),\mathtt{e}(1)\end{pmatrix};\ & id;\ & \mathcal{S};\,\{1,\ldots,8\}\rangle \\
\xrightarrow{\{\mathtt{c}(7,\{\mathtt{L}/[\,],\mathtt{N}/0\})\}} & \langle [\mathtt{b}(2)];\,[8,1];\ \begin{pmatrix}\mathtt{length}(\mathtt{L}',\mathtt{N}'),\mathtt{e}(8),\\ \mathtt{rev}([\mathtt{X}|\mathtt{L}'],[\,],\mathtt{R}),\mathtt{e}(1)\end{pmatrix};\ & \{\mathtt{L}/[\mathtt{X}|\mathtt{L}'],\mathtt{N}/\mathtt{s}(\mathtt{N}')\};\ & \mathcal{S};\,\{1,\ldots,8\}\rangle \\
\xrightarrow{\{\mathtt{c}(7,\{\mathtt{L}'/[\,],\mathtt{N}'/0\}),\ldots\}} & \langle [\,];\,[2];\ \mathtt{e}(2);\ & \{\mathtt{R}/\mathtt{error}\};\ & [\,];\,\{1,\ldots,8\}\rangle \\
\xrightarrow{\{\}} & \langle [\,];\,[\,];\ \mathtt{true};\ & \{\mathtt{R}/\mathtt{error}\};\ & [\,];\,\{1,3,\ldots,8\}\rangle
\end{array}
$$

with $\mathcal{S} = [(2;[2];\{\mathtt{R}/\mathtt{error}\};\mathtt{e}(2))]$;

Fig. 5. Concolic execution for $\langle [\mathtt{u}(1,2),\mathtt{u}(8),\mathtt{b}(2)];[\,];\mathtt{main}(\mathtt{L},\mathtt{N},\mathtt{R});id;[\,];\{1,2,\ldots,8\}\rangle$

Observe that, in this example, using a random generation of test cases would be useless since the length of the generated list and the second argument would hardly coincide. Also, using standard symbolic execution might be difficult too since the search space is infinite and, moreover, due to the use of predicate `rev` that includes an accumulating parameter, there are goals that are not instances of any previous goal, thus requiring some powerful abstraction operators.

Using concolic execution, though, we can easily generate appropriate test cases.

First iteration. We start with a random initial goal, e.g., $\mathtt{main}([\mathtt{a},\mathtt{b}],\mathtt{s}(0),\mathtt{R})$, where the input arguments $[1,2]$ are assumed ground. The associated concrete execution is the following:

$$
\begin{array}{l}
\langle \mathtt{main}([\mathtt{a},\mathtt{b}],\mathtt{s}(0),\mathtt{R});\ id;\ [\,]\rangle \\
\quad \xrightarrow{\mathtt{u}(1,2)} \langle \mathtt{length}([\mathtt{a},\mathtt{b}],\mathtt{s}(0)),\mathtt{rev}([\mathtt{a},\mathtt{b}],[\,],\mathtt{R});\ id;\ [(2;\{\mathtt{R}/\mathtt{error}\};\mathtt{true})]\rangle \\
\quad \xrightarrow{\mathtt{u}(8)} \langle \mathtt{length}([\mathtt{b}],0),\mathtt{rev}([\mathtt{a},\mathtt{b}],[\,],\mathtt{R});\ id;\ [(2;\{\mathtt{R}/\mathtt{error}\};\mathtt{true})]\rangle \\
\quad \xrightarrow{\mathtt{b}(2)} \langle \mathtt{true};\ \{\mathtt{R}/\mathtt{error}\};\ [\,]\rangle
\end{array}
$$

and its associated trace is thus $\tau = [\mathtt{u}(1,2),\mathtt{u}(8),\mathtt{b}(2)]$.

Now, we use concolic execution and produce the computation shown in Figure 5. Therefore, by executing $\mathtt{main}([\mathtt{a},\mathtt{b}],\mathtt{s}(0),\mathtt{R})$ only clause (2) is completely evaluated. According to the previous algorithm for test case generation, we now consider the following prefix of the concolic execution:

$$
\langle [\mathtt{u}(1,2),\mathtt{u}(8),\mathtt{b}(2)];[\,];\mathtt{main}(\mathtt{L},\mathtt{N},\mathtt{R});id;[\,];\{1,\ldots,8\}\rangle \xrightarrow{\{\}} \ldots \xrightarrow{\{\mathtt{c}(7,\{\mathtt{L}'/[\,],\mathtt{N}'/0\})\}} \langle\ldots\rangle
$$

and the associated substitution $\{\mathtt{L}/[\mathtt{X}],\mathtt{N}/\mathtt{s}(0)\}$.

Second iteration. Now, we consider the goal $\mathtt{main}([\mathtt{X}],\mathtt{s}(0),\mathtt{R})$. Since the first two arguments must be ground, as mentioned before, we apply a minimal ground-

$\langle \mathtt{main}([\mathtt{a}], \mathtt{s}(\mathtt{0}), \mathtt{R}); id; [\,] \rangle$

$\overset{\mathtt{u}(1,2)}{\rightarrow}$ $\langle \mathtt{length}([\mathtt{a}], \mathtt{s}(\mathtt{0})), \mathtt{rev}([\mathtt{a}], [\,], \mathtt{R}); id; [(2; \{\mathtt{R}/\mathtt{error}\}; \mathtt{true})] \rangle$

$\overset{\mathtt{u}(8)}{\rightarrow}$ $\langle \mathtt{length}([\,], \mathtt{0}), \mathtt{rev}([\mathtt{a}], [\,], \mathtt{R}); id; [(2; \{\mathtt{R}/\mathtt{error}\}; \mathtt{true})] \rangle$

$\overset{\mathtt{u}(7)}{\rightarrow}$ $\langle \mathtt{rev}([\mathtt{a}], [\,], \mathtt{R}); id; [2; (\{\mathtt{R}/\mathtt{error}\}; \mathtt{true})] \rangle$

$\overset{\mathtt{u}(4)}{\rightarrow}$ $\langle \mathtt{is\_list}([\,]), \mathtt{rev}([\,], [\mathtt{a}], \mathtt{R}); id; [(2; \{\mathtt{R}/\mathtt{error}\}; \mathtt{true})] \rangle$

$\overset{\mathtt{u}(5)}{\rightarrow}$ $\langle \mathtt{rev}([\,], [\mathtt{a}], \mathtt{R}); id; [(2; \{\mathtt{R}/\mathtt{error}\}; \mathtt{true})] \rangle$

$\overset{\mathtt{u}(3)}{\rightarrow}$ $\langle \mathtt{true}; \{\mathtt{R}/[\mathtt{a}]\}; [(2; \{\mathtt{R}/\mathtt{error}\}; \mathtt{true})] \rangle$

Fig. 6. Concrete execution for $\mathtt{main}([\mathtt{a}], \mathtt{s}(\mathtt{0})), \mathtt{R})$.

ing substitution and get, e.g., $\mathtt{main}([\mathtt{a}], \mathtt{s}(\mathtt{0}), \mathtt{R})$. The concrete execution, which is shown in Figure 6, computes the following trace:

$$\tau'' = [\mathtt{u}(1,2), \mathtt{u}(8), \mathtt{u}(7), \mathtt{u}(4), \mathtt{u}(5), \mathtt{u}(3)]$$

Then, we use concolic execution again as shown in Figure 7. Therefore, according to the algorithm, we consider the following prefix of the concolic execution:

$$\langle [\mathtt{u}(1,2), \mathtt{u}(8), \ldots]; []; \mathtt{main}(\mathtt{L}, \mathtt{N}, \mathtt{R}); []; \{4,5,6,8\} \rangle \overset{\{\}}{\rightarrow} \langle \ldots \rangle \overset{\{\mathtt{c}(7,\theta_1)\}}{\rightarrow} \langle \ldots \rangle \overset{\{\mathtt{c}(8,\theta_2)\}}{\rightarrow} \langle \ldots \rangle$$

with the associated substitution $\sigma_1\theta_2 = \{\mathtt{L}/[\mathtt{X}, \mathtt{Y}|\mathtt{L}''], \mathtt{N}/\mathtt{s}(\mathtt{s}(\mathtt{N}''))\}$.

Third (and last) iteration As in the previous case, the instantiated goal, $\mathtt{main}([\mathtt{X}, \mathtt{Y}|\mathtt{L}''], \mathtt{s}(\mathtt{s}(\mathtt{N}'')), \mathtt{R})$, is not ground enough according to its input mode and, thus, we apply a minimal grounding substitution. In this case, we get the initial goal $\mathtt{main}([\mathtt{a}, \mathtt{b}], \mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{R})$. Here, the concrete execution is basically the same shown in Figure 2, except for the last (backtracking) step. Therefore, the associated trace is

$$\tau''' = [\mathtt{u}(1,2), \mathtt{u}(8), \mathtt{u}(8), \mathtt{u}(7), \mathtt{u}(4), \mathtt{u}(5), \mathtt{u}(4), \mathtt{u}(6), \mathtt{u}(5), \mathtt{u}(3)]$$

Now, concolic execution from the initial state

$$\langle [\mathtt{u}(1,2), \mathtt{u}(8), \mathtt{u}(8), \mathtt{u}(7), \mathtt{u}(4), \mathtt{u}(5), \mathtt{u}(4), \mathtt{u}(6), \mathtt{u}(5), \mathtt{u}(3)]; []; \mathtt{main}(\mathtt{L}, \mathtt{N}, \mathtt{R}); id; []; \{6\} \rangle$$

proceeds similarly to the derivation shown in Figure 7, but now clause (6) is also completely evaluated, which means that the algorithm terminates successfully.

To summarize, concolic testing generated four test cases:

$\mathtt{main}([\mathtt{a}, \mathtt{b}], \mathtt{s}(\mathtt{0}), \mathtt{R})$	$\mathtt{main}([\mathtt{a}], \mathtt{s}(\mathtt{0}), \mathtt{R})$
$\mathtt{main}([\,], \mathtt{0}, \mathtt{R})$	$\mathtt{main}([\mathtt{a}, \mathtt{b}], \mathtt{s}(\mathtt{s}(\mathtt{0})), \mathtt{R})$

which suffice to cover the complete evaluation of all program clauses.

$$
\begin{array}{lrlll}
 & \langle [\mathtt{u}(1,2),\mathtt{u}(8),\ldots]; [\,]; & \mathtt{main}(\mathtt{L},\mathtt{N},\mathtt{R}); & id; & [\,]; \{4,5,6,8\}\rangle \\
\xrightarrow{\{\}} & \langle [\mathtt{u}(8),\mathtt{u}(7),\ldots]; [1]; & \begin{pmatrix}\mathtt{length}(\mathtt{L},\mathtt{N}),\\ \mathtt{rev}(\mathtt{L},[\,],\mathtt{R}),\mathtt{e}(1)\end{pmatrix}; & id; & \mathcal{S}; \{4,5,6,8\}\rangle \\
\xrightarrow{\{\mathtt{c}(7,\theta_1)\}} & \langle [\mathtt{u}(7),\mathtt{u}(4),\ldots]; [8,1]; & \begin{pmatrix}\mathtt{length}(\mathtt{L}',\mathtt{N}'),\mathtt{e}(8),\\ \mathtt{rev}([\mathtt{X}|\mathtt{L}'],[\,],\mathtt{R}),\mathtt{e}(1)\end{pmatrix}; & \sigma_1; & \mathcal{S}; \{4,5,6,8\}\rangle \\
\xrightarrow{\{\mathtt{c}(8,\theta_2)\}} & \langle [\mathtt{u}(4),\mathtt{u}(5),\mathtt{u}(3)]; [7,8,1]; & \begin{pmatrix}\mathtt{e}(7),\mathtt{e}(8),\\ \mathtt{rev}([\mathtt{X}|\mathtt{L}'],[\,],\mathtt{R}),\mathtt{e}(1)\end{pmatrix}; & \sigma_1\sigma_2; & \mathcal{S}; \{4,5,6,8\}\rangle \\
\xrightarrow{\{\}} & \langle [\mathtt{u}(4),\mathtt{u}(5),\mathtt{u}(3)]; [8,1]; & \mathtt{e}(8),\mathtt{rev}([\mathtt{X}|\mathtt{L}'],[\,],\mathtt{R}),\mathtt{e}(1); & \sigma_1\sigma_2; & \mathcal{S}; \{4,5,6,8\}\rangle \\
\xrightarrow{\{\}} & \langle [\mathtt{u}(4),\mathtt{u}(5),\mathtt{u}(3)]; [1]; & \mathtt{rev}([\mathtt{X}|\mathtt{L}'],[\,],\mathtt{R}),\mathtt{e}(1); & \sigma_1\sigma_2; & \mathcal{S}; \{4,5,6\}\rangle \\
\xrightarrow{\{\}} & \langle [\mathtt{u}(5),\mathtt{u}(3)]; [4,1]; & \begin{pmatrix}\mathtt{is\_list}([\,]),\mathtt{rev}(\mathtt{L}',[\mathtt{X}],\mathtt{R}),\\ \mathtt{e}(4),\mathtt{e}(1)\end{pmatrix}; & \sigma_1\sigma_2; & \mathcal{S}; \{4,5,6\}\rangle \\
\xrightarrow{\{\}} & \langle [\mathtt{u}(3)]; [5,4,1]; & \mathtt{e}(5),\mathtt{rev}(\mathtt{L}',[\mathtt{X}],\mathtt{R}),\mathtt{e}(4),\mathtt{e}(1); & \sigma_1\sigma_2; & \mathcal{S}; \{4,5,6\}\rangle \\
\xrightarrow{\{\}} & \langle [\mathtt{u}(3)]; [4,1]; & \mathtt{rev}(\mathtt{L}',[\mathtt{X}],\mathtt{R}),\mathtt{e}(4),\mathtt{e}(1); & \sigma_1\sigma_2; & \mathcal{S}; \{4,6\}\rangle \\
\xrightarrow{\{\}} & \langle [\,]; [3,4,1]; & \mathtt{e}(3),\mathtt{e}(4),\mathtt{e}(1); & \sigma_1\sigma_2; & \mathcal{S}; \{4,6\}\rangle \\
\xrightarrow{\{\}} & \langle [\,]; [4,1]; & \mathtt{e}(4),\mathtt{e}(1); & \sigma_1\sigma_2; & \mathcal{S}; \{4,6\}\rangle \\
\xrightarrow{\{\}} & \langle [\,]; [1]; & \mathtt{e}(1); & \sigma_1\sigma_2; & \mathcal{S}; \{6\}\rangle \\
\xrightarrow{\{\}} & \langle [\,]; [\,]; & \mathtt{true}; & \sigma_1\sigma_2; & \mathcal{S}; \{6\}\rangle
\end{array}
$$

with $\mathcal{S} = [(2;[2];\{\mathtt{R}/\mathtt{error}\};\mathtt{e}(2))]$;

Fig. 7. Concolic execution for $\langle [\mathtt{u}(1,2),\mathtt{u}(8),\ldots]; id; [\,]; \mathtt{main}(\mathtt{L},\mathtt{N},\mathtt{R}); [\,]; \{4,5,6,8\}\rangle$

In general, when the test case generation algorithm terminates, concolic testing is *sound* (i.e., there are no false positives since only concrete executions are considered) and *complete* (in the sense that all clauses are completely evaluated when using the computed test cases, i.e., we get a 100% coverage). When the process is stopped (e.g., because it does not terminate or takes too much time), our test case generation is only sound. Note that this contrasts with other approaches to test case generation in Prolog (and CLP), e.g., [8, 15], where full coverage is not considered. In [1], however, the authors consider a refined notion of coverage criterion: a pair $\langle TC, SC\rangle$, where TC is a termination criterion (e.g., the maximum number of recursive calls to a predicate allowed in symbolic execution) and SC is the selection criterion, used to determine which test cases must be produced. In particular, using the SC program-points(P), where P includes all program points, amounts to the statement coverage that we consider in this paper. On the other hand, the authors have introduced a refinement for driving symbolic execution by means of "trace terms" (terms representing the shape of

a particular subset of the SLD search space). The idea behind this technique is closer to that of concolic execution, although they do not have the possibility of using concrete data in symbolic executions (which is one of the main advantages of concolic execution).

6 Concluding Remarks and Future Work

We have introduced a novel approach to Prolog testing and debugging. The so called concolic execution that mixes concrete and symbolic execution has been shown quite successful in other programming paradigms, especially when dealing with large applications. Therefore, it might have a great potential for Prolog testing, too.

In this paper, we have only considered a simple form of coverage, *statement* coverage, and a limited scenario: pure Prolog without negation. Nevertheless, the main distinctive features of the Prolog programming language—i.e., unification, non-determinism and backtracking—are present here, so adapting the standard concolic execution approach [6, 16] to Prolog was not trivial. The challenge, now, is experimentally verifying the effectiveness and scalability of our approach with real Prolog programs. For this purpose, though, we first need to extend concolic execution to deal with negation, built-in's, extra-logical features, etc. For this purpose, we will consider the linear operational semantics of [17], and its symbolic version [5], as a promising starting point.

Acknowledgements

The author gratefully acknowledges the anonymous referees and the participants of LOPSTR 2014 for many useful comments and suggestions. I would also like to thank Fred Mesnard and Etienne Payet for their remarks to improve the paper.

References

1. E. Albert, P. Arenas, M. Gómez-Zamalloa, and J.M. Rojas. Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-Based Instance, and Actor-Based Concurrency. In *SFM 2014*, Springer LNCS 8483, pages 263–309,, 2014.
2. F. Belli and O. Jack. Implementation-Based Analysis and Testing of Prolog Programs. In *ISSTA*, pages 70–80. ACM, 1993.
3. L.A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference (ACM'76)*, pages 488–491, 1976.
4. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M.H. Sørensen. Conjunctive Partial Deduction: Foundations, Control, Algorihtms, and Experiments. *Journal of Logic Programming*, 41(2&3):231–277, 1999.
5. J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs. Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In *PPDP'12*, pages 1–12. ACM, 2012.
6. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proc. of PLDI'05*, pages 213–223. ACM, 2005.

7. P. Godefroid, M.Y. Levin, and D.A. Molnar. Sage: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.
8. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test case generation for object-oriented imperative languages in CLP. *TPLP*, 10(4-6):659–674, 2010.
9. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
10. M. Leuschel. The DPPD (Dozens of Problems for Partial Deduction) Library of Benchmarks, 2007. `http://www.ecs.soton.ac.uk/~mal/systems/dppd.html`.
11. J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.
12. J.W. Lloyd and J.C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.
13. B. Martens and J. Gallagher. Ensuring Global Termination of Partial Deduction while Allowing Flexible Polyvariance. In *Proc. of ICLP'95*, pages 597–611. MIT Press, 1995.
14. C.S. Pasareanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE*, pages 179–180. ACM, 2010.
15. J.M. Rojas and M. Gómez-Zamalloa. A Framework for Guided Test Case Generation in Constraint Logic Programming. In *Proc. of LOPSTR'12*, pages 176–193. Springer LNCS 7844, 2012.
16. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. of ESEC/SIGSOFT FSE 2005*, pages 263–272. ACM, 2005.
17. T. Ströder, F. Emmes, P. Schneider-Kamp, J. Giesl, and C. Fuhs. A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog. In *LOPSTR'11*, pages 237–252. Springer LNCS 7225, 2011.