On the Generation of Test Data for Prolog by Partial Evaluation

Miguel Gómez-Zamalloa[1], Elvira Albert[1], and Germán Puebla[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. In recent work, we have proposed an approach to Test Data Generation (TDG) of imperative bytecode by *partial evaluation* (PE) of CLP which consists in two phases: (1) the bytecode program is first transformed into an equivalent CLP program by means of interpretive compilation by PE, (2) a second PE is performed in order to supervise the generation of test-cases by execution of the CLP decompiled program. The main advantages of TDG by PE include flexibility to handle new coverage criteria, the possibility to obtain test-case generators and its simplicity to be implemented. The approach in principle can be directly applied for TDG of any imperative language. However, when one tries to apply it to a declarative language like Prolog, we have found as a main difficulty the generation of test-cases which cover the more complex control flow of Prolog. Essentially, the problem is that an intrinsic feature of PE is that it only computes non-failing derivations while in TDG for Prolog it is essential to generate test-cases associated to failing computations. Basically, we propose to transform the original Prolog program into an equivalent Prolog program with *explicit failure* by partially evaluating a Prolog interpreter which captures failing derivations w.r.t. the input program. Another issue that we discuss in the paper is that, while in the case of bytecode the underlying constraint domain only manipulates integers, in Prolog it should properly handle the symbolic data manipulated by the program. The resulting scheme is of interest for bringing the advantages which are inherent in TDG by PE to the field of logic programming.

1 Introduction

Test data generation (TDG) aims at automatically generating test-cases for interesting test *coverage criteria*. The coverage criteria measure how well the program is exercised by a test suite. Examples of coverage criteria are: *statement coverage* which requires that each line of the code is executed; *path coverage* which requires that every possible trace through a given part of the code is executed; etc. There are a wide variety of approaches to TDG (see [22] for a survey). Our work focuses on *glass-box* testing, where test-cases are obtained from the concrete program in contrast to *black-box* testing, where they are deduced from a specification of the program. Also, our focus is on *static* testing, where we assume no knowledge about the input data, in contrast to *dynamic* approaches [6] which execute the program to be tested for concrete input values.

The standard approach to generating test-cases statically is to perform a *symbolic* execution of the program [18,14,11], where the contents of variables are expressions rather than concrete values. The symbolic execution produces a system of *constraints* consisting of the conditions to execute the different paths. This happens, for instance, in branching instructions, like if-then-else, where we might want to generate test-cases for the two alternative branches and hence accumulate the conditions for each path as constraints. The symbolic execution approach is usually combined with the use of *constraint solvers* in order to: handle the constraints systems by solving the feasibility of paths and, afterwards, to instantiate the input variables.

TDG for declarative languages has received comparatively less attention than for imperative languages. In general, declarative languages pose different problems to testing related to their own execution models, like laziness in functional programming (FP) and failing derivations in constraint logic programming (CLP). The majority of existing tools for FP are based on black-box testing (see e.g. [4]). An exception is [7] where a glass-box testing approach is proposed to generate test-cases for Curry. In the case of CLP, test-cases are obtained for Prolog in [16,3,21]; and very recently for Mercury in [5]. Basically the test-cases are obtained by first computing constraints on the input arguments that correspond to execution paths of logic programs and then solving these constraints to obtain test inputs for such paths.

In recent work [2], we have proposed to employ existing *partial evaluation* (PE) techniques developed for CLP in order to automatically generate *test-case generators* for glass-box testing of bytecode. PE [13] is an automatic program transformation technique which has been traditionally used to specialise programs w.r.t. a known part of its input data and, as Futamura predicted, can also be used to compile programs in a (source) language to another (object) language (see [8]). The approach to TDG by PE of [2] consists of two independent CLP PE phases. (1) First, the bytecode is transformed into an equivalent (decompiled) CLP program by specialising a bytecode interpreter by means of existing PE techniques. (2) A second PE is performed in order to supervise the generation of test-cases by execution of the CLP decompiled program. Interestingly, it is possible to employ control strategies previously defined in the context of CLP PE in order to capture *coverage criteria* for glass-box testing of bytecode. A unique feature of this approach is that, this second PE phase allows generating not only test-cases but also test-case *generators*. Another important advantage is that, in contrast to previous work to TDG of bytecode, it does not require devising a dedicated symbolic virtual machine.

In this work, we study the application of the above approach to TDG by means of PE to the Prolog language. Compared to TDG of an imperative language [2], dealing with Prolog brings in as the main difficulty to generate test-cases associated to failing computations. This happens because an intrinsic feature of PE is that it only produces results associated to the *non-failing* derivations. While this is what we need for TDG of an imperative language (like bytecode above), we now want to capture non-failing derivations in Prolog and

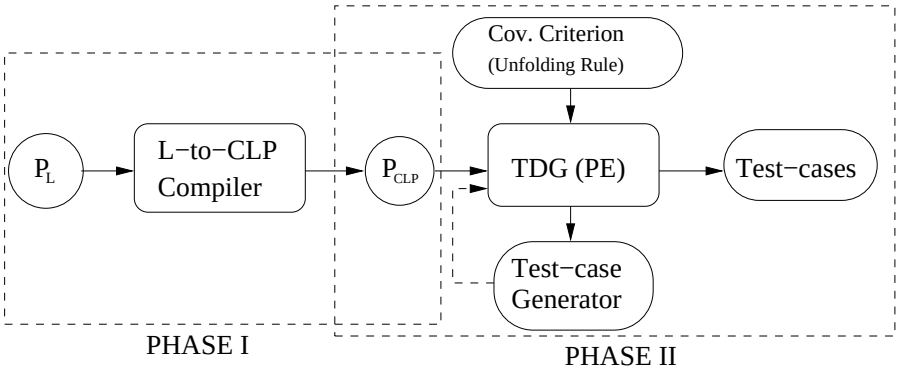

Fig. 1. General scheme of TDG by Partial Evaluation of CLP

still rely on a standard partial evaluator. Our proposal is to transform the original Prolog program into an equivalent Prolog program with explicit failure by partially evaluating a Prolog interpreter which captures failing derivations w.r.t. the input program. This transformation is done in the phase (1) above. As another difference, in the case of bytecode, the underlying constraint domain only manipulates integers. However, the above phase (2) should properly handle the data manipulated by the program in the case of Prolog. Compared to existing approaches to TDG of Prolog [3,16], our approach basically is of interest for bringing the advantages which are inherent in TDG by PE to the field of Prolog:

(i) It is *more powerful* in that we can produce test-case generators which are CLP programs whose execution in CLP returns further test-cases on demand without the need to start the TDG process from scratch;
(ii) It is more *flexible*, as different coverage criteria can be easily incorporated to our framework just by adding the appropriate local control to the partial evaluator.
(iii) It is *simpler* to implement compared to the development of a dedicated test-case generator, as long as a CLP partial evaluator is available.

The rest of the paper is organized as follows. In the next section, we give some basics on PE of logic programs and describe in detail the approach to TDG by PE proposed in [2]. Sect. 3 discusses some fundamental issues like the Prolog control-flow and the notion of computation path. Then, Sect. 4 describes the program transformation to make failure explicit, Sect. 5 outlines existing methods to properly handle symbolic data during the TDG phase, and finally Sect. 6 concludes and discusses some ideas for future work.

2 Basics of TDG by Partial Evaluation

In this section we recall the basics of partial evaluation of logic programming and summarize the general approach of relying on partial evaluation of CLP for TDG of an imperative language, as proposed in [2].

2.1 Partial Evaluation and its Application to Compilation

We assume familiarity with basic notions of logic programming and partial evaluation (see e.g. [9]). Partial evaluation is a semantics-based program transformation technique which specialises a program w.r.t. given input data, hence, it is often called *program specialisation*. Essentially, partial evaluators are non-standard interpreters which evaluate goals as long as termination is guaranteed and specialisation is considered profitable. In logic programming, the underlying technique is to construct (possibly) *incomplete* SLD trees for the set of atoms to be specialised. In an incomplete tree, it is possible to choose *not* to further unfold a goal. Therefore, the tree may contain three kinds of leaves: failure nodes, success nodes (which contain the empty goal), and non-empty goals which are not further unfolded. The latter are required in order to guarantee termination of the partial evaluation process, since the SLD being built may be infinite. Even if the SLD trees for fully instantiated initial atoms (as regards the *input* arguments) are finite, the SLD trees produced for partially instantiated initial atoms may be infinite. This is because the SLD for partially instantiated atoms can have (infinitely many) more branches than the actual SLD tree at run-time.

The role of the *local control* is to determine how to construct the (incomplete) SLD trees. In particular, the *unfolding rule* decides, for each resolvent, whether to stop unfolding or to continue unfolding it and, if so, which atom to select from the resolvent. On the other hand, partial evaluators need to compute SLD-trees for a number of atoms in order to ensure that all atoms which appear in non-failing leaves of incomplete SLD trees are "covered" by the root of some tree (this is known as the closedness condition of partial evaluation [9]). The role of the *global control* is to ensure that we do not try to compute SLD trees for an infinite number of atoms. The usual way of achieving this is by applying an *abstraction operator* which performs "generalizations" on the atoms for which SLD trees are to be built. The global control returns a set of atoms T. Finally, the partial evaluation can then be systematically extracted from the set T (see [9] for details).

Traditionally, there have been two different approaches regarding the way in which control decisions are taken, *on-line* and *off-line* approaches. In *online* PE, all control decisions are dynamically taken during the specialisation phase. In *offline* PE, a set of previously computed annotations (often manually provided) gives information to the control operators to decide, 1) when to stop unfolding (*memoise*) in the local control, and 2) how to perform generalizations in the global control.

The development of PE techniques has allowed the so-called "interpretative approach" to compilation which consists in specialising an interpreter w.r.t. a fixed object code. Interpretive compilation was proposed in Futamura's seminal work [8], whereby compilation of a program P written in a (*source*) programming language L_S into another (*object*) programming language L_O is achieved by partially evaluating an interpreter for L_S written in L_O w.r.t. P. The advantages of interpretive (de-)compilation w.r.t. dedicated (de-)compilers are well-known and discussed in the PE literature (see, e.g., [1]). Very briefly, they include: *flexibility*,

it is easier to modify the interpreter in order to tune the decompilation (e.g., observe new properties of interest); *easier to trust*, it is more difficult to prove that ad-hoc decompilers preserve the program semantics; *easier to maintain*, new changes in the language semantics can be easily reflected in the interpreter.

2.2 A General Scheme to TDG of Imperative Languages by PE

In recent work, we have proposed an approach to Test Data Generation (TDG) by PE of CLP [2] and used it for TDG of bytecode. The approach is generic in that the same techniques can be applied to TDG other both low and high-level imperative languages. In Figure 1 we overview the main two phases of this technique. In **Phase I**, the input program written in some (imperative) language L is compiled into an equivalent CLP program P_{CLP}. This compilation can be achieved by means of an ad-hoc decompiler (e.g., an ad-hoc decompiler of bytecode to Prolog [17]) or, more interestingly, can be achieved automatically by relying on the first Futamura projection by means of PE for logic programs as explained above (e.g., [12,1,10]).

Now, the aim of **Phase II** is to generate test-cases which traverse as many different execution paths of P_L as possible, according to a given coverage criteria. From this perspective, different test data will correspond to different execution paths. With this aim, rather than executing the program starting from different input values, the standard approach consists in performing *symbolic execution* such that a single symbolic run captures the behavior of (infinitely) many input values. The central idea in symbolic execution is to use constraint variables instead of actual input values and to capture the effects of computation using constraints. Hence, the compilation from L to CLP allows us to use the standard CLP execution mechanism to carry out this phase. In particular, by running the P_{CLP} program without input values, each successful execution corresponds to a different computation path in P_L.

Rather than relying on the standard execution mechanism, we have proposed in [2] to use PE of CLP to carry out **Phase II**. Essentially, we can rely on a CLP partial evaluator which is able to solve the constraint system, in much the same way as a symbolic abstract machine would do. Note that performing symbolic execution for TDG consists in building a finite (possibly unfinished) evaluation tree by using a non-standard execution strategy which ensures both a certain coverage criterion and termination. This is exactly the problem that *unfolding rules*, used in partial evaluators of (C)LP, solve. In essence, partial evaluators are non-standard interpreters which receive a set of partially instantiated atoms and evaluate them as determined by the so-called unfolding rule. Thus, the role of the unfolding rule is to supervise the process of building finite (possibly unfinished) SLD trees for the atoms. This view of TDG as a PE problem has important advantages. First, we can directly apply existing, powerful, unfolding rules developed in the context of PE. Second, it is possible to explore additional abilities of partial evaluators in the context of TDG. In particular, the generation of a residual program from the evaluation tree returns a program which can be used as a *test-case generator*, i.e., a CLP program whose execution in CLP

returns further test-cases on demand without the need to start the TDG process from scratch. In the rest of the paper, we study the application of this general approach to TDG of Prolog programs.

3 Computation Paths for Test Data Generation of Prolog

As we have already mentioned, test data generation is about producing test-cases which traverse as many different execution paths as possible. From this perspective, different test data should correspond to different execution paths. Thus, a main concern is to specify the computation paths for which we will produce test-cases. This requires first to determine the control flow of the considered language. In this section, we aim at defining the control flow of Prolog programs that we will use for TDG. Test data will be generated for the computation paths in the control flow.

3.1 The Control Flow of Prolog

As usual a Prolog program consists of a set of predicates, where each predicate is defined as a sequence of clauses of the form H :- $B_1, \ldots, B_m$ with $m \geq 0$. A predicate is univocally determined by its *predicate signature* p/n, being p the name of the predicate and n its arity. Throughout the rest of the paper we will consider Prolog programs with the following features:

- Rules are normalized, i.e., arguments in the head of the rule are distinct variables. The corresponding bindings will appear explicitly in the body as unifications.
- Atoms appearing in the bodies of rules can be: unifications (considered as builtins), calls to defined predicates, term checking builtins (`==/2`, `\==/2`, etc), and arithmetic builtins (`is/2`, `</2`, `=</2`, etc). Other typical Prolog builtins like `fail/0`, `!/0`, `if/3`, etc, have been deliberately left out to simplify the presentation.
- All predicates must be moded and well-typed. We will assume the existence of a "`:- pred`" declaration associated with each predicate specifying the type expected for each argument (see as example the declarations in Fig. 2). Note that this assumption is sensible in the context of TDG (as the aim is the automatic generation of test *input*). Also, it should not be a limitation as analyses that can automatically infer this information exist.

The control flow in Prolog programs is significantly more complex than in traditional imperative languages. The declarative semantics of Prolog implies some additional features like: 1) several forms of backtracking, induced by the failure of a sub-goal, or by non-deterministic predicates; or 2) forced control flow change by the predicate "cut". Traditionally, control-flow graphs (CFGs for short) are used to statically represent the control-flow of programs. Typically, in a CFG, nodes are blocks containing a set of sequential instructions, and edges represent the flows that the program can follow w.r.t. the semantics of the corresponding

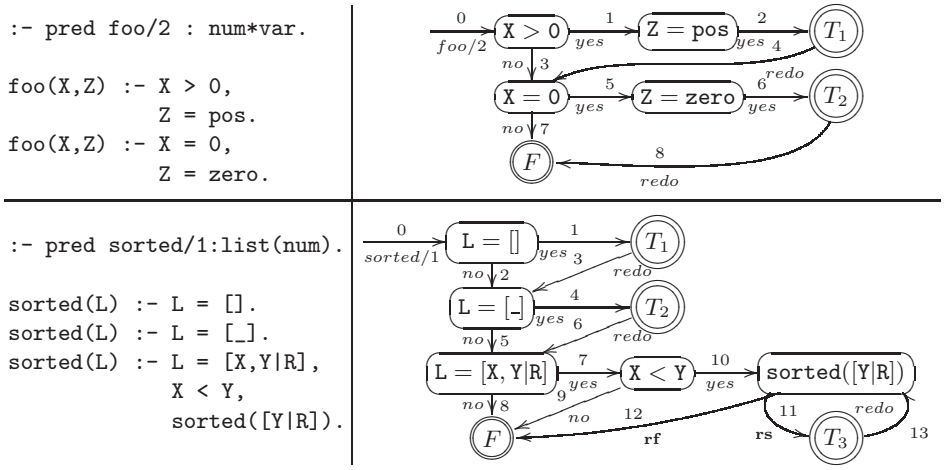

Fig. 2. Working example. Prolog code and CFGs.

programming language. In the literature, CFGs for Prolog (and Mercury) have been used for the aim of TDG in [16,21] ([5] for Mercury). In particular, CFGs determine the computation paths for which test-cases will be produced. Our framework relies on the CFGs of [16,21] which are known as *p-flowgraph*'s.[3] As will be explained later, there are some differences between these CFGs and the ones in [5] which lead to different test-cases.

Figure 2 depicts the Prolog code together with the corresponding CFGs for predicates `foo/2` and `sorted/1`. Predicate `foo/2`, given a number in its first argument, returns, in the second one, the value `pos` if the number is positive and `zero` if it is zero. If the number is negative, it just fails. Predicate `sorted/1`, given a list of numbers, checks whether the list is strictly sorted, in that case it succeeds, otherwise it fails. The CFGs contain the following nodes:

- a non-terminal node associated to each atom in the body of each clause,
- a set of terminal nodes "T_i" representing the success of the i-th clause, and
- the terminal node "F" to represent failure.

As regards edges, in principle all non-terminal nodes have two output flows, corresponding to the cases where the builtin or predicate call succeeds or fails respectively. They are labeled as "yes" or "no" for builtins (including unifications), and as "**rs**" (*return-after-success*) or "**rf**" (*return-after-failure*) for predicate calls. There is an exception in the case of unifications where one of the arguments is a variable, in which case the unification cannot fail. This can be known statically by using the mode information. See for example nodes "`Z=pos`"

[3] The difference with the CFGs in [16,21] is that they consider one additional node per clause to explicitly represent the unification of the head of the rule. This is not needed in our case since predicates are normalised.

and "`Z=zero`" in the `foo/2` CFG. Both "yes" and "**rs**" edges point to the node representing the next atom in the clause or to the corresponding "T_i" node if the atom is the last one. Finally, each "T_i" node has an output edge labeled as "redo" to represent the case in which the predicate is asked for more solutions. All "no", "**rf**" and "redo" edges point either to the node corresponding to the first previous non-deterministic call in the same clause, or the first node of the following clause, or the "F" node if no node meets the above conditions. See as an example the "**rs**" and "**rf**" edges from the non-terminal node for `sorted([Y|R])`.

3.2 Generating Test Data for Computation Paths

In order to define the computation paths determined by the CFGs, every edge in every CFG is labeled with a unique natural number. An special edge labeled with "0" and p/n represents the entry of predicate p/n.

Definition 1 (Computation sub-path). *Given the CFG for predicate P, a* computation sub-path *is a sequence of numeric labels (natural numbers) $\langle l_1, \ldots, l_n \rangle$ s.t.:*

- *l_1 corresponds to either an entry, an "**rs**", an "**rf**" or a "redo" edge,*
- *l_n leads to a terminal node or to a predicate call, and*
- *for all consecutive labels l_i, l_j, there exists a node corresponding to a builtin in the CFG of P, for which l_i is an input flow and l_j is an output flow.*

Definition 2 (Computation path). *Given the CFGs corresponding to the set of predicates defining a program, a* computation path *(CP for short) for predicate p is a concatenation $sp_1 \cdots sp_m$ ($m \geq 1$) of computation sub-paths such that:*

- *First label in sp_1 is either 0, in which case we say it is a* full *CP, or corresponds to a "redo" edge, in which case we say it is a* partial *CP (PCP for short).*
- *Last label in sp_m leads to a terminal node in the CFG of p. If it is a T node the CP is said to be* successful *otherwise it is called* failing.
- *For all sp_k whose last label leads to a node corresponding to a predicate call, $cp = sp_{k+1} \cdots sp_j$, $j > k$ is a CP for the called predicate, and:*
  - *if cp is successful then the first label in sp_{j+1} corresponds to an "**rs**" edge,*
  - *otherwise (cp is failing), it corresponds to an **rf** edge.*
- *For all sp_k whose first label corresponds to a "redo" edge flowing from a "T_a" node in the CFG of predicate q, $\exists sp_j$, $j < k$, whose first label corresponds either to an entry edge or to a "redo" edge flowing from "T_b", $b < a$, of the CFG of q.*

If a CP contains at least one label corresponding to a "redo" flow, then the CP is said to be an after-retry *CP. The rest of the CPs are* first-try *CPs.*

For example in `foo/2`, p_1=$\langle 0,1,2\rangle$ and p_2=$\langle 0,3,5,6\rangle$ are first-try successful CPs; p_3=$\langle 0,3,7\rangle$ is a first-try failing branch; p_4=$\langle 0,1,2\rangle \cdot \langle 4,5,6\rangle$ is an after-retry successful CP (although this one is unfeasible as $X > 0$ and $X = 0$ are disjoint conditions), and p_5=$\langle 0,1,2\rangle \cdot \langle 4,7\rangle$ is an after-retry failing branch. In `sorted/1`, p_6=$\langle 0,2,5,7,10\rangle \cdot \langle 0,2,4\rangle \cdot \langle 11\rangle$ is a first-try successful CP and p_7=$\langle 0,2,5,7,10\rangle \cdot \langle 0,2,5,7,9\rangle \cdot \langle 12\rangle$ is a first-try failing CP. It is interesting to observe the correspondence between the CPs and the test data that make the program traverse them. In `foo/2`, p_1 is followed by goal `foo(1,Z)`, p_2 by goal `foo(0,Z)`, p_3 by `foo(-1,Z)`, p_4 is an unfeasible path, and p_5 is followed by `foo(0,Z)` when we ask for more solutions. As regards `sorted/1`, p_6 is followed by the goal `sorted([0,1])` and p_7 by `sorted([0,1,0])`. As we will see in Sect. 5, these will become part of the test-cases that we automatically infer.

A key feature of our CFGs is that they make explicit the fact that after failing with a clause the computation has to re-try with the following clause, unless a non-deterministic call is left behind. E.g., in `foo/2` the CFG makes explicit that the only way to get a first-try failing branch is through the CP $\langle 0,3,7\rangle$, hence traversing, and failing in, both conditions $X > 0$ and $X = 0$. Therefore, a test data to obtain such a behavior will be a negative number for argument X. Other approaches, like the one in [5], do not handle flows after failure in the same way. In fact, in [5], edge "3" in `foo/2` goes directly to node "F". It is not clear if these approaches are able to obtain such a test data. As another difference with previous approaches to TDG of Prolog, we want to highlight that we use CFGs just to reason about the program transformation that will be presented in the following section and, in particular, to clarify which features we want to capture. However, in previous approaches, test-cases are deduced directly from the CFGs.

4 A Program Transformation to Make Failure Explicit

As we outlined in Sect. 1, an intrinsic feature of the second phase of our approach is that it can only produce results associated to non-failing derivations. This is the main reason why the general approach to TDG by PE sketched in Sect. 2 is directly applicable only to TDG of imperative languages. To enable its application to Prolog, we propose a program transformation which makes failure explicit in the Prolog program. The specialisation of meta-programs has been proved to have a large number of interesting applications[9]. Futamura projection's to derive compiled code, compilers and compiler generators fall into this category. The specialization of meta-interpreters for non-standard computation rules has also been studied. Furthermore, language extensions and enhancements can be easily expressed as meta-interpreters which perform additional operations to the standard computation. In short, program specialisation offers a general compilation technique for the wide variety of procedural interpretations of logic programs. Among them, we propose to carry out our transformation which makes failure in logic programs explicit by partially evaluating a Prolog meta-interpreter which captures failing derivations w.r.t. the original program. First, in Sect. 4.1 we describe such a meta-interpreter emphasizing the Prolog control features which

we want to capture. Then, Sect. 4.2 describes the control strategies which have to be used in PE in order to produce an effective transformation.

4.1 A Prolog Meta-Interpreter to Capture Failure

Given a Prolog program and given a goal, our aim is to define an interpreter in which the computation of the program and goal produces the same results as the ones obtained by using the standard Prolog computation but with the difference that failure is never reported. Instead, an additional argument *Answer* will be bound to the value "yes", if the computation corresponds to a successful derivation, and to "no" if it corresponds to a failing derivation. Predicate `solve/4` is the main predicate of our meta-interpreter whose first and second arguments are the predicate signature and arguments of the goal to be executed; and its third argument is the answer; by now we ignore the last argument. For instance, the call `solve(foo/2,[0,Z],Answer,_)` succeeds with `Z` = `zero` and `Answer` = `yes`, and `solve(foo/2,[-1,Z],Answer,_)` also succeeds, but with `Answer` = `no`. The interpreter has to handle the following issues:

1. The Prolog *backtracking* mechanism has to be explicitly implemented. To this aim, a stack of *choice points* is carried along during the computation so that:
   - if the derivation fails: (1) when the stack is empty, it ends up with success and returns the value "no", (2) otherwise, the computation is resumed from the last choice point, if any;
   - if it successfully ends: (1) when the stack is empty, the computation finishes with answer "yes", (2) otherwise, the computation is resumed from the last choice point.
2. When backtracking occurs, all variable bindings, between the current point and the choice point to resume from, have to be undone.
3. The interpreter has to be implemented in a *big-step* fashion. This is a requirement for obtaining an effective decompilation. More details are given in Sect. 4.2.

Figure 3 shows an implementation of a meta-interpreter which handles the above issues. The fourth argument of the main predicate `solve/4`, named `TNCPS`, contains upon success the total number of choice points not yet considered, whose role will be explained later. The interpreter assumes that the program is represented as a set of `pred/2` and `clause/3` facts. There is a `pred/2` fact per predicate providing its predicate signature, number of clauses and mode information; and a `clause/3` fact per clause providing the actual code and clause identifier. Predicate `solve/4` basically builds an initial state on `S0`, by calling `build_s0/4`, and then delegates on `exec/3` to obtain the final state `Sf` of the computation. The output information, `OutVs`, is taken from `Sf`. The state carried along is of the form `st(PP,G,CPs,OutVs,Ans,NCPs)`, where `PP` is the current program point, `G` the current goal, `CPs` is the stack of choice points (list of program points), `OutVs` the list of variables in `G` corresponding to the output

```
solve(P/Ar,Args,Answer,TNCPs) :-
   pred(P/Ar,_),
   build_s0(P/Ar,Args,S0,OutVs),
   exec(Args,S0,Sf),
   Sf = st(_,_,_,OutVs',Answer,TNCPs/_),
   OutVs' = OutVs.

exec(_,S,Sf) :-
   S = st(_,[],[],OutVs,yes,NCPs),
   Sf = st(_,_,_,OutVs,yes,NCPs).
exec(_,S,Sf) :-
   S = st(_,[],[_|_],OutVs,yes,NCPs),
   Sf = st(_,_,_,OutVs,yes,NCPs).
exec(_,S,Sf) :-
   S = st(_,_,[],OutVs,no,TNCPs/0),
   Sf = st(_,_,_,OutVs,no,TNCPs/0).
exec(Args,S,Sf) :-
   S = st(_,[],[CP|CPs],_,yes,TNCPs/0),
   build_retry_state(Args,CP,CPs,TNCPs,S'),
   exec(Args,S',Sf).
exec(Args,S,Sf) :-
   S = st(_,_,[CP|CPs],_,no,TNCPs/0),
   build_retry_state(Args,CP,CPs,TNCPs,S'),
   exec(Args,S',Sf).

exec(Args,S,Sf) :-
   S = st(PP,[A|As],CPs,OutVs,yes,TNCPs/ENCPs),
   PP = pp(P/Ar,ClId,Pt),
   internal(A),
   functor(A,A_f,A_ar),
   A =..[A_f|A_args],
   next(Pt,Pt'),
   solve(A_f/A_ar,A_args,Ans,ENCPs'),
   TNCPs' is TNCPs + ENCPs',
   ENCPs'' is ENCPs + ENCPs',
   PP' = pp(P/Ar,ClId,Pt'),
   S' = st(PP',As,CPs,OutVs,Ans,TNCPs'/ENCPs''),
   exec(Args,S',Sf).
exec(Args,S,Sf) :-
   S = st(PP,[A|As],CPs,OutVs,yes,NCPs),
   PP = pp(P/Ar,ClId,Pt),
   builtin(A),
   next(Pt,Pt'),
   run_builtin(PP,A,Ans),
   PP' = pp(P/Ar,ClId,Pt'),
   S' = st(PP',As,CPs,OutVs,Ans,NCPs),
   exec(Args,S',Sf).
```

Fig. 3. Code of Prolog meta-interpreter to capture failure

parameters of the original goal, `Ans` the current answer ("yes" or "no") and `NCPs` the number of choice points left behind. A program point is of the form `pp(P/Ar,ClId,Pt)`, where `P/Ar`, `ClId` and `Pt` are the predicate signature, the clause identifier and the program point of the clause at hand. Predicate `exec/3` implements the main loop of the interpreter. Given the current state in its second argument it produces the final state of the computation in the third one. It is defined by the seven clauses which are applied in they following situations:

$1^{st}cl$. *The current goal is empty, the answer "yes" and there are no pending choice points.* Then, the computation finishes with answer "yes". The current answer is actually used as a flag to indicate whether the previous step in the computation succeeded or failed (see the last two `exec/3` clauses).

$2^{nd}cl$. *As* $1^{st}cl$. *but having at least one choice point.* This clause represents the solution in which the computation ends. The 4^{th} clause takes the other alternatives.

$3^{rd}cl$. *The previous step failed and there are no pending choice points.* Then, the computation ends with answer "no".

$4^{th}cl$. *The current goal is empty, the answer "yes" and there is at least one pending choice point.* This is the same situation as in the 2^{nd} clause, however in this case the alternative of resuming from the last choice point is taken. The corresponding state `S'` is built by means of `build_retry_state/5` and the computation is resumed from `S'` by recursively calling `exec/3`.

$5^{th}cl$. *The previous step failed and there is at least one pending choice point.* Then, the computation is resumed from the last choice point in the same way as in the previous clause.

$6^{th}cl$. *The first atom to be solved is user-defined.* A call to `solve/4` handles the atom, and the computation proceeds with the next program point of the same clause which was the current one before calling `solve/4`. This way of solving a predicate call makes the interpreter *big-step* (issue (3) above).

$7^{th}cl$. *The first atom to be solved is a builtin.* Then, `run_builtin/3` produces the corresponding answer, and the computation proceeds with the following program point. An interesting observation (also applicable for the previous clause) is that the answer obtained from `run_builtin/3` (or `solve/4`) is now set up as the answer of the next state. This will make the computation go through the 3^{rd} or 5^{th} clauses in the following step, if the obtained answer was "no".

The correspondence between these clauses and the flows in the CFGs is as follows: clauses 1^{st}, 2^{nd} and 4^{th} represent the output edges from every "T" node. Clause 3^{rd} represents the "no" edges to "F" nodes and 5^{th} the "no" edges to non-terminal nodes. Finally clauses 6^{th} and 7^{th} represents the execution of builtins and predicate calls in non-terminal nodes and their corresponding "yes" edges.

Let us now explain how the interpreter handles the above three issues. To handle (1), a stack of choice points is carried along within the state, initialised to contain all initial program points of each clause defining the predicate to be solved, except for the first one. E.g., the initial stack of choice points for `sorted/1` is `[pp(sorted/1,2,1),pp(sorted/1,3,1)]`. How this stack is used to perform the backtracking is already explained in the description of the 4^{th} and 5^{th} `exec/3` clauses above. As regards issue (2), a quite simple way to implement this in Prolog is to produce the necessary fresh variables every time the computation is resumed. This is done inside `build_retry_state/5`. The corresponding unification to link the fresh variables with the original goal variables is made at the end (see last line of `solve/4`). This is the reason why 1) the list of the actual variables used in the current goal needs to be carried along within the state; and 2) the original arguments are carried along as the first argument of `exec/3`, as the original ground arguments provided, have to be used when resuming from a choice point.

Finally, it is worth mentioning that `solve/4` does not return the actual stack of choice points but only the number of them. This means that during a computation the interpreter only considers choice points of the predicate being solved. The question is then, how can the interpreter backtrack to the last choice point, including those induced by other computations of `solve/4`? E.g., how can the interpreter follow edge "13" in the CFG of `sorted/1`? The interpreter performs the backtracking in the following way: 1) The total number of choice points left behind, `TNCPs`, is carried along within the state and finally returned in the last argument of `solve/4`. 2) The number of choice points corresponding to invoked predicates, `ENCPs`, is also carried along. It is updated right after the call to `solve/4` in the 6^{th} clause of `exec/3`. Both numbers are stored in the last argument of the state as `TNCPs/ENCPs`. 3) Execution is resumed from choice points of the current predicate only if `ENCPs` $= 0$, as it can be seen in the 4^{th} and 5^{th} clauses. Otherwise, the computation just fails and Prolog's backtracking

mechanism is used to ask the last invoked predicate for more solutions. This indeed means that the non-determinism of the program is still implicit.

4.2 Controlling Partial Evaluation

The specialisation of interpreters has been studied in many different contexts, see e.g. [9,10,19]. Very recently, [10] proposed control strategies to successfully specialise low-level code interpreters w.r.t. non trivial programs. Here we demonstrate how such guidelines can be, and should be, used in the specialisation of non-trivial Prolog meta-interpreters. They include:

1. *Big-step* interpreter. This solves the problem of handling recursion (see [10]) and enables a compositional specialisation w.r.t. the program procedures (or predicates). Note that an effective treatment of recursion is specially important in Prolog programs where recursion is heavily used.
2. *Optimality* issues. Optimality must ensure that: a) the code to be transformed is traversed exactly once, and b) residual code is emitted once in the transformed program. To achieve optimality, during unfolding, all atoms corresponding with *divergence* or *convergence points* in the CFG of the program to be transformed, has to be *memoised* (see Sect. 2.1). A divergence (convergence) point is a program point from (to) which two or more flows originate (converge).

We already explained that the interpreter in Fig. 3 is big-step. As regards optimality, by looking at the CFGs of Fig. 2, we can observe: 1) all program points are divergence points except those corresponding with unifications in which one argument is a variable, and 2) the first program point of every clause, except for the one of the first clause, is a convergence point. We assume that conv_points(P) and div_points(P) denote, respectively, the set of convergence points and divergence points of a predicate P. We follow the syntax of [10] for PE annotations. An annotation is of the form "[*Precond*] $\Rightarrow$ *Ann Pred*" where *Precond* is an optional precondition defined as a logic formula, *Ann* is the kind of annotation (only **memo** in this case), and *Pred* is a predicate descriptor, i.e., a predicate function and distinct free variables. Then, to achieve an effective transformation, we specialise the interpreter in Fig. 3 w.r.t. the program to be transformed by using the following annotation for each predicate `P/Ar` in the program:

$$\texttt{PP} \in \mathsf{div\_points}(\texttt{P/Ar}) \cup \mathsf{conv\_points}(\texttt{P/Ar}) \Rightarrow \textbf{memo}\ \texttt{exec}(\_, \texttt{st}(\texttt{PP}, \_, \_, \_, \_, \_), \_)$$

Additionally `solve/4` and `run_builtin/3` are also annotated to be memoised always to avoid code duplications.

This already describes how the specialisation has to be steered in the local control. As regards the global control, the only predicate which can introduce non-termination is `exec/3`. Its first and third arguments contain a fixed structure with variables. The second one might be problematic as it ranges over the set of all computable states at specialisation time. Note that the number of computable states remains finite thanks to the big-step nature of the interpreter. Still, it can

```
solve(foo/2,[C,D],A,B) :-
   run_builtin_1(E,C),
   exec_1(C,E,F,A,B), F = [D].

exec_1(A,no,F,G,H) :- exec_2(A,F,G,H).
exec_1(_,yes,[pos],yes,1).
exec_1(A,yes,F,G,H) :- exec_2(A,F,G,H).

exec_2(A,G,H,I) :-
   run_builtin_2(K,A), exec_3(K,G,H,I).

exec_3(no,[_],no,0).
exec_3(yes,[zero],yes,0).

run_builtin_1(yes,A) :- A#>0.
run_builtin_1(no,A) :- \+ A#>0.

run_builtin_2(yes,A) :- A#=0.
run_builtin_2(no,A) :- \+ A#=0.
```

Fig. 4. Transformed code with explicit failure for `foo/2`

happen that the same program point is reached with different values for the `NCPs` sub-term of the state. Therefore, if one wants to achieve the optimality criterion above, such argument has to be always generalised in global control.

Example 1. Figure 4 depicts the transformed code we obtain for predicate `foo/2`. It can be observed that there is a clear correspondence between the transformed code and the CFG in Fig. 2. Thus, predicate `solve/4` represents the node "`X>0`", `exec_1/5` implements its continuation, whose three clauses correspond to the three sub-paths $\langle 3 \rangle$, $\langle 1, 2 \rangle$ and $\langle 1, 2, 4 \rangle$ respectively. Predicate `exec_2/4` represents the node "`X=0`" and `exec_3/5` implements its continuation, whose two clauses correspond to the sub-paths $\langle 7 \rangle$ and $\langle 5, 6 \rangle$. Note that edge "8" is not considered in the meta-interpreter (nor in the transformed program) as it is meaningless for TDG. It is worth mentioning that the transformed program captures the way in which variable bindings are undone. For instance in `solve(foo/2,[C,D],...)`, if we keep track of variables `C` and `D`, it can be seen that `D`, which corresponds to variable `Z` in the original code, is only used for the final unification `F=[D]`, while new fresh variables are used for the unifications with `pos` and `zero`. However, variable `C`, which corresponds to variable `X` in the original code, is actually used for the checks in `run_builtin_1/2` and `run_builtin_2/2`. This turns out to be fundamental when trying to obtain test data associated to the *first-try failing* CP $\langle 0, 3, 7 \rangle$. It must be the same variable the one which, at the same time, is not "> 0" and not "$=0$". Otherwise we cannot obtain a negative number as test data for such CP. Finally, observe that the original Prolog arithmetic builtins have been (automatically) transformed into their `clpfd` counterparts [4].

5 Generating Test Cases by Partial Evaluation

Once the original Prolog program has been transformed into an equivalent Prolog program with explicit failure, we can use the approach of [2] to carry out **phase**

[4] We are using the `clpfd` library of `Sicstus` Prolog. See [20] for details.

II (see Fig. 1) and generate test data both for successful and failing derivations. As we have explained in Sect. 2.2, the idea is to perform a second PE over the CLP transformed program where the unfolding rule plays the role of the coverage criterion. In [2] an unfolding rule implementing the *block-count(k)* coverage criterion was proposed. A set of computation paths satisfies the *block-count*(k) *criterion* if it includes all terminating computation paths which can be built in which the number of times each block is visited does not exceed the given k. The blocks the criterion refers to are the blocks or nodes in the CFGs of the original Prolog program. As the only form of loops in Prolog are recursive calls, the "k" in the *block-count*(k) actually corresponds to the number of recursive calls which are allowed.

Unfortunately, the presence of Prolog's negation in our transformed programs complicates this phase. The negation will appear in the transformed program for "no" branches originating from nodes corresponding to a (possibly) failing builtin. See for example predicates `run_builtin_1/3` and `run_builtin_2/3` in the transformed code of `foo/2` in Fig. 4. While Prolog's negation works well for ground arguments, it gives no information for free variables, as it is required in the evaluation performed during this TDG phase. In particular, in the `foo/2` example, given the computation which traverses the calls "`\+ A#>0`" and "`\+ A#=0`" (corresponding to the path $\langle 0, 3, 7\rangle$ in the CFG), we need to infer that "`A<0`". In other words, we need somehow to turn the *negative* information into *positive* information. This transformation is straightforward for arithmetic builtins: we just have to replace "`\+` e_1`#=`e_2" by "e_1`#\=`e_2" and "`\+` e_1`#>`e_2" by "e_1`#=<`e_2", etc.

Example 2. This transformation allows us to obtain the following set of test-cases for `foo/2`:

$$\left\{\begin{array}{l}\langle\texttt{[1],[pos],yes/first-try}\rangle,\ \ \langle\texttt{[1],[\_],no/after-retry}\rangle,\\ \langle\texttt{[0],[zero],yes/first-try}\rangle,\ \langle\texttt{[-100],[\_],no/first-retry}\rangle\end{array}\right\}$$

They correspond respectively (reading by rows) to the CPs $\langle 0, 1, 2\rangle$, $\langle 0, 1, 2\rangle \cdot \langle 4, 7\rangle$, $\langle 0, 3, 5, 6\rangle$ and $\langle 0, 3, 7\rangle$. Each test-case is represented as a 3-tuple $\langle Ins, Outs, Ans\rangle$ being Ins the list of input arguments, $Outs$ the list of output arguments and Ans the answer. The answer takes the form A/B with $A \in \{$`yes,no`$\}$ and $B \in \{$`first-try,after-retry`$\}$[5], so that we obtain sufficient information about the kind of CP to which the test-case corresponds (see Sect. 3). As there are no recursive calls in `foo/2` such test-cases are obtained using the *block-count*(k) criterion for any k (greater than 0). The domain used for the integer number is $\{-100..100\}$.

However, it can be the case that negation involves unifications with symbolic data. For example, the transformed code for `sorted/1` includes the negations "`\+ L=[]`" and "`\+ L=[_|_]`". As before, we might write transformations for the negated unifications involving lists, so that at the end it is inferred that "

[5] To simplify the presentation in Sect. 4.1, we decided not include in the interpreter the support to calculate the `first-try/after-retry` value.

`L=[_,_|_]`". However this would be too an ad-hoc solution as many distinct term structures, different from lists, can appear on negated unifications. A solution for this problem has been recently proposed for Mercury in the same context [5]. It roughly consists in the following: 1) It is assumed that each predicate argument is well-typed. 2) A domain is initialised for each variable, containing the set of possible functors the variable can take. 3) When a negated unification involving an output variable is found (in their terminology a negated *decomposition*), the corresponding functor is removed from the variable domain. It is crucial at this point the assumption that complex unifications are broken down into simple ones. 4) Finally, a search algorithm is described to generate particular values from the type definition and final domain for the variable. The technique is implemented using CHR and can be directly used in principle for our purposes as well.

On the other hand, advanced declarative languages like TOY [15] make possible the co-existence of different constraint domains. In particular, the co-existence of boolean and numeric constraint domains enables the possibility of using *disequalities* involving both symbolic data and numbers. This allows for example expressing the negated unifications "`\+ L=[]`" and "`\+ L=[_|_]`" as disequality constraints " `L/=[]`" and " `L/=[_|_]`". Additionally, by relying on the boolean constraint solver, the negated arithmetic builtins "`\+ A#>0`" and "`\+ A#=0`" can be encoded as "`(A#>0) == false`" and "`(A#=0) == false`". This is in principle a more general solution that we want to explore, although a thorough experimental evaluation needs to be carried out to demonstrate its applicability to our particular context.

Example 3. Now, by using any of the techniques outlined above, we obtain the following set of test-cases for `sorted/1`, using *block-count*(2) as the coverage criterion:

$$\left\{\begin{array}{ll}\langle\texttt{[[]],[],yes/first-try}\rangle, & \langle\texttt{[[0]],[],yes/first-try}\rangle,\\ \langle\texttt{[[0,1]],[],yes/first-try}\rangle, & \langle\texttt{[[0,1,2]],[],yes/first-try}\rangle,\\ \langle\texttt{[[0,1,2,0|\_]],[],no/first-try}\rangle, & \langle\texttt{[[0,1,0|\_]],[],no/first-try}\rangle,\\ \langle\texttt{[[0,0|\_]],[],no/first-try}\rangle & \end{array}\right\}$$

They correspond respectively (reading by rows) to the CPs "$\langle 0,1\rangle$", "$\langle 0,2,4\rangle$", "$\langle 0,2,5,7,10\rangle \cdot \langle 0,2,4\rangle \cdot \langle 11\rangle$", "$\langle 0,2,5,7,10\rangle \cdot \langle 0,2,5,7,10\rangle \cdot \langle 0,2,4\rangle \cdot \langle 11\rangle \cdot \langle 11\rangle$", "$\langle 0,2,5,7,10\rangle \cdot \langle 0,2,5,7,10\rangle \cdot \langle 0,2,5,7,9\rangle \cdot \langle 12\rangle \cdot \langle 12\rangle$", "$\langle 0,2,5,7,10\rangle \cdot \langle 0,2,5,7,9\rangle \cdot \langle 12\rangle$", "$\langle 0,2,5,7,9\rangle$". They are indeed all the paths that can be followed with no more than 3 recursive calls. This time the domain has been set up to $\{0..100\}$.

6 Conclusions and Ongoing work

Very recently, we proposed in [2] a generic approach to TDG by PE which in principle can be used for any imperative language. However, applying this approach to TDG of a declarative language like Prolog introduces some difficulties like the handling of failing derivations and of symbolic data. In this work, we

have sketched solutions to overcome such difficulties. In particular, we have proposed a program transformation, based on PE, to make failure explicit in the Prolog programs. To handle Prolog's negation in the transformed programs, we have outlined existing solutions that make it possible to turn the negative information into positive information. Though our preliminary experiments already suggest that the approach can be very useful to generate test-cases for Prolog, we plan to carry out a thorough practical assessment. This requires to cover additional Prolog features like the module system, builtins like `cut/0`, `fail/0`, `if/3`, etc. and also to compare the results with other TDG systems. We also want to study the integration of other kinds of coverage criteria like *data-flow* based criteria. Finally, we would like to explore the use of static analyses in the context of TDG. For instance, the information inferred by a *failure analysis* can be very useful to prune some of the branches that our transformed programs have to consider.

Acknowledgments This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and by the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

1. E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In *Proc. PADL*, number 4354 in LNCS. Springer-Verlag, 2007.
2. E. Albert, M. Gómez-Zamalloa, and G. Puebla. Test Data Generation of Bytecode by clp Partial Evaluation. In *18th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'08)*, LNCS. Springer-Verlag, July 2008. To appear.
3. F. Belli and O. Jack. Implementation-based analysis and testing of prolog programs. In *ISSTA*, pages 70–80, 1993.
4. Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.
5. F. Degrave, T. Schrijvers, and W. Vanhoof. Automatic generation of test inputs for mercury. In *18th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'08)*, LNCS. Springer-Verlag, 2008. To appear.
6. R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.
7. S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *PPDP*, pages 63–74, 2007.
8. Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
9. J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proc. of PEPM'93*, pages 88–98. ACM Press, 1993.
10. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Modular Decompilation of Low-Level Code by Partial Evaluation. In *8th International Working Conference on*

Source Code Analysis and Manipulation (SCAM'08). IEEE Computer Society, September 2008. To appear.
11. A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Computational Logic*, pages 399–413, 2000.
12. Kim S. Henriksen and John P. Gallagher. Abstract interpretation of pic programs through logic programming. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 184–196. IEEE Computer Society, 2006.
13. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, New York, 1993.
14. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
15. F.J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
16. G. Luo, G. Bochmann, B. Sarikaya, and M. Boyer. Control-flow based testing of prolog programs. In *In Proc. of the 3rd International Symposium on Software Reliability Engineering*, pages 104–113, 1992.
17. M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, August 2007.
18. C. Meudec. Atgen: Automatic test data generation using constraint logic programming and symbolic execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.
19. J.C. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs through analysis of constraint logic programs. In *Proc. of SAS'98*, volume 1503 of *LNCS*, pages 246–261, 1998.
20. Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. *SICStus Prolog 3.8 User's Manual*, 3.8 edition, October 1999. Available from `http://www.sics.se/sicstus/`.
21. L. Zhao, T. Gu, J. Qian, and G. Cai. A novel test case generation method for prolog programs based on call patterns semantics. In *APLAS*, pages 105–121, 2007.
22. Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.