

Compositional CLP-Based Test Data Generation for Imperative Languages

Elvira Albert¹, Miguel Gómez-Zamalloa¹,
José Miguel Rojas², and Germán Puebla²

¹ DSIC, Complutense University of Madrid, E-28040 Madrid, Spain

² Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. Glass-box test data generation (TDG) is the process of automatically generating test input data for a program by considering its internal structure. This is generally accomplished by performing symbolic execution of the program where the contents of variables are expressions rather than concrete values. The main idea in CLP-based TDG is to translate imperative programs into equivalent CLP ones and then rely on the standard evaluation mechanism of CLP to symbolically execute the imperative program. Performing symbolic execution on large programs becomes quickly expensive due to the large number and the size of paths that need to be explored. In this paper, we propose *compositional reasoning* in CLP-based TDG where large programs can be handled by testing parts (such as components, modules, libraries, methods, etc.) separately and then by composing the test cases obtained for these parts to get the required information on the whole program. Importantly, compositional reasoning also gives us a practical solution to handle native code, which may be unavailable or written in a different programming language. Namely, we can model the behavior of a native method by means of test cases and compositional reasoning is able to use them.

1 Introduction

Test data generation (TDG) is the process of automatically generating test cases for interesting test *coverage criteria*. Coverage criteria aim at measuring 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; *loop- k* (resp. *block- k*) which limits to a threshold k the number of times we iterate on loops (resp. visit blocks in the control flow graph [1]). Among the wide variety of approaches to TDG (see e.g. [22]), 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 [7,14] 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 [15,5,13,18,19,6,21], where the contents of variables are expressions rather than concrete values. 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* [19,13,21] in order to handle the constraint systems by solving the feasibility of paths and, afterwards, to instantiate the input variables. For instance, a symbolic JVM machine which integrates several constraint solvers has been designed in [19] for TDG of Java (bytecode) programs. In general, a symbolic machine requires non-trivial extensions w.r.t. a non-symbolic one like the JVM: (1) it needs to execute (imperative) code symbolically as explained above, (2) it must be able to non-deterministically execute multiple paths (as without knowledge about the input data non-determinism usually arises).

In recent work [11], we have proposed a CLP-based approach to TDG of imperative programs consisting of three main ingredients: (i) The imperative program is first translated into an equivalent CLP one, named *CLP-translated program* in what follows. The translation can be performed by partial evaluation [10] or by traditional compilation. (ii) Symbolic execution on the CLP-translated program can be performed by relying on the standard evaluation mechanism of CLP, which provides backtracking and handling of symbolic expressions for free. (iii) The use of dynamic memory requires to define heap-related operations that, during TDG, take care of constructing complex data structures with unbounded data (e.g., recursive data structures). Such operations can be implemented in CLP [11].

It is well-known that symbolic execution might become computationally intractable due to the large number of paths that need to be explored and also to the size of their associated constraints (see [20]). While compositionality has been applied in many areas of static analysis to alleviate these problems, it is less widely used in TDG (some notable exceptions in the context of dynamic testing are [8,3]). In this paper, we propose a compositional approach to static CLP-based TDG for imperative languages. In symbolic execution, compositionality means that when a method m invokes another method p for which TDG has already been performed, the execution can *compose* the *test cases* available for p (also known as *method summary* for p) with the current execution state and continue the process, instead of having to symbolically execute p again. By test cases or method summary, we refer to the set of path constraints obtained by symbolically executing p using a certain coverage criterion. Compositional TDG has several advantages over global TDG. First, it avoids repeatedly performing TDG of the same method. Second, components can be tested with higher precision when they are chosen small enough. Third, since separate TDG is done on parts and not on the whole program, total memory consumption may be reduced. Fourth, separate TDG can be performed in parallel on independent computers and the global TDG time can be reduced as well.

```

class R {
    int n; int d;
    void simplify(){
        int gcd = A.gcd(n,d);
        n = n/gcd; d = d/gcd;}
    static R[] simp(R[] rs){
        int length = rs.length;
        R[] oldRs = new R[length];
        arraycopy(rs,oldRs,length);
        for (int i = 0; i < length; i++){
            rs[i].simplify();
        }
        return oldRs;}}
}

class A {
    static int abs(int x){
        if (x >= 0) return x;
        else return -x;
    }
    static int gcd(int a,int b){
        int res;
        while (b != 0){
            res = a%b; a = b; b = res;}
        return abs(a);
    }
}

```

Fig. 1. Java source of working example

Furthermore, having a compositional TDG approach in turn facilitates the handling of *native code*, i.e., code which is implemented in a different language. This is achieved by modeling the behavior of native code as a method summary which can be composed with the current state during symbolic execution in the same way as the test cases inferred automatically by the testing tool are. By treating native code, we overcome one of the inherent limitations of symbolic execution (see [20]). Indeed, improving the efficiency of TDG and handling native code are considered main challenges in the fields of symbolic execution and TDG.

We report on a prototype implementation in PET [2], a Partial-Evaluation based Test case generation tool for Java bytecode. Experimental results on a set of medium-sized benchmarks already show that compositional TDG can highly improve the performance over non-compositional TDG.

2 CLP-Based Test Case Generation

In this section, we summarize the CLP-based approach to TDG for imperative languages introduced in [1] and recently extended to object-oriented languages with dynamic memory in [11]. For simplicity, we do not take aliasing of references into account and simplify the language by excluding inheritance and virtual invocations. However, these issues are orthogonal to compositionality and our approach could be applied to the complete framework of [11]. Also, although our approach is not tied to any particular imperative language, as regards dynamic memory, we assume a Java-like language. In [10], it has been shown that Java bytecode (and hence Java) can be translated into the equivalent CLP-programs shown below.

2.1 From Imperative to Equivalent CLP Programs

A *CLP-translated program* as defined in [11] is made up of a set of predicates. Each predicate is defined by one or more mutually exclusive clauses. Each clause

receives as input a possibly empty list of arguments $Args_{in}$ and an input heap H_{in} , and returns a possibly empty output $Args_{out}$, a possibly modified output heap H_{out} , and an exception flag indicating whether the execution ends normally or with an uncaught exception. The body of a clause may include a set of guards (comparisons between numeric data or references, etc.) followed by different types of instructions: arithmetic operations and assignment statements, calls to other predicates, instructions to create objects and arrays and to consult the array length, read and write accesses to object fields or array positions, as defined by the following grammar:

$$\begin{aligned}
 \text{Clause} &::= \text{Pred}(Args_{in}, Args_{out}, H_{in}, H_{out}, \text{ExFlag}) :- [G,]B_1, B_2, \dots, B_n. \\
 G &::= \text{Num}^* \text{ROp} \text{Num}^* \mid \text{Ref}_1^* \setminus = \text{Ref}_2^* \\
 B &::= \text{Var} \# = \text{Num}^* \text{AOp} \text{Num}^* \mid \text{Pred}(Args_{in}, Args_{out}, H_{in}, H_{out}, \text{ExFlag}) \mid \\
 &\quad \text{new_object}(H, C^*, \text{Ref}^*, H) \mid \text{new_array}(H, T, \text{Num}^*, \text{Ref}^*, H) \mid \text{length}(H, \text{Ref}^*, \text{Var}) \mid \\
 &\quad \text{get_field}(H, \text{Ref}^*, \text{FSig}, \text{Var}) \mid \text{set_field}(H, \text{Ref}^*, \text{FSig}, \text{Data}^*, H) \mid \\
 &\quad \text{get_array}(H, \text{Ref}^*, \text{Num}^*, \text{Var}) \mid \text{set_array}(H, \text{Ref}^*, \text{Num}^*, \text{Data}^*, H) \\
 \\
 \text{Pred} &::= \text{Block} \mid \text{MSig} & \text{ROp} &::= \# > \mid \# < \mid \# > = \mid \# = < \mid \# = \mid \# \setminus = \\
 \text{Args} &::= [] \mid [\text{Data}^* \mid \text{Args}] & \text{AOp} &::= + \mid - \mid * \mid / \mid \text{mod} \\
 \text{Data} &::= \text{Num} \mid \text{Ref} \mid \text{ExFlag} & T &::= \text{bool} \mid \text{int} \mid C \mid \text{array}(T) \\
 \text{Ref} &::= \text{null} \mid r(\text{Var}) & \text{FSig} &::= C:\text{FN} \\
 \text{ExFlag} &::= \text{ok} \mid \text{exc}(\text{Var}) & H &::= \text{Var}
 \end{aligned}$$

Non-terminals Block , Num , Var , FN , MSig and C denote, resp., the set of predicate names, numbers, variables, field names, method signatures and class names. Clauses can define both methods which appear in the original source program (MSig), or additional predicates which correspond to intermediate blocks in the program (Block). An asterisk on a non-terminal denotes that it can be either as defined by the grammar or a (possibly constraint) variable.

Example 1. Fig. 1 shows the Java source of our running example and Fig. 2 the CLP-translated version of method `simp` obtained from the bytecode. The main features that can be observed from the translation are: (1) All clauses contain input and output arguments and heaps, and an exception flag. Reference variables

```

simp([r(Rs)], [Ret], H0, H3, EF) :- length(H0, Rs, Len), Len #>= 0, new_array(H0, 'R', Len, OldRs, H1),
  arraycopy([r(Rs), r(OldRs)], Len, [], H1, H2, EFp), r1([EFp, r(Rs), r(OldRs)], Len, [Ret], H2, H3, EF).
simp([null], -, Hin, Hout, exc(ERef)) :- new_object(Hin, 'NPE', ERef, Hout).
r1([ok, Rs, OldRs, Length], [Ret], H1, H2, EF) :- loop([Rs, OldRs, Length, 0], [Ret], H1, H2, EF).
r1([exc(ERef), -, -, -, -, -, H, H, exc(ERef)]).
loop([_, OldRs, Length, I], [OldRs], H, H, ok) :- I #>= Length.
loop([Rs, OldRs, L, I], [Ret], H1, H2, EF) :- I #< L, loopbody1([Rs, OldRs, L, I], [Ret], H1, H2, EF).
loopbody1([r(Rs), OldRs, Length, I], [Ret], H1, H2, EF) :- length(H1, Rs, L), L #>= 0, I #< L,
  get_array(H1, Rs, I, RSi), loopbody2([r(Rs), OldRs, Length, I, RSi], [Ret], H1, H2, EF).
loopbody2([Rs, OldRs, Length, I, r(RSi)], [Ret], H1, H3, EF) :- simplify([r(RSi)], [], H1, H2, EFp),
  loopbody3([EFp, Rs, OldRs, Length, I], [Ret], H2, H3, EF).
loopbody3([_, -, -, -, null], -, H1, H2, exc(ERef)) :- new_object(H1, 'NPE', ERef, H2).
loopbody3([ok, Rs, OldRs, L, I], [Ret], H1, H2, EF) :- Ip #= I+1, loop([Rs, OldRs, L, Ip], [Ret], H1, H2, EF).
loopbody3([exc(ERef), -, -, -, -, -, H, H, exc(ERef)]).

```

Fig. 2. CLP Translation associated to bytecode of method `simp`

are of the form $r(V)$ and we use the same variable name V as in the program. E.g., argument Rs of `simp` corresponds to the method input argument. (2) Java exceptions are made explicit in the translated program, e.g., the second clauses for predicates `simp` and `loopbody2` capture the null-pointer exception (NPE). (3) Conditional statements and iteration in the source program are transformed into guarded rules and recursion in the CLP program, respectively, e.g., the for-loop has been converted to the recursive predicate `loop`. (4) Methods (like `simp`) and intermediate blocks (like `r1`) are uniformly represented by means of predicates and are not distinguishable in the translated program.

2.2 Symbolic Execution

When the imperative language does not use dynamic memory, CLP-translated programs can be executed by using the standard CLP execution mechanism with all arguments being free variables. However, in order to generate heap-allocated data structures, it is required to define heap-related operations which build the heap associated with a given path by using only the constraints induced by the visited code. Fig. 3 summarizes the CLP-implementation of the operations in [11] to create heap-allocated data structures (like `new_object` and `new_array`) and to read and modify them (like `set_field`, etc.) which use some auxiliary predicates (like deterministic versions of member `member_det`, of replace `replace_det`, and `nth0` and `replace_nth0` for arrays) which are quite standard and hence their implementation is not shown.

The intuitive idea is that the heap during symbolic execution contains two parts: the *known part*, with the cells that have been explicitly created during symbolic execution appearing at the beginning of the list, and the *unknown part*, which is a logic variable (tail of the list) in which new data can be added.

Importantly, the definition of `get_cell/3` distinguishes two situations when searching for a reference: (i) It finds it in the known part (second clause). Note the use of syntactic equality rather than unification, since references at execution time can be variables. (ii) Otherwise, it reaches the unknown part of the heap (a logic variable), and it allocates the reference (in this case a variable) there (first clause).

The heaps generated by using the operations in Fig. 3 adhere to this grammar:

$$\begin{aligned} \text{Heap} &::= [] \mid [\text{Loc} \mid \text{Heap}] & \text{Cell} &::= \text{object}(C^*, \text{Fields}^*) \mid \text{array}(T^*, \text{Num}^*, \text{Args}^*) \\ \text{Loc} &::= (\text{Num}^*, \text{Cell}) & \text{Fields} &::= [] \mid [f(\text{FN}, \text{Data}^*) \mid \text{Fields}^*] \end{aligned}$$

Observe that a heap is represented as a list of locations which are pairs made up of a unique reference and a cell, which in turn can be an object or an array. An object contains its type and its list of fields, each of them contains its signature and data contents. An array contains its type, its length and the list of its elements. An important point to note is that the list of fields of an object is always in normal form, i.e., all fields of the class are present and ordered. This is accomplished by the calls to predicate `normalize/2` within `get_field/3` and `set_field/4`, which initializes the list of fields producing the corresponding template list if it

has not been initialized yet. Note that this initialization is only produced the first time a call to `get_field/3` or `set_field/4` is performed on an object. In contrast, in [11] the list of fields can be partial (not all fields are present but just those that have been accessed) and is not ordered (fields occur in the order they are accessed during the corresponding execution). The need for normalization is motivated later in Sec. 3.1.

Example 2. Let us consider a branch of the symbolic execution of method `simp` which starts from `simp(Ain, Aout, Hin, Hout, EF)`, the empty state $\phi_0 = \langle \emptyset, \emptyset \rangle$ and which (by ignoring the call to `arraycopy` for simplicity) executes the predicates `simp1 → length → ≥ → new_array → r11 → loop1 → ≥ → true`. The subindex 1 indicates that we pick up the first rule defining a predicate for execution. As customary in CLP, a state ϕ consists of a set of bindings σ and a constraint store θ . The final state of the above derivation is $\phi_f = \langle \sigma_f, \theta_f \rangle$ with $\sigma_f = \{A_{in} = [r(Rs)], A_{out} = [r(C)], H_{in} = [(Rs, \text{array}(T, L, -))|_-, H_{out} = [(C, \text{array}(R', L, -))|H_{in}], EF = \text{ok}\}$ and $\theta_f = \{L = 0\}$. This can be read as “if the array at location `Rs` in the input heap has length 0, then it is not modified and a new array of length 0 is returned”. This derivation corresponds to the first test case in Table 3 where a graphical representation for the heap is used. For readability, in the table we have applied the store substitution to both $Heap_{in}$ and $Heap_{out}$ terms.

2.3 Method Summaries Obtained by TDG

It is well-known that the execution tree to be traversed in symbolic execution is in general infinite. This is because iterative constructs such as loops and recursion whose number of iterations depends on the input values usually induce an infinite number of execution paths when executed with unknown input values. It is therefore essential to establish a *termination criterion* which, in the context of TDG, is usually defined in terms of the so-called *coverage criterion* (see Sec. 1). Given a method m and a coverage criteria, \mathcal{C} , in what follows, we denote by `SYMBOLIC-EXECUTION(m, \mathcal{C})` the process of generating the minimal execution tree which guarantees that the test cases obtained from it will meet the given coverage criterion. The concept of *method summary* corresponds to the finite representation of its symbolic execution for a given coverage criterion.

Definition 1 (method summary). Let \mathcal{T}_m^C be the finite symbolic execution tree of method m obtained by using a coverage criterion C . Let B be the set of successful branches in \mathcal{T}_m^C and $m(Args_{in}, Args_{out}, H_{in}, H_{out}, EF)$ be its root. A method summary for m w.r.t. C , denoted \mathcal{S}_m^C , is the set of 6-tuples associated to each branch $b \in B$ of the form: $\langle \sigma(Args_{in}), \sigma(Args_{out}), \sigma(H_{in}), \sigma(H_{out}), \sigma(EF), \theta \rangle$, where σ and θ are the set of bindings and constraint store, resp., associated to b .

Each tuple in a summary is said to be a (*test*) case of the summary, denoted c , and its associated *state* ϕ_c comprises its corresponding σ and θ , also referred to as *context* ϕ_c . Intuitively, a method summary can be seen as a *complete specification* of the method for the considered coverage criterion, so that each summary case

```

new_object(H,C,Ref,H') :- build_object(C,Ob), new_ref(Ref), H' = [(Ref,Ob)|H].
new_array(H,T,L,Ref,H') :- build_array(T,L,Arr), new_ref(Ref), H' = [(Ref,Arr)|H].
length(H,Ref,L) :- get_cell(H,Ref,Cell), Cell = array(_,L,_).
get_field(H,Ref,FSig,V) :- get_cell(H,Ref,Ob), FSig = C:FN, Ob = object(T,Fields),
                           T = C, normalize(C,Fields), member_det(field(FN,V),Fields).
get_array(H,Ref,I,V) :- get_cell(H,Ref,Arr), Arr = array(_,Xs), nth0(I,Xs,V).
set_field(H,Ref,FSig,V,H') :- get_cell(H,Ref,Ob), FSig = C:FN, Ob = object(T,Fields),
                              T = C, normalize(C,Fields),
                              replace_det(Fields,field(FN,_),field(FN,V),Fields'),
                              set_cell(H,Ref,object(T,Fields'),H').
set_array(H,Ref,I,V,H') :- get_cell(H,Ref,Arr), Arr = array(T,L,Xs),
                           replace_nth0(Xs,I,V,Xs'), set_cell(H,Ref,array(T,L,Xs'),H').
-----
get_cell(H,Ref,Cell) :- var(H), !, H = [(Ref,Cell)|_].
get_cell([(Ref',Cell')|_],Ref,Cell) :- Ref == Ref', !, Cell = Cell'.
get_cell([_|RH],Ref,Cell) :- get_cell(RH,Ref,Cell).
set_cell(H,Ref,Cell,H') :- var(H), !, H' = [(Ref,Cell)|H].
set_cell([(Ref',_)|H],Ref,Cell,H') :- Ref == Ref', !, H' = [(Ref,Cell)|H].
set_cell([(Ref',Cell')|H'],Ref,Cell,H) :- H = [(Ref',Cell')|H'], set_cell(H',Ref,Cell,H'').


```

Fig. 3. Heap operations for symbolic execution [11]

corresponds to the *path constraints* associated to each finished path in the corresponding (finite) execution tree. Note that, though the specification is complete for the criterion considered, it will be, in general, a *partial specification* for the method, since the finite tree may contain incomplete branches which, if further expanded, may result in (infinitely) many execution paths.

Example 3. Table 1 shows the summary obtained by symbolically executing method `simplify` using the *block-2* coverage criterion of [1] (see Sec. 1): The summary contains 5 cases, which correspond to the different execution paths induced by calls to methods `gcd` and `abs`. For the sake of clarity, we adopt a graphical representation for the input and output heaps. Heap locations are shown as arrows labeled with their reference variable names. Split-circles represent objects of type `R` and fields `n` and `d` are shown in the upper and lower part,

Table 1. Summary of method `simplify`

A_{in}	A_{out}	$Heap_{in}$	$Heap_{out}$	EF	$Constraints$
$r(A)$	$A \rightarrow$	$\begin{array}{c} \text{F} \\ \hline 0 \end{array}$	$A \rightarrow$	$\begin{array}{c} \text{M} \\ \hline 0 \end{array}$	ok $F < 0, N = -F, M = F/N$
$r(A)$	$A \rightarrow$	$\begin{array}{c} \text{F} \\ \hline 0 \end{array}$	$A \rightarrow$	$\begin{array}{c} 1 \\ \hline 0 \end{array}$	ok $F > 0$
$r(A)$	$A \rightarrow$	$\begin{array}{c} 0 \\ \hline 0 \end{array}$	$A \rightarrow$	$\begin{array}{c} 0 \\ \hline 0 \end{array}$	$B \rightarrow$  exc(B)
$r(A)$	$A \rightarrow$	$\begin{array}{c} \text{F} \\ \hline \text{G} \end{array}$	$A \rightarrow$	$\begin{array}{c} \text{M} \\ \hline \text{N} \end{array}$	ok $G < 0, F \bmod G = 0, K = -G, M = F/K, N = G/K$
$r(A)$	$A \rightarrow$	$\begin{array}{c} \text{F} \\ \hline \text{G} \end{array}$	$A \rightarrow$	$\begin{array}{c} \text{M} \\ \hline 1 \end{array}$	ok $G > 0, F \bmod G = 0, M = F/G$

<pre> compose_summary(Call) :- Call = ..[M,A_{in},A_{out},H_{in},H_{out},EF], summary(M,SA_{in},SA_{out},SH_{in},SH_{out},SEF,σ), SA_{in} = A_{in}, SA_{out} = A_{out}, SEF = EF, compose_hin(H_{in},SH_{in}), compose_hout(H_{in},SH_{out},H_{out}), load_store(σ). </pre>	<pre> compose_hin(.,SH) :- var(SH), !. compose_hin(H,[(R,Cell)]SH) :- get_cell(H,R,Cell'), Cell' = Cell, compose_hin(H,SH). compose_hout(H,SH,H) :- var(SH), !. compose_hout(H_{in},[(Ref,Cell)]SH_{out},H_{out}) :- set_cell(H_{in},Ref,Cell,H'), compose_hout(H',SH_{out},H_{out}). </pre>
---	--

Fig. 4. The composition operation

respectively. Exceptions are shown as starbursts, like in the special case of the fraction “0/0”, for which an arithmetic exception (AE) is thrown due to a division by zero. In summary examples of Tables 2 and 3, split-rectangles represent arrays, with the length of the array in the upper part and its list of values (in Prolog syntax) in the lower one.

In a subsequent stage, it is possible to produce actual values from the obtained path constraints (e.g., by using labeling mechanisms in standard *clpfd* domains) therefore obtaining executable test cases. However, this is not an issue of this paper and we will rely on the method summaries only in what follows.

3 A Compositional CLP-Based TDG Approach

The goal of this section is to study the compositionality of the CLP-based approach to TDG of imperative languages presented in the previous section.

3.1 Composition in Symbolic Execution

Let us assume that during the symbolic execution of a method m , there is a method invocation to p within a state ϕ . In the context of our CLP approach, the challenge is to define a composition operation so that, instead of symbolically executing p its (previously computed) summary \mathcal{S}_p can be reused. For this, TDG for m should produce the same results regardless of whether we use a summary for p or we symbolically execute p within TDG for m , in a non-compositional way.

Fig. 4 shows such a composition operation (predicate `compose_summary/1`). The idea is therefore to replace, during symbolic execution, every method invocation to p by a call `compose_summary(p(...))` when there is a summary available for it. Intuitively, given the variables of the call to p , with their associated state ϕ , `compose_summary/1` produces, on backtracking, a branch for each *compatible* case $c \in \mathcal{S}_p$, composes its state ϕ_c with ϕ and produces a new state ϕ' to continue the symbolic execution with. We assume that the summary for a method p is represented as a set of facts of the form `summary(p,SAin,SAout,SHin,SHout,SEF,θ)`. Roughly speaking, state ϕ_c is *compatible* with ϕ if: 1) the bindings and constraints on the arguments can be conjoined, and 2) the structures of the input

heaps match. This means that, for each location which is present in both heaps, its associated cells match, which in turn requires that their associated bindings and constraints can be conjoined. Note that compatibility of a case is checked on the fly, so that if ϕ is not compatible with ϕ_c some call in the body of `compose_summary/1` will fail.

As it can be observed by looking at the code of `compose_summary/1`, the input and output arguments, and the exception flags are simply unified, while the constraint store θ is trivially incorporated by means of predicate `load_store/1`. However, the heaps require a more sophisticated treatment, mainly due to the underlying representation of sets (of objects) as Prolog lists. Predicate `compose_hin/2` composes the input heap of the summary case SH_{in} with the current heap H_{in} , producing the composed input heap in H_{in} . To accomplish this, `compose_hin/2` traverses each cell in SH_{in} , and: 1) if its associated reference is not present in H_{in} (first rule of `get_cell/3` succeeds), it is added to it, 2) if it is present in H_{in} (second rule of `get_cell/3` succeeds) then the cells are unified. This is possible since we are assuming that every object that arises in the heap during symbolic execution has its list of fields in normal form (see Sec. 2.2). This allows using just unification ($\text{Cell} = \text{Cell}'$) for the aim of matching cells.

Similarly, `compose_hout/3` composes the output heap of the summary case SH_{out} with the current heap H_{in} , producing the composed output heap in H_{out} . As can be seen in Figure 4, `compose_hout/3` traverses each cell in SH_{out} and, if its associated reference is not present in H (first rule of `set_cell/4` succeeds), then it is added to it. Otherwise (second rule of `set_cell/4` succeeds) it overwrites the current cell. In both cases, `set_cell/4` produces a new heap H' which is passed as first argument to the recursive call to `compose_hout/3`. This process continues until there are no more cells in SH_{out} , in which case the current heap is returned. Again this is possible thanks to the normal form of object fields.

As noticed before, further features of imperative languages not considered in this paper, such as inheritance and pointer aliasing, can be handled by `compose_summary/1` for free by just using the corresponding extensions of `get_cell/3` and `set_cell/4` defined in [11].

Example 4. When symbolically executing `simp`, the call `simplify(Ain,Aout,Hin,Hout,EF)` arises in one of the branches with state $\sigma = \{\text{A}_{\text{in}} = [\text{r}(\text{E0})], \text{A}_{\text{out}} = [], \text{H}_{\text{in}} = [(0, \text{array}(\text{R}', \text{L}, [\text{E0}|_]))], (\text{Rs}, \text{array}(\text{R}', \text{L}, [\text{E0}|_])) | \text{RH}_{\text{in}}\}$ and $\theta = \{\text{L} \geq 0\}$. The composition of this state with the second summary case of `simplify` succeeds and produces the state $\sigma' = \sigma \cup \{\text{E0} = \text{B}, \text{RH}_{\text{in}} = [(\text{B}, \text{ob}(\text{R}', [\text{field}(\text{n}, \text{F}), \text{field}(\text{d}, 0))])|_], \text{H}_{\text{out}} = [\dots, (\text{B}, \text{ob}(\text{R}', [\text{field}(\text{n}, 1), \text{field}(\text{d}, 0))])|_]\}$ and $\theta' = \{\text{L} \geq 0, \text{F} > 0\}$. The dots in H_{out} denote the rest of the cells in H_{in} .

3.2 Approaches to Compositional TDG

In order to perform compositional TDG, two main approaches can be considered:

Table 2. Summary of method `arraycopy`

A_{in}	A_{out}	$Heap_{in}$	$Heap_{out}$	EF	$Constraints$
$[X, Y, 0]$	H	H	H	ok	\emptyset
$[r(A), null, Z]$	$A \rightarrow \boxed{\begin{array}{c} L \\ [V] \end{array}}$	$A \rightarrow \boxed{\begin{array}{c} L \\ [V] \end{array}}$	$A \rightarrow \boxed{\begin{array}{c} L \\ [V] \end{array}} B \rightarrow \text{NPE}$	exc(B)	$Z > 0, L > 0$
$[null, Y, Z]$	H	$A \rightarrow \text{NPE}$	$A \rightarrow \text{NPE}$	exc(A)	$Z > 0$
$[X, Y, Z]$	H	$A \rightarrow \text{NPE}$	$A \rightarrow \text{NPE}$	exc(A)	$Z < 0$
$[r(A), r(B), 1]$	$A \rightarrow \boxed{\begin{array}{c} L1 \\ [V] \end{array}} B \rightarrow \boxed{\begin{array}{c} L2 \\ [V2] \end{array}}$	$A \rightarrow \boxed{\begin{array}{c} L1 \\ [V] \end{array}} B \rightarrow \boxed{\begin{array}{c} L2 \\ [V] \end{array}}$	$A \rightarrow \boxed{\begin{array}{c} L1 \\ [V] \end{array}} B \rightarrow \boxed{\begin{array}{c} L2 \\ [V] \end{array}}$	ok	$L1 > 1, L2 > 0$

Context-sensitive. Starting from an entry method m (and possibly a set of pre-conditions), TDG performs a top-down symbolic execution such that, when a method call p is found, its code is executed from the actual state ϕ . In a context-sensitive approach, once a method is executed, we store the summary computed for p in the context ϕ . If we later reach another call to p within a (possibly different) context ϕ' , we first check if the stored context is sufficiently general. In such case, we can adapt the existing summary for p to the current context ϕ' (by relying on the operation in Fig. 4). At the end of each execution, it can be decided which of the computed (context-sensitive) summaries are stored for future use. In order to avoid the problems of computing summaries which end up being not sufficiently general, in the rest of this paper we focus in the context-insensitive approach presented below.

Context-insensitive. Another possibility is to perform the TDG process in a context-insensitive way. Algorithm 1 presents this strategy, by abstracting some implementation-related details. Intuitively, the algorithm proceeds in the following steps. First, it computes the call graph (line 3) for the entry method m_P of the program under test, which gives us the set of methods that must be tested. The strongly connected components (SCCs for short) for such graph are then computed in line 4. SCCs are then traversed in reverse topological order starting from an SCC which does not depend on any other (line 6). The idea is that each SCC is symbolically executed from its entry m_{scc} w.r.t. the most general context (i.e., *true*) (line 8). If there are several entries to the same SCC, the process is repeated for each of them. Hence, it is guaranteed that the obtained summaries can always be adapted to more specific contexts.

In general terms, the advantages of the context-insensitive approach are that composition can always be performed and that only one summary needs to be stored per method. However, since no context information is assumed, summaries can contain more test cases than necessary and can be thus more expensive to obtain. In contrast, the context-sensitive approach ensures that only the required information is computed, but it can happen that there are several invocations to the same method that cannot reuse previous summaries (because the associated contexts are not sufficiently general). In such case, it is more efficient to obtain the summary without assuming any context.

Algorithm 1. Context-insensitive compositional TDG

Input: Program \mathcal{P} , Coverage criterion \mathcal{C}
Output: Test suite \mathcal{T} for program \mathcal{P} w.r.t. \mathcal{C}

- 1: **procedure** BOTTOM-UP-TDG(\mathcal{P}, \mathcal{C})
- 2: Let $m_{\mathcal{P}}$ be the entry method of \mathcal{P}
- 3: $\mathcal{G} \leftarrow \text{callGraph}(m_{\mathcal{P}})$
- 4: $\text{SCC} \leftarrow \text{stronglyConnectedComponents}(\mathcal{G})$
- 5: $\text{SCC}' \leftarrow \text{buildTopologicalOrderList}(\text{SCC})$
- 6: **for all** $\text{scc} \in \text{SCC}'$ **do**
- 7: **for all** $m_{\text{scc}} \in \text{entryMethods}(\text{scc})$ **do**
- 8: $S_{\mathcal{C}}^{m_{\text{scc}}} \leftarrow \text{SYMBOLIC-EXECUTION}(m_{\text{scc}}, \mathcal{C})$
- 9: **end for**
- 10: **end for**
- 11: **end procedure**

3.3 Handling Native Code during Symbolic Execution

An inherent limitation of symbolic execution is the handling of native code, i.e., code implemented in another (lower-level) language. Symbolic execution of native code is not possible since the associated code is not available and it can only be handled as a black box. In the context of hybrid approaches to TDG which combine symbolic and concrete execution, a solution is concolic execution [9] where concrete execution is performed on random inputs and path constraints are collected at the same time; native code is executed for concrete values. Although we believe that such approach could be also adapted to our CLP framework, we concentrate here on a purely symbolic approach. In this case, the only possibility is to model the behavior of the native code by means of specifications. Such specifications can be in turn treated as summaries for the corresponding native methods. They can be declared by the code provider or automatically inferred by a TDG tool for the corresponding language. Interestingly, the composition operator uses them exactly in the same way as it uses the summaries obtained by applying our own symbolic execution mechanism. Let us see an example.

Example 5. Assume that method `arraycopy` is native. A method summary for `arraycopy` can be provided, as shown in Table 2, where we have (manually) specified five cases: the first one for arrays of length zero, the second and third ones for null arrays, the fourth one for a negative length, and finally a normal execution of non-null arrays. Now, by using our compositional reasoning, we can continue symbolic execution for `simp` by composing the specified summary of `arraycopy` within the actual context. In Table 3, we show the entire summary of method `simp` for a *block-2* coverage criterion obtained by relying on the summaries for `simplify` and `arraycopy` shown before.

A practical question is how method summaries for native code should be provided. A standard way is to use assertions (e.g., in JML in the case of Java) which could be parsed and easily transformed into our Prolog syntax.

3.4 Compositionality of Different Coverage Criteria

Though we have presented in Sec. 3.1 above a mechanism for reusing existing summaries during TDG, not all coverage criteria behave equally well w.r.t. compositionality. A coverage criterion C is *compositional* if whenever performing TDG of a method m w.r.t. C , if we use a previously computed summary for a method p w.r.t. C in a context which is sufficiently general, the results obtained for m preserve criterion C . In other words, if a criterion is compositional, we do not lose the required coverage because of using summaries.

Unfortunately, not all coverage criteria are compositional. For example, statement coverage is not compositional, as illustrated in the example below.

Example 6. Consider the following simple method:

$$p(\text{int } a, \text{int } b) \{ \text{if } (a > 0 \parallel b > 0) \text{ S}; \}$$

where S stands for any statement, and the standard shortcut semantics for Java boolean expressions is used. This means that as soon as the expression has a definite *true* or *false* value, it is not further evaluated. In our case, once the subexpression $a > 0$ takes the value *true*, the whole condition definitely takes the value *true*, the subexpression $b > 0$ is not evaluated, and S is executed.

If assuming the top (most general) context, a summary with a single case $\{a > 0\}$ is sufficient to achieve statement coverage. Consider now that p is called from an outer scope with a more restricted context in which $a \leq 0$. Then, using such summary instead of performing symbolic execution of p does not preserve statement coverage, since it is not guaranteed that statement S is visited. It depends on the particular value picked for b for testing, which is unconstrained in the summary. If the value for b is picked to be greater than zero, statement coverage is satisfied, but not otherwise. Note that by considering a context where $a \leq 0$ from the beginning, a summary with the single entry $\{b > 0\}$ would be computed instead.

Table 3. Summary of method `simp`

A_{in}	A_{out}	$Heap_{in}$	$Heap_{out}$	EF	$Constraints$
$r(A)$	$r(B)$	$A \rightarrow \begin{array}{ c } \hline 0 \\ \hline \end{array}$	$A \rightarrow \begin{array}{ c } \hline 0 \\ \hline \end{array} B \rightarrow \begin{array}{ c } \hline 0 \\ \hline \end{array}$	ok	\emptyset
null	X	H	$A \rightarrow \text{NPE}$	exc(A)	\emptyset
$r(A)$	$r(C)$	$A \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array} B \rightarrow \begin{array}{ c } \hline F \\ \hline \end{array}$	$A \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array} B \rightarrow \begin{array}{ c } \hline M \\ \hline \end{array} C \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array}$	ok	$F < 0, K = -F, M = F/K$
$r(A)$	$r(C)$	$A \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array} B \rightarrow \begin{array}{ c } \hline F \\ \hline \end{array}$	$A \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array} B \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array} C \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array}$	ok	$F > 0$
$r(A)$	X	$A \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array} B \rightarrow \begin{array}{ c } \hline 0 \\ \hline \end{array}$	$A \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array} B \rightarrow \begin{array}{ c } \hline 0 \\ \hline \end{array} C \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array} D \rightarrow \text{NPE}$	exc(D)	\emptyset
$r(A)$	$r(C)$	$A \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array} B \rightarrow \begin{array}{ c } \hline F \\ \hline \end{array}$	$A \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array} B \rightarrow \begin{array}{ c } \hline M \\ \hline \end{array} C \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array}$	ok	$G < 0, F \bmod G = 0, K = -G, M = F/K, N = G/K$
$r(A)$	$r(C)$	$A \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array} B \rightarrow \begin{array}{ c } \hline F \\ \hline \end{array}$	$A \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array} B \rightarrow \begin{array}{ c } \hline M \\ \hline \end{array} C \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array}$	ok	$G > 0, F \bmod G = 0, M = F/G$
$r(A)$	X	$A \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array}$	$A \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array} C \rightarrow \begin{array}{ c } \hline 1 \\ \hline \end{array} B \rightarrow \text{NPE}$	exc(B)	\emptyset

As this example illustrates, a challenge in compositional reasoning is to preserve coverage when using summaries previously computed for a context ϕ which is sufficiently general, but not identical to ϕ' , the one which appears during the particular invocation of the method. More precisely, compositionality of coverage criteria requires that the following property holds: given a summary S obtained for p in a context ϕ w.r.t. C , a summary S' for a more restricted context ϕ' can be obtained by removing from S those entries which are incompatible with context ϕ' .

For instance, the block- k coverage criterion used in the examples of the paper is compositional. This is because there is a one to one correspondence between entries in the summary and non-failing branches in the symbolic execution tree obtained for ϕ . If we are now in a more restricted context ϕ' , those branches which become failing branches are exactly those whose precondition is incompatible with ϕ' . Therefore, we obtain identical results by working at the level of the symbolic execution tree or that of the entries in the summary.

Table 4. Benchmarks

Benchmark	NMs	RCs	RMs	RIs	T_{dec}
NodeStack	6	3	12	94	32
ArrayStack	7	3	11	103	39
NodeQueue	6	3	15	133	48
NodeList	19	9	33	449	277
DoublyLinkedList	13	2	20	253	107
SortedListInt	6	2	8	155	58
SLPriorityQueue	12	14	42	515	330
BinarySearchTree	14	15	54	717	620
SLIntMaxNode	9	2	12	232	99
SearchTreeInt	9	1	10	189	75

In fact, we can classify coverage criteria into two categories: local and global. A criterion is *local* when the decision on whether the path should be included or excluded in the summary can be taken by looking at the corresponding path only. A criterion is *global* when we need to look at other paths before determining whether to include a given path in the summary or not. As examples, both block- k and depth- k are local, whereas statement coverage is global: a given path is not needed if it does not visit statements not covered by any of the previously considered paths.

In general, local criteria are compositional, whereas global criteria are not. The reason for such non-compositionality is that we may decide not to include certain paths which are not required for achieving the criterion in a context ϕ but which are needed in a more specific context ϕ' , since other paths which achieved the criterion are now incompatible with the context and have been removed from the summary.

3.5 Reusing Summaries Obtained for Different Criteria

In the discussion about compositionality presented in Sec. 3.4 above, we always assume that the same criterion C is used both for m and the summary of p . Another interesting practical question is: given a summary computed for p w.r.t. a criteria C' , can we use it when computing test cases for m w.r.t. a criteria C ?

As an example, by focusing on block- k , assume that C' corresponds to $k = 3$ and C to $k = 2$. We can clearly adapt the summaries obtained for $k = 3$ to the current criteria $k = 2$. Even more, if one uses the whole summary for $k = 3$, the required coverage $k = 2$ is ensured, although unnecessary test cases are introduced. On the contrary, if C' corresponds to $k = 2$ and C to $k = 3$ the coverage criterion is not preserved for m . However, this can be acceptable when we would like to perform TDG of different levels to different parts of the code, depending on their size, relevance, level of trust, etc. For instance, code which is safety-critical can be more exhaustively tested using coverage criteria that ensures a higher degree of coverage. In contrast, code which is more stable (e.g., library methods) can be tested using more lightweight coverage criteria.

Another issue is what happens when the existing summary is for a completely unrelated criterion. There, it is not possible to guarantee that the criterion for m is guaranteed. Nevertheless, such summaries can be used for obtaining information on the output states of p in order to be able to continue the symbolic execution of m after the calls to p terminate. This is especially relevant when p is native, since in that case performing symbolic execution instead of using the summary is not an alternative.

Table 5. Experimental results

Benchmark	Cov. Crit.	Non-Compositional				Compositional						Gains ΔT_{tdg}
		T_{tcg}	N	US	CC	T_{tcg}	N	US	CC	SG	SC	
NodeStack	block-2	6.7	9	112	100%	10.3	9	94	100%	11	12	0.65
	block-3	6.7	9	112	100%	6.7	9	94	100%	12	11	1.00
ArrayStack	block-2	13.0	15	203	100%	13.0	15	161	100%	10	10	1.00
	block-3	13.3	15	203	100%	13.0	15	161	100%	10	10	1.03
NodeQueue	block-2	10.3	15	160	100%	13.0	15	149	100%	13	13	0.79
	block-3	10.3	15	160	100%	13.3	15	149	100%	13	13	0.78
NodeList	block-2	120.0	102	1856	96%	103.7	102	684	96%	52	28	1.16
	block-3	130.0	102	1856	96%	110.3	102	684	96%	28	52	1.18
DoublyLinkedList	block-2	160.3	116	5060	99%	130.3	116	1967	99%	39	12	1.23
	block-3	200.0	133	6068	99%	160.3	133	2164	99%	12	51	1.25
SortedListInt	block-2	133.3	49	2781	100%	77.0	49	775	100%	22	5	1.73
	block-3	4233.7	447	63593	100%	1713.0	447	6542	100%	5	176	2.47
SLPriorityQueue	block-2	533.3	266	5828	94%	566.7	266	1195	94%	114	62	0.94
	block-3	786.7	350	7910	94%	856.3	350	1369	94%	62	128	0.92
BinarySearchTree	block-2	693.0	307	9210	96%	706.3	307	5203	96%	463	63	0.98
	block-3	1526.7	559	21162	96%	1486.7	559	11143	96%	63	733	1.03
SLIntMaxNode	block-2	299.7	109	5012	100%	230.0	109	1172	100%	27	8	1.30
	block-3	28150.3	2957	328096	100%	20419.7	2957	34770	100%	8	520	1.38
SearchTreeInt	block-2	279.7	90	5707	100%	143.0	90	726	100%	33	8	1.96
	block-3	45003.0	5140	553536	100%	20236.7	5140	12980	100%	8	232	2.22

4 Experimental Results

We have implemented our proposed compositional approach using the context-insensitive algorithm in Sec. 3.2 within PET [2], an automatic TDG tool for

Java bytecode, which is available for download and for online use through its web interface at <http://costa.ls.fi.upm.es/pet>.

In this section we report on some experimental results which aim at demonstrating the applicability and effectiveness of compositional TDG. As benchmarks, we consider a set of classes implementing some traditional data structures ranging from simple stacks and queues to sorted and non-sorted linked lists (both singly and doubly linked lists), priority queues and binary search trees. Some benchmarks are taken from the `net.datastructures` library [12], a well-known library of algorithms and data-structures for Java. Table 4 shows the list of benchmarks we have used, together with some static information: the number of methods for which we have generated test cases (column **NMs**); the number of reachable classes, methods and Java bytecode instructions (columns **RCs**, **RM**s and **RI**s) (excluding code of the Java libraries) and the time taken by PET to decompile the bytecode to CLP (\mathbf{T}_{dec}), including parsing and loading all reachable classes. All times are in milliseconds and are obtained as the arithmetic mean of five runs on an Intel(R) Core(TM)2 Quad CPU Q9300 at 2.5GHz with 1.95GB of RAM, running Linux 2.6.26 (Debian lenny).

Table 5 aims at comparing the performance and effectiveness of the compositional approach against the non-compositional one by using two coverage criteria, block-2 and block-3. In general the latter one expands much further the symbolic execution tree, thus allowing us to compare scalability issues. For each run, we measure, for both the compositional and non-compositional approaches, the time taken by PET to generate the test cases (column \mathbf{T}_{tcg}), the number of obtained test cases (**N**), the number of unfolding steps performed during the symbolic execution process (**US**), and the *code coverage* (**CC**). In the case of the compositional approach we also measure the number of generated summaries (**SG**) and summary compositions performed (**SC**). The code coverage measures, given a method, the percentage of bytecode instructions which are exercised by the obtained test cases, among all reachable instructions (including all transitively called methods). This is a common measure in order to reason about the effectiveness of the TDG. As expected, the code coverage is the same in both approaches, and so is the number of obtained test cases. Otherwise, this would indicate a bug in the implementation. The last column ($\Delta\mathbf{T}_{tdg}$) shows the speedup of the compositional approach computed as X/Y where X is the \mathbf{T}_{tcg} value of the non-compositional approach and Y the \mathbf{T}_{tcg} value of the compositional one.

By looking at the gains, we observe that the compositional approach outperforms the non-compositional one in most benchmarks. Let us observe also that, in general, the further the symbolic execution tree is expanded (i.e., when the block-3 criterion is used), the higher the gains are. There are, however, cases where the performance of the compositional approach is equal to, or even worse than, that of the non-compositional one. Importantly, those cases usually correspond to very simple methods whose complexity is not enough so that the overhead of applying the compositional scheme pays off.

After a careful study of the obtained results, we conclude that there can be many factors that influence the performance of the compositional approach.

The most important ones are: the complexity of the program under test (especially that of its call graph and its strongly connected components), the constraint solving library, and, the kind of constraint-based operations performed and, in particular, whether they are arithmetic constraints or heap related operations. In this direction, we have carried out the following experiment. Given a method p which simply calls repeatedly (three or four times) method q , we consider two versions of it: q_1 which performs both heap and arithmetic operations, and q_2 with arithmetic operations only. This allows us to detect whether the kind of constraint-based operations performed influences the performance of compositional TDG. As expected, with q_1 compositional TDG improves notably (two or even three times faster) over non-compositional TDG. Surprisingly, with q_2 the performance of compositional TDG is basically the same (or even worse). This explains the lack of improvements in some of our benchmarks. The reason is that the cost of the TDG process is totally dominated by the constraint solving operations, in this case by the `clpfd` solver. Interestingly, if we simplify by hand the constraints on the summary of q_2 we do get significant improvements with compositional TDG. This illustrates the flexibility of the approach in the sense that, provided that a summary is available for a method, it can be worth spending resources in simplifying the constraint stores. Once this is done, they will be used every time a call to the method is found and thus producing a performance improvement. On the other hand, this demonstrates that compositional TDG can significantly benefit from more efficient constraint solving libraries.

Overall, our experimental results support our claim that compositional TDG improves over non-compositional TDG in terms of scalability. Let us observe that, in general, the benchmarks where the improvements are higher correspond to those for which a larger number of unfolding steps (column **US**) is required. We have seen also that such improvement could be higher by using a more efficient constraint solving library. It remains as future work to experiment in such direction.

5 Conclusions

Much effort has been devoted in the area of symbolic execution to improve scalability and three main approaches co-exist which, in a sense, complement each other. Probably, the most widely used is abstraction, a well-known technique to reduce large data domains of a program to smaller domains (see [17]). Another scaling technique which is closely related to abstraction is path merging [4,16], which consists in defining points where the merging of symbolic execution should occur. In this paper, we have focused on yet another complementary approach, *compositional* symbolic execution, a technique widely used in static analysis but notably less common in the field of TDG. We have presented a CLP-based approach to TDG of imperative languages which can be applied in a compositional way. This can be important in order to make the approach scalable and, as we have seen, also provides a practical way of dealing with native code.

Acknowledgements

This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS* project.

References

1. Albert, E., Gómez-Zamalloa, M., Puebla, G.: Test Data Generation of Bytecode by CLP Partial Evaluation. In: Hanus, M. (ed.) LOPSTR 2008. LNCS, vol. 5438, pp. 4–23. Springer, Heidelberg (2009)
2. Albert, E., Gómez-Zamalloa, M., Puebla, G.: PET: A Partial Evaluation-based Test Case Generation Tool for Java Bytecode. In: ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM), Madrid, Spain, January 2010, pp. 25–28. ACM Press, New York (2010)
3. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 367–381. Springer, Heidelberg (2008)
4. Arons, T., Elster, E., Ozer, S., Shalev, J., Singerman, E.: Efficient symbolic simulation of low level software. In: DATE, pp. 825–830. IEEE, Los Alamitos (2008)
5. Clarke, L.A.: A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.* 2(3), 215–222 (1976)
6. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 169–188. Springer, Heidelberg (2007)
7. Ferguson, R., Korel, B.: The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.* 5(1), 63–86 (1996)
8. Godefroid, P.: Compositional dynamic test generation. In: Hofmann, M., Felleisen, M. (eds.) POPL, pp. 47–54. ACM, New York (2007)
9. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: Sarkar, V., Hall, M.W. (eds.) PLDI, pp. 213–223. ACM, New York (2005)
10. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology* 51, 1409–1427 (2009)
11. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Test Case Generation for Object-Oriented Imperative Languages in CLP. In: Theory and Practice of Logic Programming, 26th Int’l. Conference on Logic Programming (ICLP 2010) Special Issue, vol. 10(4-6), pp. 659–674 (July 2010)
12. Goodrich, M.T., Tamassia, R., Zamore, R.: The net.datastructures package, version 3 (2003), <http://net3.datastructures.net>
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 (LNAI), 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. King, J.C.: Symbolic execution and program testing. *Commun. ACM* 19(7), 385–394 (1976)
16. Kölbl, A., Pixley, C.: Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming* 33(6), 645–666 (2005)
17. Lakhnech, Y., Bensalem, S., Berezin, S., Owre, S.: Incremental verification by abstraction. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 98–112. Springer, Heidelberg (2001)
18. Meudec, C.: Atgen: Automatic test data generation using constraint logic programming and symbolic execution. *Softw. Test., Verif. Reliab.* 11(2), 81–96 (2001)
19. 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)
20. Pășăreanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.* 11(4), 339–353 (2009)
21. Schrijvers, T., Degraeve, F., Vanhoof, W.: Towards a framework for constraint-based test case generation. In: De Schreye, D. (ed.) LOPSTR 2009. LNCS, vol. 6037, pp. 128–142. Springer, Heidelberg (2010)
22. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. *ACM Comput. Surv.* 29(4), 366–427 (1997)