Test Data Generation of Bytecode by CLP Partial Evaluation

Elvira Albert[1], Miguel Gómez-Zamalloa[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. We employ existing *partial evaluation* (PE) techniques developed for Constraint Logic Programming (CLP) in order to automatically generate *test-case generators* for glass-box testing of bytecode. Our approach consists of two independent CLP PE phases. (1) First, the bytecode is transformed into an equivalent (decompiled) CLP program. This is already a well studied transformation which can be done either by using an ad-hoc decompiler or 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, we 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 our approach is that, this second PE phase allows generating not only test-cases but also test-case *generators*. To the best of our knowledge, this is the first time that (CLP) PE techniques are applied for test-case generation as well as to generate test-case generators.

1 Introduction

Bytecode (e.g., Java bytecode [19] or .Net) is becoming widely used, especially, in the context of mobile applications for which the source code is not available and, hence, there is a need to develop verification and validation tools which work directly on bytecode programs. Reasoning about complex bytecode programs is rather difficult and time consuming. In addition to object-oriented features such as objects, virtual method invocation, etc., bytecode has several low-level language features: it has an unstructured control flow with several sources of branching (e.g., conditional and unconditional jumps) and uses an operand stack to perform intermediate computations.

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 [27] 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 [9,14] 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 [7,22,23,17,13], 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 has been combined with the use of *constraint solvers* [23,13] in order to: handle the constraints systems by solving the feasibility of paths and, afterwards, to instantiate the input variables. For the particular case of Java bytecode, a symbolic JVM machine (SJVM) which integrates several constraints solvers has been designed in [23]. A SJVM requires non-trivial extensions w.r.t. a JVM: (1) it needs to execute the bytecode symbolically as explained above, (2) it must be able to backtrack, as without knowledge about the input data, the execution engine might need to execute more than one path. The backtracking mechanism used in [23] is essentially the same as in logic programming.

We propose a novel approach to TDG of bytecode which is based on PE techniques developed for CLP and which, in contrast to previous work, does not require the devising a dedicated symbolic virtual machine. Our method comprises two CLP PE phases which are independent. In fact, they rely on different execution and control strategies:

1. *The decompilation of bytecode into a CLP program.* This has been the subject of previous work [15,3,12] and can be achieved automatically by relying on the first Futamura projection by means of partial evaluation for logic programs, or alternatively by means of an adhoc decompiler [21].
2. *The generation of test-cases.* This is a novel application of PE which allows generating test-case generators from CLP decompiled bytecode. In this case, we 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. The two control operators of a CLP partial evaluator play an essential role: (1) The local control applied to the decompiled code will allow capturing interesting coverage criteria for TDG of the bytecode. (2) The global control will enable the generation of *test-case generators*. Intuitively, the TDG generators we produce are CLP programs whose execution in CLP returns further test-cases on demand without the need to start the TDG process from scratch.

We argue that our CLP PE based approach to TDG of bytecode has several advantages w.r.t. existing approaches based on symbolic execution: (i) It is more *generic*, as the same techniques can be applied to other both low and high-level imperative languages. In particular, once the CLP decompilation is done, the language features are abstracted away and, the whole part related to TDG generation is totally *language independent*. This avoids the difficulties of dealing with recursion, procedure calls, dynamic memory, etc. that symbolic abstract machines typically face. (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 more *powerful* as we can generate test-case generators. (iv) It is *simpler* to implement compared to the development of a dedicated SJVM, as long as a CLP partial evaluator is available.

The rest of the paper is organized as follows. The next section recalls some preliminary notions. Sec. 3 describes the notion of CLP *block-level* decompilation which corresponds to the first phase above. The second phase is explained in the remainder of the paper. Sec. 4 presents a naïve approach to TDG using CLP decompiled programs. In Sec. 5, we introduce the block count-k coverage criterion and outline an evaluation strategy for it. In Sec. 6, we present our approach to TDG by partial evaluation of CLP. Sec. 7 discusses related work and concludes.

2 Preliminaries and Notation in Constraint Logic Programs

We now introduce some basic notions about *Constraint Logic Programming* (CLP). See e.g. [20] for more details. A *constraint store*, or *store* for short, is a conjunction of expressions built from predefined predicates (such as term equations and equalities or inequalities over the integers) whose arguments are constructed using predefined functions (such as addition, multiplication, etc.). We let $\overline{\exists}_L\theta$ be the constraint store θ restricted to the variables of the syntactic object L. An *atom* has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and the t_i are terms. A *literal* L is either an atom or a constraint. A *goal* $L_1, \ldots, L_n$ is a possibly empty finite conjunction of literals. A *rule* is of the form H :- B where H, the *head*, is an atom and B, the *body*, is a goal. A *constraint logic program*, or *program*, is a finite set of rules. We use *mgu* to denote a most general unifier for two unifiable terms.

The operational semantics of a program P is in terms of its *derivations* which are sequences of reductions between *states*. A *state* $\langle G \,|\, \theta \rangle$ consists of a goal G and a constraint store θ. A state $\langle L, G \,|\, \theta \rangle$ where L is a literal can be *reduced* as follows:

1. If L is a constraint and $\theta \wedge L$ is satisfiable, it is reduced to $\langle G \,|\, \theta \wedge L \rangle$.
2. If L is an atom, it is reduced to $\langle B, G \,|\, \theta \wedge \theta' \rangle$ for some renamed apart rule (L' :- B) in P such that L and L' unify with mgu θ'.

A *derivation* from state S for program P is a sequence of states $S_0 \rightarrow_P S_1 \rightarrow_P ... \rightarrow_P S_n$ where S_0 is S and there is a reduction from each S_i to S_{i+1}. Given a non-empty derivation D, we denote by *curr_state(D)* and *curr_store*(D) the last state in the derivation, and the store in this last state, respectively. E.g., if D is the derivation $S_0 \rightarrow_P^* S_n$, where $\rightarrow^*$ denotes a sequence of steps, with $S_n = \langle G \,|\, \theta \rangle$ then *curr_state*$(D) = S_n$ and *curr_store*$(D) = \theta$. A query is a pair (L, θ) where L is a literal and θ a store for which the CLP system starts a computation from $\langle L \,|\, \theta \rangle$.

The observational behavior of a program is given by its "answers" to queries. A finite derivation D from a query $Q = (L, \theta)$ for program P is *finished* if $curr\_state(D)$ cannot be reduced. A finished derivation D from a query $Q = (L, \theta)$ is *successful* if $curr\_state(D) = \langle \epsilon \mid \theta' \rangle$, where ϵ denotes the empty conjunction. The constraint $\bar{\exists}_L \theta'$ is an *answer* to Q. A finished derivation is *failed* if the last state is not of the form $\langle \epsilon \mid \theta \rangle$. Since evaluation trees may be infinite, we allow *unfinished* derivations, where we decide not to further perform reductions. Derivations can be organized in execution trees: a state S has several children when its leftmost atom unifies with several program clauses.

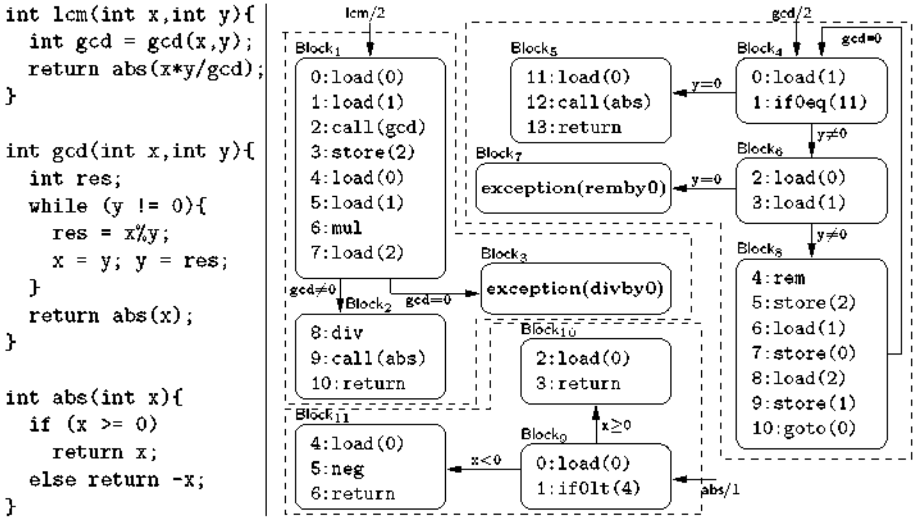
3 Decompilation of Bytecode to CLP

Let us first briefly describe the bytecode language we consider. It is a very simple imperative low-level language in the spirit of Java bytecode, without object-oriented features and restricted to manipulate only integer numbers. It uses an operand stack to perform computations and has an unstructured control flow with explicit conditional and unconditional goto instructions. A bytecode program is organized in a set of methods which are the basic (de)compilation units of the bytecode. The code of a method m consists of a sequence of bytecode instructions $BC_m = < pc_0 : bc_0, \ldots, pc_{n_m} : bc_{n_m} >$ with $pc_0, \ldots, pc_{n_m}$ being consecutive natural numbers. The instruction set is:

$$\begin{aligned} BcInst ::= {} & \mathsf{push(x)} \mid \mathsf{load(v)} \mid \mathsf{store(v)} \mid \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{rem} \mid \mathsf{neg} \mid \\ & \mathsf{if} \diamond \mathsf{(pc)} \mid \mathsf{if0} \diamond \mathsf{(pc)} \mid \mathsf{goto(pc)} \mid \mathsf{return} \mid \mathsf{call(mn)} \end{aligned}$$

where $\diamond$ is a comparison operator (eq, le, gt, etc.), v a local variable, x an integer, pc an instruction index and mn a method name. The instructions push, load and store transfer values or constants from a local variable to the stack (and vice versa); add, sub, mul, div, rem and neg perform arithmetic operations, rem is the division remainder and neg the negation; if and if0 are conditional branching instructions (with the special case of comparisons with 0); goto is an unconditional branching; return marks the end of methods returning an integer and call invokes a method.

Figure 1 depicts the control flow graphs (CFGs) [1] and, within them, the bytecode instructions associated to the methods lcm (on the left), gcd (on the right) and abs (at the bottom). A Java-like source code for them is shown to the left of the figure. It is important to note that we show source code only for clarity, as our approach works directly on the bytecode. The use of the operand stack can be observed in the example: the bytecode instructions at pc 0 and 1 in lcm load the values of parameters x and y (resp.) to the stack before invoking the method gcd. Method parameters and local variables in the program are referenced by consecutive natural numbers starting from 0 in the bytecode. The result of executing the method gcd has been stored on the top of the stack. At pc 3, this value is popped and assigned to variable 2 (called gcd in the Java program). The branching at the end of $\mathtt{Block}_1$ is due to the fact that the division bytecode instruction div can throw an exception if the divisor is zero (control



```
int lcm(int x,int y){
  int gcd = gcd(x,y);
  return abs(x*y/gcd);
}

int gcd(int x,int y){
  int res;
  while (y != 0){
    res = x%y;
    x = y; y = res;
  }
  return abs(x);
}

int abs(int x){
  if (x >= 0)
    return x;
  else return -x;
}
```

Fig. 1. Working example. Source code and CFGs for the bytecode.

goes to Block$_3$). In the bytecode for gcd, we find: conditional jumps, like **if0eq** at *pc* 1, which corresponds to the loop guard, and unconditional jumps, like **goto** in *pc* 10, where the control returns to the loop entry. Note that the bytecode instruction rem can throw an exception as before.

3.1 Decompilation by PE and Block-Level Decompilation

The decompilation of low-level code to CLP has been the subject of previous research, see [15,3,21] and their references. In principle, it can be done by defining an adhoc decompiler (like [2,21]) or by relying on the technique of PE (like [15,3]). The decompilation of low-level code to CLP by means of PE consists in specializing a bytecode interpreter implemented in CLP together with (a CLP representation of) a bytecode program. As the first Futamura projection [11] predicts, we obtain a CLP residual program which can be seen as a decompiled and translated version of the bytecode into high-level CLP source. The approach to TDG that will be presented in the remaining of this paper is independent of the technique used to generate the CLP decompilation. Thus, we will not explain the decompilation process (see [15,3,21]) but rather only state the decompilation requirements our method imposes.

The *correctness* of decompilation must ensure that there is a one to one correspondence between execution paths in the bytecode and derivations in the CLP decompiled program. In principle, depending on the particular type of decompilation –and even on the options used within a particular method– we can obtain different correct decompilations which are valid for the purpose of execution. However, for the purpose of generating useful test-cases, additional requirements

```
lcm([X,Y],Z) :- gcd([X,Y],GCD),P #= X*Y,
                lcm1c([GCD,P],Z).
lcm1c([GCD,P],Z) :- GCD #\= 0,D #= P/GCD,
                        abs([D],Z).
lcm1c([0,_],divby0).

abs([X],Z) :- abs9c(X,Z).
abs9c(X,X) :- X #>= 0.
abs9c(X,Z) :- X #< 0, Z #= -X.

gcd([X,Y],Z) :- gcd4(X,Y,Z).
```

```
gcd4(X,Y,Z) :- gcd4c(X,Y,Z).

gcd4c(X,0,Z) :- abs([X],Z).
gcd4c(X,Y,Z) :- Y #\= 0,
                gcd6c(X,Y,Z).

gcd6c(X,Y,Z) :- Y #\= 0,
                R #= X mod Y,
                 gcd4(Y,R,Z).
gcd6c(_,0,remby0).
```

Fig. 2. Block-level decompilation to CLP for working example

are needed: we must be able to define coverage criteria on the CLP decompilation which produce test-cases which cover the *equivalent* coverage criteria for the bytecode. The following notion of *block-level* decompilation, introduced in [12], provides a sufficient condition for ensuring that equivalent coverage criteria can be defined.

Definition 1 (block-level decompilation). *Given a bytecode program BC and its CLP-decompilation P, a block-level decompilation ensures that, for each block in the CFGs of BC, there exists a* single *corresponding rule in P which contains all bytecode instructions within the block.*

The above notion was introduced in [12] to ensure optimality in decompilation, in the sense that each program point in the bytecode is traversed, and decompiled code is generated for it, at most once. According to the above definition there is a one to one correspondence between blocks in the CFG and rules in P, as the following example illustrates. The block-level requirement is usually an implicit feature of adhoc decompilers (e.g., [2,21]) and can be also enforced in decompilation by PE (e.g., [12]).

Example 1. Figure 2 shows the code of the block-level decompilation to CLP of our running example which has been obtained using the decompiler in [12] and uses CLP(FD) built-in operations (in particular those in the `clpfd` library of `Sicstus` Prolog). The input parameters to methods are passed in a list (first argument) and the second argument is the output value. We can observe that each block in the CFG of the bytecode of Fig. 1 is represented by a corresponding clause in the above CLP program. For instance, the rules for `lcm` and `lcm1c` correspond to the three blocks in the CFG for method `lcm`. The more interesting case is for method `gcd`, where the `while` loop has been converted into a cycle in the decompiled program formed by the predicates `gcd4`, `gcd4c`, and `gcd6c`. In this case, since `gcd4` is the head of a loop, there is one more rule (`gcd`) than blocks in the CFG. This additional rule corresponds to the method *entry*. Bytecode instructions are decompiled and translated to their corresponding operations in CLP; conditional statements are captured by the continuation rules.

For instance, in gcd4, the bytecode instruction at *pc* 0 is executed to unify a stack position with the local variable y. The conditional if0eq at *pc* 1 leads to two continuations, i.e. two rules for predicate gcd4c: one for the case when y=0 and another one for y≠0. Note that we have explicit rules to capture the exceptional executions (which will allow generating test-cases which correspond to exceptional executions). Note also that in the decompiled program there is no difference between calls to blocks and method calls. E.g., the first rule for lcm includes in its body a method call to gcd and a block call lcm1c.

4 Test Data Generation Using CLP Decompiled Programs

Up to now, the main motivation for CLP decompilation has been to be able to perform static analysis on a decompiled program in order to infer properties about the original bytecode. If the decompilation approach produces CLP programs which are executable, then such decompiled programs can be used not only for static analysis, but also for dynamic analysis and execution. Note that this is not always the case, since there are approaches (like [2,21]) which are aimed at producing static analysis targets only and their decompiled programs cannot be executed.

4.1 Symbolic Execution for Glass-Box Testing

A novel interesting application of CLP decompilation which we propose in this work is the automatic generation of glass-box test data. We will aim at generating test-cases which traverse as many different execution paths as possible. From this perspective, different test data should correspond to different execution paths. With this aim, rather than executing the program starting from different input values, a well-known approach consists in performing *symbolic execution* such that a single symbolic run captures the behaviour 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 (see Sec. 1).

Several symbolic execution engines exist for languages such as Java [4] and Java bytecode [23,22]. An important advantage of CLP decompiled programs w.r.t. their bytecode counterparts is that symbolic execution does not require, at least in principle, to build a dedicated symbolic execution mechanism. Instead, we can simply run the decompiled program by using the standard CLP execution mechanism with all arguments being distinct free variables. E.g., in our case we can execute the query lcm([X,Y],Z). By running the program without input values on a block level decompiled program, each successful execution corresponds to a different computation path in the bytecode. Furthermore, along the execution, a constraint store on the program's variables is obtained which can be used for inferring the conditions that the input values (in our case X and Y) must satisfy for the execution to follow the corresponding computation path.

4.2 From Constraint Stores to Test Data

An inherent assumption in the symbolic execution approach, regardless of whether a dedicated symbolic execution engine is built or the default CLP execution is used, is that all valuations of constraint variables which satisfy the constraints in the store (if any) result in input data whose computation traverses the same execution path. Therefore, it is irrelevant, from the point of view of the execution path, which actual values are chosen as representatives of a given store. In any case, it is often required to find a valuation which satisfies the store. Note that this is a strict requirement if we plan to use the bytecode program for testing, though it is not strictly required if we plan to use the decompiled program for testing, since we could save the final store and directly use it as input test data. Then, execution for the test data should load the store first and then proceed with execution. In what follows, we will concentrate on the first alternative, i.e., we generate actual values as test data.

This postprocessing phase is straightforward to implement if we use CLP(FD) as the underlying constraint domain, since it is possible to enumerate values for variables until a solution which is consistent with the set of constraints is found (i.e., we perform *labeling*). Note, however, that it may happen that some of the computed stores are indeed inconsistent and that we cannot find any valuation of the constraint variables which simultaneously satisfies all constraints in the store. This may happen for unfeasible paths, i.e., those which do not correspond to any actual execution. Given a decompiled method M, an integer subdomain [RMin,RMax], the predicate **generate_test_data/4** below produces, on backtracking, a (possibly infinite) set of values for the variables in Args and the result value in Z.

```
generate_test_data(M,Args,[RMin,RMax],Z) :-
    domain(Args,RMin,RMax), Goal =..[M,Args,Z],
    call(Goal), once(labeling([ff],Args)).
```

Note that the generator first imposes an integer domain for the program variables by means of the call to domain/3; then builds the Goal and executes it by means of call(Goal) to generate the constraints; and finally invokes the enumeration predicate labeling/2 to produce actual values compatible with the constraints[1]. The test data obtained are in principle specific to some integer subdomain; indeed our bytecode language only handles integers. This is not necessarily a limitation, as the subdomain can be adjusted to the underlying bytecode machine limitations, e.g., $[-2^{31}, 2^{31}-1]$ in the Java virtual machine. Note that if the variables take floating point values, then other constraint domains such as CLP(R) or CLP(Q) should be used and then, other mechanisms for generating actual values should be used.

[1] We are using the clpfd library of Sicstus Prolog. See [26] for details on predicates domain/3, labeling/2, etc.

5 An Evaluation Strategy for *Block-Count*(k) Coverage

As we have seen in the previous section, an advantage of using CLP decompiled programs for test data generation is that there is no need to build a symbolic execution engine. However, an important problem with symbolic execution, regardless of whether it is performed using CLP or a dedicated execution engine, is that the execution tree to be traversed is in most cases infinite, since programs usually contain iterative constructs such as loops and recursion which induce an infinite number of execution paths when executed without input values.

Example 2. Consider the evaluation of the call `lcm([X,Y],Z)`, depicted in Fig. 3. There is an infinite derivation (see the rightmost derivation in the tree) where the cycle {`gcd4`,`gcd4c`,`gcd6c`} is traversed forever. This happens because the value in the second argument position of `gcd4c` is not ground during symbolic computation.

Therefore, it is essential to establish a *termination criterion* which guarantees that the number of paths traversed remains finite, while at the same time an interesting set of test data is generated.

5.1 *Block-count*(k): A Coverage Criteria for Bytecode

In order to reason about how interesting a set of test data is, a large series of *coverage criteria* have been developed over the years which aim at guaranteeing that the program is exercised on interesting control and/or data flows. In this section we present a coverage criterion of interest to bytecode programs. Most existing coverage criteria are defined on high-level, structured programming languages. A widely used control-flow based coverage criterion is loop-count(k), which dates back to 1977 [16], and limits the number of times we iterate on loops to a threshold k. However, bytecode has an unstructured control flow: CFGs can contain multiple different shapes, some of which do not correspond to any of the loops available in high-level, structured programming languages. Therefore, we introduce the block-count(k) coverage criterion which is not explicitly based on limiting the number of times we iterate on loops, but rather on counting how many times we visit each block in the CFG within each computation. Note that the execution of each method call is considered as an independent computation.

Definition 2 (block-count(k)). *Given a natural number k, a set of computation paths satisfies the* block-count(k) criterion *if the set includes all finished computation paths which can be built such that the number of times each block is visited within each computation does not exceed the given k.*

Therefore, if we take $k = 1$, this criterion requires that all non-cyclic paths be covered. Note that $k = 1$ will in general not visit all blocks in the CFG, since traversing the loop body of a **`while`** loop requires $k \geq 2$ in order to obtain a finished path. For the case of structured CFGs, block-count(k) is actually equivalent to loop-count(k'), by simply taking k' to be k-1. We prefer to formulate things in terms of block-count(k) since, formulating loop-count(k) on unstructured CFGs is awkward.

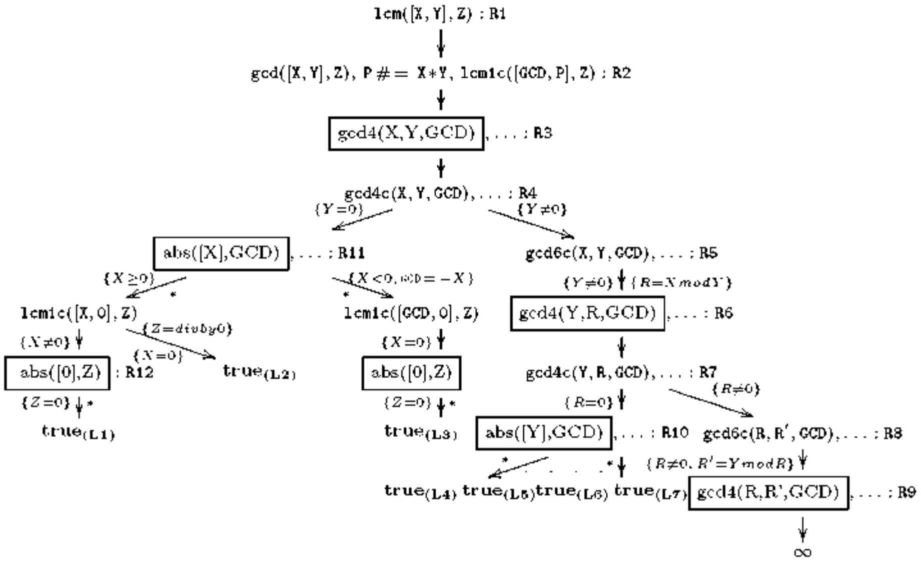

lcm([X,Y],Z) : R1
gcd([X,Y],Z), P #= X*Y, lcm1c([GCD,P],Z) : R2
gcd4(X,Y,GCD), . . . : R3
gcd4c(X,Y,GCD), . . . : R4
{Y=0}
{Y≠0}
abs([X],GCD), . . . : R11
gcd6c(X,Y,GCD), . . . : R5
{X≥0}
*
{X<0, GCD=−X}
*
{Y≠0}
{R=X mod Y}
lcm1c([X,0],Z)
lcm1c([GCD,0],Z)
gcd4(Y,R,GCD), . . . : R6
{Z=div by 0}
{X≠0}
{X=0}
{X=0}
abs([0],Z) : R12
true(L2)
abs([0],Z)
gcd4c(Y,R,GCD), . . . : R7
{R≠0}
{Z=0}
*
{Z=0}
*
{R=0}
true(L1)
true(L3)
abs([Y],GCD), . . . : R10
gcd6c(R,R′,GCD), . . . : R8
*
*
{R≠0, R′=Y mod R}
true(L4) true(L5) true(L6) true(L7)
gcd4(R,R',GCD), . . . : R9
∞

Fig. 3. An evaluation tree for `lcm([X,Y],Z)`

5.2 An Intra-procedural Evaluation Strategy for Block-Count(k)

Fig. 3 depicts (part of) an evaluation tree for `lcm([X,Y],Z)`. Each node in the tree represents a state, which as introduced in Sec. 2, consists of a goal and a store. In order not to clutter the figure, for each state we only show the relevant part of the goal, but not the store. Also, an arc in the tree may involve several reduction steps. In particular, the constraints which precede the leftmost atom (if any) are always processed. Likewise, at least one reduction step is performed on the leftmost atom w.r.t. the program rule whose head unifies with the atom. When more than one step is performed, the arc is labelled with "*". Arcs are annotated with the constraints processed at each step. Each branch in the tree represents a *derivation*.

Our aim is to supervise the generation of the evaluation tree so that we generate sufficiently many derivations so as to satisfy the block-count(k) criterion while, at the same time, guaranteeing termination.

Definition 3 (intra-procedural evaluation strategy). *The following two conditions provide an evaluation strategy which ensures block-count(k) in intra-procedural bytecode (i.e., we consider a single CFG for one method):*

(i) *annotate every state in the evaluation tree with a multiset, which we refer to as* visited, *and which contains the predicates which have been already reduced during the derivation;*

(ii) *atoms can only be reduced if there are at most $k-1$ occurrences of the corresponding predicate in* visited.

It is easy to see that this evaluation strategy is guaranteed to always produce a finite evaluation tree since there is a finite number of rules which can unify with any given atom and therefore non-termination can only be introduced by cycles which are traversed an unbounded number of times. This is clearly avoided by limiting the number of times which resolution can be performed w.r.t. the same predicate.

Example 3. Let us consider the rightmost derivation in Fig. 3, formed by goals R1 to R9. Observe the framed atoms for `gcd4`, the goals R3, R6 and R9 contain an atom for `gcd4` as the leftmost literal. If we take $k = 1$ then resolvent R6 cannot be further reduced since the termination criterion forbids it, as `gcd4` is already once in the multiset of visited predicates. If we take $k = 2$ then R6 can be reduced and the termination criterion is fired at R9, which cannot be further reduced.

5.3 An Inter-procedural Evaluation Strategy Based on Ancestors

The strategy of limiting the number of reductions w.r.t. the same predicate guarantees termination. Furthermore, it also guarantees that the block-count(k) criterion is achieved, but only if the program consists of a single CFG, i.e., at most one method. If the program contains more than one method, as in our example, this evaluation strategy may force termination too early, without achieving block-count(k) coverage.

Example 4. Consider the predicate `abs`. Any successful derivation which does not correspond to exceptions in the bytecode program has to execute this predicate twice, once from the body of method `lcm` and another one from the body of method `gcd`. Therefore, if we take $k = 1$, the leftmost derivation of the tree in Fig. 3 will be stopped at R12, since the atom to be reduced is considered to be a repeated call to predicate `abs`. Thus, the test-case for the successful derivation L1 is not obtained. As a result, our evaluation strategy would not achieve the block-count(k) criterion.

The underlying problem is that we are in an inter-procedural setting, i.e., bytecode programs contain method calls. In this case –meanwhile decompiled versions of bytecode programs without method calls always consist of binary rules– decompiled programs may have rules with several atoms in their body. This is indeed the case for the rule for `lcm` in Ex. 1, which contains an atom for predicate `gcd` and another one for predicate `lcm1c`. Since under the standard left-to-right computation rule, the execution of `gcd` is finished by the time execution reaches `lcm1c` there is no need to take the computation history of `gcd` into account when supervising the execution of `lcm1c`. In our example, the execution of `gcd` often involves an execution of `abs` which is finished by the time the call to `abs` is performed within the execution of `lcm1c`. This phenomenon is well known problem in the context of partial evaluation. There, the notion of *ancestor* has been introduced [5] to allow supervising the execution of conjuncts independently by only considering visited predicates which are actually ancestors of the current goal. This allows improving accuracy in the specialization.

Given a reduction step where the leftmost atom A is substituted by $B_1, \ldots, B_m$, we say that A is the *parent* of the instance of B_i for $i = 1, \ldots, m$ in the new goal and in each subsequent goal where the instance originating from B_i appears. The *ancestor* relation is the transitive closure of the parent relation. The multiset of ancestors of the atom for `abs` in goal R12 in the SLD tree is {`lcm1c`,`lcm`}, as `lcm1c` is its parent and `lcm` the parent of its parent. Importantly, `abs` is not in such multiset. Therefore, the leftmost computation in Fig. 3 will proceed upon R12 thus producing the corresponding test-case for every $k \geq 1$. The evaluation strategy proposed below relies on the notion of ancestor sequence.

Definition 4 (inter-procedural evaluation strategy). *The following two conditions provide an evaluation strategy which ensures block-count(k) in inter-procedural bytecode (i.e., we consider several CFGs and methods):*

(i) annotate every atom in the evaluation tree with a multiset which contains its ancestor sequence which we refer to as ancestors;
(ii) atoms can only be reduced if there are at most $k - 1$ occurrences of the corresponding predicate in its ancestors.

The next section provides practical means to implement this strategy.

6 Test Data Generation by Partial Evaluation

We have seen in Sec. 5 that a central issue when 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. An important observation is that 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, as we show in Sec. 6.1, we can directly apply existing, powerful, unfolding rules developed in the context of PE. Second, in Sec. 6.2, we show that it is possible to explore additional abilities of partial evaluators in the context of TDG. Interestingly, the generation of a residual program from the evaluation tree returns a program which can be used as a *test-case generator* for obtaining further test-cases.

6.1 Using an Unfolding Rule for Implementing Block-Count(k)

Sophisticated unfolding rules exist which incorporate non-trivial mechanisms to stop the construction of SLD trees. For instance, unfolding rules based on comparable atoms allow expanding derivations as long as no previous *comparable* atom (same predicate symbol) has been already visited. As already discussed, the use of ancestors [5] can reduce the number of atoms for which the comparability test has to be performed.

In PE terminology, the evaluation strategy outlined in Sec. 5 corresponds to an unfolding rule which allows k comparable atoms in every ancestor sequence. Below, we provide an implementation, predicate **unfold/3**, of such an unfolding rule. The CLP decompiled program is stored as **clause/2** facts. Predicate **unfold/3** receives as input parameters an atom as the initial goal to evaluate, and the value of constant k. The third parameter is used to return the resolvent associated with the corresponding derivation.

```
unfold(A,K,[load_st(St)|Res]) :-
    unf([A],K,[],Res),
    collect_vars([A|Res],Vars),
    save_st(Vars,St).

unf([],_K,_AS,[]).
unf([A|R],K,AncS,Res) :-
    constraint(A),!, call(A),
    unf(R,K,AncS,Res).
unf(['$pop$'|R],K,[_|AncS],Res) :-
    !, unf(R,K,AncS,Res).
```

```
unf([A|R],K,AncS,Res) :-
    clause(A,B), functor(A,F,Ar),
    (check(AncS,F,Ar,K) ->
       append(B,['$pop$'|R],NewGoal),
       unf(NewGoal,K,[F/Ar|AncS],Res)
    ;  Res = [A|R]).

check([],_,_,K) :- K > 0.
check([F/Ar|As],F,Ar,K) :- !, K > 1,
    K1 is K - 1, check(As,F,Ar,K1).
check([_|As],F,Ar,K) :- check(As,F,Ar,K).
```

Predicate **unfold/3** first calls **unf/4** to perform the actual unfolding and then, after collecting the variables from the resolvent and the initial atom by means of predicate **collect_vars/2**, it saves the store of constraints in variable **St** so that it is included inside the call **load_st(St)** in the returned resolvent. The reason why we do this will become clear in Sect. 6.2. Let us now explain intuitively the four rules which define predicate **unf/4**. The first one corresponds to having an empty goal, i.e., the end of a successful derivation. The second rule corresponds to the first case in the operational semantics presented in Sec. 2, i.e., when the leftmost literal is a constraint. Note that in CLP there is no need to add an argument for explicitly passing around the store, which is implicitly maintained by the execution engine by simply executing constraints by means of predicate **call/1**. The second case of the operational semantics in Sec. 2, i.e., when the leftmost literal is an atom, corresponds to the fourth rule. Here, on backtracking we look for all rules asserted as **clause/2** facts whose head unifies with the leftmost atom. Note that depending on whether the number of occurrences of comparable atoms in the ancestors sequence is smaller than the given k or not, the derivation continues or it is stopped. The termination check is performed by predicate **check/4**.

In order to keep track of ancestor sequences for every atom, we have adopted the efficient implementation technique, proposed in [25], based on the use of a global *ancestor stack*. Essentially, each time an atom **A** is unfolded using a rule $\mathtt{H} : -\mathtt{B}_1, \ldots, \mathtt{B}_n$, the predicate name of **A**, $pred(A)$, is pushed on the ancestor stack (see third argument in the recursive call). Additionally, a \$**pop**\$ mark is added to the new goal after $\mathtt{B}_1, \ldots, \mathtt{B}_n$ (call to **append/3**) to delimit the scope of the predecessors of **A** such that, once those atoms are evaluated, we find the mark \$**pop**\$ and can remove $pred(A)$ from the ancestor stacks. This way, the ancestor stack, at each stage of the computation, contains the ancestors of the next atom which will be selected for resolution. If predicate `check/4` detects that the number of occurrences of *pred*(A) is greater than k, the derivation is stopped and the current goal is returned in `Res`.The third rule of `unf/4` corresponds to the case where the leftmost atom is a $pop$ literal. This indicates that the the execution of the atom which is on top of the ancestor stack has been completed. Hence, this atom is popped from the stack and the $pop$ literal is removed from the goal.

Example 5. The execution of `unfold(lcm([X,Y],Z),2,[_])` builds a finite (and hence unfinished) version of the evaluation tree in Fig. 3. For $k = 2$, the infinite branch is stopped at goal R9, since the ancestor stack at this point is `[gcd6c, gcd4c,gcd4,gcd6c,gcd4c,gcd4,lcm]` and hence it already contains `gcd4` twice. This will make the `check/4` predicate fail and therefore the derivation is stopped. More interestingly, we can generate test-cases, if we consider the following call:

```
findall(([X,Y],Z),unfold([gen_test_data(lcm,[X,Y],[-1000,1000],Z)],2,[_]),TCases).
```

where `generate_test_data` is defined as in Sec. 4. Now, we get on backtracking, concrete values for variables `X`, `Y` and `Z` associated to each finished derivation of the tree.[2] They correspond to test data for the block-count(2) coverage criteria of the bytecode. In particular, we get the following set of test-cases: TCases = [([1,0],0), ([0,0],divby0), ([-1000,0],0), ([0,1],0), ([-1000,1],1000), ([-1000,-1000], 1000),([1,-1],1)] which correspond, respectively, to the leaves labeled as **(L1)**,...,**(L7)** in the evaluation tree of Fig. 3. Essentially, they constitute a particular set of concrete values that traverses all possible paths in the bytecode, including exceptional behaviours, and where the loop body is executed at most once.

The soundness of our approach to TDG amounts to saying that the above implementation, executed on the CLP decompiled program, ensures termination and block-count(k) coverage on the original bytecode.

Proposition 1 (soundness). *Let m be a method with n arguments and BC_m its bytecode instructions. Let* $\mathtt{m}([\mathtt{X}_1,\ldots,\mathtt{X}_n],\mathtt{Y})$ *be the corresponding decompiled method and let the CLP block-level decompilation of BC_m be asserted as a set of* `clause/2` *facts. For every positive number k, the set of successful derivations computed by* $\mathtt{unf}(\mathtt{m}([\mathtt{X}_1,\ldots,\mathtt{X}_n],\mathtt{Y}),\mathtt{k},[\,],[\,],\_)$ *ensures block-count(k) coverage of BC_m.*

Intuitively, the above result follows from the facts that: (1) the decompilation is correct and block-level, hence all traces in the bytecode are derivations in the decompiled program as well as loops in bytecode are cycles in CLP; (2) the unfolding rule computes all feasible paths and traverses cycles at most k times.

[2] We force to consider just finished derivations by providing `[_]` as the obtained resultant.

6.2 Generating Test Data Generators

The final objective of a partial evaluator is to generate optimized *residual* code. In this section, we explore the applications of the code generation phase of partial evaluators in TDG. Let us first intuitively explain how code is generated. Essentially, the residual code is made up by a set of *resultants* or residual rules (i.e., a program), associated to the root-to-leaf derivations of the computed evaluation trees. For instance, consider the rightmost derivation of the tree in Fig. 3, the associated resultant is a rule whose head is the original atom (applying the mgu's to it) and the body is made up by the atoms in the leaf of the derivation. If we ignore the constraints gathered along the derivation (which are encoded in `load_st`(S) as we explain below), we obtain the following resultant:

```
lcm([X,Y],Z) :- load_st(S), gcd4(R,R',GCD),P #= X*Y, lcm1c([GCD,P],Z).
```

The residual program will be (hopefully) executed more efficiently than the original one since those computations that depend only on the static data are performed once and for all at specialization time. Due to the existence of incomplete derivations in evaluation trees, the residual program might not be complete (i.e., it can miss answers w.r.t. the original program). The partial evaluator includes an *abstraction* operator which is encharged of ensuring that the atoms in the leaves of incomplete derivations are "covered" by some previous (partially evaluated) atom and, otherwise, adds the uncovered atoms to the set of atoms to be partially evaluated. For instance, the atoms `gcd4(R,R',GCD)` and `lcm1c([GCD,P],Z)` above are not covered by the single previously evaluated atom `lcm([X,Y],Z)` as they are not instances of it. Therefore, a new unfolding process must be started for each of the two atoms. Hence the process of building evaluation trees by the unfolding operator is iteratively repeated while new atoms are uncovered. Once the final set of trees is obtained, the resultants are generated from their derivations as described above.

Now, we want to explore the issues behind the application of a full partial evaluator, with its code generation phase, for the purpose of TDG. Novel interesting questions arise: (*i*) *what kind of partial evaluator do we need to specialize decompiled CLP programs?*; (*ii*) *what do we get as residual code?*; (*iii*) *what are the applications of such residual code?* Below we try to answer these questions.

As regards question (*i*), we need to extend the mechanisms used in standard PE of logic programming to support constraints. The problem has been already tackled, e.g., by [8] to which we refer for more details. Basically, we need to take care of constraints at three different points: first, during the execution, as already done by `call` within our unfolding rule `unfold/3`; second, during the abstraction process, we can either define an accurate abstraction operator which handles constraints or, as we do below, we can take a simpler approach which safely ignores them; third, during code generation, we aim at generating *constrained* rules which integrate the *store* of constraints associated to their corresponding derivations. To handle the last point, we enhance our schema with the next two basic operations on constraints which are used by `unfold/3` and were left unexplained in Sec. 6.1. The store is saved and projected by means of predicate `save_st/2`, which given a set of variables in its first argument, saves the current store of the CLP execution, projects it to the given variables and returns the result in its second argument. The store is loaded by means of `load_st/1` which given an explicit store in its argument adds the constraints to the current store. Let us illustrate this process by means of an example.

Example 6. Consider a partial evaluator of CLP which uses as control strategies: predicate `unfold/3` as unfolding rule and a simple abstraction operator based on the combination of the *most specific generalization* and a check of comparable terms (as the unfolding does) to ensure termination. Note that the abstraction operator ignores the constraint store. Given the entry, `gen_test_data(lcm,[X,Y],[-1000,1000],Z)`, we would obtain the following residual code for $k = 2$:

```
gen_test_data(lcm,[1,0],[-1000,1000],0).
gen_test_data(lcm,[0,0],[-1000,1000],
              divby0).
...
gen_test_data(lcm,[X,Y],[-1000,1000],Z) :-
    load_st(S1), gcd4(R,R',GCD),
    P #= X*Y, lcm1c([GCD,P],Z),
    once(labeling([ff],[X,Y])).
```

```
gcd4(R,0,R) :- load_st(S2).
gcd4(R,0,GCD) :- load_st(S3).
gcd4(R,R',GCD) :- load_st(S4),
    gcd4(R',R'',GCD).

lcm1c([GCD,P],Z) :- load_st(S5).
lcm1c([GCD,P],Z) :- load_st(S6).
lcm1c([0,_P],divby0).
```

The residual code for `gen_test_data/4` contains eight rules. The first seven ones are facts corresponding to the seven successful branches (see Fig. 3). Due to space limitations here we only show two of them. Altogether they represent the set of test-cases for the block-count(2) coverage criteria (those in Ex. 6.1). It can be seen that all rules (except the facts[3]) are constrained as they include a residual call to `load_st/1`. The argument of `load_st/1` contains a syntactic representation of the store at the last step of the corresponding derivation. Again, due to space limitations we do not show the stores. As an example, `S1` contains the store associated to the rightmost derivation in the tree of Fig. 3, namely {`X in -1000..1000, Y in (-1000..-1)`$\vee$`(1..1000), R in (-999.. -1)` $\vee$ `(1..999), R' in -998..998, R = X mod Y, R' = Y mod R`}. This store acts as a guard which comprises the constraints which avoid the execution of the paths previously computed to obtain the seven test-cases above.

We can now answer issue (*ii*): it becomes apparent from the example above that we have obtained a program which is a *generator* of test-cases for larger values of k. The execution of the generator will return by backtracking the (infinite) set of values exercising all possible execution paths which traverse blocks more than twice. In essence, our test-case generators are CLP programs whose execution in CLP returns further test-cases on demand for the bytecode under test and without the need of starting the TDG process from scratch.

[3] For the facts, there is no need to consider the store, because a call to `labeling` has removed all variables.

Here, it comes issue (*iii*): Are the above generators useful? How should we use them? In addition to execution (see inherent problems in Sec. 4), we might further partially evaluate them. For instance, we might partially evaluate the above specialized version of **gen_test_data/4** (with the same entry) in order to incrementally generate test-cases for larger values of k. It is interesting to observe that by using $k = 1$ for all atoms different from the initial one, this further specialization will just increment the number of **gen_test_data/4** facts (producing more concrete test-cases) but the rest of the residual program will not change, in fact, there is no need to re-evaluate it later.

7 Conclusions and Related Work

We have proposed a methodology for test data generation of imperative, low-level code by means of existing partial evaluation techniques developed for constraint logic programs. Our approach consist of two separate phases: (1) the compilation of the imperative bytecode to a CLP program and (2) the generation of test-cases from the CLP program. It naturally raises the question whether our approach can be applied to other imperative languages in addition to bytecode. This is interesting as existing approaches for Java [23], and for C [13], struggle for dealing with features like recursion, method calls, dynamic memory, etc. during symbolic execution. We have shown in the paper that these features can be uniformly handled in our approach after the transformation to CLP. In particular, all kinds of loops in the bytecode become uniformly represented by recursive predicates in the CLP program. Also, we have seen that method calls are treated in the same way as calls to blocks. In principle, this transformation can be applied to any language, both to high-level and to low-level bytecode, the latter as we have seen in the paper. In every case, our second phase can be applied to the transformed CLP program.

Another issue is whether the second phase can be useful for test-case generation of CLP programs, which are not necessarily obtained from a decompilation of an imperative code. Let us review existing work for declarative programs. Test data generation has received comparatively less attention than for imperative languages. The majority of existing tools for functional programs are based on black-box testing [6,18]. Test cases for logic programs are obtained in [24] 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 the corresponding paths. This corresponds essentially to the naive approach discussed in Sec. 4, which is not sufficient for our purposes as we have seen in the paper. However, in the case of the generation of test data for regular CLP programs, we are interested not only in successful derivations (execution paths), but also in the failing ones. It should be noted that the execution of CLP decompiled programs, in contrast to regular CLP programs, for any actual input values is guaranteed to produce exactly one solution because the operational semantics of bytecode is deterministic. For functional logic languages, specific coverage criteria are defined in [10] which capture the control flow of these languages as well as new language features are considered, namely laziness. In general, declarative languages pose different problems to testing related to their own execution models –like laziness in functional languages and failing derivations in (C)LP– which need to be captured by appropriate coverage criteria. Having said this, we believe our ideas related to the use of PE techniques to generate test data generators and the use of unfolding rules to supervise the evaluation could be adapted to declarative programs and remains as future work.

Our work is a proof-of-concept that partial evaluation of CLP is a powerful technique for carrying out TDG in imperative low-level languages. To develop our ideas, we have considered a simple imperative bytecode language and left out object-oriented features which require a further study. Also, our language is restricted to integer numbers and the extension to deal with real numbers is subject of future work. We also plan to carry out an experimental evaluation by transforming Java bytecode programs from existing test suites to CLP programs and then trying to obtain useful test-cases. When considering realistic programs with object-oriented features and real numbers, we will surely face additional difficulties. One of the main practical issues is related to the scalability of our approach. An important threaten to scalability in TDG is the so-called infeasibility problem [27]. It happens in approaches that do not handle constraints along the construction of execution paths but rather perform two independent phases (1) path selection and 2) constraint solving). As our approach integrates both parts in a single phase, we do not expect scalability limitations in this regard. Also, a challenging problem is to obtain a decompilation which achieves a manageable representation of the heap. This will be necessary to obtain test-cases which involve data for objects stored in the heap. For the practical assessment, we also plan to extend our technique to include further coverage criteria. We want to consider other classes of coverage criteria which, for instance, generate test-cases which cover a certain statement in the program.

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. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers - Principles, Techniques and Tools. Addison-Wesley, Reading (1986)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
3. Albert, E., Gómez-Zamalloa, M., Hubert, L., Puebla, G.: Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 124–139. Springer, Heidelberg (2006)

4. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS, vol. 4334. Springer, Heidelberg (2007)
5. Bruynooghe, M., De Schreye, D., Martens, B.: A General Criterion for Avoiding Infinite Unfolding during Partial Deduction. New Generation Computing 1(11), 47–79 (1992)
6. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. In: ICFP, pp. 268–279 (2000)
7. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Trans. Software Eng. 2(3), 215–222 (1976)
8. Craig, S.-J., Leuschel, M.: A compiler generator for constraint logic programs. In: Ershov Memorial Conference, pp. 148–161 (2003)
9. Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Trans. Softw. Eng. Methodol. 5(1), 63–86 (1996)
10. Fischer, S., Kuchen, H.: Systematic generation of glass-box test cases for functional logic programs. In: PPDP, pp. 63–74 (2007)
11. Futamura, Y.: Partial evaluation of computation process - an approach to a compiler-compiler. Systems, Computers, Controls 2(5), 45–50 (1971)
12. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Modular Decompilation of Low-Level Code by Partial Evaluation. In: 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), pp. 239–248. IEEE Computer Society, Los Alamitos (2008)
13. Gotlieb, A., Botella, B., Rueher, M.: A clp framework for computing structural test data. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS, vol. 1861, pp. 399–413. Springer, Heidelberg (2000)
14. Gupta, N., Mathur, A.P., Soffa, M.L.: Generating test data for branch coverage. In: Automated Software Engineering, pp. 219–228 (2000)
15. Henriksen, K.S., Gallagher, J.P.: Abstract interpretation of pic programs through logic programming. In: SCAM 2006: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, pp. 184–196. IEEE Computer Society, Los Alamitos (2006)
16. Howden, W.E.: Symbolic testing and the dissect symbolic evaluation system. IEEE Transactions on Software Engineering 3(4), 266–278 (1977)
17. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
18. Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: Gast: Generic automated software testing. In: IFL, pp. 84–100 (2002)
19. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Reading (1996)
20. Marriot, K., Stuckey, P.: Programming with Constraints: An Introduction. MIT Press, Cambridge (1998)
21. Méndez-Lojo, M., Navas, J., Hermenegildo, M.: A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915. Springer, Heidelberg (2008)
22. Meudec, C.: Atgen: Automatic test data generation using constraint logic programming and symbolic execution. Softw. Test., Verif. Reliab. 11(2), 81–96 (2001)
23. Müller, R.A., Lembeck, C., Kuchen, H.: A symbolic java virtual machine for test case generation. In: IASTED Conf. on Software Engineering, pp. 365–371 (2004)
24. Mweze, N., Vanhoof, W.: Automatic generation of test inputs for mercury programs. In: Pre-proceedings of LOPSTR 2006 (July 2006) (extended abstract)

25. Puebla, G., Albert, E., Hermenegildo, M.: Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 149–165. Springer, Heidelberg (2005)
26. 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), http://www.sics.se/sicstus/
27. Zhu, H., Patrick, A., Hall, V., John, H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. 29(4), 366–427 (1997)