Concolic Testing in CLP

FRED MESNARD, ÉTIENNE PAYET
LIM - Université de la Réunion, France
(*e-mail:* {frederic.mesnard, etienne.payet}@univ-reunion.fr)

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

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

Abstract

Concolic testing is a popular software verification technique based on a combination of concrete and symbolic execution. Its main focus is finding bugs and generating test cases with the aim of maximizing code coverage. A previous approach to concolic testing in logic programming was not sound because it only dealt with positive constraints (by means of substitutions) but could not represent negative constraints. In this paper, we present a novel framework for concolic testing of CLP programs that generalizes the previous technique. In the CLP setting, one can represent both positive and negative constraints in a natural way, thus giving rise to a sound and (potentially) more efficient technique. Defining verification and testing techniques for CLP programs is increasingly relevant since this framework is becoming popular as an intermediate representation to analyze programs written in other programming paradigms.

KEYWORDS: CLP, verification, concolic testing.

1 Introduction

Symbolic execution was first proposed by King (1976) as a technique for automated test case generation. Essentially, the program is run with some unknown (symbolic) input data. Symbolic execution then proceeds by speculatively exploring all possible computations. Let us consider a simple imperative language with *conditionals* and that the *trace* of an execution is denoted by the sequence of choices made in the conditionals of this execution (*e.g.*, the trace `tft` denotes that execution entered the `true` branch of the first conditional, then the `false` branch of the second conditional, and finally the `true` branch of the third conditional).

During symbolic execution, whenever a conditional with condition c is found, one should explore both branches. In one of the branches, c is assumed; in the other branch, one can assume the negation of this condition *i.e.*, $\neg c$. By gathering all the constraints assumed in a symbolic execution, and solving them, one can produce values for the input arguments. Symbolic execution methods are *sound* in the following sense: if a symbolic execution with trace π collects constraints $c_1, \ldots, c_n$, then solving $c_1 \wedge \ldots \wedge c_n$ will produce values for a concrete call whose execution will have the same trace π (*i.e.*, it will follow the same execution path of the symbolic

∗ This author has been partially supported by EU (FEDER) and Spanish MCI/AEI under grants TIN2016-76843-C4-1-R and PID2019-104735RB-C41, and by the *Generalitat Valenciana* under grant Prometeo/2019/098 (DeepTrust).

execution that produced these constraints). This is a key property in order to achieve a good program coverage. Note that test case generation based on symbolic execution is in principle aimed at a full path coverage.

Concolic testing (Godefroid et al. 2005; Sen et al. 2005) can be seen as an evolution of test case generation methods based on symbolic execution. The main difference is that, now, both concrete and symbolic executions are performed in parallel (thus the term "concolic": *conc*rete + symb*olic*). Roughly speaking, concolic testing proceeds iteratively as follows. It starts with an arbitrary concrete call. Then, this call is executed with the standard semantics, together with a corresponding symbolic call that mimics the execution of the concrete one. This is called a concolic execution. Once this concolic execution terminates, one can produce alternative test cases by negating some of the collected constraints and, then, solving them. For example, if we gathered the sequence of constraints c_1, c_2, c_3 (*e.g.*, associated to the execution of three conditionals) with associated trace `ttt`, we can now solve the constraints $\neg c_1$ (trace `f`), $c_1 \wedge \neg c_2$ (trace `tf`) and $c_1 \wedge c_2 \wedge \neg c_3$ (trace `ttf`) in order to produce three new, alternative test cases that will follow a different execution path. A new iteration starts by considering any of the new test cases, and so forth. In principle, the process terminates when all alternative test cases have been processed. Nevertheless, the search space is typically infinite (as in symbolic execution based methods).

Concolic execution has gained popularity because of some advantages over the symbolic execution based methods. For instance, one can automatically detect some run-time errors since concolic testing performs standard (concrete) executions and, thus, if some error is spotted, we know that this is an actual run-time error. Furthermore, when the constraints become too complex for state-of-the-art solvers and the methods based on symbolic execution just give up, concolic testing can still inject some concrete data (from the concrete component) and simplify the constraints in order to make them tractable.

Although concolic testing is quite popular in imperative and object-oriented languages, only a few works can be found in the context of functional and logic programming languages. Some notable exceptions are those of Giantsios et al. (2015) and Palacios and Vidal (2015) for a functional language, and those of Vidal (2014) and Mesnard et al. (2015a) for a logic language. In the context of logic programming, concolic execution becomes particularly challenging because computing the alternatives of a predicate call is not as straightforward as in imperative programming, where negating a condition suffices. Consider, *e.g.*, a predicate call that matches rules r_1 and r_2. Here, a full path coverage should include test cases for all the following alternatives: no rule is matched; only rule r_1 is matched; only rule r_2 is matched; and both rules r_1 and r_2 are matched (assuming all these cases are feasible). The problem of finding all these alternative test cases is based on so-called *selective unification* (Mesnard et al. 2015a; Mesnard et al. 2017).

A limitation of the approach to concolic testing of Mesnard et al. (2015a) is that only *positive* constraints (represented as substitutions) are gathered during concolic execution. As a consequence, the algorithm is not sound in the above sense, as witnessed by the following example:

Example 1
Let us consider the following simple program:

$$\begin{array}{ll} p(f(a)). & (r_1) \\ p(f(X)) \leftarrow q(X). & (r_2) \\ q(b). & (r_3) \end{array}$$

where terms are built, *e.g.*, from constants a, b, c and the unary function symbol f. If we consider a semantics that only computes the first solution of a goal (as in the approach by Mesnard et

al. (2015a)), the only *feasible* execution paths for an initial goal that calls predicate p are the following:

- A call that matches no rule, *e.g.*, $p(a)$.
- A call that matches both rules r_1 and r_2 and then succeeds, *e.g.*, $p(f(a))$.
- A call that matches only rule r_2 and, then, calls predicate q and matches rule r_3, *e.g.*, $p(f(b))$.
- A call that matches only rule r_2 and, then, calls predicate q but does not match rule r_3, *e.g.*, $p(f(c))$.

However, the concolic testing procedure of Mesnard et al. (2015a) may fail to compute the last test case. For instance, let us consider that the process starts with the initial call $p(a)$, which matches no rule. Now, the computed alternatives could be $p(f(a))$ that matches both r_1 and r_2 and $p(f(b))$ that only matches r_2.[1] Let us first consider $p(f(a))$. This call immediately succeeds, so there are no more alternatives to be computed. Consider now the call $p(f(b))$. This call first matches rule r_2 and, then, calls $q(b)$, which succeeds. Here, one can still generate a new alternative test case: one that (only) matches rule r_2 and, then, fails to match rule r_3. Unfortunately, the concolic testing algorithm of Mesnard et al. (2015a) may generate $p(f(a))$ again since it only knows that the argument of p must unify with $f(X)$ (to match rule r_2) and that X must not unify with b (to avoid matching rule r_3). Thus, $p(f(a))$ is a solution. However, this is not the solution we expected, since this call will match rule r_1 and succeed immediately.

In this work, we consider the development of a concolic testing framework for CLP programs, where both positive and *negative* constraints can be represented in a natural way. Our main contributions are the following:

- We extend the original framework (Mesnard et al. 2015a) to CLP programs. In particular, we illustrate our approach with two instances: CLP($\mathscr{T}erm$) and CLP($\mathscr{N}$). As an advantage of this formulation, efficient external constraint solvers can be used to produce test cases.
- In contrast to previous approaches, we prove the soundness of our approach, *i.e.*, whenever a test case for a given execution path is produced, we can ensure that the execution of this test case will indeed follow the associated path. This can be ensured thanks to the use of *negative* constraints.
- We prove that, if the constraint domain is decidable, then the so-called selective unification problem is decidable too. Thus we extend the results of Mesnard et al. (2017).

Defining verification and testing techniques for CLP programs is increasingly relevant since this setting is becoming popular as an intermediate representation to analyze programs written in other programming paradigms, see, *e.g.*, the work of Gange et al. (2015) and Gurfinkel et al. (2015). Furthermore, concolic testing may be useful in the context of run-time verification techniques; see, *e.g.*, the work of Stulova et al. (2014). Therefore, our approach to concolic testing may constitute a significant contribution to these research areas.

Some more details and proofs of technical results can be found in the extended version of this paper (Mesnard et al. 2020).

[1] Note that matching only r_1 is not feasible in this case. *E.g.*, there is no call of the form $p(t)$ for some term t such that $p(t)$ matches rule r_1 but not r_2.

2 Preliminaries

We assume some familiarity with the standard definitions and notations for logic programming as introduced by Apt (1997) and for constraint logic programming as introduced by Jaffar et al. (1998). 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.

We denote by $|S|$ the cardinality of the set S and by $\mathbb{N}$ the set of natural numbers. From now on, we fix an infinite countable set $\mathscr{V}$ of *variables* together with a *signature* Σ, *i.e.*, a pair $\langle F, \Pi_C \rangle$ where F is a finite set of *function symbols* and Π_C is a finite set of *predicate symbols* with $F \cap \Pi_C = \{\}$ and $(F \cup \Pi_C) \cap \mathscr{V} = \{\}$. Every element of $F \cup \Pi_C$ has an *arity* which is the number of its arguments. We write $f/n \in F$ (resp. $p/n \in \Pi_C$) to denote that f (resp. p) is an element of F (resp. Π_C) whose arity is $n \geq 0$. A *constant symbol* is an element of F whose arity is 0.

A *term* is a variable, a constant symbol or an entity $f(t_1, \ldots, t_n)$ where $f/n \in F$, $n \geq 1$ and $t_1, \ldots, t_n$ are terms. For any term t, we let $\mathscr{V}ar(t)$ denote the set of variables occurring in t. This notation is naturally extended to sets of terms. We say that t is *ground* when $\mathscr{V}ar(t) = \{\}$.

An *atomic constraint* is an element $p/0$ of Π_C or an entity $p(t_1, \ldots, t_n)$ where $p/n \in \Pi_C$, $n \geq 1$ and $t_1, \ldots, t_n$ are terms. A first-order *formula* on Σ is built from atomic constraints in the usual way using the logical connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$ and the quantifiers $\exists$ and $\forall$. For any formula φ, we let $\mathscr{V}ar(\varphi)$ denote its set of free variables and $\exists\varphi$ (resp. $\forall\varphi$) its existential (resp. universal) closure.

We fix a Σ-*structure* $\mathscr{D}$, *i.e.*, a pair $\langle D, [\cdot] \rangle$ which is an interpretation of the symbols in Σ. The set D is called the *domain* of $\mathscr{D}$ and $[\cdot]$ maps each $f/0 \in F$ to an element of D, each $f/n \in F$ with $n \geq 1$ to a function $[f] : D^n \rightarrow D$, each $p/0 \in \Pi_C$ to an element of $\{0,1\}$, and each $p/n \in \Pi_C$ with $n \geq 1$ to a boolean function $[p] : D^n \rightarrow \{0,1\}$. We assume that the binary predicate symbol $=$ is in Σ and is interpreted as identity in D. A *valuation* is a mapping from $\mathscr{V}$ to D. Each valuation v extends by morphism to terms. A valuation v induces a valuation $[\cdot]_v$ of terms to D and of formulas to $\{0,1\}$.

Given a formula φ and a valuation v, we write $\mathscr{D} \models_v \varphi$ when $[\varphi]_v = 1$. We write $\mathscr{D} \models \varphi$ when $\mathscr{D} \models_v \varphi$ for all valuations v. Notice that $\mathscr{D} \models \forall\varphi$ if and only if $\mathscr{D} \models \varphi$, that $\mathscr{D} \models \exists\varphi$ if and only if there exists a valuation v such that $\mathscr{D} \models_v \varphi$, and that $\mathscr{D} \models \neg\exists\varphi$ if and only if $\mathscr{D} \models \neg\varphi$. We say that a formula φ is *satisfiable* (resp. *unsatisfiable*) in $\mathscr{D}$ when $\mathscr{D} \models \exists\varphi$ (resp. $\mathscr{D} \models \neg\varphi$).

We fix a set $\mathscr{L}$ of admitted formulas, the elements of which are called *constraints*. In this paper, we suppose that $\mathscr{L}$ contains all the atomic constraints, the always satisfiable constraint `true` and the unsatisfiable constraint `false`, and any quantified boolean combination of such formulae (while usually $\mathscr{L}$ only contains conjunctions of atomic constraints which are implicitly existentially quantified). We assume that there is a computable function *solv* which maps each $c \in \mathscr{L}$ to one of `true` or `false` indicating whether c is satisfiable or unsatisfiable in $\mathscr{D}$. In particular, it implies that the constraint domain has to be decidable. We call *solv* the *constraint solver*.

Example 2 (*CLP($\mathscr{N}$) and CLP($\mathscr{T}erm$)*)
The constraint domain $\mathscr{N}$ has $<, \leq, =, \neq, \geq, >$ as predicate symbols, $+$ as function symbol and sequences of digits as constant symbols. The domain of computation is the structure with the set of naturals, denoted by $\mathbb{N}$, as domain and where the predicate symbols and the function symbol are interpreted as the usual relations and function over the naturals. A constraint solver for $\mathscr{N}$ is described by, *e.g.*, Comon and Kirchner (1999).

The constraint domain $\mathscr{T}erm$ has $=, \neq$ as predicate symbols and strings of alphanumeric characters as function symbols. The domain of computation is the set of *finite trees* (or, equivalently,

of finite terms), *Tree*. The interpretation of a constant is a tree with a single node labeled with the constant. The interpretation of an n-ary function symbol f is the function $f_{Tree} : Tree^n \to Tree$ mapping the trees $T_1, \ldots, T_n$ to a new tree with root labeled with f and with $T_1, \ldots, T_n$ as child nodes. A constraint solver for $\mathscr{T}erm$ is also described in (Comon and Kirchner 1999).

We let $\overline{o_n}$ denote the finite sequence of syntactic objects $o_1, \ldots, o_n$; we also write $\overline{o}$ when the number of elements is not relevant. We let ε denote the empty sequence and $\overline{o}, \overline{o'}$ denote the concatenation of sequences $\overline{o}$ and $\overline{o'}$. Sequences of distinct variables are denoted by $\overline{X}$, $\overline{Y}$ or $\overline{Z}$ and are sometimes considered as sets of variables. Sequences of (not necessarily distinct) terms are denoted by $\overline{s}$, $\overline{t}$ or $\overline{u}$. Given two sequences of n terms $\overline{s_n}$ and $\overline{t_n}$, we write $\overline{s_n} = \overline{t_n}$ to denote the constraint $s_1 = t_1 \wedge \cdots \wedge s_n = t_n$. We also extend the notation $[\cdot]_v$ by letting $[\overline{s_n}]_v$ denote the sequence $[s_1]_v, \ldots, [s_n]_v$.

The signature in which all programs and queries under consideration are included is $\Sigma_L = \langle F, \Pi_C \cup \Pi_P \rangle$ where Π_P is the set of predicate symbols that can be defined in programs, with $\Pi_C \cap \Pi_P = \{\}$. An *atom* has the form $p(\overline{s_n})$ where $p/n \in \Pi_P$ and $\overline{s_n}$ is a sequence of terms. The definitions and notations on terms ($\mathscr{V}ar$, ground,...) are extended to atoms in the natural way. We write $[p(\overline{s_n})]_v$ to denote the atom $p([\overline{s_n}]_v)$. For any sequence $\overline{A_m}$ of atoms we let $[\overline{A_m}]_v$ denote the sequence $[A_1]_v, \ldots, [A_m]_v$. A *rule* has the form $H \leftarrow c \wedge \overline{B}$ where H is an atom called the *head* of the rule, c is a satisfiable constraint and $\overline{B}$ is a finite sequence of atoms. For the sake of readability, in examples we may simplify rules of the form $H \leftarrow c \wedge \varepsilon$ to $H \leftarrow c$. A *program* is a finite set of rules. A *state* has the form $\langle d \,|\, \overline{B} \rangle$ where $\overline{B}$ is a finite sequence of atoms and d is a constraint. A *constraint atom* is a state of the form $\langle d \,|\, p(\overline{t}) \rangle$. We denote states as Q, $Q' \ldots$ or R, $R' \ldots$ and constraint atoms as C, $C' \ldots$ For any state $Q := \langle d \,|\, \overline{B} \rangle$ and any constraint d', we let $Q \wedge d'$ denote the state $\langle d \wedge d' \,|\, \overline{B} \rangle$.

Any state can be seen as a finite description of a possibly infinite set of sequences of atoms, the arguments of which are values from D. More precisely, the *set described by* a state $Q := \langle d \,|\, \overline{B} \rangle$ is defined as $Set(Q) = \{ [\overline{B}]_v \mid \mathscr{D} \models_v d \}$. For instance, for $Q := \langle Y \leq X + 2 \,|\, p(X), q(Y) \rangle$ in $\mathscr{N}$, we have $p(0), q(2) \in Set(Q)$. For any states $Q := \langle c \,|\, \overline{A} \rangle$ and $Q' := \langle d \,|\, \overline{B} \rangle$, we say that Q' is *less instantiated* (or *more general*) than Q (equivalently, that Q is more restricted than Q'), and we write $Q \leq Q'$, when $\overline{A}$ and $\overline{B}$ are variants and, moreover, $Set(Q) \subseteq Set(Q')$; furthermore, we say they are *equivalent* when $Set(Q) = Set(Q')$ (instead of $Set(Q) \subseteq Set(Q')$). Furthermore, we say that Q and Q' are *equivalent*, and we write $Q \equiv Q'$, when $Set(Q) = Set(Q')$.

We consider the usual operational semantics given in terms of *derivations* from states to states. Let $\langle d \,|\, p(\overline{u}), \overline{B} \rangle$ be a state and $p(\overline{s}) \leftarrow c \wedge \overline{B'}$ be a fresh copy of a rule r. When $solv(\overline{s} = \overline{u} \wedge c \wedge d) =$ `true` then (in this work, a fixed leftmost selection rule is assumed)

$$\langle d \,|\, p(\overline{u}), \overline{B} \rangle \longrightarrow_r \langle \overline{s} = \overline{u} \wedge c \wedge d \,|\, \overline{B'}, \overline{B} \rangle$$

is a *derivation step* of $\langle d \,|\, p(\overline{u}), \overline{B} \rangle$ with respect to r with $p(\overline{s}) \leftarrow c \wedge \overline{B'}$ as its *input rule*. A state $Q := \langle d \,|\, \overline{B} \rangle$ is said to be *successful* if $\overline{B}$ is empty; it is said to be *failed* if $\overline{B}$ is not empty and no derivation step is possible. We write $Q \longrightarrow_P^+ Q'$ to summarize a finite number (> 0) of derivation steps from Q to Q' where each input rule comes from program P. Let Q_0 be a state. A sequence of derivation steps $Q_0 \longrightarrow_{r_1} Q_1 \longrightarrow_{r_2} \cdots$ of maximal length is called a *finished derivation* of $P \cup \{Q_0\}$ when r_1, r_2, ... are rules from P and the *standardization apart* condition holds, *i.e.*, each input rule used is variable disjoint from the initial state Q_0 and from the input rules used at earlier steps.

3 Concolic Execution

In this section, we introduce a *concolic execution* semantics for CLP programs that combines both *conc*rete and symb*olic* execution. Let us now introduce some auxiliary definitions. First, we consider unification on constraint atoms:

Definition 1 (*≈, unification*)
Let C and C' be two constraint atoms. If they have the same predicate symbol, *i.e.*, C has the form $\langle c \,|\, p(\bar{s})\rangle$ and C' has the form $\langle d \,|\, p(\bar{t})\rangle$ then $C \approx C'$ denotes the formula $\bar{s} = \bar{t} \wedge c \wedge d$. Otherwise, $C \approx C'$ is false. We say that C and C' *unify*, or that C *unifies with* C', when $C \approx C'$ is satisfiable (*i.e.*, $\mathscr{D} \models \exists(C \approx C')$ holds).

The following auxiliary function, *c-atom*, produces a constraint atom associated to either a state, a rule or a collection of rules. It selects the leftmost atom together with the constraint.

Definition 2 (*c-atom*)
For any state $Q = \langle c \,|\, \overline{A_n}\rangle$, $n > 0$, we let $c\text{-}atom(Q) = \langle c \,|\, A_1\rangle$. For any rule $r = H \leftarrow c \wedge \overline{B}$, we let $c\text{-}atom(r) = \langle c \,|\, H\rangle$. For any set of rules $\mathscr{R}$ (resp. sequence of rules $\overline{r_n}$), we let $c\text{-}atom(\mathscr{R}) = \{c\text{-}atom(r) \mid r \in \mathscr{R}\}$ (resp. $c\text{-}atom(\overline{r_n}) = c\text{-}atom(r_1), \ldots, c\text{-}atom(r_n)$).

Function rules is then used to determine the program rules that match a particular state:

Definition 3 (rules)
Given a state Q and a set of rules P, we let

$$\mathsf{rules}(Q,P) = \left\{ r \in P \;\middle|\; \begin{array}{l} solv(c\text{-}atom(Q) \approx c\text{-}atom(r')) = \mathtt{true} \\ \text{for some fresh copy } r' \text{ of } r \end{array} \right\}.$$

The following function, *neg_constr*, will be essential to guarantee that symbolic execution is sound, so that symbolic states do not unify with more rules than expected (see below).

Definition 4 (*neg_constr*)
Let $C := \langle c \,|\, p(\bar{s})\rangle$ and $H := \langle d \,|\, p(\bar{t})\rangle$ be some variable disjoint constraint atoms. The constraint $neg\_constr(C,H)$ denotes $\forall V\ (\bar{s} \neq \bar{t} \vee \neg d)$, where V denotes the set of variables occurring in H.

Let $\mathscr{H} := \{\overline{H_k}\}$ be a finite set of constraint atoms that have the same predicate symbol as C and are variable disjoint with C. Then, we let $neg\_constr(C,\mathscr{H}) = neg\_constr(C,H_1) \wedge \ldots \wedge neg\_constr(C,H_k)$. In particular, if $\mathscr{H} = \{\}$, then we have $neg\_constr(C,\mathscr{H}) = \mathsf{true}$.

Given a constraint atom C and a set of constraint atoms $\mathscr{H}$, we have that $C \wedge neg\_constr(C,\mathscr{H})$ does not unify with any constraint atom in $\mathscr{H}$, as expected; moreover, it is *maximal* in the sense that, for any constraint d such that $C \wedge d$ does not unify with any constraint atom in $\mathscr{H}$, d will be less general than $neg\_constr(C,\mathscr{H})$.

In this work, we assume that we are interested in producing test cases that achieve a so-called full *path coverage*, so that every predicate is called in all possible ways, as explained in the introduction. More precisely, given an initial state of the form $\langle \mathsf{true} \,|\, p(X_1,\ldots,X_n)\rangle$, we aim at producing test cases that cover all *feasible* subtrees of the execution space of $\langle \mathsf{true} \,|\, p(X_1,\ldots,X_n)\rangle$. We note that, since the execution space of $\langle \mathsf{true} \,|\, p(X_1,\ldots,X_n)\rangle$ is typically infinite, so is the number of feasible subtrees and, thus, the number of test cases. Therefore, achieving a full path coverage is not possible and one should introduce some strategy to ensure the termination of concolic testing (see below).

In order to identify each derivation so that we can keep track of the already considered derivations in the execution space, we introduce the following notion:

Definition 5 (*trace*)
Given a rule r, we let $\ell(r)$ denote its label, which is unique in a program. A *trace* is a sequence of rule labels. The empty trace is denoted by ε. Given a trace π and a rule label ℓ, we denote by $\pi.\ell$ the concatenation of ℓ to the end of trace π.

Given a derivation with the standard operational semantics, $Q_0 \longrightarrow_{r_1} Q_1 \longrightarrow_{r_2} \ldots \longrightarrow_{r_n} Q_n$, the associated trace is $\ell_1\ell_2\ldots\ell_n$, where $\ell(r_i) = \ell_i$, $i = 1,\ldots,n$.

In the following, we consider that states can be labelled with a trace *i.e.*, S_π denotes a state S which is labelled with trace π. Let us now introduce the notion of concolic state:

Definition 6 (*concolic state*)
A *concolic state* has the form $\langle\!|Q \,]\![\, S_\pi|\!\rangle$ where Q, S are states such that $Q \leq S$ and π is a trace labelling state S. Here, Q is called the *concrete* state of $\langle\!|Q \,]\![\, S_\pi|\!\rangle$, while S_π is called its *symbolic* state; we sometimes omit the trace π from the symbolic state when it is not relevant.

In contrast to other programming paradigms, the notion of *symbolic* execution is very natural in CLP: the structure of both Q and S is the same (*i.e.*, the sequence of atoms are variants), and the only difference (besides some labeling for symbolic states) is that some states might be more restricted in Q than in S.

The standard operational semantics is now extended to concolic states as follows:

Definition 7 (*concolic execution*)
Let P be a program and let $\langle\!|Q \,]\![\, S_\pi|\!\rangle$ be a concolic state. Then, we have a concolic execution step

$$\langle\!|Q \,]\![\, S_\pi|\!\rangle \stackrel{r}{\Longrightarrow}_{\pi,R_Q,R_S} \langle\!|Q' \,]\![\, S'_{\pi.\ell(r)}|\!\rangle$$

if the following conditions hold:

- $\mathsf{rules}(Q,P) = R_Q \neq \{\}$, $\mathsf{rules}(S,P) = R_S$,
- $\gamma = \mathit{neg\_constr}(\mathit{c\text{-}atom}(S), \mathit{c\text{-}atom}(R_S \setminus R_Q))$,
- $r \in R_Q$, $Q \longrightarrow_r Q'$ and $S \wedge \gamma \longrightarrow_r S'$.

Besides the applied rule, r, the step is labelled with the current trace, π, the set of rules matching the concrete state, R_Q, and the set of rules matching the symbolic state, R_S.[2] The applied rule is often omitted when it is not relevant.

A concolic state $\langle\!|\langle c \,|\, \overline{A}\rangle \,]\![\, \langle d \,|\, \overline{B}_\pi\rangle|\!\rangle$ is said to be *successful* if $\overline{A} = \overline{B} = \varepsilon$; it is said to be *failed* if they are not empty and no derivation step is possible. In either case, we say that π is the trace of the derivation. The notion of (finished) derivation is extended from the standard semantics in the natural way.

For each concolic state $\langle\!|Q \,]\![\, S|\!\rangle$ in a derivation, the symbolic component, S, typically unifies with more rules than the concrete component, Q, since Q is more restricted than S (and, thus, $R_Q \subseteq R_S$; see below). However, we want the execution of the symbolic state to mimic that of the concrete state. Therefore, both the concrete and symbolic states can only be unfolded using a rule from R_Q. Furthermore, we introduce a *negative* constraint, γ, into the symbolic state in order to avoid matching more rules than the concrete state. For this purpose, we use function *neg_constr* introduced above. In the remainder of the paper, we assume a fixed program P.

Let $\langle\!|Q \,]\![\, S|\!\rangle$ be a concolic state with $\mathsf{rules}(Q,P) = R_Q$ and $\mathsf{rules}(S,P) = R_S$. Our notion of concolic execution enjoys the following properties:

[2] This information can be safely ignored in this section. It will become relevant in the next section in order to generate test cases.

- $Q \leq S$ implies $\mathsf{rules}(Q,P) \subseteq \mathsf{rules}(S,P)$.
- $\mathsf{rules}(S \wedge \gamma) = R_Q$, where $\gamma = \mathit{neg\_constr}(\mathit{c\text{-}atom}(S), \mathit{c\text{-}atom}(R_S \setminus R_Q))$. Therefore, γ achieves the desired effect of preventing S to unify with the rules in $R_S \setminus R_Q$.
- If $\langle\!|Q \,]\![\, S|\!\rangle \Longrightarrow_{\pi,R_Q,R_S} \langle\!|Q' \,]\![\, S'|\!\rangle$, then $\langle\!|Q' \,]\![\, S'|\!\rangle$ is also a concolic state, which means that concolic execution is well defined in the sense that the property $Q \leq S$ is correctly propagated by concolic execution steps.

W.l.o.g., we only consider *initial* concolic states of the form $\langle\!|Q \,]\![\, S|\!\rangle$, where $Q = \langle c \,|\, p(\overline{X})\rangle$, $S = \langle \mathsf{true} \,|\, p(\overline{Y})_\varepsilon\rangle$, Q and S are variable disjoint, and ε is the empty trace. Trivially, we have $Q \leq S$.

In the following, we assume that all concolic execution derivations start from an *initial* concolic state, so they are well formed.

Example 3
Consider the following CLP($\mathscr{T}erm$) program:

$$\begin{array}{lll} \ell_1: & p(X) \leftarrow X = a. & (r_1) \\ \ell_2: & p(s(Y)) \leftarrow \mathsf{true} \wedge q(Y). & (r_2) \\ \ell_3: & q(W) \leftarrow W = a. & (r_3) \end{array}$$

with rules, r_1, r_2 and r_3, where ℓ_1, ℓ_2, ℓ_3 are unique identifiers for these rules. Given the initial concolic state $\langle\!|\langle X = s(a) \,|\, p(X)\rangle \,]\![\, \langle \mathsf{true} \,|\, p(N)_\varepsilon\rangle|\!\rangle$, we have the following concolic execution:

$$\begin{array}{l} \langle\!|\langle X = s(a) \,|\, p(X)\rangle \,]\![\, \langle \mathsf{true} \,|\, p(N)\rangle_\varepsilon|\!\rangle \\ \Longrightarrow_{\varepsilon,\{r_2\},\{r_1,r_2\}} \langle\!|\langle s(Y) = X \wedge X = s(a) \,|\, q(Y)\rangle \,]\![\\ \qquad\qquad]\![\, \langle s(Y') = N \wedge \forall X' \, (N \neq X' \vee X' \neq a) \,|\, q(Y')\rangle_{\ell_2}|\!\rangle \\ \Longrightarrow_{\ell_2,\{r_3\},\{r_3\}} \langle\!|\langle W = Y \wedge W = a \wedge s(Y) = X \wedge X = s(a) \,|\, \varepsilon\rangle \\ \qquad\qquad]\![\, \langle W' = Y' \wedge W' = a \wedge s(Y') = N \wedge \forall X' \, (N \neq X' \vee X' \neq a) \,|\, \varepsilon\rangle_{\ell_2\ell_3}|\!\rangle \end{array}$$

In the first step, the following negative constraint is computed:

$$\gamma_1 = \mathit{neg\_constr}(\langle \mathsf{true} \,|\, p(N)\rangle, \{\langle X' = a \,|\, p(X')\rangle\}) = \forall X' (N \neq X' \vee X' \neq a)$$

so that $\langle \mathsf{true} \,|\, p(N)\rangle \wedge \gamma_1 = \langle \forall X' (N \neq X' \vee X' \neq a) \,|\, p(N)\rangle$. In the second step, we have $\gamma_2 = \mathsf{true}$ since the matching rules are the same for both the concrete and symbolic states. Hence, no additional negative constraint is added to the symbolic state. The trace of the derivation is thus $\ell_2\ell_3$, *i.e.*, an application of rule r_2 followed by an application of rule r_3.

Now, we can state that concolic execution is indeed a conservative extension of the standard operational semantics:

Theorem 1
Let $\langle\!|Q \,]\![\, S|\!\rangle$ be an initial concolic state. Then, we have $Q \longrightarrow^* Q'$ iff $\langle\!|Q \,]\![\, S|\!\rangle \Longrightarrow^* \langle\!|Q'' \,]\![\, S'|\!\rangle$, where $Q' \equiv Q''$. Moreover, the trace of both derivations is the same.

Finally, the next property states that the constraints computed for the symbolic state ensure—when applied to the initial symbolic state—that the standard semantics will follow the same path. Therefore, our approach to concolic testing can be considered sound. This property did not hold in the original approach of Mesnard et al. (2015a), as explained in the introduction.

Theorem 2 (*soundness*)
Let $\langle\!|Q \,]\![\, S_\varepsilon|\!\rangle$ be an initial concolic state with $\langle\!|Q \,]\![\, S_\varepsilon|\!\rangle \Longrightarrow^* \langle\!|Q' \,]\![\, S'_\pi|\!\rangle$. Let $S = \langle \mathsf{true} \,|\, \overline{A}\rangle$ and $S' = \langle d \,|\, \overline{B}\rangle$. Then, we have $\langle d \,|\, \overline{A}\rangle \longrightarrow^* S''$ such that $S' \equiv S''$ and the associated trace is π.

$$(\mathsf{backtrack})\ \frac{\mathsf{rules}(Q,P)=\{\}\wedge\mathsf{rules}(S,P)=R_S\wedge|\overline{Q}|>0}{\langle\!|Q,\overline{Q}\,]\![\,S_\pi,\overline{S}|\!\rangle\hookrightarrow_{\pi,\{\},R_S}\langle\!|\overline{Q}\,]\![\,\overline{S}|\!\rangle}$$

$$(\mathsf{next})\ \frac{Q=\langle c\,|\,\varepsilon\rangle}{\langle\!|Q,\overline{Q}\,]\![\,S_\pi,\overline{S}|\!\rangle\hookrightarrow_{\pi,\{\},\{\}}\langle\!|\overline{Q}\,]\![\,\overline{S}|\!\rangle}$$

$$(\mathsf{choice})\ \frac{\mathsf{rules}(Q,P)=\{\overline{r_n}\}\wedge n>0\wedge\mathsf{rules}(S,P)=R_S\wedge\ \gamma=\mathit{neg\_constr}(\mathit{c\text{-}atom}(S),\mathit{c\text{-}atom}(R_S\setminus\{\overline{r_n}\}))}{\langle\!|Q,\overline{Q}\,]\![\,S_\pi,\overline{S}|\!\rangle\hookrightarrow_{\pi,\{\overline{r_n}\},R_S}\langle\!|Q^{r_1},\ldots,Q^{r_n},\overline{Q}\,]\![\,\gamma\wedge S_\pi^{r_1},\ldots,\gamma\wedge S_\pi^{r_n},\overline{S}|\!\rangle}$$

$$(\mathsf{unfold})\ \frac{Q\longrightarrow_r R\wedge S\longrightarrow_r T}{\langle\!|Q^r,\overline{Q}\,]\![\,S_\pi^r,\overline{S}|\!\rangle\hookrightarrow_{\pi,\{\},\{\}}\langle\!|R,\overline{Q}\,]\![\,T_{\pi.\ell(r)},\overline{S}|\!\rangle}$$

Fig. 1. Concolic CLP execution semantics (deterministic)

4 Concolic Testing

In this section, we present our concolic testing procedure, which is based on the concolic execution semantics of the previous section.

First, we introduce a *deterministic* version of concolic execution that implements a depth-first search through the concolic execution space (loosely inspired by the linear operational semantics for Prolog introduced by Ströder et al. (2011)). This deterministic semantics better reflects the current implementation (Mesnard et al. 2015b) and, moreover, allows one to keep the information that must survive backtracking steps (*e.g.*, generated test cases and already considered traces).

The deterministic concolic execution semantics is defined by means of a (labelled) transition relation, $\hookrightarrow$, as shown in Figure 1. Now, concolic states have the form $\langle\!|\overline{Q}\,]\![\,\overline{S}|\!\rangle$, where $\overline{Q}$ and $\overline{S}$ are sequences of states (possibly labelled with a rule). Let us briefly explain the rules:

- In contrast to the nondeterministic concolic execution semantics, unfolding is now split into two rules: choice and unfold. Rule choice creates as many copies of the states (both concrete and symbolic) as rules *match the concrete state*. Then, rule unfold just unfolds the leftmost state (both concrete and symbolic) using the rule labeling these states.
  Consider, for example, a concolic state $\langle\!|Q\,]\![\,S_\pi|\!\rangle$. If the nondeterministic version of concolic execution (cf. Definition 7) performs, e.g., the following step
  $$\langle\!|Q\,]\![\,S_\pi|\!\rangle\stackrel{r_1}{\Longrightarrow}_{\pi,R_Q,R_S}\langle\!|Q'\,]\![\,S'_{\pi.\ell(r_1)}|\!\rangle$$
  with $R_Q=\{r_1,\ldots,r_n\}$, then the deterministic version of Figure 1 will perform the choice step
  $$\langle\!|Q\,]\![\,S_\pi|\!\rangle\hookrightarrow_{\pi,R_Q,R_S}\langle\!|Q^{r_1},\ldots,Q^{r_n}\,]\![\,\gamma\wedge S_\pi^{r_1},\ldots,\gamma\wedge S_\pi^{r_n}|\!\rangle$$
  followed by the unfolding step
  $$\langle\!|Q^{r_1},\ldots,Q^{r_n}\,]\![\,\gamma\wedge S_\pi^{r_1},\ldots,\gamma\wedge S_\pi^{r_n}|\!\rangle\hookrightarrow_{\pi,\{\},\{\}}\langle\!|Q',Q^{r_2}\ldots,Q^{r_n}\,]\![\,S'_{\pi.\ell(r_1)},\gamma\wedge S_\pi^{r_2}\ldots,\gamma\wedge S_\pi^{r_n}|\!\rangle$$
  Therefore, we reach the same states, Q' and S'. The only difference is that alternative paths are stored explicitly in the concolic state (*i.e.*, $Q^{r_2},\ldots,Q^{r_n}$ and $\gamma\wedge S_\pi^{r_2},\ldots,\gamma\wedge S_\pi^{r_n}$) and will be explored after a backtracking step (or when looking for more solutions, where an implicit backtracking step is performed).
- When the concrete state does not match any rule, rule backtrack is applied. As before, the step is labeled with the current trace and the constraint atoms associated to the rules

matching both the concrete (the empty set) and symbolic states. Note that we assume that the sequence $\overline{Q}$ is not empty; otherwise, the execution would be finished.
For instance, if state Q' in the example above does match any rule, we will perform the following backtracking step:

$$\langle\!|Q', Q^{r_2} \ldots, Q^{r_n}]\![S'_{\pi.\ell(r_1)}, \gamma \wedge S^{r_2}_{\pi} \ldots, \gamma \wedge S^{r_n}_{\pi} |\!\rangle \hookrightarrow_{\pi.\ell(r_1),\{\},R_{S'}} \langle\!| Q^{r_2} \ldots, Q^{r_n}]\![\gamma \wedge S^{r_2}_{\pi} \ldots, \gamma \wedge S^{r_n}_{\pi} |\!\rangle$$

where $R_{S'} = \mathsf{rules}(S', P)$.

- Finally, rule next is applied when a solution is reached in order to consider alternative solutions (if any). In other words, our calculus explores the complete execution space for the initial state rather than stopping after the first solution is found.

The deterministic version of the concolic execution semantics constitutes an excellent basis for implementing a concolic testing procedure. For instance, one can consider only the computation of the first solution by removing rule next. Furthermore, one can easily guarantee termination by either limiting the length of the considered concolic execution derivations or the "depth" of the search tree in order to only partially explore the execution space.

The following result stating the soundness of the deterministic concolic execution semantics is straightforward:

Theorem 3
Let $\langle\!|Q_0]\![S_0|\!\rangle$ be an initial concolic state. If $\langle\!|Q_0]\![S_0|\!\rangle \hookrightarrow^* \langle\!|Q, \overline{Q}]\![S, \overline{S}|\!\rangle$, then $\langle\!|Q_0]\![S_0|\!\rangle \Longrightarrow^* \langle\!|Q]\![S|\!\rangle$.

Note that the deterministic version is sound but incomplete in general since it implements a depth-first search strategy.

Now, we introduce a function to compute alternative test cases in a concolic execution. In the following definition, we consider a (symbolic) initial state (I), since test cases will always be particular instances of this state, the current (symbolic) state in a derivation (C), the set of atoms matching the concrete state ($\mathscr{H}_Q$), and the set of atoms matching the corresponding symbolic state ($\mathscr{H}_S$). Intuitively speaking, function *alts* produces alternative test cases by restricting the initial symbolic state I so that the current symbolic state C unifies with a subset $\mathscr{H}^+$ of constraint atoms from $\mathscr{H}_S$, except for the set $\mathscr{H}_Q$ which was already considered.

Definition 8 (*alts*)
Let I, C be constraint atoms, with $C = \langle c \,|\, B \rangle$, and $\mathscr{H}_Q, \mathscr{H}_S$ be finite sets of constraint atoms that have the same predicate symbol as C and all atoms are variable disjoint with each other. Then,

$$alts(I, C, \mathscr{H}_Q, \mathscr{H}_S) = \left\{ I \wedge c \wedge \gamma \;\middle|\; \begin{array}{l} \mathscr{H}^+ \in \mathscr{P}(\mathscr{H}_S),\ \mathscr{H}^+ \neq \mathscr{H}_Q \\ \mathscr{H}^- = \mathscr{H}_S \setminus \mathscr{H}^+ \\ \gamma = neg\_constr(C, \mathscr{H}^-) \\ c \wedge \gamma \text{ is satisfiable} \\ \forall H \in \mathscr{H}^+ \ (C \wedge \gamma) \approx H \end{array} \right\}.$$

Example 4 (*CLP(Term)*)
Let us consider the call $alts(I, C, \{H_2\}, \{H_1, H_2\})$, where $I := \langle true \,|\, p(W) \rangle$, $C := \langle c \,|\, q(N) \rangle$ with $c := (W = N)$, $H_1 := \langle X = a \,|\, q(X) \rangle$, and $H_2 := \langle \mathsf{true} \,|\, q(s(M)) \rangle$. For brevity, we remove the occurrences of true in the formulæ below.

Let us consider the case $\mathscr{H}^+ := \{H_1\}$ and $\mathscr{H}^- := \{H_2\}$. Then we have $\gamma = neg\_constr(C, \mathscr{H}^-)$

$$(\mathsf{skip})\ \frac{\mathscr{C} \hookrightarrow_{\pi,R_Q,R_S} \mathscr{C}' \wedge (\pi \in \mathtt{TR} \vee R_S = \{\})}{(\mathtt{PTC},\mathtt{TC},\mathtt{TR},I,\mathscr{C}) \rightsquigarrow (\mathtt{PTC},\mathtt{TC},\mathtt{TR}\cup\{\pi\},I,\mathscr{C}')}$$

$$(\mathsf{alts})\ \frac{\mathscr{C} \hookrightarrow_{\pi,R_Q,R_S} \mathscr{C}' \wedge R_S \neq \{\} \wedge \pi \notin \mathtt{TR}}{(\mathtt{PTC},\mathtt{TC},\mathtt{TR},I,\mathscr{C}) \rightsquigarrow (\mathtt{PTC}\cup \mathit{alts}(I,\mathit{c\text{-}atom}(\mathscr{C}),\mathit{c\text{-}atom}(R_Q),\mathit{c\text{-}atom}(R_S)),\mathtt{TC},\mathtt{TR}\cup\{\pi\},I,\mathscr{C}')}$$

$$(\mathsf{restart})\ \frac{\mathscr{C} \not\hookrightarrow}{(\mathtt{PTC}\cup\{\langle c \,|\, p(\overline{X})\rangle\},\mathtt{TC},\mathtt{TR},I,\mathscr{C}) \rightsquigarrow (\mathtt{PTC},\mathtt{TC}\cup\{\langle c \,|\, p(\overline{X})\rangle\},\mathtt{TR},I,\langle\!|\langle c \,|\, p(\overline{X})\rangle\ ]\![\ I_\varepsilon|\!\rangle)}$$

Fig. 2. Concolic testing

$= \forall M(N \neq s(M))$. As $\mathscr{D} \models_v (c \wedge \gamma)$ holds for any valuation v with $\{W \mapsto a, N \mapsto a\} \subseteq v$, $c \wedge \gamma$ is satisfiable.

Now, we should check that $C \wedge \gamma \approx H_1$ holds. Since $C \wedge \gamma = \langle W = N \wedge \forall M\ (N \neq s(M)) \,|\, q(N)\rangle$ and $\mathit{solv}(N = X \wedge X = a \wedge W = N \wedge \forall M\ (N \neq s(M))) = \mathsf{true}$, it holds. Therefore, we have $I \wedge c \wedge \gamma \in \mathit{alts}(I,C,\{H_2\},\{H_1,H_2\})$, *i.e.*, we produce the state: $\langle W = N \wedge \forall M\ (N \neq s(M)) \,|\, p(W)\rangle$ which could be simplified to $\langle \forall M\ W \neq s(M) \,|\, p(W)\rangle$.

Example 5 (*CLP*($\mathscr{N}$))
Let us consider the call $\mathit{alts}(I,C,\{H_1\},\{H_1,H_2,H_3\})$, where $I := \langle \mathsf{true} \,|\, p(W)\rangle$, $C := \langle c \,|\, q(X)\rangle$, $c := (W = X \wedge X \leq 10)$, $H_1 := \langle Y \leq 2 \,|\, q(Y)\rangle$, $H_2 := \langle 8 \leq Z \leq 10 \,|\, q(Z)\rangle$, and $H_3 := \langle T < 5 \,|\, q(T)\rangle$.

Let us consider the case $\mathscr{H}^+ = \{H_1,H_2\}$ and $\mathscr{H}^- = \{H_3\}$. First, we should compute $\gamma = \mathit{neg\_constr}(C,\mathscr{H}^-)$, *i.e.*, $\forall T\ (X \neq T \vee 5 \leq T)$, which can be simplified to $\gamma = (5 \leq X)$. So, $c \wedge \gamma = (W = X \wedge X \leq 10 \wedge 5 \leq X)$ can be simplified to $c \wedge \gamma = (W = X \wedge 5 \leq X \leq 10)$, which is clearly satisfiable. Now, we should check that $C \wedge \gamma = \langle W = X \wedge 5 \leq X \leq 10 \,|\, q(X)\rangle$ unifies with both H_1 and H_2 in order to produce an element of $\mathit{alts}(I,C,\mathscr{H}_Q,\mathscr{H}_S)$:

- $C \wedge \gamma \approx H_1$. In this case, we have $\mathit{solv}(X = Y \wedge Y \leq 2 \wedge W = X \wedge 5 \leq X \leq 10) = \mathsf{false}$.
- $C \wedge \gamma \approx H_2$. In this case, we have $\mathit{solv}(X = Z \wedge 8 \leq Z \leq 10 \wedge W = X \wedge 5 \leq X \leq 10) = \mathsf{true}$ (consider, *e.g.*, any valuation v with $\{X \mapsto 9, Z \mapsto 9, W \mapsto 9\} \subseteq v$).

Therefore, this case is not feasible and no new test case is produced for it.

4.1 A Concolic Testing Procedure

Now, we consider a concolic testing procedure that aims at achieving a full path coverage. Let us first informally explain the concolic testing procedure. The process starts with some arbitrary test case *i.e.*, an initial concrete state of the form $\langle c \,|\, p(\overline{X})\rangle$. Then, concolic testing proceeds iteratively as follows:

1. First, we form the initial concolic state $\langle\!|\langle c \,|\, p(\overline{X})\rangle\ ]\![\ \langle \mathsf{true} \,|\, p(\overline{Y})\rangle|\!\rangle$ and apply the rules of concolic execution (Figure 1) as much as possible (or up to a number of steps or a time bound, in order to ensure the termination of the process).
2. Now, for each choice or backtrack steps in this derivation, we use function *alts* to compute alternative test cases that will produce a different execution tree. Moreover, we keep track of the traces where alternative test cases have been produced in order to avoid producing the same alternative test cases once and again.

3. When all alternative test cases for the considered concolic execution have been produced, we go back to step (1) above and consider any of the pending test cases produced in the previous step. The iterative algorithm terminates when all pending test cases have been considered and, moreover, no new test cases are produced.

In order to formalise the above process, we introduce *configurations* of the form $(\mathtt{PTC}, \mathtt{TC}, \mathtt{TR}, I, \mathscr{C})$, where PTC is the set of *pending test cases* (test cases that have not been explored yet), TC is the set of test cases already explored, TR is the set of execution traces already considered, I is the initial symbolic state, and $\mathscr{C}$ is a concolic state. The rules of the concolic testing procedure are shown in Figure 2.

Concolic testing starts with an arbitrary concrete state, say $\langle c \,|\, p(\overline{X})\rangle$. Then, we form the initial configuration

$$(\{\}, \{\langle c \,|\, p(\overline{X})\rangle\}, \{\}, \langle \mathsf{true} \,|\, p(\overline{Y})\rangle, \langle\!| \langle c \,|\, p(\overline{X})\rangle \,]\![\, \langle \mathsf{true} \,|\, p(\overline{Y})\rangle_{\varepsilon} |\!\rangle)$$

where $\overline{Y}$ are fresh variables, and apply the rules of Figure 2 until no rule is applicable. The second component of the last configuration will contain the generated test cases. Let us briefly explain the rules of the concolic testing procedure:

- Rule skip applies when either the trace of the current state, π, is already visited or the set of rules matching the symbolic state is empty. The second situation happens in rules next and unfold of the concolic execution semantics, and also when applying rule backtrack but no rule matches the symbolic state. In this case, we simply update the concolic state and the set of considered traces (if any), but no new alternative test cases are produced.
- Rule alts applies when the current trace, π, has not been considered yet and, moreover, the set of rules matching the symbolic state is not empty. This situation happens when applying rules backtrack or choice for the first time. In this case, we update the set of pending test cases using the auxiliary function *alts*. Here, we let $c\text{-}atom(\mathscr{C}) = c\text{-}atom(S)$ when $\mathscr{C} = \langle\!| Q, \overline{Q} \,]\![\, S, \overline{S} |\!\rangle$.
- Finally, rule restart applies when the concolic execution semantics cannot proceed. In this case, we restart the process with a new concrete state from the set of pending test cases.

The procedure terminates when the set of pending tests cases is empty.[3] Then, the generated test cases can be found in the second component of the configuration.

We note that, in general, concolic testing might produce *nonterminating* test cases. Here, one could use the output of some termination analysis to further restrict test cases in order to guarantee terminating computations (*e.g.*, requiring ground arguments or fixed variables). This is an orthogonal issue that constitutes an interesting topic for further research.

4.2 Connections with the Constraint Selective Unification Problem

Here, we fix a constraint atom $C = \langle c \,|\, p(\bar{s})\rangle$ with c satisfiable and two finite sets $\mathscr{H}^+$ and $\mathscr{H}^-$ of constraint atoms. We assume that all constraint atoms are variable disjoint with each other and that C unifies with any constraint atom from $\mathscr{H}^+ \cup \mathscr{H}^-$. We recall the definition of a *constraint selective unification problem* minus its groundness condition (Mesnard et al. 2017).

[3] Note that termination of concolic testing is ensured when concolic execution terminates; see the previous section for some possible strategies.

Definition 9 (*Constraint Selective Unification Problem, $\mathscr{P}$*)
The *constraint selective unification problem* for C with respect to $\mathscr{H}^+$ and $\mathscr{H}^-$ consists in determining whether the following set of constraint atoms is empty:

$$\mathscr{P}(C,\mathscr{H}^+,\mathscr{H}^-) = \left\{ C \wedge d \;\middle|\; \begin{array}{l} c \wedge d \text{ is satisfiable} \\ C \wedge d \text{ is variable disjoint with } \mathscr{H}^+ \cup \mathscr{H}^- \\ \forall H \in \mathscr{H}^+ : C \wedge d \text{ and } H \text{ unify} \\ \forall H \in \mathscr{H}^- : C \wedge d \text{ and } H \text{ do not unify} \end{array} \right\}.$$

For brevity, and as C, $\mathscr{H}^+$ and $\mathscr{H}^-$ are fixed in this section, below we write $\mathscr{P}$ instead of $\mathscr{P}(C,\mathscr{H}^+,\mathscr{H}^-)$ and we let $\gamma = neg\_constr(C,\mathscr{H}^-)$.

Proposition 1

1. Suppose that $c \wedge \gamma$ is satisfiable and $C \wedge \gamma$ unifies with each element of $\mathscr{H}^+$. Then, we have $C \wedge \gamma \equiv C'$ for some $C' \in \mathscr{P}$.
2. For each $C' \in \mathscr{P}$ we have $C' \leq (C \wedge \gamma)$.
3. If $\mathscr{P} \neq \{\}$ then $C \wedge \gamma$ unifies with each constraint atom in $\mathscr{H}^+$.

Below, we naturally let $Set(\mathscr{P}) = \cup_{C' \in \mathscr{P}} Set(C')$.

Theorem 4
If $C \wedge \gamma$ unifies with each element of $\mathscr{H}^+$ then $Set(C \wedge \gamma) = Set(\mathscr{P})$.

Corollary 1
The *constraint selective unification problem* for C with respect to $\mathscr{H}^+$ and $\mathscr{H}^-$ is decidable.

5 Related Work

Concolic testing was originally introduced in the context of imperative programming languages (Godefroid et al. 2005; Sen et al. 2005) and, then, extended to a concurrent language like Java by Sen and Agha (2006). To the best of our knowledge, the first work that considered concolic execution in the context of a nondeterministic, logic programming language was that of Vidal (2014), where some preliminary ideas were introduced. However, the paper presented no formal results nor an implementation of the technique. Later, a more mature approach was proposed by Mesnard et at. (2015a), where the formal concept of a selective unification problem, together with a correct, terminating but incomplete algorithm to solve it, were introduced. The soundness of concolic execution itself was not considered and, indeed, it was not sound, as illustrated in Section 1.

A publicly available proof-of-concept implementation of a concolic testing tool for (pure) Prolog has been developed: contest (Mesnard et al. 2015b). Our present paper generalizes the approach to CLP with first order constraints, which provides some crucial help thanks to negative constraints to prove the operational soundness of our concolic scheme: a generated test case will indeed follow the intended execution path.

Mesnard et al. (2017) showed that requiring a traditional constraint solver (*i.e.*, a decision procedure for existentially quantified conjunction of atomic constraints) is not enough to decide the constraint selective unification problem (CSUP). Indeed, we presented a CLP instance based on the theory of arrays where we proved that the CSUP is undecidable. Then we showed that assuming variable elimination together with a traditional constraint solver is enough to decide

the CSUP. Of course, a constraint domain with both a traditional constraint solver and a variable elimination algorithm is decidable. But solving the CSUP without variable elimination was an open question in the paper by Mesnard et al. (2017). In the present paper, we have presented a more general approach that can solve the CSUP for decidable constraint domains without variable elimination. CLP($\mathcal{N}$) and CLP($\mathcal{T}erm$) are two such constraint domains.

In turn, Fortz et al. (2020) essentially showed that one could rely on an SMT solver to implement a concolic testing tool for Prolog. The paper is focused on designing a more efficient alternative implementation of contest, as well as trying to avoid the unsoundness of the original approach by Mesnard et al. (2015a). Unfortunately, the ideas in this paper are preliminary and it does not provide any theoretical result. Moreover, it only considers pure logic programs, so even if negative constraints are used during concolic testing, they cannot be represented in the generated test cases.

Finally, one can also find some similarities with an approach proposed by Leuschel and De Schreye (1998) in the context of partial deduction (Lloyd and Shepherdson 1991). In particular, the partial deduction algorithm of Gallagher and Bruynooghe (1991) introduced the use of abstract interpretation based on so-called *characteristic paths* which, roughly speaking, described the deterministic part of the unfolding of an atom. The authors aimed at preserving these characteristic paths when computing resultants and their (most specific) generalisation. However, as noted by Leuschel and De Schreye (1998), this property does not hold, since the generated resultants are sometimes less deterministic than the original rules. In order to overcome this problem, Leuschel and De Schreye (1998) extended the framework of Gallagher and Bruynooghe (1991) to a constraint setting and, moreover, introduce some *pruning* constraints to avoid matching more rules than expected. Although in a different context, this is essentially the same solution that we have proposed in this paper in order to overcome the limitations of Mesnard et al. (2015a).

6 Conclusion and Future Work

In this paper, we have extended concolic testing to CLP. Thanks to the availability of negative constraints, we have formulated and proved a precise operational soundness criteria. Moreover, we have proved that for decidable constraint domains, the selective unification problem is decidable too. Hence, our approach constitutes an excellent basis for designing a powerful concolic testing tool for CLP programs.

For future work, we consider the definition of a post-processing that takes the generated test cases, and further restricts them (if needed) in order to ensure that their execution is always terminating. For this purpose, we may consider the output of some termination analysis for CLP programs. Moreover, we plan to deal with a subset of built-ins in order to cope with practical issues. Finally, we will explore the use of types (as defined in Typed Prolog (Schrijvers et al. 2008) or Mercury (Somogyi et al. 1996)) to further restrict the possible values a variable can take when generating test cases.

Acknowledgements

The authors thank the anonymous reviewers for their many helpful comments and constructive criticisms.

References

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

COMON, H. AND KIRCHNER, C. 1999. Constraint solving on terms. In *Constraints in Computational Logics: Theory and Applications, International Summer School, CCL'99, Revised Lectures*, H. Comon, C. Marché, and R. Treinen, Eds. Lecture Notes in Computer Science, vol. 2002. Springer, 47–103.

FORTZ, S., MESNARD, F., PAYET, É., PERROUIN, G., VANHOOF, W., AND VIDAL, G. 2020. An SMT-based concolic testing tool for logic programs (poster). In *Proc. of the 15th International Symposium on Functional and Logic Languages (FLOPS 2020)*, K. Nakano and K. Sagonas, Eds. Springer LNCS. *To appear*. Extended version at `https://arxiv.org/abs/2002.07115`.

GALLAGHER, J. P. AND BRUYNOOGHE, M. 1991. The derivation of an algorithm for program specialisation. *New Generation Computing 9,* 3/4, 305–334.

GANGE, G., NAVAS, J. A., SCHACHTE, P., SØNDERGAARD, H., AND STUCKEY, P. J. 2015. Horn clauses as an intermediate representation for program analysis and transformation. *Theory and Practice of Logic Programming 15,* 4-5, 526–542.

GIANTSIOS, A., PAPASPYROU, N. S., AND SAGONAS, K. 2015. Concolic testing for functional languages. In *Proc. of the 17th International Symposium on Principles and Practice of Declarative Programming (PPDP 2015)*, M. Falaschi and E. Albert, Eds. ACM, 137–148.

GODEFROID, P., KLARLUND, N., AND SEN, K. 2005. DART: directed automated random testing. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)*, V. Sarkar and M. W. Hall, Eds. ACM, 213–223.

GURFINKEL, A., KAHSAI, T., KOMURAVELLI, A., AND NAVAS, J. A. 2015. The SeaHorn verification framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, D. Kroening and C. S. Pasareanu, Eds. Lecture Notes in Computer Science, vol. 9206. Springer, 343–361.

JAFFAR, J., MAHER, M. J., MARRIOTT, K., AND STUCKEY, P. J. 1998. The semantics of constraint logic programs. *Journal of Logic Programming 37,* 1-3, 1–46.

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

LEUSCHEL, M. AND SCHREYE, D. D. 1998. Constrained partial deduction and the preservation of characteristic trees. *New Generation Computing 16,* 3, 283–342.

LLOYD, J. W. AND SHEPHERDSON, J. C. 1991. Partial evaluation in logic programming. *J. Log. Program. 11,* 3&4, 217–242.

MESNARD, F., PAYET, É., AND VIDAL, G. 2015a. Concolic testing in logic programming. *Theory and Practice of Logic Programming 15,* 4-5, 711–725.

MESNARD, F., PAYET, É., AND VIDAL, G. 2015b. Contest website. URL: `http://kaz.dsic.upv.es/contest.html`.

MESNARD, F., PAYET, É., AND VIDAL, G. 2017. Selective unification in constraint logic programming. In *Proc. of the 19th International Symposium on Principles and Practice of Declarative Programming (PPDP'17)*, W. Vanhoof and B. Pientka, Eds. ACM, 115–126.

MESNARD, F., PAYET, É., AND VIDAL, G. 2020. Concolic Testing in CLP. *CoRR abs/2008.00421*.

PALACIOS, A. AND VIDAL, G. 2015. Concolic execution in functional programming by program instrumentation. In *Proc. of the 25th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2015)*, M. Falaschi, Ed. Lecture Notes in Computer Science, vol. 9527. Springer, 277–292.

SCHRIJVERS, T., COSTA, V. S., WIELEMAKER, J., AND DEMOEN, B. 2008. Towards typed Prolog. In *Proc. of the 24th International Conference on Logic Programming (ICLP'08)*, M. G. de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer, 693–697.

SEN, K. AND AGHA, G. 2006. CUTE and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*, T. Ball and R. B. Jones, Eds. Lecture Notes in Computer Science, vol. 4144. Springer, 419–423.

SEN, K., MARINOV, D., AND AGHA, G. 2005. CUTE: a concolic unit testing engine for C. In *Proc. of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, M. Wermelinger and H. C. Gall, Eds. ACM, 263–272.

SOMOGYI, Z., HENDERSON, F., AND CONWAY, T. C. 1996. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. Log. Program. 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 *Proc. of the 21st International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'11)*, G. Vidal, Ed. Lecture Notes in Computer Science, vol. 7225. Springer, 237–252.

STULOVA, N., MORALES, J. F., AND HERMENEGILDO, M. V. 2014. Assertion-based debugging of higher-order (C)LP programs. In *Proc. of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP 2014)*, O. Chitil, A. King, and O. Danvy, Eds. ACM, 225–235.

VIDAL, G. 2014. Concolic execution and test case generation in Prolog. In *Proc. of the 24th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2014)*, M. Proietti and H. Seki, Eds. Lecture Notes in Computer Science, vol. 8981. Springer, 167–181.