Towards Compositional CLP-based Test Data Generation for Imperative Languages

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

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

1 Introduction

Test data generation (TDG) is the process of 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; *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 [20]), our work focuses on *glass-box* testing, where test-cases are obtained from the concrete program in contrast to *black-box* testing, where they are deduced from a specification of the program. Also, our focus is on *static* testing, where we assume no knowledge about the input data, in contrast to *dynamic* approaches [6,12] 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 [5,16,17,13,11,19], 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* [17,11,19] in order to: handle the constraints systems by solving the feasibility of paths and, afterwards, to instantiate the input variables. For instance, a symbolic JVM machine which integrates several constraints solvers has been designed in [17] 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 backtrack, as without knowledge about the input data, the execution engine might need to execute more than one path.

In recent work [10], we have proposed a CLP-based approach to TDG of imperative programs which consists 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 [9] 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 which, during TDG, take care of constructing complex data structures with unbounded data (e.g., recursive data structures). Such operations can be implemented in CLP [10].

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

```
class R {                             class A {
  int n; int d;                         static int abs(int x){
  void simplify(){                        if (x >= 0) return x;
    int gcd = A.gcd(n,d);                 else return -x;
    n = n/gcd; d = d/gcd;}              }
  static R[] simp(R[] rs){              static int gcd(int a,int b){
    int length = rs.length;               int res;
    R[] oldRs = new R[length];            while (b != 0){
    arraycopy(rs,oldRs,length);             res = a%b; a = b; b = res;}
    for (int i = 0; i < length; i++)      return abs(a);
      rs[i].simplify();                 }
    return oldRs;}}                   }
```

Fig. 1. Java source of working example

the size of their associated constraints (see [18]). Currently, one of the main challenges of symbolic execution (and thus of glass-box TDG) is to scale up to larger applications. While compositionality has been applied in many areas of static analysis to improve scalability, it is less widely used in TDG (some notable exceptions in the context of dynamic testing are [7,3]). 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 p, 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. By test cases or method summary, we refer to the set of path constraints obtained by symbolically executing method p using a certain coverage criterion.

Having a compositional TDG approach in turn facilitates the handling of *native code* during symbolic execution, i.e., code which is implemented in a different language. This is achieved by modeling the behavior of native code as preconditions in the input state and postconditions in the output state. Such model 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 [18]). Indeed, both scaling techniques and handling native code are considered main challenges in the fields of symbolic execution and 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 [10]. 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 [10]. Also, although our approach is not tied to any particular imperative language, as regards dynamic memory, we assume a Java-like language. In [9], 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 [10] 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},

```
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,EF'), r1([EF',r(Rs),r(OldRs),Len],[Rt],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,EF'),
        loopbody3([EF',Rs,OldRs,Length,I],[Ret],H2,H3,EF).
loopbody2([_,_,_,_,null],_,H1,H2,exc(ERef)) :- new_object(H1,'NPE',ERef,H2).

loopbody3([ok,Rs,OldRs,L,I],[Ret],H1,H2,EF) :- I' #= I+1, loop([Rs,OldRs,L,I'],[Ret],H1,H2,EF).
loopbody3([exc(ERef),_,_,_,_],_,H,H,exc(ERef)).
```

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

and an exceptional 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:

$$\begin{array}{rcl}
\mathit{Clause} & ::= & \mathit{Pred}\ (\mathit{Args}_{in},\mathit{Args}_{out},H_{in},H_{out},\mathit{ExFlag})\ \text{:-}\ [G,]B_1,B_2,\ldots,B_n. \\
G & ::= & \mathit{Num^*}\ \mathit{ROp}\ \mathit{Num^*} \mid \mathit{Ref}_1^*\ \backslash{==}\ \mathit{Ref}_2^* \\
B & ::= & \mathit{Var}\ \#{=}\ \mathit{Num^*}\ \mathit{AOp}\ \mathit{Num^*} \mid \mathit{Pred}\ (\mathit{Args}_{in},\mathit{Args}_{out},H_{in},H_{out},\mathit{ExFlag}) \mid \\
 & & \mathsf{new\_object}(H,C^*,\mathit{Ref}^*,H) \mid \mathsf{new\_array}(H,T,\mathit{Num}^*,\mathit{Ref}^*,H) \mid \mathsf{length}(H,\mathit{Ref}^*,\mathit{Var}) \mid \\
 & & \mathsf{get\_field}(H,\mathit{Ref}^*,\mathit{FSig},\mathit{Var}) \mid \mathsf{set\_field}(H,\mathit{Ref}^*,\mathit{FSig},\mathit{Data}^*,H) \mid \\
 & & \mathsf{get\_array}(H,\mathit{Ref}^*,\mathit{Num}^*,\mathit{Var}) \mid \mathsf{set\_array}(H,\mathit{Ref}^*,\mathit{Num}^*,\mathit{Data}^*,H)
\end{array}$$

$$\begin{array}{rclrcl}
\mathit{Pred} & ::= & \mathit{Block} \mid \mathit{MSig} & \mathit{ROp} & ::= & \#> \mid \#< \mid \#>= \mid \#=< \mid \#= \mid \#\backslash= \\
\mathit{Args} & ::= & [\,] \mid [\mathit{Data}^* | \mathit{Args}] & \mathit{AOp} & ::= & + \mid - \mid * \mid / \mid \mathit{mod} \\
\mathit{Data} & ::= & \mathit{Num} \mid \mathit{Ref} \mid \mathit{ExFlag} & T & ::= & \mathit{bool} \mid \mathit{int} \mid C \mid \mathit{array(T)} \\
\mathit{Ref} & ::= & \mathit{null} \mid \mathit{r(Var)} & \mathit{FSig} & ::= & C{:}\mathit{FN} \\
\mathit{ExFlag} & ::= & \mathit{ok} \mid \mathit{exc(Var)} & H & ::= & \mathit{Var}
\end{array}$$

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 exceptional flag. Reference variables 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`) and the second one for `r1` which a negative array size exception (`NASE`). (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 corresponds 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 [10] 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_nth 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 which appear 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).

Example 2. Let us consider a branch of the symbolic execution of method `simp` which starts from `simp(`$\mathsf{A_{in}}$`,`$\mathsf{A_{out}}$`,`$\mathsf{H_{in}}$`,`$\mathsf{H_{out}}$`,EF)`, the empty state $\phi_0 = \langle\emptyset,\emptyset\rangle$ and which (by ignoring the call to `arraycopy` for simplicity) executes the predicates $\mathtt{simp}_1\rightarrow$ length$\rightarrow\ \geq\rightarrow$ new_array$\rightarrow$ $\mathtt{r1}_1 \rightarrow \mathtt{loop}_1 \rightarrow\ \geq\rightarrow$ `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 $\sigma_f = \{\mathsf{A_{in}} = [\mathsf{r(Rs)}], \mathsf{A_{out}} = [0], \mathsf{H_{in}} = [(\mathsf{Rs}, \mathsf{array}(\mathsf{T}, \mathsf{L}, \_))|\_], \mathsf{H_{out}} = [(0, \mathsf{array}('\mathsf{R}', \mathsf{L}, \_))|\mathsf{H_{in}}], \mathsf{EF} = \mathsf{ok}\}$ and $\theta_f = \{\mathsf{L} = 0\}$. 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. 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 2 where a graphical representation for the heap is used.

2.3 Method Summaries obtained by TDG

It is well-known that, in symbolic execution, the execution tree to be traversed is in general infinite. This is because iterative constructs such as loops and recursion usually induce an infinite number of execution paths when executed without 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). The concept of *method summary* corresponds to the finite representation of its symbolic execution for a given coverage criterion.

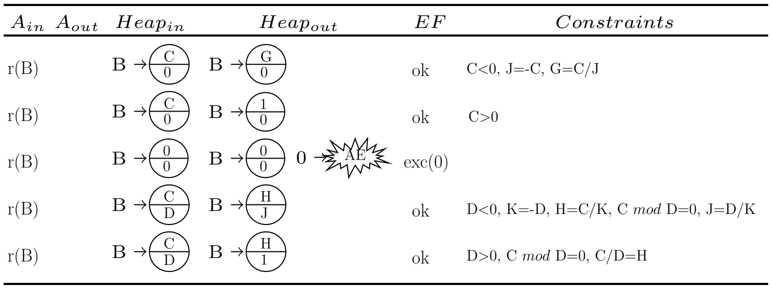
Definition 1 (method summary). *Let $\mathcal{T}_m^C$ be the finite symbolic execution tree of method m obtained by using a coverage criterion C. Let B be the set of successful branches in $\mathcal{T}_m^C$ and $m(Args_{in},Args_{out},H_{in},H_{out},EF)$ be its root. A* method summary *for m w.r.t. C, 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.*

```
     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, 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, replace_det(Fields,field(FN,_),field(FN,V),Fds'),
                                  set_cell(H,Ref,object(T,Fds'),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([(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 [10]

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 corresponds to the *path constraints* associated with 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, as long as there are incomplete branches in the finite tree.

Example 3. The next table shows the summary obtained by symbolically executing method `simplify` using the *block-2* coverage criterion of [1] (see Sec. 1):



A_{in}	A_{out}	$Heap_{in}$	$Heap_{out}$	EF	$Constraints$
r(B)		B → $\frac{C}{0}$	B → $\frac{G}{0}$	ok	C<0, J=-C, G=C/J
r(B)		B → $\frac{C}{0}$	B → $\frac{1}{0}$	ok	C>0
r(B)		B → $\frac{0}{0}$	B → $\frac{0}{0}$ 0 → AE	exc(0)	
r(B)		B → $\frac{C}{D}$	B → $\frac{H}{J}$	ok	D<0, K=-D, H=C/K, C *mod* D=0, J=D/K
r(B)		B → $\frac{C}{D}$	B → $\frac{H}{1}$	ok	D>0, C *mod* D=0, C/D=H

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, respectively. 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. 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 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.

```
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).

compose_hin(_,SH) :- var(SH), !.
compose_hin(_,[]).
compose_hin(H,[(R,Ce)|SH]) :-
    get_cell(H,R,Ce'), unify_cells(Ce',Ce),
    compose_hin(H,SH).

unify_cells(ob(T,Fs),ob(T,Fs')) :-
    unify_fields(Fs,Fs').
unify_cells(array(T,L,Vs),array(T,L,Vs)).

unify_fields(_,Fs) :- var(Fs),!.
unify_fields(_,[]).
unify_fields(Fs,[field(FName,FV)|RFs]) :-
    member_det(field(FName,FV'),Fs), !,
    FV = FV', unify_fields(Fs,RFs).
```

```
compose_hout(H,SH,H') :- var(SH),!.
compose_hout(H,[],H).
compose_hout(H,[(Ref,C)|SH],H") :-
    get_cell(H,Ref,C'), combine_cell(C',C,C"),
    set_cell(H,(Ref,C"),H'),
    compose_hout(H',SH,H").

combine_cell(ob(T,Fs),ob(T,Fs'),ob(T,Fs")) :-
    set_fields(Fs,Fs',Fs").
combine_cell(array(T,L,Vs),array(T,L,Vs'),
                  array(T,L,Vs")) :-
    set_array_vs(Vs,Vs',Vs").

set_fields(Fs,Fs',Fs) :- var(Fs'),!.
set_fields(Fs,[],Fs).
set_fields(Fs,[field(FN,FV)|RFs],Fs") :-
    replace_det(Fs,field(FN,_),field(FN,FV),Fs'),
    set_fields(Fs',RFs,Fs").

set_array_vs(Vs,Vs',Vs) :- var(Vs'),!.
set_array_vs([],[],[]).
set_array_vs([_|RVs],[V|RVs'],[V|RVs"]) :-
    set_array_vs(RVs,RVs',RVs").
```

Fig. 4. The composition operation

3 Towards 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. The challenge is now to define in the context of our CLP approach a composition operation so that, instead of symbolically executing p, its (previously computed) summary $\mathcal{S}_p$, can be reused in such a way that the symbolic execution of m is equivalent as if p was indeed symbolically executed within m.

Fig. 4 shows such a composition operation (predicate compose_summary/1). The idea is therefore to replace during symbolic execution a method invocation I by a call compose_summary(I) when there is a summary available for I. Intuitively, given the variables of the call to p, with their associated state ϕ, compose_summary/1 produces a branch (on backtracking) for each *compatible* case $c \in \mathcal{S}_p$, composing its state ϕ_c with ϕ and producing a new state ϕ' to continue the symbolic execution on. We assume that the summary for a method p is represented as a set of facts of the form summary(p,$\mathsf{SA_{in}}$,$\mathsf{SA_{out}}$,$\mathsf{SH_{in}}$,$\mathsf{SH_{out}}$,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 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. The heaps however require a more sophisticated treatment, mainly due to its underlying representation of sets (of objects and fields) as Prolog lists. Predicate compose_hin/2

A_{in}	A_{out}	$Heap_{in}$	$Heap_{out}$	EF	$Constraints$
[B,C,0]		H	H	ok	∅
[r(B),null,C]		B → $\boxed{\begin{array}{c}\text{L}\\ \hline [\text{V}\|\_]\end{array}}$	B → $\boxed{\begin{array}{c}\text{L}\\ \hline [\text{V}\|\_]\end{array}}$ 1 → NPE	exc(1)	C>0, L>0
[null,B,C]		H	2 → NPE	exc(2)	C>0
[B,C,D]		H	3 → AE	exc(3)	D<0
[r(B),r(C),1]		B → $\boxed{\begin{array}{c}\text{L1}\\ \hline [\text{V}\|\_]\end{array}}$ C → $\boxed{\begin{array}{c}\text{L2}\\ \hline [\text{V2}\|\_]\end{array}}$	B → $\boxed{\begin{array}{c}\text{L1}\\ \hline [\text{V}\|\_]\end{array}}$ C → $\boxed{\begin{array}{c}\text{L2}\\ \hline [\text{V}\|\_]\end{array}}$	ok	L1>1, L2>0

Table 1. Summary of method `arraycopy`

composes the input heap of the summary case $\mathsf{SH_{in}}$ with the current heap $\mathsf{H_{in}}$ producing the composed input heap in $\mathsf{H_{in}}$. To accomplish this, compose_hin/2 traverses each cell in $\mathsf{SH_{in}}$, and binds the appropriate variables in $\mathsf{H_{in}}$ with the appropriate sub-terms in the cell. Essentially, it produces an equivalent effect as if a sequence of get_field/4 operations for each field in each object in $\mathsf{SH_{in}}$ are performed over $\mathsf{H_{in}}$ (or get_array/4 for arrays). Similarly, the effect of compose_hout/3 is equivalent as if a sequence of set_field/5 operations for each field in each object in $\mathsf{SH_{out}}$ are performed (or set_array/5 for each array element), starting with $\mathsf{H_{in}}$, and gradually obtaining new heaps until getting the composed output heap $\mathsf{H_{out}}$.

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

3.2 Compositional TDG Schemata

In order to design a compositional, incremental approach to TDG, the composition operation can be incorporated in two main different ways within the testing process:

Context-sensitive. Usually, starting from an entry method m (and possibly a set of preconditions), TDG performs a top-down symbolic execution such that, when a method call p is found, its code its 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.

Context-insensitive. Another possibility is to perform the TDG process in a context-insensitive way. This can be done by first computing the call graph for the entry method m and the strongly connected components (SCC) for such graph. The SCCs can be traversed in reverse topological order starting from the SCC which does not depend on any other. The idea is that each SCC is symbolically executed from its entry w.r.t. the most general context (i.e., *true*). 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 advantage of the context-insensitive approach is that composition can always be performed. 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

A_{in}	A_{out}	$Heap_{in}$	$Heap_{out}$	EF	$Constraints$
r(B)	r(0)	$B \rightarrow \boxed{\begin{array}{c}0\\ \hline []\end{array}}$	$B \rightarrow \boxed{\begin{array}{c}0\\ \hline []\end{array}}\ 0 \rightarrow \boxed{\begin{array}{c}0\\ \hline []\end{array}}$	ok	∅
null	X	H	$6 \rightarrow$ NPE	exc(6)	∅
r(B)	r(0)	$B \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ D \rightarrow \left(\frac{F}{0}\right)$	$B \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ 0 \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ D \rightarrow \left(\frac{L}{0}\right)$	ok	F<0, N=-F, L=F/N
r(B)	r(0)	$B \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ D \rightarrow \left(\frac{F}{0}\right)$	$B \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ 0 \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ D \rightarrow \left(\frac{1}{0}\right)$	ok	F>0
r(B)	X	$B \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ D \rightarrow \left(\frac{0}{0}\right)$	$B \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ 0 \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ D \rightarrow \left(\frac{0}{0}\right)\ 1 \rightarrow$ AE	exc(1)	∅
r(B)	r(0)	$B \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ D \rightarrow \left(\frac{F}{G}\right)$	$B \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ 0 \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ D \rightarrow \left(\frac{M}{N}\right)$	ok	G<0, F *mod* G=0, P=-G, M=F/P, N=G/P
r(B)	r(0)	$B \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ D \rightarrow \left(\frac{F}{G}\right)$	$B \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ 0 \rightarrow \boxed{\begin{array}{c}1\\ \hline [r(D)]\end{array}}\ D \rightarrow \left(\frac{M}{1}\right)$	ok	G>0, F *mod* G=0, M=F/G
r(B)	X	$B \rightarrow \boxed{\begin{array}{c}1\\ \hline [null]\end{array}}$	$B \rightarrow \boxed{\begin{array}{c}1\\ \hline [null]\end{array}}\ 0 \rightarrow \boxed{\begin{array}{c}1\\ \hline [null]\end{array}}\ 2 \rightarrow$ NPE	exc(2)	∅

Table 2. Summary of method `simp`

required information is computed, but it can happen that there are several invocations to the same method which cannot reuse previous summaries (because the associated contexts were not sufficiently general). In such case, it had been more efficient to have obtained the summary without assuming any context. It remains as future work to formalize the above approaches (and possibly hybrid variants) and experimentally evaluate them.

3.3 Handling Native Code during Symbolic Execution

An inherent limitation of symbolic execution is the handling of native code. This is code which is implemented in another language. A solution is concolic execution [8] 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 which are obtained by applying our own symbolic execution mechanism. Let us see an example.

Example 5. Assume that method `arraycopy` is native. A (complete) specification of `arraycopy` can be provided by means of a corresponding method summary, as shown in Table 1, 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 2, 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 specifications 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 Completeness w.r.t. Coverage Criteria

When selecting a coverage criterion during TDG, it must be ensured that the test cases obtained in the summary ensure the coverage established by the criterion, i.e., *completeness* w.r.t. the criterion in guaranteed. In compositional TDG, the following question arises: *given an existing summary S for p w.r.t. the coverage criterion C, if while computing a summary for m w.r.t. C a call to p occurs and we compose S with the current state, is it ensured that the final summary obtained for m meets criterion C?* This does not hold for all coverage criteria, e.g., for statement coverage.

Example 6. Consider the following simple method:

```
m(int a,int b){if (a > 0 || b > 0) S;}
```

Without assuming any context, a summary with a single case $\{\texttt{a} > \texttt{0}, \texttt{b} \leq \texttt{0}\}$ is sufficient to achieve statement coverage. However, if `m` is called from an outer scope with a more restricted context (e.g., $\texttt{a} < \texttt{0}$), such summary is no longer valid to achieve statement coverage since statement `S` would not be visited.

Intuitively, completeness in compositional reasoning does not hold for a criterion when it does not meet the following property: *given a summary S obtained for m in a context ϕ w.r.t. C, test cases for a more restricted context ϕ' cannot be obtained by adapting S to context ϕ'.* On the contrary, when such property holds, the coverage criterion is compositional in the sense that completeness w.r.t. coverage is ensured. For instance, the block-k coverage criterion used in the examples of the paper is compositional. This is because, by restricting the context to ϕ', we are pruning the symbolic execution tree obtained for ϕ. Hence, the test cases for ϕ' can be obtained from the test cases in the summary obtained for ϕ (by adapting them to ϕ').

Another interesting issue to study is whether different (compositional) criteria can be combined during TDG. In other words, *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?* 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.

4 Conclusions and Future Work

Much effort has been been devoted in the area of symbolic execution to alleviate scalability problems 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 [15]). Another scaling technique which is closely related to abstraction is path merging [4,14], which consists in defining points where the merging of symbolic execution should occur. In this abstract, 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 outlined how the CLP-based approach to TDG of imperative languages can be applied in a compositional way. This is essential in order to make the approach scalable and, as we have seen, also provides a practical way of dealing with native code.

In a longer version of this abstract, we plan to extend our work along several directions. From the theoretical side, we plan to develop concrete TDG strategies which rely on the composition operator (along the lines discussed in Sec. 3.2). We also want to further study completeness issues by characterizing the features that a coverage criterion must have in order to enable (complete) compositional reasoning. On the practical side, we want to be able to generate JUnit test cases from our Prolog terms and, also, be able to translate specifications provided by means of assertions into our Prolog format. Finally, we will carry out an experimental evaluation in the PET system, a Partial Evaluation-based TDG tool [2].

References

1. E. Albert, M. Gómez-Zamalloa, and G. Puebla. Test Data Generation of Bytecode by CLP Partial Evaluation. In *18th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'08)*, number 5438 in LNCS, pages 4–23. Springer-Verlag, March 2009.
2. E. Albert, M. Gómez-Zamalloa, and G. Puebla. 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)*, pages 25–28, Madrid, Spain, January 2010. ACM Press.
3. Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 367–381. Springer, 2008.
4. Tamarah Arons, Elad Elster, Shlomit Ozer, Jonathan Shalev, and Eli Singerman. Efficient symbolic simulation of low level software. In *DATE*, pages 825–830. IEEE, 2008.
5. L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.
6. R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, 1996.
7. Patrice Godefroid. Compositional dynamic test generation. In Martin Hofmann and Matthias Felleisen, editors, *POPL*, pages 47–54. ACM, 2007.
8. Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.
9. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology*, 51:1409–1427, October 2009.
10. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming*, 2010. To appear.
11. A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Computational Logic*, pages 399–413, 2000.
12. N. Gupta, A. P. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Automated Software Engineering*, pages 219–228, 2000.
13. J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
14. Alfred Kölbl and Carl Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming*, 33(6):645–666, 2005.
15. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 98–112, Genova, Italy, April 2001. Springer-Verlag.
16. C. Meudec. Atgen: Automatic test data generation using constraint logic programming and symbolic execution. *Softw. Test., Verif. Reliab.*, 11(2):81–96, 2001.
17. R. A. Müller, C. Lembeck, and H. Kuchen. A symbolic java virtual machine for test case generation. In *IASTED Conf. on Software Engineering*, pages 365–371, 2004.
18. Corina S. Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, 2009.
19. T. Schrijvers, F. Degrave, and W. Vanhoof. Towards a framework for constraint-based test case generation. In *19th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'09)*, 2009.
20. Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.