Concolic Testing in Logic Programming

FRED MESNARD, ÉTIENNE PAYET
LIM - université de la Réunion, France
(*e-mail:* {fred,epayet}@univ-reunion.fr)

GERMÁN VIDAL
MiST, DSIC, Universitat Politècnica de València
(*e-mail:* gvidal@dsic.upv.es)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Software testing is one of the most popular validation techniques in the software industry. Surprisingly, we can only find a few approaches to testing in the context of logic programming. In this paper, we introduce a systematic approach for dynamic testing that combines both concrete and symbolic execution. Our approach is fully automatic and guarantees full path coverage when it terminates. We prove some basic properties of our technique and illustrate its practical usefulness through a prototype implementation.

KEYWORDS: Symbolic execution, logic programming, testing.

1 Introduction

Essentially, software validation aims at ensuring that the developed software complies with the original requirements. One of the most popular validation approaches is *software testing*, a process that involves producing a test suite and then executing the system with these test cases. The main drawback of this approach is that designing a test suite with a high code coverage —i.e., covering as many execution paths as possible— is a complex and time-consuming task. As an alternative, one can use a tool for the random generation of test cases, but then we are often faced with a poor code coverage; there are though some hybrid approaches where random generation is driven by the user, as in QuickCheck (Claessen and Hughes 2000), but then again the process may become complex and time-consuming.

Another popular, fully automatic approach to test case generation is based on *symbolic execution* (King 1976; Clarke 1976). Basically, symbolic execution considers unknown (symbolic) values for the input parameters, and then explores nondeterministically all feasible execution paths. Symbolic states include now a *path condition* that stores the current constraints on symbolic values, i.e., the conditions that must hold to reach a particular execution point. Then, for each final state, a test case is produced by solving the constraints in the associated path condition.

A drawback of the previous approach, though, is that the constraints in the path condition may become very complex. When these constraints are not solvable, the only sound way to proceed is to stop the execution path, often giving rise to a poor coverage. Recently, a new variant called *concolic execution* (Godefroid et al. 2005; Sen et al. 2005) that combines both *conc*rete and symb*olic* execution has been proposed as a basis for both model checking and test case generation. The main advantage is that, now, when the constraints in the symbolic execution become too complex, one can still take some values from the concrete execution to simplify them. This is sound and often allows one to explore a larger execution space. Some successful tools that are based on concolic execution are, e.g., SAGE (Godefroid et al. 2012) and Java Pathfinder (Pasareanu and Rungta 2010).

In the context of the logic programming paradigm, one can find a flurry of static, complete techniques for software analysis and verification. However, only a few dynamic, sound techniques for program validation have been proposed. Dynamic, typically incomplete, techniques have proven very useful for software validation in other paradigms. Therefore, we expect concolic execution to be also useful for testing logic programs.

In this paper, we introduce a new, fully automatic scheme for *concolic testing* in logic programming. As in other paradigms, concolic testing may help the programmer to systematically find program bugs and generate test cases with a good code coverage. As it is common, our approach is always sound but usually incomplete. In the context of logic programming, we consider that "full path coverage" involves calling each predicate in all possible ways. Consider, e.g., the logic program $P = \{p(a).,\ p(b).\}$. Here, one could assume that the execution of the goals in $\{p(a), p(b)\}$ is enough for achieving a full path coverage. However, in this paper we consider that full path coverage requires, e.g., the set $\{p(X), p(a), p(b), p(c)\}$ so that we have a goal that matches both clauses, one that only matches the first clause, one that only matches the second clause, and one that matches no clause. We call this notion *choice* coverage, and it is specific of logic programming. To the best of our knowledge, such a notion of coverage has not been considered before. Typically, only a form of *statement* coverage has been considered, where only the clauses used in the considered executions are taken into account. For guaranteeing choice coverage, a new type of unification problems must be solved: we have to produce goals in which the selected atom A matches the heads of some clauses, say $H_1, \ldots, H_n$, but does not match the heads of some other clauses, say $H'_1, \ldots, H'_m$. We provide a constructive algorithm for solving such unifiability problems.

A prototype implementation of the concolic testing scheme for pure Prolog, called `contest`, is publicly available from `http://kaz.dsic.upv.es/contest.html`. The results from an experimental evaluation point out the usefulness of the approach. Besides logic programming and Prolog, our technique might also be useful for other programming languages since there exist several transformational approaches that "compile in" programs to Prolog, like, e.g., (Gómez-Zamalloa et al. 2010).

2 Concrete Semantics

The semantics of a logic program is usually given in terms of the SLD relation on goals (Lloyd 1987). In this section, we present instead a *local* semantics which is similar to that of Ströder et al. (2011). Basically, this semantics deals with states that contain all the necessary information to perform the next step (in contrast to the usual semantics, where the SLD tree built so far is also needed, e.g., for dealing

$$(\mathsf{success})\ \frac{}{\langle \mathsf{true}_\delta \,|\, S\rangle \rightarrow \langle \textsc{success}_\delta\rangle}$$

$$(\mathsf{failure})\ \frac{}{\langle (\mathsf{fail},\mathcal{B})_\delta\rangle \rightarrow \langle \textsc{fail}_\delta\rangle} \qquad (\mathsf{backtrack})\ \frac{S \neq \epsilon}{\langle (\mathsf{fail},\mathcal{B})_\delta \,|\, S\rangle \rightarrow \langle S\rangle}$$

$$(\mathsf{choice})\ \frac{\mathsf{clauses}(A,\mathcal{P}) = (c_1,\ldots,c_n) \wedge n > 0}{\langle (A,\mathcal{B})_\delta \,|\, S\rangle \rightarrow \langle (A,\mathcal{B})^{c_1}_\delta \,|\ldots|\, (A,\mathcal{B})^{c_n}_\delta \,|\, S\rangle} \qquad (\mathsf{choice\_fail})\ \frac{\mathsf{clauses}(A,\mathcal{P}) = \{\}}{\langle (A,\mathcal{B})_\delta \,|\, S\rangle \rightarrow \langle \mathsf{fail}_\delta \,|\, S\rangle}$$

$$(\mathsf{unfold})\ \frac{\mathsf{mgu}(A,H_1) = \sigma}{\langle (A,\mathcal{B})^{H_1 \leftarrow \mathcal{B}_1}_\delta \,|\, S\rangle \rightarrow \langle (\mathcal{B}_1\sigma, \mathcal{B}\sigma)_{\delta\sigma} \,|\, S\rangle}$$

Fig. 1. Concrete semantics

with the cut). In contrast to (Ströder et al. 2011), for simplicity, in this paper we only consider definite logic programs. However, the main difference w.r.t. (Ströder et al. 2011) comes from the fact that our concrete semantics only considers the computation of the first solution for the initial goal. This is the way most Prolog applications are used and, thus, our semantics should consider this behaviour in order to measure the coverage in a realistic way.

Before presenting the transition rules of the concrete semantics, let us introduce some auxiliary notions and notations. We refer the reader to (Apt 1997) for the standard definitions and notations for logic programs. The semantics is defined by means of a transition system on *states* of the form $\langle \mathcal{B}^1_{\delta_1} \,|\ldots|\, \mathcal{B}^n_{\delta_n}\rangle$, where $\mathcal{B}^1_{\delta_1} \,|\ldots|\, \mathcal{B}^n_{\delta_n}$ is a sequence of goals labeled with substitutions (the answer computed so far, when restricted to the variables of the initial goal). We denote sequences with $S, S', \ldots$, where ϵ denotes the empty sequence. In some cases, we label a goal $\mathcal{B}$ both with a substitution and a program clause, e.g., $\mathcal{B}^{H \leftarrow \mathcal{B}}_\delta$, which is used to determine the next clause to be used for an SLD resolution step (see rules choice and unfold in Fig. 1). Note that the clauses of the program are not included in the state but considered a global parameter since they are static. In the following, given an atom A rooted by a defined predicate and a logic program P, $\mathsf{clauses}(A,P)$ returns the sequence of renamed apart program clauses $c_1,\ldots,c_n$ from P whose head unifies with A.

For simplicity, w.l.o.g., we only consider *atomic* initial goals. Therefore, given an atom A, an initial state has the form $\langle A_{id}\rangle$, where *id* denotes the identity substitution. The transition rules, shown in Figure 1, proceed as follows:

- In rules success and failure, we use fresh constants to denote a final state: $\langle \textsc{success}_\delta\rangle$ denotes that a sucessful derivation ended with computed answer substitution δ, while $\langle \textsc{fail}_\delta\rangle$ denotes a finitely failing derivation; recording δ for failing computations might be useful for debugging purposes.
- Rule backtrack applies when the first goal in the sequence finitely fails, but there is at least one alternative choice.
- Rule choice represents the first step of an SLD resolution step. If there is at least one clause whose head unifies with the leftmost atom, this rule introduces as many copies of a goal as clauses returned by function clauses. If there is at least one matching clause, unfolding is then performed by rule unfold. Otherwise, if there is no matching clause, rule choice_fail returns fail so that either rule failure or backtrack applies next.

Example 1
Consider the following logic program:

$$\begin{array}{lll} p(s(a)). & q(a). & r(a). \\ p(s(X)) \leftarrow q(X). & q(b). & r(c). \\ p(f(X)) \leftarrow r(X). & & \end{array}$$

Given the initial goal $p(f(X))$, we have the following successful computation (for clarity, we label each step with the applied rule):

$$\begin{array}{lllll} \langle p(f(X))_{id} \rangle & \rightarrow^{\mathsf{choice}} & \langle p(f(X))_{id}^{p(f(Y)) \leftarrow r(Y)} \rangle & \rightarrow^{\mathsf{unfold}} & \langle r(X)_{id} \rangle \\ & \rightarrow^{\mathsf{choice}} & \langle r(X)_{id}^{r(a)} \,|\, r(X)_{id}^{r(c)} \rangle & \rightarrow^{\mathsf{unfold}} & \langle \mathsf{true}_{\{X/a\}} \,|\, r(X)_{id}^{r(c)} \rangle \\ & \rightarrow^{\mathsf{success}} & \langle \textsc{success}_{\{X/a\}} \rangle & & \end{array}$$

Therefore, we have a successful computation for $p(f(X))$ with computed answer $\{X/a\}$. Observe that only the first answer is considered.

We do not formally prove the correctness of the concrete semantics, but it is an easy consequence of the correctness of the semantics in (Ströder et al. 2011). Note that our rules can be seen as an instance for pure Prolog without negation, where only the computation of the first answer for the initial goal is considered.

3 Concolic Execution Semantics

In this section, we introduce a concolic execution semantics for logic programs that is a conservative extension of the concrete semantics of the previous section. In this semantics, *concolic states* have the form $\langle S \,]\![\, S' \rangle$, where S and S' are sequences of (possibly labeled) concrete and symbolic goals, respectively. In logic programming, the notion of *symbolic* execution is very natural: the structure of both S and S' is the same, and the only difference is that some atoms might be less instantiated in S' than in S.

In the following, we let $\overline{o_n}$ denote the sequence of syntactic objects $o_1, \ldots, o_n$, and $\mathcal{V}ar(o)$ denotes the set of variables in the syntactic object o. Moreover, given an atom A, we let $root(A) = p/n$ if $A = p(\overline{t_n})$. Now, given an atom A with $root(A) = p/n$, an *initial concolic state* has the form $\langle A_{id} \,]\![\, p(\overline{X_n})_{id} \rangle$, where $\overline{X_n}$ are different fresh variables. In the following, we assume that every clause c has a corresponding unique label, which we denote by $\ell(c)$. By abuse of notation, we also denote by $\ell(\overline{c_n})$ the set of labels $\{\ell(c_1), \ldots, \ell(c_n)\}$.

The semantics is given by the rules of the labeled transition relation $\rightsquigarrow$ shown in Figure 2. Here, we consider two kinds of labels for the transition relation:

- The empty label, $\diamond$, which is often implicit.
- A label of the form $c(\ell(\overline{c_n}), \ell(\overline{d_k}))$, which represents a choice step. Here, $\ell(\overline{c_n})$ are the labels of the clauses matching the selected atom in the concrete goal, while $\ell(\overline{d_k})$ are the labels of the clauses matching the selected atom in the corresponding symbolic goal. Note that $\ell(\overline{c_n}) \subseteq \ell(\overline{d_k})$ since the concrete goal is always an instance of the symbolic goal (see Theorem 1 below).

For each transition step $\mathcal{C}_1 \rightsquigarrow_{c(\mathcal{L}_1, \mathcal{L}_2)} \mathcal{C}_2$, the first set of labels, $\mathcal{L}_1$, is used to determine the execution *trace* of a concrete goal (see below). Traces are needed to

$$\text{(success)}\ \frac{}{\langle \mathsf{true}_\delta \,|\, S \rrbracket\ \mathsf{true}_\theta \,|\, S'\rangle \leadsto_\diamond \langle \textsc{success}_\delta \rrbracket\ \textsc{success}_\theta\rangle}$$

$$\text{(failure)}\ \frac{}{\langle (\mathsf{fail},\mathcal{B})_\delta \rrbracket\ (\mathsf{fail},\mathcal{B}')_\theta\rangle \leadsto_\diamond \langle \textsc{fail}_\delta \rrbracket\ \textsc{fail}_\theta\rangle}$$

$$\text{(backtrack)}\ \frac{S \neq \epsilon}{\langle (\mathsf{fail},\mathcal{B})_\delta \,|\, S \rrbracket\ (\mathsf{fail},\mathcal{B}')_\theta \,|\, S'\rangle \leadsto_\diamond \langle S \rrbracket\ S'\rangle}$$

$$\text{(choice)}\ \frac{\mathsf{clauses}(A,\mathcal{P}) = \overline{c_n} \wedge n > 0 \wedge \mathsf{clauses}(A',\mathcal{P}) = \overline{d_k}}{\begin{array}{r}\langle (A,\mathcal{B})_\delta \,|\, S \rrbracket\ (A',\mathcal{B}')_\theta \,|\, S'\rangle \leadsto_{c(\ell(\overline{c_n}),\ell(\overline{d_k}))} \langle (A,\mathcal{B})_\delta^{c_1} \,|\, \ldots \,|\, (A,\mathcal{B})_\delta^{c_n} \,|\, S \\ \rrbracket\ (A',\mathcal{B}')_\theta^{c_1} \,|\, \ldots \,|\, (A',\mathcal{B}')_\theta^{c_n} \,|\, S'\rangle\end{array}}$$

$$\text{(choice\_fail)}\ \frac{\mathsf{clauses}(A,\mathcal{P}) = \{\} \wedge \mathsf{clauses}(A',\mathcal{P}) = \overline{c_k}}{\langle (A,\mathcal{B})_\delta \,|\, S \rrbracket\ (A',\mathcal{B}')_\theta \,|\, S'\rangle \leadsto_{c(\{\},\ell(\overline{c_k}))} \langle \mathsf{fail}_\delta \,|\, S \rrbracket\ \mathsf{fail}_\theta \,|\, S'\rangle}$$

$$\text{(unfold)}\ \frac{\mathsf{mgu}(A,H_1) = \sigma \wedge \mathsf{mgu}(A',H_1) = \sigma'}{\langle (A,\mathcal{B})_\delta^{H_1 \leftarrow \mathcal{B}_1} \,|\, S \rrbracket\ (A',\mathcal{B}')_\theta^{H_1 \leftarrow \mathcal{B}_1} \,|\, S'\rangle \leadsto_\diamond \langle (\mathcal{B}_1\sigma,\mathcal{B}\sigma)_{\delta\sigma} \,|\, S \rrbracket\ (\mathcal{B}_1\sigma',\mathcal{B}'\sigma')_{\theta\sigma'} \,|\, S'\rangle}$$

Fig. 2. Concolic execution semantics

keep track of the execution paths already explored. The second set of labels, $\mathcal{L}_2$, is used to compute new goals that follow alternative paths not yet explored, if any.

In the concolic execution semantics, we perform both concrete and symbolic execution steps in parallel. However, the symbolic execution does not explore all possible execution paths but only mimics the steps of the concrete execution; observe, e.g., rule choice in Figure 2, where the clauses labeling the copies of the symbolic goal are the same clauses $\overline{c_n}$ matching the concrete goal, rather than the set of clauses $\overline{d_k}$ (typically a superset of $\overline{c_n}$).

Example 2
Consider again the logic program of Example 1, now with clause labels:

$$\begin{array}{lll}(\ell_1)\ p(s(a)). & (\ell_4)\ q(a). & (\ell_6)\ r(a). \\ (\ell_2)\ p(s(X)) \leftarrow q(X). & (\ell_5)\ q(b). & (\ell_7)\ r(c). \\ (\ell_3)\ p(f(X)) \leftarrow r(X). & & \end{array}$$

Given the initial goal $p(f(X))$, we have the following concolic execution:

$$\begin{array}{lll}\langle p(f(X))_{id} \rrbracket\ p(N)_{id}\rangle & \leadsto^{\mathsf{choice}}_{c(\mathcal{L}_1,\mathcal{L}'_1)} & \langle p(f(X))_{id}^{p(f(Y))\leftarrow r(Y)} \rrbracket\ p(N)_{id}^{p(f(Y))\leftarrow r(Y)}\rangle \\ & \leadsto^{\mathsf{unfold}}_\diamond & \langle r(X)_{id} \rrbracket\ r(Y)_{\{N/f(Y)\}}\rangle \\ & \leadsto^{\mathsf{choice}}_{c(\mathcal{L}_2,\mathcal{L}'_2)} & \langle r(X)_{id}^{r(a)} \,|\, r(X)_{id}^{r(c)} \rrbracket\ r(Y)_{\{N/f(Y)\}}^{r(a)} \,|\, r(Y)_{\{N/f(Y)\}}^{r(c)}\rangle \\ & \leadsto^{\mathsf{unfold}}_\diamond & \langle \mathsf{true}_{\{X/a\}} \,|\, r(X)_{id}^{r(c)} \rrbracket\ \mathsf{true}_{\{N/f(a)\}} \,|\, r(Y)_{\{N/f(Y)\}}^{r(c)}\rangle \\ & \leadsto^{\mathsf{success}}_\diamond & \langle \textsc{success}_{\{X/a\}} \rrbracket\ \textsc{success}_{\{N/f(a)\}}\rangle \end{array}$$

where $\mathcal{L}_1 = \{\ell_3\}$, $\mathcal{L}'_1 = \{\ell_1,\ell_2,\ell_3\}$, and $\mathcal{L}_2 = \mathcal{L}'_2 = \{\ell_6,\ell_7\}$.

We note that, in principle, we only consider finite concolic executions. This is a reasonable assumption since one can expect concrete goals to compute the first answer finitely (unless the program is erroneous). We associate a *trace* to each concolic execution as follows:

Definition 1 (*trace*)
Let P be a program and $\mathcal{C}_0$ an initial concolic state. Let $E = (\mathcal{C}_0 \leadsto_{l_1} \ldots \leadsto_{l_m} \mathcal{C}_m)$, $m > 0$, be a concolic execution for $\mathcal{C}_0$ in P. Let $c(\mathcal{L}_1, \mathcal{L}'_1), \ldots, c(\mathcal{L}_k, \mathcal{L}'_k)$, $k \leqslant m$, be the sequence of labels in $l_1, \ldots, l_m$ which are different from $\diamond$. Then, the trace associated to the concolic execution E is $\mathit{trace}(E) = \mathcal{L}_1, \ldots, \mathcal{L}_k$.

Roughly speaking, a trace is just a sequence with the sets of labels of the matching clauses in each choice step. For instance, the trace associated to the concolic execution of Example 2 is $(\{\ell_3\}, \{\ell_6, \ell_7\})$, i.e., we have two unfolding steps with matching clauses $\{\ell_3\}$ and $\{\ell_6, \ell_7\}$, respectively. Note that traces ending with $\{\,\}$ represent failing derivations.

The following result states an essential invariant for concolic execution:

Theorem 1
Let P be a program and $\mathcal{C}_0 = \langle p(\overline{t_n})_{id} \rrbracket\, p(\overline{X_n})_{id}\rangle$ be an initial concolic state. Let $\mathcal{C}_0 \leadsto \ldots \leadsto \mathcal{C}_m$, $m \geqslant 0$, be a finite (possibly incomplete) concolic execution for $\mathcal{C}_0$ in P. Then, for all concolic states $C_i = \langle \mathcal{B}^c_\delta \,|\, S \rrbracket\, \mathcal{D}^{c'}_\theta \,|\, S'\rangle$, $i = 0, \ldots, m$, the following invariant holds: $|S| = |S'|$, $\mathcal{D} \leqslant \mathcal{B}$, $c = c'$ (if any), and $p(\overline{X_n})\theta \leqslant p(\overline{t_n})\delta$.

4 Concolic Testing

In this section, we introduce a concolic testing procedure for logic programs based on the concolic execution semantics of the previous section.

4.1 The Procedure

As we have seen in Section 3, the concolic execution steps labeled with $c(\mathcal{L}_1, \mathcal{L}_2)$ give us a hint of (potential) alternative execution paths. Consider, for instance, the concolic execution of Example 2. The first step is labeled with $c(\{\ell_3\}, \{\ell_1, \ell_2, \ell_3\})$. This means that the selected atom in the concrete goal only matched clause ℓ_3, while the selected atom in the symbolic goal matched clauses ℓ_1, ℓ_2 and ℓ_3. In principle, there are as many alternative execution paths as elements in $\mathcal{P}(\{\ell_1, \ell_2, \ell_3\}) \setminus \{\ell_3\}$; e.g., $\{\,\}$ denotes an execution path where the selected atom matches no clause, $\{\ell_1\}$ another path in which the selected atom *only* matches clause ℓ_1, $\{\ell_1, \ell_2, \ell_3\}$ another path where the selected atom matches all three clauses ℓ_1, ℓ_2 and ℓ_3, and so forth.

As mentioned before, when aiming at full choice coverage we need to solve both unification and disunification problems. Consider, e.g., that A is the selected atom in a goal, and that we want it to unify with the head of clause ℓ_1 but not with the heads of clauses ℓ_2 and ℓ_3. For this purpose, we introduce the following auxiliary function alt, which also includes some groundness requirements (see below). In the following, we let $\approx$ denote the unifiability relation, i.e., given atoms A, B, $A \approx B$ holds if $\mathsf{mgu}(A, B) \neq \mathsf{fail}$; correspondingly, $\neg(A \approx B)$ holds if $\mathsf{mgu}(A, B) = \mathsf{fail}$.

Definition 2 (alt)
Let A be an atom and $\mathcal{L}, \mathcal{L}'$ be sets of clause labels. Let $\mathcal{V}$ be a set of variables. The function $\mathsf{alt}(A, \mathcal{L}, \mathcal{L}', \mathcal{V})$ returns a substitution θ such that the following holds:

$$A\theta \approx H_1 \wedge \ldots \wedge A\theta \approx H_n \wedge \neg(A\theta \approx H_{n+1}) \wedge \ldots \wedge \neg(A\theta \approx H_m) \wedge \mathcal{V}\theta \text{ are ground}$$

where $H_1, \ldots, H_n$ are the heads of the clauses labeled by $\mathcal{L}$ and $H_{n+1}, \ldots, H_m$ are the heads of the clauses labeled by $\mathcal{L}' \backslash \mathcal{L}$, respectively. If such a substitution does not exist, then function alt returns fail.

We postpone to the next section the definition of a constructive algorithm for function alt. Now, we present an algorithm to systematically produce concrete initial goals so that all *feasible* choices in the execution paths are covered (unless the process runs forever). First, we introduce the following auxiliary definitions:

Definition 3 (conc, symb)
Let $\mathcal{C} = \langle \mathcal{B}^1_{\delta_1} \,|\, \ldots \,|\, \mathcal{B}^n_{\delta_n} \llbracket\, \mathcal{D}^1_{\theta_1} \,|\, \ldots \,|\, \mathcal{D}^n_{\theta_n} \rangle$ be a concolic state. Then, we let $\mathsf{conc}(\mathcal{C}) = \mathcal{B}^1_{\delta_1}$ denote the (first) concrete goal and $\mathsf{symb}(\mathcal{C}) = \mathcal{D}^1_{\theta_1}$ the (first) symbolic goal.

Definition 4 ($\mathsf{alt\_trace}$)
Let P be a program, $\mathcal{C}_0$ an initial concolic state, and $E = (\mathcal{C}_0 \rightsquigarrow_{l_1} \ldots \rightsquigarrow_{l_n} \mathcal{C}_n \rightsquigarrow_{c(\mathcal{L},\mathcal{L}')} \mathcal{C}_{n+1})$ be a (possibly incomplete) concolic execution for $\mathcal{C}_0$ in P. Then, the function $\mathsf{alt\_trace}$ denotes the following set of (potentially) alternative traces:

$$\begin{array}{ll} \mathsf{alt\_trace}(E) = \{\mathcal{L}_1, \ldots, \mathcal{L}_k, \mathcal{L}'' \mid & \mathit{trace}(\mathcal{C}_0 \rightsquigarrow_{l_1} \ldots \rightsquigarrow_{l_n} \mathcal{C}_n) = \mathcal{L}_1, \ldots, \mathcal{L}_k \\ & \text{and } \mathcal{L}'' \in (\mathcal{P}(\mathcal{L}') \setminus \mathcal{L}) \qquad\qquad\qquad \} \end{array}$$

For instance, given the following (partial) concolic execution E (from Example 2):

$$\begin{array}{lll} \langle p(f(X))_{id} \llbracket\, p(N)_{id} \rangle & \rightsquigarrow^{\mathsf{choice}}_{c(\mathcal{L}_1,\mathcal{L}'_1)} & \langle p(f(X))^{p(f(Y)) \leftarrow r(Y)}_{id} \llbracket\, p(N)^{p(f(Y)) \leftarrow r(Y)}_{id} \rangle \\ & \rightsquigarrow^{\mathsf{unfold}}_{\diamond} & \langle r(X)_{id} \llbracket\, r(Y)_{\{N/f(Y)\}} \rangle \\ & \rightsquigarrow^{\mathsf{choice}}_{c(\mathcal{L}_2,\mathcal{L}'_2)} & \langle r(X)^{r(a)}_{id} \,|\, r(X)^{r(c)}_{id} \llbracket\, r(Y)^{r(a)}_{\{N/f(Y)\}} \,|\, r(Y)^{r(c)}_{\{N/f(Y)\}} \rangle \end{array}$$

where $\mathcal{L}_2 = \mathcal{L}'_2 = \{\ell_6, \ell_7\}$, we have $\mathsf{alt\_trace}(E) = \{(\mathcal{L}_1, \{\,\}), (\mathcal{L}_1, \{\ell_6\}), (\mathcal{L}_1, \{\ell_7\})\}$.

Now, we introduce our concolic testing procedure. It takes as input a program and a random —e.g., provided by the user— initial atomic goal to a distinguished predicate $main/n$. In the following, we assume that each concrete initial goal $main(\overline{t_n})$ is existentially terminating w.r.t. Prolog's leftmost computation rule, i.e., either computes the *first* answer in a finite number of steps or finitely fails (Vasak and Potter 1986). For this purpose, we assume that $main/n$ has some associated *input* arguments, determined by a function input, so that an initial goal $main(\overline{t_n})$ existentially terminates if the terms $\mathsf{input}(main(\overline{t_n}))$ are ground. One could also consider that there are several combinations of input arguments that guarantee existential termination —this is similar to the modes of a predicate— but we only consider one set of input arguments for simplicity (extending the concolic testing algorithm would be straightforward). As mentioned before, assuming that concrete initial goals are existentially terminating is a reasonable assumption in practice.

Definition 5 (*concolic testing*)
Input: a logic program P and an atom $main(\overline{t_n})$ with $\mathsf{input}(main(\overline{t_n}))$ ground.
Output: a set TC of test cases.

1. Let $\mathit{Pending} := \{main(\overline{t_n})\}$, $TC := \{\}$, $\mathit{Traces} := \{\}$.
2. While $|\mathit{Pending}| \neq 0$ do

    (a) Take $A \in \mathit{Pending}$, $\mathit{Pending} := \mathit{Pending} \backslash \{A\}$, $TC := TC \cup \{A\}$.

   (b) Let $\mathcal{C}_0 = \langle A_{id} \,[\![\, main(\overline{X_n})_{id}\rangle$ and compute a successful or finitely failing derivation $E = (\mathcal{C}_0 \leadsto_{l_1} \ldots \leadsto_{l_m} \mathcal{C}_m)$.
   (c) Let $\mathit{Traces} := \mathit{Traces} \cup \mathit{trace}(E)$.
   (d) For each prefix $\mathcal{C}_0 \leadsto_{l_1} \ldots \leadsto_{l_j} \mathcal{C}_j \leadsto_{c(\mathcal{L},\mathcal{L}')} \mathcal{C}_{j+1}$ of E and for each (possibly partial) trace $\overline{\mathcal{L}_k}, \mathcal{L}_{k+1} \in \mathsf{alt\_trace}(\mathcal{C}_0 \leadsto_{l_1} \ldots \leadsto_{l_j} \mathcal{C}_j \leadsto_{c(\mathcal{L},\mathcal{L}')} \mathcal{C}_{j+1})$ which is not the prefix of any trace in Traces, add $main(\overline{X_n})\theta\theta'$ to $\mathit{Pending}$, where $\mathsf{symb}(\mathcal{C}_j) = (A_1, \mathcal{B})_\theta$ and $\mathsf{alt}(A_1, \mathcal{L}_{k+1}, \mathcal{L}', \mathcal{V}ar(\mathsf{input}(main(\overline{X_n})\theta))) = \theta' \neq \mathsf{fail}$.

3. Return the set TC of test cases

The soundness of concolic testing is immediate, since every error spotted in the concrete executions (e.g., a non-terminating call or a call with an incorrect number of arguments) is obviously a real bug. Completeness and termination are more subtle properties though.

In principle, one could argue that the concolic testing algorithm is partially correct in the sense that, if the algorithm terminates, the generated test cases cover all feasible paths. Our assumptions trivially guarantee that every considered concrete execution is finite (i.e., step 2.b in the loop of the concolic testing algorithm). Unfortunately, the algorithm will often run forever by producing infinitely many test cases. Consider, e.g., the following simple program:

$$(\ell_1)\ nat(0). \qquad\qquad (\ell_2)\ nat(s(X)) \leftarrow nat(X).$$

Even if every goal $nat(t)$ with t ground is terminating, our algorithm will still produce infinitely many test cases, e.g., $nat(0)$, $nat(s(0))$, $nat(s(s(0)))$, ..., since each goal will explore a different path (i.e., will produce a different execution trace: $(\{\ell_1\})$, $(\{\ell_2\},\{\ell_1\})$, $(\{\ell_2\},\{\ell_2\},\{\ell_1\})$,...). In practice, though, the quality of the generated test cases should be experimentally evaluated using a coverage tool.

Therefore, in general, we will sacrifice completeness in order to guarantee the termination of concolic testing. For this purpose, one can fix a maximum term depth for the arguments of the generated test cases. As it is common, the depth $\mathit{depth}(t)$ of a term t is defined as follows: $\mathit{depth}(t) = 0$ if t is a variable or a constant symbol, and $\mathit{depth}(f(t_1,\ldots,t_n)) = 1 + \mathsf{max}(\mathit{depth}(t_1),\ldots,\mathit{depth}(t_n))$, otherwise. Then, we can replace the use of function alt in step 2.d of Definition 5 by a function alt_k such that $\mathsf{alt}_k(A,\mathcal{L},\mathcal{L}',G) = \mathsf{alt}(A,\mathcal{L},\mathcal{L}',G) = \theta$ if $\mathit{depth}(t) \leqslant k$ for all $X/t \in \theta$, and $\mathsf{alt}_k(A,\mathcal{L},\mathcal{L}',G) = \mathsf{fail}$ otherwise.

For instance, by requiring a maximum term depth of 1, the generated test cases for the program nat above would be $nat(0)$, $nat(1)$, $nat(s(0))$ and $nat(s(1))$, where 1 is a fresh constant symbol, with associated traces $(\{\ell_1\})$, $(\{\,\})$, $(\{\ell_2\},\{\ell_1\})$, and $(\{\ell_2\},\{\,\})$, respectively. This is the solution we implemented in the concolic testing tool described in Section 4.3.

4.2 Solving Unifiability Problems

In this section, we present a constructive algorithm for function alt. We first define our unifiability problems and then we propose an algorithmic solution in two steps.

Definition 6 (*unifiability problem*)
Let A be an atom and $\mathcal{H}_{pos}$, $\mathcal{H}_{neg}$ be two sets of atoms, the elements of which are variable disjoint with A and unify with A, and a set of variables G. The problem consists in finding a substitution σ such that $A\sigma \approx H^+$ for each $H^+ \in \mathcal{H}_{pos}$, $\neg(A\sigma \approx H^-)$ for each $H^- \in \mathcal{H}_{neg}$, and $G\sigma$ is ground.

4.2.1 The Positive Atoms

First, we try to find a substitution σ such that $A\sigma \approx H^+$ for each $H^+ \in \mathcal{H}_{pos}$. Note that $\sigma := id$ works, but we may find a more precise substitution with the following algorithm, which will become useful when dealing with the negative atoms. We use a special set $\mathcal{U}$ of variables, the elements of which will replace *disagreement pairs* (see (Apt 1997) p. 27). The elements of $\mathcal{U}$ are denoted by U, U', $U_1\ldots$ Roughly speaking, given terms s and t, a subterm s' of s and a subterm t' of t form a disagreement pair if the root symbols of s' and t' are different, but the symbols from s' up to the root of s and from t' up to the root of t are the same. For instance, $X, g(a)$ and $b, h(Y)$ are disagreement pairs of the terms $f(X, g(b))$ and $f(g(a), g(h(Y)))$. A disagreement pair t, t' is called *simple* if one of the terms is a variable that does not occur in the other term and no variable of $\mathcal{U}$ occurs in t, t'. We say that the substitution $\{X/s\}$ is determined by t, t' if $\{X, s\} = \{t, t'\}$.

Basically, given a set of atoms $\mathcal{A}$, the following algorithm computes an atom B such that $B\theta$ still unifies with all the atoms in $\mathcal{A}$ as long as θ does not bind variables from $\mathcal{U}$.

Definition 7 (*Pos*)
Input: a non-empty set $\mathcal{A}$ of atoms with the same predicate symbol.
Output: an atom B.

```
1. Let $\mathcal{B} := \mathcal{A}$.
2. While simple disagreement pairs occur in $\mathcal{B}$ do
       (a) nondeterministically choose a simple disagreement pair $t, t'$ in $\mathcal{B}$;
       (b) set $\mathcal{B}$ to $\mathcal{B}\eta$ where $\eta$ is a substitution determined by $t, t'$.
3. While $|\mathcal{B}| \neq 1$ do
       (a) nondeterministically choose a disagreement pair $t, t'$ in $\mathcal{B}$;
       (b) replace $t, t'$ in $\mathcal{B}$ by a fresh variable of $\mathcal{U}$.
4. Return $B$ in $\mathcal{B}$.
```

The correctness of this algorithm is stated as follows. In the following, for any atom B and substitution η, we let $\eta|B$ denote the substitution obtained from η by restricting its domain to $\mathcal{V}ar(B)$.

Theorem 2
Let $\mathcal{A}$ be a non-empty set of atoms with the same predicate symbol. The algorithm in Definition 7 with input $\mathcal{A}$ always terminates and returns an atom B such that $B\eta$ unifies with all the atoms of $\mathcal{A}$ for any substitution η with $Dom(\eta|B) \cap \mathcal{U} = \{\}$ and $Ran(\eta|B) = \{\}$.

Example 3
Let $\mathcal{A} := \{p(a, X), p(b, Y), p(Z, Z)\}$. First the algorithm sets $\mathcal{B} := \mathcal{A}$, then it considers the simple disagreement pairs in $\mathcal{B}$. The substitution $\eta_1 := \{Y/X\}$ is determined by X, Y. Action (2b) sets $\mathcal{B}$ to $\mathcal{B}\eta_1 = \{p(a, X), p(b, X), p(Z, Z)\}$. The substitution $\eta_2 := \{Z/X\}$ is determined by X, Z. Action (2b) sets $\mathcal{B}$ to $\mathcal{B}\eta_2 = \{p(a, X), p(b, X), p(X, X)\}$. The substitution $\eta_3 := \{X/a\}$ is determined by a, X. Action (2b) sets $\mathcal{B}$ to $\mathcal{B}\eta_3 = \{p(a, a), p(b, a)\}$. Now, no simple disagreement pair occurs in $\mathcal{B}$, hence the algorithm jumps to the loop at line 3. Action (3b) replaces the disagreement pair a, b with a fresh variable $U \in \mathcal{U}$, hence $\mathcal{B}$ is set to $\{p(U, a)\}$. As $|\mathcal{B}| = 1$ the loop at line 3 stops and the algorithm returns $p(U, a)$.

4.2.2 The Negative Atoms

Now we deal with the negative atoms by means of the following algorithm which is the basis of our implementation of function alt:

Definition 8 (*PosNeg*)
Input: an atom A and two sets of atoms $\mathcal{H}_{pos}$, $\mathcal{H}_{neg}$, the elements of which are variable disjoint with A and unify with A, and a set of variables G.
Output: fail or a substitution σ.

1. Let B be the atom returned by the algorithm of Definition 7 with input $\mathcal{H}_{pos} \cup \{A'\}$ where A' is a copy of A with fresh variables.
2. Let $\theta := \mathsf{mgu}(A, B)$.
3. Let η be a substitution such that $G\theta\eta$ is ground.
4. Let $\sigma := \theta\eta$.
5. Check that $Dom(\sigma|B) \cap \mathcal{U} = \{\}$ and $Ran(\sigma|B) = \{\}$ otherwise return fail.
6. Check that for each $H^- \in \mathcal{H}_{neg}$, $\neg(A\sigma \approx H^-)$, otherwise return fail.
7. Return $\sigma|A$.

The correctness of this algorithm is stated as follows:

Theorem 3
Given an atom A and two sets of atoms $\mathcal{H}_{pos}$, $\mathcal{H}_{neg}$, the elements of which are variable disjoint with A and unify with A, and a set of variables G, the algorithm in Definition 8 terminates. Moreover, if it returns a substitution σ then $\bigwedge_{H \in \mathcal{H}_{pos}} A\sigma \approx H \wedge \bigwedge_{H' \in \mathcal{H}_{neg}} \neg(A\sigma \approx H')$ holds and $G\sigma$ is ground.

Example 4
Let $A := p(X)$, $\mathcal{H}_{pos} := \{p(s(Y))\}$, $\mathcal{H}_{neg} := \{p(s(0))\}$, and $G := \{X\}$. The algorithm of Section 4.2.1 applied to $\mathcal{H}_{pos} \cup \{p(X')\}$ returns $B := p(s(Y))$. We have $\theta = \{X/s(Y)\}$ and $G\theta = \{s(Y)\}$. We take $\eta = \{Y/s(0)\}$, then $\theta\eta = \{X/s(s(0)), Y/s(0)\}$, $\theta\eta|B = \{Y/s(0)\}$, and $Ran(\theta\eta|B) = \{\}$. Let $\sigma = \{X/s(s(0)), Y/s(0)\}$, $Dom(\sigma|B) = \{Y\}$ does not intersect with $\mathcal{U}$. Finally, we check that $p(s(s(0)))$ does not unify with $p(s(0))$. The algorithm returns $\{X/s(s(0))\}$.

Table 1. *Clause coverage analysis results (SICStus Prolog)*

paper	100%	*paper2*	100%	*nat*	100%	*advisor*	91%	*applast*	100%
depth	88%	*regexp*	86%	*relative*	100%	*rotateprune*	100%	*transpose*	100%
mult	100%	*hanoi*	100%	*automaton*	100%	*qsort*	80%	*inclist*	100%
doubleflip	100%	*recacctype*	100%	*ackermann*	100%	*fibonacci*	100%	*preorder*	100%

Example 5

Let $A := p(X)$, $\mathcal{H}_{pos} := \{p(a), p(b)\}$, $\mathcal{H}_{neg} := \{p(f(Z))\}$, and $G := \{\}$. The algorithm of Section 4.2.1 applied to $\mathcal{H}_{pos} \cup \{p(X')\}$ returns $B := p(U)$. However, we cannot find σ such that $B\sigma$ does not unify with $p(f(Z))$ without binding U to a non-variable term. The algorithm thus returns fail.

4.3 A Tool for Concolic Testing

In this section, we present a prototype implementation of the concolic testing scheme. The tool, called contest, is publicly available from the following URL

```
http://kaz.dsic.upv.es/contest.html
```

It consists of approx. 1000 lines of Prolog code and implements the concolic testing algorithm of Definition 5 with function alt as described in Section 4.2 and a maximum term depth that can be fixed by the user in order to guarantee the termination of the process. For instance, for the program of Example 2, with initial goal $p(s(a))$, and assuming its argument is an input argument, i.e., $\mathsf{input}(p(s(a))) = s(a)$, and a maximum term depth of 2, we get the following test cases: $p(s(a))$, $p(c)$, $p(s(c))$, $p(s(b))$, $p(f(c))$, $p(f(b))$, and $p(f(a))$, thus achieving full choice coverage.

Table 1 shows a summary of the coverage achieved by the test cases automatically generated using contest. The complete benchmarks –including the source code, initial goal, input arguments, maximum term depth, and a timeout– can be found in the above URL. We used the coverage analysis tool of SICStus Prolog 4.3.1, which basically measures the number of times each clause is used. The results are very satisfactory, achieving a full coverage in most of the examples.

The current version is a proof-of-concept implementation and only deals with pure Prolog without negation. We plan to extend it to cope with full Prolog. The concrete semantics can be extended following (Ströder et al. 2011), and concolic execution is in general a natural extension of the semantics in Figure 2. Dealing with negation or the cut is simple since one just follows the rules of the concrete semantics. In other cases, e.g., for relational built-in's or equalities, we should label the execution step with an associated constraint, which can then be used to produce alternative execution paths by solving its negation. In this context, our tool will be useful not only for test case generation, but also to detect program errors during concolic testing (e.g., negated atoms which are not instantiated enough, incorrect calls to arithmetic built-in's, etc). See Appendix A for more details on extending concolic execution to full Prolog.

5 Related Work and Concluding Remarks

Mera et al. (2009) present a framework unifying unit testing and run-time verification for the Ciao system (Hermenegildo et al. 2012). The ECLiPSe constraint programming system (Schimpf and Shen 2012) and SICStus Prolog (Carlsson and Mildner 2012) both provide tools which run a given goal and compute how often program points in the code were executed. SWI-Prolog (Wielemaker et al. 2012) offers a unit testing tool associated to an optional interactive generation of test cases. It also includes an experimental coverage analysis which runs a given goal and computes the percentage of the used clauses and failing clauses. Belli and Jack (1993) and Degrave et al. (2008) consider automatic generation of test inputs for strongly typed and moded logic programming languages like the Mercury programming language (Somogyi et al. 1996), whereas we only require moding the top-level predicate of the program.

One of the closest approaches to our work is the test case generation technique by (Albert et al. 2014). The main difference, though, is that their technique is based solely on traditional symbolic execution. As mentioned before, concolic testing may scale better since one can deal with more complex constraints by using data from the concrete component of the concolic state. Another difference is that we aim at full path coverage (i.e., choice coverage), and not only a form of statement coverage.

Another close approach is (Vidal 2015), where a concolic execution semantics for logic programs is presented. However, this approach only considers a simpler statement coverage and, thus, it can be seen as a particular instance of the technique in the present paper. Another significant difference is that, in (Vidal 2015), concolic execution proceeds in a stepwise manner: first, concrete execution produces an execution *trace*, which is then used to drive concolic execution. Although this scheme is conceptually simpler, it may give rise to poorer results in practice since one cannot use concrete values in symbolic executions, one of the main advantages of concolic execution over traditional symbolic execution. Moreover, Vidal (2015) presents no formal results nor an implementation of the concolic execution technique.

Summarizing the paper, we have introduced a novel scheme for concolic testing in logic programming. It offers a sound and fully automatic technique for detecting run time errors (without false positives) and test case generation with a good code coverage. We have proved a number of properties for concolic testing, including the particular type of unification problems that should be solved to produce new test cases. Furthermore, we have developed a publicly available proof-of-concept implementation of the concolic testing scheme, contest, that shows the usefulness of our approach. To the best of our knowledge, this is the first fully automatic testing tool for Prolog that aims at full path coverage (here called choice coverage).

As future work, we plan to extend the scheme to full Prolog (see the remarks in Section 4.3). Another interesting subject for further research is the definition of appropriate heuristics to drive concolic testing w.r.t. a given coverage criterion. This might have a significant impact on the quality of the test cases when the process is incomplete. Finally, from the experimental evaluation, we observed that the results could be improved by introducing type information, so that the generated values are restricted to the right type. Improving concolic testing with type annotations is also a promising line of future work.

References

Albert, E., Arenas, P., Gómez-Zamalloa, M., and Rojas, J. 2014. Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-Based Instance, and Actor-Based Concurrency. In *SFM 2014*. Springer LNCS 8483, 263–309.

Apt, K. 1997. *From Logic Programming to Prolog.* Prentice Hall.

Belli, F. and Jack, O. 1993. Implementation-based analysis and testing of Prolog programs. In *ISSTA*. 70–80.

Carlsson, M. and Mildner, P. 2012. SICStus Prolog - the first 25 years. *Theory and Practice of Logic Programming 12,* 1-2, 35–66.

Claessen, K. and Hughes, J. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. of (ICFP 2000)*. ACM, 268–279.

Clarke, L. 1976. A program testing system. In *Proceedings of the 1976 Annual Conference (ACM'76)*. 488–491.

Degrave, F., Schrijvers, T., and Vanhoof, W. 2008. Automatic generation of test inputs for Mercury. In *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008*. 71–86.

Godefroid, P., Klarlund, N., and Sen, K. 2005. DART: directed automated random testing. In *Proc. of PLDI'05*. ACM, 213–223.

Godefroid, P., Levin, M., and Molnar, D. 2012. Sage: whitebox fuzzing for security testing. *Commun. ACM 55,* 3, 40–44.

Gómez-Zamalloa, M., Albert, E., and Puebla, G. 2010. Test case generation for object-oriented imperative languages in CLP. *TPLP 10,* 4-6, 659–674.

Hermenegildo, M. V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J. F., and Puebla, G. 2012. An overview of Ciao and its design philosophy. *TPLP 12,* 1-2, 219–252.

King, J. C. 1976. Symbolic execution and program testing. *Commun. ACM 19,* 7, 385–394.

Lloyd, J. 1987. *Foundations of Logic Programming.* Springer-Verlag, Berlin. Second edition.

Mera, E., López-García, P., and Hermenegildo, M. V. 2009. Integrating software testing and run-time checking in an assertion verification framework. In *25th International Conference on Logic Programming, ICLP 2009, Pasadena*. 281–295.

Pasareanu, C. and Rungta, N. 2010. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 179–180.

Schimpf, J. and Shen, K. 2012. Ecl$^{\mathrm{i}}$ps$^{\mathrm{e}}$ - from LP to CLP. *Theory and Practice of Logic Programming 12,* 1-2, 127–156.

Sen, K., Marinov, D., and Agha, G. 2005. CUTE: a concolic unit testing engine for C. In *Proc. of ESEC/SIGSOFT FSE 2005*. ACM, 263–272.

Somogyi, Z., Henderson, F., and Conway, T. 1996. The execution algorithm of Mercury, an efficient purely declarative Logic Programming language. *The Journal of Logic Programming 29,* 1–3, 17–64.

Ströder, T., Emmes, F., Schneider-Kamp, P., Giesl, J., and Fuhs, C. 2011. A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog. In *LOPSTR'11*. Springer LNCS 7225, 237–252.

Vasak, T. and Potter, J. 1986. Characterization of terminating logic programs. In *Proc. of the 1986 Intl. Symp. on Logic Programming.* IEEE, 140–147.

Vidal, G. 2015. Concolic Execution and Test Case Generation in Prolog. In *Proc. of the 24th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'14)*, M. Proietti and H. Seki, Eds. Springer LNCS 8981, 167–181.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12,* 1-2, 67–96.

In this appendix we report, for the sake of completeness, some auxiliary contents that, for space limitations, we could not include in the paper. Should this paper be accepted, this appendix will not be included in the final version but in an accompanying technical report.

Appendix A Towards Extending Concolic Testing to Full Prolog

In this section, we show a summary of our preliminary research on extending concolic execution to deal with full Prolog. First, we consider the extension of the concrete semantics. Here, we mostly follow the linear semantics of (Ströder et al. 2011), being the main differences that we consider built-ins explicitly, we excluded dynamic predicates for simplicity —but could be added along the lines of (Ströder et al. 2011)— and that, analogously to what we did in Section 2, only the first answer for the initial goal is considered.

In the following, we let the Boolean function defined return true when its argument is an atom rooted by a defined predicate symbol, and false otherwise (i.e., a built-in). Moreover, for evaluating relational and arithmetic expressions, we assume a function eval such that, given an expression e, $\mathsf{eval}(e)$ either returns the evalua-

$$\text{(success)}\ \frac{}{\langle \mathsf{true}_\delta \,|\, S\rangle \rightarrow \langle \textsc{success}_\delta\rangle}$$

$$\text{(failure)}\ \frac{}{\langle (\mathsf{fail},\mathcal{B})_\delta\rangle \rightarrow \langle \textsc{fail}_\delta\rangle} \qquad \text{(backtrack)}\ \frac{S \neq \epsilon}{\langle (\mathsf{fail},\mathcal{B}) \,|\, S\rangle \rightarrow \langle S\rangle}$$

$$\text{(choice)}\ \frac{\mathsf{defined}(A) \wedge \mathsf{clauses}(A,\mathcal{P}) = (c_1,\ldots,c_n) \wedge n > 0 \wedge m \text{ is fresh}}{\langle (A,\mathcal{B})_\delta \,|\, S\rangle \rightarrow \langle (A,\mathcal{B})_\delta^{c_1[!/!^m]} \,|\ldots|\, (A,\mathcal{B})_\delta^{c_n[!/!^m]} \,|\ ?_\delta^m \,|\, S\rangle}$$

$$\text{(choice\_fail)}\ \frac{\mathsf{defined}(A,\mathcal{P}) \wedge \mathsf{clauses}(A,\mathcal{P}) = \{\}}{\langle (A,\mathcal{B})_\delta \,|\, S\rangle \rightarrow \langle \mathsf{fail}_\delta \,|\, S\rangle}$$

$$\text{(unfold)}\ \frac{\mathsf{mgu}(A,H_1) = \sigma}{\langle (A,\mathcal{B})_\delta^{H_1 \leftarrow \mathcal{B}_1} \,|\, S\rangle \rightarrow \langle (\mathcal{B}_1\sigma, \mathcal{B}\sigma)_{\delta\sigma} \,|\, S\rangle}$$

$$\text{(cut)}\ \frac{}{\langle (!^m,\mathcal{B})_\delta \,|\, S' \,|\ ?_{\delta'}^m \,|\, S\rangle \rightarrow \langle \mathcal{B}_\delta \,|\ ?_{\delta'}^m \,|\, S\rangle} \qquad \text{(cut\_fail)}\ \frac{}{\langle ?_\delta^m \,|\, S\rangle \rightarrow \langle \mathsf{fail}_\delta \,|\, S\rangle}$$

$$\text{(call)}\ \frac{A \notin \mathcal{V} \wedge m \text{ is fresh}}{\langle (\mathit{call}(A),\mathcal{B})_\delta \,|\, S\rangle \rightarrow \langle (A[\mathcal{V}/\mathsf{call}(\mathcal{V}), !/!^m],\mathcal{B})_\delta \,|\ ?_\delta^m \,|\, S\rangle}$$

$$\text{(call\_error)}\ \frac{A \in \mathcal{V}}{\langle (\mathit{call}(A),\mathcal{B})_\delta \,|\, S\rangle \rightarrow \langle \textsc{error}_\delta\rangle}$$

$$\text{(not)}\ \frac{m \text{ is fresh}}{\langle (\backslash{+}(A),\mathcal{B})_\delta \,|\, S\rangle \rightarrow \langle (\mathsf{call}(A), !^m, \mathsf{fail})_\delta \,|\, \mathcal{B}_\delta \,|\ ?_\delta^m \,|\, S\rangle}$$

$$\text{(unify)}\ \frac{\mathsf{mgu}(t_1,t_2) = \sigma \neq \mathsf{fail}}{\langle (t_1 = t_2,\mathcal{B})_\delta \,|\, S\rangle \rightarrow \langle \mathcal{B}\sigma_{\delta\sigma} \,|\, S\rangle} \qquad \text{(unify\_fail)}\ \frac{\mathsf{mgu}(t_1,t_2) = \mathsf{fail}}{\langle (t_1 = t_2,\mathcal{B})_\delta \,|\, S\rangle \rightarrow \langle \mathsf{fail}_\delta \,|\, S\rangle}$$

$$\text{(is)}\ \frac{\mathsf{eval}(e_2) = t_2 \neq \mathsf{error}}{\langle (t_1 \ \mathsf{is}\ e_2,\mathcal{B})_\delta \,|\, S\rangle \rightarrow \langle (t_1 = t_2,\mathcal{B})_\delta \,|\ S\rangle} \qquad \text{(is\_error)}\ \frac{\mathsf{eval}(e_2) = \mathsf{error}}{\langle (t_1 \ \mathsf{is}\ e_2,\mathcal{B})_\delta \,|\, S\rangle \rightarrow \langle \textsc{error}_\delta\rangle}$$

$$\text{(rel)}\ \frac{\mathsf{eval}(t_1 \oplus t_2) = A \in \{\mathsf{true},\mathsf{fail}\}}{\langle (t_1 \oplus t_2,\mathcal{B})_\delta \,|\, S\rangle \rightarrow \langle (A,\mathcal{B})_\delta \,|\ S\rangle} \qquad \text{(rel\_error)}\ \frac{\mathsf{eval}(t_1 \oplus t_2) = \mathsf{error}}{\langle (t_1 \oplus t_2,\mathcal{B})_\delta \,|\, S\rangle \rightarrow \langle \textsc{error}_\delta\rangle}$$

Fig. A 1. Extended concrete semantics

tion of e (typically a number or a Boolean value) or the special constant error when the expression is not instantiated enough to be evaluated. E.g., $\mathsf{eval}(2+2) = 4$, $\mathsf{eval}(3 > 1) = \mathit{true}$, but $\mathsf{eval}(X > 0) = \mathsf{error}$.

The transitions rules are shown in Figure A 1. In the following, we briefly explain the novelties w.r.t. the rules of Section 2:

- In rule choice we use the notation $c[!/!^m]$ to denote a copy of clause c where the occurrences of (possibly labeled) cuts ! at predicate positions (e.g., not inside a call), if any, are replaced by a *labeled* cut $!^m$, where m is a fresh label. Also, in the derived state, we add a *scope delimiter* $?^m$.
- Rule cut removes some alternatives from the current state, while rule $\mathsf{cut\_fail}$ applies when a goal reaches the scope delimiter without success.
- The rules for call and negation should be clear. Let us only mention that the notation $A[\mathcal{V}/\mathsf{call}(\mathcal{V}), !/!^m]$ denotes the atom A in which all variables X on predicate positions are replaced by $\mathsf{call}(X)$ and all (possibly labeled) cuts on predicate positions are replaced by $!^m$.
- Calls to the built-in predicate is are dealt with rules is and $\mathsf{is\_error}$ by means of the auxiliary function eval. Rules rel and $\mathsf{rel\_error}$ proceed analogously with relational operators like $>$, $<$, $==$, etc.

Regarding the concolic execution semantics, we follow a similar approach to that of Section 3. The labeled transition rules can be seen in Figure A 2. Now, we consider six kinds of labels for $\rightsquigarrow$:

- The labels $\diamond$ and $c(\mathcal{L}_1, \mathcal{L}_2)$ with the same meaning as in the concolic semantics of Section 3.
- The label $u(t_1, t_2)$, which is used to denote a unification step, i.e., the step implies that t_1 and t_2 should unify.
- In contrast, the label $d(t_1, t_2)$ denotes a disunification, i.e., the step implies that t_1 and t_2 should not unify.
- The label $is(X, t)$ denotes a step where is is evaluated (see below).
- Finally, the label $r(A', A)$ denotes that the relational expression A' should be equal to $A \in \{\mathsf{true}, \mathsf{fail}\}$.

In particular, in rules unify and $\mathsf{unify\_fail}$, the labels store the unification that must hold in the step. Note that the fact that $\mathsf{mgu}(t_1, t_2) = \mathsf{fail}$ does not imply $\mathsf{mgu}(t'_1, t'_2) = \mathsf{fail}$ since t'_1 and t'_2 might be less instantiated than t_1 and t_2.

In rule is, we label the step with $is(X, t'_2)$ which means that the fresh variable X should be bound to the evaluation of t'_2 after grounding it. Note that introducing such a fresh variable is required to avoid a failure in the subsequent step with rule unify because of, e.g., a non-ground arithmetic expression that could not be evaluated yet to a value using function $\mathsf{sym\_eval}$. Note that rule $\mathsf{is\_error}$ does not include any label since we assume that an error in the concrete computation just aborts the execution and also the test case generation process.

Finally, in rule rel we label the step with $r(A', A)$ where A is the value $\mathsf{true}/\mathsf{fail}$ of the relational expression in the concrete goal, and A' is a (possibly nonground) corresponding expression in the symbolic goal. Here, we use the auxiliary function $\mathsf{sym\_eval}$ to simplify the relational expression as much as possible. E.g., $\mathsf{sym\_eval}(3 > 0) = \mathsf{true}$ but $\mathsf{sym\_eval}(3 + 2 > X) = 5 > X$.

These labels can be used for extending the concolic testing algorithm of Section 4. For instance, given a concolic execution step labeled with $r(X > 0, \mathsf{true})$, we have that solving $\neg(X > 0)$ will produce a binding for X (e.g., $\{X/0\}$) that will follow an alternative path. Here, the concolic testing procedure will integrate a constraint solver for producing solutions to negated constraints. We find this extension of the concolic testing procedure an interesting topic for future work.

$$(\mathsf{success})\ \frac{}{\langle \mathsf{true}_\delta \mid S \rrbracket\, \mathsf{true}_\theta \mid S' \rangle \leadsto_\diamond \langle \textsc{success}_\delta \rrbracket\, \textsc{success}_\theta \rangle}$$

$$(\mathsf{failure})\ \frac{}{\langle (\mathsf{fail},\mathcal{B})_\delta \rrbracket\, (\mathsf{fail},\mathcal{B}')_\theta \rangle \leadsto_\diamond \langle \mathsf{fail}_\delta \rrbracket\, \mathsf{fail}_\theta \rangle}$$

$$(\mathsf{backtrack})\ \frac{S \neq \epsilon}{\langle (\mathsf{fail},\mathcal{B}) \mid S \rrbracket\, (\mathsf{fail},\mathcal{B}') \mid S' \rangle \leadsto_\diamond \langle S \rrbracket\, S' \rangle}$$

$$(\mathsf{choice})\ \frac{\mathsf{defined}(A) \wedge \mathsf{clauses}(A,\mathcal{P}) = \overline{c_n} \wedge n > 0 \wedge m \text{ is fresh} \wedge \mathsf{clauses}(A',\mathcal{P}) = \overline{d_k}}{\begin{array}{l}\langle (A,\mathcal{B})_\delta \mid S \rrbracket\, (A',\mathcal{B}')_\theta \mid S' \rangle \\ \quad \leadsto_{c(\ell(\overline{c_n}),\ell(\overline{d_k}))} \langle (A,\mathcal{B})_\delta^{c_1[!/!^m]} \mid \ldots \mid (A,\mathcal{B})_\delta^{c_n[!/!^m]} \mid\ ?^m_\delta \mid S \\ \qquad \rrbracket\, (A',\mathcal{B}')_\theta^{c_1[!/!^m]} \mid \ldots \mid (A',\mathcal{B}')_\theta^{c_n[!/!^m]} \mid\ ?^m_\theta \mid S' \rangle\end{array}}$$

$$(\mathsf{choice\_fail})\ \frac{\mathsf{defined}(A,\mathcal{P}) \wedge \mathsf{clauses}(A,\mathcal{P}) = \{\} \wedge \mathsf{clauses}(A',\mathcal{P}) = \overline{c_k}}{\langle (A,\mathcal{B})_\delta \mid S \rrbracket\, (A',\mathcal{B}')_\theta \mid S' \rangle \leadsto_{c(\{\},\ell(\overline{c_k}))} \langle \mathsf{fail}_\delta \mid S \rrbracket\, \mathsf{fail}_\theta \mid S' \rangle}$$

$$(\mathsf{unfold})\ \frac{\mathsf{mgu}(A,H_1) = \sigma \wedge \mathsf{mgu}(A',H_1) = \sigma'}{\langle (A,\mathcal{B})_\delta^{H_1 \leftarrow \mathcal{B}_1} \mid S \rrbracket\, (A',\mathcal{B}')_\theta^{H_1 \leftarrow \mathcal{B}_1} \mid S' \rangle \leadsto_\diamond \langle (\mathcal{B}_1\sigma,\mathcal{B}\sigma)_{\delta\sigma} \mid S \rrbracket\, (\mathcal{B}_1\sigma',\mathcal{B}'\sigma')_{\theta\sigma'} \mid S' \rangle}$$

$$(\mathsf{cut})\ \frac{}{\langle (!^m,\mathcal{B})_\delta \mid S_1 \mid\ ?^m_{\delta'} \mid S \rrbracket\, (!^m,\mathcal{B}')_\theta \mid S_1' \mid\ ?^m_{\theta'} \mid S' \rangle \leadsto_\diamond \langle \mathcal{B}_\delta \mid\ ?^m_{\delta'} \mid S \rrbracket\, \mathcal{B}'_\theta \mid\ ?^m_{\theta'} \mid S' \rangle}$$

$$(\mathsf{cut\_fail})\ \frac{}{\langle ?^m_\delta \mid S \rrbracket\, ?^m_\theta \mid S' \rangle \leadsto_\diamond \langle \mathsf{fail}_\delta \mid S \rrbracket\, \mathsf{fail}_\theta \mid S' \rangle}$$

$$(\mathsf{call})\ \frac{A \notin \mathcal{V} \wedge m \text{ is fresh}}{\begin{array}{l}\langle (\mathit{call}(A),\mathcal{B})_\delta \mid S \rrbracket\, (\mathit{call}(A'),\mathcal{B}')_\theta \mid S' \rangle \\ \quad \leadsto_\diamond \langle (A[\mathcal{V}/\mathsf{call}(\mathcal{V}),!/!^m],\mathcal{B})_\delta \mid\ ?^m_\delta \mid S \rrbracket\, (A'[\mathcal{V}/\mathsf{call}(\mathcal{V}),!/!^m],\mathcal{B}')_\theta \mid\ ?^m_\theta \mid S' \rangle\end{array}}$$

$$(\mathsf{call\_error})\ \frac{A \in \mathcal{V}}{\langle (\mathit{call}(A),\mathcal{B})_\delta \mid S \rrbracket\, (\mathit{call}(A'),\mathcal{B}')_\theta \mid S' \rangle \leadsto_\diamond \langle \mathsf{error}_\delta \rrbracket\, \mathsf{error}_\theta \rangle}$$

$$(\mathsf{not})\ \frac{m \text{ is fresh}}{\begin{array}{l}\langle (\backslash\!+(A),\mathcal{B})_\delta \mid S \rrbracket\, (\backslash\!+(A'),\mathcal{B}')_\theta \mid S' \rangle \\ \quad \leadsto_\diamond \langle (\mathsf{call}(A),!^m,\mathsf{fail})_\delta \mid \mathcal{B}_\delta \mid\ ?^m_\delta \mid S \rrbracket\, (\mathsf{call}(A'),!^m,\mathsf{fail})_\theta \mid \mathcal{B}'_\theta \mid\ ?^m_\theta \mid S' \rangle\end{array}}$$

$$(\mathsf{unify})\ \frac{\mathsf{mgu}(t_1,t_2) = \sigma \wedge \mathsf{mgu}(t_1',t_2') = \sigma'}{\langle (t_1 = t_2,\mathcal{B})_\delta \mid S \rrbracket\, (t_1' = t_2',\mathcal{B}')_\theta \mid S' \rangle \leadsto_{u(t_1',t_2')} \langle \mathcal{B}\sigma_{\delta\sigma} \mid S \rrbracket\, \mathcal{B}'\sigma'_{\delta\sigma'} \mid S' \rangle}$$

$$(\mathsf{unify\_fail})\ \frac{\mathsf{mgu}(t_1,t_2) = \mathsf{fail}}{\langle (t_1 = t_2,\mathcal{B})_\delta \mid S \rrbracket\, (t_1' = t_2',\mathcal{B}')_\theta \mid S' \rangle \leadsto_{d(t_1',t_2')} \langle \mathsf{fail}_\delta \mid S \rrbracket\, \mathsf{fail}_\theta \mid S' \rangle}$$

$$(\mathsf{is})\ \frac{\mathsf{eval}(e_2) = t_2 \neq \mathsf{error} \wedge \mathsf{sym\_eval}(e_2') = t_2' \wedge X \text{ is fresh}}{\langle (t_1 \ \mathsf{is}\ e_2,\mathcal{B})_\delta \mid S \rrbracket\, (t_1' \ \mathsf{is}\ e_2',\mathcal{B}')_\theta \mid S' \rangle \leadsto_{is(X,t_2')} \langle (t_1 = t_2,\mathcal{B})_\delta \mid S \rrbracket\, (t_1' = X,\mathcal{B}')_\theta \mid S' \rangle}$$

$$(\mathsf{is\_error})\ \frac{\mathsf{eval}(e_2) = \mathsf{error}}{\langle (t_1 \ \mathsf{is}\ e_2,\mathcal{B})_\delta \mid S \rrbracket\, (t_1' \ \mathsf{is}\ e_2',\mathcal{B}')_\theta \mid S' \rangle \leadsto_\diamond \langle \mathsf{error}_\delta \rrbracket\, \mathsf{error}_\theta \rangle}$$

$$(\mathsf{rel})\ \frac{\mathsf{eval}(t_1 \oplus t_2) = A \in \{\mathsf{true},\mathsf{fail}\} \wedge \mathsf{sym\_eval}(t_1' \oplus t_2') = A'}{\langle (t_1 \oplus t_2,\mathcal{B})_\delta \mid S \rrbracket\, (t_1' \oplus t_2',\mathcal{B}')_\theta \mid S' \rangle \leadsto_{r(A',A)} \langle (A,\mathcal{B})_\delta \mid S \rrbracket\, (A',\mathcal{B}')_\theta \mid S' \rangle}$$

$$(\mathsf{rel\_error})\ \frac{\mathsf{eval}(t_1 \oplus t_2) = \mathsf{error}}{\langle (t_1 \oplus t_2,\mathcal{B})_\delta \mid S \rrbracket\, (t_1' \oplus t_2',\mathcal{B}')_\theta \mid S' \rangle \leadsto_\diamond \langle \mathsf{error}_\delta \rrbracket\, \mathsf{error}_\theta \rangle}$$

Fig. A 2. Extended concolic execution semantics

Appendix B Proofs of Technical Results

B.1 Concolic Execution Semantics

Proof of Theorem 1
Since the base case $i = 0$ trivially holds, in the following we only consider the inductive case $i > 0$. Let $C_i = \langle \mathcal{B}^c_\delta \mid S \rrbracket \mathcal{D}^{c'}_\theta \mid S' \rangle$. By the inductive hypothesis, we have $|S| = |S'|$, $\mathcal{D} \leqslant \mathcal{B}$, $c = c'$ (if any), and $p(\overline{X_n})\theta \leqslant p(\overline{t_n})\delta$. Now, we consider the step $C_i \rightsquigarrow C_{i+1}$ and distinguish the following cases, depending on the applied rule:

- If the rule applied is $\mathsf{success}$, $\mathsf{failure}$, $\mathsf{backtrack}$ or $\mathsf{choice\_fail}$, the claim follows trivially by induction.
- If the rule applied is choice, let us assume that we have $\mathcal{B} = (A, \mathcal{B}')$, $\mathcal{D} = (A', \mathcal{D}')$ and $\mathsf{clauses}(A, \mathcal{P}) = \overline{c_j}$, $j > 0$. Therefore, we have $C_{i+1} = \langle \mathcal{B}^{c_1}_\delta \mid \ldots \mid \mathcal{B}^{c_j}_\delta \mid S \rrbracket \mathcal{D}^{c_1}_\theta \mid \ldots \mid \mathcal{D}^{c_j}_\theta \mid S' \rangle$, and the claim follows straightforwardly by the induction hypothesis.
- Finally, if the applied rule is unfold, then we have that $\mathcal{B}^c_\delta = (A, \mathcal{B}')^c_\delta$, $\mathcal{D}^c_\theta = (A', \mathcal{D}')^c_\theta$ for some clause $c = H_1 \leftarrow \mathcal{B}_1$. Therefore, we have $C_{i+1} = \langle (\mathcal{B}_1\sigma, \mathcal{B}'\sigma)_{\delta\sigma} \mid S \rrbracket (\mathcal{B}_1\sigma', \mathcal{D}'\sigma')_{\theta\sigma'} \mid S' \rangle$, where $\mathsf{mgu}(A, H_1) = \sigma$ and $\mathsf{mgu}(A', H_1) = \sigma'$. First, $c = c'$ holds by vacuity since the goals are not labeled with a clause. Also, the number of concrete and symbolic goals is trivially the same since $|S| = |S'|$ by the inductive hypothesis. Now, by the inductive hypothesis, we have $\mathcal{D} \leqslant \mathcal{B}$ and thus $A' \leqslant A$ and $\mathcal{D}' \leqslant \mathcal{B}'$. Then, since $\sigma = \mathsf{mgu}(A, H_1)$, $\sigma' = \mathsf{mgu}(A', H_1)$, $\mathcal{V}ar(H_1 \leftarrow \mathcal{B}_1) \cap \mathcal{V}ar(A) = \{\}$, and $\mathcal{V}ar(H_1 \leftarrow \mathcal{B}_1) \cap \mathcal{V}ar(A') = \{\}$, it is easy to see that $A'\sigma' \leqslant A\sigma$ (and thus $\mathcal{D}'\sigma' \leqslant \mathcal{B}'\sigma$) and $\sigma'|H_1 \leqslant \sigma|H_1$ (and thus $\mathcal{B}_1\sigma' \leqslant \mathcal{B}_1\sigma$). Therefore, we can conclude $(\mathcal{B}_1\sigma', \mathcal{D}'\sigma') \leqslant (\mathcal{B}_1\sigma, \mathcal{B}'\sigma)$. Finally, using a similar argument, we have $p(\overline{X_n})\theta\sigma' \leqslant p(\overline{t_n})\delta\sigma$.

$\square$

B.2 Solving Unifiability Problems

The following auxiliary results are useful to prove the correctness of the algorithms *Pos* (Definition 7) and *PosNeg* (Definition 8).

Lemma 1
Suppose that $A\theta = B\theta$ for some atoms A and B and some substitution θ. Then we have $A\theta\eta = B\eta\theta\eta$ for any substitution η with $[\mathit{Dom}(\eta) \cap \mathcal{V}ar(B)] \cap \mathit{Dom}(\theta) = \{\}$ and $\mathit{Ran}(\eta) \cap \mathit{Dom}(\theta\eta) = \{\}$.

Proof
For any $X \in \mathcal{V}ar(B)$,

- either $X \notin \mathit{Dom}(\eta)$ and then $X\eta\theta\eta = X\theta\eta$
- or $X \in \mathit{Dom}(\eta)$ and then $X\eta\theta\eta = (X\eta)\theta\eta = X\eta$ because $\mathit{Ran}(\eta) \cap \mathit{Dom}(\theta\eta) = \{\}$. Moreover, $X \notin \mathit{Dom}(\theta)$ because $[\mathit{Dom}(\eta) \cap \mathcal{V}ar(B)] \cap \mathit{Dom}(\theta) = \{\}$, so $X\theta\eta = X\eta$. Finally, $X\eta\theta\eta = X\theta\eta$.

Consequently, $B\eta\theta\eta = B\theta\eta$. As $A\theta = B\theta$, we have $A\theta\eta = B\theta\eta$ i.e. $A\theta\eta = B\eta\theta\eta$.
$\square$

Proposition 1
The loop at line 2 always terminates and the following statement is an invariant of this loop.

(inv) For each $A \in \mathcal{A}$ there exists $B_A \in \mathcal{B}$ and a substitution θ_A such that $A\theta_A = B_A\theta_A$; moreover, $\mathcal{V}ar(\mathcal{B}) \cap Dom(\theta_A \mid A \in \mathcal{A}) = \{\}$.

Proof
Action (2b) reduces the number of simple disagreement pairs in $\mathcal{B}$ which implies termination of the loop at line 2.

Let us prove that (inv) is an invariant. First, (inv) clearly holds upon initialization of $\mathcal{B}$. Suppose it holds prior to an execution of action (2b). Let t, t' be the selected simple disagreement pair. Then, we consider a substitution η determined by t, t'. For any $X \in Ran(\eta)$, we have $X \in \mathcal{V}ar(\mathcal{B})$ so $X \notin Dom(\theta_A \mid A \in \mathcal{A})$ by (inv). Hence $Ran(\eta) \cap Dom(\theta_A \mid A \in \mathcal{A}) = \{\}$. Moreover, as t, t' is a simple pair we have $Ran(\eta) \cap Dom(\eta) = \{\}$. Hence,

$$Ran(\eta) \cap Dom(\theta_A\eta \mid A \in \mathcal{A}) = \{\} \ . \tag{B1}$$

For any $A \in \mathcal{A}$ and any $X \in Dom(\eta) \cap \mathcal{V}ar(B_A)$, we have $X \notin Dom(\theta_A)$ by (inv). Therefore, we have $[Dom(\eta) \cap \mathcal{V}ar(B_A)] \cap Dom(\theta_A) = \{\}$. Consequently, by (B1) and Lemma 1 we have

$$A\theta_A\eta = B_A\eta\theta_A\eta \ .$$

Suppose by contradiction that $\mathcal{V}ar(\mathcal{B}\eta) \cap Dom(\theta_A\eta \mid A \in \mathcal{A})$ is not empty and let X be one of its elements. We have $X \notin Dom(\eta)$ because $Ran(\eta) \cap Dom(\eta) = \{\}$, so $X \in Dom(\theta_A \mid A \in \mathcal{A})$. Moreover, $X \notin Ran(\eta)$ by (B1) so $X \in \mathcal{V}ar(\mathcal{B})$. Therefore, $X \in \mathcal{V}ar(\mathcal{B}) \cap Dom(\theta_A \mid A \in \mathcal{A})$ which by (inv) gives a contradiction. Consequently,

$$\mathcal{V}ar(\mathcal{B}\eta) \cap Dom(\theta_A\eta \mid A \in \mathcal{A}) = \{\} \ .$$

So, (inv) holds upon termination of action (2b) because $\mathcal{B}$ is set to $\mathcal{B}\eta$. □

Proposition 2
The loop at line 3 always terminates and the following statement is an invariant of this loop.

(inv′) For each $A \in \mathcal{A}$ there exists $B_A \in \mathcal{B}$ and a substitution θ_A such that $A\theta_A = B_A\theta_A$; moreover, $\mathcal{V}ar(\mathcal{B}) \cap Dom(\theta_A \mid A \in \mathcal{A}) \subseteq \mathcal{U}$.

Proof
Action (3b) reduces the number of disagreement pairs in $\mathcal{B}$ which implies termination of the loop at line 3.

Let us prove that (inv′) is an invariant. By Proposition 1, (inv) holds upon termination of the loop at line 2, hence (inv′) holds just before execution of the loop at line 3. Suppose it holds prior to an execution of action (3b). Let t, t' be the selected disagreement pair. Then t, t' is replaced in $\mathcal{B}$ by a fresh variable $U \in \mathcal{U}$. Let $\eta := \{U/t\}$ and $\eta' := \{U/t'\}$. Both η and η' are substitutions because $U \neq t$ and $U \neq t'$ (U is fresh). Let B, B' be the atoms of $\mathcal{B}$ where t, t' come from and C, C' be the atoms obtained by replacing t, t' in B, B' by U. Then $B = C\eta$ and $B' = C'\eta'$.

For any $A \in \mathcal{A}$, we have $A\theta_A = B_A\theta_A$ by (inv$'$). Moreover, $A = A\eta = A\eta'$ because U does not occur in A. So if $B_A = B$ then $A\eta\theta_A = C\eta\theta_A$ and if $B_A = B'$ then $A\eta'\theta_A = C'\eta'\theta_A$. Consequently, for any $A \in \mathcal{A}$ let us set

- $\theta'_A := \theta_A$ and $B'_A := B_A$ if $B_A \notin \{B, B'\}$
- $\theta'_A := \eta\theta_A$ and $B'_A := C$ if $B_A = B$
- $\theta'_A := \eta'\theta_A$ and $B'_A := C'$ if $B_A = B'$.

Then we have

$$\forall A \in \mathcal{A} \quad A\theta'_A = B'_A\theta'_A \ . \tag{B2}$$

Moreover, $\mathit{Dom}(\theta'_A \mid A \in \mathcal{A}) \subseteq \mathit{Dom}(\theta_A \mid A \in \mathcal{A}) \cup \mathit{Dom}(\eta) \cup \mathit{Dom}(\eta')$ i.e.

$$\mathit{Dom}(\theta'_A \mid A \in \mathcal{A}) \subseteq \mathit{Dom}(\theta_A \mid A \in \mathcal{A}) \cup \{U\} \ . \tag{B3}$$

As $\mathcal{V}ar(C, C') \subseteq \mathcal{V}ar(B, B') \cup \{U\}$ then

$$\mathcal{V}ar(C, C') \cap \mathit{Dom}(\theta'_A \mid A \in \mathcal{A}) \subseteq \mathcal{U}$$

because $\mathcal{V}ar(B, B') \cap \mathit{Dom}(\theta_A \mid A \in \mathcal{A}) \subseteq \mathcal{U}$ by (inv$'$) and $\mathcal{V}ar(B, B') \cap \{U\} = \{U\} \cap \mathit{Dom}(\theta_A \mid A \in \mathcal{A}) = \{\}$ and $\{U\} \cap \{U\} \subseteq \mathcal{U}$. Moreover, by (inv$'$) we have $\mathcal{V}ar(\mathcal{B}) \cap (\mathit{Dom}(\theta_A \mid A \in \mathcal{A}) \cup \{U\}) \subseteq \mathcal{U}$ so by (B3)

$$\mathcal{V}ar(\mathcal{B}) \cap \mathit{Dom}(\theta'_A \mid A \in \mathcal{A}) \subseteq \mathcal{U} \ .$$

Hence, $\mathcal{V}ar(\mathcal{B} \setminus \{B, B'\} \cup \{C, C'\}) \cap \mathit{Dom}(\theta'_A \mid A \in \mathcal{A}) \subseteq \mathcal{U}$. With (B2) this implies that upon termination of action (3b) the invariant (inv$'$) holds because B is set to C and B' to C'. $\square$

Proof of Theorem 2
Proposition 1 and Proposition 2 imply termination of the algorithm. Upon termination of the loop at line 3 we have $|\mathcal{B}| = 1$. Let B be the element of $\mathcal{B}$ and η be a substitution with $\mathit{Dom}(\eta|B) \cap \mathcal{U} = \{\}$ and $\mathit{Ran}(\eta|B) = \{\}$. Let us set $\eta' := \eta|B$. Let $A \in \mathcal{A}$. By Proposition 2, there exists a substitution θ_A such that $A\theta_A = B\theta_A$. For any $X \in \mathit{Dom}(\eta') \cap \mathcal{V}ar(B)$, we have $X \notin \mathcal{U}$ hence $X \notin \mathit{Dom}(\theta_A)$ by Proposition 2. Therefore $[\mathit{Dom}(\eta') \cap \mathcal{V}ar(B)] \cap \mathit{Dom}(\theta_A) = \{\}$. Moreover, $\mathit{Ran}(\eta') \cap \mathit{Dom}(\theta_A\eta') = \{\}$ because $\mathit{Ran}(\eta') = \{\}$. Consequently, by Lemma 1 we have $A\theta_A\eta' = B\eta'\theta_A\eta'$ i.e., A and $B\eta' = B\eta$ unify.
$\square$

Proof of Theorem 3
Each step of the algorithm terminates, hence the algorithm terminates. Note that θ always exists as B unifies with each atom of the input of the first step, which includes a copy of A. Assume that the algorithm returns a substitution σ. The set $G\sigma$ is ground by construction. Moreover, we have $A\theta = B\theta$ hence $A\theta\eta = B\theta\eta$ i.e. $A\sigma = B\sigma$. As $\mathit{Dom}(\sigma|B) \cap \mathcal{U} = \{\}$ and $\mathit{Ran}(\sigma|B) = \{\}$, by Theorem 2 we have that $B\sigma$, hence $A\sigma$, unifies with all the atoms of $\mathcal{H}_{pos}$. Finally, the last check ensures that $A\sigma$ does not unify with any atom of H^-. $\square$

The following example shows that the algorithm *PosNeg* is not complete.

Example 6 (*PosNeg*)
Let $A := p(X,Y)$, $\mathcal{H}_{pos} := \{p(a,a), p(b,b)\}$, $\mathcal{H}_{neg} := \{p(c,d)\}$, and $G := \{\}$. The algorithm of Section 4.2.1 applied to $\mathcal{H}_{pos} \cup \{p(X',Y')\}$ returns $B := p(U,U')$. We have $\theta = \{X/U, Y/U'\}$ and $G\theta = \{\}$. We take $\eta = id$, then $\theta\eta = \{X/U, Y/U'\}$, $\theta\eta|B = id$, and $Ran(\theta\eta|B) = \{\}$. Let $\sigma = \{X/U, Y/U'\}$, $Dom(\sigma|B) = \{\}$ does not intersect with $\mathcal{U}$. At last, we have that $A\sigma = p(U,U')$ unifies with $p(c,d) \in \mathcal{H}_{neg}$, hence the algorithm returns fail. But for $\sigma' := \{X/U, Y/U\}$ we have that $A\sigma'$ unifies with each element of $\mathcal{H}_{pos}$ and does not unify with the element of $\mathcal{H}_{neg}$.

Appendix C Some More Examples on Solving Unifiability Problems

Example 7 (*Pos*)
Let $\mathcal{A} := \{p(s(a)), p(s(b)), p(X)\}$. First the algorithm *Pos* sets $\mathcal{B} := \mathcal{A}$, then it considers the simple disagreement pairs in $\mathcal{B}$. The substitution $\eta := \{X/s(a)\}$ is determined by the pair $s(a), X$ and action (2b) sets $\mathcal{B}$ to $\mathcal{B}\eta = \{p(s(a)), p(s(b))\}$. Now no simple disagreement pair occurs in $\mathcal{B}$ hence the algorithm skips to the loop at line 3. Action (3b) replaces the pair a, b with a fresh variable $U \in \mathcal{U}$, hence $\mathcal{B}$ is set to $\{p(s(U))\}$. As $|\mathcal{B}| = 1$ the loop at line 3 stops and the algorithm returns $p(s(U))$.

Example 8 (*Pos*)
Let $\mathcal{A} := \{p(s(a), s(c)), p(s(b), s(c)), p(Z,Z)\}$. First the algorithm sets $\mathcal{B} := \mathcal{A}$, then it considers the simple disagreement pairs in $\mathcal{B}$. The substitution $\eta := \{Z/s(c)\}$ is determined by $Z, s(c)$. Action (2b) sets $\mathcal{B}$ to $\mathcal{B}\eta$ i.e. to

$$\{p(s(a), s(c)), p(s(b), s(c)), p(s(c), s(c))\} .$$

Now no simple disagreement pair occurs in $\mathcal{B}$ hence the algorithm skips to the loop at line 3.

- Action (3b) replaces the disagreement pair a, b with a fresh variable $U \in \mathcal{U}$, hence $\mathcal{B}$ is set to $\{p(s(U), s(c)), p(s(c), s(c))\}$.
- Action (3b) replaces the disagreement pair U, c with a fresh variable $U' \in \mathcal{U}$, hence $\mathcal{B}$ is set to $\{p(s(U'), s(c))\}$.

As $|\mathcal{B}| = 1$ the loop at line 3 stops and the algorithm returns $p(s(U'), s(c))$.

Example 9 (*Pos*)
Let $\mathcal{A} : \{p(s(f(X)), X), p(s(g(a)), Y), p(Z,Z)\}$. First the algorithm sets $\mathcal{B} := \mathcal{A}$, then it considers the simple disagreement pairs in $\mathcal{B}$.

- The pair X, Y is simple and $\eta_1 := \{Y/X\}$ is determined by X, Y. Action (2b) sets $\mathcal{B}$ to $\mathcal{B}\eta_1$ i.e. to $\{p(s(f(X)), X), p(s(g(a)), X), p(Z,Z)\}$.
- The pair X, Z is simple and $\eta_2 := \{Z/X\}$ is determined by X, Z. Action (2b) sets $\mathcal{B}$ to $\mathcal{B}\eta_2$ i.e. to $\{p(s(f(X)), X), p(s(g(a)), X), p(X,X)\}$.
- The pair $X, s(g(a))$ is simple and $\eta_3 := \{X/s(g(a))\}$ is determined by $X, s(g(a))$. Action (2b) sets $\mathcal{B}$ to $\mathcal{B}\eta_3$ i.e. to

$$\{p(s(f(s(g(a)))), s(g(a))),\ p(s(g(a)), s(g(a)))\} .$$

Now no simple disagreement pair occurs in $\mathcal{B}$ hence the algorithm skips to the loop at line 3. Action (3b) replaces the disagreement pair $f(s(g(a))), g(a)$ with a fresh variable $U \in \mathcal{U}$, hence $\mathcal{B}$ is set to $\{p(s(U), s(g(a)))\}$. As $|\mathcal{B}| = 1$ the loop at line 3 stops and the algorithm returns $p(s(U), s(g(a)))$.

We note that the set $\mathcal{B}$ used by the algorithm *Pos* of Definition 7 may contain several occurrences of a same, non-simple, disagreement pair. In such a situation, the algorithm replaces each occurrence with a different variable of $\mathcal{U}$, which may introduce some imprecision. A solution may consist in replacing each occurrence by the same variable but we have to be careful as illustrated by the following example.

Example 10 (*Pos*)
Let $\mathcal{A} := \{p(a, a), p(b, b)\}$. The disagreement pair a, b is not simple and it occurs twice in $\mathcal{A}$. Action (3b) replaces each occurrence of a, b with a different variable of $\mathcal{U}$ and the algorithm returns $p(U, U')$. Instead, if we replace each occurrence of a, b with the same variable we get $p(U, U)$ which unifies both with $p(a, a)$ and $p(b, b)$.

Now let $\mathcal{A} := \{p(a, b), p(b, a)\}$. The disagreement pair a, b is not simple. It occurs twice in $\mathcal{A}$ but we cannot replace each occurrence with the same variable U of $\mathcal{U}$ because $p(U, U)$ unifies with none of the atoms of $\mathcal{A}$.